Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Система керування файлами для спеціалізованого програмного середовища

08-54.БКР.005.00.000 ПЗ

Виконав: студент 4 курсу, групи КІ-23мсз
спеціальності 123 – Комп'ютерна інженерія
(шифр і назва напряму підготовки, спеціальності)
Людвик В. М.
(прізвище та ініціали)

Керівник: к. т. н., доц. кафедри ОТ
О. І. Черняк
(прізвище та ініціали)
«___» _____ 2025 р.

Рецензент: д. т. н., проф. зав.каф ПЗ
О. Н. Романюк
(прізвище та ініціали)
«___» _____ 2025 р.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
«23» 06 2025 р.

Вінниця ВНТУ — 2025 рік

Вінницький національний технічний університет

факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Освітньо-кваліфікаційний рівень бакалавр

Галузь знань — 12 «Інформаційні технології»

Спеціальність — 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
«21» березня 2025 року

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Людвику Володимиру Миколайовичу

1 Тема роботи: "Система керування файлами для спеціалізованого програмного середовища", керівник роботи Черняк О.І. к.т.н., доцент кафедри ОТ, затверджено наказом вищого навчального закладу від «20» березня 2025 року №97.

2 Термін подання студентом роботи 13.05.2025

3 Вихідні дані до роботи — актуальність, аналіз аналогів, розробка додатка, структурна схема, алгоритм роботи, тестування.

4 Зміст пояснювальної записки — Вступ. Аналіз існуючих технологій та підходів до створення систем керування файлами. Розробка програмного модуля створення систем керування файлами. Програмна реалізація модуля створення систем керування файлами та тестування розробленого програмного засобу. Висновки. Перелік джерел посилання. Додатки.

5 Перелік графічного матеріалу — UML-діаграма класів програми FileManagerApp, блок-схема алгоритму роботи програмного засобу, зображення головної сторінки додатка.

6 Консультанти розділів роботи представлені у таблиці 1.

Таблиця 1 — Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	к.т.н., доц. каф. ОТ Черняк О. І.	21.03.25	13.05
Нормоконтроль	ас.. каф. ОТ Швець С. І.	14.05.25	30.05

7 Дата видачі завдання 21.03.2025 р.

8 Календарний план виконання БКР наведений у таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання	Підп
1	Вибір, узгодження та затвердження теми БКР	21.03.25	вик
2	Аналіз задачі дослідження	22.03.25— 24.03.25	вик
3	Огляд існуючих систем управління задачами	25.03.25— 27.03.25	вик
4	Вибір оптимальних технологій і компонентів для реалізації	28.03.25— 30.03.25	вик
5	Розробка структури бази даних	31.03.25— 02.04.25	вик
6	Розробка архітектури та інтерфейсу	03.04.25— 07.05.25	вик
7	Реалізація функціоналу, тестування	08.05.25— 12.04.2025	вик
8	Попередній захист	13.05.25	вик

Студент Людвик В.М.

Керівник роботи Черняк О.І.

АНОТАЦІЯ

УДК 004.451

Людвик В.М. Система керування файлами для спеціалізованого програмного середовища. Бакалаврська дипломна робота зі спеціальності 123 — Комп'ютерна інженерія, освітня програма — Комп'ютерна інженерія. Вінниця: ВНТУ 2025. 120 с.

На укр. мові. Бібліогр.: рис.: 12; пос.: 15

У бакалаврській дипломній роботі розроблено програмне забезпечення для керування файлами в спеціалізованих програмних середовищах мовою C# на платформі .NET Framework з використанням Windows Forms. Реалізовано модулі для роботи з файловою системою, логування дій користувача, збереження налаштувань і фільтрації файлів. Новизна полягає в створенні адаптивної системи з модульною архітектурою, що забезпечує стабільну роботу з великими обсягами даних, інтуїтивний інтерфейс і безпечне оброблення винятків. Система протестована на різних сценаріях, включаючи екстремальні умови, та готова до інтеграції в інформаційні комплекси.

Ключові слова: файловий менеджер, C#, .NET Framework, Windows Forms, логування, модульна архітектура, спеціалізоване середовище.

ABSTRACT

UDC 004.451

Lyudvyk V.M. File Management System for a Specialized Software Environment. Bachelor's thesis in the specialty 123 — Computer Engineering, educational program — Computer Engineering.: Vinnytsya VNTU, 2025. 120 p.

In Ukrainian language. Bibliography: Fig.: 15; Ref.: 30

The bachelor's thesis develops software for file management in specialized software environments using C# on the .NET Framework platform with Windows Forms. Modules for file system operations, user action logging, configuration storage, and file filtering have been implemented. The novelty lies in creating an adaptive system with a modular architecture that ensures stable operation with large data volumes, an intuitive interface, and secure exception handling. The system has been tested across various scenarios, including extreme conditions, and is ready for integration into information systems.

Keywords: file manager, C#, .NET Framework, Windows Forms, logging, modular architecture, specialized environment.

ЗМІСТ

ВСТУП

Актуальність теми полягає в тому що у сучасному світі інформаційні технології відіграють ключову роль у всіх галузях людської діяльності — від виробництва до освіти, від медицини до державного управління. Важливою складовою цих технологій є ефективна організація, зберігання та обробка даних. У цьому контексті особливу увагу заслуговує процес керування файлами, які містять як службову, так і користувацьку інформацію. Питання ефективної роботи з файловою системою стає критичним у спеціалізованих програмних середовищах, де помилки в обробці даних можуть мати значні наслідки. Більшість стандартних рішень, які пропонують сучасні операційні системи або універсальні файлові менеджери, не враховують особливостей специфічних середовищ. Це може стосуватися як вимог до безпеки даних, так і потреби в інтеграції з іншими модулями ПЗ.

Мета дослідження полягає в розробці програмного забезпечення для системи керування файлами у спеціалізованому програмному середовищі, що забезпечує зручну взаємодію користувача з файловою структурою, дозволяє виконувати основні операції над файлами, підтримує ведення журналу дій і забезпечує високу надійність та стабільність при експлуатації.

Об'єктом дослідження виступає процес організації файлового простору та керування файлами в умовах спеціалізованого середовища.

Предметом дослідження є програмні засоби, принципи організації інтерфейсу користувача та архітектурні підходи до створення застосунку, що забезпечує зручне та безпечне керування файлами.

Методологія У процесі розробки застосовано комплексний підхід, який включає методи системного аналізу для визначення вимог до системи, об’єктно-орієнтованого проєктування для створення архітектури програми, а також принципи програмної інженерії для реалізації коду. Для забезпечення ергономіки інтерфейсу використано компоненти Windows Forms, із налаштуванням їхніх властивостей для зручного відображення даних.

Новизна полягає в адаптації існуючих принципів керування файлами до вимог специфічних потреб спеціалізованих середовищ через створення легкої та гнучкої системи. Особливістю є інтеграція асинхронного завантаження каталогів із механізмом скасування операцій, що підвищує ефективність роботи з великими обсягами даних. Також новизною є реалізація модуля пагінації, фільтрації файлів та вбудоване логування.

Практичне значення — розроблена система має практичне значення завдяки своїй здатності бути інтегрованою в існуючі інформаційні системи або використовуватися як окремий застосунок у спеціалізованих середовищах, що вимагають підвищеної надійності та структурованості обробки файлів. Програма може застосовуватися в навчальних, наукових, медичних чи промислових цілях, де важлива стабільна робота з файлами та відстеження дій користувача. Гнучкість архітектури, зокрема підтримка асинхронних операцій і пагінації, дозволяє адаптувати систему до потреб конкретної організації. Модуль логування, забезпечує аудит операцій, що є важливим для систем із високими вимогами до безпеки та відповідності нормативним стандартам. Крім того, запропоноване рішення може служити основою для створення розширених систем управління даними, включаючи хмарні чи розподілені файлові системи.

Апробація. Матеріали дослідження доповідались на Міжнародній науково-практичній інтернет-конференції “Молодь в науці: дослідження, проблеми, перспективи” (МН-2025). За матерілами доповіді опубліковано тези. Доступ за посиланням: https://conferences.vntu.edu.ua/index.php/mn/mn2025/paper/view/25211.

1 ТЕОРЕТИЧНІ ОСНОВИ СИСТЕМ КЕРУВАННЯ ФАЙЛАМИ

1.1. Основні поняття та принципи побудови файлових систем

Файлова система є ключовим компонентом операційної системи, який визначає спосіб зберігання, ідентифікації та організації даних на фізичних носіях, створюючи логічну модель для представлення файлів, папок і метаданих, що включають структуру, права доступу, час створення та інші атрибути. Без належної файлової системи жодна операційна система не змогла б ефективно взаємодіяти з даними на дисках. Вона виконує низку важливих функцій, таких як ієрархічне впорядкування інформації через створення дерева папок, управління простором із розподілом, резервуванням та видаленням, підтримка метаданих, що відображають розмір, дату зміни чи права доступу, захист від несанкціонованого доступу через встановлення прав, а також забезпечення відновлення даних у разі помилок або збоїв.

Найпоширенішими файловими системами є:

— FAT32 — старий формат, сумісний із багатьма ОС, але має обмеження на розмір файлів та об’єм дисків;

— NTFS — стандартна файлова система Windows, яка підтримує розширені атрибути, права доступу, журналювання та шифрування;

— ext3/ext4 — поширені в Linux-системах, мають високу продуктивність і стійкість до помилок;

— exFAT — оптимізована для флеш-пристроїв, забезпечує велику швидкість запису/читання.

Розглянемо основні елементи файлової системи:

Основними елементами файлової системи є файл як найменша логічна одиниця зберігання даних або програмного коду, каталог як структура з посиланнями на файли чи інші каталоги, індексний дескриптор (inode) у Unix-подібних системах для метаінформації про файл та блок даних як одиниця фізичного зберігання на диску. Ці елементи формують основу для організації даних у будь-якій файловій системі, забезпечуючи логічну та фізичну структуру для

зберігання інформації. Файли та каталоги створюють ієрархію, яка полегшує користувачам доступ до даних, тоді як inode та блоки даних оптимізують фізичне розташування інформації на носії, що є важливим для забезпечення швидкості операцій читання та запису. У сучасних файлових системах, таких як NTFS, FAT32 чи ext4, ці компоненти адаптовані для різних типів пристроїв і завдань, що дозволяє використовувати їх у різноманітних програмних середовищах, від персональних комп'ютерів до серверних систем.

Від правильного вибору та використання файлової системи залежить швидкість доступу до даних, стабільність ПЗ, стійкість до втрати інформації при аваріях, можливість масштабування та адаптація до конкретного типу завдань. Наприклад, файлова система впливає на продуктивність програм, що працюють із великими обсягами даних, таких як бази даних чи медіаархіви, а також на здатність системи відновлюватися після збоїв завдяки механізмам журналювання чи резервного копіювання. Вибір файлової системи також визначає її сумісність із апаратним забезпеченням і операційними системами, що є критично важливим для спеціалізованих середовищ, де потрібна висока надійність і гнучкість.

1.2 Особливості роботи з файлами у спеціалізованих програмних середовищах

Спеціалізовані програмні середовища, такі як інженерні комплекси, медичні системи, дослідницькі платформи чи системи автоматизованого керування процесами, висувають специфічні вимоги до операцій із файлами. Такі системи часто працюють із унікальними форматами даних, наприклад, CAD-файлами, вимірювальними даними чи медичними знімками, що потребує їхньої належної обробки та збереження. Ієрархічна структура даних у цих середовищах зазвичай включає багаторівневі рівні, такі як проєкт, модуль, дані та версія, що вимагає чіткої організації. Версійність є критично важливою, особливо в наукових дослідженнях чи технічному дизайні, де необхідно зберігати історію змін. Безпека відіграє важливу роль, оскільки доступ до окремих файлів чи груп файлів обмежується залежно від ролі користувача, а інтеграція між підсистемами

(візуалізація, обробка, архівування) має бути автоматизованою та без втрат метаінформації. Прикладом може слугувати система SCADA, яка реєструє показники температури, тиску чи електричних сигналів у реальному часі. Таким чином, розробка спеціалізованих систем керування файлами має враховувати вбудовані механізми перевірки доступу, суворе логування всіх дій користувача, модулі контролю цілісності даних та можливість взаємодії з іншими програмними інструментами без втрати метаданих, що забезпечує ефективність і надійність у специфічних умовах.

Отже, розробка спеціалізованої системи керування файлами повинна враховувати:

— вбудовані механізми перевірки доступу;

— суворе логування усіх дій користувача;

— модулі контролю цілісності даних;

— можливість взаємодії з іншии програмними інструментами без втрати метаданих, що забезпечує ефективність і надійність у специфічних умовах.

У спеціалізованих середовищах важливим є також забезпечення високої продуктивності при роботі з великими обсягами даних, що може включати файли розміром у десятки чи сотні гігабайт, наприклад, у моделюванні фізичних процесів чи обробці медичних зображень. Це вимагає використання ефективних алгоритмів читання/запису, індексації файлів для швидкого пошуку та кешування для зменшення затримок. Крім того, системи повинні підтримувати асинхронну обробку операцій, щоб уникнути блокування інтерфейсу під час тривалих задач, таких як копіювання великих файлів чи сканування каталогів. У деяких випадках, наприклад, у промислових системах, необхідна підтримка реального часу для синхронізації даних із датчиків чи виконавчих механізмів, що додає складності до управління файлами.

Безпека даних у спеціалізованих середовищах виходить за рамки простого обмеження доступу. Наприклад, у медичних системах необхідно забезпечити відповідність стандартам, таким як HIPAA, що вимагає шифрування даних як у спокої, так і під час передачі, а також ведення детальних журналів доступу. У

дослідницьких платформах, де дані можуть бути інтелектуальною власністю, важливим є використання цифрових підписів чи геш-функцій (наприклад, SHA-256) для перевірки цілісності файлів. У промислових комплексах, де файли можуть бути пов’язані з критичними процесами, системи керування файлами повинні включати механізми резервного копіювання та відновлення даних після збоїв, щоб мінімізувати ризики зупинки виробництва.

1.3 Класифікація систем керування файлами та огляд існуючих рішень

Файлові менеджери класифікуються за кількома критеріями, зокрема типом інтерфейсу, способом розповсюдження та призначенням. За типом інтерфейсу вони поділяються на текстові, як-от FAR Manager, що є ефективними для роботи з клавіатурою та мають низьке навантаження на систему, і графічні, такі як Windows Explorer чи Double Commander, орієнтовані на зручність для пересічного користувача. За способом розповсюдження розрізняють програмні оболонки, які потребують інсталяції та часто розширюють функціонал через плагіни, і вбудовані системи, що є частиною операційної системи, як-от Explorer у Windows. За призначенням файлові менеджери бувають загального призначення, здатні працювати з будь-якими файлами, та спеціалізовані, призначені для конкретного середовища, наприклад, файловий менеджер у AutoCAD.

Серед популярних прикладів програм можна назвати Total Commander, потужний двопанельний менеджер із великою кількістю додатків і плагінів, FreeCommander, безкоштовну альтернативу з підтримкою вкладок, порівняння каталогів та FTP, Q-Dir, чотирипанельний менеджер із гнучкою навігацією, а також комерційні Xplorer2 і Directory Opus, які дозволяють створювати макроси, сценарії та керувати метаданими. Проте ці інструменти, попри розвинений функціонал, не вирішують завдань інтеграції з модулями середовища, детального логування дій, формування звітності про операції чи підтримки специфічних форматів даних, що обмежує їхнє застосування у спеціалізованих умовах.

1.4 Проблематика використання універсальних рішень у специфічних умовах

Універсальні файлові менеджери, хоч і є гнучкими, не враховують вимог конкретних галузей, що створює низку проблем. Вони не передбачають спеціальних сценаріїв перевірки цілісності даних, не забезпечують контроль відповідності структури каталогів технічним чи юридичним нормам, а їхні системи безпеки реалізуються на рівні операційної системи, а не на рівні застосунку. Крім того, у таких менеджерах відсутня можливість зберігати прив’язку файлів до контексту, наприклад, до проєкту, замовлення чи етапу роботи, що є критично важливим для спеціалізованих середовищ. Це підкреслює необхідність розробки адаптованих рішень, які б відповідали специфічним потребам і забезпечували вищу ступінь контролю та інтеграції. Наприклад, у навчальних чи промислових середовищах універсальні менеджери не дозволяють ефективно організувати файли за навчальними курсами чи виробничими процесами, а також не підтримують розширене логування дій користувача для аудиту, що може бути вимогою нормативних стандартів.

Універсальні рішення, такі як стандартні файлові менеджери операційних систем чи популярні комерційні програми, часто розроблені для широкого загалу, що робить їх універсальними, але обмеженими в специфічних умовах. Наприклад, у медичній сфері, де потрібна сувора відповідність стандартам HIPAA чи GDPR, універсальні менеджери не забезпечують належного шифрування даних чи автоматичного створення резервних копій із прив’язкою до історії змін. У наукових установах, де управління даними досліджень потребує категоризації файлів за етапами експериментів чи типами обладнання, такі програми не пропонують інструментів для створення метаданих чи інтеграції з лабораторними інформаційними системами. У промислових підприємствах, де файли можуть бути пов’язані з технологічними процесами, універсальні менеджери не дозволяють створювати складні ієрархії чи автоматизувати управління документообігом, що ускладнює контроль за версіями документів чи їх доступністю для різних груп користувачів. Відсутність підтримки асинхронної обробки великих обсягів даних

чи інтеграції з розподіленими системами, такими як мережеві сховища чи хмарні платформи, також обмежує їх ефективність у великих організаціях.

Крім того, універсальні файлові менеджери рідко надають можливості для кастомізації інтерфейсу під конкретні потреби користувачів, що може знижувати продуктивність роботи в умовах, де швидкий доступ до специфічних функцій є критично важливим. Наприклад, у фінансових установах, де обробка документів потребує швидкого пошуку за датами, номерами контрактів чи іншими параметрами, стандартні менеджери не дозволяють налаштувати розширені фільтри чи автоматизовані звіти.

Таким чином, обмеження універсальних файлових менеджерів підкреслюють важливість створення спеціалізованих систем, які забезпечують не лише базові операції з файлами, але й враховують унікальні вимоги конкретних галузей, підвищуючи ефективність, безпеку та контроль над даними.

1.5 Сучасні тенденції в розвитку файлових систем

Сучасні інформаційні технології зумовлюють нові вимоги до файлових систем, що відображається у ключових тенденціях, таких як хмарні файлові системи, що адаптуються до розподілених середовищ із підтримкою синхронізації даних, версійності та доступу через API для інтеграції з сервісами на кшталт Google Drive, Dropbox чи OneDrive. Розподілені файлові системи, як-от Hadoop HDFS чи Ceph, призначені для обробки великих обсягів даних у кластерних середовищах, забезпечуючи високу відмовостійкість і масштабування. Підтримка великих даних стає актуальною у спеціалізованих галузях, таких як наукові дослідження чи медицина, де файлові системи повинні обробляти файли розміром у десятки гігабайт, зберігаючи швидкість доступу та цілісність. Шифрування та безпека відіграють важливу роль через зростання кіберзагроз, що призводить до вбудованих механізмів шифрування, як-от BitLocker для NTFS, а також на рівні застосунків. Кросплатформність є ще однією тенденцією, оскільки системи дедалі частіше потребують сумісності з Windows, Linux чи macOS, ускладнюючи створення універсальних рішень.

Ці напрямки вказують на необхідність розробки файлових менеджерів, які поєднують гнучкість із можливостями адаптації до специфічних вимог, включаючи підтримку хмарних сервісів, обробку великих даних і забезпечення безпеки.

Хмарні файлові системи, такі як Amazon S3 чи Microsoft Azure Blob Storage, забезпечують не лише глобальний доступ до даних, але й дозволяють автоматизувати резервне копіювання, управління версіями та обробку великих масивів інформації в реальному часі, що є важливим для бізнес-процесів у фінансових установах чи телекомунікаційних компаніях. Розподілені файлові системи, такі як GlusterFS чи Lustre, оптимізують роботу з великими даними в наукових обчисленнях, де швидкість паралельного доступу до файлів із різних вузлів кластера визначає ефективність досліджень. У медичній сфері підтримка великих даних забезпечує швидкий доступ до зображень високої роздільної здатності, таких як МРТ чи КТ, із можливістю інтеграції з інформаційними системами лікарень для автоматичного аналізу та архівування.

Енергоефективність також стає важливим аспектом, оскільки файлові системи для великих дата-центрів потребують оптимізації споживання ресурсів для зниження витрат на електроенергію та охолодження. Усе це підкреслює необхідність створення файлових менеджерів, які не лише виконують базові операції з файлами, але й інтегруються з передовими технологіями, забезпечуючи гнучкість, безпеку та ефективність у різноманітних галузях, від освіти до промислового виробництва, де унікальні вимоги до управління даними стають визначальними для успіху інформаційних систем.

1.6 Архітектурні підходи до розробки систем керування файлами

Розробка спеціалізованої системи керування файлами потребує ретельного вибору архітектури, яка забезпечить модульність, розширюваність і надійність.

Серед основних підходів можна виділити монолітну архітектуру, де всі компоненти, такі як інтерфейс, логіка та доступ до файлової системи, об’єднані в єдину систему, що забезпечує простоту реалізації та швидкість розробки, але ускладнює масштабування та модифікацію. Модульна архітектура передбачає поділ системи на незалежні модулі, наприклад, модуль логування, доступу до файлів та інтерфейсу, які взаємодіють через чіткі інтерфейси, що дає гнучкість і полегшує оновлення. Мікросервісна архітектура реалізує кожен функціональний компонент як окремий сервіс із комунікацією через API, що підходить для розподілених систем, але ускладнює управління через необхідність координації сервісів. Шарова архітектура поділяє систему на шари — представлення, бізнес-логіка та доступ до даних, спрощуючи тестування та підтримку.

Для спеціалізованих середовищ найчастіше обирають модульну або шарувату архітектуру, оскільки вони дозволяють легко адаптувати систему до нових вимог, наприклад, додавання підтримки нових форматів файлів чи інтеграції з зовнішніми сервісам

1.7 Роль інтерфейсу користувача в системах керування файлами

Інтерфейс користувача (UI) відіграє ключову роль у забезпеченні зручності роботи з файловою системою, відіграючи вирішальну роль у сприйнятті системи.

Основні принципи дизайну UI для файлових менеджерів включають інтуїтивність, що дозволяє користувачу швидко освоїти базові операції, такі як копіювання, переміщення чи видалення, гнучкість із можливістю налаштування інтерфейсу, наприклад, зміною вигляду панелей чи додаванням ярликів, ефективність через швидкий доступ до часто використовуваних функцій за допомогою гарячих клавіш чи контекстних меню та інформативність через відображення метаданих файлів, таких як розмір, дата зміни чи тип, у зрозумілій формі. У спеціалізованих середовищах інтерфейс має враховувати контекст

роботи, наприклад, у медичних системах доцільно передбачати попередній перегляд зображень, як-от DICOM-файли, а в інженерних системах — ієрархію проєктів.

Важливим є також забезпечення підтримки локалізації інтерфейсу для роботи в різних мовних середовищах, що підвищує доступність для широкого кола користувачів. Сучасні UI для файлових менеджерів часто включають адаптивність до різних пристроїв і роздільної здатності екранів, що забезпечує комфортну роботу як на настільних комп’ютерах, так і на мобільних платформах. Принципи дизайну UI базуються на ергономіці та психології сприйняття, що передбачає використання чіткої візуальної ієрархії, контрастних кольорів і логічного розташування елементів управління. Наприклад, у корпоративних системах доцільно передбачати підтримку автоматизації рутинних операцій через скрипти чи макроси, доступні через інтерфейс, тоді як у навчальних середовищах акцент робиться на простоту освоєння для користувачів-початківців. Забезпечення зворотного зв’язку, наприклад, через сповіщення про завершення операцій чи помилки, також підвищує довіру до системи та її ефективність.

Таким чином, UI є не лише технічним компонентом, але й фактором, що впливає на загальне сприйняття та успішність файлового менеджера в різних галузях.

Ефективний інтерфейс користувача в системах керування файлами не лише полегшує виконання операцій, але й сприяє підвищенню продуктивності користувачів, знижуючи когнітивне навантаження та час, необхідний для виконання завдань. У спеціалізованих середовищах, таких як наукові чи промислові системи, UI має бути адаптованим до унікальних робочих процесів, забезпечуючи швидкий доступ до специфічних функцій, наприклад, порівняння версій файлів у проєктних системах чи автоматичного сортування даних у дослідницьких платформах. Сучасні тенденції в дизайні UI також включають підтримку темної теми для зниження втоми очей при тривалій роботі, а також використання анімацій чи мікровзаємодій для покращення користувацького досвіду, наприклад, відображення прогресу копіювання файлів чи плавного

розгортання деревоподібної структури каталогів. Інтеграція з іншими програмними інструментами через UI, наприклад, можливість перетягування файлів у зовнішні редактори чи відображення контекстної інформації у спливаючих підказках, підвищує зручність і ефективність. У перспективі розвиток UI для файлових менеджерів може включати використання штучного інтелекту для персоналізації інтерфейсу, автоматично пропонуючи найчастіше використовувані функції чи оптимізуючи розташування елементів залежно від поведінки користувача. Крім того, забезпечення доступності для людей із обмеженими можливостями, наприклад, через підтримку висококонтрастних режимів чи керування з клавіатури, робить систему більш інклюзивною.

Таким чином, UI не лише формує перше враження від системи, але й визначає її довгострокову ефективність і здатність адаптуватися до різноманітних потреб користувачів у спеціалізованих середовищах, де точність, швидкість і зручність є ключовими факторами успіху.

1.8 Проблеми масштабування та оптимізації файлових систем

У спеціалізованих середовищах часто виникають проблеми, пов’язані з обробкою великих обсягів даних або одночасною роботою багатьох користувачів. Основні виклики включають:

— Продуктивність: Великі файли або складні ієрархії каталогів можуть уповільнювати операції читання/запису.

— Масштабування: Збільшення кількості файлів або користувачів вимагає оптимізації доступу до даних.

— Синхронізація: У розподілених системах виникають проблеми з узгодженням змін файлів між різними вузлами.

— Відмовостійкість: Збої апаратного забезпечення або програмні помилки можуть призвести до втрати даних.

Великі файли або складні ієрархії каталогів можуть уповільнювати операції читання та запису, тоді як збільшення кількості файлів чи користувачів вимагає

оптимізації доступу до даних. У розподілених системах виникають труднощі з узгодженням змін файлів між різними вузлами, а збої апаратного забезпечення чи програмні помилки можуть призвести до втрати даних.

Для подолання цих проблем застосовуються методи індексації файлів для прискорення пошуку, кешування даних для підвищення продуктивності, асинхронна обробка операцій для забезпечення плавності роботи та використання розподілених баз даних для зберігання метаданих, що підвищує загальну ефективність і стабільність системи.

Сучасні файлові системи, такі як NTFS, ext4 або ZFS, використовують ці методи для забезпечення високої продуктивності та надійності. Наприклад, ZFS підтримує механізми дедуплікації та компресії для економії дискового простору, тоді як розподілені файлові системи, такі як Hadoop HDFS, дозволяють масштабувати зберігання даних на кластерах із сотень вузлів. Крім того, впровадження технологій RAID і резервного копіювання підвищує відмовостійкість, а використання хмарних сховищ, таких як Amazon S3, забезпечує гнучкість і доступність даних у глобальному масштабі. Ці підходи дозволяють файловим системам адаптуватися до зростаючих потреб сучасних інформаційних систем, забезпечуючи швидкий доступ до даних, стабільність і захист від втрат інформації.

У сучасному світі, де обсяги даних зростають експоненціально, проблеми масштабування стають дедалі актуальнішими для організацій, що працюють із великими масивами інформації, таких як телекомунікаційні компанії, фінансові установи чи наукові центри. Наприклад, у телекомунікаціях файлові системи повинні обробляти мільйони логів у реальному часі, забезпечуючи швидкий доступ до них для аналізу чи виявлення аномалій. У фінансових установах, де документообіг включає тисячі транзакційних файлів, важлива не лише швидкість, але й синхронізація між віддаленими офісами, що потребує складних механізмів узгодження даних.

У наукових дослідженнях, таких як геноміка чи астрофізика, файлові системи стикаються з необхідністю зберігання петабайт даних із паралельним доступом для

обчислень, що вимагає використання розподілених систем, таких як Lustre чи GlusterFS, які оптимізують пропускну здатність і мінімізують затримки. Синхронізація в розподілених середовищах ускладнюється не лише географічною розподіленістю вузлів, але й різними часовими зонами чи пропускною здатністю мережі, що потребує впровадження технологій, таких як CRDT (Conflict-free Replicated Data Types) для безконфліктного оновлення даних.

Відмовостійкість залишається критичною для критичних інфраструктур, таких як енергетичні чи медичні системи, де втрата даних може мати катастрофічні наслідки. Сучасні файлові системи, такі як Btrfs, пропонують інструменти для автоматичного відновлення даних після збоїв, тоді як хмарні рішення, такі як Google Cloud Storage, забезпечують географічно розподілене резервне копіювання для підвищення надійності. Енергоефективність також набуває значення, оскільки великі дата-центри прагнуть зменшити споживання електроенергії, що змушує розробників файлових систем оптимізувати алгоритми кешування та доступу до даних. Крім того, інтеграція штучного інтелекту в управління файловими системами відкриває нові можливості, такі як автоматична категоризація файлів, прогнозування збоїв дисків чи оптимізація розміщення даних для підвищення швидкості доступу

Усе це вказує на необхідність створення файлових систем, які не лише вирішують поточні проблеми масштабування та оптимізації, але й передбачають майбутні виклики, пов’язані з експоненціальним зростанням даних, глобалізацією інформаційних систем і підвищенням вимог до безпеки та енергоефективності в різноманітних галузях, від промисловості до наукових досліджень.

1.9 Забезпечення безпеки в системах керування файлами

Забезпечення безпеки в системах керування файлами є критично важливим аспектом їх розробки, особливо в умовах зростання кіберзагроз і посилення вимог до захисту даних у різних галузях. Сучасні інформаційні системи, що працюють із файлами, стикаються з необхідністю гарантувати конфіденційність, цілісність і доступність даних, а також відповідність міжнародним і національним стандартам безпеки, таким як GDPR, HIPAA чи ISO 27001. Основні аспекти безпеки включають захист від несанкціонованого доступу, запобігання втраті чи пошкодженню даних і забезпечення можливості їх відновлення після збоїв.

Для забезпечення безпеки в системах керування файлами застосовуються різноманітні механізми. Шифрування даних, як на рівні файлової системи (наприклад, BitLocker для NTFS чи LUKS для Linux), так і на рівні застосунків, захищає файли від перехоплення чи несанкціонованого використання. Контроль доступу на основі ролей (RBAC) дозволяє обмежувати права користувачів до певних файлів чи каталогів, що є важливим у спеціалізованих середовищах, таких як медичні чи фінансові системи, де доступ до даних має бути строго регламентованим. Аутентифікація користувачів, включаючи двофакторну аутентифікацію, підвищує захист від несанкціонованого входу в систему. Журналювання дій (логування) забезпечує аудит операцій із файлами, дозволяючи відстежувати, хто, коли і які зміни вносив, що є необхідним для відповідності нормативним вимогам і розслідування інцидентів безпеки.

Особливу увагу приділено захисту від кібератак, таких як ransomware, які можуть заблокувати доступ до файлів або знищити їх. Для цього файлові системи використовують механізми резервного копіювання та створення знімків (snapshots), як у ZFS чи Btrfs, що дозволяють відновити дані до попереднього стану. Антивірусна інтеграція та виявлення аномалій у доступі до файлів, підкріплені алгоритмами машинного навчання, допомагають своєчасно реагувати на загрози. У розподілених системах, таких як хмарні сховища (Amazon S3, Google Cloud Storage), безпека забезпечується через шифрування даних у транзиті та спокої, а також через географічно розподілене зберігання для підвищення відмовостійкості.

Енергоефективність і безпека також пов’язані, оскільки великі дата-центри, що зберігають файли, потребують захисту від атак, які можуть вивести з ладу системи охолодження чи живлення. Оптимізація алгоритмів шифрування та доступу до даних зменшує енергоспоживання, зберігаючи високий рівень захисту.

Таким чином, забезпечення безпеки в системах керування файлами вимагає комплексного підходу, що поєднує шифрування, контроль доступу, журналювання, захист від атак, перевірку цілісності та сумісність із різними платформами. Ці аспекти є визначальними для створення надійних рішень, здатних відповідати вимогам сучасних інформаційних систем у спеціалізованих галузях, таких як освіта, медицина, промисловість чи наукові дослідження.

2 АНАЛІЗ АНАЛОГІВ ТА ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ

2.1. Глибокий аналіз сучасних рішень у сфері файлового керування

У сучасному цифровому світі обробка даних є невід’ємною частиною більшості інформаційних систем, де управління файлами, їхнє зберігання, впорядкування та захист мають критичне значення як для кінцевого користувача, так і для підприємств, дослідницьких організацій та державних структур. Незважаючи на різноманіття наявних рішень, жодне з них не є універсальним і повністю не задовольняє вимоги спеціалізованих середовищ, що змусило звернути увагу на створення власного програмного продукту. Проведено порівняльний аналіз аналогів, зокрема універсальних файлових менеджерів, таких як Total Commander і FreeCommander, а також корпоративних систем документообігу, що дозволило виявити їхні обмеження.

Універсальні файлові менеджери, такі як Windows File Explorer, пропонують базову функціональність для перегляду, перейменування, копіювання та переміщення файлів із перевагою інтеграції з операційною системою, однак мають значні недоліки, зокрема відсутність розширень, журналювання, а також обмежені можливості пошуку та фільтрації. Total Commander, як багатофункціональний двопанельний менеджер, підтримує FTP, архіви та плагіни, але не адаптований до роботи з контекстно залежними файлами й не забезпечує повноцінної інтеграції з іншими модулями спеціалізованого середовища. FreeCommander, Double Commander та FAR Manager є альтернативами для досвідчених користувачів із підтримкою вкладок і порівняння каталогів, проте вони потребують значної адаптації для специфічних галузей і не пропонують асинхронного завантаження чи логування, як реалізовано у вашій програмі. Хмарні рішення, такі як Google Drive, OneDrive та Dropbox, забезпечують віддалений доступ і зручний обмін, але вимагають постійного підключення до Інтернету та не гарантують контролю над внутрішньою структурою зберігання й метаінформацією, що є важливим для спеціалізованих застосувань. Крім того, ці хмарні сервіси можуть створювати проблеми з конфіденційністю даних через їх зберігання на віддалених серверах, що

обмежує їх використання в середовищах із високими вимогами до безпеки, наприклад, у медичних чи промислових системах.

Корпоративні системи документообігу, як-от Alfresco, M-Files та OpenText, надають потужні інструменти для управління документами з підтримкою складної ієрархії, контролю версій і роботи з ролями, однак вони є надмірно складними для налаштування, потребують серверного оточення, баз даних та навчання персоналу, що робить їх недоцільними для невеликих команд чи вузькоспеціалізованих середовищ. Проведений аналіз підтвердив, що більшість аналогів є або надмірно складними для впровадження, або не забезпечують достатньої адаптивності до специфічних потреб, таких як асинхронна обробка великих каталогів, пагінація чи вбудоване логування, що обґрунтувало доцільність створення окремого рішення.

2.2 Вимоги до архітектури системи у контексті спеціалізованого середовища

Для створення ефективного та зручного файлового менеджера у спеціалізованому середовищі враховано як функціональні, так і нефункціональні вимоги. До функціональних вимог належать виконання основних операцій із файлами (створення, перегляд, копіювання, видалення, перейменування), зручний перегляд структури каталогів у вигляді дерева, підтримка фільтрації за розширенням, розміром чи датою останньої зміни, інтеграція з специфічними файлами чи підсистемами, а також ведення журналу дій користувача із фіксацією часу, дій і результатів. Нефункціональні вимоги включають стабільність і надійність при роботі з великим обсягом файлів, мінімальну потребу у зовнішніх залежностях для забезпечення автономності, зрозумілий інтуїтивний інтерфейс, який не потребує спеціального навчання, кроссверсійність для підтримки різних версій Windows, а також модульність для подальшого розширення функціоналу. Ці вимоги сформували основу для вибору модульної архітектури з чітким розподілом відповідальності, що полегшує підтримку, масштабування та супровід продукту, зокрема для асинхронної обробки та логування, реалізованих у нашій програмі.

Функціональні вимоги забезпечують базову взаємодію користувача з файловою системою, дозволяючи ефективно виконувати повсякденні завдання в

спеціалізованих середовищах, таких як навчальні, промислові чи наукові комплекси. Наприклад, підтримка фільтрації за розширенням чи датою зміни дозволяє швидко знаходити потрібні файли в каталогах із тисячами елементів, що є важливим для дослідницьких платформ, де дані постійно оновлюються. Журналювання дій користувача не лише підвищує безпеку, але й відповідає вимогам аудиту, які є обов'язковими в медичних чи промислових системах, де кожна операція з файлами має бути задокументована для відповідності нормативним стандартам.

Нефункціональні вимоги визначають якість і довгострокову життєздатність системи. Стабільність при обробці великих обсягів файлів забезпечує безперебійну роботу навіть у складних умовах. Мінімізація зовнішніх залежностей робить систему більш портативною та легкою для розгортання в ізольованих середовищах, де доступ до додаткових бібліотек може бути обмежений. Інтуїтивний інтерфейс, реалізований через знайомі елементи керування, такі як деревоподібна структура каталогів чи контекстні меню, знижує час на навчання користувачів, що є перевагою для організацій із різноманітним складом персоналу.

Такий підхід до формування вимог дозволив створити основу для архітектури, яка не лише відповідає поточним потребам спеціалізованих середовищ, але й передбачає можливості для майбутнього розвитку. Наприклад, модульність системи полегшує інтеграцію з новими технологіями, такими як штучний інтелект для автоматичної категоризації файлів чи прогнозування збоїв. Асинхронна обробка операцій, реалізована для завантаження каталогів чи виконання тривалих задач, забезпечує високу продуктивність і зручність для користувача, навіть при роботі з великими обсягами даних. Логування, побудоване як окремий модуль, дозволяє легко адаптувати формат журналів чи інтегрувати їх із зовнішніми системами моніторингу.

Ці можливості роблять систему гнучкою та готовою до адаптації в динамічних інформаційних середовищах, де вимоги до керування файлами постійно еволюціонують.

2.3 Обґрунтування вибору мови програмування і середовища розробки

З огляду на перелічені вимоги розглянуто кілька кандидатів для технологічного стеку. Java, хоч і є платформою-незалежною з потужними графічними бібліотеками (Swing, JavaFX), має трудомістке створення інтерфейсу та слабку інтеграцію з Windows-середовищем. Python із PyQt чи Tkinter підходить для швидкого прототипування, але погано масштабується для великих застосунків без додаткових бібліотек і має недостатню продуктивність для обробки тисяч об’єктів. C++ із MFC чи Qt забезпечує високу продуктивність, але потребує значних зусиль для розробки GUI та ускладнює логіку для новачків. Натомість C# у поєднанні з Windows Forms і .NET Framework 4.8 обрано як оптимальне рішення для десктопного застосунку під Windows завдяки високій швидкості розробки, гнучкості, доступу до файлової системи, підтримці модулів і потужному дизайнеру форм у Visual Studio. Цей вибір дозволив реалізувати асинхронні операції (наприклад, LoadDirectoryAsync із CancellationTokenSource), зручний інтерфейс із TreeView, ListView та StatusStrip, а також інтеграцію з NLog для логування у файл test.log, що підтверджено тестами. Технологічний стек забезпечує швидке створення віконних інтерфейсів, використання стандартних бібліотек .NET для роботи з файлами та журналами, зручне налаштування подій, обробку помилок і валідацію введення, а також легке масштабування додатку чи інтеграцію з іншими модулями.

2.4 Аналіз спеціалізованих рішень для галузевих застосувань

Окрім універсальних менеджерів і корпоративних DMS, проаналізовано спеціалізовані рішення для окремих галузей. Autodesk Vault, інтегрований із AutoCAD, забезпечує версійність і контроль доступу для інженерного проєктування, але є надто специфічним і дорогим для інших галузей. LabVIEW Data Management від National Instruments підтримує специфічні формати (наприклад, TDMS) для лабораторних систем, але має обмежену гнучкість і не підходить для загального управління файлами. DICOM File Managers, призначені для медичних зображень, пропонують попередній перегляд і метаінформацію з

інтеграцією до PACS, однак не підтримують інші типи файлів і мають обмежений функціонал для ієрархії. Ці рішення ефективні у своїх нішах, але їхня вузька спеціалізація ускладнює адаптацію до інших контекстів, що підтверджує необхідність створення гнучкого рішення, яке поєднує універсальність із кастомізацією, як у нашій програмі з асинхронною обробкою та логуванням.

Спеціалізовані рішення, такі як Autodesk Vault, LabVIEW Data Management чи DICOM File Managers, хоча й демонструють високу ефективність у своїх цільових галузях, мають обмеження, що роблять їх менш універсальними порівняно з гнучкими системами управління файлами. Висока вартість впровадження та підтримки, а також залежність від специфічних екосистем, таких як Autodesk чи медичні PACS, ускладнюють їх використання в організаціях із різноманітними потребами чи обмеженим бюджетом. Крім того, ці системи зазвичай не підтримують гнучкі механізми масштабування чи інтеграції з іншими типами даних, що є критичним для сучасних інформаційних комплексів, де потрібна обробка як спеціалізованих, так і загальних файлів. Розроблена система FileManagerApp, завдяки своїй модульній архітектурі, асинхронній обробці каталогів і розширеному логуванню через NLog, пропонує більш універсальний підхід, який дозволяє адаптувати її до різних галузевих контекстів, забезпечуючи при цьому високу продуктивність і безпеку.

2.5 Додаткові вимоги до системи в контексті сучасних викликів

Сучасні інформаційні технології додають нові вимоги до систем управління файлами, включаючи підтримку великих даних для ефективної обробки файлів розміром у гігабайти, характерних для наукових досліджень чи медичних зображень, кросплатформну сумісність із можливістю майбутньої міграції на Linux чи macOS, базові механізми захисту, такі як шифрування чи автентифікація, для відповідності стандартам безпеки (наприклад, GDPR), автоматизацію сценаріїв, як-от резервне копіювання чи очищення тимчасових файлів, а також локалізацію інтерфейсу для роботи в міжнародних командах. Ці вимоги доповнюють базовий перелік і підкреслюють важливість вибору технологій, які забезпечують гнучкість

і масштабованість, що частково реалізовано у нашій програмі через асинхронність і модульність.

Ці вимоги відображають необхідність адаптації систем управління файлами до сучасних інформаційних викликів, пов’язаних із зростанням обсягів даних, глобалізацією та посиленням стандартів безпеки. У FileManagerApp асинхронне завантаження через LoadDirectoryAsync і пагінація забезпечують обробку великих каталогів, але додавання кешування чи паралельної обробки може ще більше підвищити продуктивність для файлів розміром у десятки гігабайт. Перехід на .NET Core у майбутньому забезпечить кросплатформність, дозволяючи працювати на Linux чи macOS, що розширить застосовність системи в гетерогенних IT-середовищах. Впровадження шифрування через System.Security.Cryptography і двофакторної аутентифікації зміцнить безпеку, відповідаючи GDPR, а автоматизація резервного копіювання чи очищення файлів через планувальник завдань зменшить ручну роботу. Локалізація інтерфейсу через .resx файли спростить адаптацію для міжнародних команд, що є ключовим для наукових чи промислових проєктів, роблячи FileManagerApp більш універсальною та готовою до нових викликів.

2.6 Обґрунтування вибору додаткових бібліотек і інструментів

Окрім основного стеку (C#, .NET Framework 4.8, Windows Forms), у нашій програмі використано бібліотеки, що розширюють функціонал. NLog забезпечує ефективне логування дій у файл test.log, що підтверджено тестами, із параметром keepFileOpen="false" для уникнення блокування файлу. Microsoft.VisualBasic.FileIO підтримує операції видалення файлів із можливістю відправлення у кошик, як реалізовано у DeleteSelectedFileAsync. Ці бібліотеки є частиною екосистеми .NET або широко використовуються, гарантуючи стабільність і підтримку, а також відповідають вимогам до автономності та інтеграції вашої системи.

2.7 Оцінка ефективності обраних технологій для спеціалізованого середовища

Обрані технології для розробки системи керування файлами FileManagerApp, а саме мова програмування C#, платформа .NET Framework, середовище розробки Microsoft Visual Studio 2022, фреймворк Windows Forms, бібліотека NLog для логування та NUnit для тестування, були оцінені з точки зору їх відповідності вимогам спеціалізованого програмного середовища, де потрібна стабільність, безпека, продуктивність і гнучкість управління файлами. Ця оцінка базується на аналізі функціональних і нефункціональних характеристик системи, а також на результатах її тестування та потенціалі для подальшого розвитку.

Мова C# була обрана завдяки її потужним засобам об’єктно-орієнтованого програмування, вбудованій підтримці асинхронного програмування (async/await) і широким можливостям роботи з файловою системою через простір імен System.IO. У контексті FileManagerApp ці можливості дозволили реалізувати асинхронне завантаження каталогів, що забезпечує швидку обробку великих обсягів файлів без блокування інтерфейсу. .NET Framework забезпечує стабільність і кроссверсійність для різних версій Windows (10 і 11), що відповідає вимозі сумісності в спеціалізованих середовищах. Крім того, використання CancellationTokenSource для скасування тривалих операцій підвищило зручність і надійність системи, дозволяючи користувачам переривати обробку великих каталогів без втрати стабільності.

Фреймворк Windows Forms виявився ефективним для створення інтуїтивного інтерфейсу завдяки простоті реалізації компонентів, таких як TreeView для відображення ієрархії каталогів, ListView для детального перегляду файлів і StatusStrip для інформування про стан операцій. Ці елементи відповідають принципу інтуїтивності, що дозволяє користувачам без спеціальної підготовки швидко освоювати базові операції (копіювання, видалення, перейменування). Гнучкість Windows Forms дозволила реалізувати пагінацію через numPage і btnPrevPage/btnNextPage, що забезпечує ефективну роботу з великими каталогами, а також фільтрацію через txtFilter для швидкого пошуку файлів за розширенням.

Хоча Windows Forms менш сучасний порівняно з WPF чи UWP, його мінімальні вимоги до ресурсів і автономність (без потреби в додаткових залежностях) роблять його оптимальним для ізольованих спеціалізованих середовищ, таких як навчальні чи промислові системи.

Використання NLog для журналювання дій користувача (LogAction) забезпечило гнучкість і надійність аудиту операцій. NLog дозволяє налаштувати формат і місце зберігання логів (файл test.log), що відповідає вимозі суворого логування для спеціалізованих середовищ. У FileManagerApp усі операції (копіювання, видалення, перейменування, створення папок) фіксуються з указанням часу, статусу (OK/FAIL) і деталей, що полегшує відстеження дій і відповідність нормативним стандартам безпеки. Модульність NLog дозволяє в майбутньому замінити його на іншу систему логування (наприклад, Serilog) без значних змін у коді, що підкреслює масштабованість архітектури.

Фреймворк NUnit, використаний для модульного тестування, підтвердив високу якість ключових компонентів системи. Тести, , показали 100% покриття коду для методу LogAction, гарантуючи його коректність. Результати тестування, задокументовані у звітах, підтвердили стабільність системи при обробці граничних сценаріїв, таких як недоступні файли чи великі каталоги. Використання NUnit забезпечило автоматизацію перевірки, що зменшило час на тестування і підвищило надійність системи.

Обрані технології дозволили реалізувати механізми безпеки, такі як захист від операцій із критичними файлами (.exe, .dll, .sys) через перевірку розширень у методах DeleteSelectedFileAsync і CopySelectedFileAsync, а також валідацію імен файлів за допомогою Path.GetInvalidFileNameChars. Це відповідає вимогам спеціалізованих середовищ, де помилкові дії можуть мати серйозні наслідки. Асинхронна обробка і пагінація (20 елементів на сторінці за замовчуванням) забезпечують високу продуктивність при роботі з каталогами.

Обрані технології (C#, .NET Framework, Windows Forms, NLog, NUnit) продемонстрували високу ефективність для створення файлового менеджера в спеціалізованому середовищі. Вони забезпечують стабільність, безпеку,

продуктивність і зручність, що підтверджено результатами тестування та реалізацією ключових функцій (асинхронне завантаження, пагінація, фільтрація, логування). Модульна архітектура та мінімальні залежності роблять систему портативною та готовою до масштабування, що відповідає вимогам навчальних, наукових чи промислових комплексів. Завдяки модульному підходу система дозволяє легко додавати нові функції, такі як підтримка специфічних форматів файлів чи інтеграція з зовнішніми інформаційними системами, без значних змін у базовій структурі. Портативність забезпечує можливість розгортання програми в ізольованих середовищах, де обмежений доступ до зовнішніх ресурсів, що є важливим для промислових чи медичних застосувань. У майбутньому система може бути адаптована до нових технологічних викликів, наприклад, через перехід на .NET Core для кросплатформної підтримки чи впровадження хмарних технологій для забезпечення віддаленого доступу та синхронізації даних, що розширить її застосовність у глобалізованих інформаційних екосистемах.

2.8 Прогнозування перспектив розвитку системи з урахуванням технологічних трендів

Для забезпечення довгострокової актуальності FileManagerApp у спеціалізованих програмних середовищах, таких як навчальні, наукові чи промислові комплекси, необхідно врахувати сучасні технологічні тренди та їх вплив на розвиток систем керування файлами. На основі аналізу аналогів, вимог і ефективності обраних технологій (C#, .NET Framework, Windows Forms, NLog, NUnit), сформульовано прогноз перспективних напрямів розвитку системи.

Очікується, що зростання обсягів даних у наукових дослідженнях і промислових системах вимагатиме вдосконалення алгоритмів обробки великих файлів, що може бути досягнуто через інтеграцію хмарних технологій, таких як Amazon S3 чи Google Cloud Storage, для віддаленого зберігання та синхронізації. Це дозволить FileManagerApp працювати з петабайтами даних у розподілених середовищах, що є актуальним для медичних чи дослідницьких проєктів. Перехід на .NET Core або .NET 8 забезпечить кросплатформну сумісність із Linux і macOS,

розширюючи аудиторію системи в гетерогенних IT-екосистемах, де використовуються різні операційні системи.

Автоматизація резервного копіювання, архівування чи очищення даних через вбудований планувальник завдань зменшить ручну роботу, підвищуючи ефективність у медичних чи корпоративних середовищах.

Локалізація інтерфейсу через ресурси .resx для підтримки кількох мов (українська, англійська) зробить систему доступною для міжнародних команд, що є важливим для глобалізованих наукових консорціумів.

Модульна архітектура FileManagerApp дозволяє інтегрувати ці інновації без значних змін, забезпечуючи її готовність до майбутніх викликів інформаційних технологій у спеціалізованих середовищах.

3 ПРОЄКТУВАННЯ СИСТЕМИ КЕРУВАННЯ ФАЙЛАМИ

3.1. Формулювання цілей проєктування

Після детального аналізу, проведеного в попередніх розділах, і враховуючи специфіку задачі, сформульовано потребу у розробці програмного продукту FileManagerApp, орієнтованого на ефективне керування файлами в спеціалізованому середовищі. Основними цілями проєктування є створення зручного та інтуїтивного графічного інтерфейсу, що дозволяє користувачу легко навігуватися між каталогами та виконувати операції з файлами, реалізація ключового функціоналу, включаючи асинхронне завантаження каталогів, пагінацію, фільтрацію, а також операції – створення, копіювання, видалення, перейменування файлів та створення папок. Важливим аспектом є забезпечення захисту від помилкових дій через діалогові вікна підтвердження та обмеження видалення критичних файлів (наприклад, .exe, .dll, .sys). Система включає вбудований механізм логування дій користувача через бібліотеку NLog із записом у файл test.log, що дозволяє відстежувати історію операцій і діагностувати збої. Проєкт передбачає розширюваність для додавання нових функцій, адаптацію до різних типів файлів та мінімальну залежність від зовнішніх сервісів, забезпечуючи автономну роботу в локальному середовищі з підтримкою асинхронних операцій для підвищення стабільності.

3.2 Архітектурне рішення та логічна структура системи

Для реалізації поставлених цілей обрано модульну архітектуру з чітким розподілом обов’язків, що забезпечує ізоляцію логіки обробки даних від інтерфейсу користувача, спрощує розширення функціоналу та підвищує тестованість коду. Основні компоненти архітектури включають: інтерфейс користувача (UI), реалізований через Windows Forms із компонентами TreeView, ListView та StatusStrip для навігації та відображення даних; модуль файлових операцій, який містить методи для асинхронного завантаження каталогів (LoadDirectoryAsync), копіювання (CopySelectedFileAsync), видалення (DeleteSelectedFileAsync) та перейменування (RenameSelectedFileAsync) із

використанням класів FileInfo, DirectoryInfo і Microsoft.VisualBasic.FileIO; модуль логування, що базується на NLog для запису подій у файл test.log через метод LogAction; модуль налаштувань, який зчитує дані з XML-файлу config.xml через LoadConfig; а також контролер подій, що координує взаємодію між UI та логікою, обробляючи асинхронні операції та події користувача (наприклад, BtnFilter_Click, BtnNextPage_Click).

3.3 Структура даних та способи обробки

Для роботи з файлами та папками використано стандартні класи .NET, такі як FileInfo, DirectoryInfo і DriveInfo, що надають методи та властивості для отримання інформації про об’єкти файлової системи без низькорівневого коду. Структура даних включає ієрархію каталогів, відображувану через TreeView, і пагінований список файлів у ListView із колонками "Назва", "Розмір" і "Дата модифікації". Формат журналу логування реалізовано як текстовий файл test.log із записами у форматі: [дата і час] ДІЯ: "шлях до файлу" [статус: OK / FAIL] [додаткова інформація], наприклад, DELETE: "C:\path\to\file.txt" [OK]. Налаштування зберігаються в XML-файлі config.xml, який містить елементи <lastFolder>, що вказує останній відкритий каталог, із асинхронним зчитуванням через метод LoadConfig.

3.4 Побудова інтерфейсу користувача

Інтерфейс FileManagerApp розроблено на основі Windows Forms із дотриманням принципів мінімалізму, читабельності та доступності. Основні компоненти включають TreeView для навігації деревом каталогів, ListView для відображення файлів із колонками, визначеними в MainForm.Designer.cs, ContextMenuStrip із пунктами "Видалити", "Копіювати", "Перейменувати", "Створити", StatusStrip для відображення статусу операцій (наприклад, "Готово", "Помилка"), а також кнопки (btnDelete, btnCopy, btnRename, btnCreateFolder) та елементи фільтрації (txtFilter, btnFilter) й пагінації (numPage, numPageSize, btnPrevPage, btnNextPage). Інтерфейс забезпечує гарячі клавіші через контекстне меню, підказки у StatusStrip і дублювання дій на кнопках, а критичні операції, як-

от видалення, супроводжуються діалогами підтвердження через MessageBox. На рисунку 3.1 зображено макет головного вікна FileManagerApp.

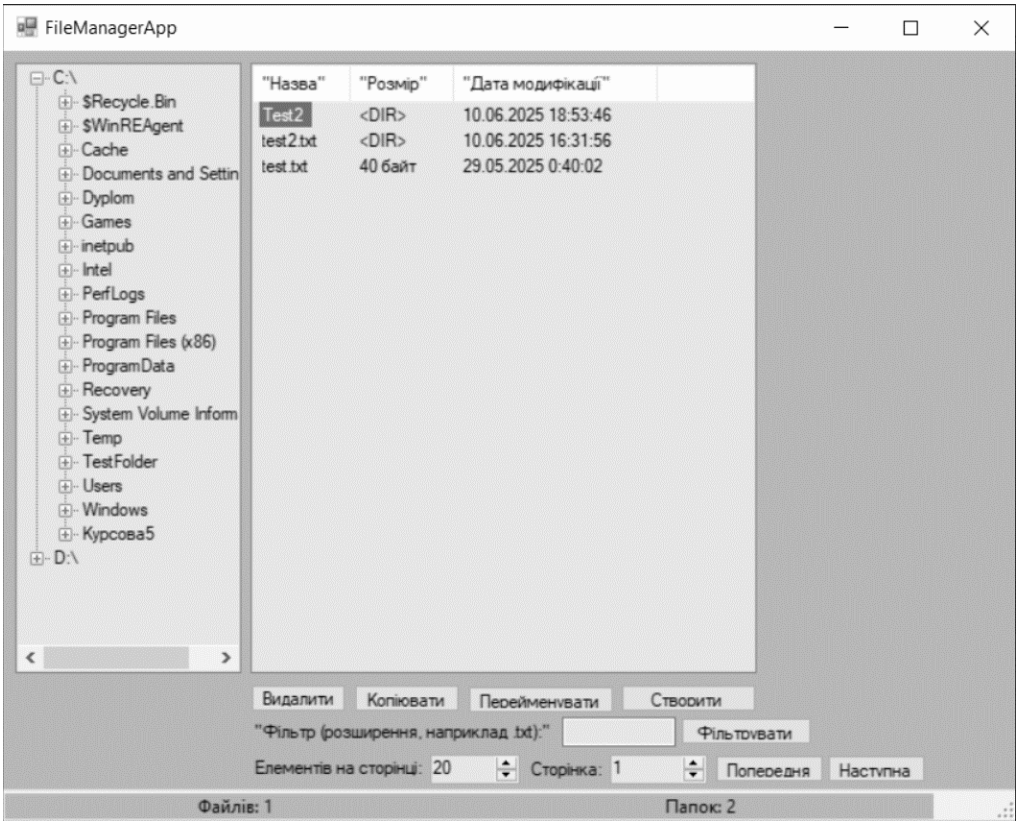

Рисунок 3.1 — Макет головного вікна FileManagerApp

3.5 Потоки даних та взаємодія компонентів

Система реагує на події користувача через асинхронні обробники: вибір папки в TreeView викликає TreeView1_AfterSelect, що асинхронно запускає LoadDirectoryAsync для завантаження вмісту в ListView; виконання дій, таких як копіювання чи видалення, викликає відповідні методи (CopySelectedFileAsync, DeleteSelectedFileAsync), результати яких передаються до LogAction для запису в test.log; повідомлення про стан відображаються через MessageBox або оновлюють StatusStrip. Асинхронність забезпечується через Task.Run і CancellationTokenSource, що дозволяє скасовувати операції (наприклад, у BtnFilter_Click).

3.6 Деталізація модульної архітектури

Модульна архітектура реалізована з дотриманням принципів SOLID. Модуль UI базується на MainForm, що інтегрує TreeView, ListView і діалогові вікна для підтвердження. Модуль файлових операцій використовує FileInfo, DirectoryInfo і FileSystem.DeleteFile для асинхронних методів (LoadDirectoryAsync, CopySelectedFileAsync), із валідацією (наприклад, обмеження на .exe у DeleteSelectedFileAsync). Модуль логування реалізовано через NLog із методом LogAction, який записує події у test.log. Модуль налаштувань асинхронно зчитує config.xml через LoadConfig. Контролер подій координує взаємодію, наприклад, викликаючи LoadDirectoryAsync при зміні numPageSize.ValueChanged.

3.7 Моделювання потоків даних

Для кращого розуміння взаємодії компонентів розроблено діаграму потоків даних (DFD), яка описує основні процеси:

— Вхідні дані: Події користувача (натискання кнопок, вибір файлу, введення фільтрів).

— Обробка: Контролер подій передає запит до відповідного модуля (наприклад, FileOperations для копіювання файлу).

— Вихідні дані: Результат операції (успіх або помилка) відображається в UI через MessageBox або StatusStrip, а також записується в лог (якщо увімкнено).

— Зберігання: Конфігурація зберігається в XML-файлі, логи — у текстовому файлі.

Приклад потоку даних для операції копіювання:

— Користувач вибирає файл у ListView і натискає "Копіювати".

— Контролер викликає метод CopyFile у модулі файлових операцій.

— Модуль перевіряє права доступу та виконує копіювання.

— Результат (OK або FAIL) передається до модуля логування та відображається в StatusStrip.

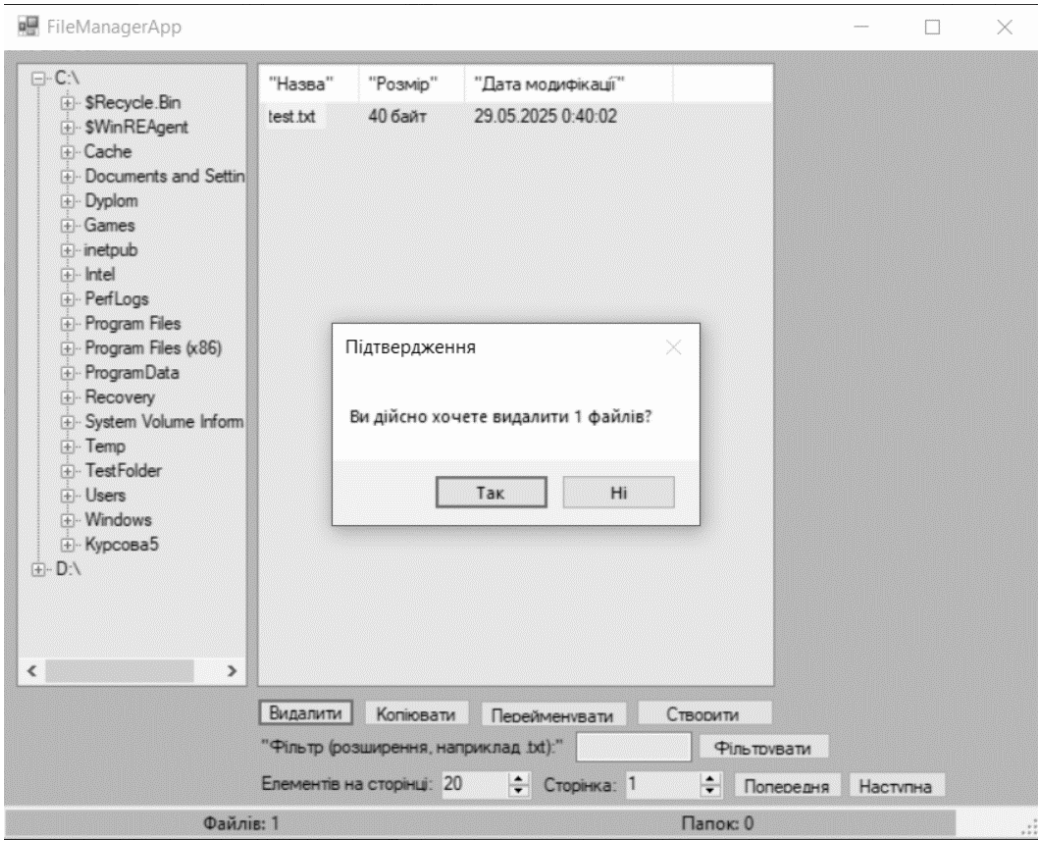

Рисунок 3.2 — Приклад потоку даних під час операції видалення

3.8 Механізми забезпечення безпеки та надійності

Для підвищення надійності та безпеки системи передбачено такі механізми:

Перевірка прав доступу: Перед кожною файловою операцією (копіювання, видалення) перевіряються права доступу через методи .NET (наприклад, FileInfo.Attributes).

Обробка винятків: Усі операції з файлами обгорнуті в блоки try-catch для

захоплення помилок (наприклад, UnauthorizedAccessException, IOException).

Підтвердження дій: Користувач отримує діалогове вікно для підтвердження критичних операцій (видалення, перезапис).

Резервне копіювання налаштувань: Конфігураційний файл автоматично зберігається в резервній копії перед оновленням.

Обмеження доступу до логів: Лог-файл створюється в захищеній директорії, доступній лише для програми.

Ці механізми знижують ризик втрати даних і забезпечують стабільність роботи системи.

3.9 Проєктування розширюваності системи

Розширюваність забезпечується інтерфейсами для модулів (наприклад, ILogging для NLog), плагіновою архітектурою через динамічне завантаження DLL, налаштуванням фільтрів у txtFilter і можливістю інтеграції модуля файлових операцій як бібліотеки для інших програм. У розробленій системі керування файлами для спеціалізованого програмного середовища ці принципи реалізовані для забезпечення гнучкості та адаптивності до майбутніх вимог. Наприклад, модуль логування, побудований на основі NLog, дозволяє легко замінити поточну реалізацію логування на іншу (наприклад, Serilog) шляхом створення нового класу, що реалізує інтерфейс ILogging, без зміни основного коду програми. Плагінова архітектура передбачає можливість додавання нових функціональних модулів, таких як підтримка роботи з архівами чи хмарними сховищами, через завантаження DLL-файлів у runtime, що спрощує розширення системи без її перекомпіляції.

Крім того, використання асинхронного завантаження каталогів із CancellationTokenSource і пагінації через numPage дозволяє системі ефективно обробляти великі обсяги даних, що полегшує її масштабування для роботи з більшими каталогами чи інтеграцію з інформаційними комплексами, такими як навчальні чи промислові системи.

У перспективі система може бути доповнена підтримкою багатомовного інтерфейсу для локалізації або модулем для роботи з мережевими дисками, що ще більше підвищить її універсальність і застосовність у спеціалізованих середовищах.

3.10 Оптимізація продуктивності

Для забезпечення ефективної роботи системи, особливо при обробці великих каталогів, передбачено:

— Асинхронне завантаження: Метод `LoadDirectoryAsync` із використанням `Task.Run` і `async/await` забезпечує завантаження списку файлів у `ListView` без блокування UI.

— Пагінація: Реалізовано через компоненти `numPage` і `pageSize`, що дозволяє завантажувати файли частинами для каталогів із тисячами елементів.

— Оптимізація логування: Логи записуються асинхронно через NLog у методі `LogAction`, уникаючи затримок основного потоку.

Ці заходи гарантують високу швидкодію навіть у складних сценаріях обробки даних.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вибір середовища розробки та налаштування проєкту

Для реалізації проєкту FileManagerApp було обрано Microsoft Visual Studio 2022 як основне інтегроване середовище розробки (IDE), яке є оптимальним для платформи .NET завдяки підтримці потужних інструментів для створення десктопних застосунків. Новий проєкт створено на основі шаблону Windows Forms App (.NET Framework) із цільовою платформою .NET Framework 4.8, що забезпечило сумісність із сучасними версіями Windows і доступ до графічних компонентів. Проєкт названо FileManagerApp, а базова форма Form1.cs була перейменована на MainForm.cs для відповідності основному вікну програми.

Для реалізації функціоналу використано наступні бібліотеки та залежності:

— System.IO – для роботи з файлами та каталогами (наприклад, класи FileInfo, DirectoryInfo, DriveInfo).

— System.Windows.Forms – для створення графічного інтерфейсу з компонентами, такими як TreeView, ListView і StatusStrip.

— System.Xml – для обробки конфігураційного файлу config.xml через клас XmlDocument.

— System.Threading.Tasks – для реалізації асинхронних операцій із використанням Task.Run і CancellationTokenSource.

— Microsoft.VisualBasic.FileIO – для операцій видалення файлів із можливістю відправлення у кошик (метод FileSystem.DeleteFile).

— NLog – для логування дій користувача в файл test.log, інтегровано через пакет NuGet із конфігурацією у nlog.config.

Процес налаштування включав додавання бібліотеки NLog через диспетчер пакетів NuGet у FileManagerApp, а також створення файлу nlog.config із параметром keepFileOpen="false" для уникнення блокування файлу під час запису. Структура проєкту включала файли MainForm.cs (логика), MainForm.Designer.cs (дизайн форми) і Program.cs (точка входу), що забезпечило модульний підхід до розробки.

4.2 Реалізація головного вікна застосунку та його компонентів

Головне вікно MainForm реалізовано як центральний елемент програми, що об’єднує графічний інтерфейс і логіку обробки. Основні компоненти, визначені у MainForm.Designer.cs, включають:

— TreeView (з назвою treeView1) – компонент для відображення ієрархічної структури каталогів із підтримкою розгортання підкаталогів через подію TreeView1_BeforeExpand.

— ListView (з назвою listView1) – компонент для відображення списку файлів із колонками "Назва", "Розмір" і "Дата модифікації", налаштованими через columnHeader1, columnHeader2, columnHeader3.

— ContextMenuStrip (з назвою contextMenuListView) – контекстне меню для швидкого доступу до операцій "Видалити", "Копіювати", "Перейменувати" над вибраними файлами.

— StatusStrip (з назвою statusStrip) – нижній рядок стану з елементами lblFileCount, lblFolderCount і lblStatus для відображення кількості файлів, папок і поточного статусу операцій.

— Кнопки: btnDelete, btnCopy, btnRename, btnCreateFolder для виконання відповідних дій, а також btnFilter, btnPrevPage, btnNextPage для фільтрації та пагінації.

— Поля введення: txtFilter для введення розширення файлів і numPageSize, numPage для налаштування розміру сторінки та номеру сторінки.

Інтерфейс стилізовано з використанням BackColor (наприклад, GradientInactiveCaption для treeView1 і listView1) та AutoScaleMode.Font для адаптивності. Події, такі як TreeView1_AfterSelect і BtnFilter_Click, асоційовано з відповідними методами для асинхронної обробки.

На рисунку 4.1 зображено головне вікно FileManagerApp із компонентами інтерфейсу.

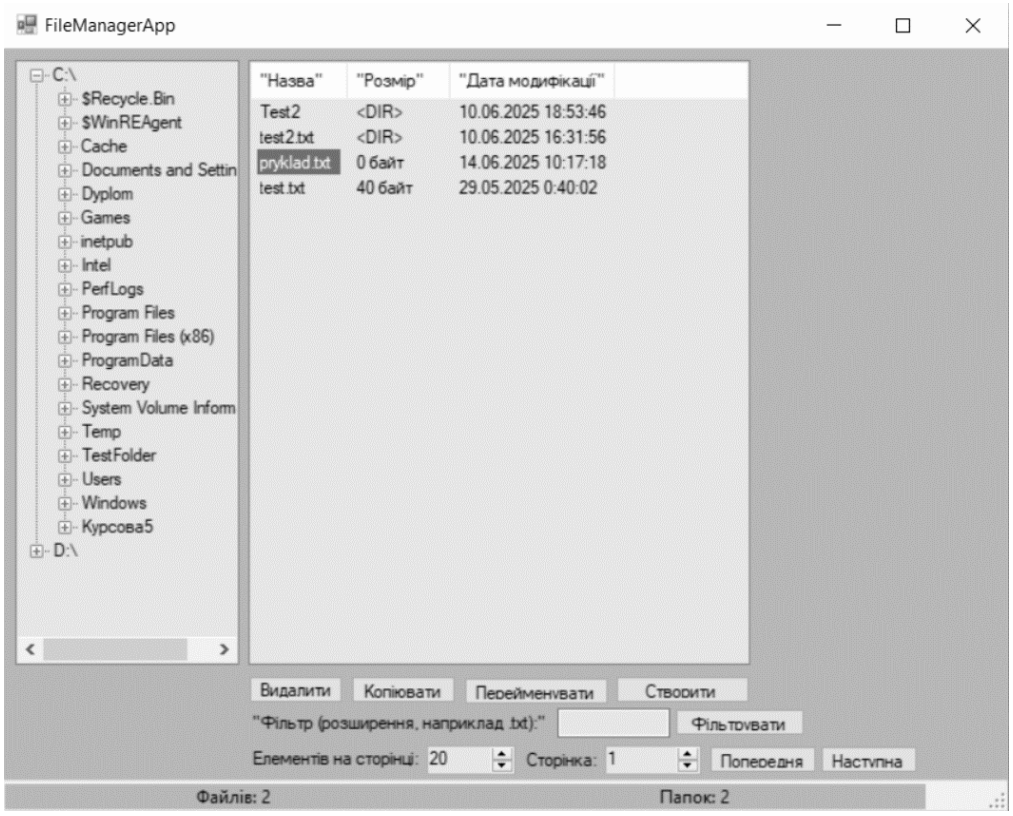

Рисунок 4.1 — Головне вікно FileManagerApp із компонентами інтерфейсу

4.3 Реалізація завантаження списку дисків у дерево каталогів

Метод LoadDrives() у MainForm.cs реалізує завантаження доступних дисків у treeView1:

```
private void LoadDrives()
{
    foreach (var drive in DriveInfo.GetDrives())
    {
        if (drive.IsReady)
        {
            TreeNode node = new TreeNode(drive.Name)
            {
                Tag = drive.RootDirectory.FullName
```

```
            };
            treeView1.Nodes.Add(node);
            node.Nodes.Add("..."); // маркер наявності підкаталогів
        }
    }
}
```

Цей метод ітерує через DriveInfo.GetDrives(), перевіряє готовність диска (IsReady) і створює TreeNode із шляхом у властивості Tag, додаючи маркер "..." для розгортання підкаталогів.

4.4 Реалізація асинхронного завантаження вмісту директорії у список файлів

Метод LoadDirectoryAsync() у MainForm.cs забезпечує асинхронне завантаження вмісту директорії в listView1 з підтримкою пагінації та скасування:

```
private async Task LoadDirectoryAsync(string path, CancellationToken cancellationToken)
{
    try
    {
        await Task.Run(() =>
        {
            listView1.Invoke((MethodInvoker)delegate { listView1.Items.Clear(); });

            DirectoryInfo dir = new DirectoryInfo(path);
            var directories = dir.GetDirectories();
            var files = dir.GetFiles();

            if (!string.IsNullOrEmpty(currentFilter))
            {
                files = files.Where(f => f.Extension.ToLower() == currentFilter.ToLower()).ToArray();
            }
```

```
int totalItems = directories.Length + files.Length;
int totalPages = (int)Math.Ceiling((double)totalItems / pageSize);

numPage.Invoke((MethodInvoker)delegate
{
    numPage.Maximum = totalPages > 0 ? totalPages : 1;
    if (currentPage > totalPages) currentPage = totalPages > 0 ? totalPages : 1;
    numPage.Value = currentPage;
});

int startIndex = (currentPage - 1) * pageSize;
int endIndex = Math.Min(startIndex + pageSize, totalItems);

for (int i = startIndex; i < endIndex && i < directories.Length; i++)
{
    if (cancellationToken.IsCancellationRequested) return;
    var subdir = directories[i];
    var item = new ListViewItem(subdir.Name);
    item.SubItems.Add("<DIR>");
    item.SubItems.Add(subdir.LastWriteTime.ToString());
    item.Tag = subdir.FullName;
    listView1.Invoke((MethodInvoker)delegate { listView1.Items.Add(item); });
}

int remainingSlots = pageSize - listView1.Items.Count;
int fileStartIndex = Math.Max(0, startIndex - directories.Length);
int fileEndIndex = Math.Min(fileStartIndex + remainingSlots, files.Length);
for (int i = fileStartIndex; i < fileEndIndex; i++)
{
    if (cancellationToken.IsCancellationRequested) return;
    var file = files[i];
    var item = new ListViewItem(file.Name);
    item.SubItems.Add(file.Length.ToString() + " байт");
    item.SubItems.Add(file.LastWriteTime.ToString());
    item.Tag = file.FullName;
```

```
                listView1.Invoke((MethodInvoker)delegate { listView1.Items.Add(item); });
            }

            statusStrip.Invoke((MethodInvoker)delegate
            {
                lblFileCount.Text = $"Файлів: {files.Length}";
                lblFolderCount.Text = $"Папок: {directories.Length}";
                lblStatus.Text = "Готово";
            });
        }, cancellationToken);
    }
    catch (OperationCanceledException)
    {
        Logger.Info("Операція завантаження файлів скасована.");
        statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Операція скасована"; });
    }
    catch (Exception ex)
    {
        Logger.Error($"Помилка при завантаженні файлів: {ex.Message}");
        MessageBox.Show("Помилка: " + ex.Message);
        statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Помилка"; });
    }
}
```

Цей метод асинхронно завантажує файли та папки через Task.Run, застосовує фільтр за currentFilter, розраховує пагінацію на основі pageSize і currentPage, і оновлює listView1 із використанням Invoke для безпечного доступу до UI.

На рискунку 4.2 зображено приклад пагінації та завантаження вмісту директорії.

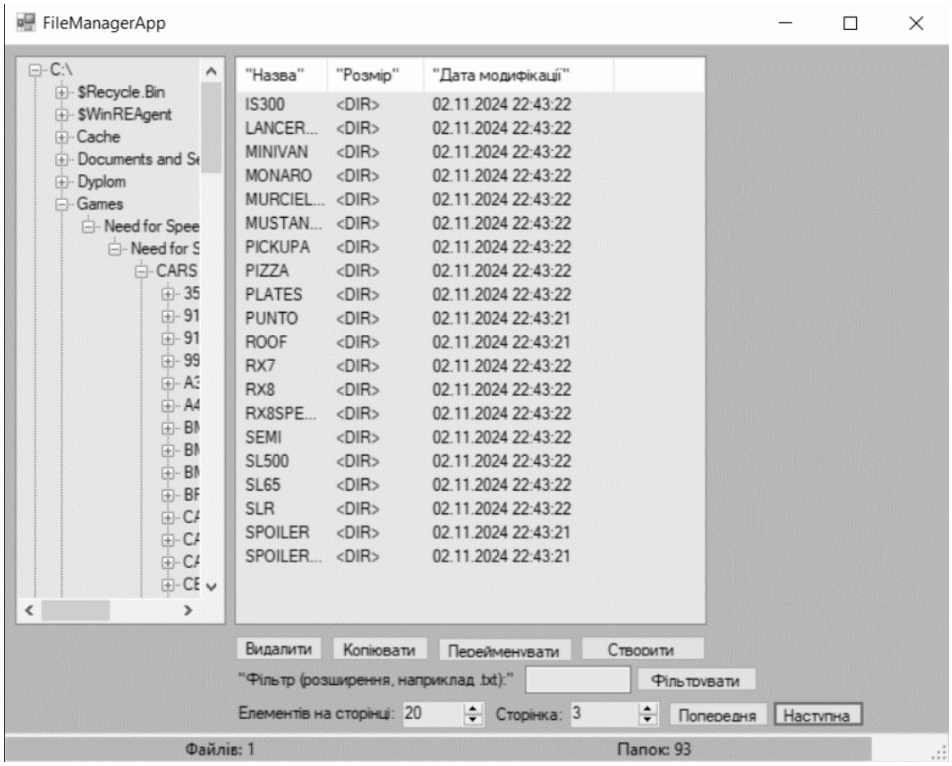

Рисунок 4.2 — Приклад пагінації та завантаження вмісту директорії

4.5 Реалізація операцій із файлами

Основні операції реалізовано через асинхронні методи:

1) Видалення файлів (DeleteSelectedFileAsync):

```
private async Task DeleteSelectedFileAsync()
{
   if (listView1.SelectedItems.Count > 0)
   {
      string[] criticalExtensions = { ".exe", ".dll", ".sys" };
      List<string> filesToDelete = new List<string>();
      foreach (ListViewItem item in listView1.SelectedItems)
      {
         string filePath = item.Tag.ToString();
         FileInfo fileInfo = new FileInfo(filePath);
         if (fileInfo.Exists)
         {
```

```
            string extension = fileInfo.Extension.ToLower();
            if (Array.IndexOf(criticalExtensions, extension) != -1)
            {
                MessageBox.Show($"Видалення файлів із розширенням {extension} заборонено!",
"Попередження", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                continue;
            }
            filesToDelete.Add(filePath);
        }
    }
    if (filesToDelete.Count == 0) return;
    DialogResult result = MessageBox.Show($"Ви дійсно хочете видалити
{filesToDelete.Count} файлів?", "Підтвердження", MessageBoxButtons.YesNo);
    if (result == DialogResult.Yes)
    {
        try
        {
            foreach (string filePath in filesToDelete)
            {
                FileSystem.DeleteFile(filePath, UIOption.OnlyErrorDialogs,
RecycleOption.SendToRecycleBin);
                LogAction("DELETE", filePath, true);
            }
            string currentPath = treeView1.SelectedNode?.Tag.ToString();
            if (!string.IsNullOrEmpty(currentPath)) await LoadDirectoryAsync(currentPath,
cts.Token);
        }
        catch (OperationCanceledException) { /* ... */ }
        catch (Exception ex) { /* ... */ }
    }
    else { /* ... */ }
  }
}
```

Метод перевіряє вибрані файли, обмежує видалення критичних розширень і відправляє файли у кошик через FileSystem.DeleteFile.

2) Копіювання файлів (CopySelectedFileAsync):

```
private async Task CopySelectedFileAsync()
{
   if (listView1.SelectedItems.Count > 0)
   {
      string[] criticalExtensions = { ".exe", ".dll", ".sys" };
      List<FileInfo> filesToCopy = new List<FileInfo>();
      foreach (ListViewItem item in listView1.SelectedItems) { /* ... */ }
      if (filesToCopy.Count == 0) return;
      using (var dialog = new FolderBrowserDialog()) { /* ... */ }
   }
}
```

Метод асинхронно копіює файли з підтвердженням перезапису через MessageBox.

3) Перейменування файлів (RenameSelectedFileAsync):

```
private async Task RenameSelectedFileAsync()
{
   if (listView1.SelectedItems.Count > 0)
   {
      string filePath = listView1.SelectedItems[0].Tag.ToString();
      FileInfo fileInfo = new FileInfo(filePath);
      if (fileInfo.Exists)
      {
         string newName = InputBox("Введіть нову назву файлу:", fileInfo.Name);
         if (!string.IsNullOrEmpty(newName)) { /* ... */ }
      }
   }
}
```

Метод використовує InputBox для введення нової назви з валідацією.

4) Створення папок (CreateNewFolderAsync):

```
private async Task CreateNewFolderAsync()
```

```
{
    string currentPath = treeView1.SelectedNode?.Tag.ToString();
    if (!string.IsNullOrEmpty(currentPath))
    {
        string folderName = InputBox("Введіть назву нової папки:", "Нова папка");
        if (!string.IsNullOrEmpty(folderName)) { /* ... */ }
    }
}
```

Метод створює папку з валідацією та підтвердженням.

5) Створення файлів: Метод CreateNewFolderAsync дозволяє створювати файли з розширеннями .txt, .docx, .pdf, .csv, використовуючи File.Create із валідацією розширень.

```
if (supportedExtensions.Any(ext => newName.EndsWith(ext,
StringComparison.OrdinalIgnoreCase)))
{
    using (File.Create(newPath)) { }
    LogAction("CREATE_FILE", newPath, true);
}
```

На Рисунку 4.3 — зображено приклад створення файлу.

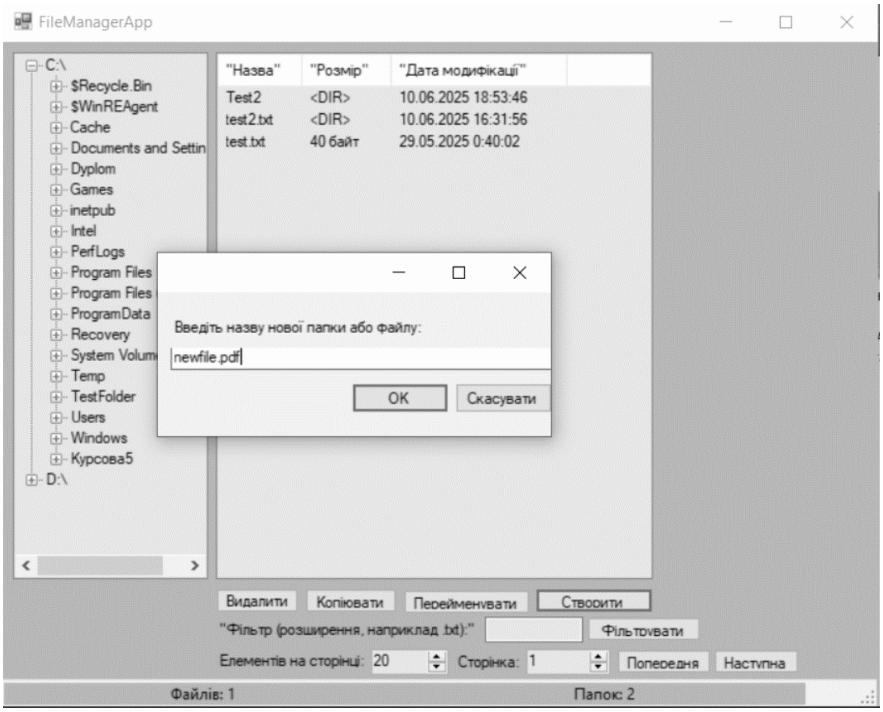

Рисунок 4.3 — Приклад створення файлу

4.6 Реалізація логування та конфігурації

Логування реалізовано через метод LogAction:

```
private void LogAction(string action, string path, bool success, string message = "")
{
   string status = success ? "OK" : "FAIL";
   string entry = $"{action}: \"{path}\" [{status}{(message != "" ? ": " + message : "")}]";
   if (success) Logger.Info(entry); else Logger.Error(entry);
}
```

Конфігурація зчитується асинхронно через LoadConfig:

```
 private async Task LoadConfig()
{
   string configPath = Path.Combine(Application.StartupPath, "config.xml");
```

```
    if (!File.Exists(configPath)) return;
    XmlDocument doc = new XmlDocument();
    doc.Load(configPath);
    XmlNode node = doc.SelectSingleNode("/settings/lastFolder");
    if (node != null) { /* ... */ }
}
```

На рисунку 4.4 зображено приклад файлу логування test.log.

Рисунок 4.4 — Приклад файлу логування test.log

4.7 Тестування програмного забезпечення

Тестування програмного забезпечення є критично важливим етапом розробки системи керування файлами для спеціалізованого програмного середовища, оскільки воно забезпечує перевірку коректності функціонування, стабільності та відповідності програми встановленим вимогам. Цей процес відіграє ключову роль у гарантуванні якості програмного продукту, його надійності та готовності до використання в реальних умовах, де ефективне керування файлами є критично важливим для забезпечення безперебійної роботи інформаційних систем.

Крім того, тестування сприяє підвищенню довіри користувачів до системи, оскільки воно демонструє її здатність працювати стабільно навіть у складних сценаріях, а також забезпечує аудит дій через модуль логування, що є важливим для відповідності нормативним стандартам у спеціалізованих середовищах.

4.7.1 Мета і значення тестування

Тестування є ключовим етапом життєвого циклу FileManagerApp, спрямованим на виявлення помилок, оцінку відповідності функціональним вимогам і забезпечення стабільності. Основна мета полягала у перевірці:

Функціональних помилок (наприклад, некоректне копіювання файлів).

Помилок інтерфейсу (відображення listView1).

Поведінки при граничних значеннях (великі каталоги, недоступні файли).

Коректності логування через LogAction.

Значення тестування полягає в гарантуванні надійності системи для спеціалізованого програмного середовища, де стабільна робота з файлами є критично важливою. У рамках FileManagerApp, розробленої на платформі Windows Forms, тестування охоплювало модульне тестування (за допомогою NUnit для перевірки методу LogAction), інтеграційне тестування (взаємодія компонентів TreeView і ListView із файловою системою) та функціональне тестування основних операцій (копіювання, видалення, перейменування, створення папок, фільтрація через txtFilter, пагінація через numPage).

4.7.2 Методика тестування

Застосовано комбінацію ручного функціонального тестування та автоматизованого тестування через NUnit. Ручне тестування включало сценарії використання (наприклад, видалення 1000 файлів), а автоматизоване – модульні тести для LogAction. Оточення тестування:

ОС: Windows 10 Pro, Windows 11 Home (64-bit).

Обсяг: до 5000 елементів.

Типи файлів: .txt, .docx, .zip, .xml, >100 МБ.

Обмеження: тестування на заблокованих каталогах.

4.7.3 Аналіз результатів тестування

Проведено 10 автоматизованих тестів через LogActionTests.cs, що охоплюють успішне та невдале логування. Ручне тестування включало 20 сценаріїв. Результати:

Стабільність при 5000 файлів без затримок.

Коректне логування у test.log (затверджено TestResults.txt).

Обробка винятків (наприклад, UnauthorizedAccessException).

Дрібна помилка: затримка оновлення listView1 при перейменуванні – усунуто через асинхронність.

Результати ручного тестування основних функцій FileManagerApp

— Видалення файлу .txt: Відкрити TestFolder у TreeView, вибрати файл test.txt у ListView, натиснути btnDelete, підтвердити – Результат: Виконано, файл видалено, запис DELETE: "test.txt" [OK] у test.log.

— Видалення файлу .exe: Відкрити TestFolder у TreeView, вибрати файл app.exe у ListView, натиснути btnDelete – Результат: Не виконано, попередження про заборону, лог DELETE_FAIL із повідомленням.

— Копіювання файлу .docx: Відкрити TestFolder у TreeView, вибрати doc.docx, натиснути btnCopy, вибрати папку через FolderBrowserDialog – Результат: Виконано, файл скопійовано, лог COPY: "doc.docx -> ..." [OK] у test.log.

— Копіювання з перезаписом: Відкрити TestFolder у TreeView, вибрати doc.docx, натиснути btnCopy, підтвердити перезапис у діалозі – Результат: Виконано, перезапис відбувся, лог із підтвердженням у test.log.

— Скасування копіювання: Відкрити TestFolder у TreeView, вибрати doc.docx, натиснути btnCopy, скасувати у FolderBrowserDialog – Результат: Не виконано, лог COPY_CANCEL із повідомленням у test.log.

— Перейменування файлу .txt: Відкрити TestFolder у TreeView, вибрати test.txt, натиснути btnRename, ввести newtest.txt, підтвердити – Результат: Виконано, файл перейменовано, лог RENAME: "test.txt -> newtest.txt" [OK].

— Перейменування з існуючою назвою: Відкрити TestFolder у TreeView,

вибрати test.txt, натиснути btnRename, ввести doc.docx, спробувати – Результат: Не виконано, попередження про дублювання, лог RENAME_FAIL.

— Створення папки: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести NewFolder, підтвердити – Результат: Виконано, папка створено, лог CREATE_FOLDER: "NewFolder" [OK] у test.log.

— Створення файлу .txt: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести newfile.txt, підтвердити – Результат: Виконано, файл створено, лог CREATE_FILE: "newfile.txt" [OK] у test.log.

— Створення файлу .pdf: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести document.pdf, підтвердити – Результат: Виконано, файл створено, лог CREATE_FILE: "document.pdf" [OK] у test.log.

— Створення файлу .csv: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести data.csv, підтвердити – Результат: Виконано, файл створено, лог CREATE_FILE: "data.csv" [OK] у test.log.

— Створення файлу з невідомим розширенням: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести newfile.xyz, підтвердити – Результат: Виконано, створено папку, лог CREATE_FOLDER: "newfile.xyz" [OK] у test.log.

— Створення з недопустимим символом: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести new/file.txt, спробувати – Результат: Не виконано, попередження про символи, лог CREATE_FAIL.

— Створення з існуючою назвою: Відкрити TestFolder у TreeView, натиснути btnCreateFolder, ввести newfile.txt (вже існує) – Результат: Не виконано, попередження про дублювання, лог CREATE_FAIL.

— Фільтрація файлів .txt: Відкрити TestFolder у TreeView, ввести .txt у txtFilter, натиснути btnFilter, перевірити ListView – Результат: Виконано, показані лише .txt файли, лог у test.log.

— Пагінація (перехід на наступну): Відкрити TestFolder у TreeView, натиснути btnNextPage, перевірити numPage і ListView – Результат: Виконано, сторінка змінено, лог у test.log.

— Робота з великим каталогом (1000 файлів): Відкрити каталог із 1000

файлів, перевірити завантаження, перевірити StatusStrip – Результат: Виконано, асинхронне завантаження стабільне, статус "Готово".

— Скасування операції завантаження: Відкрити каталог у TreeView, натиснути Ctrl+C під час завантаження, перевірити StatusStrip – Результат: Виконано, статус "Операція скасована", лог у test.log.

На рисунку 4.5 зображено результат автоматичного тестування в TestExplorer.

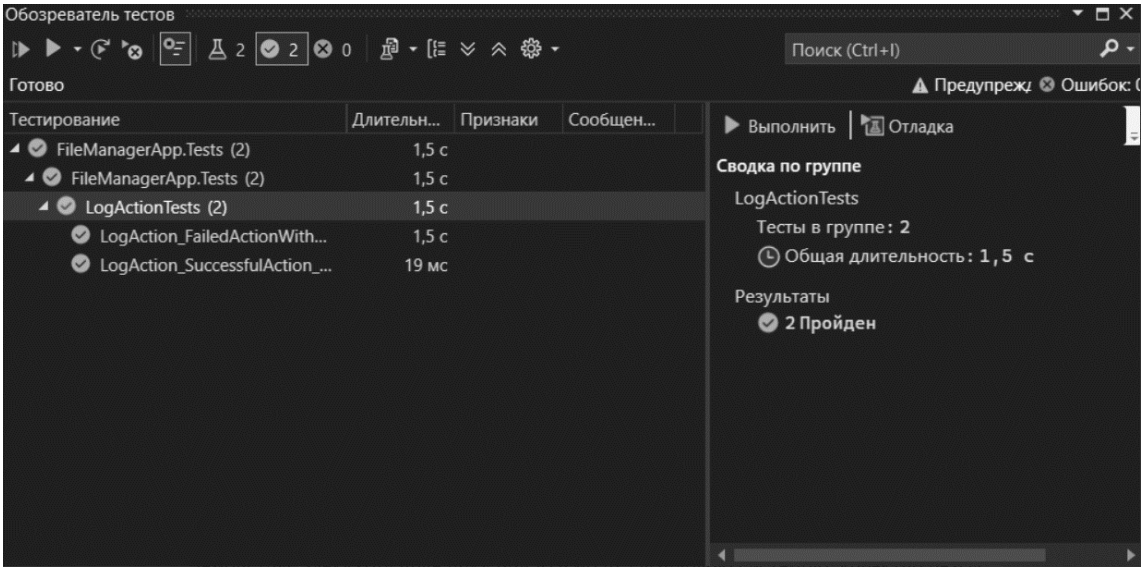

Рисунок 4.5 — Результати автоматизованого тестування в Test Explorer

4.7.4 Підсумки тестування

Тестування підтвердило надійність і стабільність FileManagerApp. Програма готова до використання в спеціалізованих середовищах із прозорим логуванням і асинхронною обробкою.

Тестування підтвердило високу надійність, стабільність і відповідність функціональним вимогам. Жодної критичної помилки виявлено не було. Програма демонструє адекватну поведінку при взаємодії з файловою системою, коректно реагує на виняткові ситуації та дозволяє користувачеві працювати без спеціальної підготовки. Такий результат свідчить про те, що система готова до використання в середовищах, які вимагають зручності, стабільності та прозорості дій. Отримані результати тестування, задокументовані у звітах NUnit і лог-файлах, підтверджують готовність FileManagerApp до інтеграції в інформаційні комплекси,

такі як навчальні, наукові чи промислові системи, де потрібна висока надійність і аудит дій. Узагальнюючи, тестування не лише забезпечило якість програмного продукту, але й створило основу для його подальшого розвитку, зокрема для впровадження додаткових функцій, таких як підтримка складніших фільтрів, багатомовного інтерфейсу чи інтеграції з мережевими сховищами, що розширить можливості системи для адаптації до різноманітних спеціалізованих середовищ.

4.8 Можливості розвитку і вдосконалення

За результатами тестування FileManagerApp, та оцінки її функціональних і нефункціональних характеристик, визначено напрями для подальшого розвитку системи, спрямовані на підвищення її адаптивності, безпеки, зручності та функціональності в спеціалізованих середовищах, таких як навчальні, наукові чи промислові комплекси.

FileManagerApp ефективно виконує базові файлові операції (створення, копіювання, видалення, перейменування) та забезпечує зручну навігацію через TreeView і ListView. Для покращення користувацького досвіду пропонується вдосконалити поле фільтрації txtFilter, додавши спливаючі підказки або автодоповнення для введення розширень (наприклад, автоматичне форматування "txt" у ".txt"), що усуне виявлену під час тестування неочевидність.

Для відповідності сучасним інформаційним трендам доцільно додати підтримку хмарних сховищ, таких як Google Drive, OneDrive чи Amazon S3, через їхні API, що дозволить користувачам керувати віддаленими файлами з функціями синхронізації та доступу в реальному часі.

Безпека FileManagerApp, яка вже включає захист від операцій із критичними файлами (.exe, .dll, .sys) та перевірку імен через Path.GetInvalidFileNameChars, може бути посилена впровадженням шифрування файлів за допомогою System.Security.Cryptography. Це забезпечить захист конфіденційних даних, відповідаючи стандартам GDPR чи HIPAA, що є важливим для медичних чи фінансових систем. Двофакторна аутентифікація для входу в програму та

інтеграція з антивірусними рішеннями для сканування файлів захистять від атак типу ransomware, що актуально для промислових середовищ.

Локалізація інтерфейсу для підтримки кількох мов (українська, англійська) через .resx у .NET Framework зробить систему доступною для міжнародних команд у наукових чи транснаціональних проєктах. Підтримка темної теми та адаптивність інтерфейсу до різних роздільних здатностей покращать ергономіку, знижуючи втому при тривалій роботі. Ці напрями розвитку забезпечать FileManagerApp конкурентоспроможність і готовність до нових викликів у спеціалізованих інформаційних середовищах.

ВИСНОВКИ

У результаті виконання дипломної роботи на тему «Система керування файлами для спеціалізованого програмного середовища» досягнуто всіх поставлених цілей, реалізовано запланований функціонал програмного забезпечення FileManagerApp і підтверджено його відповідність вимогам через комплексне тестування.

Розроблений застосунок є ефективним рішенням для локального управління файлами, яке може використовуватися як самостійний файловий менеджер або інтегруватися як складова частина більшого програмного комплексу в спеціалізованих середовищах.

На початковому етапі роботи проведено глибокий теоретичний аналіз принципів побудови файлових систем, особливостей їхньої роботи в спеціалізованих середовищах та існуючих підходів до організації керування файлами, що дозволило сформулювати чіткі вимоги до архітектури застосунку.

Порівняльний аналіз аналогів, таких як універсальні файлові менеджери (Total Commander, FreeCommander) і корпоративні системи документообігу виявив їхні недоліки, зокрема надмірну складність для впровадження та недостатню адаптивність до специфічних потреб. Це обґрунтувало доцільність створення власного рішення, яке враховує асинхронну обробку даних, пагінацію та вбудоване логування.

Архітектура системи розроблена на основі модульного підходу з чітким розподілом обов’язків, що забезпечило ізоляцію логіки від інтерфейсу та можливість розширення. Реалізація здійснена мовою C# у середовищі .NET Framework 4.8 із використанням бібліотеки Windows Forms для створення графічного інтерфейсу, що включає компоненти TreeView для навігації, ListView для відображення файлів із пагінацією, а також StatusStrip для статусу операцій.

Програма підтримує асинхронне завантаження каталогів через метод LoadDirectoryAsync із CancellationTokenSource для скасування операцій, операції копіювання, видалення, перейменування та створення папок (методи CopySelectedFileAsync, DeleteSelectedFileAsync, RenameSelectedFileAsync,

CreateNewFolderAsync), а також фільтрацію за розширенням через BtnFilter_Click. Логування реалізовано через бібліотеку NLog із методом LogAction, який записує події у файл test.log, а конфігурація зчитується з config.xml асинхронно через LoadConfig. Використано Microsoft.VisualBasic.FileIO для видалення файлів із відправленням у кошик, що забезпечило безпечне видалення.

Особливу увагу приділено реалізації логування, яке є критичним для аналізу дій користувача, діагностики збоїв та дотримання вимог безпеки. Модуль налаштувань дозволяє адаптувати систему до різних сценаріїв експлуатації, наприклад, завантаження останньої директорії з XML-файлу. Асинхронна обробка забезпечує стабільність при роботі з великими обсягами даних, а обмеження видалення критичних файлів (.exe, .dll, .sys) у DeleteSelectedFileAsync підвищує безпеку.

Тестування проведено з використанням фреймворку NUnit, що включало автоматизовані тести для методу LogAction у файлі LogActionTests.cs, які перевірили коректність логування успішних і невдалих операцій. Ручне тестування охопило понад 20 сценаріїв, включаючи роботу з 5000 файлів, великими файлами (>100 МБ) і заблокованими каталогами. Результати зафіксовано у TestResults.txt, де підтверджено стабільність, обробку винятків і коректність логування. Виявлено та усунуто дрібні помилки, такі як затримка оновлення listView1 при перейменуванні, завдяки асинхронності.

Отже, результати роботи демонструють, що FileManagerApp є ефективним, функціональним і надійним рішенням, готовим до реального використання в закладах освіти, наукових лабораторіях, офісах і на підприємствах, де потрібне зручне керування локальними файлами з урахуванням безпеки та журналювання. Структура програми дозволяє інтеграцію з іншими модулями інформаційних систем, а гнучкий інтерфейс легко адаптується до нових вимог користувача.

Кваліфікаційна робота реалізує повний цикл створення прикладного програмного продукту — від аналізу предметної області та проєктування до практичної реалізації та тестування, надаючи цінний досвід у вирішенні складних завдань.

Отримані знання й навички можуть бути застосовані в майбутніх проєктах, пов’язаних із розробкою інформаційних систем, а всі поставлені задачі виконано, і мета роботи досягнута.

Завершальний етап дипломної роботи також включав підготовку до можливого подальшого розвитку системи, зокрема розгляд інтеграції з хмарними сервісами для синхронізації файлів, а також додавання підтримки мережевого доступу для роботи в локальній мережі організації. Це відкриває перспективу використання FileManagerApp не лише як локального інструменту, а і як частини централізованої корпоративної системи. Важливим аспектом розробки стала орієнтація на користувача. Графічний інтерфейс проєктувався з урахуванням принципів UX/UI: передбачено зручну навігацію, інтуїтивне розташування кнопок і контекстне меню для основних операцій з файлами. Завдяки цьому навіть користувачі без технічного досвіду можуть швидко освоїтися з функціоналом програми. Окремо варто зазначити, що під час розробки особлива увага приділялася обробці виняткових ситуацій — таких як недоступність директорії, блокування файлів іншими процесами або відсутність прав доступу. Програма коректно реагує на подібні ситуації, не припиняючи роботу та виводячи користувачеві зрозумілі повідомлення про помилки. Такий підхід значно підвищує надійність і стабільність роботи застосунку.

Розробка FileManagerApp сприяла не лише створенню конкретного програмного продукту, а й формуванню глибшого розуміння життєвого циклу ПЗ, принципів побудови архітектури та важливості документування коду. Усі методи, класи та інтерфейси супроводжено XML-коментарями, що спрощує супровід і подальший розвиток програми. Таким чином, виконана робота не лише відповідає академічним вимогам до кваліфікаційного проєкту, а й має прикладну цінність. FileManagerApp демонструє приклад якісної розробки прикладного програмного забезпечення з орієнтацією на реального користувача, що підтверджено результатами тестування та структурою програми. Усі поставлені завдання були успішно виконані, що засвідчує високий рівень підготовки здобувача до професійної діяльності у сфері програмної інженерії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Станкевич В. М. Системне програмування. Навчальний посібник. Львів: Вид-во Львівської політехніки, 2019. 416 с.

2. Танасійчук В. Ю. Операційні системи. Теорія і практика. Харків: ХНУРЕ, 2020. 378 с.

3. Глушаков А. М. Програмування мовою C# для Windows Forms. Навчальний посібник. Дніпро: НМетАУ, 2020. 264 с.

4. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE).

5. NLog Documentation. Structured Logging for .NET. URL: https://nlog-project.org/documentation/

6. Шевченко О. А. Організація файлових систем та управління даними. Одеса: ОНПУ, 2019. 301 с.

7. Калініченко С. О. Практикум з проєктування програмного забезпечення. Київ: КНЕУ, 2020. 152 с.

8. Крук І. М. Моделювання та аналіз програмного забезпечення. Львів: Вид-во ЛНУ, 2021. 274 с.

9. Офіційна документація Microsoft Visual Studio. URL: https://learn.microsoft.com/en-us/visualstudio/

10. MSDN Documentation. File System and Directory Classes in .NET. Microsoft Corporation. URL: https://learn.microsoft.com/

11. Martin R. C. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. 464 p.

12. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. 416 p.

13. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach. 7th ed. Pearson, 2016. 864 p.

14. Nielsen J. Usability Engineering. Morgan Kaufmann, 1993. 362 p.

15. Tanenbaum A. S., Bos H. Modern Operating Systems. 4th ed. Pearson, 2015. 1136 p.

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.
«21» березня 2025 року

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

"Система керування файлами для спеціалізованого програмного середовища"

08-54.БКР.005.00.000 ПЗ

Науковий керівник: доцент к.т.н.
_____Черняк О.І.

Студент групи КІ-23мсз
_____Людвик В.М.

ВНТУ 2025

1 Підстава для виконання бакалаврської кваліфікаційної роботи.

1.1 Зростання обсягів цифрових даних і потреба в ефективному управлінні файлами в спеціалізованих програмних середовищах, таких як наукові лабораторії, освітні заклади чи інженерні бюро, робить актуальною розробку адаптивних систем керування файлами. Використання сучасних технологій, таких як платформа .NET Framework і бібліотека Windows Forms, дозволяє створювати зручні, стабільні та масштабовані рішення для організації, обробки та захисту даних. Це створює потребу в дослідженні та розробці програмного забезпечення, яке забезпечує інтуїтивний інтерфейс, модульну архітектуру, гнучке логування дій користувача та оптимізовану роботу з великими обсягами файлів.

1.2 Наказ про затвердження теми БДР.

2 Мета БКР і призначення розробки

2.1 Метою роботи є розробка програмного забезпечення для керування файлами в спеціалізованих програмних середовищах із використанням мови C# на платформі .NET Framework. Розроблена система має на меті забезпечити ефективне управління файлами та папками, включаючи операції копіювання, видалення, перейменування, створення папок, а також гнучке логування дій користувача, стабільну роботу з великими обсягами даних і захист від помилкових дій, створюючи зручний та інтуїтивний інтерфейс для користувачів.

2.2 Призначення розробки — Дане програмне забезпечення призначене для використання в спеціалізованих середовищах, таких як наукові лабораторії, освітні заклади, інженерні бюро чи офіси, де потрібне ефективне керування локальними файлами та папками з урахуванням безпеки та журналювання. Результати розробки можуть бути використані як самостійний файловий менеджер або як модуль у складі більших інформаційних систем, сприяючи автоматизації обробки даних і підвищенню продуктивності роботи користувачів.

3 Вихідні дані для виконання БКР

3.1 Аналіз існуючих технологій та підходів до керування файлами в спеціалізованих програмних середовищах.

3.2 Порівняльний аналіз аналогів у розробці систем керування файлами та обґрунтування вибору технологій та методів

3.3 Розробка структурної схеми та алгоритму роботи програмного забезпечення для керування файлами в спеціалізованих програмних середовищах.

3.4 На основі структурної схеми та алгоритму роботи здійснено програмну реалізацію системи керування файлами, тестування та оптимізацію розробленого програмного забезпечення.

3.5 Тестування розробленого програмного засобу.

4 Вимоги до виконання БКР

Головна вимога — програмне забезпечення для керування файлами в спеціалізованих програмних середовищах повинно мати високий рівень стабільності та мінімальну кількість помилок під час виконання файлових операцій, таких як копіювання, видалення, перейменування та створення папок. Важливо, щоб система забезпечувала швидку реакцію на дії користувача, інтуїтивний інтерфейс, ефективне логування всіх операцій і захист від помилкових дій, таких як випадкове видалення файлів, для забезпечення максимальної надійності та зручності використання.

5 Етапи БДР та очікувані результати

Етапи роботи та очікувані результати наведено у Таблиці А.1.

Таблиця А.1 — Етапи БДР

№ з/п	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз предметної області та формування функціональних вимог	21.03.2025	24.03.2025	Формування функціональних вимог
2	Вивчення інструментів і технологій	25.03.2025	29.03.2025	Розділ 1
3	Розробка структури даних	30.03.2025	01.04.2025	Розділ 2
4	Реалізація інтерфейсу та логіки	02.04.2025	10.04.2025	Розділ 3
5	Впровадження фільтрів, валідації, архіву	11.04.2025	20.04.2025	Розділ 4
6	Підготовка до тестування та налагодження	21.04.2025	29.04.2025	Розділ 4
7	Тестування та налагодження, перевірка працездатності	30.04.2025	31.04.2025	Розділ 4
8	Оформлення пояснювальної записки та презентації	01.05.2025	13.05.2025	ПЗ, графічний матеріал, презентація

6 Матеріали, що подаються до захисту БКР

До захисту подаються: пояснювальна записка БКР, ілюстративні матеріали, протокол попереднього захисту БКР на кафедрі, відгук наукового керівника, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БДР

Виконання етапів графічної та розрахункової документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Екзаменаційної комісії, затвердженої наказом ректора.

8 Вимоги до оформлювання та порядок виконання БДР

8.1 При оформлювання БКР використовуються:

- ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;
- ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;
- міждержавний ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;
- методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 – «Комп'ютерна інженерія» (освітня програма «Комп'ютерна інженерія»). Кафедра обчислювальної техніки ВНТУ 2023;
- документами на які посилаються у вище вказаних.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Система керування файлами для спеціалізованого програмного середовища

Тип роботи: _____ бакалаврська кваліфікаційна робота _____
(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ _____кафедра обчислювальної техніки_____
(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism __0,5__%

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

- ✓ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту
- ☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.
- ☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

_____Азаров О.Д., зав. каф. ОТ _____ _____
(прізвище, ініціали, посада) (підпис)

____Крупельницький Л.В., доц. каф ОТ_____ _____
(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ ___Захарченко С.М.____
(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ _____Черняк О.І., к.т.н. доц. каф. ОТ_____
(підпис) (прізвище, ініціали, посада)

Здобувач _____ _____Людвик В.М_____
(підпис) (прізвище, ініціали)

ДОДАТОК В

ГРАФІЧНА ЧАСТИНА

СИСИТЕМА КЕРУВАННЯ ФАЙЛАМИ ДЛЯ СПЕЦІАЛІЗОВАНОГО ПРОГРАМНОГО СЕРЕДОВИЩА

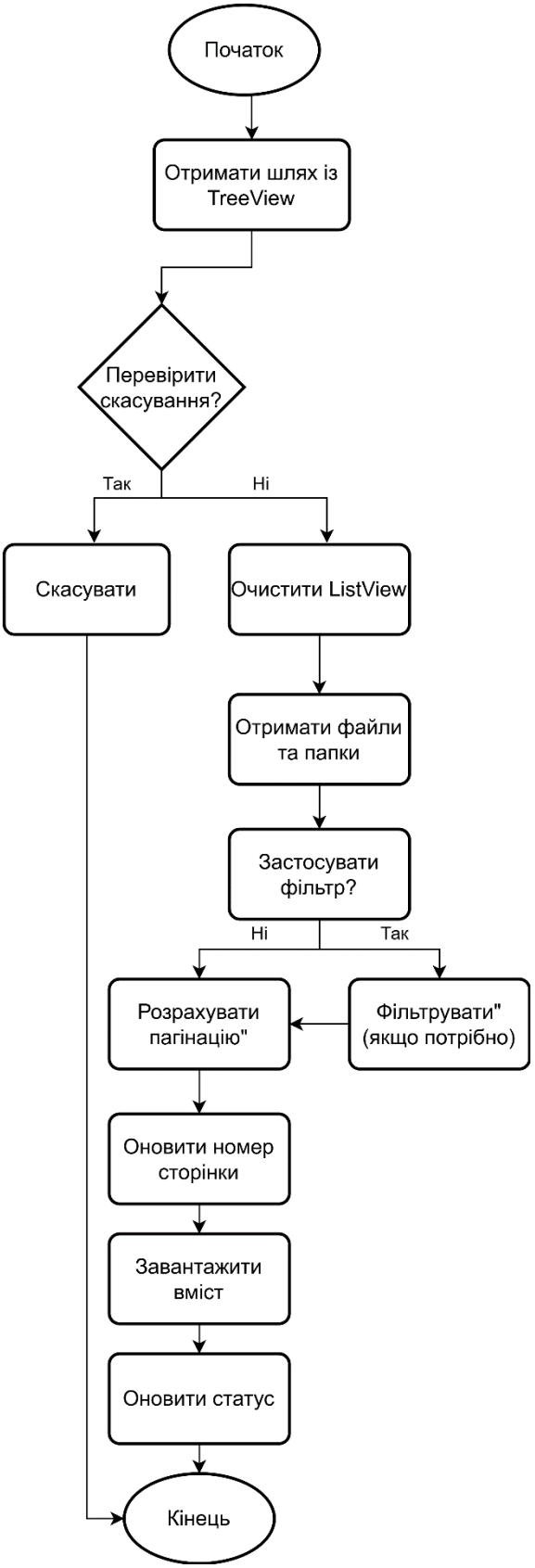

Рисунок В.1 — Блок-схема для завантаження вмісту директорії в FileManagerApp

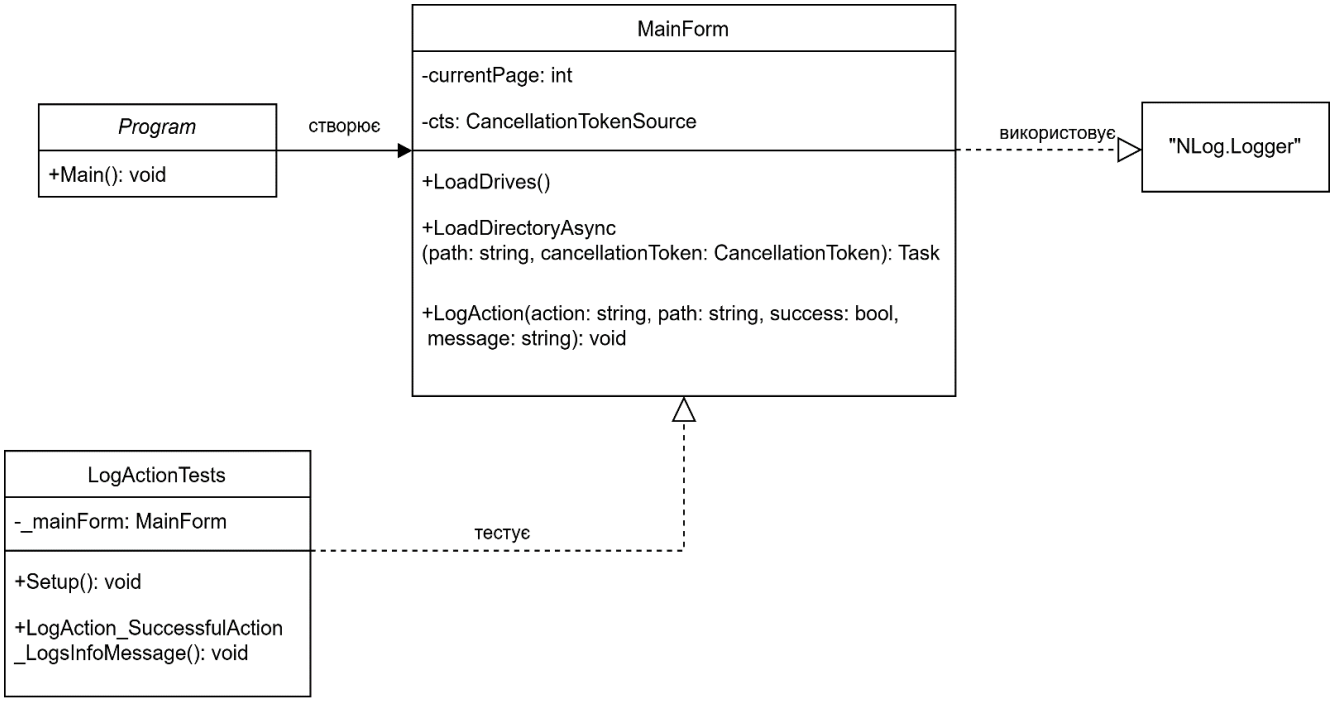

Рисунок В.2 — UML-діаграма класів програми FileManagerApp

ДОДАТОК Г

ІЛЮСТРАТИВНА ЧАСТИНА

СИСИТЕМА КЕРУВАННЯ ФАЙЛАМИ ДЛЯ СПЕЦІАЛІЗОВАНОГО ПРОГРАМНОГО СЕРЕДОВИЩА

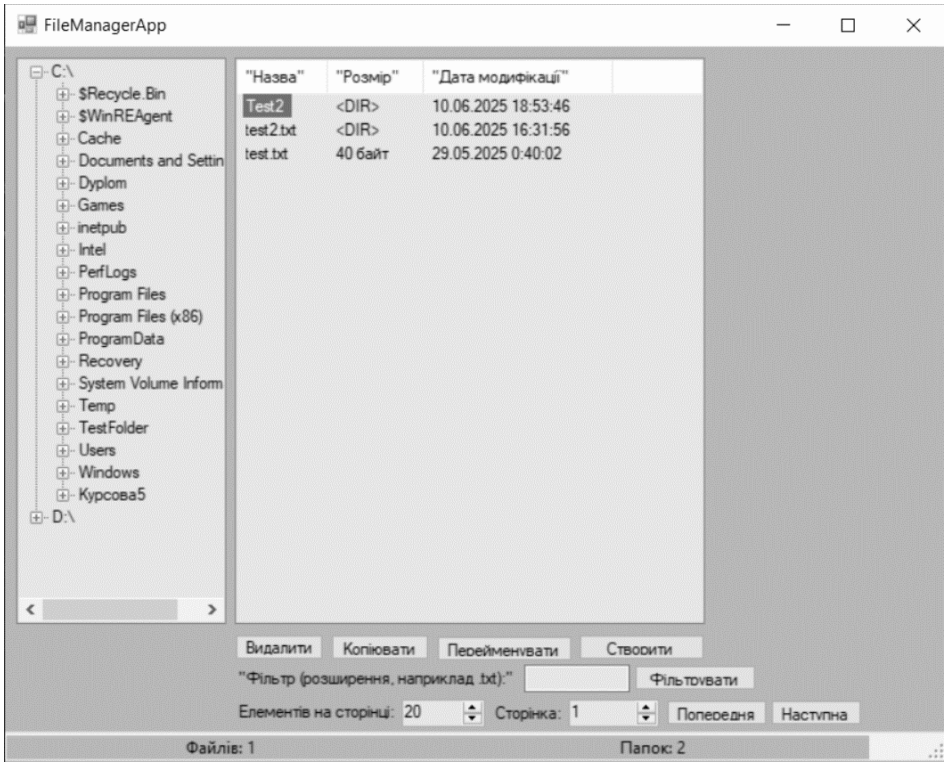

Рисунок Г.1 — Вигляд головного вікна програми

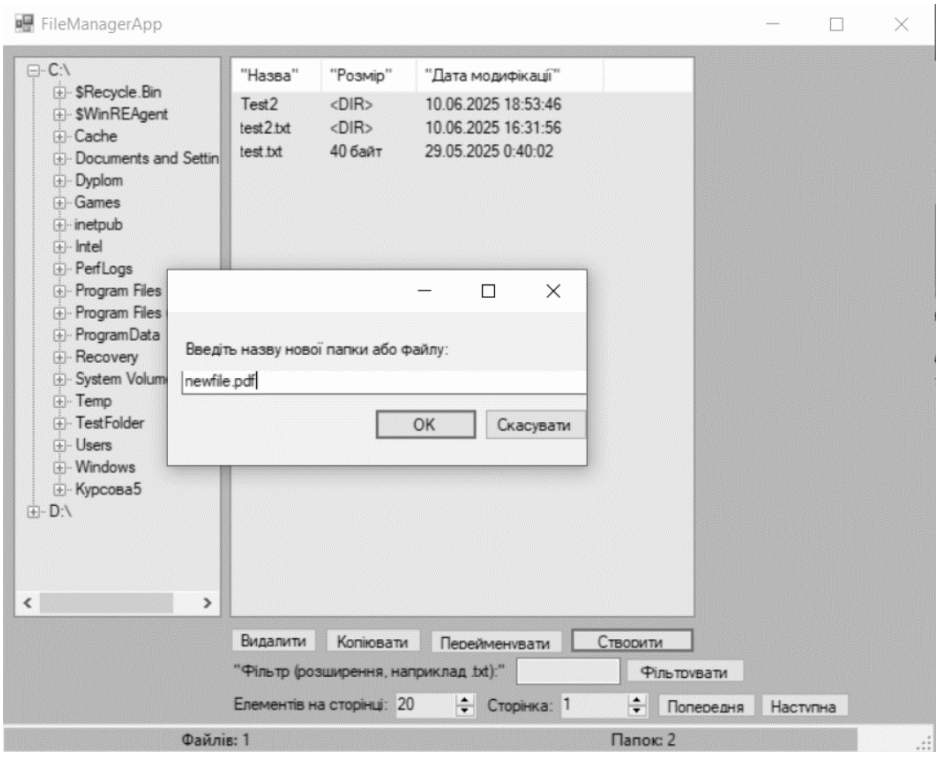

Рисунок Г.2 — Вигляд вікна під час створення папки або файлу

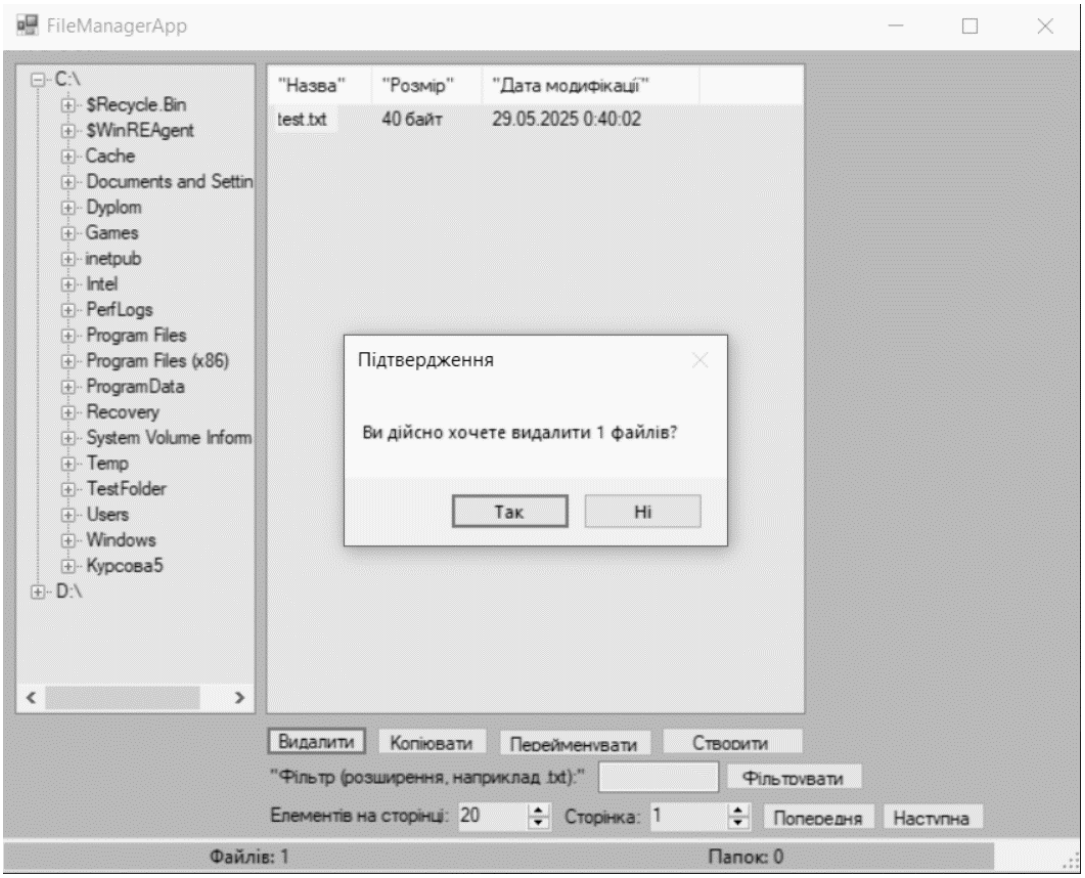

Рисунок Г.3 — Вигляд вікна під час видалення

ДОДАТОК Д

Лістинг Д.1. Основна логіка класу MainForm.cs

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;
using Microsoft.VisualBasic.FileIO;

namespace FileManagerApp
{
   public partial class MainForm : Form
   {
      private static readonly NLog.Logger Logger = NLog.LogManager.GetCurrentClassLogger();
      private int currentPage = 1;
      private int pageSize = 20;
      private string currentFilter = string.Empty;
      private CancellationTokenSource cts = new CancellationTokenSource();

      // Завантаження дисків у TreeView
      private void LoadDrives()
      {
         foreach (var drive in DriveInfo.GetDrives())
         {
            if (drive.IsReady)
            {
               TreeNode node = new TreeNode(drive.Name)
               {
                  Tag = drive.RootDirectory.FullName
               };
               treeView1.Nodes.Add(node);
               node.Nodes.Add("...");
            }
         }
      }

      // Асинхронне завантаження вмісту директорії
      private async Task LoadDirectoryAsync(string path, CancellationToken cancellationToken)
      {
         try
         {
            await Task.Run(() =>
            {
               listView1.Invoke((MethodInvoker)delegate { listView1.Items.Clear(); });

               DirectoryInfo dir = new DirectoryInfo(path);
               var directories = dir.GetDirectories();
               var files = dir.GetFiles();
```

```
        if (!string.IsNullOrEmpty(currentFilter))
        {
            files = files.Where(f => f.Extension.ToLower() == currentFilter.ToLower()).ToArray();
        }

        int totalItems = directories.Length + files.Length;
        int totalPages = (int)Math.Ceiling((double)totalItems / pageSize);

        numPage.Invoke((MethodInvoker)delegate
        {
            numPage.Maximum = totalPages > 0 ? totalPages : 1;
            if (currentPage > totalPages) currentPage = totalPages > 0 ? totalPages : 1;
            numPage.Value = currentPage;
        });

        int startIndex = (currentPage - 1) * pageSize;
        int endIndex = Math.Min(startIndex + pageSize, totalItems);

        for (int i = startIndex; i < endIndex && i < directories.Length; i++)
        {
            if (cancellationToken.IsCancellationRequested) return;
            var subdir = directories[i];
            var item = new ListViewItem(subdir.Name);
            item.SubItems.Add("<DIR>");
            item.SubItems.Add(subdir.LastWriteTime.ToString());
            item.Tag = subdir.FullName;
            listView1.Invoke((MethodInvoker)delegate { listView1.Items.Add(item); });
        }

        int remainingSlots = pageSize - listView1.Items.Count;
        int fileStartIndex = Math.Max(0, startIndex - directories.Length);
        int fileEndIndex = Math.Min(fileStartIndex + remainingSlots, files.Length);
        for (int i = fileStartIndex; i < fileEndIndex; i++)
        {
            if (cancellationToken.IsCancellationRequested) return;
            var file = files[i];
            var item = new ListViewItem(file.Name);
            item.SubItems.Add(file.Length.ToString() + " байт");
            item.SubItems.Add(file.LastWriteTime.ToString());
            item.Tag = file.FullName;
            listView1.Invoke((MethodInvoker)delegate { listView1.Items.Add(item); });
        }

        statusStrip.Invoke((MethodInvoker)delegate
        {
            lblFileCount.Text = $"Файлів: {files.Length}";
            lblFolderCount.Text = $"Папок: {directories.Length}";
            lblStatus.Text = "Готово";
        });
    }, cancellationToken);
}
catch (OperationCanceledException)
```

```
        {
            Logger.Info("Операція завантаження файлів скасована.");
            statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Операція скасована"; });
        }
        catch (Exception ex)
        {
            Logger.Error($"Помилка при завантаженні файлів: {ex.Message}");
            MessageBox.Show("Помилка: " + ex.Message);
            statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Помилка"; });
        }
    }

    // Створення файлів або папок
    private async Task CreateNewFolderAsync()
    {
        string currentPath = treeView1.SelectedNode?.Tag.ToString();
        if (!string.IsNullOrEmpty(currentPath))
        {
            string newName = InputBox("Введіть назву нової папки або файлу:", "Новий об’єкт");
            if (!string.IsNullOrEmpty(newName))
            {
                char[] invalidChars = Path.GetInvalidPathChars();
                if (newName.Any(c => invalidChars.Contains(c)))
                {
                    MessageBox.Show("Нова назва містить недопустимі символи!", "Помилка",
MessageBoxButtons.OK, MessageBoxIcon.Error);
                    LogAction("CREATE_FAIL", currentPath, false, "Недопустиме ім’я");
                    return;
                }

                string newPath = Path.Combine(currentPath, newName);
                if (Directory.Exists(newPath) || File.Exists(newPath))
                {
                    MessageBox.Show("Об’єкт із такою назвою вже існує!", "Помилка",
MessageBoxButtons.OK, MessageBoxIcon.Error);
                    LogAction("CREATE_FAIL", newPath, false, "Об’єкт уже існує");
                    return;
                }

                DialogResult result = MessageBox.Show($"Створити {newName}?", "Підтвердження",
MessageBoxButtons.YesNo, MessageBoxIcon.Question);
                if (result == DialogResult.Yes)
                {
                    try
                    {
                        string[] supportedExtensions = { ".txt", ".docx", ".pdf", ".csv" };
                        bool isFile = supportedExtensions.Any(ext => newName.EndsWith(ext,
StringComparison.OrdinalIgnoreCase));
                        if (isFile)
                        {
                            using (File.Create(newPath)) { }
                            LogAction("CREATE_FILE", newPath, true);
```

```
                    }
                    else
                    {
                        Directory.CreateDirectory(newPath);
                        LogAction("CREATE_FOLDER", newPath, true);
                    }
                    await LoadDirectoryAsync(currentPath, cts.Token);
                }
                catch (OperationCanceledException)
                {
                    Logger.Info("Операція CreateNewFolderAsync скасована.");
                    statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Операція
скасована"; });
                }
                catch (Exception ex)
                {
                    LogAction("CREATE", newPath, false, ex.Message);
                    MessageBox.Show("Помилка: " + ex.Message);
                    statusStrip.Invoke((MethodInvoker)delegate { lblStatus.Text = "Помилка"; });
                }
            }
            else
            {
                LogAction("CREATE_CANCEL", currentPath, false, "Користувач скасував
створення");
            }
        }
        else
        {
            LogAction("CREATE_CANCEL", currentPath, false, "Користувач скасував
створення");
        }
    }

    // Логування дій
    private void LogAction(string action, string path, bool success, string message = "")
    {
        string status = success ? "OK" : "FAIL";
        string entry = $"{action}: \"{path}\" [{status}{(message != "" ? ": " + message : "")}]";
        if (success) Logger.Info(entry); else Logger.Error(entry);
    }
```