

Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: Розробка утиліти для безпечного очищення тимчасових файлів та кешу операційної системи

Засвідчую, що в цій
кваліфікаційній роботі немає
запозичень із праць інших
авторів без відповідних
посилань.

Студент гр. ІПЗ–22–1

_____ / Є. І. Двойніков /

Керівник кваліфікаційної
роботи

_____ / Н. О. Карабут /

В.о. завідувача кафедри

_____ / А. М. Стрюк /

Кривий Ріг

2026

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

В.о. зав. кафедри

_____ А. М. Стрюк

«_____» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ–22–1 Двойнікову Єгору Ігоровичу

1. Тема: «Розробка утиліти для безпечного очищення тимчасових файлів та кешу операційної системи» затверджена наказом по КНУ № 284с від «28» травня 2026 р.
2. Термін подання студентом закінченої роботи: «31» травня 2026 р.
3. Вихідні дані по роботі: розроблений програмний комплекс має забезпечити очищення тимчасових файлів та кешу операційної системи Windows.
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести аналіз існуючих на ринку аналогічних програмних продуктів, обґрунтувати функціонал розроблюваної системи, створити програмне забезпечення, здійснити тестування розробленого додатку.
5. Перелік ілюстративного матеріалу: функціональна схема, блок–схема алгоритму, зображення екранних форм додатку.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	03.01.26 – 24.01.26
2	Аналіз існуючих методів вирішення проблеми	25.01.26 – 07.02.26
3	Формулювання актуальності роботи і постановка завдань	08.02.26 – 18.02.26
4	Оформлення матеріалів першого розділу роботи	18.02.26 – 02.03.26
5	Створення функціональної системи та алгоритму додатку	03.03.26 – 15.03.26
6	Оформлення матеріалів другого розділу роботи	16.04.26 – 09.04.26
7	Розробка баз даних, інтерфейсу програмного забезпечення, програмних модулів	10.04.26 – 01.05.26
8	Оформлення додатків	02.05.26 – 07.05.26
9	Тестування розробленої програми	08.05.26 – 15.05.26
10	Оформлення пояснювальної записки	16.05.26 – 30.05.26

Дата видачі завдання: «02» січня 2026 р.

Студент: _____ / Є. І. Двойніков /

Керівник роботи: _____ / Н. О. Карабут /

РЕФЕРАТ

ТИМЧАСОВІ ФАЙЛИ, ОЧИЩЕННЯ ДИСКА, БЕЗПЕЧНЕ ВИДАЛЕННЯ, РЕЗЕРВНЕ КОПЮВАННЯ, КЕШ, C#, .NET, WINDOWS FORMS, JSON.

Пояснювальна записка: 63 с., 7 рис., 1 дод., 29 джерел.

Об'єкт дослідження: процес управління тимчасовими даними та залишковими файлами в середовищі операційної системи Windows.

Предмет дослідження: алгоритми безпечного очищення дискового простору, методи асинхронної обробки вводу-виводу (I/O) та засоби конструювання програмних утиліт мовою C#.

У роботі досліджено проблему накопичення залишкових даних під час роботи операційної системи та прикладного програмного забезпечення. Проведено огляд популярних аналогів і виявлено їхні недоліки, такі як закритий вихідний код, агресивні алгоритми втручання в реєстр та відсутність зручного механізму локальних бекапів.

Для вирішення цих проблем спроектовано власну програму. Архітектура додатку розділяє логіку взаємодії з диском та графічний інтерфейс користувача. Основну логіку реалізовано на платформі .NET. Графічну оболонку побудовано за допомогою Windows Forms.

У межах практичної частини написано модулі для рекурсивного сканування системних директорій та обробки виняткових ситуацій. Для запобігання втраті даних створено сервіс пакування обраних файлів у ZIP-архіви перед очищенням.

Отримана програма є інструментом, який дозволяє користувачам ефективно керувати вільним місцем на накопичувачі.

ABSTRACT

TEMPORARY FILES, DISK CLEANUP, SECURE DELETION, BACKUP, CACHE, C#, .NET, WINDOWS FORMS, JSON.

Explanatory note: 63 pages, 7 figures, 1 appendix, 29 sources.

Object of research: the process of managing temporary data and residual files in the Windows operating system environment.

Subject of research: algorithms for secure disk space cleanup, methods of asynchronous input/output (I/O) processing, and tools for constructing software utilities using the C# language.

The paper investigates the problem of residual data accumulation during the operation of the operating system and application software. A review of popular analogues was conducted, revealing their drawbacks, such as closed source code, aggressive registry intervention algorithms, and the lack of a convenient mechanism for local backups.

To solve these problems, a custom application was designed. The application architecture separates the disk interaction logic from the graphical user interface. The core logic is implemented on the .NET platform. The graphical shell is built using Windows Forms.

Within the practical part, modules for the recursive scanning of system directories and exception handling were written. To prevent data loss, a service was created to pack selected files into ZIP archives prior to cleanup.

The resulting program is a tool that allows users to effectively manage free space on their storage drives.

ЗМІСТ

ВСТУП.....	7
1 РОЗГЛЯД ПИТАННЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ	9
1.1. Особливості накопичення тимчасових файлів у сучасних операційних системах.....	9
1.2. Аналіз причин зниження продуктивності операційної системи через накопичення службових даних.....	10
1.3. Аналіз методів очищення тимчасових файлів та кешу	12
1.4. Огляд існуючих програмних рішень для очищення системи	14
1.5. Постановка задачі дипломної роботи.....	16
2 ПРОЕКТУВАННЯ УТИЛІТИ ДЛЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ.....	18
2.1 Формування функціональних вимог до програмного продукту.....	18
2.2 Формування нефункціональних вимог	19
2.3 Вибір технологій розробки	21
2.4 Проєктування архітектури програмного забезпечення.....	23
2.5 Проєктування структури класів та взаємодії компонентів	25
2.6 Проєктування інтерфейсу користувача.....	27
2.7 Проєктування користувацького інтерфейсу	29
2.8 Алгоритм роботи програми	30
3 РОЗРОБКА УТИЛІТИ ДЛЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ	34
3.1 Реалізація механізмів роботи утиліти	34
3.2 Розгляд роботи розробленої утиліти	38
ВИСНОВКИ	42
ПЕРЕЛІК ПОСИЛАНЬ	44
Додаток А - Програмний код.....	47

ВСТУП

Під час роботи операційної системи та різноманітного прикладного програмного забезпечення постійно генеруються тимчасові файли, лог-записи, фрагменти пам'яті та кешовані дані. За задумом розробників, такі файли мають автоматично видалятися після завершення відповідних процесів або сесій. Проте на практиці механізми самоочищення часто працюють некоректно через аварійні завершення програм, конфлікти доступу до файлів або помилки у логіці обробки винятків. Накопичення цих залишкових даних призводить до зменшення корисного дискового простору, а у випадку сильної фрагментації файлової системи може погіршувати загальну чуйність ОС.

Окремою проблемою, яка потребує вирішення, є аспект безпеки. Стандартні засоби видалення інформації в середовищі Windows здебільшого лише стирають посилання на файл у головній файловій таблиці (MFT), залишаючи самі байти даних на фізичному носії [1]. Це створює серйозний ризик несанкціонованого відновлення конфіденційної інформації, такої, як кеш браузерів, файли cookie або збережені сесії автентифікації. Тому виникає об'єктивна потреба у застосуванні алгоритмів безпечного видалення (secure deletion), які передбачають багаторазове перезаписування секторів диска випадковими даними або нулями для унеможливлення їх відновлення апаратними чи програмними методами.

Хоча існують популярні інструменти для очищення систем, багато з них містять надлишкову функціональність, мають закритий вихідний код, збирають телеметрію або агресивно вимагають переходу на платні версії. Розробка власної утиліти дозволяє реалізувати прозорий механізм керування файлами з можливістю гнучкого налаштування через JSON-конфігурації та забезпеченням високої продуктивності за рахунок використання асинхронного програмування, що є стандартом для сучасних платформ розробки [2].

Метою дипломної роботи є проектування та програмна реалізація утиліти для комплексного та безпечного очищення тимчасових файлів, кешу браузерів і системних директорій з підтримкою механізмів резервного копіювання.

Для досягнення поставленої мети були сформульовані такі основні завдання: проаналізувати методи роботи з файловою системою ОС Windows та алгоритми безпечного затирання даних; спроектувати модульну архітектуру додатку на основі багаторівневого підходу (layered architecture); реалізувати ядро логіки сканування, видалення та створення ZIP-архівів для бекапів з використанням пулу потоків та патерну `async/await` на базі платформи .NET; розробити зрозумілий графічний інтерфейс користувача засобами Windows Forms; протестувати стабільність програми, зокрема механізми обробки винятків (I/O exceptions) та систему логування.

Об'єкт дослідження: процес управління тимчасовими даними та залишковими файлами в середовищі операційної системи Windows.

Предмет дослідження: алгоритми безпечного очищення дискового простору, методи асинхронної обробки вводу-виводу (I/O) та засоби конструювання програмних утиліт мовою C#.

Практичне значення отриманих результатів полягає у створенні готового до використання програмного інструменту, який надає користувачам повний контроль над очищенням дискового простору, дозволяє автоматизувати рутинні завдання з обслуговування ОС та мінімізує ризики витоку персональних даних через відновлення видалених файлів.

1 РОЗГЛЯД ПИТАННЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ

1.1. Особливості накопичення тимчасових файлів у сучасних операційних системах

Функціонування операційної системи Windows та прикладного програмного забезпечення супроводжується безперервним створенням короточасних об'єктів файлової системи. Тимчасові файли виконують роль проміжних сховищ даних під час виконання складних обчислювальних операцій, інсталяції оновлень, розпакування архівів або обміну інформацією між процесами за допомогою іменованих каналів чи файлів-відображень пам'яті. Основними системними директоріями для зберігання таких об'єктів є каталоги `AppData\Local\Temp` у профілі кожного користувача та загальносистемна папка `Temp`, розташована в кореневому каталозі Windows [3]. Відповідно до стандартів розробки програмного забезпечення для платформи .NET, створення таких файлів зазвичай реалізується через методи класу `System.IO.Path`, зокрема `GetTempFileName`, який автоматично генерує унікальне ім'я файлу в тимчасовій директорії користувача [4].

Окремим класом залишкових даних є кеш-файли веб-браузерів та мережевих служб. Сучасні оглядачі інтернету, такі як Google Chrome або Microsoft Edge, використовують складні структури зберігання (файли без розширення, індекси та журнали типу `data_0`, `data_1`), розміщені в підкаталогах профілю користувача. Кешування призначене для прискорення завантаження веб-сторінок шляхом збереження статичних ресурсів (зображень, стилів CSS, скриптів JavaScript) на локальному диску. Проте через високу інтенсивність веб-серфінгу обсяг цих каталогів збільшується експоненційно, перетворюючись на неструктуроване сховище застарілих даних, які більше не використовуються браузером, але займають значне місце на накопичувачі.

Процес накопичення «сміттєвих» даних стає критичним через кілька чинників. По-перше, розробники прикладного софту не завжди реалізують коректне звільнення ресурсів у блоках `finally` або при обробці критичних винятків, через що файли залишаються заблокованими або просто забутими на диску. По-друге, інсталятори програм та підсистема Windows Update після завершення своєї роботи часто залишають розпаковані пакети та лог-файли в каталогах на кшталт `C:\Windows\SoftwareDistribution\Download` [5]. Накопичення мільйонів дрібних тимчасових файлів негативно впливає на продуктивність дискової підсистеми, особливо під час використання файлової системи NTFS. Велика кількість об'єктів усередині однієї директорії призводить до розростання індексів B-дерева в головній файловій таблиці (MFT), що збільшує час пошуку, зчитування та запису нових файлів [3]. Крім того, на твердотільних накопичувачах (SSD) велика кількість дрібних операцій запису та видалення прискорює знос комірок пам'яті (ресурс TBW) та змушує контролер частіше запускати процедуру збору сміття (Garbage Collection), що знижує швидкість роботи пристрою.

1.2. Аналіз причин зниження продуктивності операційної системи через накопичення службових даних

Коли обсяг залишкових файлів сягає критичних значень, виникає явище відчутної деградації швидкодії дискової підсистеми та операційної системи в цілому. Для класичних магнітних жорстких дисків це передусім виражається у фрагментації: дрібні файли кешу браузерів та тимчасові об'єкти розкидаються по різних кластерах, змушуючи механічну зчитувальну головку постійно переміщуватися. Проте для твердотільних накопичувачів проблема має дещо іншу природу. При заповненні дискового простору понад 80-90% різко падає ефективність алгоритмів вирівнювання зносу (Wear Leveling) та виконання команди TRIM. Контролеру SSD не вистачає вільних блоків для фонових маневрів, що призводить до зростання показника посилення запису

[6]. У результаті швидкість запису нових даних може впасти в кілька разів, а ресурс комірок пам'яті вичерпується швидше.

На рівні логічної структури файлової системи NTFS накопичення десятків тисяч файлів у директоріях Temp або Cache спричиняє гіпертрофоване розростання головної файлової таблиці (MFT). Кожен файл потребує окремого запису з метаданими. Коли програмам потрібно знайти певний ресурс або ОС намагається оновити індекси, витрачається значно більше часу на обхід структур В-дерева каталогів [7]. Саме тому базові інструменти управління файлами (наприклад, Провідник Windows) часто перестають реагувати на дії користувача під час відкриття переповнених папок. Цей фактор став ключовим при проектуванні розроблюваної утиліти очищення: щоб уникнути блокування головного потоку графічного інтерфейсу (UI thread) у Windows Forms під час обходу великих масивів смітєвих даних, ядро сканера базується на методах на кшталт Directory.EnumerateFiles із застосуванням патерну async/await та об'єктів Task. Такий підхід дозволяє виконувати ресурсомісткі операції вводу-виводу (I/O) у фонових потоках пулу потоків .NET, зберігаючи програму чуйною.

Крім того, нестача вільного дискового простору безпосередньо впливає на механізми віртуальної пам'яті. Файл підкачки (pagefile.sys) та файл гібернації (hiberfil.sys) потребують великих безперервних блоків для швидкого скидання сторінок оперативної пам'яті. Якщо диск фрагментований залишковими даними або заповнений під зав'язку, робота менеджера пам'яті Windows сповільнюється, що викликає затримки під час перемикання між активними додатками [6]. Регулярне очищення таких директорій не просто вивільняє гігабайти простору, але й повертає системі здатність ефективно керувати сторінковою пам'яттю.

Надлишкове кешування також породжує конфлікти доступу. Системні служби індексації (Windows Search) або антивірусні екрани постійно намагаються сканувати тисячі новостворених тимчасових файлів, створюючи фонове навантаження на центральний процесор. Під час очищення цих

каталогів часто виникають ситуації, коли певний файл тимчасово заблокований іншим процесом. При проєктуванні архітектури нашої утиліти цей нюанс враховано через реалізацію надійного механізму обробки винятків (зокрема, перехоплення `IOException` або `UnauthorizedAccessException`). Завдяки використанню багаторівневої архітектури (layered architecture), сервісний шар логіки ізольовано обробляє такі колізії, записує попередження через систему логування та продовжує обхід дерева каталогів без аварійного завершення всієї операції очищення [8].

1.3. Аналіз методів очищення тимчасових файлів та кешу

Вирішення проблеми накопичення залишкових даних передбачає використання різних підходів, кожен з яких має свої технічні переваги та архітектурні обмеження. Традиційно розглядають три основні напрямки керування дисковим простором: ручне адміністрування, використання вбудованих компонентів операційної системи та застосування спеціалізованих комерційних або відкритих програм.

Ручне очищення полягає у самостійному переході користувача до прихованих системних каталогів та прямому видаленні знайдених файлів за допомогою графічної оболонки ОС або командного рядка. Такий підхід забезпечує повний контроль над тим, що саме знищується, але має суттєві недоліки. Процес вимагає від користувача глибокого розуміння структури директорій Windows та значних витрат часу. Крім того, операція часто переривається системними повідомленнями про неможливість доступу, оскільки частина тимчасових файлів залишається заблокованою активними процесами. У розроблюваній програмі ця проблема вирішується програмним шляхом на рівні сервісного шару (layered architecture): під час обходу каталогів використовується обробка винятків (конструкції try-catch), яка перехоплює `IOException` та `UnauthorizedAccessException`. Це дозволяє утиліті тихо фіксувати заблоковані об'єкти в журналі через підсистему логування і продовжувати сканування без зупинки основного потоку виконання.

Штатні засоби операційної системи, такі як класична програма `cleanmgr.exe` або новіший фоновий модуль `Storage Sense`, забезпечують безпечний базовий рівень обслуговування [9]. Оскільки ці інструменти розроблені самими розробниками ОС, вони гарантовано не порушують цілісність системи і коректно взаємодіють з реєстром. Проте їхня функціональність є досить жорстко зашитою в код. Вони орієнтовані переважно на системні компоненти та продукти самої компанії Microsoft, тоді як глибокий аналіз специфічних форматів кешу популярних сторонніх браузерів на рушії Chromium часто залишається поза їхньою увагою [10]. Також вбудовані засоби не надають розробникам та досвідченим користувачам можливості створювати власні правила фільтрації. Для подолання цього обмеження архітектура розроблюваної утиліти базується на використанні JSON-конфігурацій. Завдяки простору імен `System.Text.Json` програма динамічно підвантажує моделі даних (models), які містять списки цільових шляхів, маски файлів для видалення та винятки, дозволяючи гнучко адаптувати процес під конкретні потреби без перекомпіляції коду.

Ринок стороннього програмного забезпечення пропонує потужні альтернативи, які містять сотні сценаріїв очищення для різноманітного прикладного софту. Однак багато з таких утиліт еволюціонували у важкі програмні комплекси. Вони часто перевантажені модулями моніторингу, які постійно працюють у фоновому режимі, збирають телеметрію та споживають оперативну пам'ять. Значна частина таких продуктів має закритий вихідний код, що ускладнює перевірку реальних алгоритмів обробки конфіденційних даних, таких як історія вебсерфінгу чи збережені сесії.

На основі проведеного порівняння формується обґрунтування розробки власного програмного рішення. Проектована утиліта має на меті зайняти нішу між обмеженими системними засобами та перевантаженими комерційними продуктами. Побудова ядра програми на базі платформи `.NET` із застосуванням сучасних можливостей мови `C# 10` дозволяє створити швидкий та прозорий інструмент. Для того, щоб інтенсивні операції з файловою

системою (простір імен System.IO) не призводили до «зависання» графічного інтерфейсу на базі Windows Forms, уся взаємодія з диском виноситься в окремі асинхронні задачі. Використання патерна `async/await` у комбінації з токенами скасування (`CancellationToken`) гарантує, що користувач може будь-якої миті безпечно перервати процес сканування або видалення без пошкодження файлової системи.

1.4. Огляд існуючих програмних рішень для очищення системи

Аналіз ринку програмного забезпечення демонструє наявність великої кількості утиліт для обслуговування дискового простору. Для визначення архітектурних та функціональних вимог до розроблюваного додатку необхідно розглянути найбільш популярні рішення: `CCleaner`, `BleachBit`, `Wise Disk Cleaner` та `Glary Utilities`. Порівняння базується на таких критеріях, як повнота функціональності, надійність алгоритмів безпечного видалення, наявність механізмів резервного копіювання, зручність графічного інтерфейсу та рівень ризику пошкодження системних файлів.

`CCleaner` є одним із найбільш розповсюджених комерційних інструментів у цій категорії [11]. Програма виконує глибоке сканування кешу десятків браузерів та тимчасових каталогів сторонніх додатків. З точки зору безпечного очищення `CCleaner` підтримує алгоритми перезапису Гутмана (`Gutmann`) та стандарти Міністерства оборони США (`DoD 5220.22-M`), що дозволяє ефективно знищувати дані без можливості їх відновлення [1]. Однак архітектура цього рішення останніми роками стала перевантаженою модулями активного моніторингу, які постійно працюють у фоновому режимі. Механізм резервного копіювання реалізований виключно для гілок реєстру; перед видаленням самих файлів програма не створює архіви-бекапи. Інтерфейс містить значну кількість рекламних інтеграцій. Окремо виділяється високий ризик дестабілізації ОС через агресивні алгоритми чищення реєстру, які часто порушують цілісність зв'язків між системними бібліотеками.

BleachBit виступає як альтернатива з відкритим вихідним кодом, що фокусується на приватності та пропонує потужні інструменти для затирання (shredding) файлів і очищення вільного простору на диску [12]. Її архітектура дозволяє користувачам додавати власні конфігурації у форматі XML. Графічний інтерфейс, побудований на базі кросплатформних бібліотек, є мінімалістичним, але візуально застарілим і гірше інтегрується з логікою управління вікнами порівняно з нативними рішеннями, на кшталт Windows Forms. Як і в CCleaner, розробники BleachBit не реалізували функцію попереднього пакування файлів у ZIP-архів для їх швидкого відновлення у разі помилкового видалення.

Wise Disk Cleaner та Glary Utilities функціонують як комплексні пакети оптимізації, що об'єднують десятки дрібних утиліт [13]. З технічної точки зору, ці програми забезпечують високу швидкість сканування, але часто позначають як непотрібні ті конфігураційні файли або лог-журнали, які можуть знадобитися для відлагодження роботи локальних серверів чи середовищ розробки. Їхні інтерфейси є досить складними через велику кількість вкладених меню, що підвищує ймовірність помилок користувача. Закритий вихідний код не дає змоги перевірити, як саме реалізовано обробку виняткових ситуацій (наприклад, `UnauthorizedAccessException`) під час роботи з системними каталогами.

Проведений огляд виявляє спільні архітектурні недоліки більшості існуючих рішень: відсутність зручного файлового бекапу перед очищенням, закритість коду комерційних продуктів та перевантаженість інтерфейсів непотрібними або ризиковими модулями (наприклад, модифікаторами реєстру). Це підтверджує практичну цінність розробки власної профільної утиліти. Проєктований додаток на платформі .NET 6 оминатиме втручання в реєстр Windows, зосередившись виключно на безпечній роботі з файловою системою (простір імен System.IO). Використання асинхронних операцій (патерн `async/await` та клас `Task`) дозволить зберегти чуйність нативного інтерфейсу Windows Forms під час інтенсивного вводу-виводу [8], а

налаштування правил сканування через JSON-файли (бібліотека `System.Text.Json`) забезпечить необхідну гнучкість, залишаючи код утиліти лаконічним і модульним.

1.5. Постановка задачі дипломної роботи

З огляду на результати аналізу наявних інструментів та виявлені недоліки в керуванні залишковими даними, виникає потреба у створенні власної вузькоспеціалізованої програми. Головним завданням дипломного проєкту є розробка утиліти для комплексного обслуговування файлової системи, яка поєднувала б прозорість роботи, безпеку та високу швидкість виконання операцій.

Для успішної реалізації проєкту необхідно виконати низку чітких функціональних та архітектурних вимог. З точки зору базової функціональності, програма повинна вміти сканувати системні тимчасові директорії, кошик та кеш популярних веб-браузерів. Правила сканування (шляхи до папок, маски файлів для ігнорування) не повинні бути жорстко зашиті в код. Їх необхідно винести у зовнішній конфігураційний файл. Використання формату JSON та засобів простору імен `System.Text.Json` дозволить гнучко налаштовувати утиліту без її перекомпіляції.

Критичною вимогою виступає реалізація механізму безпечного очищення. Замість звичайного стирання посилання на файл (як це робить стандартний метод `File.Delete`), сервісний шар утиліти має підтримувати перезаписування фізичних секторів диска випадковими байтами перед видаленням. Це гарантуватиме неможливість відновлення конфіденційних даних (наприклад, файлів сесій або збережених паролів у кеші). Водночас для захисту від випадкової втрати потрібної інформації проєктується підсистема резервного копіювання. Перед виконанням деструктивних дій програма має запакувати обрані цільові файли у ZIP-архів засобами бібліотеки `System.IO.Compression`, надаючи можливість швидкого відновлення.

Кожна дія користувача, а також результати сканування та помилки доступу мають обов'язково фіксуватися. Підсистема журналювання відповідатиме за збір статистики (обсяг звільненого місця, кількість видалених файлів) та запис перехоплених системних винятків у структурований текстовий лог.

Архітектура додатка має спиратися на багаторівневий підхід (layered architecture) [14]. Логіка взаємодії з диском (сервіси та хелпери) повинна бути суворо відокремлена від графічного відображення (елементів керування Windows Forms) та структур зберігання даних (моделей). Базовою платформою обрано середовище .NET 6 із застосуванням мови C# 10, що забезпечує нативну підтримку цільових операційних систем Windows 10 та Windows 11.

Щоб графічний інтерфейс програми залишався чуйним під час тривалих операцій вводу-виводу на повільних носіях, усі ресурсомісткі задачі необхідно проєктувати з урахуванням принципів асинхронності. Використання патерна `async/await` у зв'язці з пулом потоків (`Task`) дозволить винести сканування та очищення у фоновий режим. Додатково вимагається імплементація механізму `CancellationToken`, завдяки якому користувач зможе безпечно та миттєво перервати будь-який процес без ризику пошкодити файлову систему.

Сформульовані вимоги визначають технічні межі проєкту та стають основою для наступного етапу – проєктування архітектури програми та розробки її графічного інтерфейсу.

2 ПРОЕКТУВАННЯ УТИЛІТИ ДЛЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ

2.1 Формування функціональних вимог до програмного продукту

Перехід до етапу проектування утиліти вимагає формалізації її ключових можливостей, які визначатимуть структуру класів та взаємодію між компонентами. Відповідно до класичних принципів програмної інженерії, визначення функціональних вимог є базисом для побудови надійної багаторівневої архітектури (layered architecture) [15]. Головним завданням розроблюваного додатку є надання користувачеві повного та безпечного контролю над процесом вивільнення дискового простору.

Первинною функцією системи є сканування. Програма повинна вміти аналізувати задані системні директорії (наприклад, локальні та загальні папки Temp), вміст кошика ОС, а також специфічні каталоги кешу популярних браузерів. Щоб уникнути жорсткого кодування (hardcoding) шляхів, висувається вимога до реалізації конфігураційної підсистеми. Налаштування цільових папок, масок файлів, що підлягають видаленню, та папок-винятків мають зберігатися у зовнішньому JSON-файлі. Під час запуску програми ці дані десеріалізуються засобами простору імен System.Text.Json у відповідні C#-моделі (models), які потім використовуються сервісним шаром для обходу файлової системи. Саме сканування має виконуватися асинхронно, щоб не блокувати головний потік виконання Windows Forms.

Логічним продовженням сканування є функція очищення. Вимагається реалізація не лише стандартного видалення через звернення до API операційної системи, але й механізму безпечного затирання інформації. Перед видаленням посилання на файл із таблиці MFT, утиліта має перезаписати його фізичні байти на диску нулями або випадковими даними, роблячи

неможливим подальше відновлення конфіденційних відомостей спеціалізованим програмним забезпеченням.

Для запобігання втраті критично важливої інформації через помилкові дії користувача, обов'язковою вимогою є наявність модулів резервного копіювання та відновлення. Перед ініціалізацією процесу очищення програма повинна формувати ZIP-архів із файлами, які плануються до видалення. Ця функціональність спиратиметься на можливості вбудованої бібліотеки `System.IO.Compression` [16]. Відповідно, модуль відновлення повинен вміти розпаковувати такі архіви, повертаючи файли за їхніми оригінальними шляхами з дотриманням попередньої ієрархії каталогів.

Кожна виконана операція має супроводжуватися записом у журнал. Підсистема логування повинна фіксувати час початку та завершення задач, обсяг звільненого місця, кількість оброблених файлів, а також деталі перехоплених винятків. Оскільки під час роботи з системними папками неминуче виникатимуть помилки доступу, коректна обробка винятків (наприклад, `UnauthorizedAccessException`) та їх запис у лог є критично важливими для стабільної роботи додатку без аварійних завершень.

Усі описані функції повинні керуватися через зрозумілий графічний інтерфейс. Оскільки операції з диском (I/O) можуть тривати значний час, інтерфейс має залишатися чуйним, відображати поточний прогрес та надавати можливість скасування операції. Це вимагає використання патерну `async/await`, об'єктів `Task` та `CancellationToken` на рівні зв'язку між UI-компонентами та сервісами бізнес-логіки.

2.2 Формування нефункціональних вимог

Поряд із функціональними можливостями, критичне значення для успішної реалізації проєкту мають нефункціональні вимоги, які визначають атрибути якості програмного забезпечення [17]. Оскільки утиліта призначена для інтенсивної взаємодії з файловою системою, першочерговим завданням є забезпечення високої швидкодії. Аналіз дискового простору, особливо на

жорстких дисках (HDD) або сильно фрагментованих SSD, є операцією з високими затримками вводу-виводу (I/O bound). Для оптимізації споживання оперативної пам'яті під час сканування каталогів із десятками тисяч об'єктів висувається вимога використовувати метод `Directory.EnumerateFiles` замість традиційного `GetFiles`. Це дозволить ітерувати колекцію файлів ліниво (lazy evaluation), не завантажуючи весь масив шляхів у пам'ять одночасно [18].

Надійність та стабільність роботи утиліти безпосередньо залежать від її здатності функціонувати в умовах часткового доступу до даних. Системні директорії Windows постійно використовуються фоновими службами, антивірусами та іншими додатками, які можуть встановлювати ексклюзивні блокування на окремі файли. Відповідно, програма не повинна аварійно завершувати роботу під час спроби прочитання або видалення такого об'єкта. Архітектура сервісного шару має передбачати точкову обробку винятків (наприклад, `IOException` або `UnauthorizedAccessException`) у циклах обходу дерева каталогів. Замість переривання процесу, система повинна фіксувати факт блокування файлу через клас-хелпер логування та продовжувати виконання наступних ітерацій.

Вимоги до безпеки охоплюють два аспекти: захист конфіденційності користувача та збереження цілісності операційної системи. Для першого аспекту реалізується алгоритм багаторазового перезапису тіла файлу випадковими байтами перед його фактичним видаленням з таблиці MFT. Для другого – утиліта має чітко розмежовувати права доступу. Програма виконуватиметься в контексті прав поточного користувача, що автоматично унеможливить випадкове видалення критичних системних файлів у каталозі `C:\Windows\System32`, доступ до яких вимагає підвищених привілеїв (`TrustedInstaller` або `Administrator`).

Зручність інтерфейсу забезпечується використанням графічної оболонки на базі Windows Forms. Головна вимога до UI – збереження чуйності під час виконання тривалих операцій. З огляду на однопотокову природу графічного інтерфейсу, будь-яке синхронне блокування призведе до зависання

вікна програми [8]. Тому всі процеси сканування, резервного копіювання (ZIP-архівації) та очищення повинні бути інкапсульовані в об'єкти Task та викликатися за допомогою ключових слів `async/await`. Інтерфейс також має підтримувати двосторонній зв'язок із фоновими задачами: відображати поточний прогрес обробки файлів та надавати користувачеві можливість дострокового скасування операції.

Масштабованість додатку забезпечується його модульною структурою. Використання багаторівневої архітектури (layered architecture) гарантує, що логіка взаємодії з диском відокремлена від візуальних компонентів [17]. Додатково висувається вимога до зберігання правил очищення (шляхів до кешу нових браузерів або програм) у зовнішніх конфігураційних JSON-файлах. Це дозволить в майбутньому додавати підтримку нових програмних продуктів простим редагуванням текстового файлу, без необхідності вносити зміни у вихідний C#-код моделей чи перекомпілювати весь проєкт.

2.3 Вибір технологій розробки

Основною мовою програмування для реалізації логіки утиліти обрано C# версії 10, а базовою платформою – .NET 6. Цей вибір зумовлений тим, що шоста версія фреймворку має статус довгострокової підтримки (LTS) та містить суттєві оптимізації продуктивності базових класів та збирача сміття (Garbage Collector). Для програми, яка потенційно опрацьовуватиме сотні тисяч дрібних тимчасових об'єктів у пам'яті під час побудови дерева каталогів, ефективне управління ресурсами є критичним. Синтаксичні можливості C# 10 дозволяють писати лаконічний код сервісного шару, використовуючи сучасні конструкції на кшталт записів (records) для передавання проміжних даних між модулями.

Ядро програми, що відповідає за взаємодію з накопичувачами, побудовано на базі простору імен System.IO [19]. Використання класів `DirectoryInfo`, `FileInfo` та `FileStream` надає прямий доступ до метаданих файлової системи NTFS. Це особливо важливо для реалізації алгоритму

безпечного видалення. Замість простого виклику функції стирання, програма відкриває файловий потік, перезаписує фізичні байти випадковою послідовністю, і лише після цього видаляє посилання на файл. Такий підхід вимагає низькорівневого контролю над операціями вводу-виводу, який забезпечується саме інструментарієм System.IO.

Оскільки операції читання та запису на диск мають високу латентність і є класичними I/O-bound задачами, їхнє виконання у синхронному режимі призвело б до повного блокування графічного інтерфейсу. Щоб уникнути цього, було застосовано патерн `async/await` у комбінації з класом `Task`. Усі методи сканування, створення ZIP-архівів для бекапів та безпосереднього очищення проєктуються як асинхронні. Вони виконуються у фонових потоках пулу (Thread Pool), повертаючи контроль системі, доки триває фізична робота диска. Для забезпечення керованості процесом у всіх асинхронних сервісах передається об'єкт `CancellationToken`, що дозволяє користувачеві миттєво перервати обхід каталогів.

Для зберігання конфігурації утиліти – переліку цільових директорій, розширень файлів-винятків та налаштувань інтерфейсу – обрано формат JSON. На відміну від жорсткого кодування (`hardcoding`) або використання системного реєстру, цей формат дозволяє гнучко змінювати параметри сканування без перекомпіляції проєкту. Обробка цих даних реалізується через високопродуктивну бібліотеку `System.Text.Json` [20]. Під час ініціалізації програми конфігураційні файли автоматично десеріалізуються у відповідні C#-моделі, які потім ін'єктуються в класи бізнес-логіки.

Розробка графічного інтерфейсу користувача виконана за допомогою технології Windows Forms. Хоча для .NET існують складніші фреймворки (WPF або WinUI), WinForms залишається оптимальним вибором для створення утиліт системного адміністрування. Ця технологія потребує мінімум оперативної пам'яті для рендерингу вікон, гарантує швидкий старт додатку та забезпечує звичний нативний вигляд у середовищі Windows. Завдяки дотриманню принципів багаторівневої архітектури, візуальні

елементи форми (кнопки, прогрес-бари, списки) виступають лише тонким шаром відображення, який отримує дані від ізольованих сервісів логіки.

2.4 Проєктування архітектури програмного забезпечення

Для забезпечення масштабованості, зручності тестування та підтримки програмного коду, архітектура розроблюваної утиліти базується на класичному багаторівневому підході. Такий шаблон проєктування дозволяє ізолювати логіку взаємодії з файловою системою від графічного інтерфейсу, що є важливим для коректної реалізації асинхронних операцій у середовищі .NET 6. Структура додатка розділена на п'ять основних логічних рівнів: шар представлення (UI Layer), шар бізнес-логіки (Services), шар даних (Models), допоміжні компоненти (Helpers) та модуль конфігурації (Config).

UI Layer (Шар представлення). Побудований з використанням технології Windows Forms. Його єдиною відповідальністю є взаємодія з користувачем: відображення елементів керування, отримання вхідних параметрів, рендеринг списку знайдених файлів та візуалізація прогресу виконання задач (через компонент ProgressBar). Класи форм (наприклад, MainForm) не містять жодної логіки сканування або видалення. Вони лише делегують виклики відповідним сервісам за допомогою патерну `async/await` та передають об'єкти `CancellationTokenSource`, щоб користувач міг безпечно перервати тривалі дискові операції без зависання графічної оболонки.

Services (Шар бізнес-логіки). Ядро програми, яке інкапсулює алгоритми роботи з операційною системою. Він складається з незалежних класів:

а) `ScannerService` – відповідає за рекурсивний обхід директорій за допомогою `Directory.EnumerateFiles`, фільтрацію тимчасових файлів та кешу браузерів.

б) `CleanerService` – реалізує механізми звичайного та безпечного видалення. Саме тут розгорнуто логіку перезаписування фізичних байтів файлу нулями перед зверненням до стандартного методу `File.Delete`.

в) BackupService – відповідає за створення резервних копій у форматі ZIP перед ініціалізацією процесу очищення.

Ці сервіси є ізольованими, що дозволяє легко покривати їх модульними тестами та модифікувати без впливу на інтерфейс користувача [17].

Models (Шар даних). Містить класи та структури (DTO – Data Transfer Objects), які використовуються для передачі інформації між різними рівнями додатку. До цього шару належать такі сутності, як FileItem (зберігає шлях, розмір та дату останнього доступу до знайденого об'єкта) та ScanSummary (містить загальну статистику для відображення в UI). Використання нововведень мови C# 10, зокрема record class, дозволяє створювати незмінні (immutable) моделі з мінімальною кількістю шаблонного коду.

Helpers (Допоміжні компоненти). Набір статичних класів або утиліт, які вирішують крос-функціональні завдання. Найважливішим компонентом цього рівня є LoggerHelper. Він викликається з шару сервісів під час перехоплення виняткових ситуацій (наприклад, IOException під час спроби доступу до заблокованого системного файлу) і забезпечує безпечний запис помилок у текстовий журнал без переривання загального процесу сканування.

Config (Модуль конфігурації). Відповідає за збереження та завантаження налаштувань програми (шляхи до користувацьких директорій, маски винятків, налаштування безпечного видалення). Замість використання реєстру Windows, конфігурація зберігається у зовнішньому файлі settings.json. Модуль ConfigManager використовує методи простору імен System.Text.Json для швидкої десеріалізації цих даних в об'єкти конфігурації під час старту програми [21].

Графічне представлення багаторівневої архітектури додатку та напрямки залежностей між компонентами наведено на UML-діаграмі нижче.

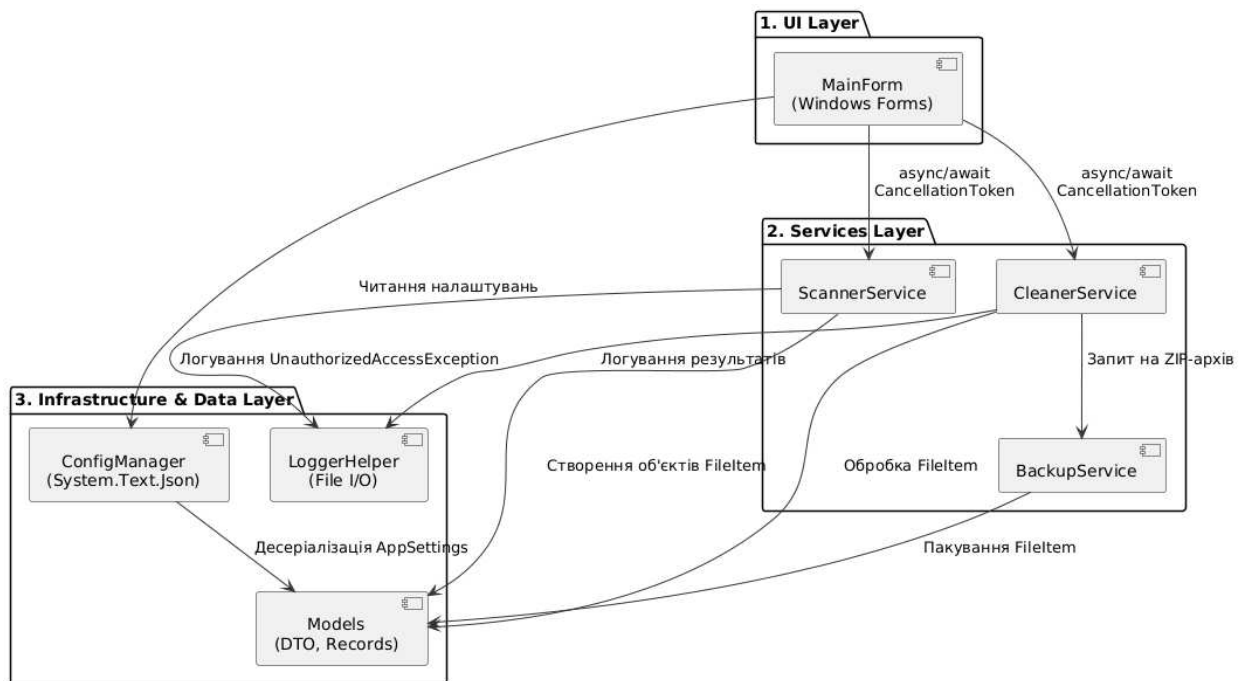


Рисунок 2.1 – Функціональна схема архітектури додатку

Запропонована структура гарантує, що залежності завжди спрямовані від верхніх рівнів (UI) до нижніх (Services та Infrastructure), що повністю відповідає принципам інверсії залежностей та полегшує подальший розвиток програмного продукту.

2.5 Проєктування структури класів та взаємодії компонентів

Для реалізації визначених вище вимог необхідно деталізувати внутрішню структуру програми. Архітектура утиліти базується на розподілі обов'язків між класами, що дозволяє уникнути надмірної зв'язності коду (tight coupling) та спростити подальше масштабування чи тестування окремих модулів [22].

Точкою входу для користувача є клас MainForm, який успадковується від базового класу System.Windows.Forms.Form. Його головна задача – обробка подій інтерфейсу та відображення результатів. У цьому класі не міститься жодної логіки сканування або видалення. Натомість обробники подій (наприклад, натискання кнопки початку сканування) позначені

модифікатором `async` і викликають відповідні методи сервісів через `await`. Для забезпечення можливості зупинки тривалих процесів у `MainForm` ініціалізується об'єкт `CancellationTokenSource`, токен якого передається в усі асинхронні виклики підлеглих класів.

Центральне місце в логіці програми займає `ScanService`. Цей клас відповідає за обхід системних каталогів та пошук тимчасових файлів. Для мінімізації споживання оперативної пам'яті під час роботи з великими деревами папок, метод сканування використовує інтерфейс `IAsyncEnumerable<T>`, що дозволяє асинхронно повертати об'єкти типу `FileItem` у міру їх знаходження у файловій системі [23]. Цей сервіс тісно взаємодіє з `SettingsService`, отримуючи від нього актуальні маски розширень та списки директорій-винятків перед початком пошуку.

За безпосереднє знищення даних відповідає `CleanupService`. Він отримує сформований сканером список файлів і виконує їх видалення. Залежно від обраних налаштувань, сервіс може викликати стандартний метод операційної системи або активувати алгоритм безпечного затирання. У другому випадку клас створює об'єкт `FileStream` для кожного файлу і циклічно перезаписує його фізичний вміст випадковим масивом байтів перед видаленням посилання з MFT. Усі операції вводу-виводу всередині `CleanupService` інкапсульовані в `Task.Run`, щоб гарантовано не блокувати головний потік графічного інтерфейсу.

Перед початком роботи `CleanupService` обов'язково перевіряється конфігурація щодо необхідності створення точок відновлення. Якщо ця опція активна, управління передається до `BackupService`. Цей клас використовує клас `ZipFile` з простору імен `System.IO.Compression` для створення архіву, куди копіюються всі знайдені файли зі збереженням їхньої оригінальної структури каталогів. Після успішного завершення архівації сервіс повертає сигнал про готовність до процесу очищення.

Керування конфігурацією покладено на `SettingsService`. Він відповідає за читання та запис файлу `settings.json`. Використовуючи високопродуктивні

методи `JsonSerializer.Deserialize` та `JsonSerializer.Serialize`, сервіс перетворює текстові дані у внутрішні моделі конфігурації. Цей клас доцільно реалізувати за патерном `Singleton`, щоб усі інші компоненти мали доступ до єдиного актуального екземпляра налаштувань під час усього сеансу роботи утиліти [24].

Усі описані вище класи під час своєї роботи можуть стикатися з винятковими ситуаціями, особливо при спробі доступу до системних ресурсів або заблокованих кеш-файлів браузерів. Для реєстрації таких подій використовується `LogService`. Він приймає повідомлення про помилки (наприклад, перехоплені об'єкти `UnauthorizedAccessException`) та інформаційні статуси, після чого асинхронно дописує їх у текстовий журнал. Щоб уникнути конфліктів спільного доступу до лог-файлу з різних фонових потоків, клас використовує конкурентні черги (`ConcurrentQueue`) або механізми синхронізації, що гарантують цілісність усіх записів.

2.6 Проєктування інтерфейсу користувача

Вибір технології для побудови графічного інтерфейсу (UI) базувався на специфіці розроблюваної утиліти. Оскільки програма призначена для виконання системних завдань і має працювати з мінімальним споживанням оперативної пам'яті, було прийнято рішення використати `Windows Forms` на базі платформи `.NET 6`. Хоча існують більш сучасні фреймворки, такі як `WPF` або `WinUI`, `Windows Forms` забезпечує найшвидший «холодний старт» додатку, має нативний вигляд у середовищі `Windows 10/11` і не перевантажує систему зайвим рендерингом [25]. Згідно з розробленою багаторівневою архітектурою, шар UI виступає лише тонкою оболонкою, яка делегує всю важку роботу фоновим сервісам і не містить складної бізнес-логіки.

Для організації простору головного вікна (`MainForm`) використовується компонент `TableLayoutPanel`. Відмова від абсолютного позиціонування елементів за координатами (x, y) є важливою практикою при створенні надійних інтерфейсів. Цей контейнер дозволяє розбити форму на динамічну

сітку рядків та стовпців. Завдяки цьому при масштабуванні вікна користувачем або зміні налаштувань DPI в операційній системі, всі вкладені елементи керування автоматично змінюють свої розміри та зберігають правильні пропорції, запобігаючи обрізанню тексту чи накладанню кнопок [26].

Взаємодія з користувачем реалізується через базові елементи керування, ключовим з яких є `Button`. Кнопки відповідають за ініціалізацію процесів сканування, створення резервних копій та очищення. Особливістю їхньої реалізації є те, що обробники подій (наприклад, `btnScan_Click`) позначені ключовим словом `async`. Під час виконання тривалої операції вводу-виводу кнопка блокується для запобігання повторним натисканням, а поруч активується кнопка скасування, яка викликає метод `Cancel()` у переданому об'єкті `CancellationTokenSource`. Для вибору об'єктів сканування використовується `CheckedListBox`. Цей компонент ідеально підходить для відображення списку доступних модулів очищення (наприклад, «Тимчасові файли Windows», «Кеш Google Chrome», «Кошик»), які динамічно завантажуються модулем конфігурації з файлу `settings.json`. Користувач може гнучко позначати галочками лише ті категорії, які потребують обробки.

Окрему увагу приділено візуалізації фонових процесів. Оскільки обхід дерева каталогів та безпечне затирання файлів можуть тривати від кількох секунд до десятків хвилин, інтерфейс має постійно інформувати про стан виконання. Для цього використовується `ProgressBar`. Він оновлюється не безпосередньо з робочих потоків (що викликало б виняток `CrossThreadMessagingException`), а через безпечний механізм `IProgress<T>`, який гарантує виконання методу оновлення в головному потоці (UI thread). Це дозволяє плавно відображати відсоток завершення операції на основі даних, отриманих від `ScannerService` або `CleanerService`.

Для відображення журналу операцій обрано компонент `RichTextBox`. На відміну від стандартного `TextBox`, він підтримує форматування тексту. Це дозволяє модулю логування (`LogService`) виводити інформацію з кольоровим

кодуванням: успішно видалені файли позначаються зеленим кольором, звичайні інформаційні повідомлення – чорним, а перехоплені виняткові ситуації (наприклад, неможливість доступу до системного файлу через `UnauthorizedAccessException`) виділяються червоним. Такий підхід значно покращує сприйняття технічної інформації та дозволяє користувачеві швидко ідентифікувати проблемні ділянки без необхідності відкривати зовнішні текстові лог-файли.

2.7 Проектування користувацького інтерфейсу

Реалізація процесів видалення даних вимагає підвищеної уваги до питань безпеки, оскільки помилкові дії утиліти можуть призвести до порушення працездатності операційної системи або втрати важливої користувацької інформації. Проектування модуля очищення (`CleanerService`) базується на концепції «захисного програмування» (*defensive programming*), яка передбачає обов'язкову перевірку стану ресурсів перед виконанням будь-якої деструктивної операції [27].

Першим етапом безпечного очищення є ідентифікація прав доступу. Перед спробою видалення або модифікації кожного файлу, сервіс перевіряє його атрибути (`FileAttributes`). Якщо файл має атрибут `ReadOnly` або `System`, програма автоматично пропускає його, генеруючи відповідне інформаційне повідомлення у лог. Крім того, утиліта працює в контексті поточного користувача без запиту підвищених привілеїв адміністратора, що на рівні операційної системи (UAC) блокує доступ до критичних системних директорій (наприклад, `C:\Windows\System32`) [28].

Обробка виняткових ситуацій є центральним елементом стабільності `CleanerService`. У середовищі Windows багато тимчасових файлів можуть бути заблоковані активними процесами (наприклад, антивірусними сканерами або самими браузером, які наразі відкриті). Спроба видалити такий файл неминуче призводить до винятку `IOException`. Архітектура передбачає обгортання кожної операції `File.Delete` у блок `try-catch`, який ізолює помилку.

Замість аварійного завершення всього процесу сканування, сервіс перехоплює винятки, передає деталі помилки до LogService і переходить до обробки наступного файлу в черзі.

Для забезпечення можливості відкату дій користувача реалізовано підсистему резервного копіювання. Перед початком очищення BackupService формує ZIP-архів цільових файлів. Важливою архітектурною вимогою є те, що цей процес відбувається транзакційно: лише якщо створення архіву завершилося успішно і без помилок доступу, CleanerService отримує сигнал на виконання фактичного видалення. Це гарантує, що в разі непередбаченого збою живлення або апаратного відключення користувач не втратить дані безповоротно.

Захист від випадкового видалення корисних даних забезпечується механізмом винятків (Exceptions/Exclusions). У модулі конфігурації settings.json передбачено масиви регулярних виразів та точних шляхів, які повністю ігноруються під час сканування (наприклад, файли з розширенням .ini або директорії на кшталт C:\Users\Username\AppData\Local\Microsoft\Windows\INetCache\Low). Цей список формується на основі офіційних рекомендацій розробників операційної системи та популярних програмних продуктів.

2.8 Алгоритм роботи програми

Логіка функціонування розроблюваної утиліти базується на чіткій послідовності станів, які контролюються асинхронними подіями графічного інтерфейсу та внутрішніми сервісами. Процес виконання програми можна умовно розділити на етапи ініціалізації, підготовки даних, фонові обробки та фіналізації [30]. Такий лінійний, але керований подіями підхід гарантує, що користувач завжди розуміє поточний стан системи і має можливість втрутитися в її роботу.

На етапі запуску програми ініціалізується головне вікно MainForm.

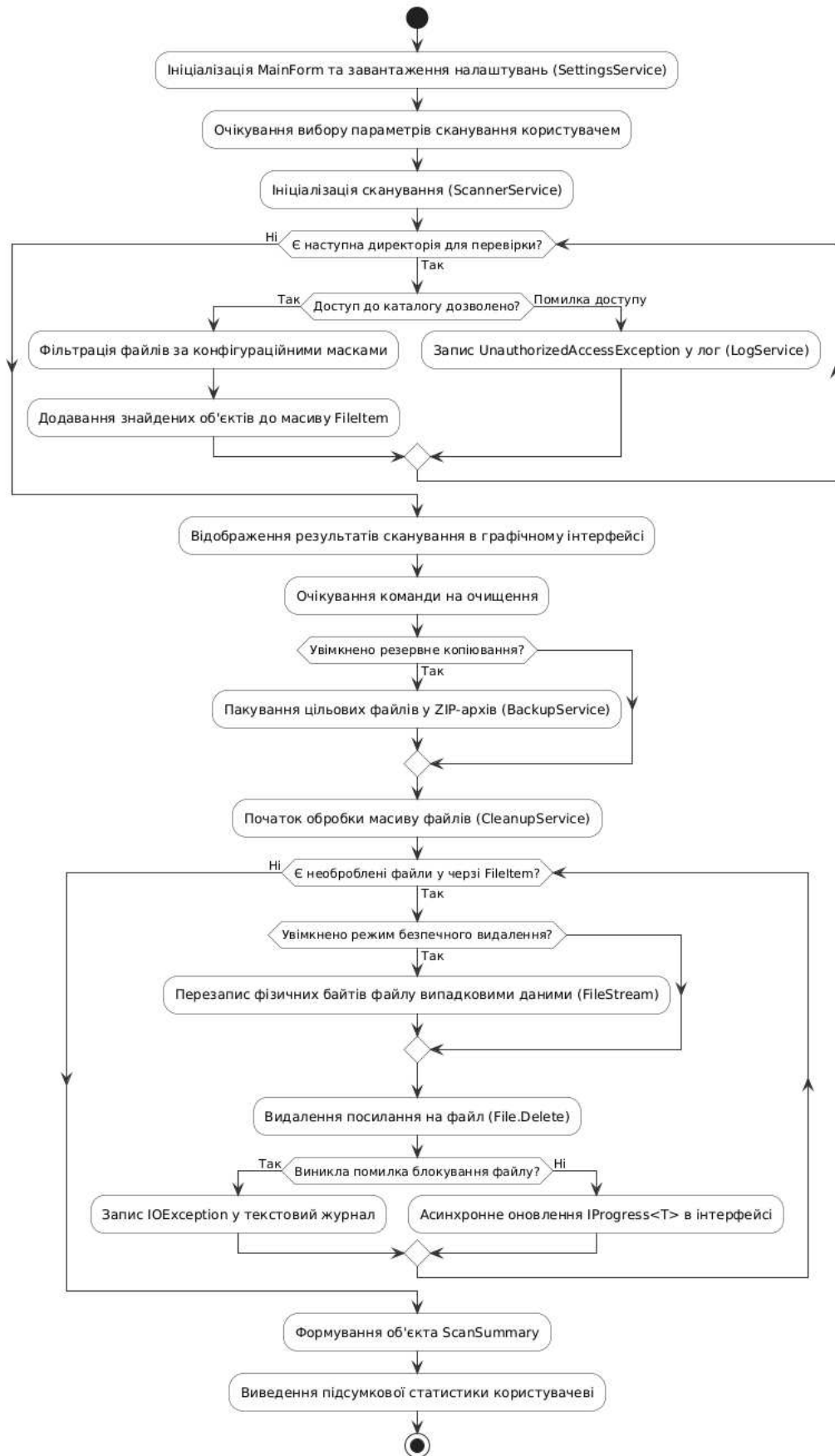


Рисунок 2.2 – Алгоритм роботи утиліти

Одразу після побудови візуальних компонентів управління передається до `SettingsService`, який зчитує конфігураційний файл `settings.json`. Використовуючи методи `System.Text.Json`, сервіс перетворює текстовий вміст у колекції об'єктів, що містять шляхи до цільових директорій, перелік розширень файлів для ігнорування та збережені користувацькі налаштування інтерфейсу. Якщо конфігураційний файл відсутній або пошкоджений, система автоматично генерує його з параметрами за замовчуванням. Після успішного завантаження налаштувань програма переходить у стан очікування дій користувача, блокуючи кнопки очищення до моменту завершення першого сканування.

Коли користувач ініціює пошук тимчасових файлів, спрацьовує асинхронний обробник події. На цьому етапі створюється новий об'єкт `CancellationTokenSource`, який передається у `ScannerService`. Алгоритм сканування реалізує рекурсивний обхід вибраних гілок файлової системи за допомогою методу `Directory.EnumerateFiles`. Під час обходу кожен знайдений файл перевіряється на відповідність критеріям (відсутність системних атрибутів, збіг із дозволеними шляхами). Якщо під час читання директорії виникає виняток `UnauthorizedAccessException`, алгоритм перехоплює його, реєструє подію через `LogService` і переходить до наступного каталогу. Інформація про знайдені файли упаковується в об'єкти `FileItem` і частинами повертається до UI-потoku через інтерфейс `IProgress<T>`, що забезпечує плавне оновлення смуги прогресу.

Після завершення сканування утиліта очікує команду на очищення. Якщо користувач підтверджує дію, алгоритм перевіряє стан прапорця резервного копіювання. За умови його активності, робочий потік передає масив шляхів до `BackupService`, який послідовно зчитує файли та пакує їх у єдиний ZIP-архів. Лише після перевірки цілісності створеного архіву система переходить до фази деструктивних дій.

Сам алгоритм видалення в класі `CleanupService` працює ітеративно. Для кожного об'єкта `FileItem` із черги спочатку знімається атрибут `ReadOnly` (якщо

це дозволено налаштуваннями). Якщо користувач обрав режим безпечного видалення, програма відкриває файл через FileStream і перезаписує його вміст випадковими даними. Після цього викликається стандартна команда File.Delete. Кожна успішна або неуспішна операція фіксується. Після завершення обробки всього масиву файлів алгоритм формує підсумковий звіт ScanSummary, виводить кількісні показники звільненого простору в RichTextBox та повертає графічний інтерфейс у початковий стан готовності до нового циклу роботи.

Завдяки такій структурі алгоритму забезпечується висока надійність виконання операцій. Відокремлення етапів читання, бекапу та видалення мінімізує ризик незворотної втрати даних у випадку непередбаченого переривання роботи програми на проміжних етапах.

3 РОЗРОБКА УТИЛІТИ ДЛЯ БЕЗПЕЧНОГО ОЧИЩЕННЯ ТИМЧАСОВИХ ФАЙЛІВ ТА КЕШУ ОПЕРАЦІЙНОЇ СИСТЕМИ

3.1 Реалізація механізмів роботи утиліти

Основою функціонування утиліти є модуль сканування, який відповідає за виявлення цільових об'єктів на накопичувачі. Програмна реалізація цього механізму базується на використанні статичних класів `Directory` та `File` з простору імен `System.IO`. Для забезпечення ефективного використання оперативної пам'яті під час обробки тисяч дрібних елементів кешу застосовано метод `Directory.EnumerateFiles`, який повертає ітератор колекції замість завантаження всього масиву шляхів у пам'ять одночасно.

Стандартний підхід із використанням параметра `SearchOption.AllDirectories` виявився архітектурно слабким, оскільки при зіткненні з першим же захищеним системним каталогом метод генерує критичний виняток і повністю припиняє обхід дерева. Для вирішення цієї проблеми було спроектовано власний алгоритм рекурсивного пошуку. Сервіс сканування ізольовано аналізує кожен директорію: спочатку ітерує файли поточного рівня, перевіряючи їхню відповідність конфігураційним маскам, а потім рекурсивно викликає сам себе для всіх вкладених папок, отриманих через `Directory.EnumerateDirectories`. Кожен такий виклик обгорнуто в конструкцію `try-catch`, що дозволяє безпечно оминати системні теки без переривання загального процесу аналізу файлової системи.

Взаємодія з дисковою підсистемою супроводжується неминучими затримками вводу-виводу, тому забезпечення постійної чуйності графічної оболонки стало одним із головних завдань. Увесь життєвий цикл операцій сканування та очищення побудовано на базі сучасного патерну `async/await`. Ресурсомісткі процеси делегуються пулу потоків за допомогою методу `Task.Run`. Для управління життєвим циклом фонових задач імплементовано

механізм скасування через `CancellationToken`. Під час запуску сканування в головній формі створюється об'єкт `CancellationTokenSource`, токен якого прокидається крізь усі рівні абстракції (layered architecture) до кінцевих методів роботи з диском. Перевірка властивості `IsCancellationRequested` всередині рекурсивних циклів гарантує безпечну зупинку пошуку за першою вимогою користувача. Проблема трансляції результатів із фонових потоків у `Windows Forms` вирішена завдяки інтерфейсу `IProgress<T>`. Клас-сервіс періодично викликає метод `Report`, передаючи проміжні метрики обробки файлів, а платформа `.NET` автоматично синхронізує цей виклик із контекстом головного потоку, що усуває необхідність використання громіздких конструкцій на кшталт `Control.Invoke`.

Логіка видалення залишкових даних інкапсульована у класі `CleanerService`. Процес ініціюється етапом фільтрації: сервіс зчитує вибрані користувачем категорії (наприклад, локальні тимчасові файли або кеш веб-браузерів) та формує фінальну чергу об'єктів для знищення. Перед зверненням до файлової системи обов'язково виконується перевірка прав доступу шляхом аналізу властивості `FileAttributes`. Якщо файл позначений атрибутом `ReadOnly`, програма робить спробу програмно зняти його, але файли з атрибутом `System` жорстко ігноруються алгоритмом заради збереження цілісності операційної системи.

Фактичне знищення об'єктів залежить від обраного рівня безпеки. У стандартному режимі застосовується базовий метод `File.Delete`. У режимі безпечного затирання сервіс відкриває `FileStream`, який багаторазово перезаписує фізичні межі файлу масивом криптографічно випадкових байтів перед остаточним видаленням метаданих з таблиці MFT. Кожна операція обгорнута в ізольований блок обробки винятків для нейтралізації переривань.

Для нівелювання ризиків випадкової втрати корисної інформації спроектовано автономний модуль `BackupService`. Його функціонування повністю спирається на інструментарій простору імен `System.IO.Compression`. Перед активацією деструктивних алгоритмів очищення, сервіс створює

цілісний ZIP-архів, назва якого містить мітку часу операції. Для оптимізації продуктивності та уникнення подвійного копіювання файлів на диску, пакування реалізовано через пряму роботу з об'єктом `ZipArchive`. Програма ітерує чергу цільових файлів, створює для кожного віртуальний запис та трансліює байти з вихідного `FileStream` безпосередньо у стиснений потік архіву. Відновлення даних винесено в окремий візуальний модуль `RestoreForm`. Цей компонент зчитує системний каталог бекапів утиліти, відображає перелік доступних точок відновлення та дозволяє ініціювати декомпресію. Логіка розпакування автоматично відтворює абсолютні шляхи об'єктів, повертаючи тимчасові сесії браузерів або конфігурації на їхні початкові місця у файловій системі.

Моніторинг виконання внутрішніх процесів та діагностика позаштатних ситуацій забезпечуються класом `LogService`. Формат журналювання розроблявся з акцентом на зручність подальшого машинного та ручного аналізу. Кожен текстовий запис містить мітку часу, ідентифікатор потоку, рівень критичності та детальний опис події. Запис операцій здійснюється асинхронно, щоб мінімізувати вплив на швидкодію дискових операцій.

Оскільки повідомлення можуть надходити одночасно з різних паралельних задач у пулі потоків, цілісність лог-файлу підтримується механізмами синхронізації (використанням об'єкта блокування `lock` навколо файлового потоку). Для запобігання неконтрольному розростанню текстових баз імплементовано механізм ротації журналів. Під час запуску `LogService` перевіряє розмір поточного документа; якщо він перевищує заданий ліміт, файл переміщується до архіву, а система ініціалізує новий порожній журнал.

Керування конфігурацією утиліти (шляхи до каталогів, списки винятків, параметри безпечного видалення) винесено за межі вихідного коду у зовнішній файл `settings.json`. Архітектура цього модуля базується на високопродуктивній бібліотеці `System.Text.Json`. Механізм серіалізації та десеріалізації налаштовано з використанням об'єкта `JsonSerializerOptions`, який забезпечує форматування відступів (`WriteIndented`) для збереження

читабельності документа при його ручному редагуванні. Під час старту програми `SettingsService` зчитує файл, трансформуючи текстовий JSON-формат у типізовані об'єктні моделі C#. Якщо фізичний файл відсутній, сервіс автоматично генерує конфігурацію зі значеннями за замовчуванням і зберігає її на накопичувачі. Будь-які зміни, які користувач здійснює через налаштування інтерфейсу, миттєво серіалізуються та оновлюють структуру фізичного файлу.

Комунікація з користувачем відбувається через графічну оболонку на базі `Windows Forms`, центральним елементом якої виступає `MainForm`. Проектування форми виконано за принципами побудови адаптивних інтерфейсів із застосуванням компонента `TableLayoutPanel`, що гарантує коректне масштабування кнопок та смуги прогресу незалежно від системних налаштувань роздільної здатності. Форма `SettingsForm` надає доступ до управління конфігурацією, дозволяючи редагувати маски винятків через інтерактивні списки та прапорці. Спеціалізована `RestoreForm` проектувалася як модальне діалогове вікно. Вона підвантажує список збережених ZIP-архівів і дозволяє ініціювати розпакування, тимчасово блокуючи взаємодію з головним вікном для запобігання конфліктам файлової системи під час процесу відновлення.

Робота з низькорівневими системними ресурсами `Windows` вимагає створення системи обробки виняткових ситуацій. У розробленій архітектурі впроваджено дворівневу модель ізоляції помилок. Локальний рівень сконцентрований у циклах взаємодії з простором імен `System.IO`. Найсерйознішою перешкодою для безперервності процесу є виняток `UnauthorizedAccessException`, який неминуче виникає під час спроби читання захищених гілок користувацьких профілів. Обгорнувши виклики `Directory.EnumerateFiles` у блоки `try-catch`, програма перехоплює цю помилку, записує попередження в лог і спокійно переходить до наступного каталогу. Іншим типовим бар'єром є `IOException`, що сигналізує про жорстке блокування файлу стороннім процесом. Глобальний рівень стабільності реалізується через

підписку на подію `Application.ThreadException`, яка перехоплює непередбачені збої в потоці графічного інтерфейсу, запобігаючи раптовому падінню програми та гарантуючи збереження останніх записів у системі журналювання.

3.2 Розгляд роботи розробленої утиліти

Експлуатація розробленої програми демонструє взаємодію всіх спроектованих архітектурних рівнів. Робочий цикл починається із запуску виконуваного файлу, під час якого середовище .NET ініціалізує головне вікно додатку (рисунок 3.1).

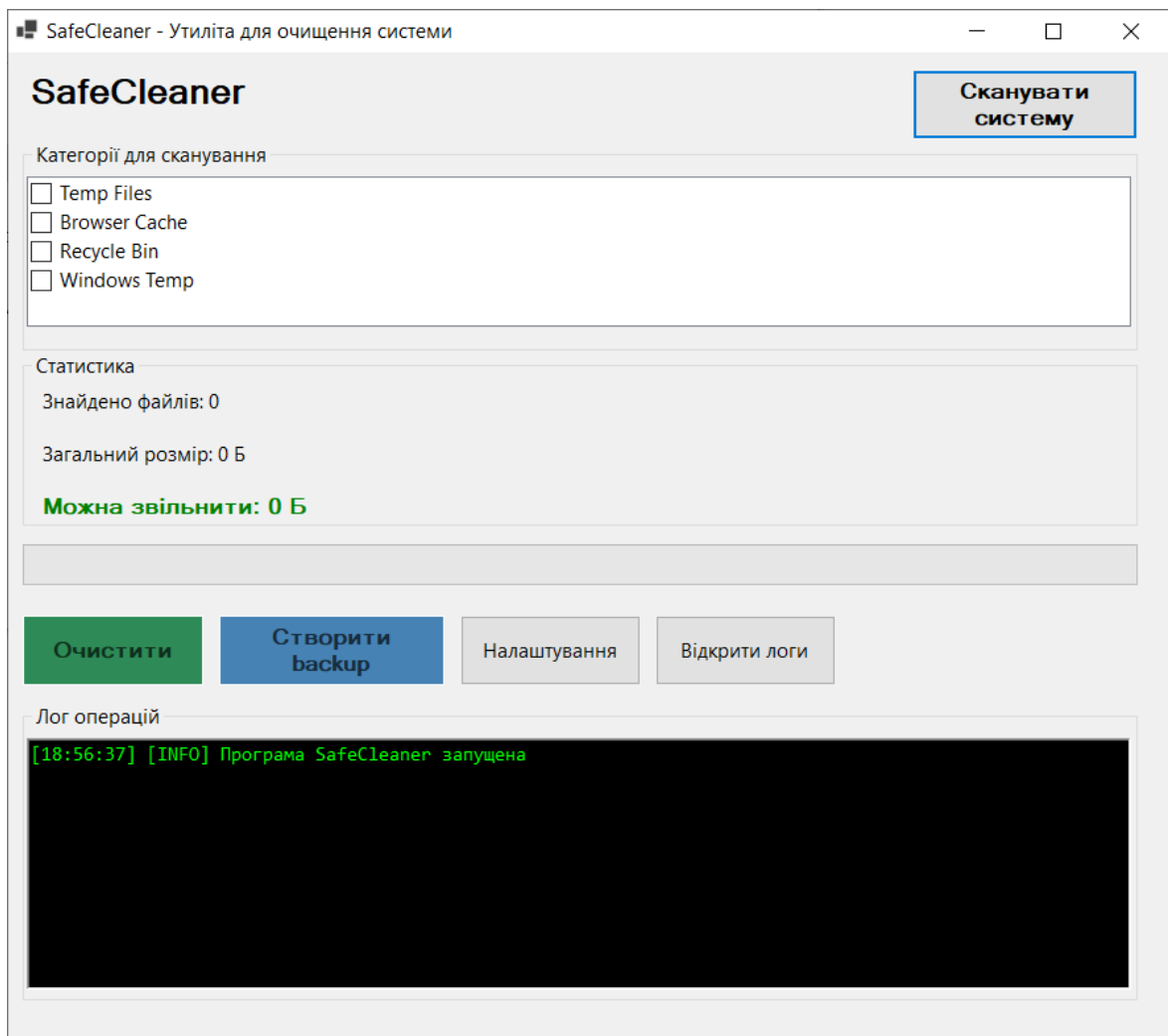


Рисунок 3.1 – Головне вікно програми для безпечного очищення тимчасових файлів та кешу операційної системи

На етапі завантаження форми автоматично спрацьовує метод звернення до `SettingsService`, який зчитує конфігураційний файл `settings.json` та відновлює попередній стан програми. Перед користувачем з'являється панель керування, де центральне місце займає компонент `CheckedListBox` із переліком доступних для очищення категорій, таких як системні директорії, кошик та кеш-папки браузерів.

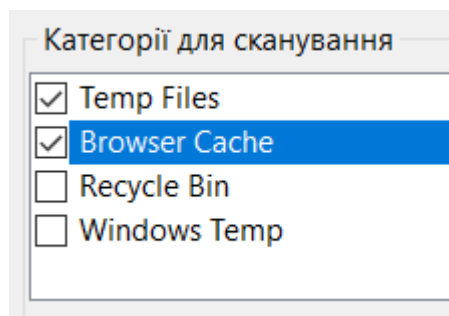


Рисунок 3.2 – Чекбокси категорій сканування

Взаємодія ініціюється вибором цільових зон та натисканням кнопки сканування. Згідно з принципами проєктування користувацьких інтерфейсів, система миттєво реагує на дію: основна кнопка блокується для запобігання повторним викликам, а замість неї активується елемент скасування операції. Водночас у фоновому пулі потоків запускається асинхронне завдання, яке виконує обхід файлової системи.

Завдяки використанню інтерфейсу `IProgress<T>`, користувач бачить плавне оновлення індикатора `ProgressBar` без ефекту «зависання» вікна, що є важливим для забезпечення комфортної роботи в додатках на базі `Windows Forms`.

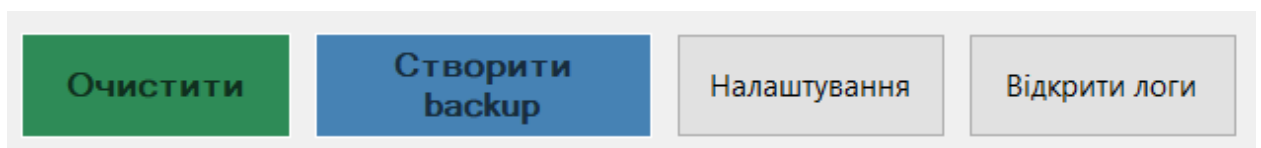


Рисунок 3.3 – Кнопки керування програмою

Після завершення аналізу директорій програма формує звіт. У багаторядковому полі RichTextBox виводиться перелік знайдених тимчасових файлів та загальний обсяг простору, який можна звільнити.

На цьому етапі утиліта переходить у стан очікування, надаючи можливість ознайомитися з результатами і прийняти фінальне рішення. Якщо в списку присутні файли, які необхідно зберегти, користувач має змогу відкрити додаткове модальне вікно SettingsForm та додати відповідні розширення або шляхи до списку конфігураційних винятків.

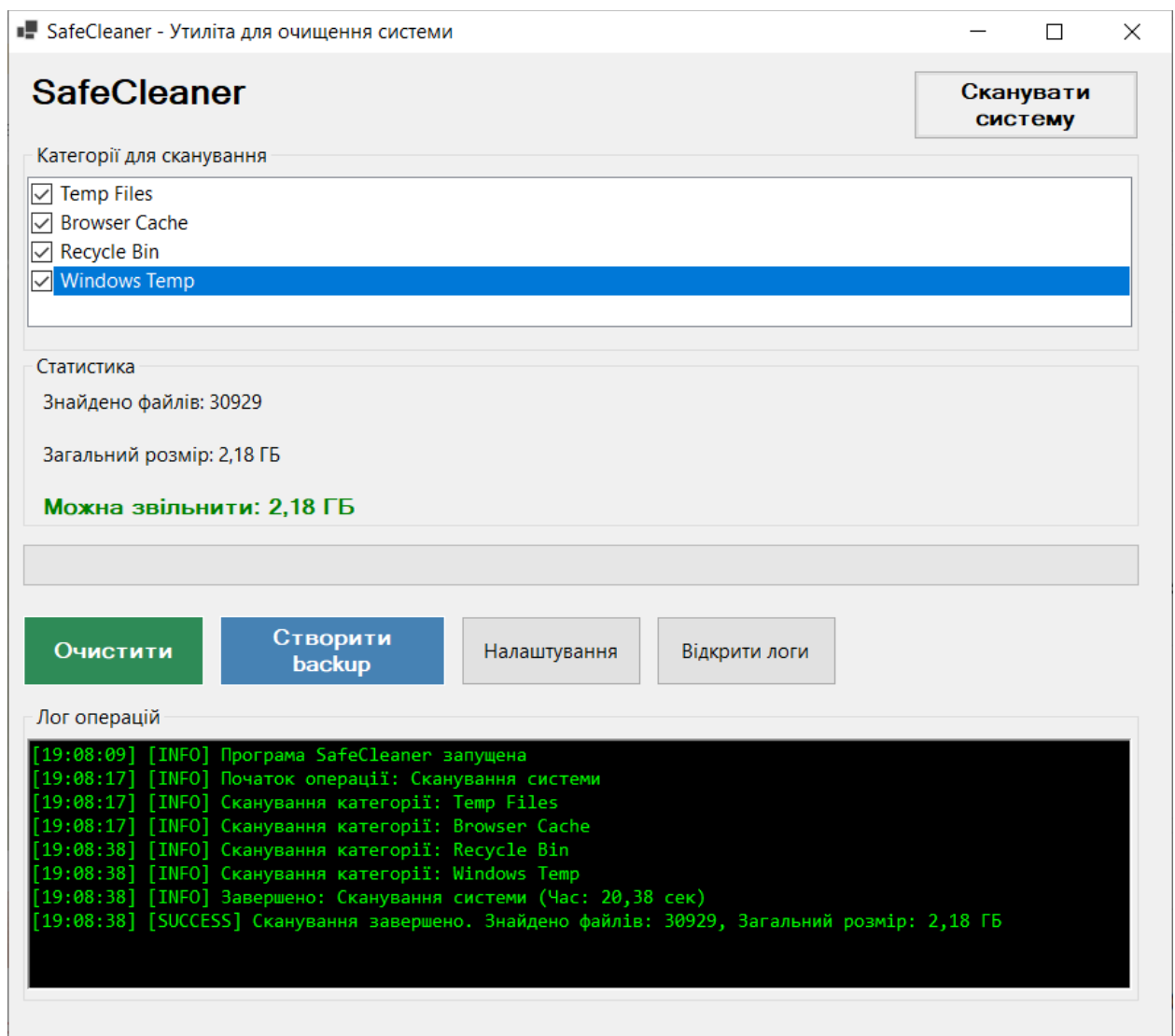


Рисунок 3.4 – Результати сканування тимчасових файлів та кешу операційної системи

Запуск процесу очищення активує ланцюжок сервісних викликів. Спочатку управління передається модулю BackupService, який у фоновому режимі пакує цільові файли у ZIP-архів. Лише після перевірки цілісності цього архіву ініціюється алгоритм очищення. У консоль логування, розміщену в нижній частині головного вікна, динамічно виводяться повідомлення про статус обробки кожного об'єкта. Програма використовує кольорове маркування: стандартним текстом фіксується успішне видалення, а червоним кольором виділяються об'єкти, доступ до яких був заблокований сторонніми процесами (наприклад, перехоплені винятки типу IOException).

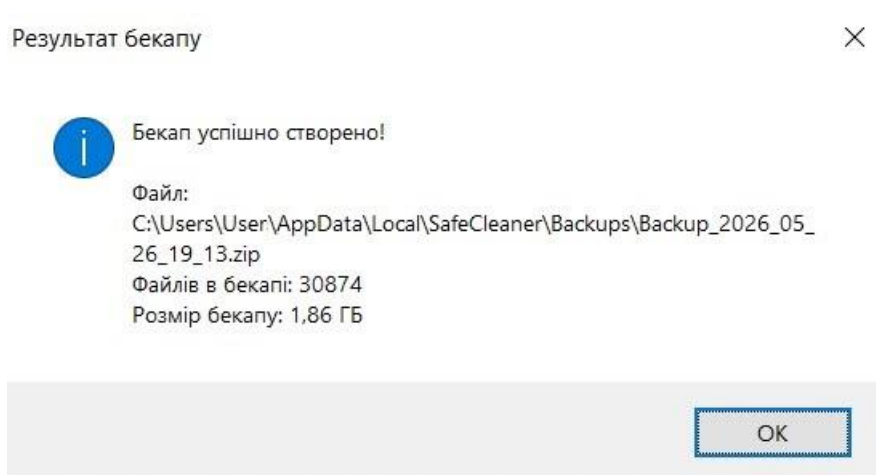


Рисунок 3.5 – Результати сканування тимчасових файлів та кешу операційної системи

Фінальним етапом робочого циклу є підрахунок фактично звільненого місця та запис підсумкової статистики у текстовий журнал подій. Після цього графічний інтерфейс розблоковується і повертається до початкового стану, готовий до виконання наступних ітерацій сканування або виходу із програми.

ВИСНОВКИ

У ході виконання дипломної роботи було успішно розв'язане актуальне завдання з розробки програмного інструменту для обслуговування та оптимізації дискового простору. Теоретичний аналіз механізмів накопичення залишкових даних в операційних системах сімейства Windows показав, що стандартні утиліти та популярні комерційні рішення часто мають закритий вихідний код, надлишкову фонову функціональність або не забезпечують належного рівня захисту під час знищення конфіденційної інформації. Це обґрунтувало необхідність створення спеціалізованої програми з прозорою та зрозумілою логікою роботи.

Програмна реалізація проекту повністю базується на платформі .NET 6 із застосуванням мови програмування C# 10. Використання багаторівневої архітектури (layered architecture) дозволило ізолювати графічний інтерфейс, побудований за допомогою технології Windows Forms, від внутрішніх сервісів бізнес-логіки. Для забезпечення гнучкості налаштувань було розроблено автономний модуль конфігурації. Він використовує можливості простору імен System.Text.Json для збереження параметрів сканування, цільових директорій та списків винятків у форматі JSON, що дозволяє користувачеві адаптувати роботу програми без втручання в код і перекомпіляції.

Ядро розробленої утиліти ефективно взаємодіє з файловою системою завдяки інструментарію System.IO. Було спроектовано та впроваджено алгоритми безпечного рекурсивного сканування тимчасових каталогів ОС, вмісту кошика та специфічних директорій кешу популярних веб-браузерів. Щоб ресурсомісткі операції вводу-виводу не блокували головний потік вікна додатку, усі процеси переведено в асинхронний режим за допомогою об'єктів Task та конструкцій async/await. Інтеграція токенів CancellationToken гарантує миттєве переривання тривалих операцій на вимогу користувача.

Для гарантування збереження важливих даних та стабільності системи загалом реалізовано підсистему резервного копіювання. Перед запуском

процесів очищення відповідний сервіс формує ZIP-архіви обраних файлів, що дозволяє відновити стан каталогів у разі випадкового видалення. Безперервна робота програми підтримується надійним механізмом перехоплення системних помилок через блоки try-catch. Утиліта коректно обробляє конфлікти доступу, зокрема винятки IOException та UnauthorizedAccessException, фіксуючи їх у деталізованому текстовому журналі через створений модуль логування.

У результаті створено готовий до практичної експлуатації програмний продукт. Розроблений додаток має інтуїтивно зрозумілий інтерфейс, не вимагає встановлення важких фонових служб і повністю задовольняє початкові функціональні та нефункціональні вимоги, надаючи користувачам безпечний та надійний інструмент для керування файлами.

ПЕРЕЛІК ПОСИЛАНЬ

1. Garfinkel S., Shelat A. Remembrance of Data Passed: A Study of Disk Sanitization Practices. IEEE Security & Privacy. 2003. Vol. 1, no. 1. P. 17–27.
2. Troelsen A., Japikse P. Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming. New York : Apress, 2022. 1306 p.
3. Russinovich M., Solomon D., Ionescu A. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more. 7th ed. Redmond : Microsoft Press, 2017. 752 p.
4. Microsoft Corporation. Path.GetTempFileName Method (System.IO). Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.io.path.gettempfilename> (дата звернення: 01.05.2026).
5. Cserhati G. Windows Server 2022 Administration Fundamentals. Birmingham : Packt Publishing, 2022. 394 p.
6. Micheloni R., Marelli A., Eshghi K. Inside Solid State Drives (SSDs). 2nd ed. Berlin : Springer, 2013. 396 p.
7. Carrier B. File System Forensic Analysis. Boston : Addison-Wesley Professional, 2005. 600 p.
8. Cleary S. Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming. 2nd ed. Sebastopol : O'Reilly Media, 2019. 254 p.
9. Microsoft Corporation. Manage drive space with Storage Sense. Microsoft Support. URL: <https://support.microsoft.com/en-us/windows/manage-drive-space-with-storage-sense-654f6ada-7bfc-45e5-966b-e24aded96ad5> (дата звернення: 02.05.2026).
10. Kaur K., Kaur A. Forensic Analysis of Web Browser: Google Chrome, Mozilla Firefox and Internet Explorer. 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI). New Delhi, 2014. P. 1383–1388.
11. Piriform Software Ltd. CCleaner: Optimize & Clean Your PC. URL: <https://www.ccleaner.com/> (дата звернення: 03.05.2026).

12. BleachBit. Open Source System Cleaner. URL: <https://www.bleachbit.org/> (дата звернення: 03.05.2026).
13. Glarysoft Ltd. Glary Utilities. URL: <https://www.glarysoft.com/> (дата звернення: 03.05.2026).
14. Sommerville I. Software Engineering. 10th ed. Boston : Pearson, 2015. 816 p.
15. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 8th ed. New York : McGraw-Hill Education, 2014. 976 p.
16. Microsoft Corporation. System.IO.Compression Namespace. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.io.compression> (дата звернення: 04.05.2026).
17. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley Professional, 2021. 448 p.
18. Microsoft Corporation. Directory.EnumerateFiles Method. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.io.directory.enumeratefiles> (дата звернення: 05.05.2026).
19. Microsoft Corporation. System.IO Namespace. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.io> (дата звернення: 05.05.2026).
20. Microsoft Corporation. How to serialize and deserialize JSON in .NET. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/how-to> (дата звернення: 05.05.2026).
21. Fowler M. Patterns of Enterprise Application Architecture. Boston : Addison-Wesley, 2002. 560 p.
22. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston : Prentice Hall, 2017. 432 p.
23. Price M. J. C# 10 and .NET 6 – Modern Cross-Platform Development. 6th ed. Birmingham : Packt Publishing, 2021. 822 p.
24. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.

25. Microsoft Corporation. Desktop Guide (Windows Forms .NET). Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/desktop/winforms/?view=netdesktop-6.0> (дата звернення: 07.05.2026).
26. Sells C., Weinhardt M. Windows Forms 2.0 Programming. 2nd ed. Boston : Addison-Wesley Professional, 2006. 1032 p.
27. McConnell S. Code Complete: A Practical Handbook of Software Construction. 2nd ed. Redmond : Microsoft Press, 2004. 960 p.
28. Windows Security Documentation. User Account Control. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/> (дата звернення: 10.05.2026).
29. Microsoft Corporation. System.Security.Cryptography Namespace. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography> (дата звернення: 10.05.2026).

Додаток А - Програмний код

ScanService.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using SafeCleaner.Helpers;
using SafeCleaner.Models;

namespace SafeCleaner.Services
{
    /// <summary>
    /// Сервіс для сканування системи на наявність тимчасових
    файлів та кешу
    /// </summary>
    public class ScanService
    {
        private readonly LogService _logService;
        private readonly AppSettings _settings;
        private CancellationTokenSource
        _cancellationTokencSource;

        /// <summary>
        /// Подія, що виникає при оновленні прогресу сканування
        /// </summary>
        public event EventHandler<ScanProgressEventArgs>
        ScanProgress;

        /// <summary>
        /// Подія, що виникає при знаходженні файлів
        /// </summary>
        public event EventHandler<ItemsFoundEventArgs>
        ItemsFound;

        /// <summary>
        /// Конструктор сервісу сканування
        /// </summary>
        /// <param name="logService">Сервіс логування</param>
        /// <param name="settings">Налаштування програми</param>
        public ScanService(LogService logService, AppSettings
        settings)
        {
            _logService = logService ?? throw new
            ArgumentNullException(nameof(logService));
            _settings = settings ?? throw new
            ArgumentNullException(nameof(settings));
        }
    }
}
```

```

        /// <summary>
        /// Асинхронно сканувати систему
        /// </summary>
        /// <param name="categories">Список категорій для
сканування</param>
        /// <returns>Результат сканування</returns>
        public async Task<ScanResult>
ScanSystemAsync(List<string> categories)
        {
            if (categories == null || categories.Count == 0)
            {
                _logService.LogWarning("Список категорій для
сканування порожній");
                return new ScanResult();
            }

            _cancellationTokenSource = new
CancellationTokenSource();
            var result = new ScanResult();
            result.ScanStartTime = DateTime.Now;

            _logService.LogOperationStart("Сканування системи");

            try
            {
                // Скануємо кожну категорію
                foreach (var category in categories)
                {
                    if
(_cancellationTokenSource.Token.IsCancellationRequested)
                    {
                        _logService.LogWarning("Сканування
скасовано користувачем");
                        break;
                    }

                    await ScanCategoryAsync(category, result,
_cancellationTokenSource.Token);
                }

                result.ScanEndTime = DateTime.Now;
                result.IsCompleted = true;

                _logService.LogOperationEnd("Сканування
системи", result.GetScanDuration());
                _logService.LogSuccess($"Сканування завершено.
Знайдено файлів: {result.TotalFiles}, " +
                    $"Загальний розмір:
{SizeFormatter.FormatBytesUkrainian(result.TotalSize)}");
            }
            catch (OperationCanceledException)
            {

```

```

        _logService.LogWarning("Сканування скасовано");
        result.IsCompleted = false;
    }
    catch (Exception ex)
    {
        _logService.LogError("Помилка при скануванні
системи", ex);
        result.IsCompleted = false;
        result.ErrorMessage = ex.Message;
    }

    return result;
}

/// <summary>
/// Скасувати поточне сканування
/// </summary>
public void CancelScan()
{
    _cancellationTokenSource?.Cancel();
}

/// <summary>
/// Асинхронно сканувати конкретну категорію
/// </summary>
/// <param name="category">Категорія для
сканування</param>
/// <param name="result">Результат сканування</param>
/// <param name="cancellationToken">Токен
скасування</param>
private async Task ScanCategoryAsync(string category,
ScanResult result, CancellationToken cancellationToken)
{
    _logService.LogInfo($"Сканування категорії:
{category}");

    try
    {
        switch (category.ToLower())
        {
            case "temp files":
                await ScanTempFilesAsync(result,
cancellationToken);
                break;

            case "browser cache":
                await ScanBrowserCacheAsync(result,
cancellationToken);
                break;

            case "recycle bin":

```

```

        await ScanRecycleBinAsync(result,
cancellationToken);
        break;

        case "windows temp":
            await ScanWindowsTempAsync(result,
cancellationToken);
            break;

        default:
            _logService.LogWarning($"Невідома
категорія: {category}");
            break;
    }
}
catch (Exception ex)
{
    _logService.LogError($"Помилка при скануванні
категорії {category}", ex);
}

/// <summary>
/// Сканувати тимчасові файли користувача (%TEMP%)
/// </summary>
private async Task ScanTempFilesAsync(ScanResult result,
CancellationToken cancellationToken)
{
    await Task.Run(() =>
    {
        string tempPath = FileHelper.GetUserTempPath();
        ScanDirectory(tempPath, "Temp Files", result,
cancellationToken);
    }, cancellationToken);
}

/// <summary>
/// Сканувати тимчасові файли Windows (C:\Windows\Temp)
/// </summary>
private async Task ScanWindowsTempAsync(ScanResult
result, CancellationToken cancellationToken)
{
    await Task.Run(() =>
    {
        string windowsTempPath =
FileHelper.GetWindowsTempPath();
        ScanDirectory(windowsTempPath, "Windows Temp",
result, cancellationToken);
    }, cancellationToken);
}

/// <summary>

```

```

        /// Сканувати кеш браузерів
        /// </summary>
        private async Task ScanBrowserCacheAsync(ScanResult
result, CancellationToken cancellationToken)
        {
            await Task.Run(() =>
            {
                // Chrome Cache
                string chromeCachePath =
FileHelper.GetChromeCachePath();
                if (Directory.Exists(chromeCachePath))
                {
                    ScanDirectory(chromeCachePath, "Browser
Cache", result, cancellationToken);
                }

                // Edge Cache
                string edgeCachePath =
FileHelper.GetEdgeCachePath();
                if (Directory.Exists(edgeCachePath))
                {
                    ScanDirectory(edgeCachePath, "Browser
Cache", result, cancellationToken);
                }

                // Firefox Cache
                string firefoxCachePath =
FileHelper.GetFirefoxCachePath();
                if (Directory.Exists(firefoxCachePath))
                {
                    // Firefox має профілі, скануємо кожен
профіль
                    var profiles =
Directory.GetDirectories(firefoxCachePath);
                    foreach (var profile in profiles)
                    {
                        string cachePath = Path.Combine(profile,
"cache2");
                        if (Directory.Exists(cachePath))
                        {
                            ScanDirectory(cachePath, "Browser
Cache", result, cancellationToken);
                        }
                    }
                }
            }, cancellationToken);
        }

        /// <summary>
        /// Сканувати кошик Windows
        /// </summary>

```

```

        private async Task ScanRecycleBinAsync(ScanResult
result, CancellationToken cancellationToken)
        {
            await Task.Run(() =>
            {
                try
                {
                    string recycleBinPath =
FileHelper.GetRecycleBinPath();

                    if (!Directory.Exists(recycleBinPath))
                    {
                        _logService.LogVerbose($"Кошик не
знайдено за шляхом: {recycleBinPath}");
                        return;
                    }

                    // Отримуємо всі піддиректорії (SID
користувачів)
                    var sids =
Directory.GetDirectories(recycleBinPath);

                    foreach (var sid in sids)
                    {
                        if
(cancellationToken.IsCancellationRequested)
                            break;

                        ScanDirectory(sid, "Recycle Bin",
result, cancellationToken);
                    }
                }
                catch (Exception ex)
                {
                    _logService.LogError("Помилка при скануванні
кошика", ex);
                }
            }, cancellationToken);
        }

        /// <summary>
        /// Рекурсивно сканувати директорію
        /// </summary>
        /// <param name="directoryPath">Шлях до
директорії</param>
        /// <param name="category">Категорія</param>
        /// <param name="result">Результат сканування</param>
        /// <param name="cancellationToken">Токен
скасування</param>
        private void ScanDirectory(string directoryPath, string
category, ScanResult result, CancellationToken
cancellationToken)

```



```
{
    try
    {
        if (!Directory.Exists(directoryPath))
        {
            _logService.LogVerbose($"Директорія не існує: {directoryPath}");
            return;
        }

        // Перевіряємо, чи директорія не виключена
        if (_settings.IsPathExcluded(directoryPath))
        {
            _logService.LogVerbose($"Директорія виключена: {directoryPath}");
            return;
        }

        _logService.LogVerbose($"Сканування директорії: {directoryPath}");

        // Отримуємо всі файли
        var files =
            FileHelper.GetFilesSafely(directoryPath, "*", true);

        foreach (var file in files)
        {
            if
                (cancellationToken.IsCancellationRequested)
                break;

            // Перевіряємо, чи файл не виключений
            string fileName = Path.GetFileName(file);
            if
                (_settings.IsFilePatternExcluded(fileName))
                {
                    _logService.LogVerbose($"Файл виключений: {file}");
                    continue;
                }

            try
            {
                var fileInfo = new FileInfo(file);
                long fileSize = fileInfo.Length;

                // Створюємо елемент очищення
                var item = new CleanupItem(file,
                    fileSize, category, false);
                result.AddItem(item);

                // Сповіщаємо про знайдені файли
            }
            catch { }
        }
    }
}
```

```

        OnItemsFound(new ItemsFoundEventArgs(1,
fileSize));
    }
    catch (Exception ex)
    {
        _logService.LogVerbose($"Помилка при
обробці файлу {file}: {ex.Message}");
    }
}

// Оновлюємо прогрес
OnScanProgress(new
ScanProgressEventArgs(category, 100));
}
catch (UnauthorizedAccessException ex)
{
    _logService.LogVerbose($"Немає доступу до
директорії {directoryPath}: {ex.Message}");
}
catch (IOException ex)
{
    _logService.LogVerbose($"Помилка IO при
скануванні {directoryPath}: {ex.Message}");
}
catch (Exception ex)
{
    _logService.LogError($"Помилка при скануванні
директорії {directoryPath}", ex);
}
}

/// <summary>
/// Сповіщати про оновлення прогресу
/// </summary>
protected virtual void
OnScanProgress(ScanProgressEventArgs e)
{
    ScanProgress?.Invoke(this, e);
}

/// <summary>
/// Сповіщати про знайдені файли
/// </summary>
protected virtual void OnItemsFound(ItemsFoundEventArgs
e)
{
    ItemsFound?.Invoke(this, e);
}

}

/// <summary>
/// Аргументи події прогресу сканування

```

```

    /// </summary>
    public class ScanProgressEventArgs : EventArgs
    {
        public string Category { get; set; }
        public int ProgressPercentage { get; set; }

        public ScanProgressEventArgs(string category, int
progressPercentage)
        {
            Category = category;
            ProgressPercentage = progressPercentage;
        }
    }

    /// <summary>
    /// Аргументи події знайдених файлів
    /// </summary>
    public class ItemsFoundEventArgs : EventArgs
    {
        public int FilesCount { get; set; }
        public long TotalSize { get; set; }

        public ItemsFoundEventArgs(int filesCount, long
totalSize)
        {
            FilesCount = filesCount;
            TotalSize = totalSize;
        }
    }
}

```

CleanupService.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using SafeCleaner.Helpers;
using SafeCleaner.Models;

namespace SafeCleaner.Services
{
    /// <summary>
    /// Сервіс для безпечного видалення файлів
    /// </summary>
    public class CleanupService
    {
        private readonly LogService _logService;
        private readonly AppSettings _settings;
    }
}

```

```

        private CancellationTokenSource
        _cancellationTokenSource;

        /// <summary>
        /// Подія, що виникає при оновленні прогресу очищення
        /// </summary>
        public event EventHandler<CleanupProgressEventArgs>
        CleanupProgress;

        /// <summary>
        /// Подія, що виникає при видаленні файлу
        /// </summary>
        public event EventHandler<FileDeletedEventArgs>
        FileDeleted;

        /// <summary>
        /// Конструктор сервісу очищення
        /// </summary>
        /// <param name="logService">Сервіс логування</param>
        /// <param name="settings">Налаштування програми</param>
        public CleanupService(LogService logService, AppSettings
        settings)
        {
            _logService = logService ?? throw new
        ArgumentNullException(nameof(logService));
            _settings = settings ?? throw new
        ArgumentNullException(nameof(settings));
        }

        /// <summary>
        /// Асинхронно видалити файли
        /// </summary>
        /// <param name="items">Список елементів для
        видалення</param>
        /// <param name="selectedCategories">Обрані категорії
        для видалення</param>
        /// <returns>Результат очищення</returns>
        public async Task<CleanupResult>
        CleanupAsync(List<CleanupItem> items, List<string>
        selectedCategories)
        {
            if (items == null || items.Count == 0)
            {
                _logService.LogWarning("Список файлів для
        видалення порожній");
                return new CleanupResult();
            }

            if (selectedCategories == null ||
        selectedCategories.Count == 0)
            {

```

```

        _logService.LogWarning("Не обрано жодної
категорії для видалення");
        return new CleanupResult();
    }

    _cancellationTokenSource = new
CancellationTokenSource();
    var result = new CleanupResult();
    result.StartTime = DateTime.Now;

    _logService.LogOperationStart("Очищення системи");

    try
    {
        // Фільтруємо елементи за обраними категоріями
        var itemsToDelete = items
            .Where(item =>
selectedCategories.Contains(item.Category,
StringComparer.OrdinalIgnoreCase))
            .ToList();

        result.TotalItems = itemsToDelete.Count;
        result.TotalSize = itemsToDelete.Sum(item =>
item.Size);

        _logService.LogInfo($"До видалення:
{result.TotalItems} файлів, " +

        $"{SizeFormatter.FormatBytesUkrainian(result.TotalSize)}");

        // Видаляємо файли асинхронно
        int processedCount = 0;
        foreach (var item in itemsToDelete)
        {
            if
(_cancellationTokenSource.Token.IsCancellationRequested)
            {
                _logService.LogWarning("Очищення
скасовано користувачем");
                break;
            }

            bool deleted = await DeleteItemAsync(item);

            if (deleted)
            {
                result.DeletedCount++;
                result.DeletedSize += item.Size;
                result.DeletedItems.Add(item);
            }
            else
            {

```

```

        result.SkippedCount++;
        result.SkippedSize += item.Size;
        result.SkippedItems.Add(item);
    }

    processedCount++;

    // Оновлюємо прогрес
    int progressPercentage =
(int)((double)processedCount / result.TotalItems * 100);
    OnCleanupProgress(new
CleanupProgressEventArgs(progressPercentage, item.Path,
deleted));
    }

    result.EndTime = DateTime.Now;
    result.IsCompleted = true;

    _logService.LogOperationEnd("Очищення системи",
result.GetDuration());
    _logService.LogSuccess($"Очищення завершено.
Видалено: {result.DeletedCount} файлів, " +

$"{SizeFormatter.FormatBytesUkrainian(result.DeletedSize)}");
    _logService.LogInfo($"Пропущено:
{result.SkippedCount} файлів, " +

$"{SizeFormatter.FormatBytesUkrainian(result.SkippedSize)}");
    }
    catch (OperationCanceledException)
    {
        _logService.LogWarning("Очищення скасовано");
        result.IsCompleted = false;
    }
    catch (Exception ex)
    {
        _logService.LogError("Помилка при очищенні
системи", ex);
        result.IsCompleted = false;
        result.ErrorMessage = ex.Message;
    }

    return result;
}

/// <summary>
/// Скасувати поточне очищення
/// </summary>
public void CancelCleanup()
{
    _cancellationTokenSource?.Cancel();
}

```

```

    /// <summary>
    /// Асинхронно видалити один елемент
    /// </summary>
    /// <param name="item">Елемент для видалення</param>
    /// <returns>True, якщо видалення успішне</returns>
    private async Task<bool> DeleteItemAsync(CleanupItem
item)
    {
        return await Task.Run(() =>
        {
            try
            {
                // Перевіряємо, чи шлях не виключений
                if (_settings.IsPathExcluded(item.Path))
                {
                    _logService.LogSkippedFile(item.Path,
"Шлях виключений в налаштуваннях");
                    item.ErrorMessage = "Шлях виключений в
налаштуваннях";
                    return false;
                }

                // Перевіряємо, чи файл не заблокований
                if (FileHelper.IsFileLocked(item.Path))
                {
                    _logService.LogSkippedFile(item.Path,
"Файл заблокований іншим процесом");
                    item.ErrorMessage = "Файл заблокований
іншим процесом";
                    return false;
                }

                bool deleted;
                if (item.IsDirectory)
                {
                    deleted =
FileHelper.DeleteDirectorySafely(item.Path, true);
                }
                else
                {
                    deleted =
FileHelper.DeleteFileSafely(item.Path);
                }

                if (deleted)
                {
                    item.IsDeleted = true;
                    _logService.LogDeletedFile(item.Path);
                    OnFileDeleted(new
FileDeletedEventArgs(item.Path, item.Size, true));
                    return true;
                }
            }
            catch { }
        });
    }
}

```

```

        }
        else
        {
            item.ErrorMessage = "Не вдалося видалити
файл";
            _logService.LogSkippedFile(item.Path,
"Не вдалося видалити");
            OnFileDeleted(new
FileDeletedEventArgs(item.Path, item.Size, false));
            return false;
        }
    }
    catch (UnauthorizedAccessException ex)
    {
        item.ErrorMessage = "Немає доступу";
        _logService.LogSkippedFile(item.Path, "Немає
доступу");
        return false;
    }
    catch (IOException ex)
    {
        item.ErrorMessage = "Помилка IO";
        _logService.LogSkippedFile(item.Path,
"Помилка IO");
        return false;
    }
    catch (Exception ex)
    {
        item.ErrorMessage = ex.Message;
        _logService.LogSkippedFile(item.Path,
ex.Message);
        return false;
    }
    });
}

```

```

/// <summary>
/// Очистити кошик Windows
/// </summary>
/// <returns>True, якщо очищення успішне</returns>
public async Task<bool> EmptyRecycleBinAsync()
{
    return await Task.Run(() =>
    {
        try
        {
            _logService.LogInfo("Очищення кошика
Windows");
            bool result =
FileHelper.EmptyRecycleBin(false, false, false);

            if (result)

```



```

        {
            _logService.LogSuccess("Кошик успішно
очищено");
        }
        else
        {
            _logService.LogWarning("Не вдалося
очистити кошик");
        }

        return result;
    }
    catch (Exception ex)
    {
        _logService.LogError("Помилка при очищенні
кошика", ex);
        return false;
    }
    });
}

/// <summary>
/// Сповіщати про оновлення прогресу
/// </summary>
protected virtual void
OnCleanupProgress(CleanupProgressEventArgs e)
{
    CleanupProgress?.Invoke(this, e);
}

/// <summary>
/// Сповіщати про видалення файлу
/// </summary>
protected virtual void
OnFileDeleted(FileDeletedEventArgs e)
{
    FileDeleted?.Invoke(this, e);
}

}

/// <summary>
/// Результат очищення
/// </summary>
public class CleanupResult
{
    public int TotalItems { get; set; }
    public long TotalSize { get; set; }
    public int DeletedCount { get; set; }
    public long DeletedSize { get; set; }
    public int SkippedCount { get; set; }
    public long SkippedSize { get; set; }
    public List<CleanupItem> DeletedItems { get; set; }
}

```

```

    public List<CleanupItem> SkippedItems { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
    public bool IsCompleted { get; set; }
    public string ErrorMessage { get; set; }

    public CleanupResult()
    {
        DeletedItems = new List<CleanupItem>();
        SkippedItems = new List<CleanupItem>();
        StartTime = DateTime.Now;
        EndTime = DateTime.Now;
        IsCompleted = false;
        ErrorMessage = string.Empty;
    }

    public TimeSpan GetDuration()
    {
        return EndTime - StartTime;
    }
}

/// <summary>
/// Аргументи події прогресу очищення
/// </summary>
public class CleanupProgressEventArgs : EventArgs
{
    public int ProgressPercentage { get; set; }
    public string FilePath { get; set; }
    public bool Success { get; set; }

    public CleanupProgressEventArgs(int progressPercentage,
string filePath, bool success)
    {
        ProgressPercentage = progressPercentage;
        FilePath = filePath;
        Success = success;
    }
}

/// <summary>
/// Аргументи події видалення файлу
/// </summary>
public class FileDeletedEventArgs : EventArgs
{
    public string FilePath { get; set; }
    public long FileSize { get; set; }
    public bool Success { get; set; }

    public FileDeletedEventArgs(string filePath, long
fileSize, bool success)
    {

```

```
        FilePath = filePath;  
        FileSize = fileSize;  
        Success = success;  
    }  
}  
}
```