МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет водного господарства та природокористування
Навчально-науковий інститут кібернетики, інформаційних технологій та інженерії
Кафедра комп'ютерних технологій та економічної кібернетики

Допущено до захисту:
Завідувач кафедри
_____ д. е. н., проф. П. М. Грицюк
«_____» _____ 20___ р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня «бакалавр»
за освітньо-професійною програмою «Інформаційні системи та технології»
спеціальності 126 «Інформаційні системи та технології»
на тему: «Розробка та реалізація інформаційно-облікової системи автотранспортних перевезень з використанням технологій баз даних»

Виконав:
здобувач вищої освіти 4 курсу, групи ІСТ-41
Калашніков Владислав Ігорович.

Керівник:
ст. викл., Шевченко Ірина Мавіївна.

Рецензент:
к.т.н., доцент, Барановський Сергій Віталійович.

Рівне – 2025

ЗМІСТ

ВСТУП

У сучасних умовах цифровізації економіки ефективність функціонування автотранспортних підприємств значною мірою залежить від рівня автоматизації облікових процесів та управління ресурсами. Незважаючи на наявність комерційних рішень, більшість із них є дорогими, складними в адаптації або перевантаженими зайвим функціоналом, що не відповідає потребам малих і середніх перевізників. Це створює об'єктивну потребу у розробці гнучкого, зручного та адаптованого програмного забезпечення для обліку автотранспортних перевезень.

Аналіз існуючих систем показав, що хоча вони охоплюють широкий спектр завдань, їх впровадження часто потребує спеціалізованих знань, додаткових витрат та складної підтримки. У зв’язку з цим розробка власного настільного застосунку на базі сучасних технологій є доцільним рішенням, яке дозволяє врахувати специфіку підприємства, підвищити керованість процесами перевезень і забезпечити надійне зберігання даних.

Актуальність теми полягає у зростаючій потребі в ефективних та доступних інформаційних системах для автоматизації обліку роботи підприємств. Невирішеними залишаються питання побудови адаптивного інтерфейсу, інтеграції обліку з аналітикою в єдиній системі та захисту даних при локальному зберіганні.

Метою роботи є розробка інформаційно-облікової системи для автотранспортного підприємства, яка забезпечить автоматизацію ключових бізнес-процесів, облік ресурсів, формування звітності та захист інформації.

Для досягнення мети необхідно вирішити такі завдання:

- здійснити аналіз предметної області та існуючих рішень;
- спроєктувати структуру бази даних;
- реалізувати інтерфейс користувача з використанням;

- розробити функціонал для обліку транспорту, водіїв, ПММ, рейсів та складів;
- реалізувати механізми авторизації та ролей;
- забезпечити можливість формування звітів;
- провести тестування та оцінку працездатності системи.

Об’єктом дослідження є процес обліку в автотранспортному підприємстві.

Предмет дослідження – методи та засоби побудови інформаційно-облікової системи автоперевезень з використанням технологій баз даних.

Практична цінність роботи полягає у створенні застосунку, який може бути легко адаптований до конкретних потреб підприємства, масштабований у майбутньому. Застосунок можна впроваджувати як у локальну, так і в мережеву інфраструктуру за допомогою незначної модифікації.

РОЗДІЛ 1. ХАРАКТЕРИСТИКА І АНАЛІЗ ПРОБЛЕМИ

1.1 Опис та аналіз об’єкта дослідження, виявлення існуючих проблем

У сучасному суспільстві транспортна галузь відіграє ключову роль у забезпеченні функціонування економіки, торгівлі та логістики. Автотранспортні перевезення, як один з найбільш гнучких та масових способів доставки вантажів, потребують ефективного управління для забезпечення своєчасності, безпеки та економічної доцільності здійснення рейсів.

У зв’язку з цим виникає необхідність використання спеціалізованого програмного забезпечення, яке б дозволяло автоматизувати ключові процеси обліку транспорту, водіїв, маршрутів, замовлень, витрат і аналітики. За допомогою інформаційних систем підприємство отримує можливість оперативного доступу до даних, централізованого контролю за перевезеннями, підвищення точності планування та скорочення витрат часу на обробку документації.

Застосування програмного забезпечення в автотранспортних підприємствах охоплює широкий спектр завдань: ведення обліку технічного стану автопарку, планування рейсів, розподіл ресурсів, контроль за виконанням перевезень, облік пального та витрат, аналіз ефективності роботи, взаємодія з клієнтами тощо. Такий підхід дозволяє не лише покращити якість обслуговування, а й знизити вплив людського фактору, оптимізувати внутрішні бізнес-процеси. Існує кілька основних типів програмного забезпечення, здатних задовольнити потреби транспортних підприємств:

- Настільні (desktop) системи – встановлюються на локальні комп’ютери, не потребують підключення до Інтернету, ідеально підходять для малих і середніх компаній;
- Клієнт-серверні рішення – працюють у локальній мережі з централізованим зберіганням даних, зручні для середніх і великих підприємств;

- Веб-орієнтовані системи – забезпечують доступ через браузер, дозволяють віддалено керувати перевезеннями, актуальні для компаній із розгалуженою структурою;
- Мобільні застосунки – доповнюють основні системи, дозволяючи водіям або диспетчерам взаємодіяти з даними в дорозі.

Попри очевидні переваги, впровадження програмного забезпечення пов’язане з рядом проблем. Серед них: висока вартість готових комерційних продуктів, складність адаптації типових рішень до конкретних бізнес-процесів підприємства, недостатній рівень IT-підготовки персоналу, а також ризики, пов’язані з перенесенням або втратою даних. У випадку недостатньо адаптованого або надто складного для користувача ПЗ, система може залишитись невикористаною або неефективною, попри всі технічні можливості.

У зв’язку з цим виникає потреба у використанні індивідуального, гнучкого, адаптованого до потреб конкретного підприємства інформаційного рішення, що поєднує в собі простоту використання, функціональну повноту та ефективність у збереженні та обробці даних. Така система дозволить максимально точно відповідати специфіці роботи підприємства, з урахуванням масштабів, структури і внутрішніх регламентів.

Потреба у використанні програмних засобів також зумовлена сучасними тенденціями у впровадженні процесів автоматизації бізнесу. Сучасна економіка дедалі більше орієнтується на автоматизацію процесів управління, що дозволяє підприємствам швидко адаптуватися до змін ринку, зменшувати витрати та підвищувати продуктивність. Автоматизація роботи підприємства полягає в перенесенні рутинних, повторюваних або складних для контролю процесів у програмне середовище, яке забезпечує зручний облік, моніторинг, аналіз і прийняття рішень. Особливо важливо це для підприємств у сфері автоперевезень, де велика кількість об’єктів (транспорт, водії, маршрути, вантажі) потребує постійного контролю.

Створення системи автоматизації критично впливає на таку важливу складову бізнес-процесів як облік роботи підприємства. Суть обліку на підприємстві полягає в систематичному зборі, обробці, зберіганні та аналізі інформації про основні виробничі та господарські процеси. У контексті автотранспортної галузі – це облік технічного стану транспорту, планування рейсів, фіксація витрат пального, розрахунок витрат, реєстрація замовлень та взаємодія з клієнтами. Без чіткого обліку підприємство ризикує втратити керованість, допустити дублювання даних або неточності у фінансовій звітності.

Водночас, потреба у підвищенні якості ведення обліку стає однією з ключових передумов автоматизації. Застарілі методи – такі як паперові журнали, електронні таблиці або нескоординовані бази даних – не дають змоги забезпечити оперативність, точність і цілісність інформації. Це особливо критично в умовах постійного зростання обсягів даних, що обробляються. Невчасно оновлена інформація про технічний стан транспорту або затримка в плануванні рейсів може призвести до збоїв у логістиці та фінансових втрат.

Згідно з сучасними тенденціями, якість обліку розглядається не лише як внутрішня функція підприємства, а як стратегічний ресурс. Прозорий, структурований і інтегрований облік дозволяє керівництву ухвалювати обґрунтовані управлінські рішення, швидко реагувати на зміни та формувати ефективну бізнес-стратегію. Саме тому автоматизація не є просто технічним оновленням, а – логічним кроком у напрямку цифрової трансформації підприємства.

Реалізація програмного забезпечення, яке підтримує автоматизований облік, забезпечує синхронізацію дій різних підрозділів, зменшує навантаження на персонал, мінімізує людські помилки та підвищує надійність збереження даних. Це особливо актуально для малих і середніх перевізників, які прагнуть залишатись конкурентоспроможними, але не мають доступу до дорогих ERP-систем.

Одним із ключових компонентів сучасних інформаційно-облікових систем є база даних, яка виконує функцію централізованого сховища всієї інформації, що генерується, обробляється та використовується підприємством. Роль баз даних у таких системах важко переоцінити: вони забезпечують збереження великих обсягів структурованих даних, гарантують швидкий доступ до них, дозволяють формувати звіти, здійснювати пошук, фільтрацію, оновлення та аналіз інформації в режимі реального часу.

У системах обліку автотранспортних перевезень база даних виконує критично важливі функції, зокрема:

- зберігання відомостей про транспортні засоби, водіїв, замовників і рейси;
- ведення історії обслуговування, витрат, маршрутів;
- реєстрацію заявок, завантаження і вивантаження вантажів;
- забезпечення контролю доступу до інформації;
- підтримку взаємодії між модулями системи (наприклад, між модулями обліку транспорту та фінансового обліку).

У залежності від масштабу системи, вимог до продуктивності та специфіки використання, використовуються різні технології баз даних. Вибір конкретної бази даних залежить від характеру завдань, які виконує облікова система, обсягів даних, вимог до безпеки, мобільності, доступності та складності супроводу.

1.2. Аналіз існуючих інформаційних методів вирішення проблеми

Розглядаючи існуючі рішення у сфері автоматизації бізнесу, основним прикладом було обрано платформу BAF (Business Automation Framework) та BAS (Business Automation Software) – спеціально розроблені конфігурації для цієї платформи. На платформі BAF розробляються та запускаються конфігурації. Розробка ведеться власною мовою високого рівня. Програмні продукти лінійки BAS – це сучасні рішення для автоматизації бізнес-процесів підприємств різних галузей та масштабів. Розроблені з урахуванням

українського законодавства та специфіки вітчизняного ринку, ці продукти забезпечують ефективне управління, облік і аналіз діяльності компаній.

В ході дослідження на прикладі BAS було детально розглянуто та опрацьовано структуру та принципи реалізації сучасних інформаційно-облікових систем. Це дозволило зробити детальні висновки стосовно того, які функціональні можливості програми мають бути реалізованими, щоб забезпечити потреби кінцевого користувача та які саме принципи реалізації інтерфейсу користувача дозволять зробити програмний продукт максимально зрозумілий та легкий у освоєнні. Також, робота з платформою BAF та дослідження усіх її можливостей дозволило зробити висновки стосовно того, яку структуру програмного продукту варто наслідувати у подальшій розробці інформаційно-облікових систем для того, щоб уможливити якнайкращу продуктивність та швидкодію програми.

Підсумовуючи основу структури інформаційної системи, можна виділити такі основні її об’єкти та їх взаємодію:

- Константи. Призначені для зберігання постійних та умовно-постійних даних.
- Довідники. Списки однорідних елементів даних. Використовуються для зберігання нормативно-довідкової інформації.
- Документи. Служать для зберігання первинної інформації про господарську операцію і для реалізації логіки впливу цієї операції на облік.
- Перелічення. Списки значень, що задаються на етапі конфігурування.
- Звіти. Використовуються для отримання вихідної інформації.
- Обробки. Використовуються для програмного виконання різноманітних дій над інформаційною базою.
- Регістри відомостей. Служать для зберігання інформації, склад якої є розгорнутим за певною комбінацією значень. За потреби, така інформація може бути розгорнута ще й у часі.

- Регістри накопичення. Служать для накопичення інформації у розрізі вимірів з можливістю отримання залишків та / або оборотів числових показників.
- Реквізити - інформація про об'єкт, доступна лише в межах цього об'єкта. Можна сказати, що реквізитами визначаються додаткові властивості об'єкта;
- Табличні частини інформація про об'єкт, що має спискову сутність;
- Реквізити табличних частин склад полів рядка табличної частини об'єкта, доступні лише у межах такої табличної частини об'єкта;
- Форми - використовуються для введення, перегляду та редагування даних;
- Команди - використовуються для реалізації дій над даними, що визначені об'єктом;
- Макети - призначені для створення друкованих форм об'єкта;

Варто також розглянути концепцію функціонування інформаційної системи. У програмах автоматизації бізнесу використовується принцип обліку від документа. Тобто, діяльність організації обліковується на рівні окремих елементарних операцій. Під кожну операцію створюється об'єкт Документ.

Рис. 1.1. Схема роботи інформаційно-облікової системи

Документами до системи вноситься первинна інформація про господарські операції що вже відбулися. Здебільшого документи, що створюються в процесі налаштування конфігурації, є електронними аналогами стандартних паперових документів, однак використання цього типу даних може виходити далеко за межі простої фіксації інформації про господарські операції. Дата і час найважливіші характеристики документа, оскільки дають змогу встановлювати сувору часову послідовність здійснення операцій. Документ може мати будь-яку кількість реквізитів та табличних частин.

Під час заповнення документів, використовується додаткова інформація з довідників. Для роботи з списком однорідних значень у системі використовуються довідники. Зазвичай довідниками є списки матеріалів, товарів, організацій, валют, співробітників тощо. Довідники можуть бути

ієрархічними (реалізовано два різновиди ієрархії) та підлеглими. Під час створення довідника можна визначити склад реквізитів, табличних частин, реквізитів табличних частин та виконати інші налаштування.

Інформація з документів потрапляє до облікових об'єктів регістрів. Дані в регістрах можуть бути скориговані різними механізмами: регламентними документами або обробками. Головне призначення регістру - зберігати суттєву для прикладного рішення інформацію, склад якої є розгорнутий за якоюсь комбінацією вимірювань. Додатково можливо розгорнути вміст регістру у часі. Ця інформація зберігається у регістрі як записи. У регістрі може бути лише один запис із певною комбінацією вимірювань та періодом. Регістри, інформація у яких розгорнуто у часі, називають періодичними. Регістр може характеризуватися вибраним режимом запису: незалежний, або підпорядкування реєстратору. З регістрів облікові дані потрапляють у звіти.

1.3. Обґрунтування вибору інструментарію вирішення проблеми

Для реалізації програмної частини інформаційно-облікової системи автотранспортних перевезень було обрано такі основні інструменти: мова програмування C#, середовище розробки Visual Studio, технологія побудови графічного інтерфейсу WPF та система управління базами даних SQLite. Такий вибір зумовлений низкою технічних, функціональних і практичних переваг цих засобів у контексті вирішення завдань розробки програмного продукту.

C# – це сучасна, об'єктно-орієнтована мова програмування, розроблена корпорацією Microsoft як частина платформи .NET. Її відзначає висока виразність, строгість типів, зручна система роботи з даними, підтримка сучасних парадигм (наприклад, LINQ, async/await), а також широка екосистема бібліотек і компонентів. Завдяки тісній інтеграції з .NET та Windows API, C# є одним із найкращих варіантів для створення настільних застосунків під Windows.[1]

На відміну від таких мов, як Python чи JavaScript, C# забезпечує вищу продуктивність, кращу інтеграцію з нативними можливостями ОС Windows, а також надає розвинуті засоби для створення типобезпечних, стабільних рішень. У порівнянні з Java, C# простіший у створенні графічного інтерфейсу, а робота з базами даних у ньому є менш обтяжливою завдяки бібліотекам ADO.NET, Entity Framework тощо.

Visual Studio – це одне з найпотужніших інтегрованих середовищ розробки програмного забезпечення, яке підтримує повний цикл створення застосунків: від написання коду до налагодження, тестування та розгортання. Завдяки широкому спектру інструментів (інтелектуальне автозаповнення, зручний відлагоджувач, підтримка дизайнерів форм, аналізу продуктивності та інтеграції з системами контролю версій) Visual Studio значно спрощує розробку великих і середніх проєктів.[4]

У порівнянні з альтернативними IDE, такими як JetBrains Rider чи Eclipse, Visual Studio має кращу підтримку для WPF, більш глибоку інтеграцію з платформою Windows та безкоштовну Community-версію для освітніх і малих комерційних проєктів.

Windows Presentation Foundation (WPF) – це технологія для створення настільних додатків на платформі Windows, що підтримує побудову багатофункціонального, стильного і гнучкого графічного інтерфейсу. Завдяки декларативному підходу на основі мови розмітки XAML, WPF дозволяє ефективно розділяти логіку програми (C#) від візуального оформлення, що сприяє реалізації шаблонів проектування, таких як MVVM (Model-View-ViewModel).[2]

У порівнянні з Windows Forms, яка була поширеним підходом у попередніх поколіннях .NET, WPF забезпечує кращу підтримку масштабування, стилізації інтерфейсу, анімацій, а також роботу з мультимедіа і 2D-графікою. Windows Forms хоч і простіша в освоєнні, однак обмежена у створенні сучасного, адаптивного та гнучкого UI.

Іншим сучасним аналогом WPF є платформа Electron, яка дозволяє створювати кросплатформенні настільні додатки на основі веб-технологій (HTML/CSS/JavaScript). Однак Electron-застосунки є значно важчими, споживають більше системних ресурсів, а їх продуктивність часто поступається нативним .NET-додаткам, що є критичним для офлайн-застосунків із великою кількістю облікових операцій.

Для збереження і обробки даних у системі було обрано СУБД SQLite – легку, вбудовану базу даних, яка ідеально підходить для настільних програм. Основними причинами вибору стали:

- відсутність потреби в окремому сервері баз даних;
- простота налаштування та використання;
- висока продуктивність при роботі з локальними даними;
- зберігання всієї бази у вигляді одного файлу;
- проста та надійна інтеграція з C#.

На відміну від серверних рішень, таких як MySQL, PostgreSQL або Microsoft SQL Server, які потребують складнішої інфраструктури, обслуговування та адміністрування, SQLite не вимагає встановлення сервера, що значно полегшує розгортання застосунку на кінцевих пристроях користувачів. Також, порівняно з XML-файлами, текстовими файлами або JSON-структурами, SQLite забезпечує надійнішу структуру даних, підтримує транзакції, індексацію, запити SQL і забезпечує більшу стабільність збереження інформації.[5]

У межах підвищення безпеки даних, які зберігаються локально, доцільно використовувати SQLCipher – розширення до SQLite, яке забезпечує прозоре шифрування всієї бази даних. SQLCipher підтримує повне 256-бітове шифрування на основі алгоритму AES (Advanced Encryption Standard), що відповідає сучасним вимогам безпеки при зберіганні конфіденційної інформації.[6] [14] [15]

Однією з ключових переваг SQLCipher є те, що він зберігає сумісність із SQLite, тобто більшість команд SQL та механізмів роботи з даними залишаються незмінними. Це дозволяє без суттєвих змін у структурі коду забезпечити шифрування на рівні всієї БД, включно з даними, метаданими, журналами транзакцій тощо.

SQLCipher особливо корисний у тих випадках, коли програма зберігає дані локально – як, наприклад, у настільних офлайн-системах, де існує ризик несанкціонованого доступу до файлу бази даних через фізичний доступ до комп'ютера. У контексті інформаційно-облікової системи автотранспортних перевезень, SQLCipher дозволяє гарантувати захист даних про фінанси, замовників, маршрути та іншу чутливу інформацію без необхідності розгортання окремого серверного середовища.

Ще одним альтернативним варіантом для невеликих локальних застосунків є Microsoft Access, але він не підтримує кросплатформенність, має обмеження щодо масштабування, а також є менш придатним для інтеграції з сучасним стеком C#/.NET.

Комбінація C#, WPF, SQLite і Visual Studio дозволяє створити ефективну, продуктивну та адаптивну інформаційно-облікову систему, яка відповідає сучасним вимогам до зручності використання, надійності збереження даних та гнучкості інтерфейсу. Обрані інструменти взаємодіють між собою на рівні платформи .NET, що забезпечує стабільність, безпеку і легкість підтримки проєкту в майбутньому.

РОЗДІЛ 2. ОПИС МОДЕЛІ ТА МЕТОДІВ ЇЇ РЕАЛІЗАЦІЇ

2.1. Формалізована постановка задачі дослідження

У сучасних умовах цифрової трансформації підприємницької діяльності ефективність функціонування автотранспортних підприємств значною мірою залежить від якості облікових процесів, автоматизації управління ресурсами та вчасного доступу до достовірної інформації. Ручне ведення документації, застосування неуніфікованих форм або застарілих засобів обліку створюють значні перешкоди для прийняття оперативних управлінських рішень, обслуговування клієнтів, оптимізації витрат і дотримання логістичних термінів.

Таким чином, проблема, яку ставить перед собою дана робота, полягає у створенні ефективної, надійної та функціонально повної інформаційно-облікової системи для управління автотранспортними перевезеннями, яка забезпечить централізоване зберігання, обробку та подання інформації про ключові ресурси і процеси діяльності автопідприємства. Основною метою дослідження є розробка архітектури програмного рішення та реалізація застосунку, що дозволяє автоматизувати облік транспорту, водіїв, маршрутів, витрат паливно-мастильних матеріалів, а також організувати формування звітності й контроль доступу до системи.

Під час формалізації проблеми було визначено головні структурні компоненти системи та їх функціональні взаємозв’язки. Застосунок повинен реалізувати такі базові підсистеми:

- Управління автопарком передбачає ведення обліку транспортних засобів, їхніх технічних характеристик, стану, належності до підрозділів, а також ведення історії обслуговування, техоглядів та ремонтів.
- Облік водіїв включає фіксацію персональних даних працівників, їх графіків роботи, медичних допусків, водійських категорій та закріплення за конкретним транспортом або маршрутом.

- Управління маршрутами забезпечує створення і редагування маршрутів перевезень, контроль точок посадки/висадки або завантаження/розвантаження, а також розрахунок відстаней, часу в дорозі і навантаження на ресурси.
- Облік паливно-мастильних матеріалів (ПММ) відповідає за фіксацію витрат пального на маршрутах, ведення залишків по автомобілях, контроль за нормами споживання та виявлення можливих відхилень.
- Звітність реалізується через генерацію документів за обраний період – це можуть бути звіти по витратах, по пробігу, по маршрутах, технічному стану транспорту, а також звіти для аналізу ефективності логістичних процесів.
- Аутентифікація та управління ролями користувачів – необхідний функціонал для розмежування рівнів доступу до даних системи. Наприклад, водій може мати доступ лише до інформації про власні рейси, тоді як диспетчер – до маршрутів, транспорту і графіків, а адміністратор – до повної системної інформації.

При постановці задачі дослідження важливим аспектом є визначення обмежень і умов, які впливають на архітектуру рішення. Серед таких: відсутність необхідності в серверному середовищі (тобто, обрана модель передбачає локальне розгортання), потреба у мінімальних системних вимогах, відсутність залежності від стороннього платного програмного забезпечення, простота оновлення та масштабування у майбутньому.

На основі цих вимог сформульовано концептуальну модель вирішення задачі, яка охоплює такі етапи:

- Аналіз структури підприємства та його облікових потреб у контексті перевезень.
- Проєктування бази даних з урахуванням взаємозв’язків між транспортом, водіями, маршрутами та витратами.
- Розробка користувацького інтерфейсу із застосуванням WPF, з акцентом на інтуїтивну навігацію та візуальне групування функцій.

- Реалізація внутрішньої логіки системи засобами C#, включно з валідацією, фільтрацією даних, авторизацією і формуванням звітів.
- Інтеграція з базою даних SQLite, а за потреби – реалізація шифрування на базі SQLCipher.
- Тестування, перевірка на коректність введення даних та стабільність функціонування всіх підсистем.

Загалом, модель ґрунтується на принципах централізації даних, автоматизації рутинних дій, підвищення точності обліку та спрощення управління логістичними ресурсами, що дозволяє підприємству ефективніше розпоряджатися своїм автопарком, зменшувати витрати, знижувати ризики людських помилок.

2.2. Опис інформаційної моделі

У рамках побудови інформаційно-облікової системи для автотранспортних перевезень основною метою моделювання є відображення об’єкта дослідження в уніфікованій формі, що дозволяє здійснити комп’ютерну реалізацію функціональних зв’язків між обліковими елементами: транспортними засобами, водіями, маршрутами, вантажами, паливно-мастильними матеріалами (ПММ) та користувачами системи.

Аналітична модель, що використовується в цій роботі, має інформаційно-реляційний характер, тобто побудована на основі реляційної структури бази даних, де кожен об’єкт реального середовища – це сутність, яка має атрибути та зв’язки з іншими сутностями. Такий тип моделі є найбільш придатним у контексті задачі, оскільки забезпечує чітке структурування даних, можливість побудови складних запитів, підтримку цілісності інформації та масштабованість рішення. Обрана модель дозволяє не лише формалізувати предметну область, але й створити основу для автоматизації дій користувача – введення, редагування, фільтрації, пошуку та аналітики.

На рисунку 2.1 наведено діаграму структури бази даних, яка відображає взаємозв’язки між основними таблицями системи, що дозволяє візуально представити логіку облікових операцій та залежностей між даними.

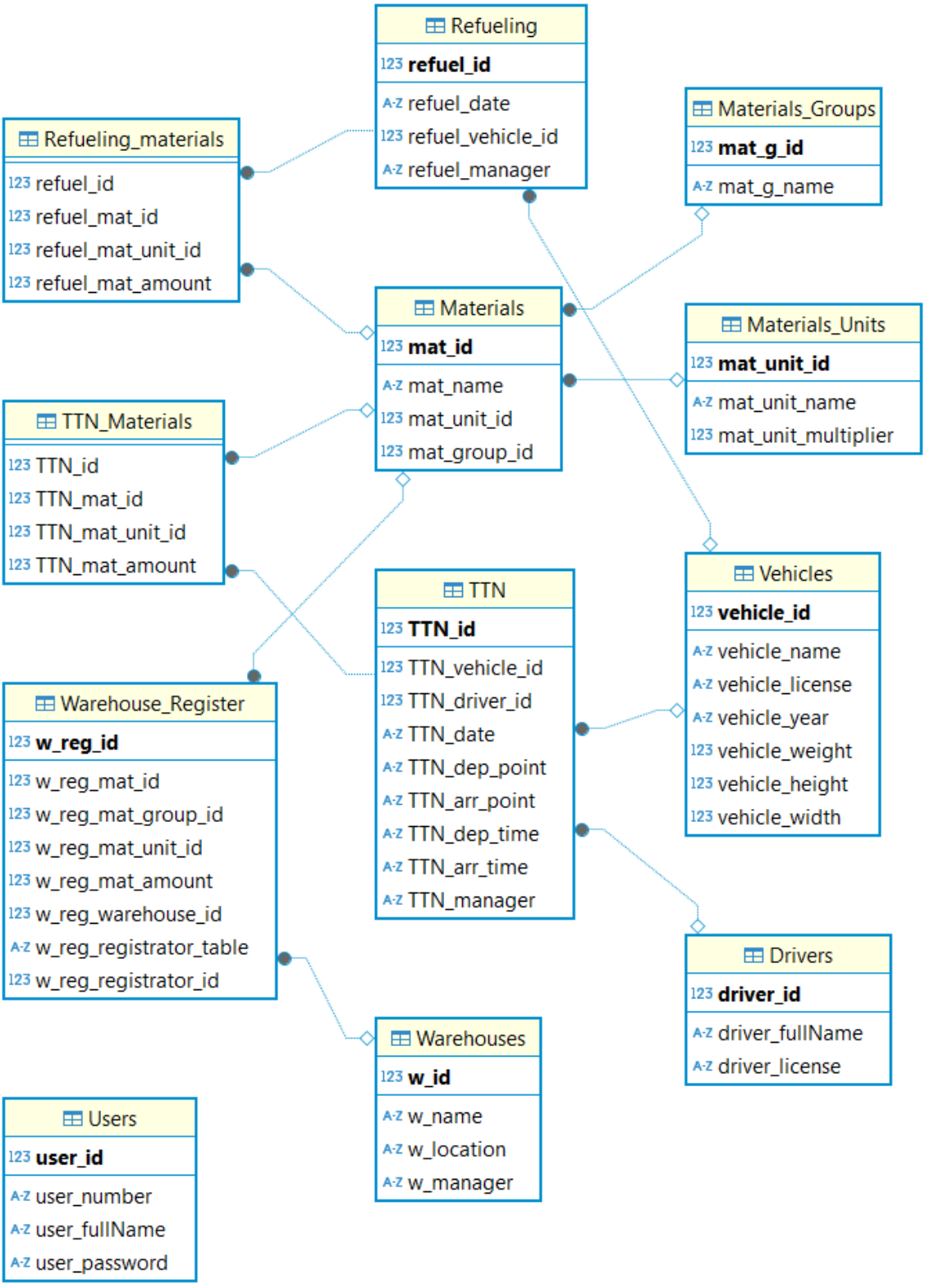

Рис. 2.1 Діаграма бази даних

Використаний тип моделі – структурна інформаційна модель у формі реляційної схеми – є оптимальним для задачі автоматизації обліку на підприємстві. По-перше, він дозволяє відобразити складну багаторівневу структуру даних у компактній, логічно впорядкованій формі. По-друге, реляційна модель є стандартизованою та підтримується широким спектром сучасних систем керування базами даних, включаючи обрану SQLite. По-третє, така модель безпосередньо адаптується до механізмів SQL-запитів, що забезпечує високу швидкість пошуку, фільтрації та формування звітів.

На відміну від ієрархічних або об'єктно-орієнтованих моделей, реляційна схема дозволяє ефективно реалізувати зв’язки типу «один до багатьох» (наприклад, один водій може мати багато рейсів, один транспорт – багато заправок), забезпечити цілісність даних через зовнішні ключі та уникнути дублювання.

Інформаційна модель охоплює основні функціональні блоки системи, які відображені через таблиці бази даних:

1. Облік транспорту реалізується через таблицю Vehicles, де кожен запис містить дані про транспортний засіб: назву, номер, рік випуску, масу, висоту, ширину тощо. Ці параметри є важливими для планування навантаження, маршруту та обмежень руху.
2. Облік водіїв ведеться у таблиці Drivers. Тут зберігається ПІБ водія, номер водійського посвідчення, а також унікальний ідентифікатор. Ці дані використовуються в документах перевезення (наприклад, ТТН), а також для контролю відповідності водія до рейсу.
3. Маршрути перевезень та їх фактичне виконання відображені в таблиці TTN (товарно-транспортна накладна). Тут містяться дані про дату, пункти відправлення та прибуття, водія, транспортний засіб, час початку і завершення рейсу та відповідального менеджера. Додатково, таблиця

TTN_Materials відображає номенклатуру вантажу, що перевозився, його одиниці виміру та кількість.

4. Облік ПММ (паливно-мастильних матеріалів) організовано через пов’язані таблиці Refueling та Refueling_materials. У першій фіксується факт заправки, дата, транспортний засіб, особа, яка здійснювала заправку. У другій – деталізація використаних матеріалів, їх кількість, одиниці виміру.
5. Матеріали систематизовано у таблиці Materials, а також у супутніх Materials_Units (одиниці виміру) та Materials_Groups (групи матеріалів). Це дозволяє підтримувати єдиний довідник ПММ та інших ресурсів, з нормалізацією структури даних.
6. Складський облік реалізований за допомогою таблиці Warehouse_Register, яка фіксує кількість ресурсів у межах певного складу (Warehouses). Зберігається інформація про матеріал, його групу, одиницю виміру, кількість, а також – хто саме здійснив реєстрацію змін.
7. Користувачі системи визначаються в таблиці Users, де кожен запис містить ідентифікатор, унікальний логін, ім’я та пароль. Це є основою для реалізації механізму аутентифікації та контролю ролей, який регламентує доступ до функціоналу (наприклад, диспетчер має ширші права, ніж водій).

Для керування інформаційною системою використовується застосунок на базі мови програмування C# та WPF. Для забезпечення роботи з таблицями розроблена відповідна структура форм. Програма реалізована у вигляді багатосторінкового застосунку з використанням шаблону MVVM, де кожна форма (сторінка або вікно) відповідає за окрему підсистему. Основне вікно – MainWindow – є контейнером для всіх інших сторінок та забезпечує навігацію між ними через головне меню або вкладки. Форма EnterWindow використовується для авторизації користувачів перед входом у систему. Вона перевіряє введені логін і пароль, звертаючись до таблиці Users, та надає доступ до функціоналу згідно з роллю користувача. Кожна пара ListPage та EditPage реалізує патерн перегляду та редагування. Наприклад, список водіїв (DriverListPage) дозволяє обрати конкретного користувача та передати його

дані у EditDriverPage, де можна змінити або видалити запис. Такий підхід дозволяє ефективно працювати з великим обсягом облікових даних та підтримувати стандартизований, зручний для користувача інтерфейс. Всі форми об'єднані навігаційною логікою, яка реалізується через головне вікно програми. [3] [7] [11]

Таблиця 2.1 Структура форм застосунку

Блок	Сторінка (XAML)	Призначення
Облік водіїв	DriverListPage.xaml	Перегляд списку водіїв з таблиці Drivers
	EditDriverPage.xaml	Створення або редагування водія (ПІБ, посвідчення)
Облік транспорту	VehicleListPage.xaml	Перелік транспортних засобів із бази Vehicles
	EditVehiclePage.xaml	Додавання або редагування транспорту (назва, габарити, номер, рік)
Облік користувачів	UserListPage.xaml	Перелік користувачів системи
	EditUserPage.xaml	Додавання або зміна користувача (ПІБ, номер, пароль)
Довідники	DirectoriesPage.xaml	Хаб для доступу до довідників
	MaterialsListPage.xaml	Перелік матеріалів (напр. ПММ)
	EditMaterialsPage.xaml	Редагування матеріалів
	Materials_UnitListPage.xaml	Перелік одиниць виміру (літри, кг тощо)
	EditMaterials_UnitPage.xaml	Редагування одиниць виміру
	Materials_GroupListPage.xaml	Перелік груп матеріалів (паливо, мастила)
	EditMaterials_GroupPage.xaml	Редагування груп матеріалів
Склади	WarehouseListPage.xaml	Список складів
	EditWarehousePage.xaml	Редагування даних про склад (місцезнаходження, відповідальний)
	RegistersPage.xaml	Облік надходження/списання матеріалів (табл. Warehouse_Register)

Документи	DocumentsPage.xaml	Доступ до обліку первинної документації
	Вкладка ТТН	Дані з TTN і TTN_Materials (вантажі)
	Вкладка Заправки	Дані з Refueling і Refueling_Materials (історія заправок)
Звіти та аналітика	ReportsPage.xaml	Побудова звітів з фільтрами, періодами, експортом (Excel, PDF)

2.3. Вибір методів, технологій та розробка алгоритмів реалізації моделі

Для розробки програмного забезпечення, яке б відповідало поставленим вимогам важливим є не лише вибір правильних інструментів розробки, а й архітектурного підходу, який дозволить забезпечити масштабованість, підтримуваність і розділення відповідальностей у програмному коді.У рамках цієї системи було прийнято рішення реалізувати програму на базі патерну MVVM (Model–View–ViewModel), який найбільш повно відповідає специфіці розробки додатків з використанням WPF (Windows Presentation Foundation) і є рекомендованим стандартом для програм з багатим графічним інтерфейсом користувача. [3] [7] [11]

MVVM є архітектурним шаблоном, який дозволяє чітко розділити логіку відображення (View), логіку представлення та взаємодії (ViewModel) та модель даних (Model). Така структура дозволяє суттєво спростити супровід, тестування та подальшу модернізацію програми, оскільки зміни в одному шарі практично не впливають на інші.

View є інтерфейсною частиною додатка. У контексті WPF вона реалізується за допомогою XAML-розмітки, що описує зовнішній вигляд елементів управління, структуру макету, шаблони стилів, компонування сторінок тощо. View не містить бізнес-логіки, і її основна роль – відображати дані, отримані з ViewModel.

ViewModel виступає посередником між View і Model. У цьому шарі реалізовано логіку керування станом інтерфейсу, обробку команд користувача,

прив’язку властивостей, реалізацію фільтрації, сортування, навігації, а також логіку взаємодії з моделлю даних. ViewModel забезпечує повноцінну підтримку патерну «двостороннього прив’язування», завдяки якому зміни в UI автоматично оновлюють стан внутрішніх змінних, і навпаки.

Model представляє об’єкти предметної області (наприклад, транспорт, водій, заправка) і включає структуру даних, сервіси з доступу до БД, серіалізацію, перевірку цілісності інформації та інші операції, що не пов’язані з представленням.

Центральним принципом MVVM є зв’язування даних, яке реалізується в WPF завдяки механізмам XAML і системі сповіщення про зміну властивостей. Це дозволяє змінювати дані в режимі реального часу, не втручаючись у код візуального інтерфейсу. Наприклад, зміна значення в полі введення кількості ПММ одразу відображається в ViewModel, а потім передається в базу даних без необхідності додаткового коду у View. [3] [8]

У межах даного проєкту шаблон MVVM дозволив ефективно розподілити логіку додатка між окремими шарами. Кожна форма програми має власний ViewModel, який забезпечує зв'язок з моделями

Для прикладу, при відкритті списку водіїв, View (DriverListPage.xaml) прив’язується до ViewModel (DriverListViewModel.cs), який, у свою чергу, викликає відповідний метод із шару Model для отримання списку водіїв з бази даних. Якщо користувач вибирає запис для редагування – це ініціює команду, яка створює інстанцію EditDriverViewModel та відкриває відповідну View для редагування. Такі переходи та обробка команд реалізовані через ICommand, що дозволяє легко керувати діями користувача в View без прямого втручання в код інтерфейсу.

Крім того, завдяки MVVM з’явилася можливість повторного використання компонентів. Наприклад, модуль авторизації (EnterWindow.xaml) використовує власний ViewModel для обробки введення логіна і пароля, що дозволяє при потребі легко замінити візуальну частину, не змінюючи логіку перевірки.

Такий підхід особливо ефективний для розробки складних форм звітності, де використовується фільтрація даних за періодами, типами ресурсів, транспортними засобами. За рахунок логіки, винесеної у ViewModel, можна легко оновлювати вміст таблиць, експортувати дані, змінювати формат представлення без впливу на базову структуру моделі.

На основі сторінок, які пов’язані з налаштуванням списку користувачів, розглянемо принцип роботи форм. У модулі управління користувачами реалізовано повний цикл операцій: від перегляду списку користувачів, створення нових облікових записів, до редагування та видалення наявних. Усі ці дії виконуються в межах чітко розділених рівнів: View, ViewModel та Model, що є класичним прикладом патерну MVVM. [3]

Модель User є простою сутністю, яка відображає структуру таблиці Users у базі даних. Вона містить такі властивості: Id – унікальний ідентифікатор, Number – умовний логін чи службовий номер, Name – ім’я користувача, та Password – пароль. Ця модель виступає базовим контейнером для передачі даних між шарами програми.

Лістинг коду User.cs

```
public class User
{
   public int Id { get; set; }
   public string Number {  get; set; }
   public string Name { get; set; }
   public string Password { get; set; }
}
```

Інтерфейс перегляду користувачів представлений у вигляді WPF-сторінки, яка реалізує дві головні частини: панель керування та таблицю зі списком користувачів. У верхній частині сторінки розміщено кнопки: «Додати», «Видалити» та «Оновити список», кожна з яких прив’язана до відповідної команди у ViewModel. Нижче відображається список користувачів через компонент ListView, дані якого прив’язані до властивості Users у

ViewModel. Обраний елемент зі списку передається у SelectedUser, який, у свою чергу, використовується для виконання операцій редагування або видалення.

Кнопка «Видалити» активується лише тоді, коли обраний певний користувач – це реалізовано через прив’язку до SelectedUser з використанням конвертера NullToBoolConverter, який перетворює значення null у логічне false.

Лістинг коду UserListPage.xaml (частина):

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
   <Button Content="Додати"
   Command="{Binding AddUserCommand}"
   Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />

   <Button Content="Видалити"
   Command="{Binding DeleteUserCommand}"
   IsEnabled="{Binding SelectedUser, Converter={StaticResource NullToBoolConverter}}"
   Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />

   <Button Content="Оновити список"
   Command="{Binding UpdateUsersList}"
   Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />
</StackPanel>
<ListView ItemsSource="{Binding Users}"
        SelectedItem="{Binding SelectedUser}"
        MouseDoubleClick="ListView_MouseDoubleClick"
        Margin="0" Grid.Row="1">
   <ListView.View>
      <GridView>
         <GridViewColumn Header="Номер" DisplayMemberBinding="{Binding Id}" Width="50"/>
```

```
        <GridViewColumn Header="Ім’я" DisplayMemberBinding="{Binding Name}" Width="150"/>
        <!--<GridViewColumn Header="Email" DisplayMemberBinding="{Binding Email}" Width="200"/>-->
      </GridView>
   </ListView.View>
</ListView>
```

Лістинг коду UserListPage.xaml.cs

```
public UserListPage()
{
   InitializeComponent();
}
private void ListView_MouseDoubleClick(object sender, MouseButtonEventArgs e)
{
   if (sender is ListView listView && listView.SelectedItem is User selectedUser)
   {
      if (DataContext is UserListViewModel vm && vm.EditUserCommand.CanExecute(selectedUser))
      {
         vm.EditUserCommand.Execute(selectedUser);
      }
   }
}
```

ViewModel UserListViewModel керує станом і логікою перегляду та управління списком користувачів. У ньому визначено властивості Users (список користувачів) і SelectedUser (обраний користувач), а також команди AddUserCommand, DeleteUserCommand, UpdateUsersList і EditUserCommand.

Усі команди реалізовані через клас RelayCommand, який дозволяє реагувати на взаємодію з інтерфейсом.

Метод LoadUsersFromDatabase відповідає за підключення до локальної бази даних SQLite, виконання SQL-запиту SELECT і заповнення колекції користувачів. Завдяки використанню ObservableCollection, інтерфейс автоматично оновлюється при зміні списку без необхідності прямої взаємодії з елементами керування. [5]

Операція додавання користувача ініціює відкриття нової вкладки за допомогою _tabHost, де створюється порожній об’єкт User, передається у EditUserViewModel, і запускається EditUserPage.xaml. Аналогічно працює редагування, але з передачею вже існуючого об’єкта.

Видалення реалізовано без підтвердження (що потенційно можна вдосконалити) – шляхом виконання SQL-запиту DELETE до бази даних із використанням ідентифікатора обраного користувача.

Лістинг коду UserListViewModel:

```
private readonly ITabHost _tabHost;
 public ObservableCollection<User> Users { get; set; } = new();

 private User _selectedUser;
 public User SelectedUser
 {
    get => _selectedUser;
    set { _selectedUser = value; OnPropertyChanged(nameof(SelectedUser)); }
 }

private void EditUser(User user)
 {
    if (user == null) return;
    var editVM = new EditUserViewModel(user, _tabHost, false);
    var editPage = new EditUserPage { DataContext = editVM };
```

```
    var tab = new TabItemViewModel
    {
        Header = $"Редагування: {user.Name}",
        Content = editPage
    };
    _tabHost.AddTab(tab);
}

private void LoadUsersFromDatabase(object parameter)
{
    string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
"DiplomaDB.db");
    var connection = new SqliteConnection($"Data Source={path}");
    connection.Open();
    var command = connection.CreateCommand();
    command.CommandText = "SELECT user_id, user_fullName FROM Users";
    using var reader = command.ExecuteReader();

private void DeleteUser(object _)
{
   if (SelectedUser == null) return;
   string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
"DiplomaDB.db");
   var connection = new SqliteConnection($"Data Source={path}");
   connection.Open();
       var command = connection.CreateCommand();
       command.CommandText = @"
       DELETE FROM Users
       WHERE user_id = $id;";
       command.Parameters.AddWithValue("$id", SelectedUser.Id);
```

command.ExecuteNonQuery();

Форма редагування містить поля для введення імені користувача та пароля, а також кнопку «Зберегти». Дані з полів TextBox прив’язано до властивостей моделі EditingUser.Name і EditingUser.Password, які обробляються у відповідному ViewModel.

Особливу увагу заслуговує кнопка збереження: її параметри формуються через MultiBinding, який за допомогою MultiParameterConverter об'єднує значення з кількох полів у масив. Це дозволяє передати кілька значень до команди одним параметром.

Лістинг коду UserEditPage.xaml

```
<UserControl.Resources>
   <models:MultiParameterConverter x:Key="MultiParamConverter"/>
</UserControl.Resources>
Label Grid.Column="0" Grid.Row="0" Margin="0">Ім'я користувача: </Label>
      <Label Grid.Column="0" Grid.Row="1" Margin="0">Пароль: </Label>
      <TextBox x:Name="UserName" Grid.Column="1" Grid.Row="0" Margin="5" Text="{Binding EditingUser.Name}" />
      <TextBox x:Name="UserPassword" Grid.Column="1" Grid.Row="1" Margin="5" Text="{Binding EditingUser.Password}"/>
      <Button Content="Зберегти" Command="{Binding SaveCommand}" Grid.Column="0" Grid.Row="2" Margin="5">
         <Button.CommandParameter>
            <MultiBinding Converter="{StaticResource MultiParamConverter}">
               <Binding ElementName = "UserName" Path="Text"/>
               <Binding ElementName = "UserPassword" Path="Text"/>
            </MultiBinding>
         </Button.CommandParameter>
      </Button>
```

ViewModel для редагування користувача управляє станом поточно редагованого запису (EditingUser) та містить команду SaveCommand. Відповідно до режиму (новий запис або редагування існуючого), ViewModel виконує або INSERT, або UPDATE у таблицю Users. Після успішного збереження ViewModel закриває поточну вкладку, викликавши метод _tabHost.CloseTab().

Реалізація SaveUser демонструє безпосередню роботу з базою SQLite через клас SqliteConnection, включає використання параметризованих SQL-запитів для уникнення SQL-ін’єкцій і дотримується принципів обробки ресурсів (відкриття-закриття з’єднання в одному методі).

```
private readonly ITabHost _tabHost;
public User EditingUser { get; set; }
public ICommand SaveCommand => new RelayCommand(SaveUser);
public void SaveUser(object parameter)
 {
    string userName = "";
    string userPassword = "";
    if (parameter is object[] values)
    {
       userName = values[0] as string;
       userPassword = values[1] as string;
    }
    string path = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "DiplomaDB.db");
    var connection = new SqliteConnection($"Data Source={path}");
    connection.Open();
    if (_isNew)
    {
       currentTabName = "Новий користувач";
       var command = connection.CreateCommand();
```

```
        command.CommandText = @"
        INSERT INTO Users (user_fullName, user_password)
        VALUES ($name, $password);";
        command.Parameters.AddWithValue("$name", userName);
        command.Parameters.AddWithValue("$password", userPassword);
        command.ExecuteNonQuery();
    }
    else
    {
        var command = connection.CreateCommand();
        command.CommandText = @"
        UPDATE Users
        SET user_fullName = $name,
        user_password = $password
        WHERE user_id = $id;";
        command.Parameters.AddWithValue("$name", userName);
        command.Parameters.AddWithValue("$password", userPassword);
        command.Parameters.AddWithValue("$id", EditingUser.Id);

        command.ExecuteNonQuery();
    }
            connection.Close();

    var currentTab = _tabHost.GetTabItem(currentTabName);
    if(currentTab != null)
        _tabHost.CloseTab(currentTab);
}
```

Інтерфейс ITabHost виконує роль універсального диспетчера вкладок у головному вікні програми, де організовано простір користувача у вигляді вкладок. Такий підхід дозволяє додавати, активувати, шукати та закривати

вкладки під час роботи програми, не перевантажуючи інтерфейс і не створюючи зайвих вікон.

У системі, що використовує патерн MVVM, важливо, щоб взаємодія між логікою та представленням була опосередкованою, тобто ViewModel не повинна прямо знати про реалізацію вкладок. Саме для цього створено абстракцію – інтерфейс ITabHost, яка дозволяє ViewModel взаємодіяти з UI у контрольований і тестований спосіб.

Інтерфейс складається з чотирьох методів, кожен з яких реалізує окрему функцію:

1. void AddTab(object parameter). Цей метод може використовуватись для додавання вкладки за допомогою узагальненого параметра. Як правило, це може бути об'єкт типу Page, UserControl, ViewModel або навіть просто string, який інтерпретується в логіці реалізації. Такий підхід дозволяє залишити реалізацію відкритою для розширення: наприклад, у майбутньому можна додати підтримку вкладок без заголовка або вкладок без ViewModel.
2. void AddTab(TabItemViewModel tab). Це основний метод для додавання нової вкладки в інтерфейс, де параметр – це вже сформована модель вкладки (TabItemViewModel). Ця модель зазвичай містить такі властивості, як Header (заголовок вкладки), Content (вміст вкладки – зазвичай UserControl), та інші – наприклад, індикатори активності або ознаки змін. Саме цей метод активно використовується у ViewModel, таких як UserListViewModel, для відкриття сторінок редагування чи створення об'єктів.
3. TabItemViewModel GetTabItem(object parameter). Метод, який використовується для отримання вже відкритої вкладки за певним ідентифікатором або умовою. Зазвичай в якості параметра передається рядок (наприклад, назва вкладки) або ViewModel, за якою потрібно знайти вкладку. Це дозволяє уникати дублювання вкладок при багаторазовому виклику команди редагування одного й того самого об'єкта. Якщо вкладка знайдена, вона активується або повертається, інакше – повертається null.

4. void CloseTab(TabItemViewModel tab). Метод для закриття вкладки, яка вже була відкрита. Ця функція викликається, наприклад, після збереження користувача або при натисканні кнопки «Закрити». Закриття вкладки також сприяє очищенню ресурсів і зменшенню візуального навантаження в інтерфейсі.

Показаний фрагмент – це типовий приклад модульного й масштабованого підходу, де інтерфейс, логіка та дані розділені між собою, що суттєво полегшує тестування, рефакторинг та розширення функціоналу. Робота з вкладками через _tabHost дозволяє динамічно відкривати і закривати форми редагування, що зручно з точки зору користувацького досвіду. Завдяки MVVM, зв’язок між візуальним інтерфейсом і бізнес-логікою реалізований без прямої залежності, що відповідає сучасним підходам до розробки WPF-додатків.

РОЗДІЛ 3. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1. Опис інтерфейсу та функціональних можливостей програмної реалізації

У результаті дослідження було розроблено функціональний прототип програмного забезпечення – інформаційно-облікову систему автотранспортних перевезень, що реалізує запропоновану модель з урахуванням усіх етапів аналізу, проєктування та формалізації задачі. Створений програмний продукт являє собою настільний WPF-застосунок із підтримкою багатовіконного інтерфейсу на основі вкладок, що дозволяє одночасно працювати з кількома модулями без втрати контексту.

Інтерфейс програми побудований з урахуванням принципів зручності, модульності та доступності. Головне вікно застосунку (MainWindow) містить меню навігації та динамічну область вмісту, де відкриваються сторінки редагування, перегляду даних, створення документів та формування звітів. Завдяки механізму вкладок, реалізованому через інтерфейс ITabHost, користувач може паралельно відкривати декілька форм – наприклад, переглядати список водіїв та одночасно редагувати інформацію про окремого користувача або транспортний засіб. Це суттєво підвищує продуктивність при роботі з великим обсягом даних.

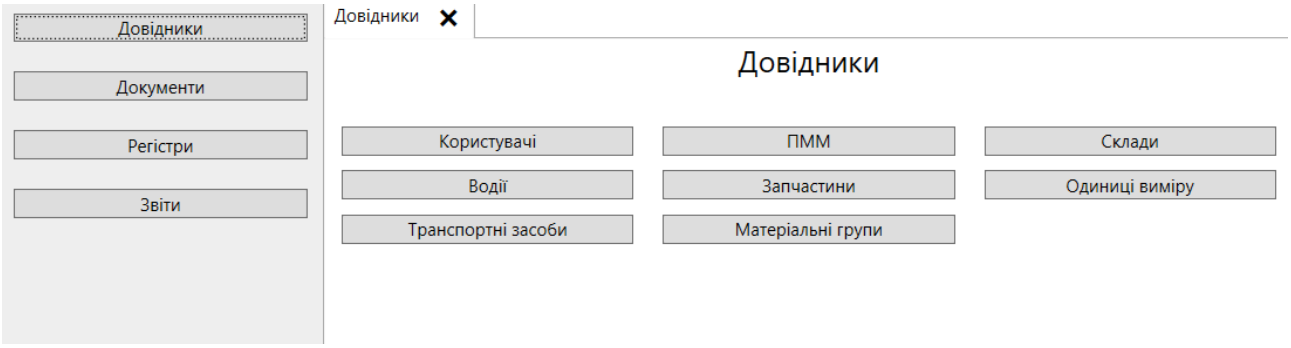

Рис. 3.1 Вигляд основного вікна програми з відкритою сторінкою довідників

Програма реалізує повний цикл взаємодії з ключовими об’єктами обліку. У модулі управління користувачами передбачено можливість створення нових облікових записів, редагування існуючих та видалення. Ці дії супроводжуються

відповідними SQL-запитами до локальної бази даних SQLite. Інтерфейс модуля включає список усіх користувачів з можливістю вибору та швидкого доступу до функцій редагування. Дані користувачів представлені у вигляді таблиці з можливістю подвійного кліку для переходу до редагування.

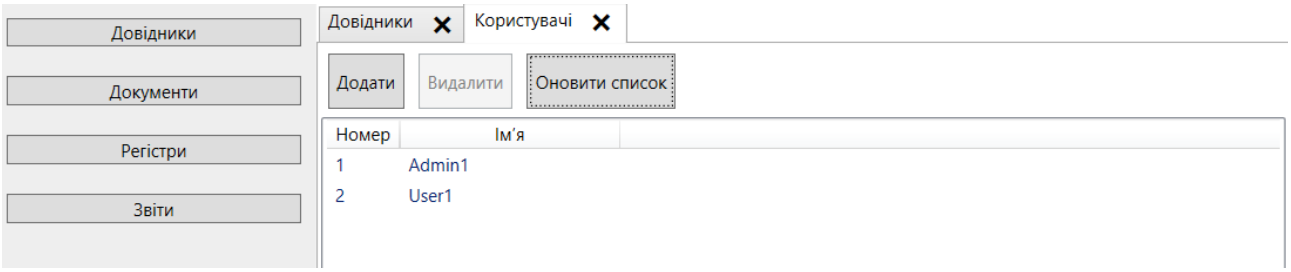

Рис. 3.2 Вигляд сторінки списку користувачів

Редагування користувача реалізоване у вигляді окремої форми (EditUserPage.xaml), де поля заповнюються з використанням двостороннього зв’язування (data binding) – це означає, що усі зміни, зроблені у формі, автоматично синхронізуються з ViewModel. Кнопка збереження реалізована з використанням RelayCommand і передає одразу кілька значень через MultiBinding, що дозволяє максимально спростити обробку дій користувача. Залежно від контексту – редагування або створення нового об'єкта – застосовується відповідно SQL-запит UPDATE або INSERT.

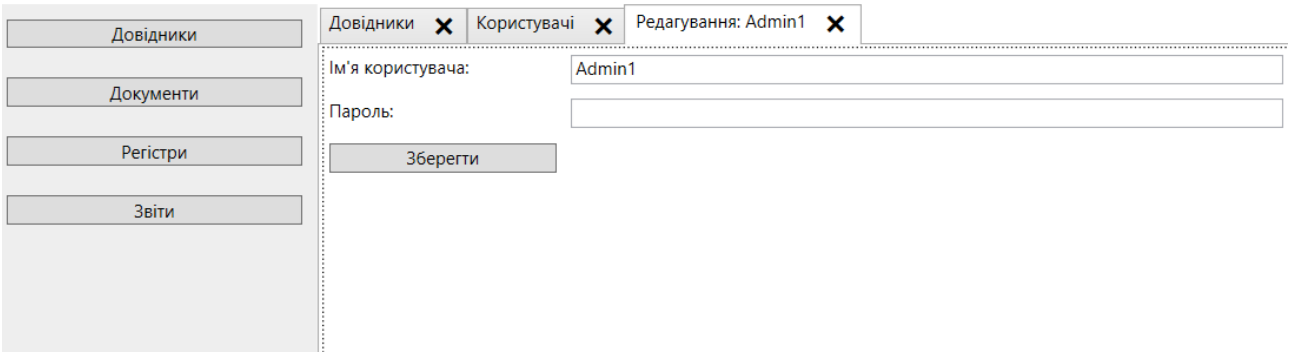

Рис. 3.3 Вигляд сторінки редагування користувача.

Крім модуля керування користувачами, у програмі реалізовано облік транспорту та облік водіїв. Відповідні сторінки містять списки об’єктів (транспортних засобів або водіїв) з можливістю швидкого перегляду та фільтрації. Також доступні форми редагування, в яких можна вказати технічні характеристики транспортного засобу або персональні дані водія. Таке

розділення дозволяє уникати дублювання інформації та підтримувати її актуальність.

Окрему увагу приділено модулю документів, у межах якого реалізується облік перевезень (товарно-транспортні накладні) та заправок. Користувач має змогу створювати нові записи, призначаючи водія, транспорт, дату та маршрут. Дані вантажу вносяться окремо, у структурі, що пов’язана з головним записом ТТН. Для заправок передбачена можливість реєстрації витрачених паливно-мастильних матеріалів із прив’язкою до транспортного засобу та відповідального менеджера.

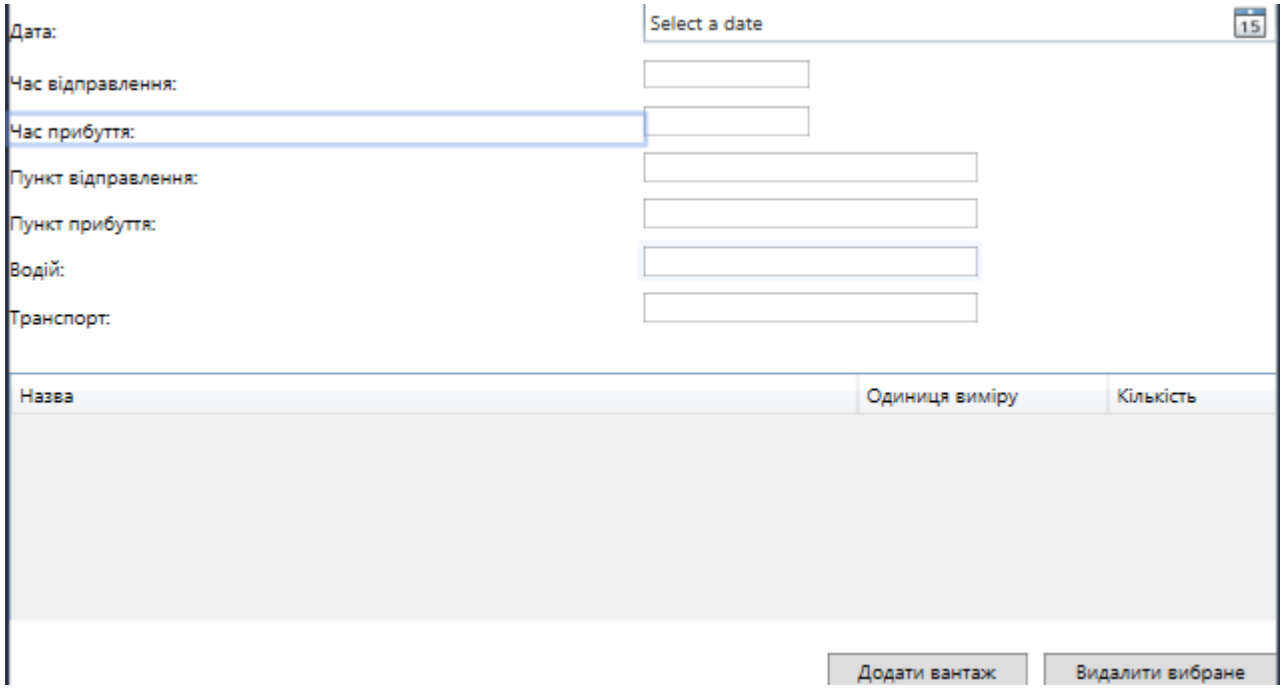

Рис. 3.4 Вигляд сторінки створення ТТН.

Для забезпечення контролю витрат та аналізу діяльності компанії реалізовано модуль звітності, в якому користувач може вибрати тип звіту, фільтрувати дані за періодом, транспортом, матеріалами або водієм. Звіти генеруються на основі запитів до бази даних і можуть бути експортовані в зручному форматі. Це дозволяє приймати обґрунтовані управлінські рішення на підставі фактичної інформації про витрати ПММ, виконані рейси, завантаження складів тощо.

Крім функціональних модулів, програма включає механізм аутентифікації, що запускається при старті системи. Введення логіна і пароля

перевіряється на відповідність з даними в таблиці Users. У разі успішної авторизації користувач отримує доступ до системи згідно зі своєю роллю.

Загалом, реалізований прототип є повністю працездатним з погляду обробки основних сценаріїв обліку. Він дозволяє протестувати ефективність реалізованих алгоритмів, перевірити коректність збереження та оновлення даних у базі, а також оцінити зручність користувацького інтерфейсу. Усі основні функції були перевірені під час ручного тестування, що підтвердило правильність реалізації взаємодії між інтерфейсом, логікою ViewModel і збереженням даних у SQLite.

Межі застосування прототипу на цьому етапі зумовлені відсутністю мережевої підтримки або багатокористувацького режиму. Проте завдяки модульній архітектурі, програму легко масштабувати – наприклад, шляхом інтеграції з серверною базою даних, розширенням ролей користувачів або додаванням нових звітів.

3.2. Тестові приклади роботи системи, аналіз результатів

Для підтвердження працездатності створеної інформаційно-облікової системи автотранспортних перевезень було проведено серію тестових прикладів, які моделюють реальні сценарії використання програми на підприємстві. Метою тестування було не лише перевірити правильність реалізації алгоритмів обробки даних, а й оцінити стабільність, логічну цілісність і відповідність інтерфейсу користувача очікуваному функціоналу.

Процес тестування охоплював основні підсистеми: роботу з користувачами, транспортними засобами, водіями, документацією перевезень, облік заправок і генерацію звітності. Кожен сценарій включав етап створення тестових записів, перевірку коректного відображення внесених даних, зміну певних параметрів і підтвердження результатів через візуальні компоненти інтерфейсу або звернення до бази даних.

На першому етапі було змодельовано створення нового користувача. Через форму авторизації програма запитувала вхідні дані, які в разі успішного введення дозволяли перейти до основного вікна системи. Після входу адміністратор відкрив вкладку «Користувачі», натиснув кнопку «Додати» та вніс тестове ім’я й пароль. Натискання кнопки «Зберегти» ініціювало виконання SQL-запиту INSERT, після чого новий користувач миттєво з’явився у списку. Повторний запуск програми підтвердив наявність нового запису в базі, що засвідчило правильність логіки збереження.

Далі було протестовано створення об’єкта «водій». У відповідній формі було заповнено ім’я та номер посвідчення, а також призначено транспортний засіб. Після збереження водій з’явився в таблиці водіїв, а під час створення нової товарно-транспортної накладної (ТТН) – відображався у випадаючому списку для призначення рейсу. Це підтвердило цілісність зв’язків між таблицями та коректне формування довідників на основі даних.

Створення ТТН стало одним із ключових прикладів комплексного тестування. Була ініційована форма додавання документа, в якій вручну вибиралися дата, маршрут, водій, транспорт і список вантажів. Після підтвердження збереження програма додавала запис до таблиці TTN, а деталі вантажу – до таблиці TTN_Materials. Було перевірено, що при перегляді ТТН у майбутньому всі дані відображаються точно, без втрати інформації або порушення зв’язків між таблицями.

Окрему увагу було приділено тестуванню обліку заправок. У формі реєстрації заправки вказувався транспортний засіб, дата, відповідальний користувач та перелік витрачених матеріалів. Після збереження система формувала два логічно зв’язані записи: один – у таблиці Refueling, другий – у Refueling_Materials. Подальший перегляд дозволяв проаналізувати витрати по кожному виду ПММ, що є важливим для оцінки ефективності експлуатації транспорту.

Тестування модуля звітності здійснювалось через фільтрацію даних за певними параметрами. Наприклад, користувач формував звіт про витрати

пального по обраному транспорту за конкретний період. Звіт будувався на основі агрегованих SQL-запитів і відображав зведену таблицю, де фіксувалася загальна кількість заправок, обсяг витрачених матеріалів та середні показники. Аналіз результатів засвідчив, що дані точно відповідають записам у таблицях, що підтверджує правильність реалізованих механізмів агрегації.

Крім функціональних сценаріїв, було змодельовано помилкові дії користувача. Наприклад, видалення об’єкта без попереднього вибору запису призводило до відсутності реакції програми – це означає, що перевірка на null у ViewModel працює коректно. Аналогічно, спроба зберегти незаповнену форму видавала помилку або блокувала виконання команди, що свідчить про наявність базової валідації.

Загалом, результати тестування підтвердили, що система стабільно обробляє введення, збереження, редагування та відображення даних. Алгоритми, реалізовані у ViewModel, коректно взаємодіють з моделями та інтерфейсом, забезпечуючи двосторонній обмін інформацією. Усі модулі програми пройшли перевірку на логічну цілісність, відповідність функціоналу та практичну зручність використання.

Варто зазначити, що прототип був протестований у середовищі розробки Visual Studio з використанням бази даних SQLite, розташованої локально. Це демонструє можливість роботи системи в автономному режимі без залежності від серверної інфраструктури. У межах модульного тестування не було виявлено критичних помилок, а межі застосування системи визначаються її архітектурною орієнтацією на локальне використання з одним користувачем або декількома користувачами послідовно. Масштабування системи на багатокористувацьке середовище або мережевий режим можливе за умови переходу на серверну СКБД і впровадження механізмів синхронізації доступу.

ВИСНОВКИ

У ході виконання дипломної роботи було реалізовано повнофункціональну інформаційно-облікову систему автотранспортних перевезень, що дозволяє автоматизувати основні облікові та управлінські процеси підприємства. Система побудована з урахуванням сучасних вимог до зручності використання, структурованості даних, безпеки та адаптивності до потреб малого і середнього бізнесу в сфері перевезень.

На основі критичного аналізу предметної області було визначено ключові проблеми: відсутність доступних, гнучких і простих у використанні рішень для обліку транспорту, водіїв, заправок, вантажів та складів. У порівнянні з існуючими комерційними платформами (зокрема BAS, BAF), розроблена система орієнтована на практичну ефективність та легкість впровадження без потреби у складному навчанні персоналу чи налаштуванні серверної інфраструктури.

В ході роботи були розв’язані завдання:

- Здійснений аналіз предметної області та існуючих рішень;
- Спроектована структура бази даних на основі СУБД SQLite ;
- Реалізований модульний інтерфейс користувача з підтримкою вкладок для ефективної навігації;
- Розроблений та реалізований функціонал для обліку транспорту, водіїв, ПММ, рейсів та складів за шаблоном MVVM у середовищі WPF;
- Забезпечена можливість авторизації та контролю доступу через систему ролей користувачів;
- розробки механізмів звітності з можливістю фільтрації, перегляду та експорту у табличні формати;
- Проведене тестування та оцінка працездатності системи.

Завдяки інтеграції з локальною базою SQLite, система досягає високої швидкодії при роботі з великим обсягом даних без необхідності в серверному забезпеченні.

Під час тестування програмного забезпечення було змодельовано типові сценарії роботи диспетчера, водія та адміністратора. Результати показали стабільну роботу системи, коректне збереження, редагування та виведення даних. Всі підсистеми працювали узгоджено, що підтверджує логічну та технічну цілісність рішення. Здійснено ручне тестування, що продемонструвало відповідність функціоналу поставленим вимогам.

У результаті дослідження було створено багатофункціональний настільний застосунок на базі C#, WPF і SQLite, що підтримує облік ресурсів, формування звітності, управління користувачами та захист даних. Практичне впровадження такої системи дозволяє підвищити ефективність управління автопарком і знизити витрати підприємства.

Таким чином, поставлену мету дослідження досягнуто повністю, а всі заплановані завдання реалізовано у межах функціонального, технічно обґрунтованого та практично застосовного програмного продукту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft Docs. C# programming guide. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення 27.05.2025)
2. Microsoft Docs. Windows Presentation Foundation (WPF). URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/ (дата звернення 26.05.2025)
3. Microsoft Docs. MVVM pattern in WPF.URL: https://learn.microsoft.com/en-us/dotnet/architecture/mvvm/ (дата звернення 28.05.2025)
4. Microsoft Docs. Visual Studio IDE documentation. URL: https://learn.microsoft.com/en-us/visualstudio/ (дата звернення 27.05.2025)
5. SQLite Documentation. SQLite Official Documentation. URL: https://www.sqlite.org/docs.html (дата звернення 26.05.2025)
6. Zetetic LLC. SQLCipher Documentation.URL: https://www.zetetic.net/sqlcipher/ (дата звернення 28.05.2025)
7. MVVM Light Toolkit Documentation. URL: https://mvvmlight.codeplex.com/documentation (дата звернення 27.05.2025)
8. Microsoft Learn. XAML overview (WPF). URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/xaml/ (дата звернення 26.05.2025)
9. Microsoft Docs. XAML Syntax In Detail (WPF). URL: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/xaml-syntax-in-detail (дата звернення 27.05.2025)
10. Zetetic LLC. SQLCipher Documentation.URL: https://www.zetetic.net/sqlcipher/ (дата звернення 28.05.2025)
11. Smith, Josh. Patterns – WPF Apps With The Model-View-ViewModel Design Pattern. MSDN Magazine, February 2009.
12. Gaffney, Kevin P., Prammer, Martin, Brasfield, Larry, Hipp, D. R., Kennedy, Dan, Patel, J. "SQLite: Past, Present, and Future." *PVLDB*, Vol. 15, № 12, 2022, pp. 3535–3547.

13. Ramachandrappa, Naveen C. "MVVM Design Pattern in Software Development." *International Journal of Computer Trends and Technology*, Vol. 72, № 9, September 2024.
14. Chin, S. T., Bhosale, et al. "SQLite: Light Database System." *International Journal of Computer Science and Mobile Computing*, Vol. 4, № 4, April 2015.
15. Mike Owens, The Definitive Guide to SQLite. Apress. May 24, 2006. 461 p.

ДОДАТКИ

Додаток А

Лістинг коду User.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Diploma.Models
{
   public class User
   {
      public int Id { get; set; }
      public string Number {  get; set; }
      public string Name { get; set; }
      public string Password { get; set; }
   }

}
```

Лістинг коду UserListPage.xaml:

```
<UserControl x:Class="Diploma.Views.UserListPage"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
        xmlns:local="clr-namespace:Diploma.Views"
        xmlns:models="clr-namespace:Diploma.Models"
        mc:Ignorable="d"
        d:DesignHeight="450" d:DesignWidth="800">
    <UserControl.Resources>
      <models:NullToBoolConverter x:Key="NullToBoolConverter"/>
    </UserControl.Resources>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="10*"/>
      </Grid.RowDefinitions>
      <StackPanel Orientation="Horizontal" Grid.Row="0">
        <Button Content="Додати"
        Command="{Binding AddUserCommand}"
        Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />

        <Button Content="Видалити"
        Command="{Binding DeleteUserCommand}"
        IsEnabled="{Binding SelectedUser, Converter={StaticResource
NullToBoolConverter}}"
        Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />

        <Button Content="Оновити список"
        Command="{Binding UpdateUsersList}"
        Margin="5" Padding="5" Grid.IsSharedSizeScope="True" />
      </StackPanel>
      <ListView ItemsSource="{Binding Users}"
            SelectedItem="{Binding SelectedUser}"
            MouseDoubleClick="ListView_MouseDoubleClick"
```

```
                Margin="0" Grid.Row="1">
            <ListView.View>
              <GridView>
                <GridViewColumn                                    Header="Номер"
DisplayMemberBinding="{Binding Id}" Width="50"/>
                <GridViewColumn                                    Header="Ім’я"
DisplayMemberBinding="{Binding Name}" Width="150"/>
              </GridView>
            </ListView.View>
          </ListView>
        </Grid>
```

Лістинг коду UserListViewModel.cs:

```
using Diploma.Models;
using Diploma.Services;
using Diploma.Views;
using Microsoft.Data.Sqlite;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Documents;
using System.Windows.Input;

namespace Diploma.ViewModels
```

```
{
    public class UserListViewModel : INotifyPropertyChanged
    {
        private readonly ITabHost _tabHost;
        public ObservableCollection<User> Users { get; set; } = new();

        private User _selectedUser;
        public User SelectedUser
        {
            get => _selectedUser;
            set { _selectedUser = value; OnPropertyChanged(nameof(SelectedUser)); }
        }

        public ICommand AddUserCommand => new RelayCommand(AddUser);
        public ICommand DeleteUserCommand => new RelayCommand(DeleteUser, CanDeleteUser);

        public ICommand UpdateUsersList => new RelayCommand(LoadUsersFromDatabase);

        public ICommand EditUserCommand => new RelayCommand<User>(EditUser);

        public UserListViewModel(ITabHost tabHost)
        {
            _tabHost = tabHost;
            LoadUsersFromDatabase(null);
        }
```

```
        private void EditUser(User user)
        {
            if (user == null) return;

            var editVM = new EditUserViewModel(user, _tabHost, false);
            var editPage = new EditUserPage { DataContext = editVM };

            var tab = new TabItemViewModel
            {
                Header = $"Редагування: {user.Name}",
                Content = editPage
            };

            _tabHost.AddTab(tab);
        }

        private void LoadUsersFromDatabase(object parameter)
        {

            string path =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "DiplomaDB.db");
            var connection = new SqliteConnection($"Data Source={path}");

            connection.Open();

            var command = connection.CreateCommand();
            command.CommandText = "SELECT user_id, user_fullName FROM
Users";

            using var reader = command.ExecuteReader();
```

```
    while (reader.Read())
    {
        User userToAdd = new User
        {
            Id = reader.GetInt32(0),
            //Number = reader.GetString(1),
            Name = reader.GetString(1)
        };
        //if (Users.Contains(userToAdd)) continue;
        if (Users.Any(usr => usr.Id == userToAdd.Id)) continue;
        else Users.Add(userToAdd);

    }

    connection.Close();
}
private void AddUser(object parameter)
{
    var newUser = new User(); // Порожній користувач
    var editVM = new EditUserViewModel(newUser, _tabHost, true);
    var editPage = new EditUserPage { DataContext = editVM };

    var tab = new TabItemViewModel
    {
        Header = "Новий користувач",
        Content = editPage
    };

    _tabHost.AddTab(tab);
}
```

```
        private bool CanDeleteUser(object _) => SelectedUser != null;

        private void DeleteUser(object _)
        {
            if (SelectedUser == null) return;

            string path =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "DiplomaDB.db");
            var connection = new SqliteConnection($"Data Source={path}");

            connection.Open();

                var command = connection.CreateCommand();
                command.CommandText = @"
                DELETE FROM Users
                WHERE user_id = $id;";

                command.Parameters.AddWithValue("$id", SelectedUser.Id);

                command.ExecuteNonQuery();

            connection.Close();

            Users.Remove(SelectedUser);
            SelectedUser = null;
        }

        public event PropertyChangedEventHandler PropertyChanged;
```

```
        protected void OnPropertyChanged([CallerMemberName] string name =
null)
            => PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(name));
        }
    }
```

Лістинг коду EditUserPage.xaml:

```
<UserControl x:Class="Diploma.Views.EditUserPage"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:Diploma.Views"
             xmlns:models="clr-namespace:Diploma.Models"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">

    <UserControl.Resources>
        <models:MultiParameterConverter x:Key="MultiParamConverter"/>
    </UserControl.Resources>

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2.5*"/>
            <ColumnDefinition Width="7.5*"/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
```

```
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
            <RowDefinition Height="30"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>

        <Label Grid.Column="0" Grid.Row="0" Margin="0">Ім'я користувача: </Label>
        <Label Grid.Column="0" Grid.Row="1" Margin="0">Пароль: </Label>

        <TextBox x:Name="UserName" Grid.Column="1" Grid.Row="0" Margin="5" Text="{Binding EditingUser.Name}" />

        <TextBox x:Name="UserPassword" Grid.Column="1" Grid.Row="1" Margin="5" Text="{Binding EditingUser.Password}"/>

        <Button Content="Зберегти" Command="{Binding SaveCommand}" Grid.Column="0" Grid.Row="2" Margin="5">
            <Button.CommandParameter>
                <MultiBinding Converter="{StaticResource MultiParamConverter}">
                    <Binding ElementName = "UserName" Path="Text"/>
                    <Binding ElementName = "UserPassword" Path="Text"/>
                </MultiBinding>
            </Button.CommandParameter>
        </Button>

        <StackPanel Margin="0" Grid.Column="1">
```

```
        </StackPanel>
    </Grid>


</UserControl>
```

Лістинг коду EditUserViewModel.cs:

```
using Diploma.Models;
using Microsoft.Data.Sqlite;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Threading.Tasks;
using System.Windows.Input;
using Diploma.Services;

namespace Diploma.ViewModels
{
    class EditUserViewModel
    {
        private readonly ITabHost _tabHost;

        public User EditingUser { get; set; }

        public ICommand SaveCommand => new RelayCommand(SaveUser);
        private string currentTabName = "";
        private bool _isNew;
```

```
        public EditUserViewModel(User user, ITabHost tabHost, bool isNew)
        {
            _tabHost = tabHost;
            EditingUser = user;
            currentTabName = $"Редагування: {user.Name}";
            _isNew = isNew;
        }

        public void SaveUser(object parameter)
        {
            string userName = "";
            string userPassword = "";

            if (parameter is object[] values)
            {
                userName = values[0] as string;
                userPassword = values[1] as string;
            }

            string path =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "DiplomaDB.db");
            var connection = new SqliteConnection($"Data Source={path}");

            connection.Open();

            if (_isNew)
            {
                currentTabName = "Новий користувач";
                var command = connection.CreateCommand();
```

```
            command.CommandText = @"
            INSERT INTO Users (user_fullName, user_password)
            VALUES ($name, $password);";
            command.Parameters.AddWithValue("$name", userName);
            command.Parameters.AddWithValue("$password", userPassword);

            command.ExecuteNonQuery();
        }
        else
        {
            var command = connection.CreateCommand();
            command.CommandText = @"
            UPDATE Users
            SET user_fullName = $name,
            user_password = $password
            WHERE user_id = $id;";
            command.Parameters.AddWithValue("$name", userName);
            command.Parameters.AddWithValue("$password", userPassword);
            command.Parameters.AddWithValue("$id", EditingUser.Id);
            command.ExecuteNonQuery();
        }

        connection.Close();

        var currentTab = _tabHost.GetTabItem(currentTabName);
        if(currentTab != null)
            _tabHost.CloseTab(currentTab);
    }
  }
}
```

Лістинг коду ITabHost.cs:

```
using Diploma.ViewModels;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Diploma.Services
{
    public interface ITabHost
    {
        void AddTab(object parameter);

        void AddTab(TabItemViewModel tab);

        TabItemViewModel GetTabItem(object parameter);

        void CloseTab(TabItemViewModel tab);
    }

}
```

Лістинг коду MultiParameterConverter.cs:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
using System.Windows.Data;

namespace Diploma.Models
{
   public class MultiParameterConverter : IMultiValueConverter
   {
      public object Convert(object[] values, Type targetType, object parameter,
CultureInfo culture)
      {
         return values.Clone();
      }

      public object[] ConvertBack(object value, Type[] targetTypes, object
parameter, CultureInfo culture)
      {
         throw new NotImplementedException();
      }
   }
}
```

Лістинг коду NullToBoolConverter.cs:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Data;

namespace Diploma.Models
```

```
{
    public class NullToBoolConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
            => value != null;

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
            => throw new NotImplementedException();
    }
}
```

Лістинг коду RelayCommand.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace Diploma.Models
{
    public class RelayCommand : ICommand
    {
        private readonly Action<object> _execute;
        private readonly Predicate<object> _canExecute;

        public RelayCommand(Action<object> execute, Predicate<object> canExecute = null)
```

```
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter) =>
            _canExecute == null || _canExecute(parameter);

        public void Execute(object parameter) => _execute(parameter);

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
    }

    public class RelayCommand<T> : ICommand
    {
        private readonly Action<T> _execute;
        private readonly Predicate<T> _canExecute;

        public RelayCommand(Action<T> execute, Predicate<T> canExecute = null)
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute;
        }
```

```
        public bool CanExecute(object parameter)
        {
            if (parameter == null && typeof(T).IsValueType)
                return _canExecute == null;

            return _canExecute == null || _canExecute((T)parameter);
        }

        public void Execute(object parameter)
        {
            _execute((T)parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
    }
}
```

Лістинг коду MainViewModel.cs:

```
using Diploma.Models;
using Diploma.Services;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;
using System.Windows.Controls;
using System.Windows.Input;

namespace Diploma.ViewModels
{
   public class MainViewModel : INotifyPropertyChanged, ITabHost
   {
      public ObservableCollection<TabItemViewModel> Tabs { get; set; }

      private TabItemViewModel _selectedTab;
      public TabItemViewModel SelectedTab
      {
         get => _selectedTab;
         set
         {
            _selectedTab = value;
            OnPropertyChanged(nameof(SelectedTab));
         }
      }

      public ICommand AddTabCommand { get; }
      public ICommand CloseTabCommand { get; }

      public ICommand SwitchToTabCommand { get; }

      public MainViewModel()
      {
         SQLitePCL.Batteries.Init();
```

```
    Tabs = new ObservableCollection<TabItemViewModel>();
    AddTabCommand = new RelayCommand(AddTab);
    CloseTabCommand = new RelayCommand(CloseTab);
    SwitchToTabCommand = new RelayCommand(SwitchToTab);
}

public void AddTab(object parameter)
{
    string tabName;
    string type;
    object view;
    object viewModel;

    if (parameter is object[] values && values.Length == 2)
    {
        tabName = values[0] as string;
        type = values[1] as string;
    }
    else
    {
        tabName = parameter?.ToString();
        type = "def";
    }
    foreach(TabItemViewModel tab in Tabs)
    {
        if(tab.Header == tabName)
        {
            SelectedTab = tab;
            return;
```

```
    }
}
switch (type)
{

    case "Довідники":
        viewModel = new SettingsViewModel(this);
        view = new Views.SettingsPage { DataContext = viewModel };
        break;

    case "Документи":
        viewModel = new DocumentsViewModel(this);
        view = new Views.SettingsPage { DataContext = viewModel };
        view = new Views.DocumentsPage();
        break;

    case "Регістри":
        viewModel = new RegistersViewModel(this);
        view = new Views.SettingsPage { DataContext = viewModel };
        view = new Views.RegistersPage();
        break;

    case "Звіти":
        viewModel = new ReportsViewModel(this);
        view = new Views.SettingsPage { DataContext = viewModel };
        view = new Views.ReportsPage();
        break;

    default:
        tabName = "Сторінка не знайдена";
```

```
            view = new TextBlock { Text = "404" };
            break;
        }

        var newTab = new TabItemViewModel
        {
            Header = tabName,
            Content = view,
            Type = type
        };

        Tabs.Add(newTab);
        SelectedTab = newTab;
    }

    public void AddTab(TabItemViewModel _newTab)
    {
        var newTab = _newTab;

        foreach (TabItemViewModel tab in Tabs)
        {
            if (tab.Header == newTab.Header)
            {
                SelectedTab = tab;
                return;
            }
        }

        Tabs.Add(newTab);
        SelectedTab = newTab;
```

```
}

private void CloseTab(object parameter)
{
    if (parameter is TabItemViewModel tab)
        Tabs.Remove(tab);
}

private void SwitchToTab(object parameter)
{
    string tabName = parameter?.ToString();

    var tab = Tabs.FirstOrDefault(t => t.Header == tabName);
    if (tab == null)
    {
        AddTab(tabName);
        tab = Tabs.FirstOrDefault(t => t.Header == tabName);
    }

    if (tab != null)
        SelectedTab = tab;
}

private TabItemViewModel? GetTabItem(object parameter)
{
    if(parameter is string tabName)
    {
        TabItemViewModel? TabItem = null;

        foreach (TabItemViewModel tab in Tabs)
```

```
            {
                if (tab.Header == tabName)
                {
                    TabItem = tab;
                }
            }

            return TabItem;
        }
        else
        {
            return null;
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string name) =>
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(name));

    void ITabHost.AddTab(object parameter)
    {
        //throw new NotImplementedException();
        AddTab(parameter);
    }

    void ITabHost.AddTab(TabItemViewModel tab)
    {
        //throw new NotImplementedException();
        AddTab(tab);
```

```
        }

        TabItemViewModel ITabHost.GetTabItem(object parameter)
        {
            return GetTabItem(parameter);
        }

        void ITabHost.CloseTab(TabItemViewModel tab)
        {
            CloseTab(tab);
        }
    }

}
```

Лістинг коду MainWindow.xaml:

```
<Window x:Class="Diploma.Views.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Diploma"
        xmlns:vm="clr-namespace:Diploma.ViewModels"
        xmlns:models="clr-namespace:Diploma.Models"
        Title="" Width="900" Height="600">
    <Window.DataContext>
        <vm:MainViewModel/>
    </Window.DataContext>
    <Window.Resources>
        <models:MultiParameterConverter x:Key="MultiParamConverter"/>
    </Window.Resources>
```

```
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="2.5*" />
                <ColumnDefinition Width="7.5*" />
            </Grid.ColumnDefinitions>

            <!-- Меню -->
            <StackPanel Grid.Column="0" Background="#EEE">
                <Button x:Name="SettingsButton" Content="Довідники"
                        Margin="10"
                        Command="{Binding AddTabCommand}">
                    <Button.CommandParameter>
                        <MultiBinding                    Converter="{StaticResource
MultiParamConverter}">
                            <Binding ElementName = "SettingsButton" Path="Content"/>
                            <Binding ElementName = "SettingsButton" Path="Content"/>
                        </MultiBinding>
                    </Button.CommandParameter>
                </Button>
                <Button x:Name="DocButton" Content="Документи"
                        Margin="10"
                        Command="{Binding AddTabCommand}">
                    <Button.CommandParameter>
                        <MultiBinding                    Converter="{StaticResource
MultiParamConverter}">
                            <Binding ElementName = "DocButton" Path="Content"/>
                            <Binding ElementName = "DocButton" Path="Content"/>
                        </MultiBinding>
                    </Button.CommandParameter>
                </Button>
```

```
            <Button x:Name="RegButton" Content="Регістри"
                    Margin="10"
                    Command="{Binding AddTabCommand}">
                <Button.CommandParameter>
                    <MultiBinding                    Converter="{StaticResource
MultiParamConverter}">
                        <Binding ElementName = "RegButton" Path="Content"/>
                        <Binding ElementName = "RegButton" Path="Content"/>
                    </MultiBinding>
                </Button.CommandParameter>
            </Button>
            <Button x:Name="RepButton"  Content="Звіти"
                    Margin="10"
                    Command="{Binding AddTabCommand}">
                <Button.CommandParameter>
                    <MultiBinding                    Converter="{StaticResource
MultiParamConverter}">
                        <Binding ElementName = "RepButton" Path="Content"/>
                        <Binding ElementName = "RepButton" Path="Content"/>
                    </MultiBinding>
                </Button.CommandParameter>
            </Button>
        </StackPanel>

        <!-- TabControl -->
        <TabControl Grid.Column="1" ItemsSource="{Binding Tabs}"
Margin="0" SelectedItem="{Binding SelectedTab, Mode=TwoWay}">
            <TabControl.ItemTemplate>
                <!-- Заголовок вкладки з кнопкою закриття -->
                <DataTemplate>
```

```
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Header}" Margin="0,0,5,0"/>
                        <Button Content="✕"
                                Command="{Binding DataContext.CloseTabCommand,
RelativeSource={RelativeSource AncestorType=TabControl}}"
                                CommandParameter="{Binding}"
                                Width="20" Height="20"
                                Padding="0" Margin="5,0,0,0"
                                Background="Transparent" BorderBrush="Transparent"
                                Cursor="Hand"/>
                    </StackPanel>
                </DataTemplate>
            </TabControl.ItemTemplate>
            <TabControl.ContentTemplate>
                <DataTemplate>
                    <ContentControl Content="{Binding Content}" />
                </DataTemplate>
            </TabControl.ContentTemplate>
        </TabControl>
    </Grid>
</Window>
```