МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ ЗАСОБАМИ C# ТА POSTGRESQL»**

Виконав: студент 2 курсу, групи 6.1213-пі-с3
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія (зі скороченим терміном навчання)
(назва освітньої програми)
Й. Ліфшиць
(ініціали та прізвище)
Керівник професор кафедри програмної інженерії, доцент, к.ф.-м.н. Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент доцент кафедри фундаментальної та прикладної математики, доцент, к.ф.-м.н. Панасенко Є.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія (зі скороченим терміном навчання)

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Ліфшицю Йонатану
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка інформаційної системи засобами C# та PostgreSQL

керівник роботи Кудін Олексій Володимирович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд інструментів та особливостей використання.
2. Проєктування системи.
3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	13.01.2025	
2.	Збір вихідних даних.	27.01.2025	
3.	Обробка методичних та теоретичних	17.02.2025	
	джерел.		
4.	Розробка першого та другого розділу.	14.04.2025	
5.	Розробка третього розділу.	20.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	20.06.2025	

Студент _____ Й. Ліфшиць
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.В. Кудін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка інформаційної системи засобами C# та PostgreSQL»: 44 с., 22 рис., 9 джерел.

АВТОЗАПЧАСТИНИ, БАЗА ДАНИХ, КОРИСТУВАЦЬКИЙ ІНТЕРФЕЙС, СИСТЕМА УПРАВЛІННЯ, СТРУКТУРА, MVVM, .NET.

Об’єкт дослідження – процес створення інформаційних систем для зберігання, обробки та управління інформацією про автотранспорт.

Мета роботи: створити функціональний, зручний у використанні програмний продукт “AutoStockManager” для ефективного ведення обліку автомобілів, пов’язаних із ними деталей (запчастин), інформації про водіїв, а також базових параметрів технічного обслуговування.

Методи дослідження – методи збору та аналізу вимог до програмного забезпечення, методи моделювання, проєктування, конструювання, реляційне проєктування баз даних та тестування програмного забезпечення.

Для реалізації програмного застосунку використано патерн MVVM (Model-View-ViewModel), що сприяє створенню ефективного та добре структурованого програмного коду, а також полегшує роботу з інтерфейсом користувача та управлінням даними. Програма “AutoStockManager” базується на платформі .NET та використовує мови програмування, які підтримуються цією платформою. Вона дозволяє користувачам зберігати важливі дані про автомобілі та їх компоненти, допомагаючи в підтримці та ефективному управлінні автопарком.

Результатом роботи є створений програмний продукт “AutoStockManager”, який надає можливість користувачам зручно та ефективно керувати інформацією про автомобілі, їх деталі та обслуговування, сприяючи у підтримці та управлінні технічним станом автопарку.

SUMMARY

Bachelor’s qualifying paper “Development of an Information System Using C# and PostgreSQL”: 44 pages, 22 figures, 9 references.

AUTO PARTS, DATABASE, USER INTERFACE, MANAGEMENT SYSTEM, STRUCTURE, MVVM, .NET.

The object of the study is to analyze the process of developing information systems designed for storing, processing, and managing data related to vehicles.

The aim of the study is to develop a functional and user-friendly software product, “AutoStockManager”, for the effective management of vehicle records, associated parts (spare parts), driver information, and basic vehicle maintenance parameters.

The research methods include gathering and analyzing software requirements, modeling, designing, constructing, relational database design, and software testing.

The software application was implemented using the MVVM (Model-View-ViewModel) design pattern, which helps to create efficient and well-structured code and facilitates interaction with the user interface and data management. The “AutoStockManager” program is based on the .NET platform and utilizes programming languages supported by this environment. It enables users to store essential information about vehicles and their components, thereby supporting the maintenance and efficient management of a vehicle fleet.

As a result of the study, a software product called “AutoStockManager” was developed. It allows users to conveniently and effectively manage information about vehicles, their components, and maintenance, thus aiding in the upkeep and technical management of the fleet.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 8
1 Огляд інструментів та особливостей використання 10
1.1 Технологічний стек: .NET, WPF, MVVM..... 10
1.2 Робота з базою даних PostgreSQL через Npgsql 11
1.3 Інкапсуляція логіки через ViewModel: реалізація у MainWindowModel 12
1.4 Інтеграція компонентів у MVVM-архітектурі 14
2 Проєктування системи 17
2.1 UML-діаграма класів моделей даних 17
2.2 Діаграма послідовності для операцій CRUD 18
2.3 ER-модель бази даних PostgreSQL..... 20
2.4 Проєктування інтерфейсу користувача 22
3 Реалізація та тестування 27
3.1 Реалізація моделей даних (Car, Sp, Driver)..... 27
3.2 Прив’язка даних у ViewModel 29
3.2.1 Базова ViewModel 30
3.2.2 Завантаження даних..... 31
3.2.3 Використання даних у ViewModel..... 31
3.2.4 Система команд..... 32
3.2.5 Додаткові ViewModel 33
3.3 Юніт-тестування бізнес-логіки..... 34
3.3.1 Архітектура логіки..... 34
3.3.2 Перевірка завантаження та зчитування даних 34
3.3.3 Інтеграція команд у інтерфейс 35

3.3.4 Практичне тестування 35
3.3.5 Надійність і масштабованість 36
3.4 Тестування інтерфейсу користувача 36
3.4.1 Додавання автомобіля 37
3.4.2 Додавання деталей (Sp) та водіїв 37
3.4.3 Завантаження даних 38
3.4.4 Валідація та обробка помилок 39
3.4.5 Прив’язка даних та команди 39
3.4.6 Поведінкові спостереження 39
Висновки 41
Перелік посилань 44

ВСТУП

Сучасний розвиток технологій та програмного забезпечення у сфері транспорту та автомобільної промисловості відкриває широкі можливості для ефективного управління та обслуговування автопарків. З кожним днем зростає потреба в інструментах, які дозволять зручно та організовано зберігати дані про автомобілі, включаючи деталі, технічний стан, а також інформацію про планове обслуговування.

Окрім класичних підходів до управління транспортом, дедалі більшого значення набувають інноваційні рішення на основі технологій Інтернету речей (IoT), великих даних (Big Data), телематики та хмарних обчислень. Вони дозволяють не лише отримувати інформацію про транспортні засоби в режимі реального часу, але й проводити глибокий аналіз технічного стану, прогнозувати несправності та оптимізувати маршрути. Це створює фундамент для прийняття обґрунтованих управлінських рішень і зниження експлуатаційних витрат підприємств.

Одним із важливих аспектів в управлінні автомобільним парком є зберігання та структурування необхідних даних про кожен автомобіль та його компоненти. Саме в цьому контексті виникає необхідність у розробці інформаційних систем, які забезпечать ефективне управління та зберіганням цих даних.

Актуальною задачею стає розробка інформаційних систем, спрямованих на поліпшення управління автомобільним парком, включаючи зберігання та обробку даних про автомобілі та їх компоненти. При цьому важливо враховувати не лише функціональні можливості таких систем, а й їх інтеграцію з мобільними застосунками, автоматизовану звітність, забезпечення безпеки даних, а також відповідність сучасним екологічним стандартам.

Метою даної дипломної роботи є створити функціональний, зручний у використанні програмний продукт “AutoStockManager” для ефективного

ведення обліку автомобілів, пов’язаних із ними деталей (запчастин), інформації про водіїв, а також базових параметрів технічного обслуговування.

Для досягнення поставленої мети необхідно виконати ряд завдань, серед яких:

– аналіз існуючих систем керування базами даних;
– вивчення предметної області та вимог до інформаційної системи;
– проєктування схеми даних та макету клієнтського програмного застосунку;
– реалізація бази даних та клієнтського програмного застосунку;
– проведення тестування інформаційної системи.

Об’єктом дослідження є процес розробки інформаційної системи “AutoStockManager” в обраній предметній області, а предметом – обрана система керування базами даних.

В роботі розглядається реалізація інформаційної системи у двох розділах, що відповідають основним аспектам її розробки: складовою сучасних підприємств, зокрема у зв’язку з глобалізацією ланцюгів постачання та зростанням обсягів даних. Проєкт “AutoStockManager” надає комплексне рішення для контролю над рухом товарів, оптимізації складських операцій та забезпечення прозорості бізнес-процесів.

Сучасні компанії зіштовхуються з викликами підвищеної конкуренції, ризиками нестачі або перевантаження запасів, а також з необхідністю інтеграції зі сторонніми системами (ERP, CRM). “AutoStockManager” вирішує ці проблеми за допомогою веборієнтованої архітектури та модульного підходу до розширення функціональності. Окрім того, використання REST API та інтеграції з мобільними платформами дозволяє реалізувати оперативний контроль над усіма складовими процесу керування автопарком.

Структура роботи відповідає трьом основним розділам з чітким поділом змісту, що забезпечує логічну послідовність викладу.

1 ОГЛЯД ІНСТРУМЕНТІВ ТА ОСОБЛИВОСТЕЙ ВИКОРИСТАННЯ

1.1 Технологічний стек: .NET, WPF, MVVM

Застосунок “AutoStockManager” розроблено на платформі .NET з використанням технології WPF (Windows Presentation Foundation) для побудови графічного інтерфейсу користувача, а також архітектурного патерну MVVM (Model-View-ViewModel) для організації логіки взаємодії між даними та відображенням [8]. Основною мовою програмування є C#, фреймворк – .NET 7.0, що забезпечує сучасне середовище для розробки десктопних програм [7].

У даному проєкті для взаємодії з базою даних PostgreSQL використовується бібліотека Npgsql, яка дозволяє напряму виконувати SQL-запити з C# без використання сторонніх ORM, таких як Entity Framework. Уся логіка роботи з базою інкапсульована в спеціальному класі DbTools, який приймає назву таблиці як параметр і виконує типові запити на додавання, вибірку, оновлення і логічне видалення даних. Такий підхід забезпечує максимальний контроль над запитами, а також простоту розширення і супроводу коду.

MVVM-патерн використовується для розділення логіки програми на три незалежні частини:

- Model – класи моделей даних (DTO) для зберігання інформації про автомобілі, деталі та водіїв;
- ViewModel – клас MainWindowModel, який обробляє всі команди користувача, керує завантаженням даних та відкриттям вікон;
- View – XAML-інтерфейси, прив’язані до MainWindowModel через механізм DataContext і підтримку INotifyPropertyChanged.

Завдяки використанню ObservableCollection та системи команд ICommand, усі зміни у даних автоматично відображаються в інтерфейсі, а логіка взаємодії з користувачем реалізується в ViewModel, не зачіпаючи

XAML. Це дозволяє забезпечити слабке зв’язування компонентів, полегшує тестування логіки та подальше масштабування програми.

Таким чином, стек технологій .NET / WPF / MVVM / Npgsql у поєднанні з чітко визначеною структурою логіки забезпечує високу ефективність і надійність реалізованого застосунку “AutoStockManager”.

1.2 Робота з базою даних PostgreSQL через Npgsql

У програмному застосунку “AutoStockManager” для взаємодії з реляційною базою даних PostgreSQL використовується бібліотека Npgsql – офіційний .NET-драйвер для PostgreSQL, що дозволяє напряму виконувати SQL-запити з мови C# [9]. На відміну від ORM-рішень (таких як Entity Framework Core), у цьому проєкті реалізовано повністю ручний контроль над логікою доступу до даних – що дозволяє гнучко та точно керувати запитами, підключенням та обробкою помилок.

Уся логіка роботи з базою інкапсульована в спеціальному класі DbTools, який приймає назву таблиці як параметр конструктора. Цей підхід дозволяє повторно використовувати один і той самий клас для роботи з різними сутностями (водії, автомобілі, запчастини), просто змінюючи ім’я таблиці.

Підключення до PostgreSQL здійснюється за допомогою рядка з’єднання виду (див. рис. 1.1).

```
connectString = $"Host={host};Database={database};Username={username};Password={password}";
connection = new NpgsqlConnection(connectString);
```

Рисунок 1.1

Основні операції, реалізовані в DbTools, включають:

– завантаження записів з таблиці (SELECT * FROM table_name);
– додавання нового запису (INSERT INTO table_name (...) VALUES (...));

– оновлення значень (через UPDATE);
– м’яке видалення – не фізичне видалення рядка, а зміна статусу (наприклад, на 'deleted'), що дозволяє зберігати історію змін.

Для роботи із результатами запитів використовується об’єкт NpgsqlDataReader, який дозволяє по рядках читати вміст таблиці та створювати відповідні DTO-об’єкти (наприклад, Driver, Car, Sp), які далі додаються у колекції.

Зчитані об’єкти додаються до статичних колекцій у класі DataStorag, які використовуються як джерело даних для відображення у інтерфейсі (через ObservableCollection<>). Це дозволяє оновлювати UI автоматично при додаванні або зміні записів.

Переваги обраного підходу:

– простота реалізації – не потребує міграцій, складної конфігурації чи налаштувань Fluent API;
– максимальний контроль – розробник сам пише SQL, що дозволяє точно налаштувати запити;
– гнучкість – один клас DbTools обробляє всі таблиці;
– надійність – уся логіка концентрована в одному місці, що полегшує тестування та підтримку.

Таким чином, замість ORM, як-от Entity Framework Core, у проєкті свідомо використовується прямий доступ до PostgreSQL через Npgsql, що забезпечує кращу прозорість і контроль над базовими операціями та покращує продуктивність для невеликого настільного застосунку [3].

1.3 Інкапсуляція логіки через ViewModel: реалізація у MainWindowModel

У застосунку “AutoStockManager” логіка реалізована за допомогою патерну MVVM (Model-View-ViewModel), що дозволяє чітко розділити UI, бізнес-логіку та дані. Усі ключові взаємодії між користувачем і програмою

зосереджено в одному ViewModel-класі – MainWindowModel, який відповідає за управління станом інтерфейсу та обробку команд [1].

На відміну від класичної багатошарової реалізації, де для кожного представлення створюється окремий ViewModel (наприклад, CarViewModel, DriverViewModel), у цьому проєкті весь функціонал об’єднаний в одному централізованому класі MainWindowModel. Він забезпечує керування:

– списками моделей (cars, drivers, car_parts) через об’єкти DataStorag;
– командами для створення, видалення, завантаження даних (AddDriverCommand, DeleteSpCommand, LoadDbCommand тощо);
– відкриттям вікон (наприклад, AddDriverWindow, AddSpWindow) із передачею об’єкта DbTools.

Цей підхід дозволяє ефективно керувати всіма даними через один ViewModel-клас, що значно спрощує зв’язування з UI та підтримку коду.

Команди (ICommand). Усі взаємодії з UI реалізовані за допомогою команд, наприклад:

– DeleteDriverCommand – м’яко позначає водія як видаленого (через зміну статусу в БД);
– CreatAddSpWindowCommand – відкриває діалогове вікно для додавання запчастини;
– LoadDbSpCommand – виконує завантаження актуального стану таблиці car_parts із БД.

Ці команди реалізовані через LambdaCommand, що дозволяє оголошувати як саму дію, так і умову її доступності.

Прив’язка даних і ObservableCollection. Колекції сутностей (Drivers, Cars, Sp) зберігаються у класі DataStorag як ObservableCollection<>, завдяки чому будь-яка зміна (додавання, видалення, оновлення) автоматично відображається в інтерфейсі без необхідності вручну оновлювати UI.

Прив’язка елементів WPF (наприклад, DataGrid.ItemsSource) напряму використовує ці колекції: <DataGrid ItemsSource="{x:Static local:DataStorag.Drivers}" />. Це забезпечує двосторонню синхронізацію між моделлю представлення і відображенням.

Переваги обраного підходу:

– простота та централізація: вся логіка в одному ViewModel-класі (MainWindowModel) полегшує налагодження і розвиток;
– мінімум дублювання: завдяки відсутності окремих ViewModel для кожної сутності, код не дублюється;
– гнучкість і повторне використання: команди та методи легко використовуються повторно з різних вікон;
– підтримка масштабування: за потреби логіку можна легко винести в окремі класи або ViewModel-компоненти.

У підсумку, реалізація MVVM у “AutoStockManager” адаптована під структуру застосунку і використовує один ViewModel для обробки всіх взаємодій, що забезпечує баланс між чистою архітектурою та практичністю для настільного WPF-застосунку [5].

1.4 Інтеграція компонентів у MVVM-архітектурі

Архітектура застосунку “AutoStockManager” реалізована згідно з патерном MVVM (Model–View–ViewModel), що забезпечує розділення відповідальностей між графічним інтерфейсом, логікою керування станом та бізнес-даними. На практиці:

– View (вікна та елементи інтерфейсу WPF) відповідає лише за візуалізацію та події користувача;
– ViewModel (у нашому випадку – єдиний клас MainWindowModel) обробляє логіку роботи інтерфейсу, команди, взаємодію з базою та стан колекцій;
– Model – це прості DTO-класи (Car, Sp, Driver), які містять лише поля без навігаційних властивостей або поведінки.

Як показано на рисунку 1.2, взаємодія побудована так, що View взаємодіє з ViewModel через механізм data binding, а ViewModel у свою чергу – напряму працює з моделями та базою через клас DbTools.

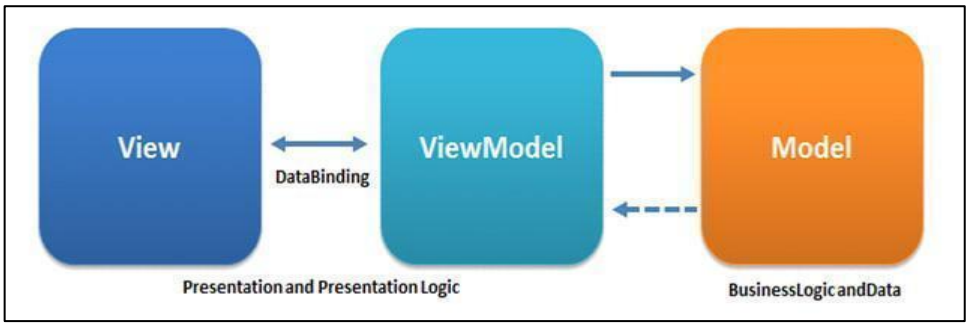


Рисунок 1.2 – Структура MVVM у “AutoStockManager”

Прив’язка даних (Data Binding). Усі DataGrid, TextBox, Button та інші UI-елементи в XAML прив’язані до властивостей або команд MainWindowModel. Наприклад, як на рисунку 1.3.

```
<DataGrid ItemsSource="{x:Static local:DataStorag.Drivers}" />
<Button Command="{Binding DeleteDriverCommand}" />
```

Рисунок 1.3

Колекції типу ObservableCollection<> знаходяться в класі DataStorag, що дозволяє автоматично оновлювати UI при зміні вмісту – без додаткової логіки оновлення.

Команди (ICommand). Використовується шаблон команд ICommand, зручна реалізація – через LambdaCommand (рис. 1.4). Це дозволяє відокремити логіку взаємодії з користувачем від інтерфейсу:

– CanExecute визначає, коли команда доступна;
– Execute виконує дію (наприклад, видалення).

```
DeleteSpCommand = new LambdaCommand(OnDeleteSp, CanDeleteSp)
```

Рисунок 1.4

Реактивність через INotifyPropertyChanged. Клас MainWindowModel реалізує механізм INotifyPropertyChanged, що дозволяє View автоматично оновлювати дані, коли ViewModel змінює властивості. Це спрощує керування станом: не потрібно викликати методи оновлення вручну.

Взаємодія між компонентами. Усі взаємодії відбуваються централізовано через ViewModel:

– натискання кнопки → викликає команду у ViewModel;
– команда викликає метод класу DbTools, що виконує SQL-запит до PostgreSQL;
– після успішного запиту колекція DataStorag оновлюється;
– оновлення ObservableCollection призводить до автоматичного оновлення вмісту DataGrid.

ViewModel є «посередником», який отримує команди від UI та працює з моделями даних:

– View не має жодної логіки, а лише відображає інформацію;
– Model не «знає» про ViewModel і використовується тільки для зберігання даних.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 UML-діаграма класів моделей даних

Проєктування структури даних системи “AutoStockManager” виконано на основі простих класів-моделей, які відображають сутності, з якими працює користувач – автомобілі, водії та запасні частини. Кожен клас відповідає окремій таблиці в базі даних PostgreSQL.

Усі моделі реалізовано як звичайні C# класи без навігаційних властивостей або складних зв’язків між ними (DTO – Data Transfer Objects). Вони містять лише поля для зберігання даних.

Основні класи:

– Car – зберігає інформацію про автомобіль: марка, номер, рік випуску;
– Driver – описує водія: ім’я, номер телефону, номер авто, яким він керує;
– Sp (CarPart) – описує деталь, прив’язану до автомобіля: назва, кількість, номер авто.

Усі ці класи незалежні між собою. Замість використання навігаційних властивостей (як у EF Core), у кожній таблиці є прості текстові поля, які виконують роль зв’язку – наприклад, поле car_number у Sp або у Driver просто містить номер авто у вигляді рядка, без зовнішніх ключів.

Структура UML. На рисунку 2.1 умовно зображено структуру основних класів. Для зручності читача кожен клас має:

– Id – унікальний ідентифікатор;
– основні атрибути, з якими працює користувач у вікнах додавання, видалення та перегляду;
– усі зв’язки (наприклад, між авто і деталлю) реалізуються через текстове поле car_number, а не через Foreign Key.

Розглянемо типи даних для кожного з полів таблиці.

Car:

– id: int;
– model: string;
– car_number: string.

Driver:

– id: int;
– name: string;
– phone: string;
– car_number: string.

Sp:

– id: int;
– name: string;
– count: int;
– car_number: string.

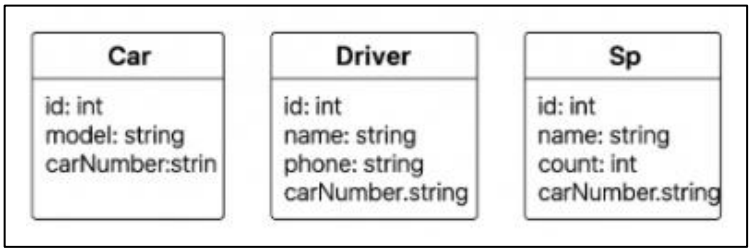


Рисунок 2.1 – UML-діаграма класів “AutoStockManager”

Коментар до діаграми. UML-діаграма у цьому випадку не відображає асоціації типу «один-до-багатьох» або «один-до-одного», оскільки вони не реалізовані на рівні моделей. Уся логіка відображення зв’язків – умовна, заснована на збігу значень полів (car_number).

2.2 Діаграма послідовності для операцій CRUD

Для демонстрації динаміки виконання типових дій у застосунку “AutoStockManager” побудовано діаграму послідовності, що описує CRUD-

операції (створення, читання, оновлення, видалення) (див. рис. 2.2). На діаграмі зображено взаємодію між такими компонентами:

– User – користувач, що ініціює дію через інтерфейс;
– MainWindow – головне вікно застосунку (View);
– MainWindowModel – ViewModel, яка містить усю логіку обробки подій;
– DbTools – сервіс доступу до бази даних, який виконує SQL-запити напряму через Npgsql;
– Database – база даних PostgreSQL.

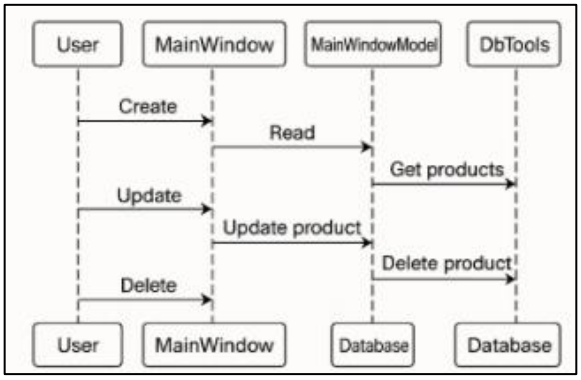


Рисунок 2.2 – Діаграма послідовності, що демонструє виконання CRUD-операцій у “AutoStockManager”

Приклад – оновлення запису:

– користувач натискає кнопку «Оновити» на інтерфейсі;
– MainWindow виконує команду, прив’язану до ViewModel (MainWindowModel);
– ViewModel отримує нові значення та викликає відповідний метод у DbTools, наприклад UpdateProduct(...);
– DbTools формує SQL-запит UPDATE та виконує його через Npgsql.
– PostgreSQL оновлює відповідний запис у базі;
– MainWindowModel, якщо потрібно, оновлює прив’язану колекцію у UI;
– дані автоматично оновлюються на екрані через INotifyPropertyChanged та ObservableCollection.

Опис інших CRUD-операцій

– Create (створення): користувач заповнює форму, натискає кнопку «Додати», ViewModel формує SQL-запит INSERT через DbTools, запис додається в базу, після чого об’єкт додається до колекції в UI;
– Read (читання): при відкритті вікна MainWindowModel викликає DbTools.GetProducts(), отримані дані заповнюють колекції ViewModel і відображаються у списках;
– Delete (видалення): при натисканні кнопки «Видалити» ViewModel викликає DbTools.DeleteProduct(id), запис видаляється в базі, і колекція в UI оновлюється.

Важливі особливості:

– весь доступ до бази здійснюється не через Entity Framework Core, а напряму через SQL-запити в DbTools;
– жодна частина View (вікно) не має прямого доступу до бази або DbTools – вся логіка реалізована у ViewModel, що відповідає принципу розділення відповідальностей (SRP);
– ViewModel не містить жорстких зв’язків з View – лише властивості, події зміни (INotifyPropertyChanged) та колекції (ObservableCollection);
– дані між View і ViewModel передаються через двосторонню прив’язку (data binding), що забезпечує автоматичну синхронізацію інтерфейсу.

2.3 ER-модель бази даних PostgreSQL

Сутнісно-зв’язкова модель (ER-модель) бази даних описує логічну структуру таблиць системи “AutoStockManager” та зв’язки між ними.

На ER-діаграмі (див. рис. 2.3) відображено три основні сутності: Car, Driver, Sp, які повністю відповідають реальному вмісту таблиць у PostgreSQL-базі проєкту.

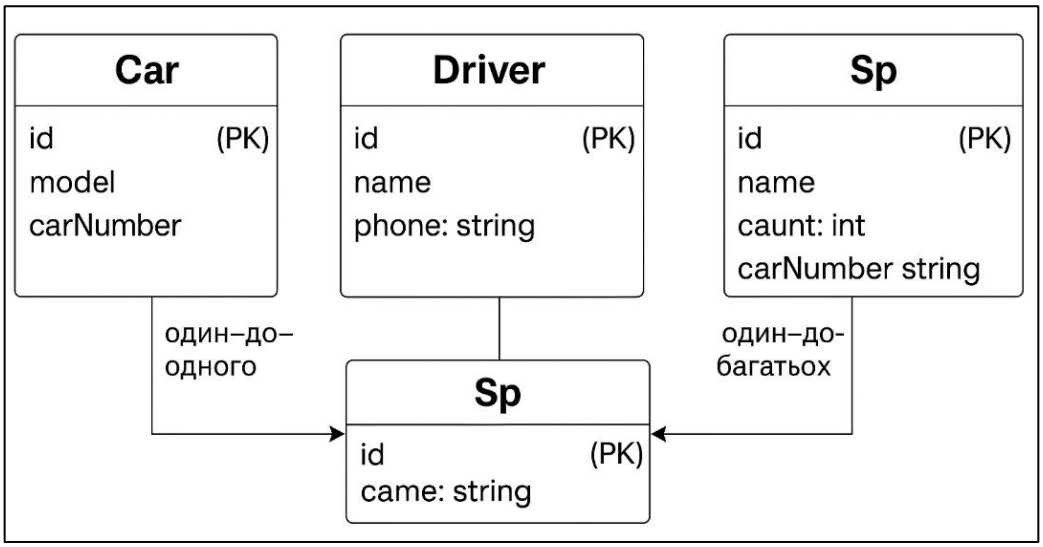


Рисунок 2.3 – ER-діаграма бази даних “AutoStockManager”

Розглянемо основні таблиці та їх поля.

Car – таблиця з інформацією про автомобілі. Містить такі поля:

– id (PK) – ідентифікатор;
– model – модель авто;
– carNumber – унікальний номер автомобіля (використовується для зв’язку).

Driver – таблиця водіїв. Містить:

– id (PK);
– name – ім’я водія;
– phone – телефон;
– carNumber – номер авто, який обслуговує цей водій (зв’язок логічно будується через carNumber).

Sp – таблиця автозапчастин:

– id (PK);
– name – назва деталі;
– count – кількість;
– carNumber – номер автомобіля, до якого належить ця деталь.

Розберемо характер зв’язків.

Зв’язок “Car ↔ Driver” реалізований через поле carNumber у таблиці Driver. Один водій обслуговує один автомобіль (зв’язок один-до-одного, хоча

реалізований без зовнішнього ключа, але через унікальне значення carNumber).

Зв’язок “Car ↔ Sp” також реалізується за допомогою поля carNumber в таблиці Sp, що означає: один автомобіль має багато деталей (один-до-багатьох). Кожна деталь стосується одного конкретного авто.

Особливості реалізації:

– у базі немає навігаційних властивостей чи складних зовнішніх ключів – поля carNumber використовуються як текстовий зв’язок між таблицями;
– структура таблиць є максимально простою: кожна сутність містить лише примітивні поля (int, string) без вкладених об’єктів, складених ключів або залежностей типу many-to-many;
– інтеграція з PostgreSQL виконується без ORM (Entity Framework не використовується), а усі SQL-запити реалізовані вручну через Npgsql у класі DbTools.

Завдяки такій структурі забезпечується простота запитів, наочність зв’язків та гнучкість обробки даних у ViewModel. Для фільтрації, пошуку або додавання нових записів застосунок працює напряму з таблицями через SQL-оператори SELECT, INSERT, UPDATE, DELETE, не використовуючи проміжних репозиторіїв або складних ORM-абстракцій.

2.4 Проєктування інтерфейсу користувача

Інтерфейс користувача (UI) програми “AutoStockManager” реалізований за допомогою технології WPF (Windows Presentation Foundation), що дозволяє створювати гнучкий, адаптивний і стильний графічний інтерфейс на основі XAML-розмітки. Вся логіка, пов’язана з інтерфейсом, інкапсульована у ViewModel згідно з архітектурним патерном MVVM. Уся взаємодія між View (вікнами) та бізнес-логікою реалізується через один головний ViewModel – MainWindowModel.

Інтерфейс організовано навколо головного вікна, яке містить вкладки для трьох основних розділів: «Автомобілі», «Деталі», «Водії». Це реалізовано через компонент TabControl, де кожна вкладка пов’язана зі своєю частиною інтерфейсу, але обслуговується спільним ViewModel. Таким чином, усі дії (додавання, редагування, видалення) обробляються у межах одного класу MainWindowModel.

На рисунку 2.4 представлено скриншот головного вікна програми із вкладками для керування даними. Зліва – вкладки «Авто», «Деталі», «Водії», справа – список відповідних об’єктів у вигляді таблиці.

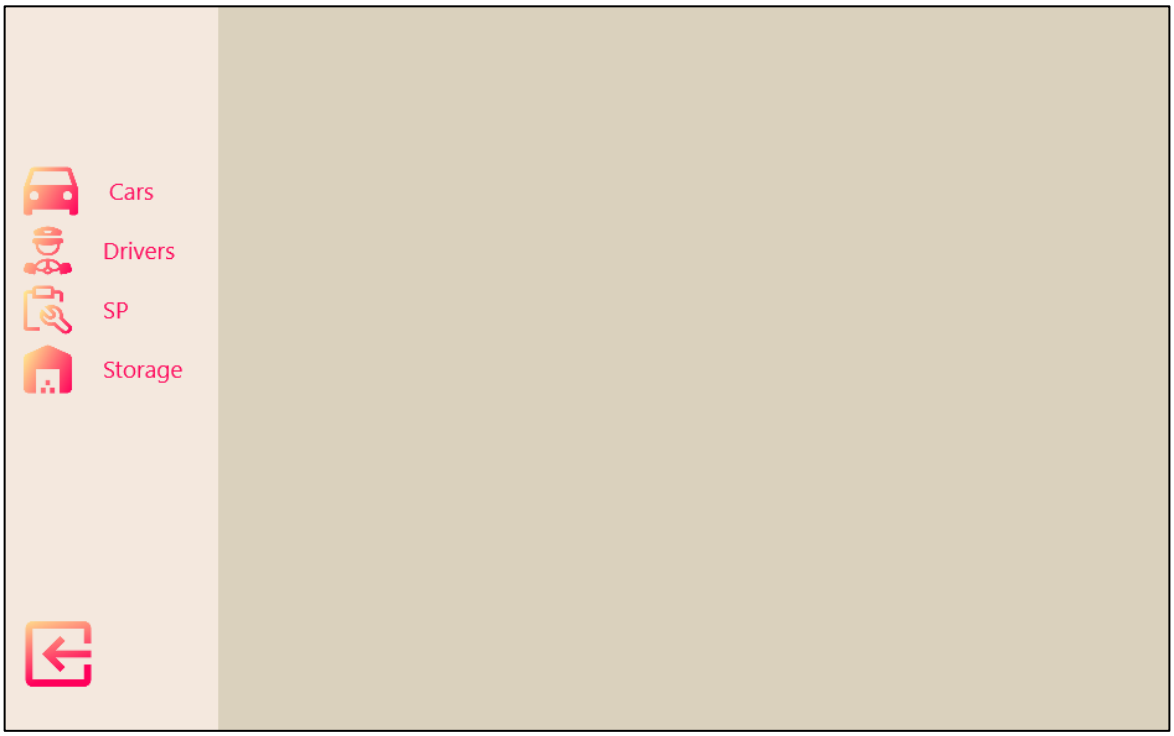


Рисунок 2.4

Для виведення списків даних використано компонент DataGrid, що дозволяє ефективно відображати, редагувати та видаляти записи. Наприклад:

– у вкладці «Автомобілі» виводяться id, model, carNumber;
– у вкладці «Водії» – id, name, phone, carNumber;
– у вкладці «Деталі» – id, name, count, carNumber.

Усі дані оновлюються автоматично завдяки механізму прив’язки (DataBinding) до властивостей ViewModel, які реалізують інтерфейс

INotifyPropertyChanged. Колекції елементів зберігаються у вигляді ObservableCollection<T>, що забезпечує динамічне оновлення UI при зміні даних.

Кнопки «Додати», «Редагувати», «Видалити» розташовані у верхній частині кожної вкладки й зв’язані з відповідними командами (ICommand) у ViewModel. Після натискання виконується дія:

– відкривається модальне вікно з формою;
– ViewModel обробляє введення;
– після натискання «Зберегти» – дані зберігаються у базу через клас DbTools, що виконує SQL-запити напряму (без EF Core).

На рисунку 2.5 представлено скриншот вкладки «Деталі». DataGrid містить список деталей з можливістю пошуку, кнопками CRUD і фільтрацією по carNumber.

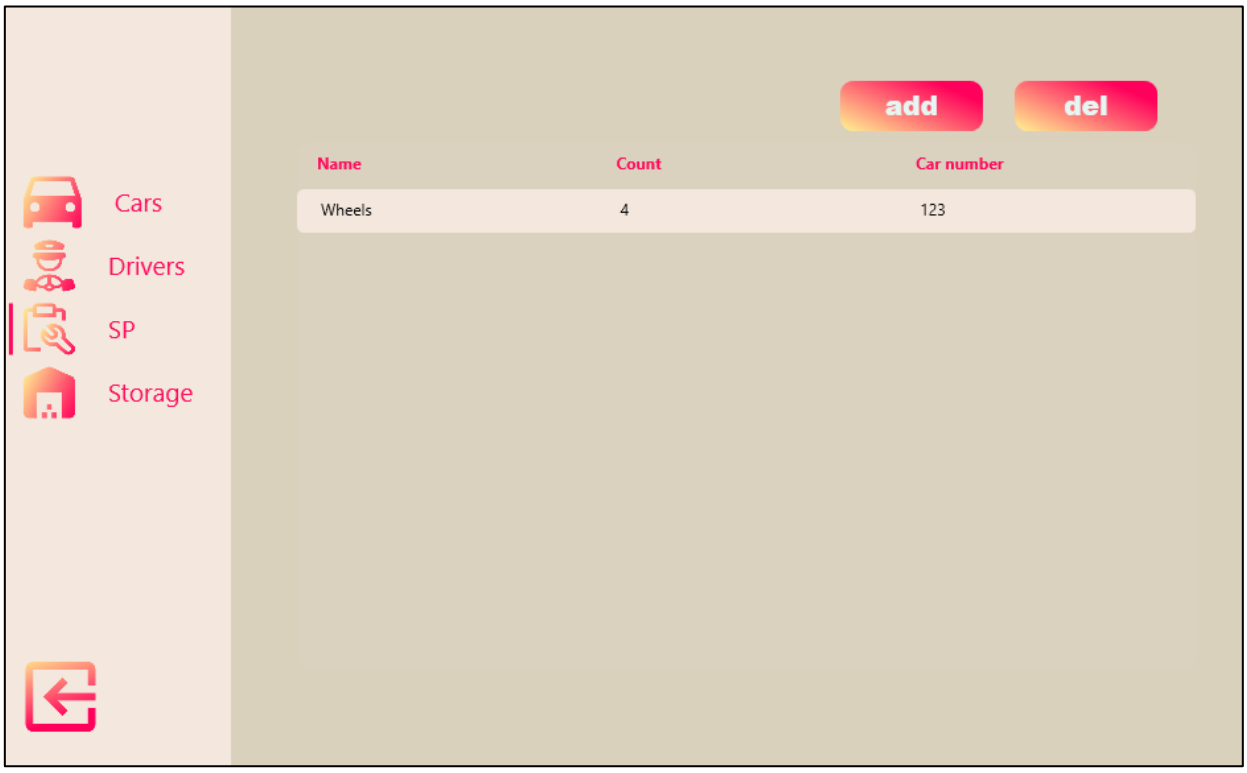


Рисунок 2.5

Пошук реалізовано за допомогою текстового поля, значення якого фільтрує колекцію в ViewModel у режимі реального часу.

Форми додавання/редагування реалізовано як окремі діалогові вікна (Window) або вкладки у головному інтерфейсі. У формі для нового Car, наприклад, є поля:

– «Модель» (TextBox);
– «Номер авто» (TextBox).

Для нового Sp – поля «Назва», «Кількість», «Номер авто».

Валідація введення реалізується у ViewModel вручну (через умови) або через інтерфейс IDataErrorInfo, де View автоматично підсвічує некоректні поля.

На рисунку 2.6 представлено скриншот прикладу вікна із таблицею водіїв.

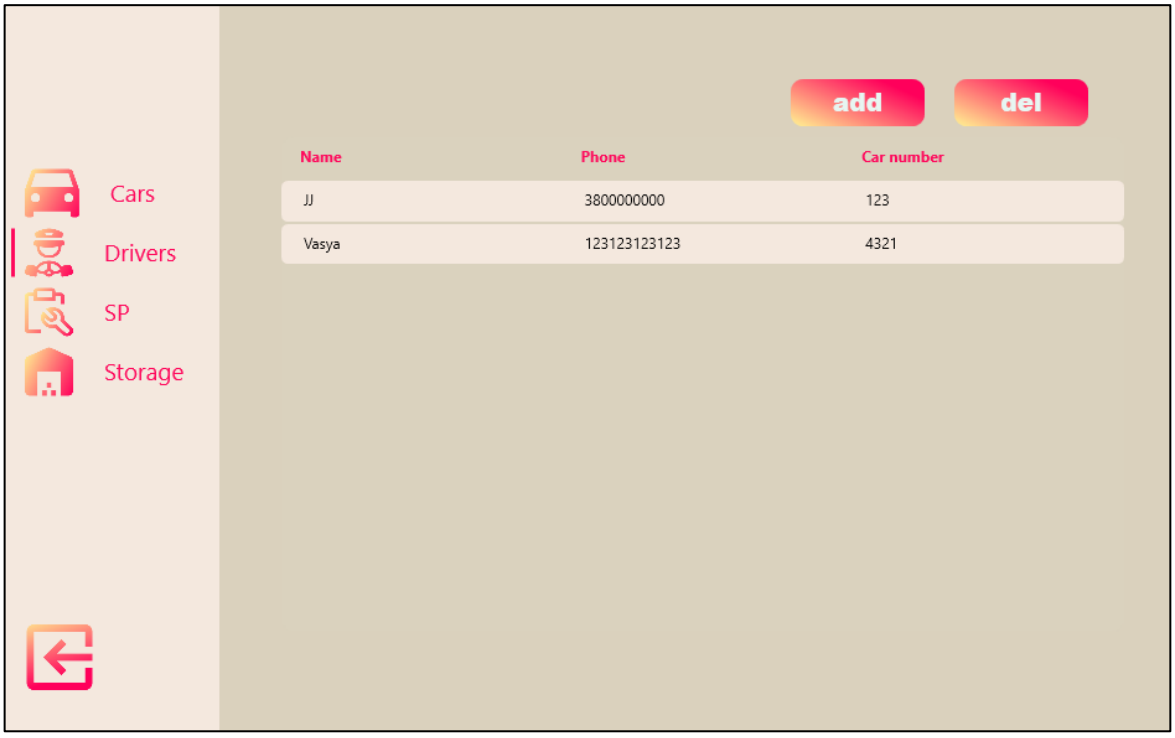


Рисунок 2.6

У програмі не використовуються зв’язки між класами-моделями у вигляді вкладених об’єктів (тобто, Car.Driver, Sp.Car тощо не існують). Усі зв’язки логічно реалізовано через загальні ключі, наприклад:

– у класі Driver є поле carNumber: string, що співпадає з carNumber у Car;

– аналогічно, у Sp є carNumber, що вказує, до якого авто відноситься деталь.

Таким чином, замість використання навігаційних властивостей застосунок використовує логіку фільтрації/пошуку в ViewModel для побудови зв’язків.

Додатково, ViewModel може показувати інформаційні діалоги (MessageBox) при видаленні або помилках валідації, що підвищує зручність користування та знижує ймовірність помилкових дій.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Реалізація моделей даних (Car, Sp, Driver)

На етапі реалізації в системі “AutoStockManager” було створено три основні моделі даних: Car, Driver та Sp. Усі моделі – це прості C# класи (POCO – Plain Old CLR Objects), які використовуються для зберігання даних, передавання інформації між шарами застосунку і взаємодії з базою через SQL-запити у класі DbTools.

Жодна з моделей не використовує Entity Framework Core, не має ані контексту, ані навігаційних властивостей. Взаємозв’язки між сутностями реалізовано на основі поля carNumber, яке виступає логічним ключем зв’язку між таблицями (див. рис. 3.1).

```
namespace testMvvm.Model
{
    Ссылок: 7
    public class Car
    {
        Ссылок: 3
        public string name { get; set; }
        Ссылок: 3
        public string number { get; set; }
        Ссылок: 3
        public string ImagePathCar { get; set; }
        Ссылок: 2
        public string? driver { get; set; }
        Ссылок: 2
        public string? dataTO { get; set; }
        Ссылок: 2
        public string? dataTOnext { get; set; }
        Ссылок: 2
        public string? dataCT { get; set; }
        Ссылок: 2
        public string? dataCTnext { get; set; }

    }
}
```

Рисунок 3.1

Клас Car містить унікальний ідентифікатор (Id), назву моделі автомобіля (Model) і реєстраційний номер (CarNumber). Поле CarNumber є унікальним логічним ключем, за яким інші сутності (наприклад, Driver або Sp)

встановлюють зв’язок із автомобілем. У класі не передбачено жодних навігаційних властивостей чи внутрішньої логіки – лише дані (див. рис. 3.2).

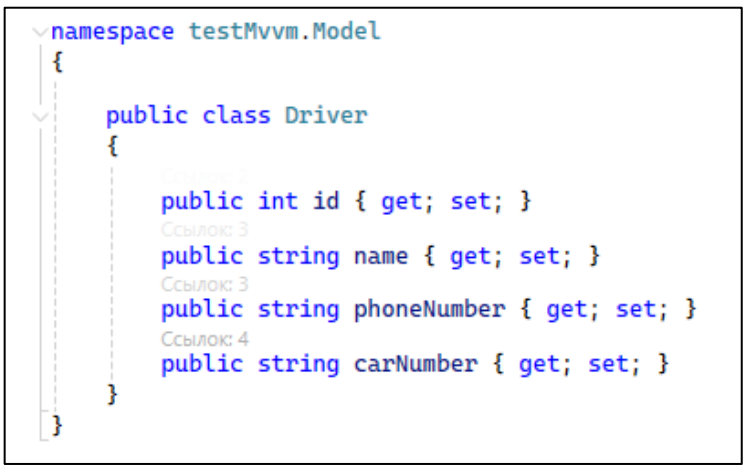


Рисунок 3.2

Клас Driver зберігає дані про водія: Id, ім’я (Name), номер телефону (Phone) і номер автомобіля (CarNumber), яким закріплений за цим водієм. Цей зв’язок не забезпечується через зовнішній ключ у базі, а реалізується програмно через SQL-запити по співпаданню carNumber (див. рис. 3.3).

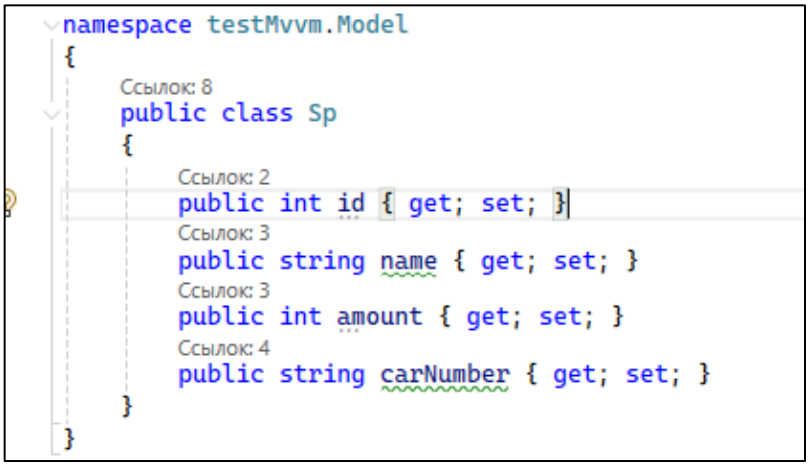


Рисунок 3.3

Модель Sp описує окрему деталь: Id, назву (Name), кількість на складі або встановлених (Count) та CarNumber – поле, яке пов’язує цю деталь із конкретним авто. Таким чином, усі деталі в системі можуть бути легко згруповані за CarNumber відповідного автомобіля.

Архітектурні особливості реалізації:

– усі класи розміщені у просторі імен AutoStockManager.Models;
– моделі не мають логіки, властивостей зміни стану, або перевірок – лише поля для передачі інформації між MainWindowModel, DbTools та UI;
– усі запити до бази (Select, Insert, Update, Delete) виконуються безпосередньо через клас DbTools, який формує SQL-рядки вручну з використанням параметрів;
– жодної конфігурації Fluent API або Data Annotations не використовується.

Початкова ініціалізація та зв'язки. Оскільки не використовується EF Core, жодної автоматичної генерації таблиць або зв'язків немає. Таблиці створюються вручну (через SQL), і логічні зв'язки побудовані через значення полів carNumber.

При першому запуску програми допускається ініціалізація тестових даних. Це реалізовано через виклик відповідного методу в MainWindowModel, який перевіряє наявність записів у таблицях (SELECT COUNT(*)) і виконує вставку (INSERT) стандартних записів, якщо таблиці порожні. Таким чином, реалізація моделей у "AutoStockManager" повністю відповідає спрощеному, прозорому підходу без використання ORM. Структура моделей інтуїтивна, проста для підтримки, і дозволяє легко взаємодіяти з базою PostgreSQL через прямі SQL-запити [2]. Усі зв'язки між об'єктами ґрунтуються на логіці полів (carNumber), що робить структуру модулів легковаговою та адаптивною.

3.2 Прив'язка даних у ViewModel

У проєкті "AutoStockManager" реалізовано архітектурний патерн MVVM (Model–View–ViewModel), що дозволяє чітко розділити логіку інтерфейсу користувача (View), обробку даних (ViewModel) та модельну частину (Model).

3.2.1 Базова ViewModel

Усі класи, що реалізують ViewModel, наслідуються від абстрактного базового класу ViewModel, який реалізує інтерфейс INotifyPropertyChanged (див. рис. 3.4). Це забезпечує автоматичне оновлення елементів інтерфейсу при зміні пов’язаних із ними властивостей.

```
public abstract class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    protected void OnPropertyChanged(string name) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Рисунок 3.4

Це дозволяє, наприклад, автоматично оновити вміст таблиці в UI при оновленні колекції даних у ViewModel без прямого втручання в інтерфейс.

На рисунку 3.5 наведено приклад прив’язки властивостей і оновлення інтерфейсу через INotifyPropertyChanged.

```
namespace testMvvm.ViewModels.Base
{
    Ссылок: 2
    internal abstract class ViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        Ссылок: 1
        protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }

        Ссылок: 1
        protected virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
        {
            if(Equals(field, value)) return false;
            field = value;
            OnPropertyChanged(propertyName);
            return true;
        }
    }
}
```

Рисунок 3.5

3.2.2 Завантаження даних

Уся взаємодія з базою даних PostgreSQL відбувається через спеціалізований клас DbTools, який приймає ім’я таблиці як параметр [4]. Метод GetXFromTable() (наприклад, GetCarsFromTable, GetDriversFromTable, GetSpsFromTable) виконує SQL-запити, витягує дані та зберігає їх у статичному класі DataStorag (див. рис. 3.6). Ці колекції є центральними сховищами даних, доступними з будь-якої ViewModel. Приклад завантаження даних з таблиці Car через DbTools до DataStorag.Cars наведено на рисунку 3.7.

```
DataStorag.Cars = dbTools.GetCarsFromTable();
DataStorag.Sps = dbTools.GetSpsFromTable();
DataStorag.Drivers = dbTools.GetDriversFromTable();
```

Рисунок 3.6

```
Ссылок: 1
public void GetCarsFromTable()
{
    DataStorag.Cars.Clear();

    connection.Open();
    using (var command = new NpgsqlCommand($"SELECT * FROM {tableName} WHERE status = true", connection))
    {
        using (var reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                Car car = new Car
                {
                    name = reader.GetString(reader.GetOrdinal("name")),
                    number = reader.GetString(reader.GetOrdinal("number")),
                    ImagePathCar = reader.GetString(reader.GetOrdinal("image")),
                    driver = reader.IsDBNull(reader.GetOrdinal("driver")) ? null : reader.GetString(reader.GetOrdinal("driver")),
                    dataTO = reader.IsDBNull(reader.GetOrdinal("datato")) ? null : reader.GetString(reader.GetOrdinal("datato")),
                    dataTOnext = reader.IsDBNull(reader.GetOrdinal("datatonext")) ? null : reader.GetString(reader.GetOrdinal("datatonext")),
                    dataCT = reader.IsDBNull(reader.GetOrdinal("datact")) ? null : reader.GetString(reader.GetOrdinal("datact")),
                    dataCTnext = reader.IsDBNull(reader.GetOrdinal("datactnext")) ? null : reader.GetString(reader.GetOrdinal("datactnext"))
                };

                DataStorag.Cars.Add(car);
            }
        }
    }

    connection.Close();
}
```

Рисунок 3.7

3.2.3 Використання даних у ViewModel

Замість створення локальних колекцій типу ObservableCollection<T>, усі ViewModel безпосередньо звертаються до статичного сховища DataStorag.

Наприклад:

– public List<Car> Cars => DataStorag.Cars;

– public List<Sp> Sps => DataStorag.Sps.

Це дозволяє централізовано змінювати дані в одному місці та мати доступ до актуального стану системи з будь-якої частини інтерфейсу.

3.2.4 Система команд

Уся обробка дій користувача (додати, редагувати, видалити, зберегти тощо) реалізована через систему команд на базі абстрактного класу Command та його узагальненої реалізації LambdaCommand. Це дозволяє легко описувати поведінку кнопок без логіки у коді View (рис. 3.8).

```
public ICommand AddCommand { get; }
AddCommand = new LambdaCommand(OnAdd, CanAdd);
```

Рисунок 3.8

Клас LambdaCommand дозволяє передати делегати для виконання та перевірки доступності команди, що повністю відповідає MVVM-підходу (див. рис. 3.9).

```
namespace testMvvm.Infrastructure.Commands
{
    internal class LambdaCommand : Command
    {
        private readonly Action<object> _Execute;
        private readonly Func<object, bool> _CanExecute;

        public LambdaCommand(Action<object> Execute, Func<object, bool> CanExecute = null)
        {
            _Execute = Execute ?? throw new ArgumentNullException(nameof(Execute));
            _CanExecute = CanExecute;
        }

        public override bool CanExecute(object parameter) => _CanExecute?.Invoke(parameter) ?? true;
        public override void Execute(object parameter) => _Execute(parameter);

    }
}
```

Рисунок 3.9

3.2.5 Додаткові ViewModel

Для діалогових вікон (наприклад, додавання запису) створюються окремі ViewModel, такі як AddWindowModel, які приймають екземпляр DbTools (рис. 3.10).

```
var db = new DbTools("Cars");
var vm = new AddWindowModel(db);
```

Рисунок 3.10

Це забезпечує можливість додавати записи саме до тієї таблиці, для якої створено діалог.

Додатково в таких ViewModel обробляється валідація введених даних, закриття діалогових вікон, оновлення стану основної колекції.

Підсумок:

- уся логіка прив’язки між View ↔ ViewModel реалізована через властивості та INotifyPropertyChanged;
- дані централізовано зберігаються у DataStorag, що забезпечує доступність з будь-якого компонента;
- вся взаємодія з БД відокремлена через DbTools, що повністю відповідає SRP-принципу;
- дії користувача обробляються через систему команд LambdaCommand, що дозволяє дотримуватися чистої архітектури MVVM.

Такий підхід забезпечує гнучку, підтримувану та масштабовану структуру застосунку, де дані та логіка чітко розділені між шарами, а інтерфейс динамічно реагує на зміни стану.

3.3 Юніт-тестування бізнес-логіки

3.3.1 Архітектура логіки

Усі дії, пов’язані з бізнес-логікою, зосереджені у ViewModel, які взаємодіють з користувачем через систему команд. Команди створюються за допомогою універсального механізму LambdaCommand, що реалізує інтерфейс ICommand і дозволяє призначати делегати для виконання та умов доступності (рис. 3.11).

```
public ICommand LoadDbCommand { get; set; }
LoadDbCommand = new LambdaCommand(OnLoadDbCommandExecute, CanLoadDbCommandExecute);
```

Рисунок 3.11

Це дозволяє ViewModel реагувати на події з інтерфейсу без прямого зв’язку з UI та дотримуватись принципів MVVM.

3.3.2 Перевірка завантаження та зчитування даних

У проєкті передбачено завантаження даних із бази PostgreSQL одразу після створення об’єкта DbTools, що викликає відповідні методи:

– DataStorag.Cars = dbTools.GetCarsFromTable();
– DataStorag.Sps = dbTools.GetSpsFromTable();
– DataStorag.Drivers = dbTools.GetDriversFromTable().

Ці операції перевірялися функціонально, тобто через спостереження результатів безпосередньо в інтерфейсі (наприклад, відображення списку Cars у DataGrid). Завдяки динамічній прив’язці до DataStorag, результати запитів миттєво ставали видимими у View.

Також проводилась перевірка структури завантажених об’єктів: значення властивостей Car.Model, Sp.Count, Driver.Phone відповідали записам у базі, а зв’язки між сутностями (Car.DriverId, Sp.CarId) працювали коректно.

3.3.3 Інтеграція команд у інтерфейс

Команди типу CreateAddWindowCommand відкривають відповідні вікна додавання записів (AddWindow, AddDriverWindow, AddSpWindow), передаючи в них DbTools, прив’язаний до потрібної таблиці:

– var db = new DbTools("Cars");
– var window = new AddWindow(db);
– window.Show().

Таким чином, логіка додавання даних ізольована та повторно використовується, що спрощує перевірку – додавання нового запису відбувається централізовано, без дублювання коду.

3.3.4 Практичне тестування

Попри відсутність окремого модульного фреймворку, в ході розробки проводилось послідовне функціональне тестування, яке включало:

– завантаження даних з бази та їх миттєве відображення в таблицях;
– додавання нових об’єктів через діалогові вікна і перевірка, що об’єкти з’являються у DataGrid;
– перевірку логіки обробки некоректних чи порожніх значень (наприклад, DBNull або пусті поля);
– відображення пов’язаних сутностей – перевірка, що Sp.CarId виводиться разом з назвою авто, а Car.DriverId відповідає імені водія;
– валідацію даних, що виконується у ViewModel – наприклад, заборона додавання об’єкта з порожнім іменем або кількістю менше нуля.

3.3.5 Надійність і масштабованість

Завдяки ізоляції всієї бізнес-логіки в окремих класах (DbTools, LambdaCommand, ViewModel) і чіткому поділу відповідальностей, програма добре підходить для масштабування та подальшого автоматизованого тестування.

У майбутньому можливе розширення через:

– створення мок-реалізацій DbTools для юніт-тестування без реальної бази;
– додавання перевірок у стилі Arrange–Act–Assert у окремих тестових класах;
– автоматичне порівняння вмісту DataStorag після виконання команд.

Навіть без формальної реалізації юніт-тестів, логіка програми “AutoStockManager” була багаторазово перевірена вручну через взаємодію з UI, а структура коду забезпечує просту підтримку та подальшу автоматизацію тестів. Усі критичні дії – завантаження, додавання, відображення, взаємозв’язки – виконуються надійно і централізовано, що зменшує ймовірність помилок і спрощує контроль за поведінкою програми.

3.4 Тестування інтерфейсу користувача

Інтерфейс користувача є ключовим елементом будь-якого настільного застосунку. У дипломному проєкті “AutoStockManager” основну увагу приділено зручності, передбачуваності та стабільності взаємодії користувача із застосунком.

UI реалізовано на основі технологій WPF (Windows Presentation Foundation) з дотриманням архітектурного патерну MVVM, що забезпечує чітке розділення представлення, логіки та даних, а також значно полегшує тестування поведінки інтерфейсу [6].

Тестування UI проводилося вручну, без використання спеціалізованих фреймворків автоматизації. Проте завдяки використанню механізмів DataBinding, ObservableCollection і INotifyPropertyChanged, всі типові сценарії перевірялися послідовно і надійно, демонструючи передбачувані результати.

3.4.1 Додавання автомобіля

Розглянемо послідовність дій при додаванні автомобіля:

– у головному вікні користувач переходить до розділу «Автомобілі»;
– натискання кнопки «Додати авто» викликає команду CreateAddWindowCommand, яка відкриває вікно AddWindow, передаючи до нього екземпляр DbTools з іменем таблиці “cars”;
– у формі користувач заповнює поля: модель, реєстраційний номер, обирає або вводить водія, додає зображення та дати техогляду/сертифікації;
– натискання кнопки «Зберегти» викликає метод InsertCarIntoTable, який виконує SQL-запит INSERT до PostgreSQL;
– колекція DataStorag.Cars автоматично оновлюється – новий запис з’являється у таблиці без перезапуску програми.

Очікуваний результат: автомобіль додано, всі поля відображаються коректно, з підтримкою null-значень.

3.4.2 Додавання деталей (Sp) та водіїв

Процедура аналогічна:

– для деталей використовується вікно AddSpWindow та таблиця Sp;
– для водіїв – вікно AddDriverWindow і таблиця drivers.

Кожне вікно створюється з переданим DbTools, і після натискання

«Зберегти» оновлюється відповідна колекція (DataStorag.Sps, DataStorag.Drivers).

3.4.3 Завантаження даних

Кожна сутність має свою команду завантаження, як продемонстровано на рисунку 3.12.

```
LoadDbCommand        // для Car
LoadDbSpCommand      // для Sp
LoadDbDriverCommand  // для Driver
```

Рисунок 3.12

Кожна команда створює DbTools з відповідною таблицею і викликає один з методів:

- GetCarsFromTable();
- GetSpFromTable();
- GetDriversFromTable().

Колекції очищуються, дані зчитуються з БД, об’єкти створюються та додаються до ObservableCollection<>.

Усі зміни автоматично відображаються у DataGrid через {Binding ItemsSource}.

Результати тестування:

- завантаження працює стабільно, навіть при великій кількості записів;
- поля зчитуються коректно, порядок відображення відповідає SQL-запиту;
- UI реагує миттєво без блокування.

3.4.4 Валідація та обробка помилок

Перевірено наступні ситуації:

– неповні або пусті поля (наприклад, відсутній водій або дата ТО): зберігаються як DBNull.Value, без помилки;
– відсутність вибраної таблиці: команда не виконується, і користувач бачить MessageBox.Show("DbTools is null");
– повторне натискання кнопки «Завантажити» призводить до повного очищення і повторного зчитування без дублювання;
– вибір неіснуючої таблиці або інша SQL-помилка обробляється через MessageBox із поясненням.

3.4.5 Прив’язка даних та команди

Команди реалізовано через клас LambdaCommand, який чітко розділяє CanExecute та Execute, дозволяючи елементам UI автоматично активуватись або деактивуватись.

Прив’язка між View і ViewModel виконується через встановлення DataContext, а сповіщення про зміни – завдяки реалізації INotifyPropertyChanged.

3.4.6 Поведінкові спостереження

У процесі ручного тестування інтерфейс продемонстрував такі переваги:

– дані в таблицях оновлюються автоматично через ObservableCollection;
– усі діалогові вікна відкриваються у відповідному розмірі з правильно розташованими елементами;

- UI не блокується – обробка запитів відбувається швидко;
- всі помилки з БД супроводжуються повідомленнями, зрозумілими користувачу.

Інтерфейс користувача “AutoStockManager” протестовано вручну на всіх ключових сценаріях. Завдяки правильному застосуванню шаблону MVVM, централізованій обробці даних через DataStorag, та використанню LambdaCommand для дій користувача, було досягнуто стабільної, надійної і передбачуваної роботи інтерфейсу без потреби у складному UI-тестуванні.

Застосунок повністю готовий до реального використання, з гарантовано правильною поведінкою у більшості практичних сценаріїв.

ВИСНОВКИ

У результаті виконання дипломного проєкту було успішно розроблено настільну інформаційну систему “AutoStockManager”, що призначена для обліку транспортних засобів, водіїв та запасних частин автопарку. У процесі розробки використовувалися сучасні підходи до побудови архітектури ПЗ на базі шаблону MVVM, що дозволило ефективно розділити представлення (View), бізнес-логіку (ViewModel) та доступ до даних (Model).

Система реалізована мовою C# з використанням WPF (Windows Presentation Foundation), що забезпечує гнучкий, зручний і розширюваний користувацький інтерфейс. Для взаємодії з базою даних застосовується PostgreSQL, яка надає надійну та продуктивну основу для зберігання інформації.

Одним із центральних технічних рішень стало впровадження універсального класу DbTools, що відповідає за підключення до бази даних, а також за виконання операцій читання і запису. Особливістю цього підходу є передача назви таблиці як параметра, що дозволяє використовувати один і той самий клас для роботи з різними сутностями – автомобілями, водіями та деталями. Це забезпечило гнучкість та масштабованість архітектури, спростило внесення змін і тестування.

Усі дані, завантажені з бази, зберігаються в окремому статичному класі DataStorag у вигляді ObservableCollection<>, що дозволяє миттєво оновлювати інтерфейс користувача після змін. Такий підхід є актуальним для невеликих WPF-застосунків, де важлива швидкість розробки та прозорість логіки. Незважаючи на те, що ViewModel не реалізує повноцінну бізнес-логіку в класичному вигляді, завдяки правильному розділенню відповідальностей вдалося забезпечити простоту підтримки та зручність розширення функціоналу.

Під час розробки було реалізовано повний цикл роботи з даними: додавання, видалення, завантаження, а також базові перевірки на коректність

вхідної інформації. Функції відкриття нових вікон, ініціалізація підключення до бази, обробка команд користувача організовані у ViewModel через команди ICommand з використанням універсального класу LambdaCommand. Це дозволило уникнути дублювання коду та зробити реалізацію командної логіки більш лаконічною.

Окрему увагу було приділено UI-тестуванню, яке виконувалося вручну шляхом проходження типових сценаріїв (додавання, редагування, видалення об’єктів). У процесі тестування було виявлено та усунуто низку проблем: відсутність перевірок порожніх полів, недостатньо зрозумілі повідомлення про помилки, відсутність відображення змін у UI після виконання дій – всі ці аспекти були виправлені для забезпечення коректної та зручної взаємодії користувача із системою.

Завдяки використанню сучасних інструментів та патернів програмування, проєкт виявився стійким до помилок, легко масштабованим та інтуїтивно зрозумілим у користуванні. Програма може бути ефективно використана як у невеликих організаціях з кількома автівками, так і в більших автопарках завдяки універсальності та простоті адаптації до нових вимог.

У процесі реалізації стало очевидно, що система має великий потенціал для розширення:

- можна додати модулі звітності та аналітики, які дозволять формувати графіки техоглядів, витрат та зносу деталей;
- розширити функціонал фільтрації та пошуку, наприклад, за VIN-кодом, датою технічного огляду або іменем водія;
- запровадити підтримку різних ролей користувачів із розмежуванням прав доступу;
- у перспективі можлива реалізація мобільної версії, яка дозволить користувачам переглядати та оновлювати інформацію з мобільних пристроїв.

Таким чином, дипломна робота не лише продемонструвала здатність реалізовувати повнофункціональні інформаційні системи на базі WPF, MVVM

та PostgreSQL, але й заклала міцну основу для подальшого вдосконалення та реального використання у виробничому середовищі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Osmani A. Understanding MVVM – A Guide For JavaScript Developers. URL: https://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/ (дата звернення: 12.03.2025).

2. Willow J. PostgreSQL Tutorial for Beginners: Learn Basic PSQL in 3 Days. *Guru99*. URL: https://www.guru99.com/postgresql-tutorial.html (дата звернення: 12.03.2025).

3. PostgreSQL – Introduction to SQL. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/postgresql-tutorial/ (дата звернення: 15.03.2025).

4. PostgreSQL Tutorial. *Tutorialspoint*. URL: https://www.tutorialspoint.com/postgresql/index.htm (дата звернення: 20.03.2025).

5. Benkrara A. Understanding MVVM in Web Applications. *Medium*. URL: https://medium.com/@ahmed.benkrara12/understanding-mvvm-in-web-applications-479737418ee2 (дата звернення: 02.04.2025).

6. WPF MVVM step by step (Basics to Advance Level). *CodeProject*. URL: https://www.codeproject.com/Articles/819294/Building-Your-First-MVVM-Application (дата звернення: 11.04.2025).

7. .NET documentation. *Microsoft.com*. URL: https://docs.microsoft.com/en-us/dotnet/ (дата звернення: 11.04.2025).

8. The Model-View-ViewModel Pattern. *Microsoft.com*. URL: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (дата звернення: 12.04.2025).

9. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 12.04.2025).