

# Лабораторная работа №4

## Содержание

|   |    |
|---|----|
| <b>1. Цель работы</b> ⚡                                     | 3  |
| <b>2. Теоретические сведения</b> 📄                          | 4  |
| <b>3. Порядок выполнения работы</b> 📅                       | 5  |
| 3.1. Первоначальная настройка базы данных                   | 5  |
| 3.2. Запуск проекта   | 5  |
| <b>4. Исходный код и комментарии</b> 💾                      | 6  |
| 4.1. RelayCommand.cs — реализация команды MVVM              | 6  |
| 4.2. StoreDbContext.cs — контекст базы данных               | 7  |
| 4.3. CartItem.cs — элемент корзины                          | 7  |
| 4.4. MainViewModel.cs — основная логика взаимодействия с UI | 8  |
| 4.5. Репозиторий ProductRepository.cs — операции с товарами | 11 |
| 4.6. Репозиторий OrderRepository.cs — работа с заказами     | 12 |
| 4.7. ProductViewModel.cs — управление данными товаров       | 14 |
| 4.8. OrderViewModel.cs — управление заказами                | 16 |
| <b>5. Работа с XAML-интерфейсом</b> 🔗                       | 20 |
| 5.1. Привязка контекста к ViewModel                         | 20 |
| 5.2. Отображение списка продуктов                           | 20 |
| 5.3. Команды кнопок   | 20 |
| <b>6. Примеры интерфейса пользователя</b> 💬                 | 21 |
| <b>7. Контрольные вопросы</b> 🤔                             | 24 |

## Листинг

|                                  |    |
|----------------------------------|----|
| 4.1 Класс RelayCommand           | 6  |
| 4.2 Контекст базы данных         | 7  |
| 4.3 Листинг «Корзины товаров»    | 8  |
| 4.4 Логика взаимодействия с UI   | 10 |
| 4.5 Основные операции с товарами | 12 |
| 4.6 Репозиторий OrderRepository  | 13 |

|     |  |    |
|-----|--|----|
| 4.7 | Код класса ProductViewModel . . . . .                          | 16 |
| 4.8 | Код класса OrderViewModel . . . . .                            | 18 |
| 5.1 | Привязка контекста к ViewModel . . . . .                       | 20 |
| 5.2 | Отображение списка продуктов . . . . .                         | 20 |
| 5.3 | Привязка кнопок к соответствующим командам ViewModel . . . . . | 20 |

## 1. Цель работы

Освоить принципы построения приложений на WPF с применением паттерна MVVM, базы данных SQLite и технологии Entity Framework Core. Разработать функциональность управления товарами и заказами в рамках настольного приложения.

**Задание:**

1. Изучить архитектуру MVVM и реализовать её на практике.
2. Использовать Entity Framework Core для доступа к базе данных.
3. Организовать взаимодействие между слоями модели, представления и логики представления.
4. Настроить графический интерфейс пользователя для работы с товарами и заказами.
5. Реализовать команды добавления, редактирования и удаления данных.

## 2. Теоретические сведения

StoreManager — это настольное WPF-приложение на платформе .NET, реализующее базовую функциональность управления магазином: добавление/редактирование товаров, формирование заказов, управление корзиной, отображение и удаление заказов. Проект построен по принципам MVVM (Model-View-ViewModel), использует Entity Framework Core для работы с SQLite базой данных.

### Основные компоненты:

- Model: классы `Product`, `Order`, `OrderItem`, `CartItem` — описывают бизнес-логику и данные.
- ViewModel: классы `MainViewModel`, `ProductViewModel`, `OrderViewModel` управляют состоянием и взаимодействием между представлением и моделью.
- View: XAML-интерфейс, представляющий UI, привязанный к ViewModel.

### Используемые технологии:

- C#, .NET
- WPF (XAML)
- Entity Framework Core
- SQLite
- MVVM-паттерн

### 3. Порядок выполнения работы №3

В данном разделе последовательно описан процесс подготовки, запуска и анализа исходного кода проекта. Необходимо **строго** придерживаться шагов, описанных ниже, чтобы успешно воспроизвести и исследовать функциональность приложения.

1. Для начала необходимо открыть проект в среде разработки Visual Studio.
2. Далее необходимо установить требуемые библиотеки через NuGet Package Manager. Список необходимых NuGet-пакетов:
  - Microsoft.EntityFrameworkCore
  - Microsoft.EntityFrameworkCore.Sqlite
  - Microsoft.Data.Sqlite

#### 3.1. Первичная настройка базы данных

После установки зависимостей необходимо выполнить инициализацию базы данных. При первом запуске приложение автоматически создаёт файл базы данных `store.db` с помощью метода `EnsureCreated()`.

#### 3.2. Запуск проекта

После выполнения всех предыдущих шагов необходимо запустить приложение. Проект `StoreManager_4lab` должен быть выбран в качестве стартового.

## 4. Исходный код и комментарии

В данном разделе рассмотрены ключевые компоненты проекта, реализующие бизнес-логику и связь с пользовательским интерфейсом. Далее последовательно представлены листинги кода с пояснениями.

### 4.1. RelayCommand.cs — реализация команды MVVM

```
1 public class RelayCommand : ICommand
2 {
3     private readonly Action<object> _execute; // делегат выполнения команды
4     private readonly Func<object, bool> _canExecute; // делегат проверки возможности выполнения
5         → команды
6
7     // Конструктор команды
8     public RelayCommand(Action<object> execute, Func<object, bool> canExecute = null)
9     {
10         _execute = execute ?? throw new ArgumentNullException(nameof(execute)); // инициализация
11             → метода выполнения
12         _canExecute = canExecute; // инициализация метода проверки доступности
13     }
14
15     // Событие, которое вызывается при изменении условий выполнения команды
16     public event EventHandler CanExecuteChanged
17     {
18         add { CommandManager.RequerySuggested += value; }
19         remove { CommandManager.RequerySuggested -= value; }
20     }
21
22     // Проверка, может ли команда быть выполнена
23     public bool CanExecute(object parameter) => _canExecute?.Invoke(parameter) ?? true;
24
25     // Выполнение команды
26     public void Execute(object parameter) => _execute(parameter);
```

Листинг 4.1 – Класс RelayCommand

В листинге 4.1 представлен класс `RelayCommand`. Данный класс реализует интерфейс `ICommand`, что позволяет привязывать действия в UI к методам `ViewModel`. Применяется для реализации кнопок и команд.

1. `RelayCommand(...)` — конструктор команды. Принимает делегаты выполнения и условия выполнения. Используется при создании команд в `ViewModel`.
2. `CanExecute(object parameter)` — возвращает `true`, если команду можно выполнить. Применяется для активации/деактивации элементов интерфейса.

3. Execute(object parameter) — запускает делегат действия команды. Вызывается при клике на кнопку или другое действие пользователя.

## 4.2. StoreDbContext.cs — контекст базы данных

```
1 public class StoreDbContext : DbContext
2 {
3     // Таблица товаров
4     public DbSet<Product> Products { get; set; }
5     // Таблица заказов
6     public DbSet<Order> Orders { get; set; }
7     // Таблица позиций заказа
8     public DbSet<OrderItem> OrderItems { get; set; }
9     // Конфигурация подключения к базе данных
10    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
11    {
12        optionsBuilder.UseSqlite("Data Source=store.db"); // указание строки подключения к SQLite
13    }
14    // Настройка модели данных
15    protected override void OnModelCreating(ModelBuilder modelBuilder)
16    {
17        modelBuilder.Entity<Order>()
18            .HasMany(o => o.Items) // один заказ содержит множество позиций
19            .WithOne()           // без явной обратной связи
20            .OnDelete(DeleteBehavior.Cascade); // при удалении заказа удаляются все его позиции
21    }
22 }
```

Листинг 4.2 – Контекст базы данных

1. OnConfiguring(DbContextOptionsBuilder optionsBuilder) — вызывается при создании экземпляра контекста. Устанавливает строку подключения к базе данных SQLite.
2. OnModelCreating(ModelBuilder modelBuilder) — выполняется при инициализации модели EF. Здесь настраивается каскадное удаление зависимых сущностей (например, позиций заказа при удалении заказа). Определяет три таблицы: Product, Order, OrderItem.

## 4.3. CartItem.cs — элемент корзины

Далее представлен листинг класса CartItem, отвечающего за временное хранение товаров (Листинг 4.3).

```
1 public class CartItem : INotifyPropertyChanged
2 {
```

```
3     public Product Product { get; set; } // ссылка на продукт
4
5     private int _quantity; // поле для хранения количества
6
7     public int Quantity
8     {
9         get => _quantity; // получение значения
10        set { _quantity = value; OnPropertyChanged(); } // установка значения и уведомление об
11           → изменении
12    }
13
14    // Событие, уведомляющее об изменении свойства
15    public event PropertyChangedEventHandler PropertyChanged;
16
17    // Метод вызова события изменения свойства
18    protected void OnPropertyChanged([CallerMemberName] string name = null) =>
19        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

Листинг 4.3 – Листинг «Корзины товаров»

CartItem служит для временного хранения товаров в корзине перед созданием заказа. Используется в ObservableCollection<CartItem>.

1. Product — ссылка на экземпляр товара, связанный с объектом корзины.
2. Quantity — количество выбранного товара в корзине. При изменении уведомляет UI через механизм INotifyPropertyChanged.
3. OnPropertyChanged(string name) — метод, вызывающий событие PropertyChanged для обновления интерфейса при изменении свойств.

#### 4.4. MainViewModel.cs — основная логика взаимодействия с UI

```
1  public class MainViewModel : INotifyPropertyChanged
2  {
3      // Контекст базы данных
4      private readonly StoreDbContext _context = new();
5
6      // Коллекция всех товаров
7      public ObservableCollection<Product> Products { get; set; } = new();
8
9      // Коллекция всех заказов
10     public ObservableCollection<Order> Orders { get; set; } = new();
11
12     // Коллекция товаров, добавленных в корзину
13     public ObservableCollection<CartItem> CartItems { get; } = new();
14
```

```
15     // Команда добавления товара
16     public ICommand AddProductCommand { get; }
17
18     // Команда создания заказа
19     public ICommand AddOrderCommand { get; }
20
21     // Конструктор ViewModel
22     public MainViewModel()
23     {
24         // Инициализация команд
25         AddProductCommand = new RelayCommand(_ => AddProduct(), _ => CanAddProduct());
26         AddOrderCommand = new RelayCommand(_ => AddOrder());
27
28         // Создание базы данных при первом запуске
29         _context.Database.EnsureCreated();
30
31         // Загрузка данных из базы
32         LoadProducts();
33         LoadOrders();
34     }
35
36     // Загрузка всех товаров из базы данных
37     private void LoadProducts()
38     {
39         Products.Clear();
40         foreach (var p in _context.Products.ToList())
41             Products.Add(p);
42     }
43
44     // Загрузка всех заказов с товарами из базы данных
45     private void LoadOrders()
46     {
47         Orders.Clear();
48         var allOrders = _context.Orders
49             .Include(o => o.Items)
50             .ThenInclude(i => i.Product)
51             .ToList();
52
53         foreach (var order in allOrders)
54             Orders.Add(order);
55     }
56
57     // Добавление нового товара
58     private void AddProduct()
59     {
60         var product = new Product
61         {
62             Name = "Новый товар",           // название товара
63             Price = 100,                  // цена по умолчанию
64             Stock = 10,                   // количество по умолчанию
65             Category = "Общая"           // категория товара
66         };
67     }
68 }
```

```
66     };
67
68     _context.Products.Add(product);      // добавление в базу данных
69     _context.SaveChanges();            // сохранение изменений
70     LoadProducts();                  // обновление списка товаров
71 }
72
73 // Добавление нового заказа
74 private void AddOrder()
75 {
76     if (!CartItems.Any()) return; // проверка наличия товаров в корзине
77
78     var order = new Order
79     {
80         CustomerName = "Клиент", // имя покупателя
81         OrderDate = DateTime.Now, // дата оформления
82         Items = CartItems.Select(c => new OrderItem
83         {
84             Product = c.Product,    // товар
85             Quantity = c.Quantity // количество
86         }).ToList()
87     };
88
89     _context.Orders.Add(order); // добавление заказа в БД
90     _context.SaveChanges();   // сохранение
91     LoadOrders();           // обновление списка
92     CartItems.Clear();       // очистка корзины
93 }
94
95 // Условие доступности команды добавления товара
96 private bool CanAddProduct() => true;
97
98 public event PropertyChangedEventHandler PropertyChanged;
99
100 // Обработка события изменения свойства
101 protected void OnPropertyChanged([CallerMemberName] string name = null) =>
102     PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
103 }
```

Листинг 4.4 – Логика взаимодействия с UI

В листинге 4.4 представлен код класса `MainViewModel`. Он связывает данные с пользовательским интерфейсом. В нём реализованы команды для добавления товаров и заказов, загрузка данных из базы и взаимодействие с коллекциями. Этот класс является ядром логики MVVM и предоставляет интерфейс между XAML и данными модели.

1. `MainViewModel()` — конструктор класса. Производит инициализацию базы, загрузку данных и настройку команд.

2. LoadProducts() — загружает все товары из базы данных в ObservableCollection для отображения.
3. LoadOrders() — загружает все заказы и их связанные товары.
4. AddProduct() — добавляет новый товар в базу и обновляет отображаемый список.
5. AddOrder() — формирует и сохраняет заказ на основе содержимого корзины.
6. CanAddProduct() — всегда возвращает true, используется как условие для активности команды добавления товара.
7. OnPropertyChanged(string name) — реализует уведомление об изменении свойства, используется в привязках.

## 4.5. Репозиторий ProductRepository.cs — операции с товарами

```
1 public class ProductRepository
2 {
3     private const string ConnectionString = "Data Source=store.db"; // строка подключения к базе
4         → данных
5
6     public List<Product> GetAll()
7     {
8         var list = new List<Product>(); // список для хранения всех продуктов
9         using var connection = new SqliteConnection(ConnectionString); // открытие соединения с
10            → базой данных
11         connection.Open(); // выполнение подключения к базе данных
12
13         var command = connection.CreateCommand(); // создание SQL-команды
14         command.CommandText = "SELECT * FROM Product"; // выборка всех записей из таблицы товаров
15
16         using var reader = command.ExecuteReader(); // выполнение команды и получение результатов
17         while (reader.Read()) // построчное чтение результатов
18         {
19             list.Add(new Product // добавление экземпляра товара в список
20             {
21                 Id = reader.GetInt32(0), // идентификатор товара
22                 Name = reader.GetString(1), // название товара
23                 Price = reader.GetDecimal(2), // цена товара
24                 Stock = reader.GetInt32(3), // остаток товара на складе
25                 Category = reader.IsDBNull(4) ? null : reader.GetString(4) // категория товара
26                     → (может отсутствовать)
27             });
28         }
29     }
30
31     return list; // возвращение списка всех товаров
32 }
```

```
29     public void Add(Product product)
30     {
31         using var connection = new SqliteConnection(ConnectionString); // открытие соединения с
32             ← базой данных
33         connection.Open(); // выполнение подключения к БД
34
35         var command = connection.CreateCommand(); // создание SQL-команды
36         command.CommandText = "INSERT INTO Product (Name, Price, Stock, Category) VALUES (@name,
37             ← @price, @stock, @category)"; // вставка нового товара в таблицу Product
38         command.Parameters.AddWithValue("@name", product.Name); // название товара
39         command.Parameters.AddWithValue("@price", product.Price); // цена товара
40         command.Parameters.AddWithValue("@stock", product.Stock); // количество товара на складе
41         command.Parameters.AddWithValue("@category", product.Category ?? (object)DBNull.Value); // ←
42             категория товара (или null)
43
44         command.ExecuteNonQuery(); // выполнение SQL-команды
45     }
46 }
```

Листинг 4.5 – Основные операции с товарами

Класс ProductRepository предоставляет низкоуровневый доступ к таблице товаров. В нём реализованы методы чтения и добавления продуктов с использованием SQLite.

1. GetAll() — выполняет подключение к базе данных и извлекает все товары из таблицы Product.  
Используется при загрузке списка товаров в интерфейсе.
2. Add(Product product) — добавляет новый товар в базу данных с помощью SQL-запроса INSERT.  
Используется при создании новых товаров пользователем.

## 4.6. Репозиторий OrderRepository.cs — работа с заказами

```
1  public class OrderRepository
2  {
3      private const string ConnectionString = "Data Source=store.db"; // строка подключения к БД
4
5      public List<Order> GetAll(List<Product> allProducts)
6      {
7          var orders = new List<Order>(); // список заказов
8          using var connection = new SqliteConnection(ConnectionString); // открытие соединения с
9              ← базой данных
10             connection.Open(); // выполнение подключения
11
12             var command = connection.CreateCommand(); // создание SQL-команды
13             command.CommandText = "SELECT Id, CustomerName, OrderDate FROM [Order]"; // выборка данных
14                 ← заказов
15             using var reader = command.ExecuteReader(); // выполнение запроса и чтение результатов
16     }
17 }
```

```
15     while (reader.Read()) // построчное чтение результата
16     {
17         var order = new Order
18         {
19             Id = reader.GetInt32(0), // идентификатор заказа
20             CustomerName = reader.GetString(1), // имя клиента
21             OrderDate = DateTime.Parse(reader.GetString(2)), // дата оформления заказа
22             Items = new List<OrderItem>() // список товаров в заказе
23         };
24         orders.Add(order); // добавление заказа в коллекцию
25     }
26
27     foreach (var order in orders) // обход заказов для загрузки позиций
28     {
29         var itemCmd = connection.CreateCommand(); // создание SQL-команды для выбора позиций
30         ← заказа
31         itemCmd.CommandText = "SELECT ProductId, Quantity FROM OrderItem WHERE OrderId =
32             @orderId"; // выборка товарных позиций
33         itemCmd.Parameters.AddWithValue("@orderId", order.Id); // параметр: идентификатор заказа
34         var itemReader = itemCmd.ExecuteReader(); // выполнение запроса
35
36         while (itemReader.Read()) // построчное чтение позиций
37         {
38             var productId = itemReader.GetInt32(0); // идентификатор товара
39             var quantity = itemReader.GetInt32(1); // количество товара
40             var product = allProducts.FirstOrDefault(p => p.Id == productId); // поиск товара в
41             ← общем списке
42
43             if (product != null)
44             {
45                 order.Items.Add(new OrderItem // добавление товара в список позиций заказа
46                 {
47                     Product = product,
48                     Quantity = quantity
49                 });
50             }
51         }
52     }
53 }
```

#### Листинг 4.6 – Репозиторий OrderRepository

В листинге 4.6 представлен код класса `OrderRepository`, который позволяет загружать заказы с их товарами из базы данных. Используется прямое взаимодействие с SQLite для выборки и объединения данных. Этот репозиторий играет роль интерфейса между бизнес-логикой и хранилищем заказов.

1.  `GetAll(List<Product> allProducts)` — извлекает список всех заказов из таблицы `Order`, а также загружает связанные позиции из таблицы `OrderItem`.

2. `Add(Order order)` — добавляет новый заказ и его позиции в базу данных в рамках транзакции. Используется при оформлении нового заказа.

3. `Delete(int orderId)` — удаляет заказ по идентификатору из таблицы `Order`. При этом связанные позиции удаляются автоматически (если настроено каскадное удаление).

## 4.7. ProductViewModel.cs — управление данными товаров

```
1 public class ProductViewModel : BaseViewModel
2 {
3     private Product _selectedProduct; // выбранный товар
4     private readonly StoreDbContext _context; // контекст базы данных
5     private string _filterCategory; // выбранная категория для фильтрации
6
7     public ObservableCollection<Product> Products { get; } = new(); // список всех товаров
8     public ObservableCollection<string> Categories { get; } = new(); // список доступных категорий
9
10    // Свойство выбранного товара
11    public Product SelectedProduct
12    {
13        get => _selectedProduct;
14        set => SetProperty(ref _selectedProduct, value);
15    }
16
17    // Свойство фильтрации по категории
18    public string FilterCategory
19    {
20        get => _filterCategory;
21        set
22        {
23            if (SetProperty(ref _filterCategory, value))
24            {
25                LoadProducts(); // при изменении категории перезагружается список
26            }
27        }
28    }
29
30    // Команды управления
31    public ICommand AddProductCommand { get; }
32    public ICommand UpdateProductCommand { get; }
33    public ICommand DeleteProductCommand { get; }
34
35    // Конструктор
36    public ProductViewModel()
37    {
38        _context = new StoreDbContext(); // инициализация контекста
39        _context.Database.EnsureCreated(); // создание базы при необходимости
40
41        LoadCategories(); // загрузка всех категорий
```

```
42     LoadProducts(); // загрузка всех товаров
43
44     // инициализация команд
45     AddProductCommand = new RelayCommand(_ => AddProduct());
46     UpdateProductCommand = new RelayCommand(_ => UpdateProduct(), _ => SelectedProduct != null);
47     DeleteProductCommand = new RelayCommand(_ => DeleteProduct(), _ => SelectedProduct != null);
48 }
49
50 // Загрузка всех товаров (с фильтрацией, если указана категория)
51 private void LoadProducts()
52 {
53     Products.Clear();
54     var products = _context.Products.AsQueryable();
55
56     if (!string.IsNullOrWhiteSpace(FilterCategory))
57         products = products.Where(p => p.Category == FilterCategory);
58
59     foreach (var product in products.ToList())
60         Products.Add(product);
61 }
62
63 // Загрузка уникальных категорий товаров
64 private void LoadCategories()
65 {
66     Categories.Clear();
67     var categories = _context.Products.Select(p => p.Category).Distinct().ToList();
68     foreach (var category in categories)
69         Categories.Add(category);
70 }
71
72 // Добавление нового товара
73 private void AddProduct()
74 {
75     var product = new Product { Name = "Новый товар", Price = 0, Stock = 0, Category = "Общая"
76     → };
77     _context.Products.Add(product);
78     _context.SaveChanges();
79     Products.Add(product);
80     SelectedProduct = product;
81     LoadCategories();
82 }
83
84 // Обновление выбранного товара
85 private void UpdateProduct()
86 {
87     _context.Products.Update(SelectedProduct);
88     _context.SaveChanges();
89     LoadProducts();
90 }
91 // Удаление выбранного товара
```

```
92     private void DeleteProduct()
93     {
94         if (SelectedProduct != null)
95         {
96             _context.Products.Remove(SelectedProduct);
97             _context.SaveChanges();
98             Products.Remove(SelectedProduct);
99             SelectedProduct = null;
100            LoadCategories();
101        }
102    }
103 }
```

Листинг 4.7 – Код класса ProductViewModel

Листинг 4.7 отражает код класса `ProductViewModel`. Эта модель управляет отображением и изменением информации о товарах. Поддерживает добавление, редактирование, удаление и фильтрацию по категориям. Используется для управления интерфейсом раздела «Товары».

1. `ProductViewModel()` — конструктор, инициализирует базу, загружает категории и товары, а также назначает команды.
2. `LoadProducts()` — загружает все товары из базы, с учётом установленной категории фильтрации.
3. `LoadCategories()` — извлекает уникальные категории товаров для отображения в выпадающем списке.
4. `AddProduct()` — создаёт новый товар, сохраняет его в базу и обновляет интерфейс.
5. `UpdateProduct()` — сохраняет изменения выбранного товара и обновляет список.
6. `DeleteProduct()` — удаляет выбранный товар из базы и обновляет список, если товар не используется в заказах.

## 4.8. OrderViewModel.cs — управление заказами

```
1 public class OrderViewModel : BaseViewModel
2 {
3     private readonly StoreDbContext _context; // контекст базы данных
4     private Order _selectedOrder; // выбранный заказ
5     private string _customerName; // имя клиента
6
7     public ObservableCollection<Order> Orders { get; } = new(); // список заказов
8     public ObservableCollection<Product> AvailableProducts { get; } = new(); // доступные товары
```

```
9     public ObservableCollection<OrderItem> SelectedItems { get; } = new(); // позиции текущего
10    →   заказа
11
12    public string CustomerName
13    {
14        get => _customerName;
15        set => SetProperty(ref _customerName, value); // установка имени клиента
16    }
17
18    public Order SelectedOrder
19    {
20        get => _selectedOrder;
21        set
22        {
23            if (SetProperty(ref _selectedOrder, value) && value != null)
24            {
25                CustomerName = value.CustomerName; // установка имени из выбранного заказа
26                SelectedItems.Clear();
27                foreach (var item in value.Items)
28                    SelectedItems.Add(item); // заполнение списка позиций заказа
29            }
30        }
31    }
32
33    // Команды управления заказами
34    public ICommand AddOrderCommand { get; }
35    public ICommand UpdateOrderCommand { get; }
36    public ICommand DeleteOrderCommand { get; }
37
38    // Конструктор
39    public OrderViewModel()
40    {
41        _context = new StoreDbContext();
42        _context.Database.EnsureCreated();
43
44        LoadOrders(); // загрузка заказов из БД
45        LoadAvailableProducts(); // загрузка доступных товаров
46
47        AddOrderCommand = new RelayCommand(_ => AddOrder());
48        UpdateOrderCommand = new RelayCommand(_ => UpdateOrder(), _ => SelectedOrder != null);
49        DeleteOrderCommand = new RelayCommand(_ => DeleteOrder(), _ => SelectedOrder != null);
50    }
51
52    // Загрузка всех заказов с товарами
53    private void LoadOrders()
54    {
55        Orders.Clear();
56        foreach (var order in _context.Orders.Include(o => o.Items).ThenInclude(i =>
57            → i.Product).ToList())
58            Orders.Add(order);
59    }
60
61    // Загрузка доступных товаров
62    private void LoadAvailableProducts()
63    {
64        AvailableProducts.Clear();
65        foreach (var product in _context.Products.ToList())
66        {
67            AvailableProducts.Add(product);
68        }
69    }
70
71    // Добавление позиции в заказ
72    public void AddOrderItem(OrderItem item)
73    {
74        if (SelectedOrder != null)
75        {
76            SelectedOrder.Items.Add(item);
77        }
78    }
79
80    // Обработка изменения имени клиента
81    public void OnCustomerNameChanged(string name)
82    {
83        if (SelectedOrder != null)
84        {
85            SelectedOrder.CustomerName = name;
86        }
87    }
88
89    // Установка нового заказа
90    public void SetSelectedOrder(Order order)
91    {
92        if (order != null)
93        {
94            SelectedOrder = order;
95        }
96    }
97
98    // Удаление заказа
99    public void DeleteOrder()
100   {
101      if (SelectedOrder != null)
102      {
103          _context.Orders.Remove(SelectedOrder);
104          _context.SaveChanges();
105      }
106  }
```

```
58         AvailableProducts.Add(product);
59     }
60     // Добавление нового заказа
61     private void AddOrder()
62     {
63         if (string.IsNullOrWhiteSpace(CustomerName) || SelectedItems.Count == 0) return;
64         var newOrder = new Order
65         {
66             CustomerName = CustomerName,
67             OrderDate = DateTime.Now,
68             Items = SelectedItems.Select(i => new OrderItem
69             {
70                 Product = i.Product,
71                 Quantity = i.Quantity
72             }).ToList()
73         };
74         _context.Orders.Add(newOrder);
75         _context.SaveChanges();
76         Orders.Add(newOrder); // добавление заказа в список
77     }
78     // Обновление существующего заказа
79     private void UpdateOrder()
80     {
81         if (SelectedOrder == null) return;
82         SelectedOrder.CustomerName = CustomerName;
83         SelectedOrder.Items = SelectedItems.Select(i => new OrderItem
84         {
85             Product = i.Product,
86             Quantity = i.Quantity
87         }).ToList();
88
89         _context.Orders.Update(SelectedOrder);
90         _context.SaveChanges();
91     }
92     // Удаление заказа
93     private void DeleteOrder()
94     {
95         if (SelectedOrder != null)
96         {
97             _context.Orders.Remove(SelectedOrder);
98             _context.SaveChanges();
99             Orders.Remove(SelectedOrder);
100            SelectedOrder = null;
101            SelectedItems.Clear();
102            CustomerName = string.Empty;
103        }
104    }
105 }
```

Листинг 4.8 – Код класса OrderViewModel

1. OrderViewModel() — конструктор. Создаёт контекст базы данных, загружает заказы и товары, и инициализирует команды.
2. LoadOrders() — загружает все существующие заказы с их товарными позициями.
3. LoadAvailableProducts() — загружает список товаров, доступных для добавления в заказ.
4. AddOrder() — создаёт новый заказ на основе текущего списка позиций и сохраняет его в базу.
5. UpdateOrder() — обновляет информацию выбранного заказа.
6. DeleteOrder() — удаляет выбранный заказ и сбрасывает связанные данные.
7. OnPropertyChanged(...) — уведомляет об изменениях свойств для обновления интерфейса.

## 5. Работа с XAML-интерфейсом </>

На завершающем этапе необходимо рассмотреть элементы графического интерфейса. Ниже представлены ключевые фрагменты XAML, демонстрирующие организацию интерфейса. Включены только ключевые элементы.

### 5.1. Привязка контекста к ViewModel

```
1 <Window.DataContext>
2   <vm:MainViewModel />
3 </Window.DataContext>
```

Листинг 5.1 – Привязка контекста к ViewModel

Листинг 5.1 отражает привязку контекста `Window.DataContext` к `ViewModel`. Это позволяет назначить `MainViewModel` как источник данных для всех привязок в окне.

### 5.2. Отображение списка продуктов

```
1 <ListBox ItemsSource="{Binding Products}" SelectedItem="{Binding SelectedProduct}">
2   <ListBox.ItemTemplate>
3     <DataTemplate>
4       <StackPanel Orientation="Horizontal">
5         <TextBlock Text="{Binding Name}" />
6         <TextBlock Text="{Binding Price}" />
7       </StackPanel>
8     </DataTemplate>
9   </ListBox.ItemTemplate>
10 </ListBox>
```

Листинг 5.2 – Отображение списка продуктов

На листинге 5.2 приведен пример отображения списка товаров с привязкой к `Products` и выбранным элементом — `SelectedProduct`.

### 5.3. Команды кнопок

Каждая кнопка привязана к определенному обработчику событий (методу), что позволяет выполнять необходимые операции при нажатии кнопки пользователем. Пример разметки для кнопок приведен на листинге 5.3.

```
1 <Button Content="Добавить" Command="{Binding AddProductCommand}" />
2 <Button Content="Обновить" Command="{Binding UpdateProductCommand}" />
3 <Button Content="Удалить" Command="{Binding DeleteProductCommand}" />
```

Листинг 5.3 – Привязка кнопок к соответствующим командам ViewModel

## 6. Примеры интерфейса пользователя

По завершению работы необходимо ознакомиться с внешним видом приложения. Ниже приведены фрагменты интерфейса с пояснениями. Ниже представлены иллюстрации ключевых элементов интерфейса (Рисунок 6.1 – 6.5):

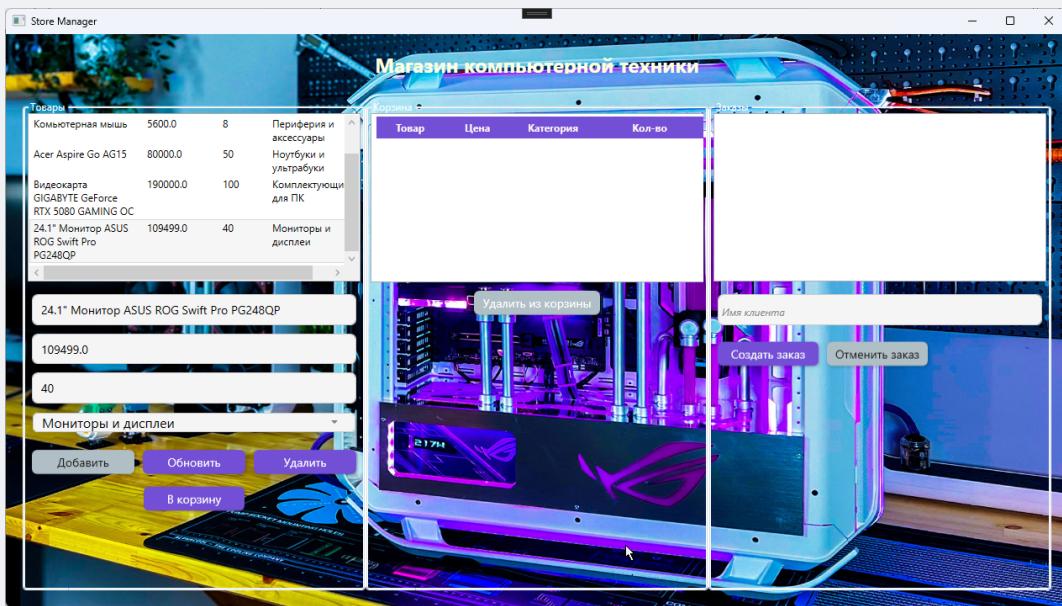


Рисунок 6.1 – Главная форма приложения с разделами «Товары», «Корзина» и «Заказы».

Пользователь может добавлять новые товары, а также редактировать уже созданные с помощью соответствующих полей и кнопок. Для редактирования товара необходимо выбрать его в списке, нажать на него и его данные будут подставлены в соответствующие поля. После изменения информации необходимо нажать кнопку «Обновить». Для добавления нового товара необходимо ввести данные в соответствующие поля и нажать кнопку «Добавить».

При нажатии кнопки «В корзину» выбранные товары добавляются в блок «Корзина». У пользователя есть возможность управлять количеством товаров, увеличивая или уменьшая его. Ограничение – количество товаров на складе. Добавление товаров в корзину изображено на рисунке 6.2.

## Примеры интерфейса пользователя

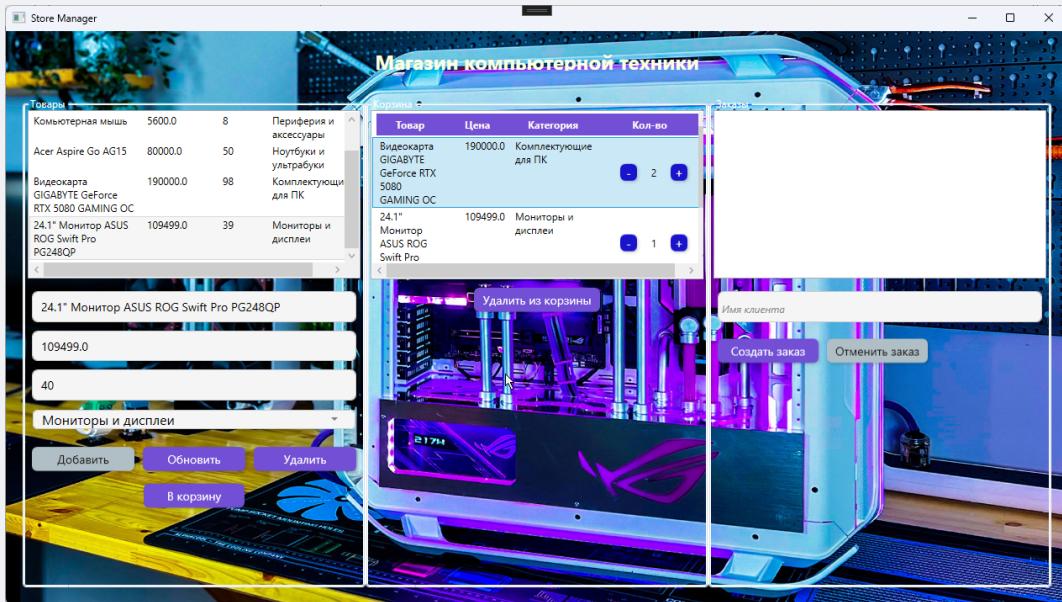


Рисунок 6.2 – Добавление товаров в «Корзину».

Также у пользователя есть возможность удалить товар из корзины с помощью кнопки «Удалить из корзины» (Рисунок 6.3). После удаления товара соответствующее количество товара вернется на склад.

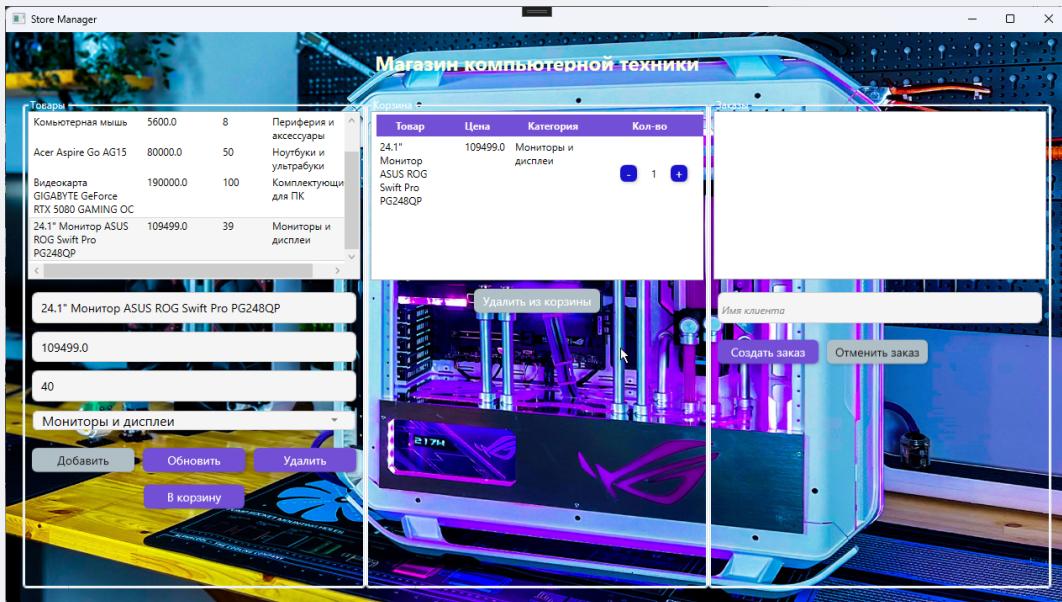


Рисунок 6.3 – Удаление выбранного товара.

После выбора необходимых товаров можно добавить новый заказ. Для создания заказа необходимо ввести имя клиента в соответствующее поле и нажать кнопку «Создать заказ» (Рисунок 6.4).

## Примеры интерфейса пользователя

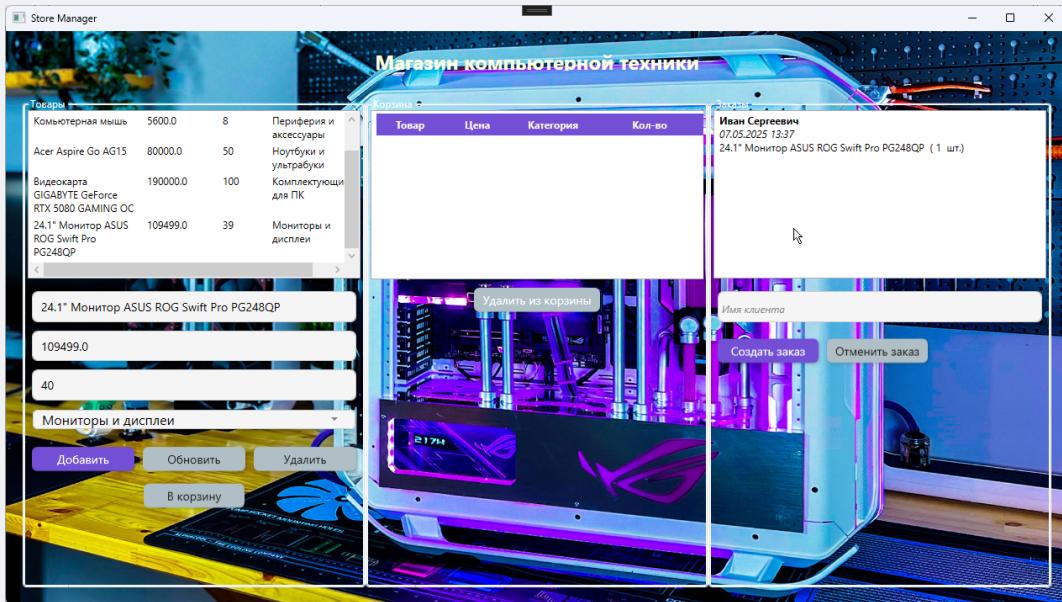


Рисунок 6.4 – Пример создания заказа.

При попытке добавить новый заказ не добавляя товары в корзину будет выведено соответствующее сообщение (Рисунок 6.5).

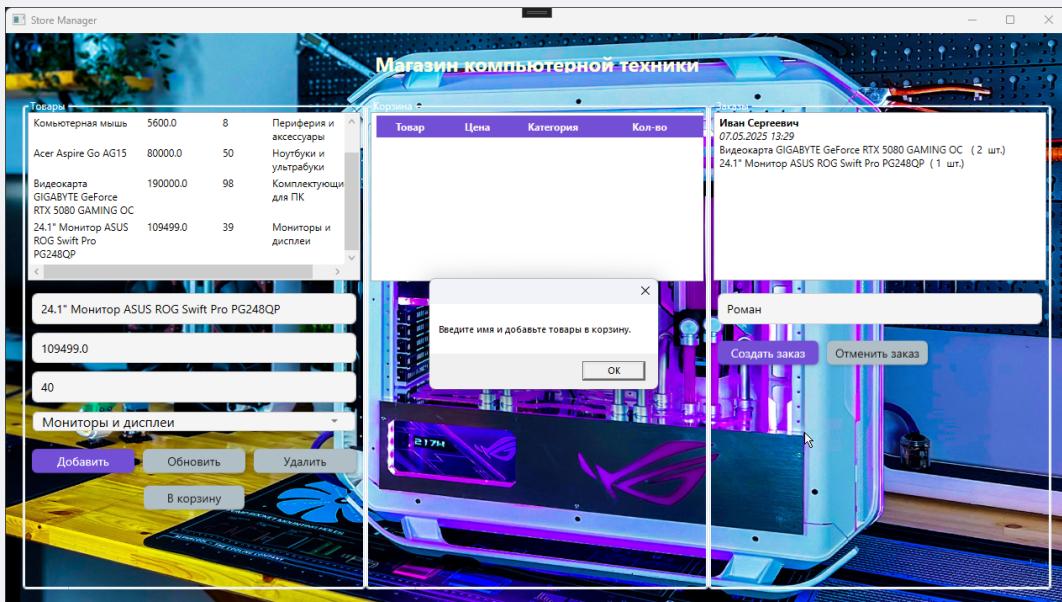


Рисунок 6.5 – Ошибка при создании заказа без товаров.

Пользователь может отменить заказ нажав на кнопку «Отменить заказ», после этого заказ удаляется из списка заказов.

## 7. Контрольные вопросы ?

### 1. Что такое MVVM-паттерн и какова его структура в рамках WPF-приложения StoreManager?

MVVM (Model-View-ViewModel) — это архитектурный шаблон, применяемый в WPF для разделения представления (XAML), логики отображения (ViewModel) и бизнес-логики (Model). В проекте StoreManager он представлен View (XAML), ViewModel-классами (MainViewModel, ProductViewModel, OrderViewModel) и моделями (Product, Order, OrderItem, CartItem).

### 2. Как реализована связь между View и ViewModel в WPF-приложении? Приведите пример привязки.

Связь реализована через механизм data binding. Пример: `Text='Binding ProductName'` привязывает значение поля к свойству во ViewModel. Контекст задаётся через `Window.DataContext`.

### 3. Какие классы представляют модель предметной области (Model) в данном проекте и за что они отвечают?

`Product` — товар, `Order` — заказ, `OrderItem` — позиция в заказе, `CartItem` — товар в корзине. Эти классы хранят бизнес-данные и реализуют интерфейс `INotifyPropertyChanged`.

### 4. Какие задачи решает класс `RelayCommand` и как обеспечивается управление доступностью команд?

Класс `RelayCommand` реализует интерфейс `ICommand` и позволяет связать действия UI с методами ViewModel. Доступность определяется функцией `CanExecute`, обновление — через событие `CanExecuteChanged`.

### 5. Как работает механизм уведомления `INotifyPropertyChanged` в контексте классов `Product`, `CartItem` и `ViewModel`?

При изменении свойства вызывается `OnPropertyChanged`, которое инициирует событие `PropertyChanged`. Это позволяет автоматически обновлять связанные элементы UI.

### 6. Как осуществляется доступ к базе данных с помощью Entity Framework Core в `StoreDbContext`?

Класс `StoreDbContext` наследуется от `DbContext` и предоставляет доступ к таблицам через `DbSet<>`. Настройки соединения и поведения моделей задаются в `OnConfiguring` и `OnModelCreating`.

### 7. В чём различие между `ProductRepository` и `OrderRepository` по архитектуре и взаимодействию с БД?

ProductRepository работает с одной таблицей Product и выполняет простые SQL-запросы. OrderRepository работает с двумя таблицами (Order, OrderItem) и управляет транзакциями и связями между сущностями.

**8. Опишите этапы формирования нового заказа. Какие сущности при этом задействуются?**

Пользователь добавляет товары в корзину (CartItem), затем оформляет заказ (Order), в который входят позиции (OrderItem) и клиентские данные. Все объекты сохраняются в базу через StoreDbContext или OrderRepository.

**9. Какие команды реализованы в MainViewModel, и как они используются в интерфейсе пользователя?**

Команды: AddProductCommand, UpdateProductCommand, DeleteProductCommand, AddOrderCommand, DeleteOrderCommand, AddToCartCommand, RemoveFromCartCommand и др. Каждая команда связана с кнопкой в UI и вызывает соответствующий метод ViewModel.

**10. Как реализуется удаление заказа и восстановление остатка товара на складе при этом действии?**

При удалении заказа вызывается метод DeleteOrder. Он возвращает количество товаров из заказа обратно в остатки соответствующих Product, после чего заказ удаляется из базы данных.