**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информатика»

Специальность 1-40 04 01 «Информатика и технологии программирования»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Модели данных и СУБД»

на тему: **«Моделирование услуг автопарка, предоставляющего аренду автомобилей населению»**

Исполнитель: студент гр. ИП-31 Лютиков М. С.

Руководитель: ст. преподаватель

Шибеко В. Н.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

По защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2023

**СОДЕРЖАНИЕ**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Введение5 | | | 1 Анализ предметной области6 | | |  | 1.1 Обзор объекта автоматизации и существующих аналогов6  1.2 Обзор используемых технологий7  1.3 Требования к проектируемому программному обеспечению8  1.4 Обзор существующих методов решения9 | | 2 Разработка архитектуры и моделирование предметной области11 | | |  | 2.1 Описание предметной области11  2.2 Структура базы данных и СУБД13 | |  | 2.3 Модели классов доменов17  2.4 Защита информации18 | | 3 Структура программного обеспечения21 | | |  | 3.1 Анализ программного кода21 | |  | 3.2 Описание архитектуры приложения23 | | 4 Тестирование приложения44 | | |  | 4.1 Общее описание тестирования проекта44 | |  | 4.2 Функции и их реализация в формах и алгоритмах48 | | Заключение50 | | | Список использованных источников51 | | | Приложение А – Листинг программного кода52 | | | |
|  | |
|  | |
|  |  |
|  | |
|  |  |
|  |  |
|  | |
|  |  |
|  |  |
|  | |
|  | |
|  | |
|  | |

**ВВЕДЕНИЕ**

Современная социально-экономическая ситуация в Республике Беларусь, как и во всем мире, характеризуется стремительным развитием технологий и услуг, включая сферу автотранспортных услуг. Актуальной проблемой, которая воздействует на уровень жизни населения и экономику страны в целом, становится необходимость эффективного использования транспортных средств, особенно в условиях ограниченных ресурсов и повышенного спроса на транспортные услуги.

В последнее время наблюдается рост популярности услуг аренды автомобилей, что связано с изменением транспортных предпочтений населения. В свою очередь, возникает потребность в развитии эффективных моделей организации и управления автопарками, предоставляющими такие услуги.

Аренда автомобилей становится все более востребованной как на короткий, так и на длительный срок. Это требует внедрения новых технологий и методов управления, которые могут повысить эффективность работы автопарка, улучшить качество обслуживания клиентов и повысить уровень удовлетворенности потребителей.

В 2020-е годы стала заметна тенденция к использованию информационных технологий в управлении автопарками. При этом, одним из основных направлений становится моделирование услуг автопарка, которое позволяет оптимизировать процессы управления и планирования, учесть предпочтения и потребности клиентов, а также повысить экономическую эффективность предприятия.

В соответствии с концепцией развития автопарка, важнейшим звеном становится создание системы моделирования услуг, которая обеспечивает анализ и прогнозирование различных сценариев работы автопарка. Она должна включать в себя комплекс подходов и методов, позволяющих управлять ресурсами автопарка и предоставлять услуги аренды автомобилей наиболее эффективным образом.

Создание системы моделирования услуг автопарка способно помочь решить множество проблем, с которыми сталкиваются такие предприятия в своей работе. Это включает, но не ограничивается, оптимизацией использования автомобилей, прогнозированием спроса на услуги, управлением рисками, связанными с простоем транспортных средств, и улучшением качества услуг.

В целом, разработка и внедрение системы моделирования услуг автопарка является важным шагом в повышении эффективности обслуживания населения и обеспечения его мобильности. Она также может способствовать снижению экологического воздействия транспорта, оптимизируя использование автомобилей и снижая их простой.

**1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ**

* 1. **Обзор объекта автоматизации и существующих аналогов**

В современном быстро развивающемся мире потребители и предприятия все больше полагаются на услуги аренды автомобилей для удовлетворения своих транспортных потребностей. Это особенно актуально в условиях глобализации и мобильности, когда необходимость в перемещениях на автомобиле возникает все чаще, а стоимость владения автомобилем может быть высокой. Предприятия, предоставляющие услуги по аренде автомобилей, сегодня находятся во многих местах: в аэропортах, городах, туристических курортах, в деловых и торговых центрах, и даже в медицинских и образовательных учреждениях. Поэтому, безусловно, требуется автоматизированная система для управления этим сложным процессом.

Система моделирования услуг автопарка, которая будет представлена здесь, нацелена на автоматизацию процесса управления автопарком и организации аренды автомобилей. Эта система обеспечивает эффективное управление ресурсами автопарка, упрощает процесс бронирования и аренды автомобилей, облегчает взаимодействие между клиентами и заказами, и в целом улучшает качество услуг.

Можно сравнить эту систему с веб-сайтами и мобильными приложениями, которые также предлагают ряд услуг, однако система моделирования предлагает ряд ключевых преимуществ. Одно из главных - более быстрая и надежная обработка данных благодаря минимальной нагрузке на источники хранения данных. Это особенно важно, учитывая объем информации, которым необходимо управлять в рамках автопарка, включая данные об автомобилях, их состоянии, информацию о клиентах и детали аренды.

Многие государственные и частные автопарки, работающие с подобными большими и разнообразными наборами данных и имеющие различные требования к уровню доступа к этим данным, нуждаются в удобной и эффектив- ной системе для просмотра, добавления, редактирования, удаления, поиска и сохранения записей. Так, моделирующая система, разработанная в рамках этого курсового проекта, будет служить таким инструментом, позволяя автопаркам гибко управлять своими ресурсами и обеспечивать высокий уровень обслуживания клиентов.

Важной частью разработки этой системы является создание простого, но функционального и удовлетворяющего всем требованиям интерфейса. Это поможет обеспечить эффективную работу сотрудников автопарка и улучшить обслуживание клиентов. Система будет обладать интуитивно понятными и удобными средствами для работы с данными, которые максимально облегчат процесс работы с ними.

Среди обязательных информационно-справочных услуг, которые будут предоставляться клиентам, пользователям и сотрудникам автопарка с помощью моделирующей системы, можно выделить следующие:

– информация о доступных для аренды автомобилях, их характеристиках и стоимости аренды, а так же местоположении после заказа;

– функция бронирования автомобилей на определенный период;

– информация о состоянии бронирования и истории бронирований пользователя.

Таким образом, разработка моделирующей системы для услуг автопарка, предоставляющего аренду автомобилей населению, представляет собой актуальную и важную задачу, решение которой позволит повысить эффективность работы автопарков и уровень удовлетворенности клиентов.

* 1. **Обзор используемых технологий**

Для реализации моделирующей системы автопарка в форме *Windows*-приложения в этом проекте будет использоваться язык программирования *C#*. Это высокоуровневый компилируемый язык общего назначения со статической типизацией, который применим для создания широкого спектра приложений. *C#* в настоящее время считается одним из самых популярных и широко используемых языков программирования.

*C#* обладает богатыми возможностями для работы с памятью, унаследованными от Си и *C++*. В связи с этим *C#* часто применяется в объектно-ориентированном программировании, включая создание операционных систем, драйверов, разнообразных утилит, антивирусов, веб-сервисов и т.д. Кроме того, он широко используется для создания графических приложений, различного рода прикладных программ и даже игр с насыщенной визуализацией.

В рамках данного проекта выбрана технология *Windows Forms* в среде разработки *Microsoft Visual Studio 2022*, основанная на платформе *C# .NET*. Эта платформа предоставляет полный набор функционала для создания приложений, работающих на операционной системе *Windows*.

*C#* является компилируемым языком, а это означает, что компилятор переводит исходный код на *C#* в исполняемый файл, содержащий набор машинных инструкций. Однако, поскольку разные платформы имеют свои особенности, скомпилированные программы нельзя просто перенести с одной платформы на другую без дополнительных действий. Несмотря на это, исходный код на *C#* в большинстве случаев обладает высокой степенью переносимости, если не используются функции, специфичные для определенной операционной системы. К тому же, наличие компиляторов, библиотек и инструментов разработки для большинства распространенных платформ позволяет компилировать один и тот же исходный код на *C#* в приложения под эти платформы.

* 1. **Требования к проектируемому программному обеспечению**

После анализа предметной области, к проектируемому приложению можно выдвинуть несколько требований.

Любой пользователь, установивший приложение на свое устройство, должен иметь возможность ознакомиться с основной информацией службы (информацией о заказах, профиле, ограничениях и т. д.). Пользователям должна быть предоставлена возможность пройти стандартный процесс авторизации (если пользователь уже является клиентом службы) или регистрации (если пользователь хочет стать клиентом службы).

Клиент службы должен обладать как минимум тем же функционалом, что и новый пользователь. Кроме того, клиент может отслеживать состояние своего заказа и просматривать историю своих заказов через личный кабинет.

Служба должна обладать возможностью добавления, редактирования, изменения и удаления различной информации о предоставляемых услугах и о клиентах. Для организации работы с клиентами должна быть предусмотрена автоматическая система, обладающая необходимым функционалом для обработки заказов и безошибочного взаимодействия с клиентами. Для обеспечения корректности работы, должна быть предусмотрена роль администратора, которая включает в себя возможность корректировки данных о персонале и обеспечения безопасности информации.

В контексте программного обеспечения для сервиса каршеринга, интерфейс играет ключевую роль. Для большинства пользователей, не знакомых с деталями программирования, интерфейс - это весь софт. Независимо от того, сколько времени и сил было вложено в разработку бэкенда и функциональности, удобство использования приложения во многом определяется его интерфейсом.

Удобство взаимодействия с приложением каршеринга важно для обеспечения комфортного пользовательского опыта. В процессе создания данного приложения учтены следующие рекомендации экспертов в области *UI/UX*:

– интерфейс должен быть интуитивно понятным, чтобы пользователю не требовалось дополнительных объяснений о том, как пользоваться приложением;

– не должно быть лишней информации, которая может отвлекать пользователя от выполнения задач. Не нужны лишние баннеры, разделы и категории;

– яркость и контрастность элементов интерфейса должны быть выбраны таким образом, чтобы обеспечить комфортное восприятие информации;

– размер символов и элементов управления должен быть оптимизирован для обеспечения быстрого и точного восприятия информации.

Приложение должно иметь возможность взаимодействовать с базой данных и предоставлять к ней доступ в удобной и подходящей форме. Особое внимание будет уделено интеграции с различными типами баз данных, включая *CSV*, *MongoDB* и *SQL*, чтобы обеспечить гибкость и масштабируемость в обработке данных. Кроме того, должны быть предусмотрены различные уровни доступа к данным и возможность взаимодействия с ними для различных ролей пользователей: неподтверждённого клиента, клиента, сотрудника и администратора.

В итоге, с учетом всех вышеуказанных требований, приложение каршеринга должно представлять собой интуитивно понятный и удобный в использовании интерфейс, что снизит порог входа для пользователей всех возрастных групп. Это, в свою очередь, уменьшит нагрузку на поддержку со стороны сервиса каршеринга, обеспечит эффективное выполнение всех задач и позволит быстро обновлять информацию.

**1.4 Обзор существующих методов решения**

Сегодня в области предоставления услуг по аренде автомобилей (каршеринг) функционирует большое количество современных платформ, оснащенных передовыми технологиями, включая веб-сайты, мобильные приложения и информационные системы. Таким образом, на рынке существует множество возможных подходов к созданию подходящего программно-аппаратного комплекса для каршеринга. Стоит отметить, что эти решения обладают более широким функционалом и возможностями по сравнению с приложением, которое планируется разработать в рамках данного проекта.

Одним из примеров уже существующих решений в данной области может служить приложение «*Zipcar*», разработанное для автоматизации процесса аренды автомобилей. Этот веб-сайт и мобильное приложение обладают понятным интерфейсом, что позволяет клиентам быстро и без лишних сложностей осуществить бронирование автомобиля. Приложение обеспечивает высокий уровень безопасности благодаря встроенным алгоритмам шифрования данных пользователей и защищенным каналам связи с базой данных сервиса. Благодаря встроенному программному обеспечению, любой клиент может ознакомиться со списком доступных автомобилей и совершить необходимую операцию.

Приложение для каршеринга не требует специальных навыков программирования для его эксплуатации. Владельцы автопарка могут самостоятельно наполнить его необходимыми данными: описаниями автомобилей, фотографиями, и прочей информацией. Приложение обладает следующими функциями:

– информирование зарегистрированных пользователей об изменениях;

– просмотр доступных для аренды автомобилей;

– возможность прохождения процесса бронирования автомобиля;

– предоставление информации о техническом состоянии автомобилей;

– предоставление контактной информации службы поддержки;

– доступ к информации о партнёрах и специальных предложениях;

– процесс оплаты аренды;

– раздел с ответами на часто задаваемые вопросы;

– возможность связаться со службой поддержки;

– возможность получения электронного чека после совершения оплаты.

Другими примерами служб с подобным функционалом являются сайты таких каршеринговых компаний как «*Car2Go*», «*DriveNow*» и «*Getaround*», которые используют продвинутые технологии, такие как: сканирование пластиковых карт, системы *GPS*-слежения, датчики движения, аудиосистемы, сканеры штрих-кодов, системы распознавания лиц и т. д.

Основываясь на анализе существующих программных решений, можно сделать вывод, что реализация всего существующего функционала может быть затруднительной в рамках данного проекта (согласно [3]) из-за ограниченности программных и технических ресурсов, а также необходимости большого количества времени на разработку. Однако основные задачи, которые должно решать приложение, включают в себя: наличие понятного пользовательского интерфейса, базы данных с информацией об автомобилях, возможность её редактирования и просмотра, а также различный уровень доступа для разных пользователей.

**2 РАЗРАБОТКА АРХИТЕКТУРЫ И МОДЕЛИРОВАНИЕ**

**ПРЕДМЕТНОЙ ОБЛАСТИ**

**2.1 Описание предметной области**

Модель предметной области в контексте каршеринга должна иллюстрировать сущности, включая автомобили, пользователей, бронирования, а также их взаимоотношения между собой. Сущности описывают объекты, которые являются предметом деятельности каршеринга, и субъекты, осуществляющие деятельность в рамках этой предметной области. Свойства объектов и субъектов реального мира описываются с помощью атрибутов.

Взаимоотношения между сущностями иллюстрируются с помощью связей. Правила и ограничения взаимоотношений описываются с помощью свойств связей. Обычно связи определяют либо зависимости между сущностями, либо влияние одной сущности на другую, согласно [4].

Для формирования представления о предметной области, а также для моделирования функциональных требований системы каршеринга и ее взаимодействия с внешними актерами используют *UML* диаграммы. Среди *UML* различают также диаграммы прецедентов, которые описывают функциональность системы в терминах ее использования. Они состоят из прецедентов (*use cases*) - действий, которые пользователи (актеры) могут выполнять в системе, и связей между ними.

Для каждого прецедента на диаграмме прецедентов указывается его название, описание и список актеров, которые могут выполнять этот прецедент. Также на диаграмме могут быть показаны связи между прецедентами, например, какой прецедент вызывает другой. Примеры таких *UML*-диаграмм в контексте каршеринга могут включать процессы бронирования автомобиля, оплаты за аренду, отмены бронирования и т.д.



Рисунок 2.1 – Диаграмма прецедентов и актёров (Клиент)

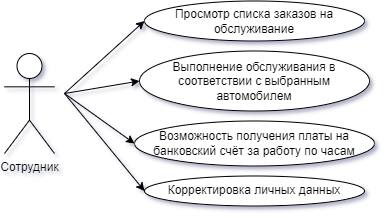


Рисунок 2.2 – Диаграмма прецедентов и актёров (Сотрудник)

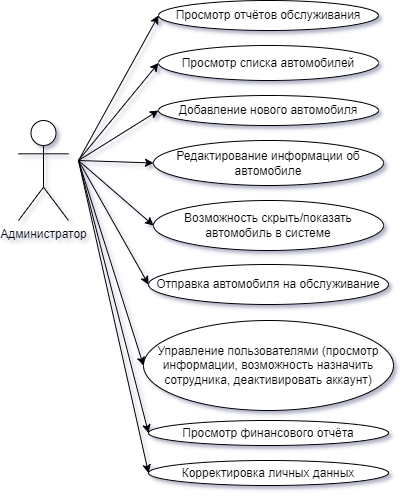


Рисунок 2.3 – Диаграмма прецедентов и актёров (Администратор)

В соответствии с моделью предметной области и *UML* диаграммами можно сделать анализ задачи. Необходимо реализовать три роли: клиент, сотрудник и администратор, которые будут взаимодействовать через услуги, оказываемые автопарком.

Клиент будет связан с услугами аренды автомобилей, сотрудник отвечает за обслуживание автомобилей, а администратор отвечает за управление данными автопарка, включая добавление, редактирование и удаление информации об автомобилях, а также, при необходимости, отключение системы.

В соответствии с анализом задачи, можно описать прецеденты и актеров:

– актер «Клиент» – это актер с возможностью забронировать автомобиль для аренды. Он имеет возможность ознакомиться с услугами автопарка;

– актер «Сотрудник» – это актер, обладающий возможностями добавления информации об обслуживании автомобилей;

– актер «Администратор» – это актер, который имеет возможность просматривать, редактировать, удалять, добавлять и обновлять всю информацию, необходимую для работы автопарка.

Описание основных прецедентов:

– прецедент «Просмотр информации об автомобилях» – прецедент для просмотра списка доступных автомобилей, их характеристик и стоимости аренды;

– прецедент «Авторизация и регистрация» – прецедент для создания аккаунта и доступа к полной версии сервиса;

– прецедент «Редактирование, удаление и добавление данных» – прецедент для просмотра и, в случае необходимости, выполнения различных манипуляций с данными, в результате которых они будут изменены.

Структура базы данных и *СУБД* в контексте моделирования услуг автопарка, предоставляющего аренду автомобилей населению (каршеринг), отличается простотой и эффективностью благодаря использованию *JSON*-формата для хранения данных и *LINQ* для их обработки.

*JSON*, что означает *JavaScript Object Notation*, это легковесный формат обмена данными, который основан на подмножестве *JavaScript*. Он обладает открытым стандартом и широко используется для представления структурированных данных в виде человеко-читаемого текста. *JSON* позволяет представлять данные в виде пар ключ-значение, и поддерживает базовые типы данных, такие как строки, числа, логические значения, массивы и вложенные объекты.

Дополнительное преимущество использования *JSON* заключается в высокой скорости чтения и записи данных. Благодаря простому и лаконичному формату, *JSON* обеспечивает быстрый парсинг и генерацию данных, что критически важно для высокопроизводительных приложений, таких как система каршеринга.

**2.2 Структура базы данных и СУБД**

База данных приложения представлена в виде списка объектов различных классов, каждый из которых отвечает за определенный аспект функционирования системы, согласно [2]. Ниже представлены основные компоненты базы данных:

– *User* - простой список объектов, содержащий информацию о пользователях системы;

– *Car* - более сложный список объектов, отражающий информацию об автомобилях, доступных для аренды;

– *Order* - сложный список объектов, в котором хранится информация обо всех заказах;

– *Payment* - сложный список объектов, содержащий данные об оплатах, произведенных пользователями;

– *BankTransaction* - простой список объектов, отслеживающий банковские транзакции, связанные с операциями аренды;

– *ServiceReport* - сложный список объектов, в котором сохраняются отчеты о техническом обслуживании автомобилей.

В рамках перехода к использованию более современных и гибких методов управления данными, структура базы данных и СУБД в проекте была переработана для интеграции с различными типами баз данных: *CSV*, *MongoDB* и *SQL*:

– *CSV* (*Comma-Separated Values*) – этот формат особенно полезен для хранения и обработки больших объемов данных с фиксированными структурами. В *CSV*-файлах данные хранятся в виде простых текстовых файлов, где каждая строка представляет собой отдельную запись, а значения разделены запятыми. Это обеспечивает легкость чтения и записи данных, а также удобство при переносе данных между различными приложениями;

– *MongoDB* - это документо-ориентированная *NoSQL* база данных, которая хранит данные в формате *BSON* (бинарный *JSON*). *MongoDB* предоставляет гибкость в управлении данными, позволяя хранить сложные иерархические структуры в одном документе. Это особенно полезно для систем, где модели данных могут часто меняться, так как *MongoDB* не требует строгой схемы данных;

– *SQL*-базы данных - это традиционные реляционные базы данных, использующие язык структурированных запросов (*SQL*) для управления данными. *SQL*-базы данных идеально подходят для систем с четко определенной структурой данных, где важна целостность и безопасность данных. Они позволяют проводить сложные запросы, объединения и транзакции, что делает их подходящими для сложных приложений с большим количеством пользователей.

Интеграция этих различных типов баз данных в проект обеспечивает ряд преимуществ:

– Гибкость - возможность выбора наиболее подходящего типа хранения данных в зависимости от конкретных требований функционала приложения;

– Масштабируемость - легкость масштабирования системы путем добавления или изменения баз данных в соответствии с изменяющимися потребностями приложения;

– Эффективность - улучшение производительности за счет распределения нагрузки между различными системами управления базами данных.

Пример *CSV*, как файловой базы данных, на рисунке 2.4:

Id,Name,Email,Phone,DrivingLicense

1,Иван Иванов,ivanov@example.com,+375291234567,AB1234567

2,Мария Петрова,petrova@example.com,+375292345678,CD2345678

3,Алексей Сидоров,sidorov@example.com,+375293456789,EF3456789

Рисунок 2.4 – Пример *CSV*

Для работы с этими разнообразными базами данных активно используется технология *LINQ*, что позволяет эффективно осуществлять поиск, сортировку, группировку и другие операции над данными, а также упрощает код и делает его более читаемым. Это обеспечивает универсальность и адаптивность приложения в различных сценариях использования данных.

Таким образом, комбинированное использование *CSV*, *MongoDB* и *SQL* баз данных в проекте позволяет достичь оптимального баланса между гибкостью, масштабируемостью и эффективностью управления данными, обеспечивая надежную и высокопроизводительную платформу для разработки и поддержки сложных приложений.

В ходе работы с различными типами базы данных и связи данных с выбранным языком программирования была разработана структура данных, соответствующая требованиям курсового проекта и активно применяемая в приложении службы социальной помощи. На рисунке 2.5 представлена схема данных, использующиеся в ней отношения, основные атрибуты и связи.

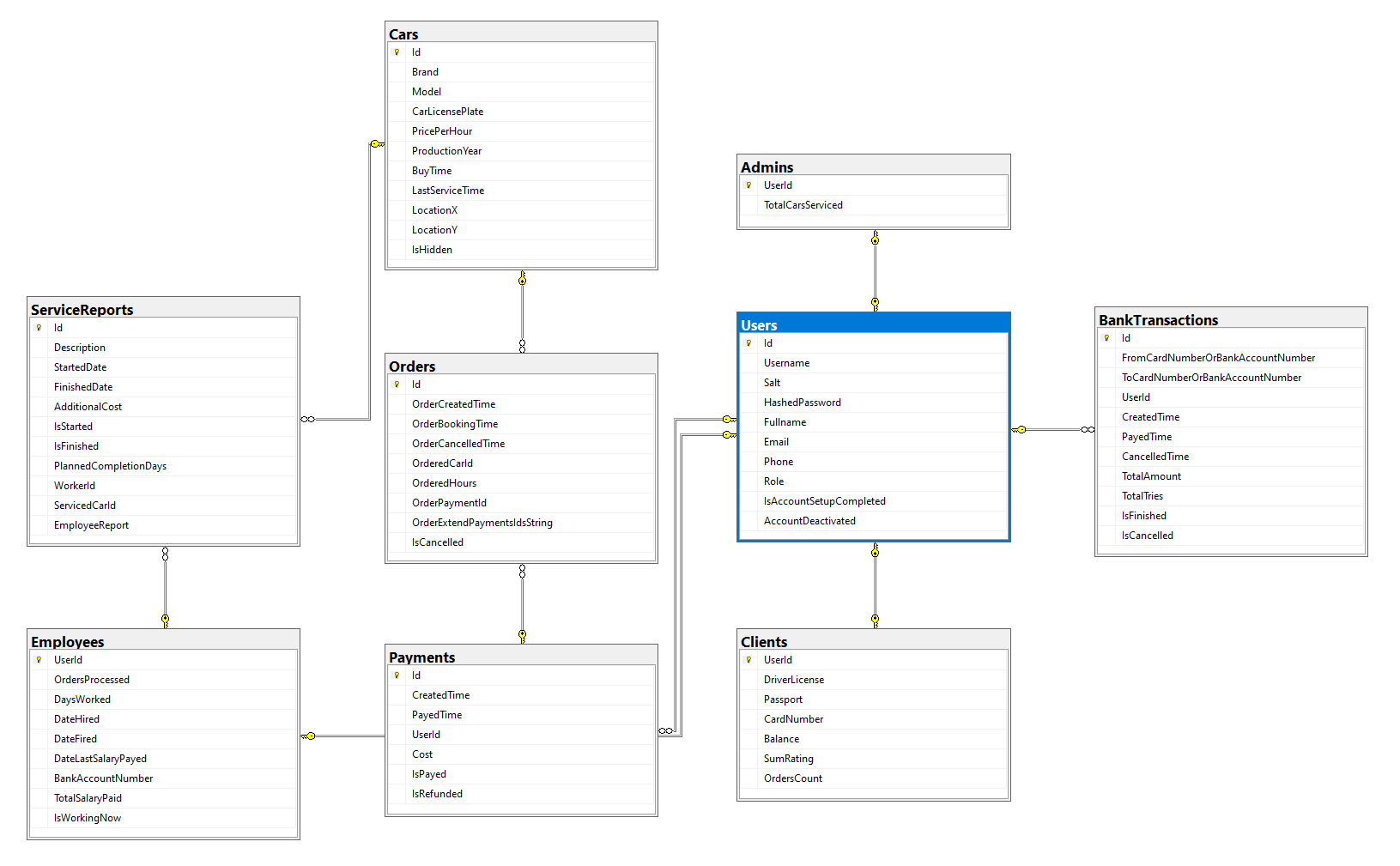


Рисунок 2.5 – Схема базы данных

Класс «*User*» (пользователь) является базовым классом для классов «*Client*», «*Employee*» и «*Admin*» в этом проекте. Он содержит общие атрибуты, такие как уникальный идентификатор (*Id*), имя пользователя (*Username*), хэш и соль пароля (*HashedPassword* и *Salt*), ФИО (*Fullname*), адрес электронной почты (*Email*), номер телефона (*Phone*) и роль (*Role*). Класс также включает атрибуты, указывающие на завершение настройки учетной записи (*IsAccountSetupCompleted*) и деактивацию учетной записи (*AccountDeactivated*).

Класс «*Client*» (клиент), наследующий от класса «*User*» в этом проекте, содержит дополнительные атрибуты, такие как номер водительского удостоверения (*DriverLicense*), паспортные данные (*Passport*), номер банковской карты (*CardNumber*), баланс (*Balance*), общий рейтинг (*SumRating*) и количество заказов (*OrdersCount*).

В этом проекте класс «*Employee*» (сотрудник) также наследует от класса «*User*» и содержит дополнительные атрибуты, такие как количество обработанных заказов (*OrdersProcessed*), количество отработанных дней (*DaysWorked*), даты принятия на работу, увольнения и последней выплаты зарплаты (*DateHired*, *DateFired*, *DateLastSalaryPayed*), номер банковского счета (*BankAccountNumber*), общая выплаченная зарплата (*TotalSalaryPaid*) и статус работы (*IsWorkingNow*).

Класс «*Admin*» (администратор), наследующий от класса «*User*» в этом проекте, содержит атрибут, указывающий на общее количество обслуживаемых автомобилей (*TotalCarsServiced*).

Класс «*Car*» (автомобиль) в этом проекте содержит атрибуты, такие как уникальный идентификатор (*Id*), марка (*Brand*), модель (*Model*), государственный номер (*CarLicensePlate*), стоимость аренды в час (*PricePerHour*), год выпуска (*ProductionYear*), время покупки (*BuyTime*), последняя дата обслуживания (*LastServiceTime*), координаты местоположения (*LocationX* и *LocationY*) и статус скрытия автомобиля (*IsHidden*).

Класс «*Order*» (заказ) содержит информацию о времени создания заказа (*OrderCreatedTime*), времени бронирования (*OrderBookingTime*), времени отмены (*OrderCancelledTime*), идентификаторе заказанного автомобиля (*OrderedCarId*), количестве часов аренды (*OrderedHours*), идентификаторе платежа (*OrderPaymentId*), строке с идентификаторами дополнительных платежей (*OrderExtendPaymentsIdsString*) и статусе отмены заказа (*IsCancelled*).

Класс «*Payment*» (оплата) содержит атрибуты, такие как уникальный идентификатор (*Id*), время создания и оплаты (*CreatedTime*, *PayedTime*), клиент (*UserId*), стоимость (*Cost*), и статусы оплаты и возврата средств (*IsPayed*, *IsRefunded*).

Класс «*BankTransaction*» (банковская транзакция) в проекте содержит атрибуты, такие как уникальный идентификатор (Id), номер счета отправителя и номер счёта получателя (*FromCardNumberOrBankAccountNumber*, *ToCardNumberOrBankAccountNumber*), пользователь (*UserId*), время создания, оплаты и отмены транзакции (*CreatedTime*, *PayedTime*, *CancelledTime*), общая сумма (*TotalAmount*), общее количество попыток (*TotalTries*), и статусы завершения и отмены транзакции (*IsFinished*, *IsCancelled*).

Класс «*ServiceReport*» (отчет об обслуживании) в проекте содержит атрибуты, такие как уникальный идентификатор (*Id*), описание (*Description*), даты начала и окончания обслуживания (*StartedDate*, *FinishedDate*), дополнительная стоимость (*AdditionalCost*), статусы начала и завершения обслуживания (*IsStarted*, *IsFinished*), планируемое количество дней до завершения (*PlannedCompletionDays*), сотрудник, выполняющий обслуживание (*WorkerId*), обслуживаемый автомобиль (*ServicedCarId*), и отчет сотрудника (*EmployeeReport*).

В завершение, «*RolesContainer*» (контейнер ролей) в проекте является перечислением для определения ролей пользователя: клиент (*Client* = 1), сотрудник (*Employee* = 2), администратор (*Admin* = 3).

**2.3 Модели классов доменов**

После определения прецедентов и сценариев, следующим шагом является анализ основных сущностей для работы с базой данных в контексте моделирования услуг автопарка, предоставляющего аренду автомобилей населению (каршеринг).

Следовательно, мы можем выделить следующие доменные классы, соответствующие сущностям базы данных:

– Класс *User* – предназначен для хранения информации о пользователе системы. Он включает в себя все необходимые поля, такие как идентификатор пользователя, имя, контактные данные и т.д., а также конструктор для создания объекта данного класса;

– Класс *Car* – содержит информацию об автомобилях, доступных для аренды, включая идентификатор автомобиля, марку, модель, год выпуска и другие специфические характеристики;

– Класс *Order* – отражает информацию о заказах, включая идентификатор заказа, идентификатор пользователя, идентификатор автомобиля, дату и время начала и окончания аренды и так далее;

– Класс *Payment* – хранит информацию о платежах пользователей, включая идентификатор платежа, идентификатор пользователя, сумму платежа и дату платежа;

– Класс *BankTransaction* – содержит информацию о банковских транзакциях, связанных с платежами пользователей;

– Класс *ServiceReport* – предназначен для хранения отчетов о техническом обслуживании автомобилей;

– Класс *Admin*, *Employee* и *Client* представляют различные роли в системе и содержат специфическую для роли информацию и функциональность. Наследуются от класса *User*;

– Перечисление (*enum*) *RolesContainer* служит для управления и хранения информации о ролях в системе.

Для управления этими моделями и выполнения операций над данными используются контроллеры, представленные классами *AesGсm256*, *CryptographyControl*, *DatabaseContext* и *UtilsControl*.

Вместо компонента *DataGridView*, используемого в *Windows Forms* для отображения и редактирования табличных данных, в данной системе используется *ListView*. *ListView* предлагает больше гибкости при отображении данных и позволяет представить данные в виде списка, таблицы или даже в виде маленьких и больших иконок.

Для управления этими моделями и выполнения операций над данными используется паттерн стратегии, позволяющий выбирать между различными базами данных - *CSV*, *MongoDB* и *SQL*, в зависимости от требований системы.

Для взаимодействия с базой данных используется *LINQ*. Существует класс «*DatabaseContext*», который содержит методы для выполнения различных операций с базами данных. Эти операции включают, но не ограничиваются, добавлением, обновлением, удалением и извлечением и сохранением данных из базы данных.

Важно отметить, что в нашей системе аренды автомобилей каждый из этих классов играет важную роль в обеспечении функциональности системы. Они вместе обеспечивают надежное и эффективное управление данными, что обеспечивает плавность и эффективность работы всей системы.

В заключение, структура классов домена и контроллеров, вместе с использованием *ListView* для визуализации данных и паттерном стратегии для работы с различными базами данных *CSV*, *MongoDB* и *SQL*, обеспечивает гибкость, масштабируемость и производительность, которые являются критически важными для успешной реализации системы каршеринга.

**2.4 Защита информации**

Защита информации представляет собой критически важную область деятельности любого предприятия, особенно в отраслях, где требуется обработка и хранение чувствительной информации, такой как персональные данные и финансовые операции. Важные принципы, которые следует соблюдать, включают обеспечение конфиденциальности, целостности и доступности данных.

В эпоху цифровизации, когда обработка, передача и хранение информации происходят в виртуальном пространстве, появляется ряд новых угроз. Для борьбы с ними в данной системе используются современные методы шифрования и защиты данных.

В конкретной системе каршеринга для шифрования файлов базы данных *CSV* используется алгоритм *AesGcm256*. Это современный алгоритм шифрования, который обеспечивает высокий уровень безопасности данных. Он преобразует исходные данные с использованием секретного ключа, таким образом, что их можно восстановить только при использовании этого же ключа. Ключ хранится в закодированном алгоритмом *Base64* виде в файле «*enc.key*».

Для повышения безопасности учетных данных пользователей применяется процесс хеширования с использованием соли, реализованный в классе *CryptographyControl*. Хеширование с использованием соли - это метод, при котором к паролю пользователя добавляется случайная последовательность символов, известная как «соль», перед процессом хеширования. Это делает хеш-функцию более устойчивой к атакам по словарю и атакам поиском с применением «радужных таблиц», так как даже небольшие изменения в исходных данных приводят к значительному изменению результата хеширования (так называемый «лавинный эффект» хеш-функции).

В дополнение к этим мерам, для защиты исполняемых файлов приложения от обратного инжиниринга используется инструмент обфускации *ConfuserEx*. Обфускация кода – это процесс преобразования кода приложения в форму, которая сохраняет его функциональность, но делает его практически невозможным для понимания и анализа. Это дополнительная защита от попыток взлома и несанкционированного использования программного обеспечения. *ConfuserEx* обеспечивает высокий уровень обфускации *.NET* приложений, затрудняя процесс декомпиляции и понимания и исходного кода. Этот инструмент предлагает множество функций обфускации, включая переименование классов, методов и переменных, управление потоком и сокрытие строк. Благодаря этому код приложения становится более сложным для анализа и взлома, что повышает общий уровень защиты программного обеспечения.

Совокупность этих методов обеспечивает комплексную защиту информационной системы. Шифрование с помощью *AesGcm256* обеспечивает безопасное хранение и передачу данных (согласно [5]), хеширование паролей с использованием соли увеличивает уровень защиты от атак на учетные данные пользователей, а применение обфускации с использованием *ConfuserEx* защищает исходный код приложения от несанкционированного использования и взлома. Все эти меры помогают создать надежную и безопасную информационную среду.

Что касается примеров хеширования и шифрования, предположим, что у нас есть следующая соль и пароль:

Соль (в виде *Base64* строки): *MTIzNDU2Nzg5MGFiY2RlZg==*

Пароль: *MyS3cur3P@ssw0rd*

При применении указанной выше функции *HashPasswordWithSalt*, соль и пароль объединяются и проходят через алгоритм хеширования. После этого пароль может выглядеть примерно так:

Хешированный пароль: *h1F1cG2bI2p4v5E+6k3Lr1E+0bR...*

Обратим внимание, что в реальных условиях хешированный пароль будет значительно длиннее и сложнее.

Теперь рассмотрим шифрование строки с помощью *AesGcm256*. Допустим, у нас есть следующая исходная строка:

Исходная строка: *Sensitive Information*

При использовании функции *encrypt* класса *AesGcm256* с определенным ключом и вектором инициализации (*IV*), исходная строка может быть преобразована в следующий зашифрованный текст:

Зашифрованный текст:

*QklFMUJDMTItN0U3MC00NjZBLUIwQzEtMjM5NTZGQjY5NTgx*

Безусловно, это всего лишь примеры того, как можно использовать технологии хеширования и шифрования. В реальном мире результаты и объемы хеширования и шифрования могут варьироваться в широких пределах. Это происходит потому, что они напрямую зависят от ряда ключевых факторов, таких как выбранные алгоритмы, ключи шифрования и другие специфические параметры. Иногда разница в результатах может быть значительной и может варьироваться в зависимости от контекста и конкретных условий применения. Также следует учесть, что эффективность и безопасность хеширования и шифрования во многом зависят от правильного выбора и применения соответствующих методов. Поэтому всегда важно принимать во внимание эти факторы при реализации систем безопасности данных.

**3 СТРУКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

**3.1 Анализ программного кода**

Основной целью данного проекта является разработка интуитивно понятного приложения для аренды автомобилей, которое поможет упростить взаимодействие между компанией и клиентами, а также повысит эффективность и качество предоставляемых услуг. Приложение должно быть разработано с учетом потребностей и особенностей пользователей, включая сотрудников компании и клиентов, и предоставлять широкий набор функций для обеспечения удобства и эффективности взаимодействия.

В рамках данного проекта был использован язык программирования *C#*, применяющий объектно-ориентированный подход к программированию. *C#* был выбран из-за его широкой популярности и возможностей для создания масштабируемых и надежных приложений. Объектно-ориентированный подход позволяет разделить приложение на отдельные модули или классы, которые могут быть использованы повторно в различных частях программы. Это способствует повышению эффективности разработки и облегчает поддержку кода в долгосрочной перспективе.

Для создания пользовательского интерфейса в проекте была использована технология *Windows Forms*, позволяющая создавать элементарные абстрактные единицы с интерфейсом, такие как формы. Разработка пользовательского интерфейса производилась путем перетаскивания элементов управления на форму и настройки их свойств. *WinForms* предлагает большое количество предопределенных элементов управления, которые покрывают большую часть потребностей в создании пользовательского интерфейса. Это позволяет быстро и легко создавать функциональные и эстетически приятные окна приложений. Каждый элемент управления в *Windows Forms* имеет широкий набор настраиваемых свойств и событий, которые позволяют полностью настроить их внешний вид и поведение.

В ходе разработки приложения также были созданы специализированные классы для защиты данных, обработки связи с базами данных и манипуляции данными из *БД*.

Для обеспечения безопасности и защиты конфиденциальной информации, такой как личные данные клиентов, были разработаны классы, отвечающие за шифрование и дешифрование данных, управление доступом и аутентификацию пользователей. Эти классы включают методы для шифрования паролей и контроля доступа к конфиденциальной информации.

В данном проекте для взаимодействия с данными и выполнения операций чтения, записи и обновления предусмотрена возможность выбора между тремя типами баз данных: *CSV*, *MongoDB* и *SQL*. Это разнообразие обеспечивает гибкость и универсальность в управлении данными, что делает систему адаптивной к различным требованиям и сценариям использования:

– *CSV*: при выборе *CSV*, данные хранятся в текстовых файлах с разделителями, что облегчает их чтение и запись. Были разработаны специальные классы для чтения и загрузки данных из *CSV* файлов, обеспечивающие простое и эффективное преобразование данных в объекты и структуры программы;

– *MongoDB*: при использовании *MongoDB*, документо-ориентированной NoSQL базы данных, данные хранятся в формате, похожем на *JSON*. Это обеспечивает высокую гибкость и динамичность в работе с данными, особенно когда требуется хранить сложные иерархические структуры;

– *SQL*-базы данных: при использовании реляционных *SQL*-баз данных, предоставляется возможность эффективно управлять структурированными данными, осуществляя сложные запросы и транзакции. Это подходит для ситуаций, где важна целостность данных и четко определенная структура.

В каждом из этих подходов обеспечивается безопасность данных, включая механизмы защиты от потери и повреждения данных в случае ошибок или исключений. Так, например, в случае неудачной записи данных предусмотрены механизмы отката, обеспечивающие сохранность исходных данных.

Использование разнообразных методов хранения данных позволяет легко адаптироваться под различные потребности проекта, обеспечивая легкость обработки данных в разных языках программирования и упрощая дальнейшее развитие и масштабирование системы.

Таким образом, в рамках проекта был создан надежный и гибкий механизм работы с данными, обеспечивающий возможность эффективного управления данными автопарка и предоставления пользовательских услуг на высоком уровне.

Структура проекта была достигнута в результате написания программного кода и проектирования предметной области по заданию курсового проекта. Ключевым элементом архитектуры системы стало применение паттерна «стратегия», который позволил гибко подходить к управлению различными типами баз данных - *CSV*, *MongoDB* и *SQL*.

Паттерн «стратегия» в данном контексте использовался для определения семейства алгоритмов, инкапсулируя каждый из них и делая их взаимозаменяемыми. Это позволило изменять алгоритмы взаимодействия с базой данных независимо от использующих их клиентских приложений. Таким образом, в зависимости от требований и условий работы приложения, можно выбирать наиболее подходящую стратегию для работы с данными - будь то быстрый доступ к данным через *CSV*, гибкое управление большими объемами неструктурированных данных с *MongoDB* или же надежное и структурированное хранение данных с использованием *SQL*-баз данных.

Применение паттерна «стратегия» также способствовало повышению уровня абстракции и уменьшению зависимости между различными компонентами системы. Это облегчило процесс тестирования и масштабирования приложения, так как изменения в одной стратегии не влекли за собой необходимость переработки всей системы. Кроме того, данный подход упростил процесс внедрения новых технологий и изменений в структуре данных, поскольку для этого требовалось лишь добавление новой стратегии, не затрагивающей остальную часть системы.

Основная структура классов и форм проекта представлена на рисунке 3.1.

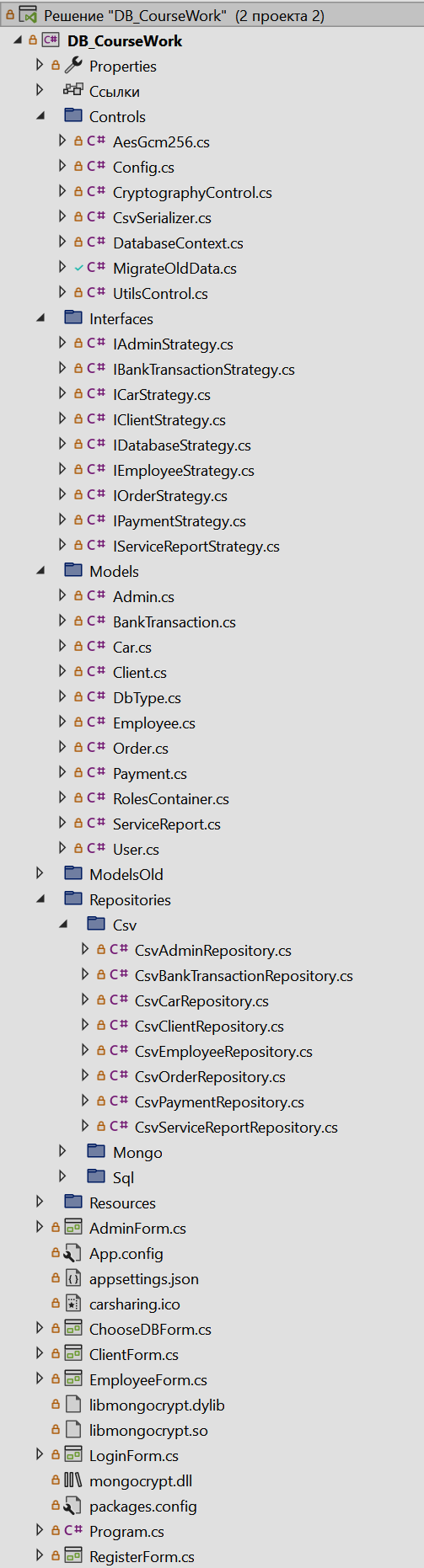


Рисунок 3.1 – Структура программы

**3.2 Описание архитектуры приложения**

Общий подход к архитектуре: в разработке данного проекта была реализована сложная и гибкая архитектура, которая позволяет взаимодействовать с разными типами баз данных - *CSV*, *MongoDB* и *SQL*. Основным принципом, лежащим в основе архитектуры приложения, является применение паттерна «стратегия». Этот подход позволяет динамично менять алгоритмы работы с данными без изменения основного кода приложения.

Использование паттерна «Стратегия»: паттерн «стратегия» в данном контексте был реализован через определение интерфейсов для каждого типа сущности (например, «*IAdminStrategy»*, «*IClientStrategy*» и т.д.), что позволяет легко переключаться между различными реализациями работы с данными. В зависимости от выбранной базы данных (*CSV*, *MongoDB* или *SQL* в данном случае), используются соответствующие классы-репозитории (например, «*CsvAdminRepository*», «*MongoAdminRepository*», «*SqlAdminRepository»*), которые реализуют эти интерфейсы.

Интерфейсы для работы с данными: в архитектуре данного приложения ключевую роль играют интерфейсы, определяющие стандартные операции взаимодействия с данными. Эти интерфейсы следуют принципам *CRUD* (*Create*, *Read*, *Update*, *Delete*), обеспечивая унифицированный подход к управлению данными в различных репозиториях.

Основные *CRUD* операции:

– *Get (Read)*: метод *Get(int id)* предназначен для получения конкретной сущности по ее уникальному идентификатору. Это основная функция для чтения данных, позволяющая извлекать конкретные записи из базы данных. Например, в *SqlAdminRepository* этот метод будет извлекать данные администратора с заданным ID из SQL базы данных;

– *GetAll (Read All)*: метод *GetAll()* обеспечивает извлечение всех существующих записей определенного типа. Этот метод широко используется для получения полных списков сущностей, например, всех пользователей или всех транзакций в системе. В *MongoAdminRepository*, к примеру, он возвращает список всех администраторов из *MongoDB*;

– *Add (Create)*: метод *Add(T entity)* используется для добавления новой сущности в базу данных. Он обрабатывает создание новой записи, управляя всеми необходимыми проверками и логикой для корректного добавления данных. В контексте *CsvAdminRepository*, это включает сериализацию объекта администратора и его запись в соответствующий *CSV* файл;

– *Update (Update)*: метод *Update(T entity)* предназначен для обновления существующей записи. Эта функция позволяет изменять данные сущности, уже находящейся в базе данных, обеспечивая актуальность и целостность информации. В репозиториях, работающих с *SQL*, этот метод может включать выполнение *SQL* команды *UPDATE*;

– *Delete (Delete)*: метод *Delete(int id)* отвечает за удаление записи из базы данных. Этот метод критически важен для управления жизненным циклом данных и обеспечивает возможность удалять устаревшие или ненужные записи. Например, в *CsvBankTransactionRepository*, он удаляет запись транзакции из соответствующего *CSV* файла.

Применение интерфейсов в архитектуре: использование этих интерфейсов обеспечивает единообразие и согласованность во всех классах-репозиториях, независимо от используемого типа базы данных. Каждый класс-репозиторий имплементирует эти интерфейсы, предоставляя конкретную реализацию методов в соответствии с логикой работы выбранной базы данных (*CSV*, *MongoDB*, *SQL*). Это обеспечивает централизованное и унифицированное взаимодействие с данными в приложении, повышая его гибкость, масштабируемость и поддерживаемость.

Строение классов-репозиториев: каждый класс-репозиторий отвечает за взаимодействие с конкретным типом базы данных. Все они имеют похожую структуру, но используют различные технологии для работы с данными:

– *CSV*: классы, работающие с данными в формате *CSV*, используют библиотеку *CsvHelper* для сериализации и десериализации данных. Примером может служить класс *CsvSerializer*, который предоставляет методы *WriteToFile<T>()* и *ReadFromFile<T>()* для записи и чтения данных. Эти методы позволяют эффективно обрабатывать данные в формате *CSV*, обеспечивая простоту и гибкость при работе с большими объемами данных;

– *MongoDB*: репозитории, работающие с *MongoDB*, используют библиотеку *MongoDB.Driver* для взаимодействия с *NoSQL* базой данных. Это обеспечивает высокую производительность и гибкость при работе с документо-ориентированными данными. Примером такого репозитория является *MongoAdminRepository*, который реализует интерфейс *IAdminStrategy* и обеспечивает доступ к данным администраторов в *MongoDB*;

– *SQL*: классы, взаимодействующие с *SQL*-базами данных, используют *ADO.NET* для подключения к базе данных и выполнения *SQL*-запросов. Это позволяет надежно и эффективно управлять структурированными данными в реляционных базах. Например, *SqlAdminRepository* предоставляет методы для получения, добавления, обновления и удаления записей администраторов в *SQL*-базе данных.

Интеграция и координация: основная координация работы с данными осуществляется через класс *DatabaseContext*, который инкапсулирует стратегии для каждого типа сущности и предоставляет унифицированный интерфейс для взаимодействия с данными независимо от используемого типа базы данных. Это обеспечивает высокий уровень абстракции и изолирует клиентский код от конкретных механизмов работы с базами данных.

Класс *DatabaseContext* играет центральную роль в архитектуре приложения, выступая в качестве координатора для всех операций с данными. Он представляет собой основу для взаимодействия с различными типами баз данных (*CSV*, *MongoDB*, *SQL*) и управляет стратегиями для обработки данных каждого типа сущности в приложении.

Особенности и функциональность:

– Универсальность: *DatabaseContext* объединяет в себе различные стратегии взаимодействия с данными, такие как *IClientStrategy*, *IEmployeeStrategy*, *IAdminStrategy* и другие. Это позволяет унифицированно обрабатывать данные независимо от их физического расположения и формата хранения;

– Гибкая настройка: класс позволяет настраивать контекст базы данных в соответствии с выбранным типом (*CSV*, *MongoDB*, *SQL*), используя метод *GenerateDBContext*. Это обеспечивает гибкость в выборе и изменении способа взаимодействия с данными, улучшая масштабируемость и адаптивность приложения;

– Централизованное управление: *DatabaseContext* служит центральной точкой для всех операций *CRUD* в приложении, предоставляя методы для получения (*GetAllUsers*), добавления, обновления и удаления данных. Это упрощает управление данными и повышает эффективность кода;

– Инкапсуляция: класс инкапсулирует сложную логику взаимодействия с базами данных, скрывая детали реализации от клиентского кода. Это уменьшает сложность системы и облегчает ее поддержку и расширение.

Примеры использования:

– При работе с *CSV* базами данных, *DatabaseContext* инициализирует соответствующие *CSV* репозитории (*CsvClientRepository*, *CsvAdminRepository* и т.д.), которые обеспечивают сериализацию и десериализацию данных в/из *CSV* файлов;

– Для *MongoDB*, класс настраивает подключение к *MongoDB* и инициализирует репозитории, работающие с документами в *NoSQL* базе данных. Это включает настройку строк подключения и управление коллекциями *MongoDB*;

– В случае *SQL* баз данных, *DatabaseContext* конфигурирует *SQL* репозитории, которые выполняют операции с данными через *SQL* запросы к реляционной базе данных, обеспечивая надежность и целостность данных.

Значение в архитектуре приложения: *DatabaseContext* выступает в роли связующего звена между различными частями приложения и базами данных, обеспечивая гибкость, масштабируемость и централизованное управление данными. Его гибкая архитектура позволяет легко адаптироваться к изменениям в требованиях и предоставляет прочную основу для дальнейшего расширения функциональности приложения.

Применение паттерна «стратегия» в сочетании с гибко настроенной архитектурой классов-репозиториев обеспечивает мощную, гибкую и масштабируемую систему управления данными, способную адаптироваться к различным требованиям и условиям эксплуатации приложения.

Использование файла конфигурации: для обеспечения гибкости и удобства управления настройками приложения, в проекте используется централизованный файл конфигурации. Этот подход позволяет легко изменять параметры системы без необходимости перекомпиляции кода, что особенно важно для управления путями к файлам, строками подключения и другими критически важными данными.

Структура файла конфигурации: файл «*appsettings.json*» используется в качестве основного хранилища конфигурационных параметров. Он содержит необходимые настройки для каждого типа базы данных, с которыми работает приложение:

– *CSV*: пути к файлам для всех сущностей (клиенты, сотрудники, администраторы и т.д.) указаны в разделе *FilePaths*. Это позволяет легко изменять местоположение хранимых данных в формате *CSV*, не изменяя код репозиториев;

– *MongoDB*: настройки подключения к *MongoDB*, включая строку подключения и названия коллекций, находятся в разделах *ConnectionStrings* и *MongoTableNames*. Это обеспечивает гибкость в управлении подключениями к базам данных *NoSQL* и позволяет легко переключаться между различными средами (например, разработка, тестирование, продакшн);

– *SQL*: строка подключения к *SQL*-базе данных находится в разделе *ConnectionStrings*. Это упрощает управление подключениями к *SQL*-серверам и обеспечивает централизованное управление конфигурациями для разных сред выполнения.

Применение в коде: класс *Config* служит центральным узлом для доступа к этим параметрам. Используя паттерн *Singleton*, он инициализирует и хранит экземпляр *IConfiguration*, который считывает данные из «*appsettings.json***»**. Это позволяет всем компонентам системы легко и безопасно получать доступ к необходимым настройкам.

Пример использования:

– в репозиториях для CSV (*CsvAdminRepository*, *CsvBankTransactionRepository* и т.д.) пути к файлам извлекаются из класса *Config*, что обеспечивает гибкость и удобство управления данными без жесткой привязки к конкретным путям файлов в коде;

– Для *MongoDB* и *SQL* используются соответствующие настройки подключения и названия таблиц/коллекций, которые также извлекаются из *Config*, обеспечивая универсальность и упрощение подключения к различным базам данных.

Таким образом, использование файла конфигурации в сочетании с гибко настроенной архитектурой классов и паттерна «стратегия» позволяет создать удобную, масштабируемую и легко поддерживаемую систему, способную адаптироваться к изменяющимся требованиям и условиям эксплуатации.

Адаптивность и модульность:основная сила данной архитектуры заключается в ее адаптивности и модульности. Реализация паттерна «стратегия» в сочетании с разнообразными базами данных (*CSV*, *MongoDB*, *SQL*) предоставляет уникальную возможность для динамического выбора и изменения подходов к хранению и обработке данных без вмешательства в основную логику приложения. Это делает систему исключительно подходящей для быстро меняющихся требований бизнеса и технологической среды.

Централизация и гибкость: *DatabaseContext*, как центральный элемент архитектуры, обеспечивает централизованное управление и взаимодействие с данными. Этот подход улучшает читаемость кода и облегчает его поддержку, позволяя разработчикам сосредоточиться на бизнес-логике, а не на деталях реализации взаимодействия с базами данных. Использование интерфейсов *CRUD* во всех репозиториях подчеркивает эту унифицированность и гибкость.

Безопасность и конфигурируемость: применение внешнего файла конфигурации «*appsettings.json»* и класса *Config* для централизованного доступа к настройкам дополнительно повышает безопасность системы. Отсутствие жестко закодированных параметров и путей в коде приложения уменьшает риск ошибок и уязвимостей, связанных с неправильной конфигурацией.

Перспективы развития: такая архитектура не только отвечает текущим требованиям, но и предоставляет прочную основу для будущего расширения и интеграции новых технологий. Она подготовлена к масштабированию и адаптации, что является важным фактором для долгосрочного развития и эволюции проекта.

Синтез современных подходов: в заключение, архитектура приложения представляет собой гармоничное сочетание современных подходов к разработке ПО. Она обеспечивает необходимую гибкость, масштабируемость и надежность, позволяя приложению эффективно адаптироваться и расти в соответствии с изменяющимися требованиями и технологическими стандартами.

**4 ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ**

**4.1 Общее описание тестирования проекта**

Обеспечение стабильной и эффективной работы приложения каршеринга требует внимательного отношения к обработке исключительных ситуаций. Это включает в себя сценарии, когда пользователь вводит недопустимые данные, а также возможные ошибки в процессе выполнения операций. В ходе этого проекта было осуществлено рассмотрение и обработка ряда таких исключений. Подробности об этих обработанных исключительных ситуациях представлены в таблице 4.1.

Таблица 4.1 – Тестирование приложения

|  |  |
| --- | --- |
| **Исключительная ситуация** | **Результат** |
| Отправка пустого поля в случае, когда поле должно быть заполнено | Сообщение о том, что поле не может быть пустым |
| Попытка ввода значений не подходящего формата | Сообщение о неправильном формате ввода |
| Ошибка ввода в поле, т.е. ввод неподходящего параметра | Сообщение о недопустимости такого ввода |
| Попытка регистрации в системе с уже существующем паролем | Сообщение о том, что данный пароль уже существует |
| Вход с несуществующим логином или паролем | Вывод сообщения о том, что следует проверить корректность вводимых данных |

В рамках тестирования приложения были произведены проверочные вводы неверных значений. Например, был проведен тест на обработку пустых или неправильных значений в форме авторизации. При подобном вводе программа генерирует и выводит соответствующее сообщение об ошибке. Результат этого теста можно увидеть на рисунке 4.1.

Также была проведена проверка на обработку ситуации, когда при регистрации нового пользователя вводится уже существующий логин или некорректный пароль. В этом случае, система корректно отклоняет попытку регистрации и выводит информативное сообщение, показывающее ошибку. Результат этого теста демонстрируется на рисунке 4.2.

Можно также провести проверку на ввод некорректных значений при бронировании автомобиля, например, ввод невозможных дат и времени. В этом случае программа должна отклонять недопустимый ввод и выводить соответствующее сообщение об ошибке.

Еще одним примером тестового сценария может быть ситуация, когда пользователь пытается продлить бронирование автомобиля, не имея достаточного баланса. В этом случае приложение должно корректно обрабатывать эту ситуацию и информировать пользователя о недостатке средств для выполнения операции (рисунок 4.3).

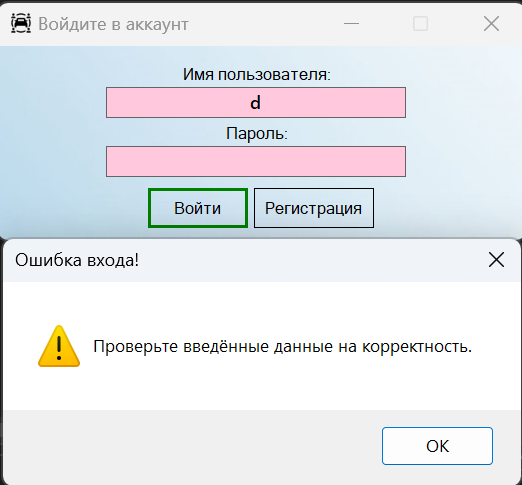


Рисунок 4.1 – Проверка на ошибки в окне авторизации

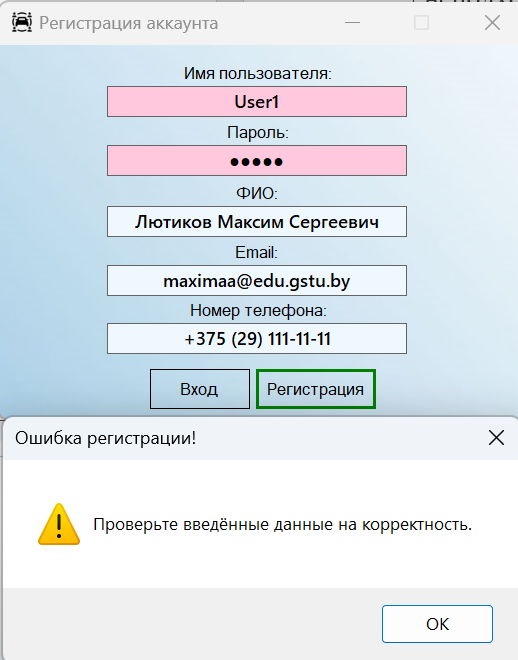


Рисунок 4.2 – Проверка на ошибки в окне регистрации

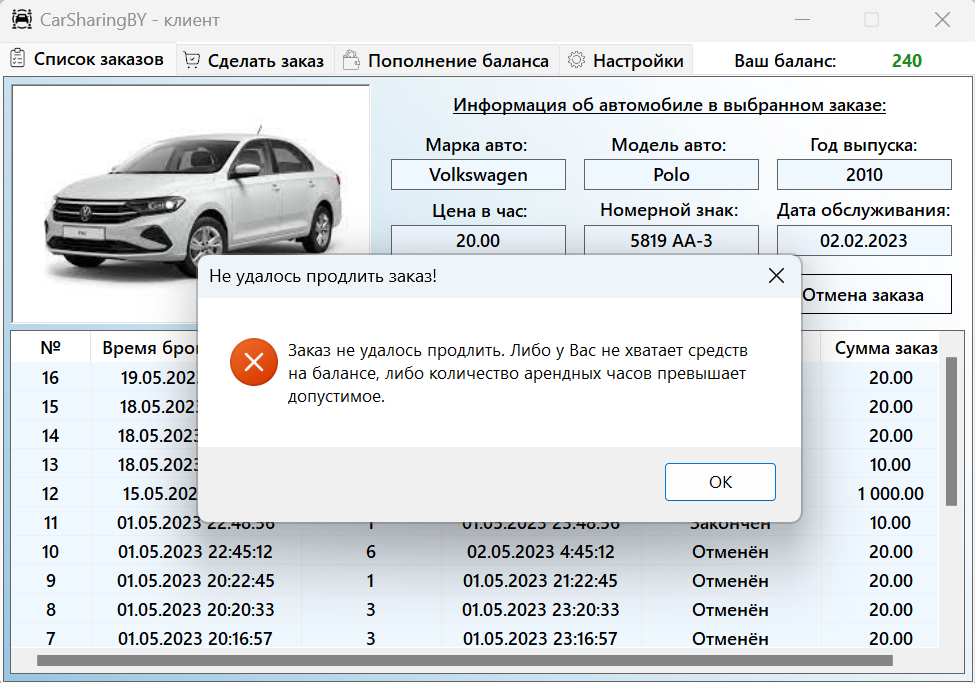


Рисунок 4.3 – Проверка на ошибки в окне авторизации

Другим примером тестового сценария является ситуация, когда пользователь пытается пополнить баланс, вводя некорректные данные. В таком случае приложение должно корректно обрабатывать эту ситуацию и информировать пользователя об ошибке введенных данных при попытке выполнить операцию пополнения баланса.

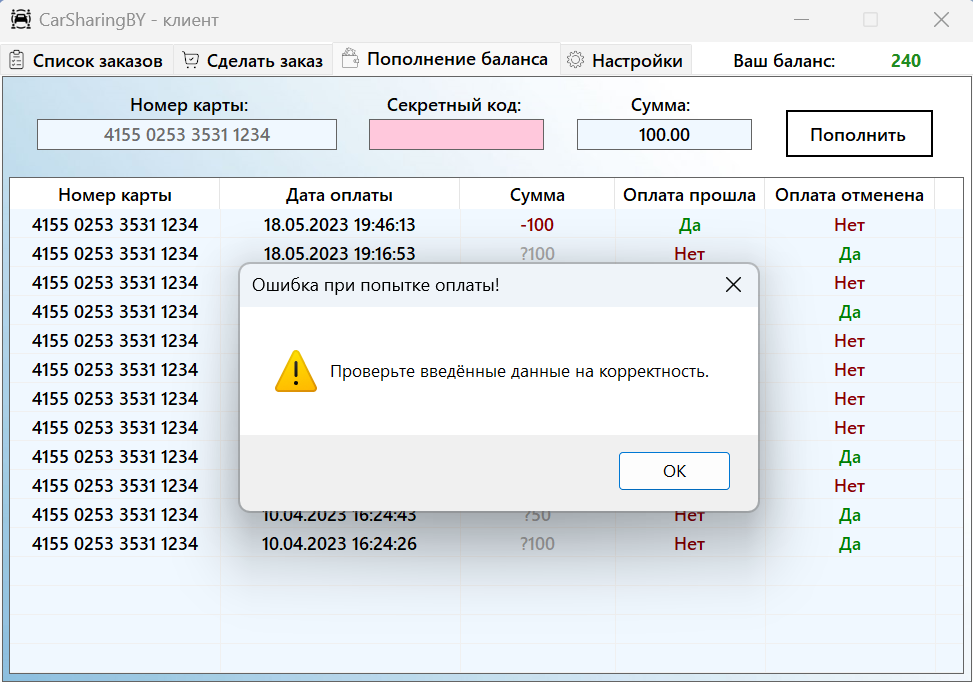


Рисунок 4.4 – Проверка на ошибки во вкладке «Пополнение баланса» клиента

Еще одним примером тестового сценария является ситуация, когда пользователь пытается изменить свои личные данные во вкладке «Настройки» клиента, вводя некорректные данные в поля «Ваш пароль», «Номер карты для списаний», «Email». Приложение в этом случае должно корректно обрабатывать такую ситуацию и информировать пользователя о неверно введенных данных при попытке сохранения изменений в настройках (рисунок 4.5).

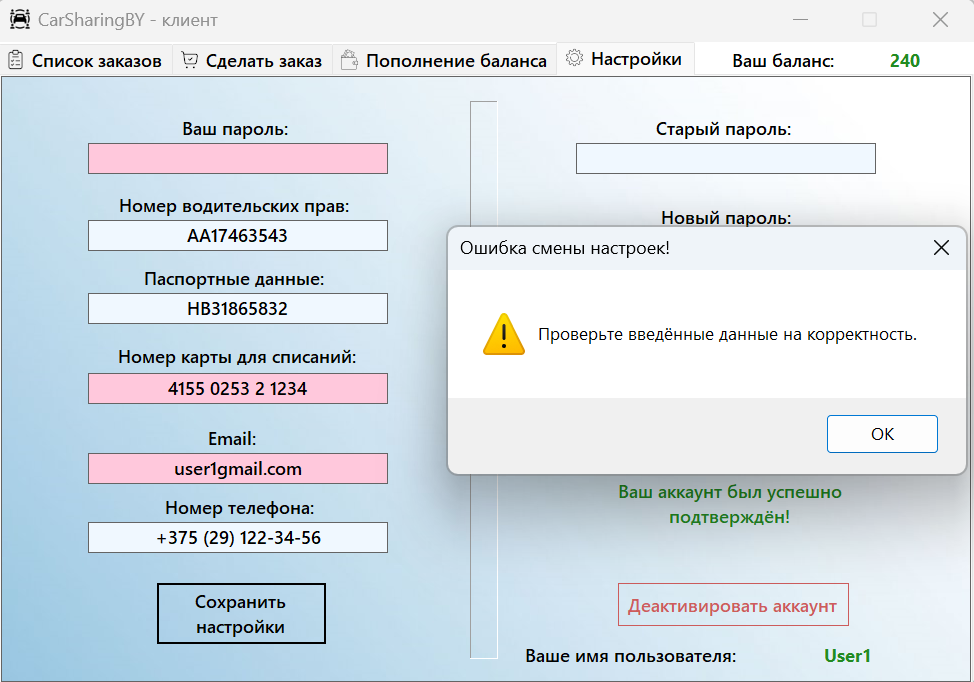


Рисунок 4.5 – Проверка на ошибки во вкладке «Настройки» клиента

Еще одним примером тестового сценария может быть ситуация, когда администратор пытается добавить новую информацию об автомобиле или обновить существующую, вводя некорректные данные в поля «Фотография», «Марка авто», «Модель авто», «Год выпуска», «Цена в час», «Номерной знак». В этом случае приложение должно корректно обрабатывать такую ситуацию и информировать администратора о неверно введенных данных при попытке сохранения изменений (рисунок 4.6).

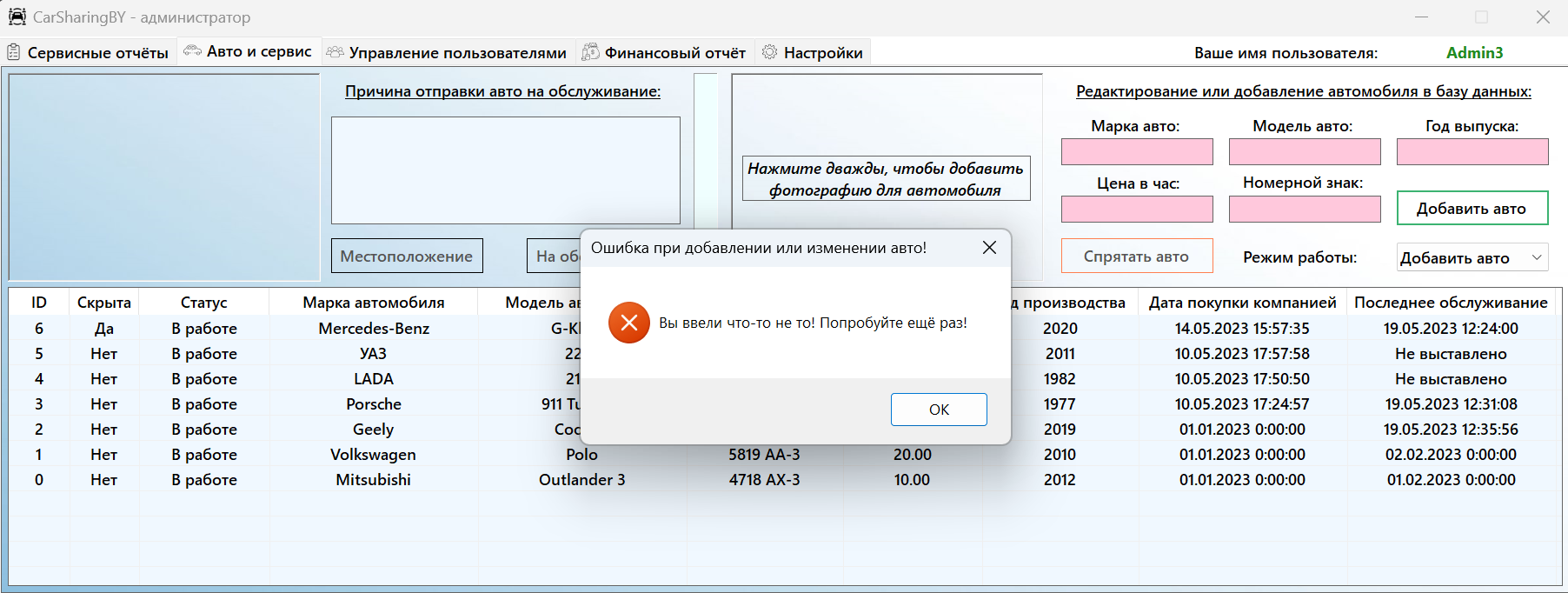


Рисунок 4.6 – Проверка на ошибки во вкладке «Авто и сервис» администратора

**4.2 Функции и их реализация в формах и алгоритмах**

Следующая таблица обозначает взаимосвязь между ключевыми пользовательскими функциями в нашем каршеринг-сервисе и их соответствующими формами и алгоритмами. Каждая функция здесь связана с определённой формой, предоставляющей необходимый пользовательский интерфейс для взаимодействия с системой, и специфическим алгоритмом, который описывает последовательность действий, выполняемых в ходе реализации функции.

Функции включают такие действия, как регистрация нового пользователя, авторизация пользователя, просмотр списка автомобилей, бронирование автомобиля и многие другие. Формы представляют собой пользовательский интерфейс, через который пользователь может взаимодействовать с данными функциями, а алгоритмы - это последовательность действий, которые система выполняет, когда пользователь активирует функцию.

Важно отметить, что это упрощенное представление реальных алгоритмов, используемых в программе. Полные алгоритмы будут гораздо более сложными и включать дополнительные детали и шаги, учитывающие различные случаи и исключения.

Таблица 4.2 – функции и их реализация в формах и алгоритмах

|  |  |  |
| --- | --- | --- |
| **Функция** | **Форма** | **Алгоритм** |
| Регистрация нового пользователя | Форма регистрации | Алгоритм проверки уникальности имени |
| Авторизация пользователя | Форма авторизации | Алгоритм проверки пароля |
| Просмотр списка автомобилей | Форма списка автомобилей | Алгоритм получения данных из базы |
| Бронирование автомобиля | Форма бронирования | Алгоритм бронирования |
| Просмотр и изменение настроек | Форма настроек | Алгоритм обновления настроек |
| Взятие автомобиля на обслуживание | Форма обслуживания | Алгоритм обслуживания |
| Закрытие отчёта обслуживания | Форма отчётов | Алгоритм закрытия отчёта |
| Просмотр отчётов обслуживания | Форма просмотра отчётов | Алгоритм просмотра данных |
| Добавление нового автомобиля | Форма добавления автомобиля | Алгоритм добавления |
| Изменение информации об автомобиле | Форма изменения информации | Алгоритм обновления данных |
| Управление пользователями | Форма управления пользователями | Алгоритм управления |

В рамках данного проекта особое внимание уделялось тестированию ролей Клиента, Сотрудника и Администратора и связанных с ними функций в системе каршеринга. Это было осуществлено с целью обеспечения бесперебойной работы приложения и повышения его эффективности. Безусловно, такое детальное исследование каждой из этих ролей в контексте системы позволяет выявить потенциальные проблемы и несоответствия, которые могут возникнуть в процессе их взаимодействия с системой.

Основной задачей тестирования в данном проекте являлось обеспечение того, что каждая функция, связанная с каждой ролью, работает как предполагалось и не вызывает нежелательных последствий. Это требует особого внимания к деталям, включая тщательное изучение и тестирование каждой пользовательской функции. Каждая роль в системе вносит свой уникальный вклад в общую эффективность и производительность приложения, поэтому их правильное функционирование является жизненно важным для обеспечения высокого качества сервиса.

Тестирование приложения в этом проекте охватывало разнообразные аспекты, включая функциональное тестирование, тестирование производительности, а также тестирование на соответствие требованиям безопасности. Наличие разнообразных пользовательских потребностей и требований подчеркивает важность глубокого понимания и проверки функций каждой роли. Необходимость обеспечения гибкого, надежного и безопасного сервиса делает тестирование ключевым этапом в процессе разработки приложения.

В результате проведенного тестирования в проекте было возможно идентифицировать и устранить различные проблемы и ошибки, что в конечном итоге привело к повышению стабильности и надежности работы приложения, улучшению его производительности и удобства использования. Это подтверждает важность тестирования в рамках процесса разработки приложений, подчеркивая его роль в обеспечении высокого уровня удовлетворенности пользователей и достижении стратегических целей проекта.

**ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсового проекта был проведен глубокий анализ предметной области, связанной с услугами автопарка и каршерингом. Это исследование позволило выявить основные недостатки и задачи в существующих системах, что привело к созданию продукта, соответствующего всем современным требованиям и стандартам, описанным в первой главе. Проект отличается использованием разнообразных типов баз данных - *CSV*, *MongoDB* и *SQL*, в сочетании с паттерном «стратегия», что обеспечивает гибкость и масштабируемость системы.

Аналитическая работа, проведенная в процессе разработки, помогла определить ключевые потребности пользователей каршеринга и создать основу для дальнейшего совершенствования сервиса. Использовались различные подходы, включая анализ технических материалов и изучение статистических данных, что способствовало разработке эффективных решений и стратегий для повышения удовлетворенности пользователей.

Проект был разработан с применением объектно-ориентированного подхода на языке *C#* и включает в себя элементы *Windows Forms* для реализации интерфейса пользователя. Особое внимание было уделено разработке многочисленных *unit*-тестов, обеспечивающих стабильность и надежность работы приложения, а также возможности его дальнейшего расширения и модернизации.

В процессе работы над проектом были учтены разнообразные аспекты эксплуатации приложения, включая обработку исключительных ситуаций для обеспечения комфортной и безопасной работы с сервисом. Итоговое приложение демонстрирует готовность к дальнейшему масштабированию и улучшению, предоставляя возможность для роста до уровня систем, описанных в первой главе, с дальнейшим развитием функционала и улучшением инфраструктуры.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Романькова, Т. Л. Конструирование программ и языки программирования: пособие по одноим. курсу для студентов техн. специальностей дневной формы обучения / Т. Л. Романькова, Е. В. Коробейникова. – Гомель.: ГГТУ им. П.О.Сухого, 2010 – 43 с.
2. Рихтер, Дж. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 3-e изд. – СПб.: Питер, 2016 – 896 с.
3. Албахари, Дж. C# 7.0. Карманный справочник. – СПб.: ООО “Альфа-книга”, 2017. – 224 с.
4. Мартин, Роберт Чистый код. Создание, анализ и рефакторинг / Роберт Мартин.-М.: «Питер», 2019.–464 с.
5. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен – М.: Вильямс, 2015. – 126 c.

**ПРИЛОЖЕНИЕ А**

**ЛИСТИНГ ПРОГРАММНОГО КОДА**

Форма «LoginForm.cs»:

Форма «RegisterForm.cs»:

Форма «ClientForm.cs»:

Форма «EmployeeForm.cs»:

Форма «AdminForm.cs»:

Файл «User.cs»:

Файл «Order.cs»:

Файл «Employee.cs»:

Файл «Client.cs»:

Файл «Car.cs»:

Файл «BankTransaction.cs»:

Файл «Admin.cs»:

Файл «UtilsControl.cs»:

Файл «SaveLoadControl.cs»:

Файл «CryptographyControl.cs»:

Файл «AesGcm256.cs»:

Файл «Program.cs»: