РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
по дисциплине «Модели данных и СУБД»

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

Исполнитель: студент гр. [redacted]
[redacted]

Руководитель: ст. преподаватель
[redacted]

Дата проверки: _____
Дата допуска к защите: _____
Дата защиты: _____

Оценка проекта: _____

Подписи членов комиссии
По защите курсового проекта: _____

2023

СОДЕРЖАНИЕ

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

1.1 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 удоб-

ными 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 также 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.

1.2 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 систе-

мы. 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.

1.3 Требования к проектируемому программному обеспечению

После всестороннего анализа предметной области и взаимосвязей данных с применяемыми технологиями программирования, мы определили комплекс требований к проектируемому приложению, учитывая влияние источников данных на архитектуру и функциональность проекта. Каждый пользователь, который решит установить приложение на свой мобильный или иной девайс, должен получить возможность легкого доступа к всесторонней информации о службе, включая детали заказов, персональный профиль, политику конфиденциальности и прочие важные аспекты. Это предполагает реализацию гибкого и надежного механизма авторизации для существующих клиентов службы, а также интуитивно понятного процесса регистрации для новых пользователей, которые желают воспользоваться предложениями службы.

Проектируя клиентскую часть приложения, особое внимание уделяется функциональности, доступной не только новым, но и существующим пользователям. Клиенты должны иметь возможность не только просматривать актуальное состояние заказов, но и доступ к истории своих обращений в службу, что предполагает использование эффективных инструментов работы с данными.

Службе необходимо иметь возможность администрирования информации о предоставляемых услугах, клиентах и внутренних процессах. Это требует разработки сложной системы управления данными, способной автоматизировать рутинные операции и в то же время обладающей гибкостью для адаптации под нестандартные задачи. Отдельное внимание стоит уделить роли администратора, который должен иметь расширенные полномочия для контроля и корректировки данных в системе, а также обеспечения безопасности информации.

Интерфейс приложения является критическим компонентом в восприятии программы конечным пользователем. Важно, чтобы интерфейс был не только функционально полноценным, но и визуально привлекательным, логически структурированным и соответствовал современным тенденциям *UI/UX* дизайна. Качественно разработанный интерфейс должен облегчать взаимодействие пользователя с приложением, минимизируя вероятность ошибок и недопонимания при его использовании.

В процессе разработки уделяется внимание также и взаимодействию с базами данных различного типа – файловыми (*CSV*), документо-ориентированными (*MongoDB*) и реляционными (*SQL*). Это взаимодействие иг-

рает ключевую роль в обеспечении масштабируемости и гибкости системы. Интеграция данных из разнообразных источников предоставляет широкие возможности для обработки, анализа и хранения информации, что является важным фактором для достижения высокого уровня производительности и стабильности работы приложения. Различные уровни доступа к информации и управление данными должны быть тщательно продуманы и реализованы с учетом требований безопасности и конфиденциальности.

В результате, учитывая все поставленные требования и уникальное влияние источников данных на проект, разрабатываемое приложение каршеринга должно стать образцом удобства и интуитивности для пользователей всех возрастных категорий. Это не только снизит нагрузку на службу поддержки, но и способствует повышению общего уровня удовлетворенности клиентов, а также ускорит процессы обновления и обработки информации внутри системы.

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.

Важным аспектом разработки приложения для каршеринга является его интеграция с существующими транспортными и городскими инфраструктурами. Это означает не только упрощение процесса аренды автомобилей для конечных пользователей, но и взаимодействие с различными системами управления городским трафиком, парковочными сервисами и платежными системами. Подобное взаимодействие позволяет оптимизировать использование транспортных средств и снизить общую нагрузку на транспортную систему города. Кроме того, внедрение экологически чистых транспортных средств в каршеринговые сервисы способствует улучшению экологической обстановки в городах и повышает уровень социальной ответственности компаний, предоставляющих эти услуги. Таким образом, разработка приложения для каршеринга не только облегчает доступ пользователей к аренде автомобилей, но и способствует реализации более широких социальных и экологических целей.

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.

На рисунке 2.1 изображена диаграмма прецендентов и актёров (Клиент).

Рисунок 2.1 – Диаграмма прецедентов и актёров (Клиент)

На рисунке 2.2 изображена диаграмма прецендентов и актёров (Сотрудник).

Рисунок 2.2 – Диаграмма прецедентов и актёров (Сотрудник)

На рисунке 2.3 изображена диаграмма прецендентов и актёров (Администратор).

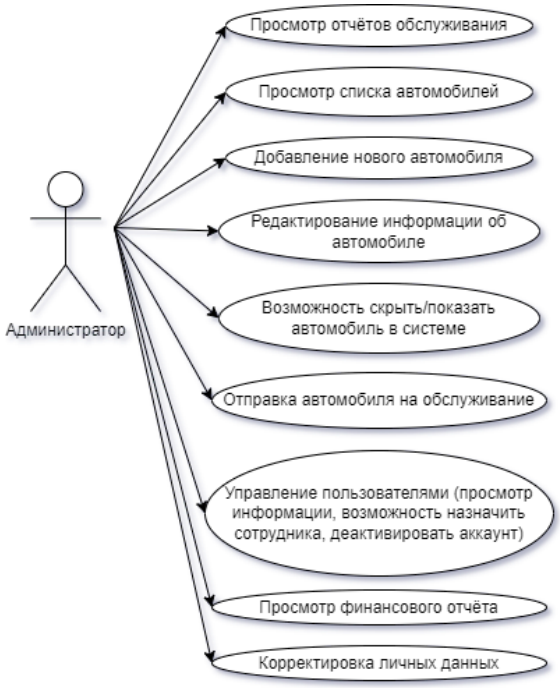

Рисунок 2.3 – Диаграмма прецедентов и актёров (Администратор)

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 данны-

ми 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 гибкостью и эффективностью благодаря использованию *CSV, MongoDB* и *SQL* для хранения данных и *LINQ* для их обработки.

Форматы данных, такие как *CSV* (*Comma-Separated Values*), *BSON* (для *MongoDB*), и *SQL* (*Structured Query Language*), представляют собой различные методы структурирования и хранения данных, каждый из которых имеет свои особенности и применяется в зависимости от конкретных требований и контекста использования.

CSV – это текстовый формат, который отличается своей простотой и идеально подходит для хранения и обработки больших табличных данных. Этот формат легко читается и редактируется, даже с использованием простых текстовых редакторов, и широко поддерживается большинством программных инструментов.

BSON, используемый в *MongoDB*, является бинарным представлением *JSON*-подобных документов. Это позволяет *MongoDB* хранить более сложные структуры данных и обеспечивает более быструю обработку данных по сравнению с текстовым *JSON*, благодаря бинарной сериализации.

SQL, язык структурированных запросов, используется в реляционных базах данных для создания, управления и извлечения данных. Он поддерживает сложные запросы, транзакции и связи между данными, что делает его мощным

инструментом для работы со структурированными данными и обеспечения их целостности.

Выбор между *CSV, BSON* (*MongoDB*) и *SQL* в приложении предоставляет гибкость и позволяет разработчикам оптимизировать систему с учетом специфических требований производительности и управления данными. Это может быть особенно важно для систем, таких как каршеринг, где скорость и надежность являются ключевыми факторами.

2.2 Структура базы данных и СУБД

База данных приложения представлена в виде списка сущностей различных классов, каждый из которых отвечает за определенный аспект функционирования системы, согласно [2]. Основные компоненты базы данных репрезентуют ключевые аспекты бизнес-логики и следующим образом организованы в различных форматах хранения данных:

Файловая база данных (*CSV*):

Клиенты (*Clients*): таблица с атрибутами *Id* (уникальный идентификатор), *Username* (имя пользователя), *Salt* (соль для пароля), *HashedPassword* (хэшированный пароль), *Fullname* (полное имя), *Email* (адрес электронной почты), *Phone* (номер телефона), *Role* (роль в системе), *IsAccountSetupCompleted* (индикатор завершения настройки аккаунта), *AccountDeactivated* (индикатор деактивации аккаунта), включая специфические для клиентов поля *DriverLicense* (номер водительского удостоверения), *Passport* (паспортные данные), *CardNumber* (номер банковской карты), *Balance* (баланс), *SumRating* (общий рейтинг), *OrdersCount* (количество заказов).

Сотрудники (*Employees*): содержат поля *Id, Username, Salt, HashedPassword, Fullname, Email, Phone, Role, IsAccountSetupCompleted, AccountDeactivated*, включая специфические поля *OrdersProcessed* (количество обработанных заказов), *DaysWorked* (количество отработанных дней), *DateHired* (дата найма), *DateFired* (дата увольнения), *DateLastSalaryPayed* (дата последней выплаты зарплаты), *BankAccountNumber* (номер банковского счета), *TotalSalaryPaid* (сумма выплаченной зарплаты), *IsWorkingNow* (статус работы).

Администраторы (*Admins*): включают поля *Id, Username, Salt, HashedPassword, Fullname, Email, Phone, Role, IsAccountSetupCompleted, AccountDeactivated* и *TotalCarsServiced* (общее количество обслуживаемых автомобилей).

Автомобили (*Cars***)** включают детальную информацию об автомобилях, включая их бренд, модель, статус и расположение.

Заказы (*Orders*) хранят записи о всех заказах, с деталями времени заказа, выбранных автомобилях и статусе заказа.

Оплаты (*Payments*) содержат данные об оплатах, с информацией о времени создания и статусе платежа.

Сервисные отчёты (*ServiceReports*) включают отчеты о техническом обслуживании автомобилей, с деталями об услугах и их статусах.

Банковские транзакции (*BankTransactions*) отслеживают банковские транзакции, связанные с операциями аренды.

Нереляционная, или документо-ориентированная база данных (*MongoDB*) содержит документы со всеми вышеупомянутыми полями для сущностей.

Реляционная база данных (*SQL*):

– *Users*: таблица с полями, общими для всех пользователей: *Id*, *Username*, *Salt*, *HashedPassword*, *Fullname*, *Email*, *Phone*, *Role*, *IsAccountSetupCompleted*, *AccountDeactivated*;

– *Clients*: таблица, связанная с *Users*, дополнительно включает поля *DriverLicense*, *Passport*, *CardNumber*, *Balance*, *SumRating*, *OrdersCount*;

– *Employees*: таблица, связанная с *Users*, дополнительно включает поля *OrdersProcessed*, *DaysWorked*, *DateHired*, *DateFired*, *DateLastSalaryPayed*, *BankAccountNumber*, *TotalSalaryPaid*, *IsWorkingNow*;

– *Admins*: таблица, связанная с *Users*, дополнительно включает поле *TotalCarsServiced*;

– *Cars*: таблица с детальной информацией о каждом автомобиле и его статусе в системе;

– *Orders*: таблица, содержащая записи о всех заказах, их статусе и связанных с ними операциях, связана с *Payments* и *Cars*;

– *Payments*: таблица, в которой фиксируются все платежи, совершенные пользователями системы, связана с пользователями;

– *ServiceReports*: таблица с отчетами о техническом обслуживании автомобилей, предоставляющая информацию о выполненных работах и их статусе, связана с *Cars* и *Employees*;

– *BankTransactions*: таблица, отслеживающая транзакции, связанные с банковскими операциями аренды, связана с пользователями.

Эти компоненты представляют собой базовые конструкции, на которых строится функционал приложения, и являются фундаментальными элементами для обработки, хранения и управления данными в различных контекстах использования сервиса.

Для реляционной базы данных (*SQL*) можно так же выделить ряд ограничений, которые обеспечивают целостность и правильное функционирование системы:

– Первичные ключи (*PRIMARY KEY*): используются для уникальной идентификации каждой записи в таблице. Например, *Id* в таблице *Users* и

UserId в таблицах *Clients*, *Admins*, *Employees* являются первичными ключами, что обеспечивает уникальность каждой записи;

– Внешние ключи (*FOREIGN KEY*): определяют связи между таблицами. Например, *UserId* в таблице *Clients* ссылается на *Id* в таблице *Users*, что обеспечивает ссылочную целостность между клиентами и пользователями;

– Уникальность (*UNIQUE*): не указано явно в коде, но может быть добавлено для обеспечения уникальности значений в определенных столбцах, таких как *Username* или *Email* в таблице *Users*;

– Не *NULL* (*NOT NULL*): гарантирует, что значение в поле не может быть *NULL*. Это обеспечивает, что критически важные данные, такие как *Username* и *HashedPassword* в таблице *Users*, всегда будут присутствовать;

– Проверка (*CHECK*): не применяется в данном коде, но может использоваться для ограничения диапазона значений или форматов данных. Например, ограничение *Role* на определенные числовые значения, соответствующие ролям;

– Значение по умолчанию (*DEFAULT*): не используется в данном коде, но может быть применено для автоматического заполнения поля, если при вставке значения не указано;

– Индексы (*INDEX*): хотя индексы не указаны в *SQL* коде, они часто используются для ускорения поиска по столбцам, которые часто участвуют в запросах, например, по *Username*;

– Ограничения на диапазон значений: для полей, таких как *Role*, *IsAccountSetupCompleted*, *AccountDeactivated*, *IsWorkingNow*, *IsStarted*, *IsFinished*, *IsPayed*, *IsRefunded*, *IsCancelled*, и *IsFinished* в разных таблицах, используется тип данных *BIT*, который может принимать значения 0 или 1, что соответствует логическому типу данных (*true* или *false*).

Эти ограничения важны для поддержания структурированности данных, обеспечения их корректности и предотвращения ошибок в базе данных. Они помогают поддерживать логику бизнес-процессов и являются ключевым элементом в проектировании надежных информационных систем.

При взаимодействии с реляционной базой данных, особенно когда работа ведётся с сущностями, такими как *Clients*, *Employees*, и *Admins*, важно учитывать их связь с таблицей *Users*. В отличие от документо-ориентированных или файловых баз данных, где сущности могут быть представлены в виде самодостаточных документов или строк, реляционные базы данных часто требуют выполнения связанных операций через отношения между таблицами.

В случае *Clients*, *Employees*, и *Admins*, каждая из этих сущностей связана с *Users* через внешний ключ *UserId*, что отражает связь «один к одному». Это означает, что для получения полной информации о клиенте, сотруднике или администраторе необходимо выполнить запрос не только к соответствующей

таблице, но и к таблице *Users*. Например, чтобы получить полные данные о клиенте, нужно сначала извлечь общие пользовательские данные из *Users*, а затем дополнительные данные о клиенте из *Clients*.

Таким образом, операции чтения, обновления или удаления информации о конкретном клиенте, сотруднике или администраторе потребуют как минимум двух *SQL* запросов:

– операция *SELECT* для извлечения данных пользователя из таблицы *Users*;

– операция *SELECT* для извлечения специфических данных из таблиц *Clients*, *Employees*, или *Admins*.

Это необходимо, поскольку общие данные пользователя хранятся отдельно от специализированной информации, связанной с его ролью в системе. Такая нормализация данных помогает избежать избыточности и повышает целостность данных, но в то же время может привести к необходимости выполнения множественных запросов и соединений таблиц (*JOIN* операций), что может повлиять на производительность при больших объемах данных.

В сравнении с файловой (*CSV*) и документо-ориентированной (*MongoDB*) базами данных, где данные могут быть извлечены одним запросом без необходимости соединения, реляционная модель требует более сложных запросов из-за разделения данных на связанные таблицы. Это различие в подходах к организации данных важно учитывать при разработке систем на основе *SQL* баз данных для оптимизации запросов и обеспечения высокой производительности системы.

Сущности для файловой (*CSV*) и нереляционной (*MongoDB*) базы данных полностью совпадают с классами, используемыми в логике приложения.

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.

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 баз данных и их связью с выбранным языком программирования были разработаны структуры данных, соответствующие требованиям курсового проекта и активно применяемые в приложении службы социальной помощи. В документации представлены схемы для трех моделей баз данных: файловой (*CSV*), документно-ориентированной (*MongoDB*) и реляционной (*SQL*). Каждая схема демонстрирует использование отношений, основные атрибуты и связи, уникальные для конкретной модели данных. На рисунке 2.5 представлена схема реляционной базы данных *SQL*, на рисунке 2.6 – схема для файловой модели данных *CSV*, а рисунок 2.7 иллюстрирует структуру документно-ориентированной базы данных *MongoDB*, каждая из которых отражает специфические методы хранения и обработки данных в соответствующем формате.

На рисунке 2.5 представлена схема реляционной базы данных *SQL*.

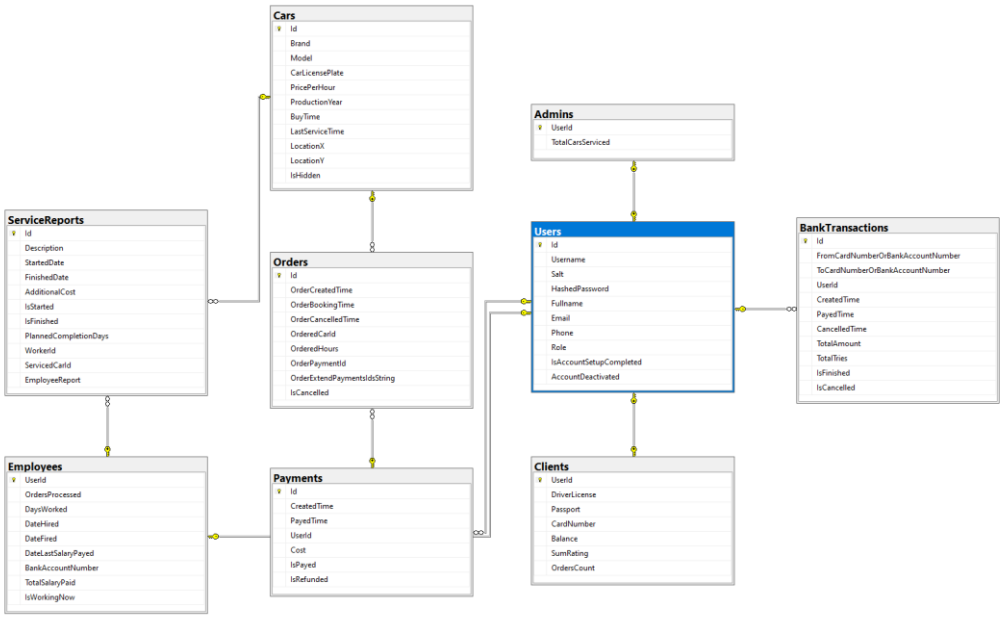

Рисунок 2.5 – Схема реляционной модели базы данных

На рисунке 2.6 представлена схема файловой базы данных *CSV*.

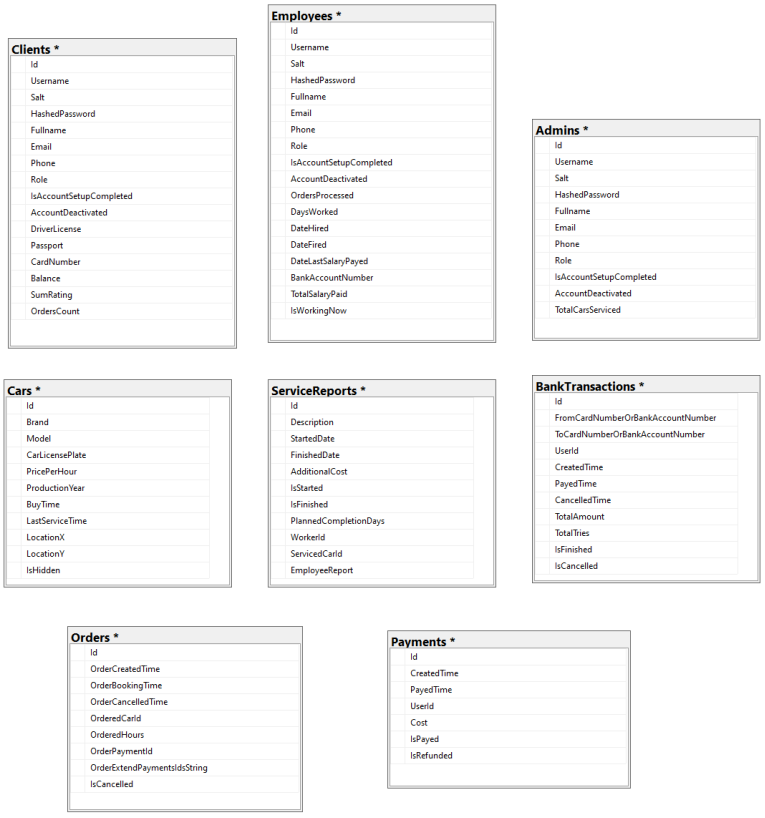

Рисунок 2.6 – Схема файловой модели базы данных

На рисунке 2.7 – схема документно-ориентированной *БД MongoDB*.

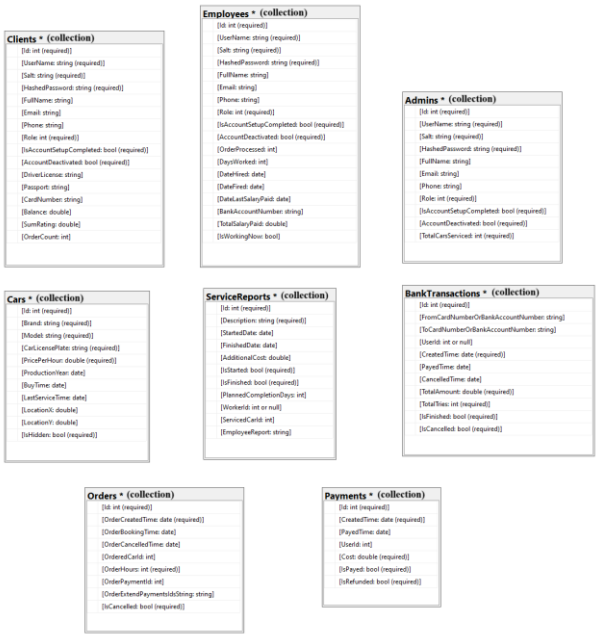

Рисунок 2.7 – Схема документно-ориентированной модели базы данных

Перейдём к описанию моделей данных в трёх типах – файловая модель данных (*CSV*), документо-ориентированная модель данных (*MongoDB*), реляционная модель данных (*SQL*).

Файловая модель данных (*CSV*):

– *CSV* представляет собой простой текстовый формат, который идеально подходит для хранения табличных данных, таких как списки и таблицы;

– этот формат отличается своей простотой и портативностью, позволяя легко передавать данные между различными системами;

– в контексте моделирования услуг автопарка, *CSV* может использоваться для хранения данных, которые не требуют сложной организации, таких как списки пользователей или транзакций;

– основным недостатком является отсутствие поддержки сложных запросов и ограниченные возможности для масштабирования.

Документо-ориентированная модель данных (*MongoDB*):

– *MongoDB* использует формат *BSON* (бинарный *JSON*) для хранения данных, позволяя хранить сложные иерархические структуры в одном документе;

– это обеспечивает высокую гибкость и динамичность при работе с данными, особенно когда требуется хранить неструктурированные или быстро меняющиеся данные;

– *MongoDB* подходит для сценариев, где часто требуется добавление или изменение структуры данных, так как не требует строгой схемы данных;

– это идеальный выбор для приложений, которым необходимо масштабируемое и гибкое управление большими объемами данных.

Реляционная модель данных (*SQL*):

– реляционные базы данных используют структурированные запросы (*SQL*) для управления данными и обеспечивают четко определенную структуру данных;

– они идеально подходят для систем, где важна целостность и безопасность данных;

– реляционные базы данных позволяют проводить сложные запросы и операции объединения, что делает их подходящими для сложных приложений с большим количеством пользователей;

– они обеспечивают эффективное управление структурированными данными и поддержку транзакционности.

Интеграция различных типов баз данных:

– интеграция *CSV, MongoDB* и *SQL* в проект обеспечивает гибкость, позволяя выбирать наиболее подходящий тип хранения данных в зависимости от требований функционала приложения;

– такое комбинированное использование различных баз данных позволяет достичь оптимального баланса между гибкостью, масштабируемостью и эффективностью управления данными;

– это создает надежную и высокопроизводительную платформу для разработки и поддержки сложных приложений, учитывая уникальные требования и характеристики каждого типа базы данных**.**

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 *AesGcm256, 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.

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.

Преимущества файловой модели данных (*CSV*):

– простота обработки: файлы *CSV* легко читаемы и редактируемы как человеком, так и программой, что упрощает их обработку и анализ;

– универсальность: формат *CSV* широко распространен и поддерживается большинством программных инструментов, что обеспечивает легкость интеграции и обмена данными;

– отсутствие зависимости от схемы: *CSV*-файлы не требуют определения строгой схемы, что позволяет быстро адаптироваться к изменениям структуры данных без необходимости модификации базы данных.

Преимущества документо-ориентированная модель данных (*MongoDB*):

– гибкость схемы данных: *MongoDB* не требует фиксированной схемы данных, что позволяет легко добавлять, удалять или изменять поля в документах без прерывания работы приложения;

– скорость разработки: простая структура таблиц упрощает процесс разработки, так как разработчикам не нужно создавать сложные запросы для объединения данных из разных таблиц;

– эффективность: хранение связанных данных в одном документе устраняет необходимость выполнения множественных операций соединения, что может улучшить производительность для некоторых операций чтения.

Преимущества реляционная модель данных (*SQL*):

– целостность данных: реляционные базы данных поддерживают транзакции и строгие правила ссылочной целостности, что обеспечивает надежность и последовательность данных;

– сложные запросы: *SQL*-язык запросов позволяет выполнять сложные запросы с использованием операций объединения и подзапросов, что делает его мощным инструментом для анализа и отчетности.

Интеграция различных типов баз данных: использование разнообразных типов баз данных в проекте обеспечивает уникальную гибкость, позволяя выбирать наиболее подходящий тип хранения данных в зависимости от конкретной задачи. Например, простые операции чтения и записи можно эффективно выполнять с помощью *CSV*, в то время как для управления сложной логикой бизнес-процессов лучше подходит реляционная модель. Документо-ориентированный подход *MongoDB* будет идеален для сценариев, требующих большой гибкости и скорости работы с неструктурированными или постоянно меняющимися данными.

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 при-

ложения, 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.

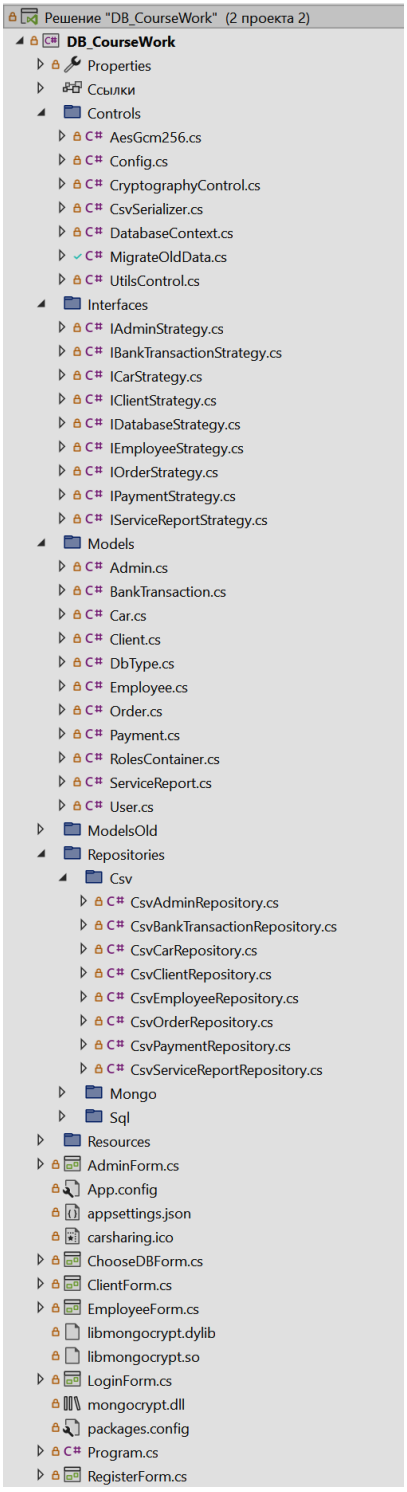

Рисунок 3.1 – Структура программы

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 проекте реализован не только на уровне определения интерфейсов для различных типов сущностей, но и через динамическое создание контекстов данных в зависимости от выбора базы данных пользователем. Это позволяет приложению гибко адаптироваться к различным средам хранения данных, обеспечивая легкость переключения между разными реализациями работы с данными, будь то *CSV*, *MongoDB* или *SQL*.

Выбор базы данных и инициализация контекста:

– в начале работы приложения, пользователь выбирает тип базы данных через графический интерфейс (*ChooseDBForm*);

– в зависимости от выбора (*DbType.CSV*, *DbType.Mongo* или *DbType.SQL*), метод *GenerateDBContext* создает соответствующий контекст данных;

– этот подход позволяет динамически настраивать приложение для работы с различными источниками данных, используя конфигурационные параметры для подключения к базам данных.

Реализация интерфейсов и репозиториев:

– каждый класс-репозиторий, такой как *CsvAdminRepository*, реализует определенный интерфейс (*IAdminStrategy*), обеспечивая специфичную для типа данных логику работы;

– например, в *CsvAdminRepository*, методы *Add, Get, GetAll, Update*, и *Delete* работают непосредственно с файлами *CSV*, используя сериализацию и десериализацию данных через *CsvSerializer*;

– эти репозитории обеспечивают необходимую абстракцию и изоляцию от конкретных механизмов хранения данных, позволяя однородно работать с разными форматами без изменения основной бизнес-логики приложения.

Гибкость и расширяемость:

– такая архитектура обеспечивает значительную гибкость, так как можно легко добавить поддержку новых типов баз данных, реализовав соответствующие стратегии и репозитории;

– приложение может быть быстро адаптировано под изменяющиеся требования или новые технологии хранения данных, что делает его устойчивым к изменениям в технологическом ландшафте.

Таким образом, использование паттерна «Стратегия» в данном проекте не только обеспечивает четкую структурированность и гибкость в управлении различными типами данных, но и подчеркивает важность архитектурного подхода в создании масштабируемых и легко адаптируемых приложений. Эта модульность и абстракция от конкретных механизмов хранения данных позволяет разработчикам сосредоточиться на бизнес-логике, обеспечивая при этом надежное и эффективное управление данными.

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.

Интерфейс *IDatabaseStrategy<T>* представляет общую стратегию для работы с данными, где *T* — это тип данных, с которыми будет производиться работа. Он содержит методы *GetAll, Get, Add, Update* и *Delete*, которые соответствуют основным операциям *CRUD* (*Create, Read, Update, Delete*). Эти методы обеспечивают универсальный шаблон для извлечения, добавления, обновления и удаления записей, делая код более модульным и легко масштабируемым.

Однако, в реальных приложениях часто возникает необходимость в специфических операциях, которые не покрываются стандартным интерфейсом *IDatabaseStrategy<T>*. В таких случаях создается прослойка в виде специализированного интерфейса, например, *IAdminStrategy*, который наследует от *IDatabaseStrategy<Admin>*. Это предоставляет гибкость для введения дополнительных методов специально для администраторов в будущем, не изменяя основной интерфейс стратегии и не влияя на другие части системы, которые используют общие операции.

Использование такой прослойки позволяет легко расширять или модифицировать функционал конкретного репозитория, обеспечивая высокую степень абстракции и соблюдение принципов *SOLID*, в частности принципа открытости/закрытости (*open/closed principle*), который говорит о том, что программные сущности должны быть открыты для расширения, но закрыты для модификации.

На рисунке 3.2 представлен для примера код *IAdminStrategy.cs*:

```
Ссылок: 5 | Maxim Lyutikov, 21 дн. назад | Автор: 1, изменение: 1
internal interface IAdminStrategy : IDatabaseStrategy<Admin>
{
}
```

Рисунок 3.2 – код интерфейса *IAdminStrategy*

На рисунке 3.3 представлен для примера код *IDatabaseStrategy.cs*:

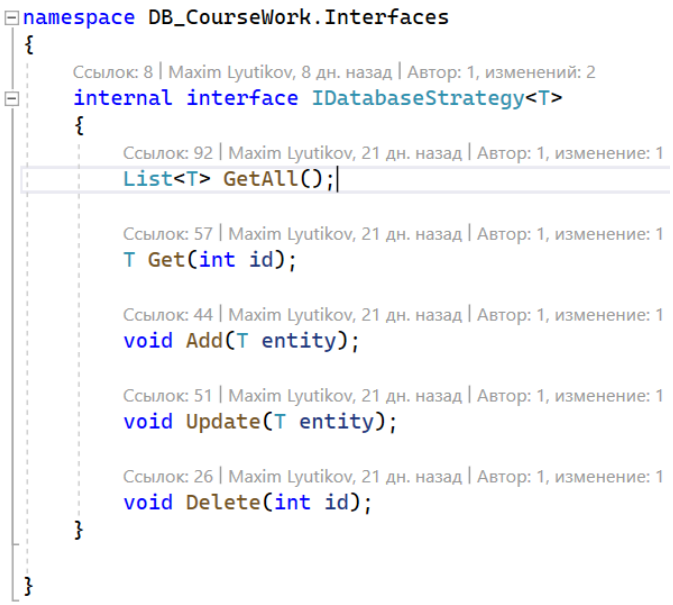

Рисунок 3.3 – код интерфейса *IDatabaseStrategy*

Основные *CRUD* операции в интерфейсах:

– *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.

Стоит отметить разницу в типах используемых баз данных в контексте *CRUD* операций. Различия и отличительные особенности *CRUD* операций в *CSV*, *MongoDB*, *SQL*, указаны ниже.

CSV базы данных:

– *Read*: чтение из *CSV* файла включает в себя десериализацию строк файла в объекты. Для поиска конкретного объекта, весь файл должен быть прочитан и проанализирован, что может быть менее эффективно по сравнению с индексированными базами данных;

– *Read All*: получение всех записей из *CSV* файла влечет за собой чтение и десериализацию каждой строки файла, что может быть ресурсоемким для больших файлов;

– *Create*: добавление новой записи в *CSV* файл включает в себя сериализацию объекта и добавление его в конец файла, что просто, но не поддерживает транзакционность;

– *Update*: обновление записи требует чтения файла, модификации данных и перезаписи всего файла, что неэффективно и рискованно в плане целостности данных;

– *Delete*: удаление записи из *CSV* файла требует чтения всего файла, удаления строки и перезаписи файла, что может быть времязатратным.

MongoDB базы данных:

– *Read*: благодаря *NoSQL* структуре, чтение данных в *MongoDB* более гибкое и может включать сложные запросы с использованием различных фильтров;

– *Read All*: получение всех записей в *MongoDB* эффективно благодаря оптимизированному хранению документов и возможности индексации;

– *Create*: добавление новой записи в *MongoDB* обычно быстрее, чем в традиционных реляционных базах, и поддерживает более гибкие структуры данных;

– *Update*: *MongoDB* позволяет обновлять как отдельные поля, так и целые документы, что обеспечивает гибкость и эффективность;

– *Delete*: удаление в *MongoDB* может быть выполнено как для отдельных документов, так и для групп документов, что делает эту операцию более гибкой.

SQL базы данных:

– *Read*: *SQL*-запросы для чтения данных обычно быстрые и эффективные благодаря хорошо оптимизированным механизмам индексации и кэширования;

– *Read All*: получение всех записей из *SQL* базы данных эффективно, особенно при наличии индексов, и поддерживает сложные запросы с использованием *JOIN*;

– *Create*: создание новой записи в *SQL* базе данных требует строгого соответствия схеме и обычно поддерживает транзакционность и целостность данных;

– *Update*: *SQL* базы данных обеспечивают мощные и гибкие возможности для обновления данных, позволяя изменять множественные записи одним запросом;

– *Delete*: удаление данных в *SQL* базах контролируется с точки зрения целостности данных, и операция может быть отменена в рамках транзакции, если это необходимо для поддержания целостности данных.

Эти различия подчеркивают, что выбор между *CSV*, *MongoDB* и *SQL* для *CRUD* операций должен быть основан на конкретных требованиях приложения, учитывая такие факторы, как объем данных, необходимость транзакционности, скорость доступа к данным, и сложность запросов.

Применение интерфейсов в архитектуре приложения предполагает использование этих интерфейсов для обеспечения единообразия и согласованности 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 подчеркивает их важную роль в архитектуре приложения. 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. Главное отличие здесь — специализация на плоском представлении данных, которое хорошо подходит для простых и четко структурированных датасетов. Как можно заметить на рисунке 3.4, работа с базой данных происходит без непосредственных команд, а через объект.

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*. Отличительной особенностью *MongoDB* является работа с документами, что позволяет более гибко структурировать данные, особенно в случаях сложных или иерархических структур данных. Как можно заметить на рисунке 3.5, работа с базой данных происходит без непосредственных команд, а через объект, как и в случае с *CSV*.

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. *SQL*-репозитории отличаются своей способностью работать с сложными запросами и транзакциями, что делает их идеальным выбором для сценариев с высокими требованиями к целостности и безопасности данных. Как можно заметить на рисунке 3.6, учитывая связи между таблицами в реляционной базе данных, в репозитории для получения *Admins*, *Employees*, *Clients*, понадобится так же обращаться к таблице *Users*.

На рисунке 3.4 изображён код получения *Admin* в *CsvAdminRepository*.

```
Ссылок: 30 | Maxim Lyutikov, 21 дн. назад | Автор: 1, изменение: 1
public Admin Get(int id)
{
    return CsvSerializer.ReadFromFile<Admin>(_path).Find(admin => admin.Id == id);
}
```

Рисунок 3.4 – код получения *Admin* в *CsvAdminRepository*

На рисунке 3.5 изображён код получения *Admin* в *MongoAdminRepository*.

```
Ссылок: 31 | Maxim Lyutikov, 8 дн. назад | Автор: 1, изменение: 1
public Admin Get(int id)
{
    return _admins.Find(admin => admin.Id == id).FirstOrDefault();
}
```

Рисунок 3.5 – код получения *Admin* в *MongoAdminRepository*

На рисунке 3.6 изображён код получения *Admin* в *SqlAdminRepository*.

Рисунок 3.6 – код получения *Admin* в *SqlAdminRepository*

Архитектура приложения с разнообразными базами данных требует гибкого подхода к обработке данных. В этом контексте репозитории играют ключевую роль, обеспечивая интерфейс для взаимодействия с данными и их конкретными реализациями для разных типов хранилищ, на примере *Admin*:

CSV: как видно на рисунке 3.4, репозитории, работающие с *CSV*, используют библиотеку *CsvHelper* для сериализации и десериализации данных. В *CsvAdminRepository*, метод *Get(int id)* реализуется через вызов *CsvSerializer.ReadFromFile<Admin>(_path).Find(admin => admin.Id == id)*, который читает весь файл и ищет запись, соответствующую предоставленному идентификатору. Это подход характеризуется непосредственной работой с файловой системой и подходит для сценариев с простым доступом к данным без сложной логики запросов.

MongoDB: как видно на рисунке 3.5, репозитории для *MongoDB* взаимодействуют с базой данных через библиотеку *MongoDB.Driver*. Например, в *MongoAdminRepository, Get(int id)* может быть реализован с использованием фильтрации документов по идентификатору, что показано в строке *_admins.Find(admin => admin.Id == id).FirstOrDefault()*. Такой подход эффективен для работы с документо-ориентированными базами данных, где данные часто представлены в виде сложных и вложенных структур.

SQL: как видно на рисунке 3.6, в репозиториях, работающих с *SQL* базами данных, таких как *SqlAdminRepository*, взаимодействие с данными осуществляется через *SQL*-запросы. Метод *Get(int id)* реализуется через выполнение запроса к базе данных, который извлекает данные, соединяя таблицы *Admins* и *Users* с помощью оператора *JOIN*: *SELECT * FROM Admins JOIN Users ON Admins.UserId = Users.Id WHERE Users.Id = @Id*. Этот подход позволяет эффективно управлять связанными данными, обеспечивая комплексные операции с реляционными структурами и поддержку транзакционности.

В целом, применение различных технологий для работы с данными в разных репозиториях позволяет оптимально использовать их преимущества в за-

висимости от требований и особенностей предметной области. Централизованное использование интерфейсов в архитектуре приложения обеспечивает единообразный подход к взаимодействию с данными, при этом каждый тип репозитория способен максимально эффективно использовать свои уникальные особенности. Это позволяет создать масштабируемое и гибкое приложение, способное работать с разнообразными типами данных и обеспечивать высокий уровень производительности и безопасности.

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. Для лучшего понимания, участок кода с созданием контекста указан на рисунке 3.7.

```
Ссылок: 1 | Maxim Lyutikov, 8 дн. назад | Автор: 1, изменений: 2
public static DatabaseContext GenerateDBContext(DbType dbType)
{
    switch (dbType)
    {
        case DbType.CSV:   return new DatabaseContext(new CsvClientRepository(), new CsvEmployeeRepository(), new CsvAdminRepository(),
                                                      new CsvCarRepository(),    new CsvOrderRepository(),    new CsvPaymentRepository(),
                                                      new CsvBankTransactionRepository(), new CsvServiceReportRepository());
        case DbType.Mongo: string mongoConnectionString = Config.MONGO_CONNECTION_STRING, mongoDatabaseName = Config.MONGO_DATABASE_NAME;
                           return new DatabaseContext(new MongoClientRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoEmployeeRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoAdminRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoCarRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoOrderRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoPaymentRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoBankTransactionRepository(mongoConnectionString, mongoDatabaseName),
                                                      new MongoServiceReportRepository(mongoConnectionString, mongoDatabaseName));
        case DbType.SQL:   string sqlConnectionString = Config.SQL_CONNECTION_STRING;
                           return new DatabaseContext(new SqlClientRepository(sqlConnectionString), new SqlEmployeeRepository(sqlConnectionString),
                                                      new SqlAdminRepository(sqlConnectionString), new SqlCarRepository(sqlConnectionString),
                                                      new SqlOrderRepository(sqlConnectionString), new SqlPaymentRepository(sqlConnectionString),
                                                      new SqlBankTransactionRepository(sqlConnectionString),
                                                      new SqlServiceReportRepository(sqlConnectionString));
    }
    return null;
}
```

Рисунок 3.7 – Функция для создания контекста

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.

Эволюция и оптимизация: в дополнение к вышеописанным преимуществам, текущая архитектура также уделяет особое внимание оптимизации производительности и эволюционному развитию системы. Эффективное управление ресурсами и производительностью, достигаемое благодаря тщательно продуманному разделению ответственности между компонентами, позволяет приложению масштабироваться в соответствии с растущими нагрузками и объемами данных. Продуманное кэширование, ленивая загрузка и асинхронные операции с данными могут быть интегрированы в систему для обеспечения мгновенного отклика и улучшения пользовательского опыта. В то же время, модульность архитектуры предоставляет возможности для непрерывного внедрения нововведений и технологических улучшений, сохраняя систему актуальной и конкурентоспособной на передовом крае технологических инноваций.

Таким образом, архитектура приложения представляет собой сложный, но гибкий механизм, в котором каждый компонент выполняет свою роль, обеспечивая бесперебойную и эффективную работу всей системы. Стремление к совершенству в каждом аспекте разработки - от выбора архитектурного стиля до реализации конкретных методов работы с данными - является залогом создания надежного и удобного приложения, которое будет служить пользователям долгие годы.

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).

На рисунке 4.1 изображена проверка на ошибки в окне авторизации.

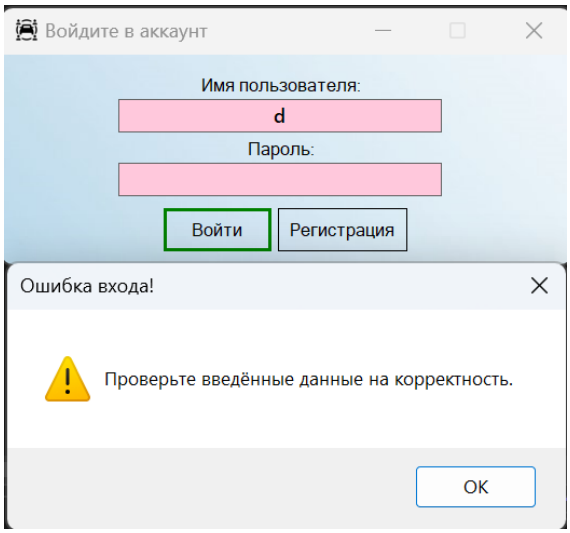

Рисунок 4.1 – Проверка на ошибки в окне авторизации

На рисунке 4.2 изображена проверка на ошибки в окне регистрации.

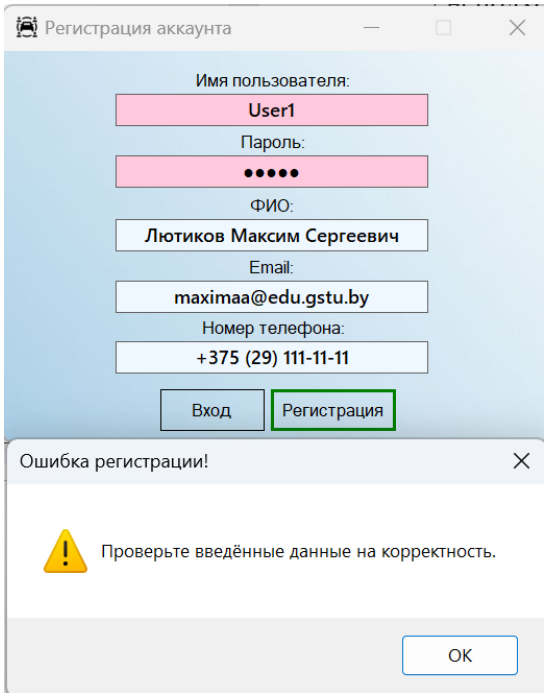

Рисунок 4.2 – Проверка на ошибки в окне регистрации

На рисунке 4.3 так же изображена проверка на ошибки в окне авторизации.

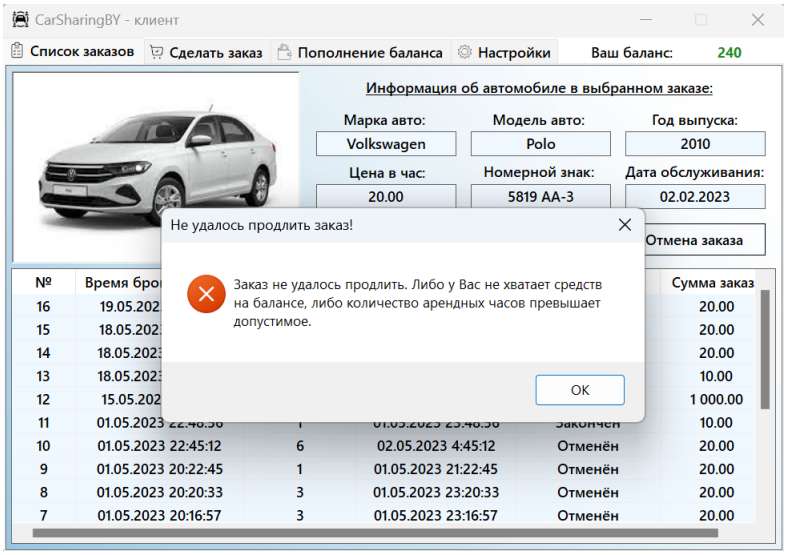

Рисунок 4.3 – Проверка на ошибки в окне авторизации

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.

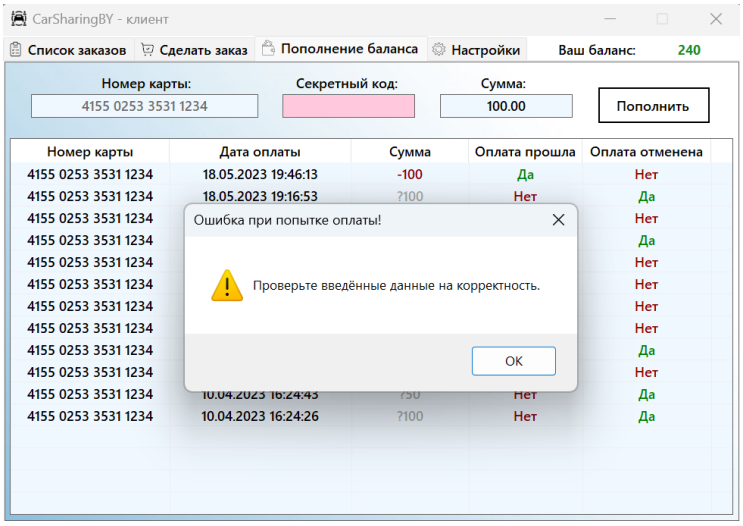

Рисунок 4.4 – Проверка на ошибки во вкладке «Пополнение баланса» клиента

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 спи-

саний», «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).

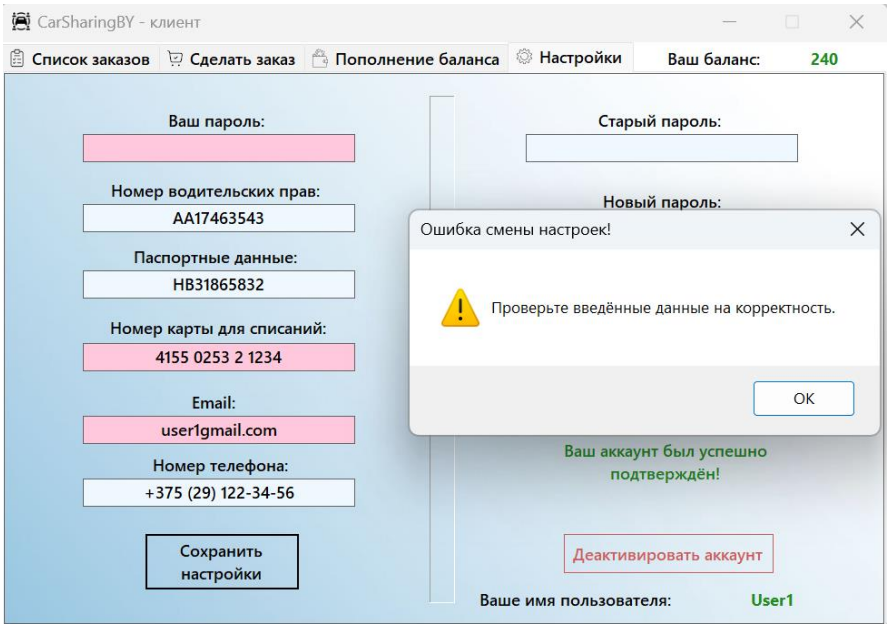

Рисунок 4.5 – Проверка на ошибки во вкладке «Настройки» клиента

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).

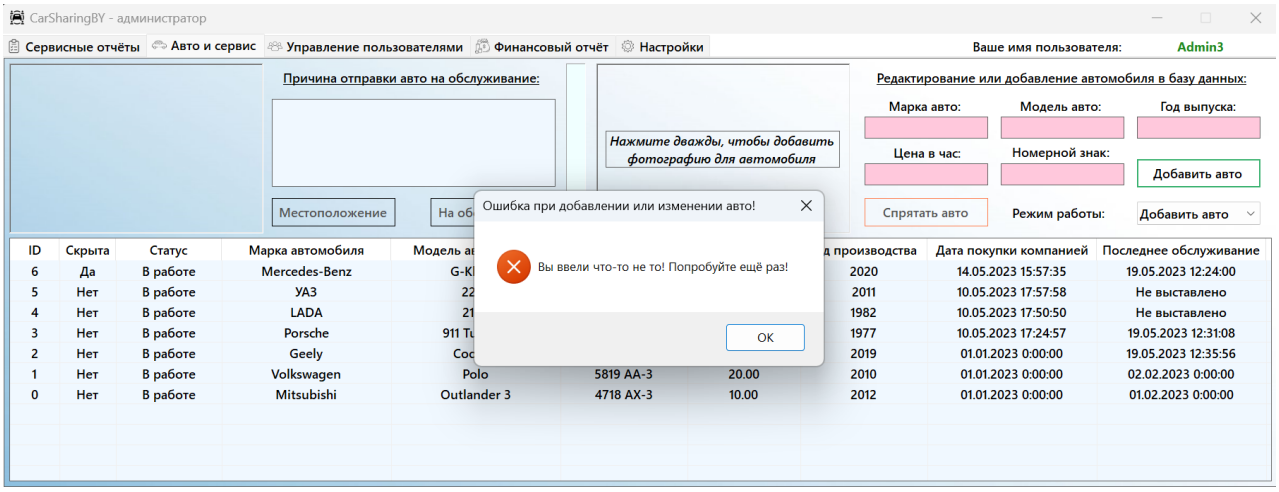

Рисунок 4.6 – Проверка на ошибки во вкладке «Авто и сервис» администратора

4.2 Модульное тестирование

Модульное тестирование является неотъемлемой частью процесса разработки программного обеспечения, особенно в контексте обеспечения надёжности и стабильности работы с базами данных. В данном курсовом проекте было особое внимание уделено модульным тестам, что позволяет проверять корректность работы каждого элемента системы независимо от остальных компонентов и гарантировать, что любые изменения в коде не нарушат существующую функциональность.

Использование *NUnit* для модульных тестов: для реализации модульных тестов был выбран фреймворк *NUnit*, который предоставляет мощные средства для написания и выполнения тестов, включая ассерты, аннотации для настройки тестового окружения и множество других полезных инструментов.

Важным аспектом в разработке программного обеспечения является понимание различных подходов к тестированию, в том числе использование так называемых *Mock*-объектов. *Mock*-объект – это вид объекта в программировании, который имитирует поведение реальных объектов в контролируемой среде. Эти объекты обычно используются для имитации поведения сложных, реальных (часто внешних) систем, что позволяет тестировать отдельные части программы в изоляции от остальной системы.

Однако, в контексте данного проекта, было принято решение не использовать *Mock*-объекты. Причина этого заключается в том, что *Mock*-объекты, хотя и полезны для изоляции тестов и проверки определенных сценариев взаимодействия компонентов, на самом деле не тестируют реальное поведение системы. Вместо этого, они создают симуляцию, которая может не полностью отражать реальные условия и взаимодействия в рабочей среде.

Использование *Mock*-объектов может быть ценным при тестировании интерфейсов взаимодействия с внешними сервисами или сложными системами, где тестирование с реальными компонентами может быть непрактичным или слишком затратным. Однако, в данном проекте основной акцент был сделан на тестировании реального взаимодействия компонентов системы, чтобы обеспечить максимально точное воспроизведение реальных условий работы приложения.

Таким образом, предпочтение было отдано реальному, а не симулированному тестированию, чтобы убедиться, что все компоненты системы работают корректно в реальных условиях. Этот подход обеспечивает более точное и достоверное тестирование, что критически важно для надежности и стабильности конечного продукта.

Структура проекта *unit*-тестов: как видно из структуры проекта *unit*-тестов на предоставленном ниже рисунке 4.7, тесты организованы по типам

хранилищ данных – *CSV*, *Mongo* и *SQL*. Для каждого репозитория, работающего с определенным типом базы данных, написаны тесты, проверяющие функциональность всех основных *CRUD*-операций: добавление, чтение, обновление и удаление данных.

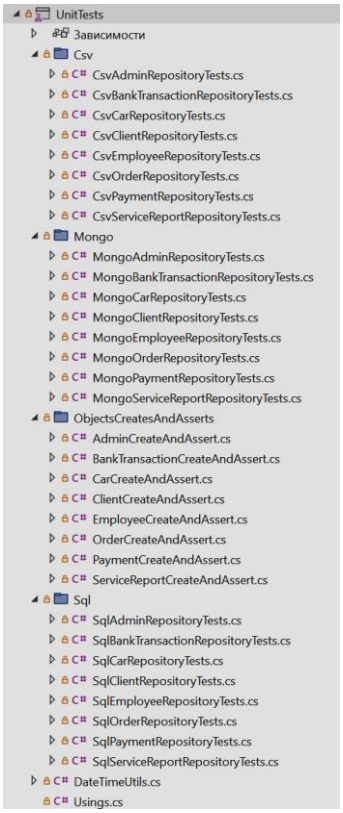

Рисунок 4.7 – Структура проекта модульного тестирования

На каждый тип репозитория написано по 14 тестов для каждого из 8 объектов, что обусловлено необходимостью всесторонней проверки каждой из *CRUD*-функций. Эти тесты включают проверку стандартных сценариев, таких как успешное добавление или получение данных, а также обработку краевых случаев и возможных ошибок, например, попытку добавления или получения несуществующих записей.

Ниже представлен детализированный анализ модульных тестов.

Add - добавление сущности:

– цель: убедиться, что добавление новой сущности в репозиторий происходит без ошибок и корректно назначает уникальный идентификатор;

– процесс: создаем новую сущность и добавляем её в репозиторий. Проверяем, что операция не вызывает исключений;

– валидация: убеждаемся, что возвращаемый идентификатор уникален в контексте репозитория.

Add - обработка дубликатов:

– цель: проверить, что система корректно обрабатывает добавление дубликатов, изменяя *ID* для новых записей для предотвращения конфликтов;

– процесс: добавляем сущность в репозиторий, затем добавляем другую сущность с тем же *ID*. Проверяем, что вторая сущность добавлена с новым, уникальным *ID*;

– валидация: подтверждаем, что *ID* второй сущности уникален и отличается от *ID* первой сущности.

Add - исключения при добавлении:

– цель: удостовериться, что система корректно обрабатывает попытку добавления нулевой ссылки (*null*);

– процесс: Пытаемся добавить *null* в репозиторий;

– валидация: Проверяем, что система вызывает *NullReferenceException*.

GetAll - возврат всех записей:

– цель: подтвердить, что метод *GetAll* возвращает полный список добавленных записей;

– процесс: добавляем несколько сущностей в репозиторий, затем вызываем метод *GetAll*;

– валидация: убедиться, что возвращаемый список содержит все добавленные сущности.

GetAll - проверка пустого списка:

– цель: проверить, что метод *GetAll* возвращает пустой список, если в базе данных нет записей;

– процесс: вызываем *GetAll* на пустом репозитории;

– валидация: убедиться, что возвращаемый список пуст.

Get - поиск по *ID*:

– цель: удостовериться, что метод *Get* корректно извлекает сущность по её *ID*;

– процесс: добавляем сущность с известным *ID*, затем извлекаем её, используя метод *Get*;

– валидация: проверяем, что извлеченная сущность соответствует добавленной.

Get - обработка несуществующих записей:

– цель: тестировать, что метод *Get* возвращает *null* для несуществующих *ID*;

– процесс: вызываем *Get* с несуществующим *ID*;

– валидация: подтверждаем, что результат равен *null*.

Update - обновление существующих данных:

– цель: проверить, что метод *Update* успешно обновляет данные существующих записей;

– процесс: добавляем сущность, затем обновляем её данные и вызываем *Update*;

– валидация: извлекаем обновленную сущность и проверяем, что её данные соответствуют сделанным изменениям.

Update - непрерывность операции обновления:

– цель: оценить, что метод *Update* не вызывает исключений при обновлении данных, даже если целевая запись отсутствует;

– процесс: пытаемся обновить несуществующую сущность;

– валидация: убедиться, что метод не вызывает исключений.

Delete - удаление данных:

– цель: подтвердить, что метод *Delete* корректно удаляет существующие записи;

– процесс: добавляем сущность, затем удаляем её, используя метод *Delete*;

– валидация: проверяем, что сущность больше не присутствует в репозитории.

Delete - стабильность при отсутствии данных:

– цель: убедиться, что метод *Delete* не вызывает исключения при попытке удаления записей в пустой базе данных;

– процесс: пытаемся удалить несуществующую сущность из пустого репозитория;

– валидация: подтверждаем, что метод не вызывает исключений.

Общая проверка работоспособности:

– цель: провести общий тест для оценки способности всех *CRUD* операций работать совместно и корректно обрабатывать различные сценарии использования;

– процесс: выполняем серию операций *CRUD* в различных комбинациях и сценариях;

– валидация: убедиться, что все операции работают корректно и взаимодействуют друг с другом без ошибок.

Этот расширенный анализ дает более глубокое понимание каждого модульного теста, его цели и методологии, что обеспечивает полное покрытие всех аспектов функциональности репозитория и уверенность в его надежности.

При работе с датами и временем в модульных тестах, важно учитывать возможные неточности из-за внутренней работы системы и задержек в выполнении операций. В данном проекте тестирование функциональности, связанной с временными метками, осуществляется с учетом допустимого отклонения в три секунды от текущего времени (*DateTime.UtcNow*). Такой подход позволяет учесть и компенсировать потенциальные различия во времени выполнения те-

стов и времени на сервере, а также обеспечить гибкость тестовых сценариев без потери их достоверности.

Эта маржа в три секунды выбрана как компромисс между точностью и практичностью, позволяя избежать ложных сбоев тестов из-за мгновенных изменений во времени выполнения кода. Такое решение особенно актуально при работе с базами данных, где запись или извлечение дат может включать небольшие задержки, влияющие на точность сравнения. Тесты, следовательно, проверяют, что сохраненные и извлеченные временные метки попадают в ожидаемый интервал времени, что подтверждает их правильность и актуальность.

Таким образом, модульное тестирование каждой функции *CRUD* для всех типов баз данных является критически важным для обеспечения надежности и устойчивости приложения. Это позволяет своевременно выявлять ошибки и уверенно вносить изменения в систему, снижая риск дефектов при развертывании или масштабировании приложения. В комплексе с умелым использованием паттерна «стратегия» и обширным использованием *unit*-тестов, проект демонстрирует высокую степень готовности к продуктивной эксплуатации и последующему развитию.

На рисунке 4.8 приведён результат прохождения всех модульных тестов. Полная работоспособность всех модульных тестов, охватывающих ключевые операции *CRUD* для разнообразных типов хранилищ данных – *CSV*, *MongoDB* и *SQL* – подтверждает надежность и стабильность всего проекта. Это свидетельствует о том, что приложение было тщательно проверено и готово к реализации в различных средах, обеспечивая устойчивую работу всех его компонентов. Каждый набор тестов, соответствующий определенному типу репозитория, был успешно пройден, что демонстрирует эффективность и правильность реализации паттерна «стратегия» и его применения в контексте взаимодействия с базами данных.

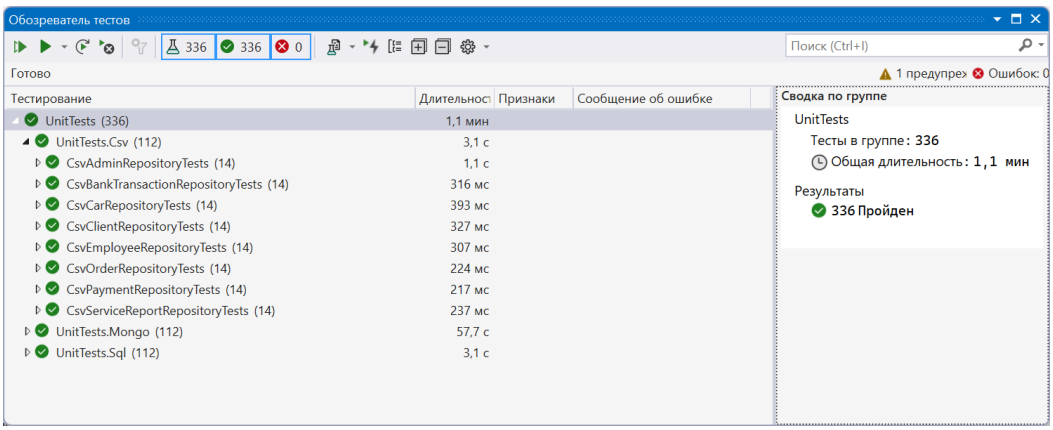

Рисунок 4.8 – Результат полного модульного тестирования проекта

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-е изд. – СПб.: Питер, 2016 – 896 с.

3. Албахари, Дж. C# 7.0. Карманный справочник. – СПб.: ООО “Альфакнига”, 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 с.