Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

Кафедра экономической информатики

|  |  |
| --- | --- |
|  | *К защите допустить*: |
|  | Руководитель курсового проекта  Ассистент кафедры ЭИ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.О. Кабак |

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

курсового проекта

на тему

**Программное средство для дистанционного управления опрыскивателем растений**

БГУИР КП 6-05-0611-01 25 ПЗ

|  |  |  |
| --- | --- | --- |
| Студент |  | А. В. Еднач |
| Руководитель |  | Н. О. Кабак |

Минск 2025

**Реферат**

БГУИР КР 6-05-0611-01 102 ПЗ

ТЕМА: Программное средство для управления обработкой растений / А.В. Еднач. – Минск : БГУИР, 2025, – п.з. – 141 с., рисунков – 43, источников – 17, приложений – 4.

*Ключевые слова:* автоматизация, управление дроном, обработка растений, клиент-сервер.

*Объект исследования:* управление устройством обработки растений.

*Предмет исследования:* программное средство для управления устройством обработки растений.

*Цель* *курсовой работы*: автоматизация расчёта состава смеси для обработки растений.

*Методология проведения работы*: в процессе разработки системы использованы методы анализа предметной области, классификации данных, функционального моделирования с применением нотации IDEF0, информационного моделирования с использованием IDEF1X, а также объектно-ориентированного моделирования с помощью UML-диаграмм. Реализованы программные модули и графический интерфейс.

*Результаты работы*: выполнена постановка задачи и определены методы ее решения; разработана функциональная модель системы с использованием IDEF0; построена информационная модель базы данных в нотации IDEF1X, соответствующая третьей нормальной форме; созданы UML-диаграммы для моделирования системы; реализованы серверная и клиентская части программного средства на языке C# с использованием технологий .Net, Unity; разработано руководство пользователя; проведено тестирование, подтвердившее соответствие системы функциональным требованиям. Программный продукт обеспечивает автоматизацию учёта препаратов, расчёта зон обработки, соотношение и количество смеси.

*Область применения результатов*: разработанное программное средство может быть применено в парниках, колхозах. Оно способствует упрощению расчёта смеси и снижению вероятности ошибки при расчёте и учёте дозы. Программное средство полностью отвечает функциональным требованиям.

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 5](#_Toc199794083)

[ВВЕДЕНИЕ 7](#_Toc199794084)

[1 ОПИСАНИЕ СИСТЕМЫ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ 9](#_Toc199794085)

[1.1 Понятие дистанционного управления опрыскивателем растений 9](#_Toc199794086)

[1.2 Виды систем дистанционного управления опрыскивателем растений и аналогичные решения 9](#_Toc199794087)

[1.3 Вывод по разделу «Виды систем дистанционного управления опрыскивателем растений» 12](#_Toc199794088)

[2 Постановка задачи для разработки программного средства дистанционного управления опрыскивателем растений 13](#_Toc199794089)

[3 Функционально моделирование программного средства для дистанционного управления опрыскивателем 15](#_Toc199794090)

[4 ИНФОРМАЦИОННАЯ МОДЕЛЬ ПРОГРАММНОГО СРЕДСТВА ДЛЯ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ 21](#_Toc199794091)

[6 ОПИСАНИЕ АЛГОРИТМОВ, РЕАЛИЗУЮЩИХ БИЗНЕС-ЛОГИКУ СЕРВЕРНОЙ ЧАСТИ ПРОГРАММНОГО СРЕДСТВА ДЛЯ УПРАВЛЕНИЯ АКЦИЯМИ ИНВЕСТИЦИОННОГО ФОНДА 24](#_Toc199794092)

[6.1 Схема жизненного цикла клиент-серверного взаимодействия 25](#_Toc199794093)

[6.2 Схема процесса авторизации пользователей 27](#_Toc199794094)

[6.3 Схема процесса расчёта инструкций лечения растения 29](#_Toc199794095)

[7 МОДЕЛИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА ДЛЯ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ 31](#_Toc199794096)

[7.1 Моделирование функциональности системы с помощью диаграммы вариантов использования 32](#_Toc199794097)

[7.2 Описание структуры системы на основе диаграммы классов 35](#_Toc199794098)

[7.3 Анализ поведения системы с использованием диаграммы последовательностей 41](#_Toc199794099)

[7.4 Моделирование жизненного цикла объектов с помощью диаграммы состояний 42](#_Toc199794100)

[7.5 Моделирование физического размещения системы с помощью диаграммы развертывания 43](#_Toc199794101)

[7.6 Вывод по разделу «Моделирование программного средства для управления акциями инвестиционного фонда с использованием UML» 44](#_Toc199794102)

[8 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 46](#_Toc199794103)

[8.1 Подготовка к эксплуатации 47](#_Toc199794104)

[8.2 Руководство для администратора 50](#_Toc199794105)

[8.3 Руководство для пользователя 58](#_Toc199794106)

[8.4 Вывод по разделу «Руководство пользователя» 62](#_Toc199794107)

[9 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ РАЗРАБОТАННОЙ СИСТЕМЫ 63](#_Toc199794108)

[ЗАКЛЮЧЕНИЕ 74](#_Toc199794109)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 76](#_Toc199794110)

[ПРИЛОЖЕНИЕ А (обязательное) Отчет о проверке на заимствования в системе «Антиплагиат» 77](#_Toc199794111)

[ПРИЛОЖЕНИЕ Б (обязательное) Листинг алгоритмов, реализующих бизнес-логику 78](#_Toc199794112)

[ПРИЛОЖЕНИЕ В (обязательное) Листинг основных элементов программы 91](#_Toc199794113)

[ПРИЛОЖЕНИЕ Г (обязательное) Листинг скрипта генерации базы данных 110](#_Toc199794114)

# ВВЕДЕНИЕ

Современные информационные технологии кардинально изменили подход к агропромышленному производству, включая управление тепличными хозяйствами. Автоматизированные системы позволяют оптимизировать процессы лечения растений, обеспечивая оперативную диагностику заболеваний, точный расчёт дозировок и визуализацию зон обработки. Такие решения интегрируются с базами агрономических знаний, предоставляя операторам и агрономам инструменты для принятия научно обоснованных решений. Внедрение программных средств для подбора терапии растений становится критически важным шагом для повышения урожайности и экологической безопасности тепличных комплексов.

Управление здоровьем растений в условиях закрытого грунта является ключевым элементом деятельности современных агропредприятий. Этот процесс требует высокой точности, учёта множества факторов: видовых особенностей культур, типа патогена, фитосовместимости препаратов и экологических ограничений. В условиях развития точного земледелия автоматизация подбора лечебных схем позволяет сократить потери урожая, повысить эффективность обработок и обеспечить оперативное реагирование на фитосанитарные угрозы.

Актуальность темы курсового проекта обусловлена необходимостью внедрения интеллектуальных систем для лечения растений в теплицах. Традиционные подходы, основанные на ручных расчётах и универсальных рекомендациях, не способны учитывать специфику конкретного тепличного комплекса, динамику развития заболеваний и современные экологические требования. Автоматизированное программное средство обеспечивает:

- Персонализированный подбор схем лечения

- Точный расчёт дозировок с учётом концентрации активных веществ

Объектом исследования является процесс лечения растений в промышленных теплицах, включающий диагностику заболеваний, подбор препаратов, расчёт рабочих растворов и планирование обработок.

Предметом исследования выступает программное средство для автоматизации подбора терапии растений, разработанное с использованием современных информационных технологий.

Целью курсового проекта является создание программного комплекса для оптимизации лечебных мероприятий в тепличных хозяйствах, обеспечивающего повышение эффективности обработок и снижение расхода препаратов.

Для достижения цели поставлены задачи:

1. Провести анализ предметной области и существующих решений в области автоматизации тепличных хозяйств
2. Разработать реляционную модель данных для хранения параметров растений, заболеваний и препаратов
3. Спроектировать архитектуру клиент-серверной системы (Unity/.NET/PostgreSQL)
4. Реализовать алгоритмы:

- Подбора препаратов по двойной совместимости (растение-заболевание-препарат)

- Расчёта зон обработки методом кластеризации точек поражения

- Генерации рецептур рабочих растворов

1. Разработать интуитивный пользовательский интерфейс с пошаговым сценарием работы
2. Обеспечить интеграцию с геоинформационными системами тепличных комплексов
3. Провести тестирование на соответствие агротехнологическим требованиям

Внедрение разработанного программного средства позволит:

- Автоматизировать процесс подбора терапии растений;

- Снизить риски фитотоксичности;

- Повысить точность обработок;

- Упростить документирование лечебных мероприятий;

- Оптимизировать расход препаратов.

Курсовая работа выполнена самостоятельно, проверена в системе «Антиплагиат». Процент оригинальности составляет 89,23%. Цитирования обозначены ссылками на публикации, указанными в «Списке использованных источников». Скриншот приведен в приложении А (рисунок А.1).

# 1 ОПИСАНИЕ СИСТЕМЫ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ

# 1.1 Понятие дистанционного управления опрыскивателем растений

Дистанционное управление опрыскивателем растений – процесс отправки инструкций по обработке растения на дрон с опрыскивателем. Для того, чтобы повысить точность операций и предотвратить гибель растения от неправильно подобранного лечения, в программу включен механизм хранения и обработки данных о растениях, заболеваниях и препаратах.

Основными составляющими дистанционного управления опрыскивателем растений являются:

1 Формирование баз данных растений, болезней, препаратов.

2 Формирование баз совместимости растений, болезней, препаратов.

3 Выбор необходимого лечения в соответствии с совместимостью растения с препаратами.

Смесь, которой обрабатывают растение определяется совокупностью нескольких факторов, таких как: вид растения, вид болезни, совместимость растения и лекарства, совместимость болезни и лекарства, ударная доза лекарства растения.

Дистанционное управление опрыскивателем растений играет ключевую роль в автоматизации обработки растений в парнике:

1 Упрощение работы – автоматизация значительно упростит работу работников сельскохозяйственных предприятий.

2 Соответствие нормативам .

3 Прозрачность управления – инвесторы получают детальную информацию о составе портфеля, доходности и рисках.

4 Оптимизацию инвестиционных процессов – автоматизированные системы ускоряют анализ данных, проведение сделок и формирование отчетности.

В современных организациях для выполнения расчетов используются специализированные программные решения, которые позволяют автоматизировать процесс начислений, уменьшить вероятность ошибок, упростить бухгалтерский учет и обеспечить соответствие законодательным требованиям.

# 1.2 Виды систем дистанционного управления опрыскивателем растений и аналогичные решения

Системы дистанционного управления опрыскивателем растений можно классифицировать по уровню автоматизации, архитектуре управления и способу взаимодействия оператора с дроном. Выбор конкретного подхода зависит от требований к точности обработки, масштаба хозяйства и наличия коммуникационной инфраструктуры.

1. Ручное дистанционное управление. Оператор напрямую задаёт траекторию полёта и параметры распыления (скорость, расход смеси) с пульта или планшета. Такой режим подходит для небольших площадей или для “точечных” обработок, когда требуется мгновенная корректировка в реальном времени.
2. Полуавтономные системы. Полёт по заранее проложенному маршруту (GPS-треке), при этом оператор может вмешиваться и менять параметры на лету. Позволяет сочетать гибкость ручного управления с быстродействием автоматизированного.
3. Полностью автономные решения. Планирование полёта и дозировки смеси выполняется заранее на основании цифровой карты поля и атласов растений (привязка по геокоординатам), а дрон самостоятельно выполняет цикл «взлёт – распыление – посадка». Требует минимального участия оператора и обеспечивает стабильность повторяемости операций.
4. Облачные платформы с клиент–серверной архитектурой. Клиентская часть (мобильное или веб-приложение) служит для подготовки миссии, загрузки карт и данных о растениях; серверная часть хранит базы данных, рассчитывает оптимальные маршруты и обеспечивает логи процедур. Позволяет объединять в единое поле зрения данные с нескольких дронов и операторов.
5. Интегрированные системы “умного” хозяйства. Являются частью более крупной экосистемы: связываются с системами мониторинга почвы, погодными датчиками и системами полива для корреляции данных и адаптивного управления.

Ниже приведены примеры распространённых программно-аппаратных комплексов для дистанционного управления опрыскивателями.



Рисунок 1.2.1 – Логотип системы DroneDeploy

Платформа позволяет загружать спутниковые ортофотокарты, автоматически строить маршруты съёмки и распыления, а также анализировать результаты обработки на облачном сервере.



Рисунок 1.2.2 – Логотип программы DJI Ground Station Pro

Предоставляет инструменты для планирования полётов дронов DJI, включая задания по автоматической залётке, настройке параметров камеры и распылителя, а также передачу телеметрии в реальном времени.



**Рисунок 1.2.3** – Логотип **John Deere Operations Center**

Интегрированная платформа для управления техникой John Deere, в том числе навесными опрыскивателями, с возможностью объединения данных о посевах, урожайности и расходе средств защиты растений.

Основные функции современных систем дистанционного управления опрыскивателем:

1. Управление миссией – создание, редактирование и сохранение маршрутов обработки полей с учётом ширины захвата и зоны перекрытия.
2. Расчёт дозировки смеси – автоматический подбор концентрации препаратов на основании характеристик растения, типа обработки и заданной площади.
3. Мониторинг в реальном времени – отображение местоположения дрона, оставшегося запаса смеси и состояния аккумулятора.
4. Хранение и анализ данных – ведение базы профильных растений, препаратов, результатов обработок и отчётов для последующего аудита.
5. Интеграция с другими сервисами – передача данных в системы учёта и ERP-решения хозяйства.

# 1.3 Вывод по разделу «Виды систем дистанционного управления опрыскивателем растений»

Современные решения для дистанционного управления обработкой растений сочетают несколько уровней автоматизации: от ручного управления до полностью автономных облачных платформ. Благодаря клиент-серверной архитектуре и модульному подходу удаётся:

* Унифицировать работу разных операторов и дронов в едином интерфейсе;
* Повысить точность за счёт расчёта дозировки и оптимизации маршрутов;
* Сократить трудозатраты и риск ошибок при планировании и исполнении миссии;
* Обеспечить прозрачность учёта обработок и требования к нормативам СЗР.

Интеграция с системами “умного” хозяйства и ERP позволяет не только автоматизировать расчёт составов смесей, но и выстраивать комплексное управление агротехнологиями: от мониторинга почвы до оценки экономической эффективности обработок. Такая синергия компонентов обеспечивает повышение урожайности при снижении затрат и рисков.

# 2 Постановка задачи для разработки программного средства дистанционного управления опрыскивателем растений

Цель проекта — создать клиент-серверную систему, позволяющую оператору удалённо планировать и выполнять обработку сельскохозяйственных угодий с помощью опрыскивателя, автоматически рассчитывая рабочие зоны, дозировки препаратов и учитывая специфику растений, болезней и медикаментозных препаратов.

Основные задачи разработки:

1 Многопоточный сервер (C# /.NET), который обрабатывает TCP-запросы от множества Unity-клиентов одновременно, гарантируя низкую задержку и стабильность работы при параллельном взаимодействии нескольких операторов.

2 Unity-клиентское приложение (C# в Unity), обеспечивающее интуитивно понятный интерфейс для:

* аутентификации пользователя (администратора или обычного оператора),
* выбора участка (список растений, их заболеваний и доступных медикаментов),
* формирования маршрута опрыскивателя и задания параметров обработки (радий действия, дозировка),
* визуализации расчётных зон и объёмов жидкости,
* мониторинга состояния опрыскивателя в реальном времени.

3 Настройка взаимодействия с реляционной базой данных для хранения информации об акциях, пользователях, компаниях, портфеле и балансе.

4 Реализация бизнес-логики на стороне сервера для обработки операций с акциями, управления пользователями, компаниями и генерации файловых отчетов.

5 Надёжное хранение и восстановление данных (PostgreSQL + Npgsql), включая:

* регулярное резервирование базы (например, Docker-контейнер с PostgreSQL),
* «требуемые» внешние ключи для обеспечения целостности (3NF — каждая таблица содержит атомарные поля, нет избыточных данных, все связи реализованы через внешние ключи).

6 Внедрение системы авторизации с разделением прав доступа для администраторов и пользователей.

В ходе разработки программного средства необходимо решить следующие задачи:

1. Реализовать серверную часть на C# с использованием платформы .NET 6 (сборка .NET 6.0), располагающуюся в среде JetBrains Rider. Сервер должен функционировать как многопоточный TCP‑сервер и обеспечивать одновременное обслуживание нескольких клиентских подключений.
2. Разработать клиентскую часть на движке Unity (язык C#), которая взаимодействует с сервером через TCP‑сокеты. Клиентское приложение должно предоставлять пользователю графический интерфейс, созданный средствами Unity UI (графический интерфейс Unity), и обеспечивать отправку команд на сервер и приём ответов.
3. Обеспечить хранение и управление данными в реляционной базе данных PostgreSQL. Подключение к базе данных осуществляется с помощью библиотеки Npgsql. В базе данных хранятся необходимые справочные данные, параметры опрыскивания, учётные записи пользователей и другая информация.
4. Определить протокол обмена командами между клиентом и сервером. Передача команд осуществляется в текстовом формате с использованием разделителей. Сервер и клиент должны обеспечивать ручную обработку (парсинг) таких текстовых сообщений и выполнение соответствующих операций.
5. Внедрить функциональность, предусмотренную предметной областью. К основным функциям системы относятся расчёт зон опрыскивания (например, вычисление площади или схемы обработки в соответствии с входными параметрами), обновление справочников (списков химикатов, параметров опрыскивания и прочих служебных данных) и отображение соответствующей информации в пользовательском интерфейсе клиента. Весь необходимый функционал реализуется в соответствии с имеющимся исходным кодом проекта.
6. Соблюсти остальные требования и замечания, предъявленные руководителем проекта. При разработке программного средства должны быть учтены все ранее указанные особенности используемых технологий и библиотек, а также замечания, представленные в исходном задании и документации (например, использование указанных сред разработки и инструментов, ручная обработка текстовых команд и т.д.).

Таким образом, в рамках данной курсовой работы предстоит разработать полнофункциональный клиент‑серверный программный комплекс, удовлетворяющий указанным требованиям и обеспечивающий надёжное дистанционное управление опрыскивателем растений с соответствующим пользовательским интерфейсом и базой данных.

# 3 Функционально моделирование программного средства для дистанционного управления опрыскивателем

Современные информационные технологии играют ключевую роль в аграрной сфере, проникая во все аспекты управления полевыми работами. В условиях стремительного развития «умного» земледелия фермерские хозяйства и агропредприятия всё чаще сталкиваются с необходимостью автоматизации процессов обработки полей. Однако многие хозяйства, особенно небольшие, продолжают полагаться на устаревшие подходы, такие как ручное планирование маршрутов и использование разрозненных инструментов для учёта операций, что приводит к низкой эффективности, ошибкам и отсутствию оперативного доступа к данным о состоянии полей и технике.

Дистанционное управление опрыскивателем растений — это комплексный процесс, который включает получение от оператора параметров обработки участка, расчёт оптимального маршрута и зон опрыскивания, контроль над выполнением операций, сбор телеметрических данных от техники и предоставление аналитической информации о расходе жидкости и охваченных площадях. Основные задачи системы управления опрыскивателем включают проверку корректности входных команд, расчёт рабочих зон с учётом плотности насаждений и рельефа, построение маршрутов с учётом радиуса обработки и объёма жидкости, а также формирование отчётов для оценки эффективности и экономии ресурсов. Без автоматизации эти процессы становятся трудоёмкими, повышают риск ошибок и затрудняют своевременное получение данных о проведённых работах.

В современных условиях ручной подход к управлению опрыскивателем становится всё менее приемлемым. Развитие технологий позволяет агропредприятиям автоматизировать ключевые этапы обработки полей, что ускоряет операции, повышает точность внесения средств защиты растений и обеспечивает доступность аналитики для операторов и агрономов. Автоматизированные системы дистанционного управления опрыскивателем делают процесс обработки полей более прозрачным, эффективным и удобным для всех участников агротехнологического цикла.

Ниже представлен процесс определения лечения растения IDEF0 [9]. Данная модель описывает процесс до выведения рассчитанных инструкций пользователю.

Процесс составления инструкций к опрыскивателю представлен в виде 4 уровней декомпозиции. Верхний уровень включает один блок: «Выбрать лечение растения».

Второй уровень представляет собой декомпозицию блоков «Подобрать список лекарственных препаратов», «Определить препарат», «Рассчитать область обработки». Декомпозиция блока «Подобрать список лекарственных препаратов» содержит три блока: «Произвести выборку болезней по растениям», «Произвести выборку лекарств по болезням», «Произвести выборку лекарств по растениям». Декомпозиция блока «Определить препарат» содержит два блока: «Выбрать необходимое лекарство», «Выбрать дозировку». Декомпозиция блока «Рассчитать область обработки» содержит три блока: «Перевести координаты поражения», «Рассчитать радиус обработки», «Рассчитать использование препарата.

Третий уровень содержит декомпозицию блока «Рассчитать использование препарата», которая содержит два блока: «Рассчитать площади обработки» , «Рассчитать литраж по ударной дозе».

Четвёртый уровень содержит декомпозицию блока «Рассчитать литраж по ударной дозе», которая содержит два блока: «Сложить все площади» , «Рассчитать литраж по общему объёму площади и ударной дозе».

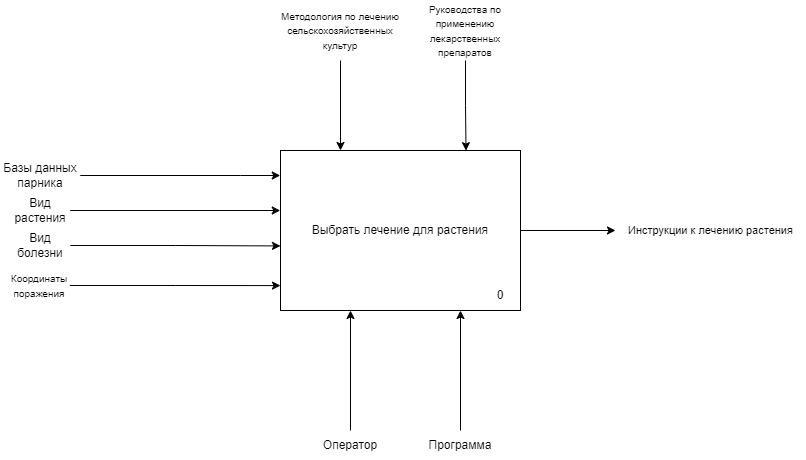


Рисунок 3.1 – Верхний уровень модели

В данной модели процесс составления инструкций к лечению растения. Управляющими факторами выступают методология по лечению сельскохозяйственных культур и руководства по применению лекарственных препаратов. Эти элементы влияют на процесс, но не являются его частью. Входными данными для процесса служат базы данных парника, вид растения, вид болезни его поразившей и координаты поражения. На выходе процесса формируется готовая инструкция по обработке растения, включающая зоны поражения, количество смеси, её концентрацию, данные о препарате, растении и болезни.

К механизмам, обеспечивающим выполнение процесса, относятся клиентское приложение (Unity), серверная часть (.Net), база данных (PostgreSQL). Клиентское приложение предоставляет интерфейс для взаимодействия с пользователем, сервер обрабатывает запросы и выполняет бизнес-логику, база данных хранит информацию о лекарствах, болезнях препаратах и их совместимостях.

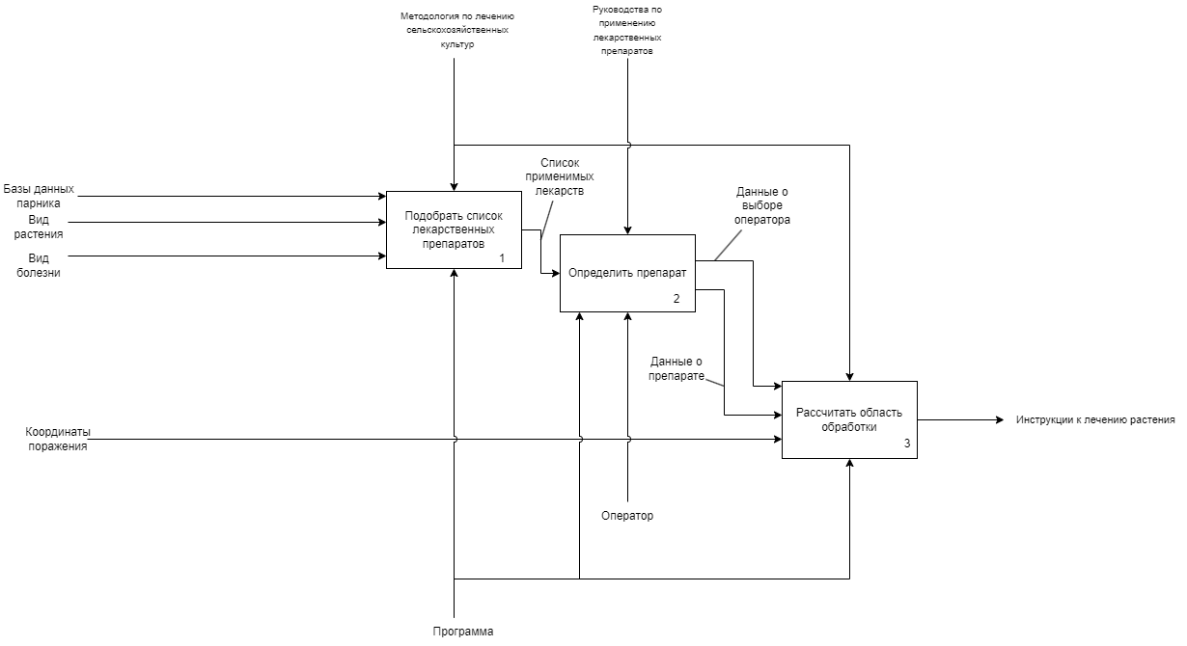


Рисунок 3.2 – Декомпозиция процесса «Выбрать лечение растения»

Процесс «Выбрать лечение для растения» разделен на три ключевых подпроцесса, каждый из которых выполняет свою специфическую задачу в рамках общей операции.

Первый подпроцесс – подобрать список лекарственных препаратов. На этом этапе система сортирует данные о лекарствах в соответствии с растением и болезнью.

Второй подпроцесс – определить препарат. На данном этапе оператор выбирает препарат, который будет использован для лечения растения, а также дозировку.

Третий подпроцесс – рассчитать область обработки. На данном этапе программа рассчитывает область обработки, а также итоговый литраж смеси для обработки.

На выходе процесса формируются инструкции к лечению растения.

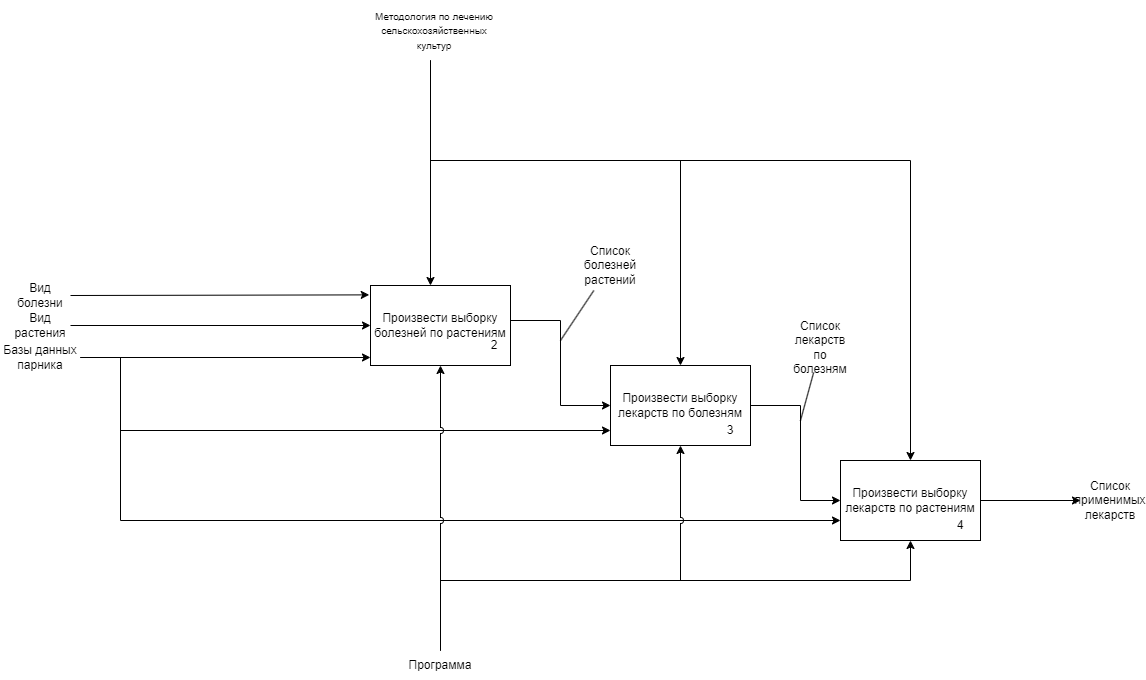


Рисунок 3.3 – Декомпозиция процесса «Подобрать список лекарственных препаратов»

Подпроцесс «Подобрать список лекарственных препаратов» включает три этапа, направленных на выборку подходящих лекарств из базы данных.

Первый этап – произвести выборку болезней по растениям. Возвращает список болезней, подходящих к растению.

Второй этап – произвести выборку лекарств по болезням. Возвращает список лекарств по болезням.

Третий этап – произвести выборку лекарств по растениям. Возвращает список лекарств по растениям.

Итогом выполнения подпроцесса становится список применимых к растению лекарств для лечения данной болезни.

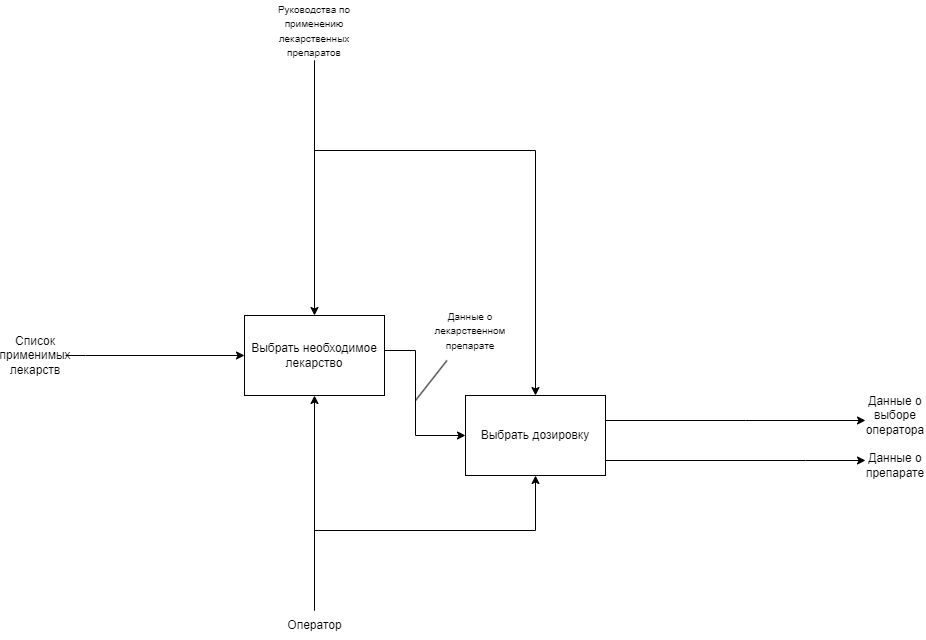


Рисунок 3.4 – Декомпозиция процесса «Определить препарат»

Процесс определения препарата состоит из двух этапов, позволяющих оператору выбрать из списка подходящий препарат для лечения и его дозировку.

Первым этапом является выбор препарата из списка подходящих.

Вторым этапом является выбор дозировки для рассчёта смеси препарата.

Итогом становится данные о препарате и выбранной дозировке.

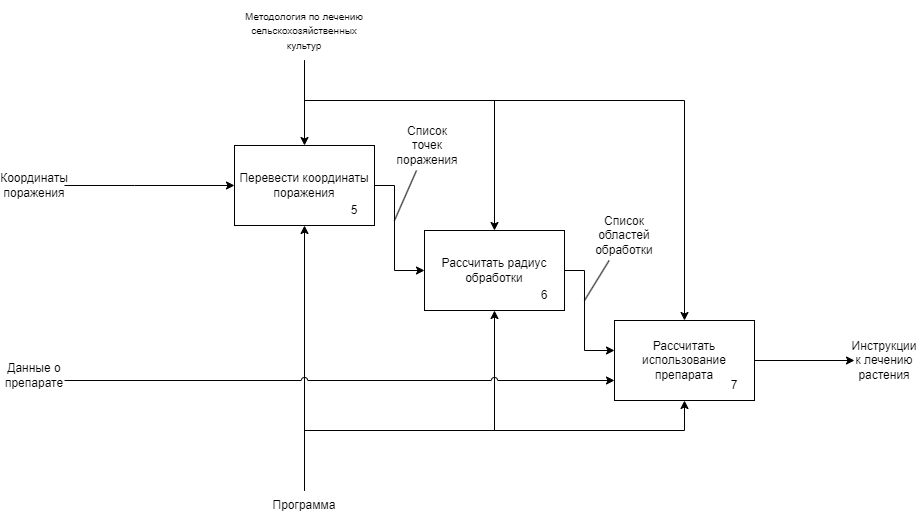


Рисунок 3.5 – Декомпозиция процесса «Рассчитать область обработки»

Процесс рассчёта области обработки рассчитывает зоны обработки и литраж смеси для лечения.

Механизмы поддержки: Операции выполняются программой сервера, обеспечивающей обработку и получение данных из базы данных.

В результате формируются инструкции к лечению растения.

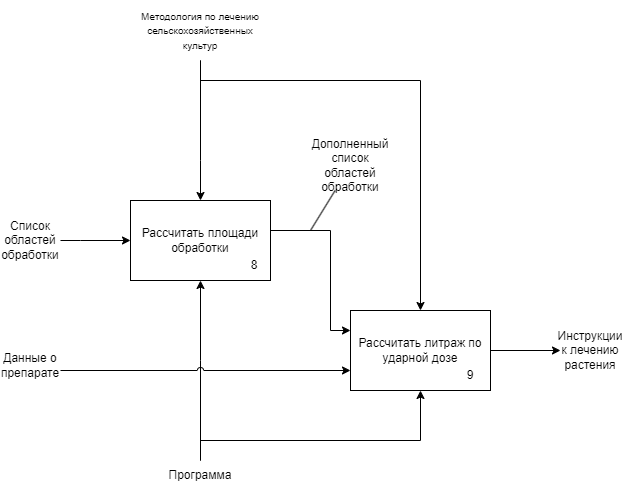


Рисунок 3.6 – Декомпозиция процесса «Рассчитать использование препарата»

Процесс расчёта использования препарата состоит из двух этапов:

1 Рассчитать площади обработки.

2 Рассчитать литраж по ударной дозе.

В результате сформированы инструкции к лечению растения, включающие области обработки и данные о препарате.

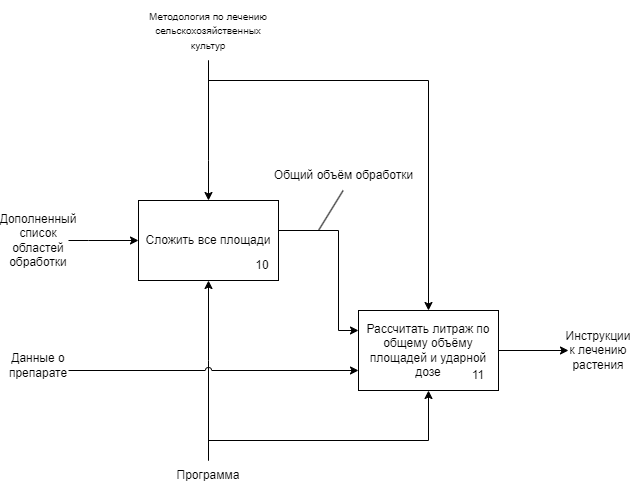


Рисунок 3.7 – Декомпозиция процесса «Рассчитать литраж по ударной дозе»

Процесс расчёта литража препарата состоит из двух этапов:

1 Сложить все площади.

2 Рассчитать литраж по общему объёму площадей и ударной дозе.

В результате сформированы инструкции к лечению растения, включающие области обработки и данные о препарате.

Все описанные процессы, от подбора списка лекарственных препаратов до расчёта областей обработки, образуют последовательную систему этапов, обеспечивающую простоту, надежность, эффективность составления инструкций лечения. Каждый этап выполняет свою специфическую задачу, но их взаимосвязь и строгая последовательность играют решающую роль в определении лечения растения.

# 4 ИНФОРМАЦИОННАЯ МОДЕЛЬ ПРОГРАММНОГО СРЕДСТВА ДЛЯ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ

Информационная модель IDEF1X, разработанная для программного средства управления опрыскивателем растений, представляет собой реляционную схему базы данных, обеспечивающую структурированное хранение и обработку данных о растениях, болезнях, препаратах, зонах опрыскивания и пользователей [10]. Она состоит из взаимосвязанных сущностей, объединённых первичными и внешними ключами, что позволяет эффективно управлять справочниками и поддерживать целостность операций при расчёте зон и маршрутов опрыскивателя. Ниже приведена информация о ключевых сущностях, их атрибутах и связях, а также подтверждение соответствия модели третьей нормальной форме (3NF), демонстрирующее отсутствие избыточности и обеспечение надёжности данных.

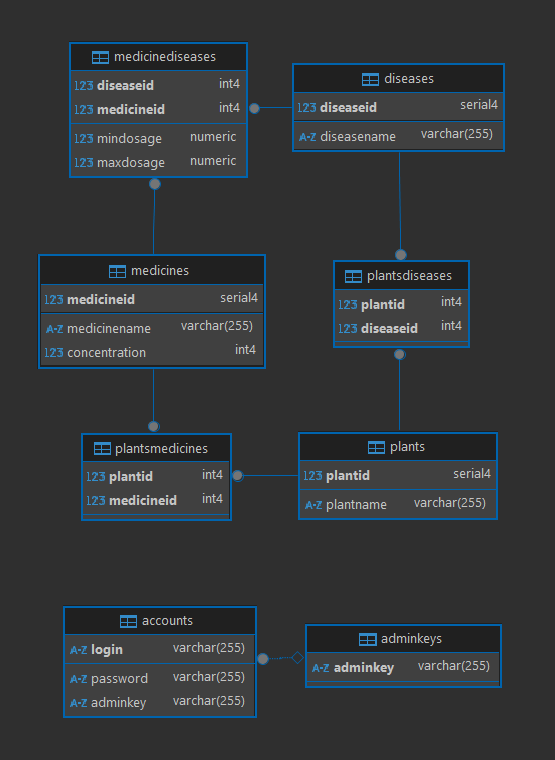


Рисунок 4.1 – Информационная модель IDEF1X

Сущность «AdminKeys» определяет ключи администраторов. Каждый ключ уникален и принадлежит лишь одному администратору. Атрибут adminKey хранит значение ключа, которое позволяет распознать администратора в системе.

Сущность «Accounts» содержит персональные данные людей, зарегистрированных в системе. Атрибуты включают login, password и adminKey (ключ администратора), предоставляя информацию об аккаунтах пользователей

Сущность «Plants» хранит информацию о растениях, на предприятии. Каждое растение идентифицируется первичным ключом plantId. Атрибут plantName содержит название растения.

Сущность «Medicines» хранит информацию о препаратах, на предприятии. Каждое растение идентифицируется первичным ключом medicineId. Атрибут medicineName содержит название растения.

Сущность «Diseases» описывает существующие болезни растений. Каждая болезнь идентифицируется первичным ключом diseaseId. Атрибуты включают disiaseName содержит название болезни.

Сущность «PlantsDiseases» является связующей таблицей, а также хранит информацию о совместимости растений и болезней. Уникальность записи обеспечивается составным первичным ключом (plantId, diseaseId), которые ссылаются на таблицы «Plants» и «Diseasses».

Сущность «PlantsMedicines» является связующей таблицей, а также хранит информацию о совместимости растений и препаратов. Уникальность записи обеспечивается составным первичным ключом (plantId, medicineId), которые ссылаются на таблицы «Plants» и «Medicines».

Сущность «MedicineDisease» является связующей таблицей, описывающей отношение между препаратами и болезнями. Она имеет составной первичный ключ (diseaseId, medicineId), которые ссылаются на таблицы «Diseases» и «Medicines». Аттрибуты у этой сущности определяют пороги минимальной и максимальной дозировок препарата на литр жидкости.

Модель устанавливает логические связи между сущностями, обеспечивая целостность данных и возможность выполнения запросов для управления транзакциями.

Связь между «Plants» и «PlantsDiseases» – один-ко-многим: одно растение может болеть разными болезнями, что реализовано через атрибут plantId в таблице «PlantsDiseases».

Связь между «Diseases» и «PlantsDiseases» – один-ко-многим: одна болезнь может встречаться у разных растений, что реализовано через атрибут diseaseId в таблице «PlantsDiseases».

Связь между «Plants» и «PlantsMedicines» – один-ко-многим: одно растение может обрабатываться разными препаратами, что реализовано через атрибут plantId в таблице «PlantsMedicines».

Связь между «Medicines» и «PlantsMedicines» – один-ко-многим: один препарат может применяться к разным растениям, что реализовано через атрибут medicineId в таблице «PlantsMedicines».

Связь между «Medicines» и «MedicinesDiseases» – один-ко-многим: одно растение может обрабатываться разными препаратами, что реализовано через атрибут medicineId в таблице «MedicinesDiseases».

Связь между «Diseases» и «MedicinesDiseases» – один-ко-многим: один препарат может применяться к разным растениям, что реализовано через атрибут diseaseId в таблице «MedicinesDiseases».

Модель соответствует первой нормальной форме (1NF), так как все атрибуты атомарны.

Для второй нормальной формы (2NF) неключевые атрибуты должны зависеть от всего первичного ключа. В например таблицах с простыми ключами (Plants, Medicines, Diseases), plantName зависит только от plantId. В таблицах с составными ключами (MedicineDiseases) Дозировки (minDosage, maxDosage) зависят от комбинации (medicineId, diseaseId) , что также удовлетворяет 2NF.

Для третьей нормальной формы (3NF) неключевые атрибуты не должны иметь транзитивных зависимостей. В «Medicines» concentration зависит только от medicineId. В «MedicineDiseases» Дозировки зависят только от пары (medicineId, diseaseId). В «Accounts» password и adminKey зависят только от login. Таким образом, модель полностью соответствует 3NF.

Информационная модель IDEF1X для системы дистанционного управления опрыскивателем растений представляет собой логически выстроенную базу данных, соответствующую третьей нормальной форме, что исключает избыточность данных и транзитивные зависимости. Она обеспечивает все необходимые данные и связи для выполнения операций с акциями, гарантируя точность и производительность. Благодаря строгой структуре и соблюдению нормальных форм, данная модель является эффективным решением для автоматизации управления акциями в инвестиционных фондах.

# 6 ОПИСАНИЕ АЛГОРИТМОВ, РЕАЛИЗУЮЩИХ БИЗНЕС-ЛОГИКУ СЕРВЕРНОЙ ЧАСТИ ПРОГРАММНОГО СРЕДСТВА ДЛЯ УПРАВЛЕНИЯ АКЦИЯМИ ИНВЕСТИЦИОННОГО ФОНДА

Разработка сложных программных систем, таких как средство для управления акциями инвестиционного фонда, требует тщательного проектирования и структурированного подхода к реализации функциональности. Одним из ключевых инструментов в этом процессе является использование схем алгоритмов, которые обеспечивают наглядное представление последовательности операций, необходимых для выполнения бизнес-логики. Схемы алгоритмов позволяют разработчикам визуализировать процессы, выявлять логические зависимости и оптимизировать выполнение задач, минимизируя ошибки и упрощая сопровождение системы. В контексте серверной части приложения, где сосредоточена основная бизнес-логика, такие схемы становятся особенно важными, поскольку они обеспечивают прозрачность, предсказуемость и согласованность обработки данных, что критично для финансовых систем [15].

Предметная область управления акциями инвестиционного фонда характеризуется высокой сложностью и динамичностью. Инвестиционные фонды оперируют значительными объемами финансовых данных, включая рыночные котировки, портфели акций, транзакции и пользовательские счета. Основные процессы включают мониторинг рыночных данных, выполнение операций покупки и продажи акций, управление портфелем, а также формирование аналитических отчетов. Эти задачи требуют высокой точности, скорости обработки и соответствия регуляторным требованиям, так как даже незначительные ошибки могут привести к финансовым потерям или нарушению законодательства. Автоматизация этих процессов с помощью программного средства позволяет повысить эффективность, обеспечить прозрачность операций и минимизировать риски, связанные с человеческим фактором.

Серверная часть программного средства играет центральную роль в управлении акциями, так как именно она отвечает за обработку запросов от клиентских приложений, выполнение бизнес-логики и взаимодействие с базой данных. Бизнес-логика на сервере включает такие процессы, как авторизация пользователей, проверка условий транзакций, обновление портфелей и счетов, а также логирование операций. Эти процессы должны быть надежными, масштабируемыми и оптимизированными для обработки множества запросов в реальном времени. Использование схем алгоритмов для описания бизнес-логики позволяет разработчикам четко определить шаги выполнения каждого процесса, установить последовательность операций и предусмотреть обработку исключительных ситуаций, таких как недостаток средств на счете или отсутствие запрашиваемых акций.

Преимущества использования схем алгоритмов многочисленны. Во-первых, они обеспечивают визуальную ясность, позволяя разработчикам и аналитикам быстро понять логику работы системы без необходимости углубляться в код. Это особенно важно на этапе проектирования, когда необходимо согласовать требования между участниками проекта. Во-вторых, схемы упрощают выявление узких мест и потенциальных ошибок, таких как логические противоречия или избыточные операции, что способствует повышению качества программного продукта. В-третьих, они служат документацией, которая остается актуальной на протяжении всего жизненного цикла системы, облегчая ее сопровождение и доработку. Для финансовых приложений, где требуется строгая проверяемость и аудит операций, схемы алгоритмов становятся незаменимым инструментом, обеспечивая прозрачность и возможность воспроизведения каждого шага.

В данном разделе будут рассмотрены схемы алгоритмов, реализующих ключевые аспекты бизнес-логики серверной части программного средства. Эти схемы охватывают такие процессы, как аутентификация пользователей, обработка транзакций и другие. Каждая схема сопровождается подробным описанием, включающим входные данные, условия выполнения, последовательность операций и возможные результаты. Такой подход позволяет не только документировать функциональность системы, но и обеспечить ее соответствие требованиям предметной области, включая точность, надежность и производительность.

Таким образом, использование схем алгоритмов в разработке программного средства для управления акциями инвестиционного фонда является важным шагом для достижения высокого качества системы. Они обеспечивают структурированный подход к реализации бизнес-логики, упрощают коммуникацию между разработчиками и другими участниками проекта, а также создают основу для дальнейшего тестирования и оптимизации. Данная глава предоставит детальное описание алгоритмов, лежащих в основе серверной части системы, подчеркивая их роль в автоматизации финансовых процессов и повышении эффективности инвестиционной деятельности.

# 6.1 Схема жизненного цикла клиент-серверного взаимодействия

В разработке программного средства для дистанционного управления опрыскивателем растений ключевое значение имеет четкое определение процессов, обеспечивающих взаимодействие между клиентской и серверной частями системы. Схема жизненного цикла клиент-серверного взаимодействия описывает последовательность операций, начиная от инициализации системы и заканчивая ее завершением, включая этапы авторизации и обработки запросов. Такая схема необходима для структурирования бизнес-логики серверной части, взаимодействие с базой данных и предоставление данных клиентам.



Рисунок 6.1 Схема жизненного цикла клиент-серверного взаимодействия

Схема иллюстрирует жизненный цикл клиент-серверного взаимодействия в системе дистанционного управления опрыскивателем растений. Рассмотрим этапы кратко:

1 Начало. Точка старта процесса.

2 Запуск сервера. Серверное приложение активируется и начинает прослушивать сеть для принятия подключений.

3 Запуск клиента. Клиентское приложение запускается на устройстве пользователя и инициирует подключение к серверу.

4 Авторизация в системе. Клиент отправляет учетные данные (логин, пароль), сервер их проверяет и предоставляет доступ при успехе.

5 Главный цикл, работает пока клиент не отключится.

6 Получение запроса. Сервер принимает запрос от клиента (например, запрос данных или команды).

7 Обработка запроса. Сервер выполняет обработку запроса (обращение к базе данных, вычисления и т.д.).

8 Отправка ответа: Сервер возвращает клиенту результат обработки.

9 Конец при отключении клиента.

# 6.2 Схема процесса авторизации пользователей



Рисунок 6.2 – Схема процесса авторизации пользователей

Схема иллюстрирует процесс авторизации и регистрации пользователя в клиент-серверной системе управления акциями инвестиционного фонда. Рассмотрим этапы кратко:

1. Начало. Точка старта процесса.
2. Вывод панели авторизации. Клиентское приложение отображает интерфейс для ввода учетных данных.
3. Пользователь решил зарегистрироваться. Пользователь выбирает между регистрацией и авторизацией:

3.1 Ввод логина и пароля. Пользователь вводит свои учетные данные.

3.2 Отправка запроса на авторизацию. Клиент отправляет данные на сервер для проверки.

3.3 Пользователь существует. Сервер проверяет наличие пользователя в системе:

3.3.1 Проверка прав пользователя. Сервер определяет права доступа пользователя.

3.3.2 Роль пользователя. Определение роли:

– User. Отображение меню пользователя;

–Admin. Отображение меню администратора.

3.3.3 Проверка прав пользователя. Сервер определяет права доступа пользователя.

3.3.4 Вывод ошибки. Клиент отображает сообщение об ошибке.

3.3.5 Ввод логина, пароля и ключа администратора. Пользователь вводит данные для создания учетной записи.

3.3.6 Отправка запроса на регистрацию. Клиент отправляет данные на сервер.

3.3.7 Данные введены корректно?. Сервер проверяет правильность введенных данных:

3.3.8 Регистрация пользователя. Сервер создает нового пользователя.

* + 1. Вывод ошибки. Клиент отображает сообщение об ошибке.

4 Конец. Завершение процесса.

# 6.3 Схема процесса расчёта инструкций лечения растения

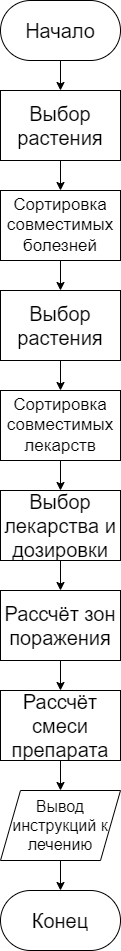


Рисунок 6.3 – Схема расчёта процесса лечения растения

Схема иллюстрирует процесс расчета лечения растения в системе дистанционного управления опрыскивателем растений. Рассмотрим этапы кратко:

1. Начало. Точка старта процесса.
2. Выбор растения. Пользователь выбирает растение для обработки в клиентском интерфейсе.
3. Сортировка совместимых болезней. Система анализирует и отбирает возможные болезни, характерные для выбранного растения.
4. Выбор растения. Подтверждение выбора растения (возможно, уточнение или корректировка).
5. Сортировка совместимых лекарств. Система определяет подходящие препараты для лечения выявленных болезней.
6. Выбор лекарства и дозировки. Пользователь или система выбирает конкретное лекарство и его дозировку.
7. Расчет зон поражения. Система определяет области растения, требующие обработки, на основе анализа.
8. Расчет смеси препарата. Система рассчитывает состав и пропорции смеси для опрыскивания.
9. Вывод инструкций к лечению. Клиенту предоставляются инструкции по применению препарата.
10. Конец. Завершение процесса.

# 7 МОДЕЛИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА ДЛЯ ДИСТАНЦИОННОГО УПРАВЛЕНИЯ ОПРЫСКИВАТЕЛЕМ РАСТЕНИЙ

Современные системы дистанционного управления опрыскивателем растений требуют эффективных и надёжных программных решений для управления справочниками растений, болезней, препаратов, а также для обеспечения взаимодействия между различными участниками процесса опрыскивания. Разработка такого программного средства является сложной задачей, которая требует тщательного проектирования, анализа и моделирования на всех этапах жизненного цикла системы. Для достижения этой цели широко применяются методы объектно-ориентированного анализа и проектирования, в частности, использование унифицированного языка моделирования UML (Unified Modeling Language). UML предоставляет мощный набор инструментов для визуализации, спецификации, конструирования и документирования программных систем, что позволяет разработчикам и аналитикам создавать чёткие и понятные модели, отражающие как функциональные, так и структурные аспекты системы [16].

В данной главе рассматривается процесс моделирования программного средства, предназначенного для дистанционного управления опрыскивателем растений. Основное внимание уделяется применению различных типов диаграмм UML, которые позволяют описать систему с разных точек зрения: функциональной, структурной, поведенческой, архитектурной и физической. Такой комплексный подход обеспечивает глубокое понимание системы, её компонентов, взаимодействия между ними, а также её поведения в различных сценариях использования. Моделирование с использованием UML не только упрощает процесс разработки, но и способствует улучшению качества программного продукта за счёт раннего выявления потенциальных проблем, таких как несогласованность требований, избыточная сложность структуры или неоптимальная архитектура [17].

Целью данной главы является создание всестороннего описания системы дистанционного управления опрыскивателем растений, начиная с её функциональных возможностей и заканчивая физическим размещением компонентов на аппаратных узлах. Каждый из разделов будет сопровождаться подробным описанием и анализом. Такой подход позволяет не только формализовать процесс проектирования, но и обеспечить согласованность между архитектурой системы и её конечной реализацией. Кроме того, использование UML-диаграмм способствует улучшению коммуникации между членами команды разработчиков, аналитиками и другими заинтересованными сторонами, что особенно важно в сложных проектах, таких как разработка программного средства для опрыскивателя растений.

Важно отметить, что система, рассматриваемая в данной главе, ориентирована на клиент-серверную архитектуру, где сервер отвечает за обработку запросов, управление данными и взаимодействие с базой данных, а клиент предоставляет интерфейс для взаимодействия оператора с системой. В рамках моделирования будут учтены такие аспекты, как безопасность (например, авторизация пользователей), производительность (оптимизация взаимодействия с базой данных) и масштабируемость (возможность обработки множества запросов от Unity-клиентов). Особое внимание будет уделено специфике предметной области – управлению справочниками растений, болезней и препаратов, что требует точного учёта данных о растениях, их заболеваниях, доступных химических составах и ролях пользователей (например, администратор, оператор).

В результате работы будет создана комплексная модель программного средства, которая может быть использована как основа для дальнейшей разработки, тестирования и внедрения системы. Модели UML, представленные в этой главе, помогут не только разработчикам, но и аналитикам, тестировщикам и менеджерам проекта, предоставляя единое визуальное представление системы на всех уровнях её проектирования. Далее в главе каждый из перечисленных аспектов будет рассмотрен подробно, начиная с моделирования функциональности системы с помощью диаграммы вариантов использования.

# 7.1 Моделирование функциональности системы с помощью диаграммы вариантов использования

Диаграмма вариантов использования (Use Case) является одним из ключевых инструментов UML, предназначенных для описания функциональности системы с точки зрения взаимодействия пользователей с ней. Она позволяет определить основные роли (актеров), которые взаимодействуют с системой, а также сценарии использования (варианты использования), которые отражают, какие действия могут выполнять пользователи для достижения своих целей. Диаграмма помогает структурировать требования к системе, выявить основные сценарии использования и обеспечить их согласованность с ожиданиями пользователей. В данном разделе будет представлена диаграмма вариантов использования, описывающая функциональные возможности системы, а также проведен ее подробный анализ с акцентом на роли пользователей и их взаимодействие с системой.

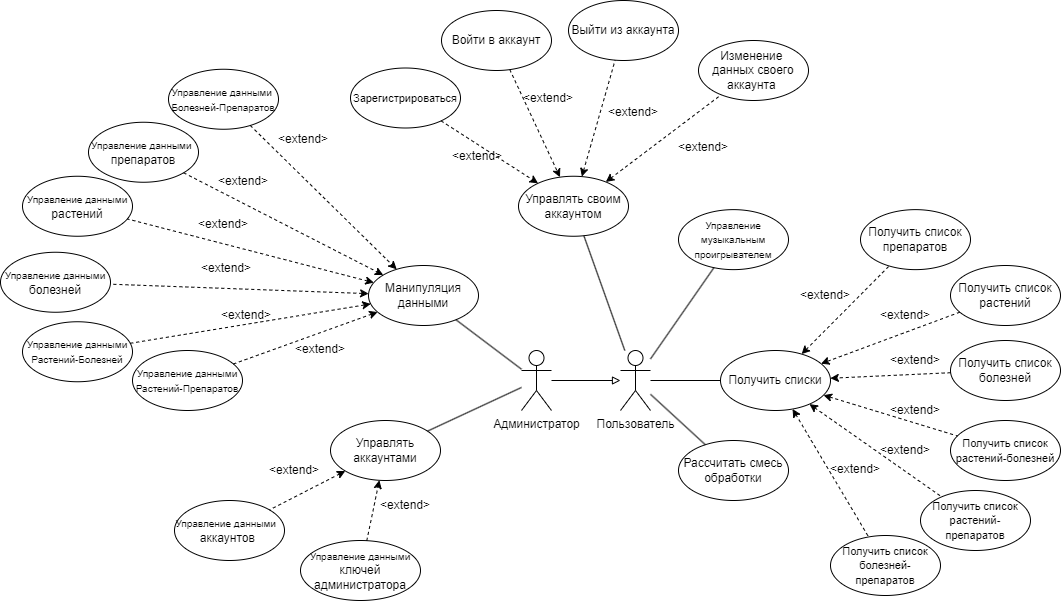


Рисунок 7.1 – Диаграмма вариантов использования системы дистанционного управления опрыскивателем.

Рассмотрим диаграмму более подробно. Начнем с актеров, присутствующих на диаграмме:

1 Администратор. Этот актёр представляет пользователя с расширенными правами, отвечающего за поддержку справочников и пользователей в системе дистанционного управления опрыскивателем растений. Администратор управляет данными об аккаунтах, ключах администратора и всеми справочниками (растений, болезней, препаратов и их связей).

2 Пользователь. Этот актёр представляет оператора, который использует клиентское приложение Unity для выбора участка, формирования маршрута опрыскивателя и расчёта смеси обработки. Пользователь имеет ограниченные права: он может авторизоваться, просматривать справочники, задавать параметры обрабатываемого участка и получать результат расчёта смеси.

Диаграмма содержит два набора вариантов использования: те, что доступны Администратору, и те, что доступны Пользователю и Администратору.

Администратор имеет доступ к следующим вариантам использования:

1 Управлять аккаунтами. Этот сценарий включает действия по созданию, чтению, обновлению и удалению учётных записей пользователей. Администратор может добавить нового оператора, изменить данные существующего (логин, роль) или удалить его из системы.

2 Управление данными ключей администратора. Подсценарий управления аккаунтами: позволяет создавать и удалять дополнительные ключи, дающие доступ к административным функциям.

3 Манипуляция данными (CRUD). Этот общий сценарий включает создание, чтение, обновление и удаление записей во всех справочниках системы. Относится к следующим под-сценариям:

3.1 Управление данными растений;

3.2 Управление данными болезней;

3.3 Управление данными препаратов;

3.4 Управление данными «растений–болезней»;

3.5 Управление данными «растений–препаратов».

4 Получить списки. Этот сценарий позволяет загрузить из базы все справочники в виде одной выборки. Посредством отношения «include» он включает:

4.1 Получить список препаратов;

4.2 Получить список растений;

4.3 Получить список болезней;

4.4 Получить список растений–болезней;

4.5 Получить список растений–препаратов;

4.6 Получить список болезней–препаратов.

5 Рассчитать смесь обработки. Этот сценарий позволяет собрать из справочников необходимые данные (растения, болезни, препараты, их совместимости), задать дозировку, а затем выполнить расчёт зоны распыления и объёма жидкости для каждой зоны.

Пользователь, а также администратор имеет доступ к следующим вариантам использования:

1 Зарегистрироваться. Позволяет создать новую учётную запись оператора;

2 Войти в аккаунт. Сценарий аутентификации: пользователь вводит логин и пароль, клиент отправляет запрос на сервер, сервер проверяет данные;

3 Управлять своим аккаунтом. После входа в систему посетитель может просматривать и изменять свои личные данные (логин, пароль). Этот сценарий включает через отношение «extend»:

3.1 Изменение данных своего аккаунта;

3.2 Выйти из аккаунта.

4 Получить списки. Позволяет получить все справочники для выбора участка и препаратов. Включает:

4.1 Получить список препаратов;

4.2 Получить список растений;

4.3 Получить список болезней;

4.4 Получить список растений–болезней;

4.5 Получить список растений–препаратов;

* 1. Получить список болезней–препаратов.

5 Рассчитать смесь обработки. После выбора болезни и препарата пользователь формирует точки обработки, указывает дозировку и радиус, а система вычисляет зоны опрыскивания и объёмы жидкости.

# 7.2 Описание структуры системы на основе диаграммы классов

Диаграмма классов является ключевым инструментом в объектно-ориентированном проектировании, позволяющим визуализировать структуру системы, её компоненты, их взаимодействие и зависимости. Она предоставляет разработчикам и архитекторам чёткое представление о классах, их атрибутах, методах и связях, что упрощает процесс проектирования, реализации и сопровождения системы. В данном разделе представлены диаграммы классов клиента и сервера, которые описывают архитектуру системы дистанционного управления опрыскивателем растений. Диаграмма клиента фокусируется на пользовательском интерфейсе Unity и логике взаимодействия с сервером, тогда как диаграмма сервера (будет представлена далее) раскроет серверную логику обработки TCP-запросов и управления данными. Эти диаграммы помогают понять, как система разделяет ответственность между клиентом и сервером, обеспечивая модульность и масштабируемость..

На рисунке 7.2 изображена диаграмма классов клиентской части приложения, которая включает в себя классы реализующие процесс взаимодействия пользователя с системой дистанционного управления опрыскивателем. Среди них присутствуют:

– сетевые компоненты (SimpleTCPClient, UIQuerySender) для установления и управления TCP-соединением с сервером;

– контроллеры панелей (AuthorisationPanel, ControlPanel, DosagePanel, AdminPanel и другие), отвечающие за логику отображения и взаимодействие с пользователем через Unity UI;.

– сущности данных (Plant, Disease, Medicine, PlantDisease, PlantMedicine, MedicineDisease, Account и т.д.), которые моделируют информацию о растениях, заболеваниях, препаратах и данных сеансов обработки;

– элементы интерфейса (UIScaling, UISliding, DataStructureElement, Panel и их наследники), обеспечивающие анимацию, плавные переходы и визуализацию данных на экране;

– перечисления (UICommandType), задающие типы команд и действий, передаваемых между клиентом и сервером.

Все эти классы и перечисления связаны ассоциациями и зависимостями, отражая архитектуру клиентской части: одни компоненты формируют и отправляют запросы, другие — обрабатывают ответы и обновляют интерфейс, третьи — управляют локальными коллекциями данных и обеспечивают удобство работы пользователя.

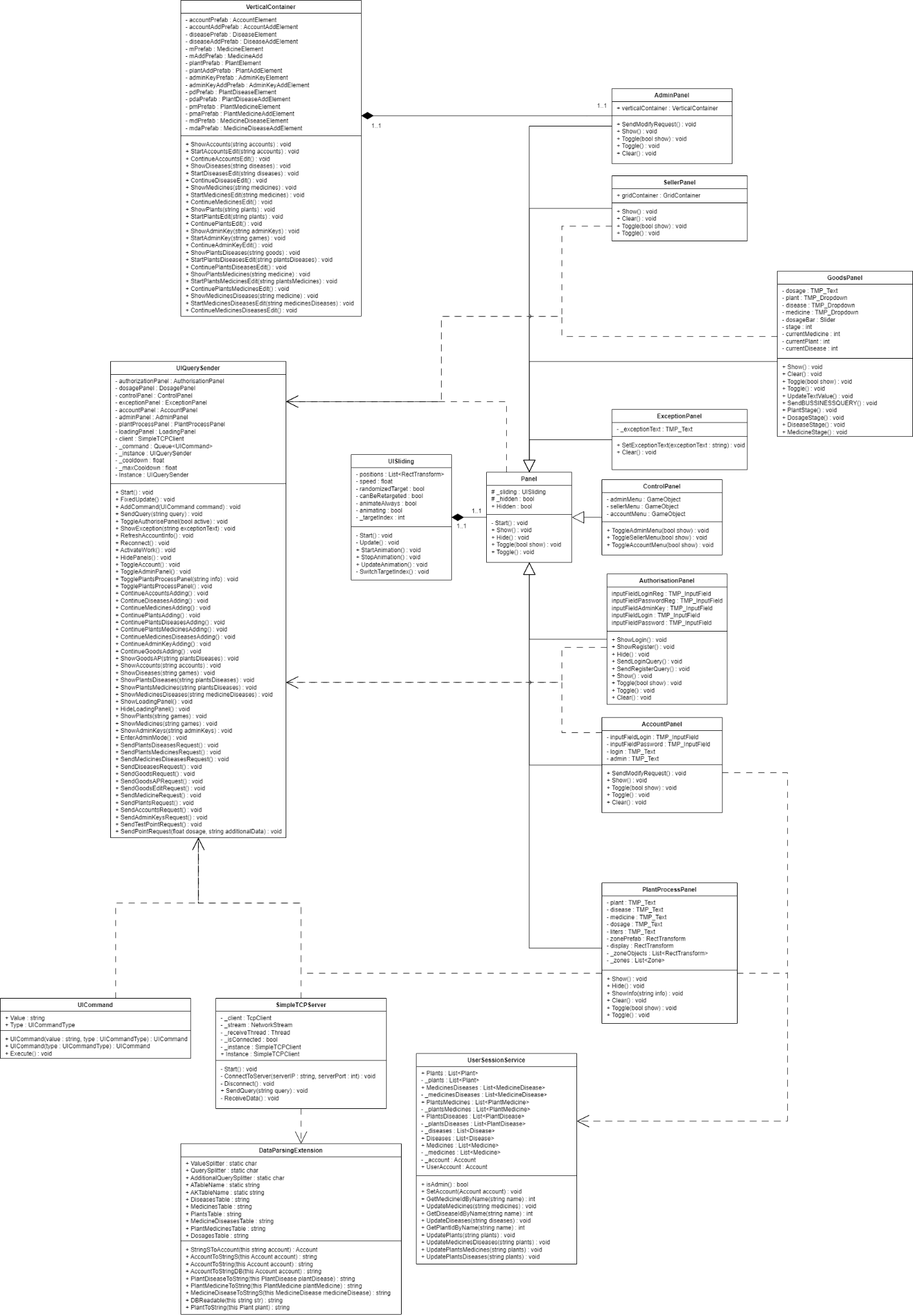


Рисунок 7.2 – Диаграмма классов клиентской части системы дистанционного управления опрыскивателем

На рисунках 7.3.1, 7.3.2 и 7.3.3 изображены диаграммы классов серверной части приложения, которые включают в себя классы и перечисления, реализующие бизнес-логику, взаимодействие с базой данных и механизмы дистанционного управления опрыскивателем. В частности, на этих диаграммах представлены:

– сетевой сервис TCPConnectorService, отвечающий за приём и обработку TCP-запросов от Unity-клиентов;

– сервис доступа к базе данных (DatabaseService), использующий Npgsql для выполнения SQL-запросов и управления транзакциями;

– модели данных (Plant, Disease, Medicine, PlantDisease, PlantMedicine, MedicineDisease, User, Zone) с полями, отражающими структуру таблиц PostgreSQL и связи «многие-ко-многим» или «один-ко-многим» через внешние ключи;

– утилитный класс для разбора входящих сообщений и формирования ответов.

– и другие классы.

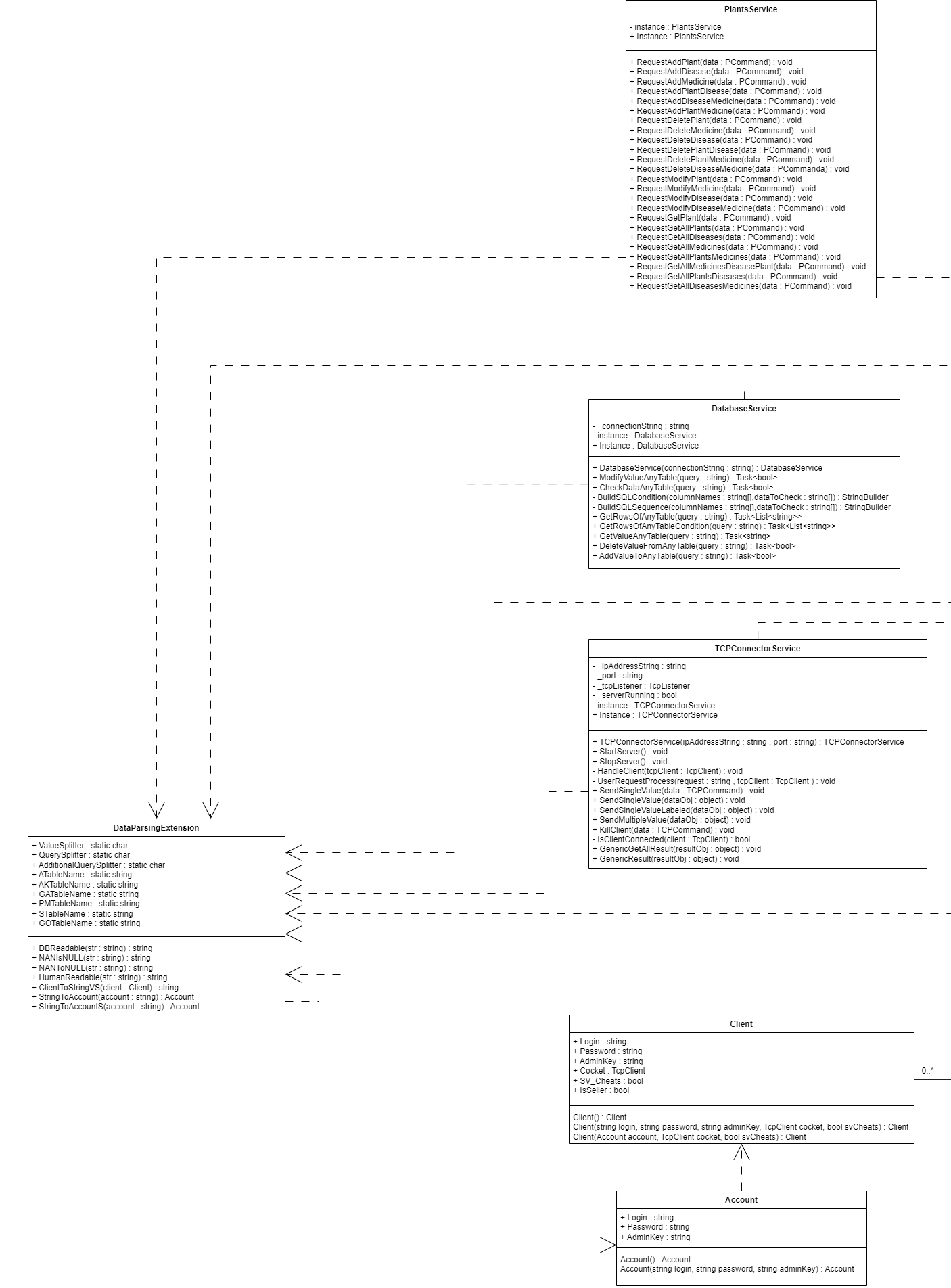


Рисунок 7.3.1 – Диаграмма классов серверной части системы дистанционного управления опрыскивателем, лист 1

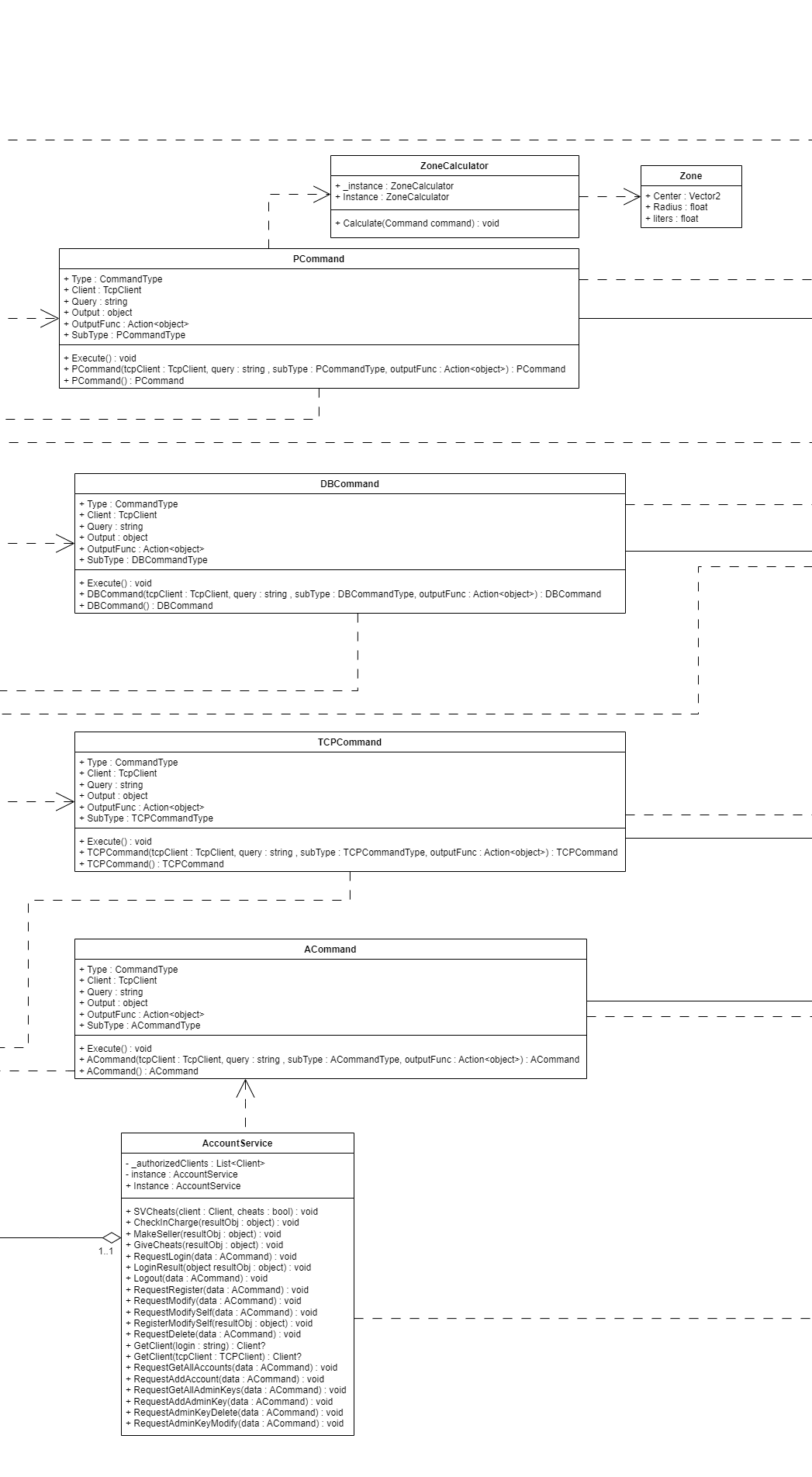


Рисунок 7.3.2 – Диаграмма классов серверной части системы дистанционного управления опрыскивателем, лист 2

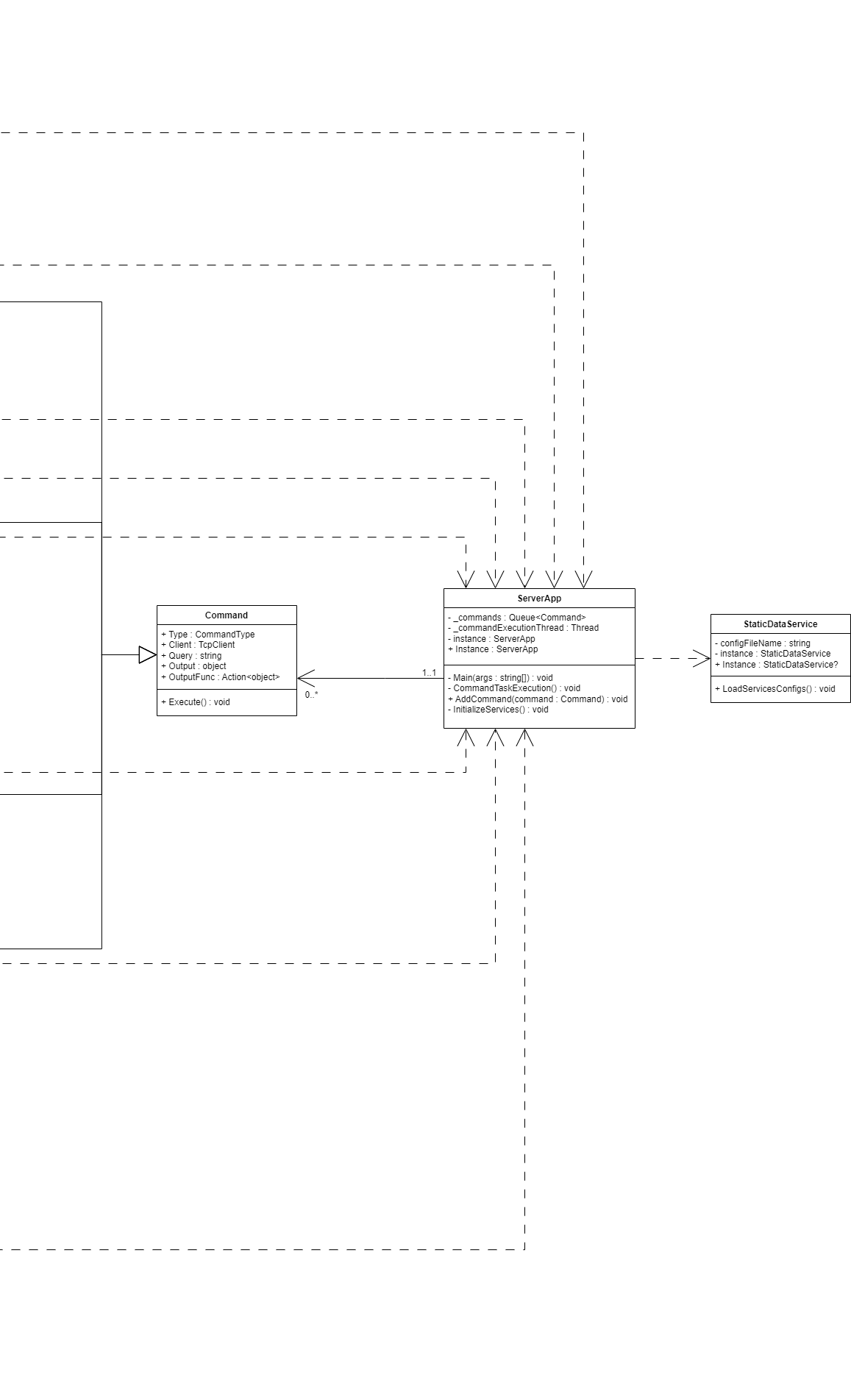


Рисунок 7.3.3 – Диаграмма классов серверной части системы дистанционного управления опрыскивателем, лист 3

# 7.3 Анализ поведения системы с использованием диаграммы последовательностей

Диаграмма последовательностей является важным инструментом для анализа поведения системы, так как она позволяет визуализировать взаимодействие между объектами и компонентами системы во времени. В контексте системы управления портфелем акций диаграмма последовательностей описывает процесс аутентификации пользователя, начиная с ввода данных на клиентской стороне и заканчивая успешным входом в систему или обработкой ответа от сервера. Это помогает понять, как данные передаются между клиентом и сервером, как обрабатываются запросы, и как система реагирует на действия пользователя. Диаграмма также выявляет потенциальные узкие места, такие как задержки в сетевом взаимодействии или ошибки в обработке данных, что делает ее полезной для проектирования, реализации и тестирования системы.

Ниже изображена диаграмма последовательностей процесса аутентификации пользователя в системе (рисунок 7.4).

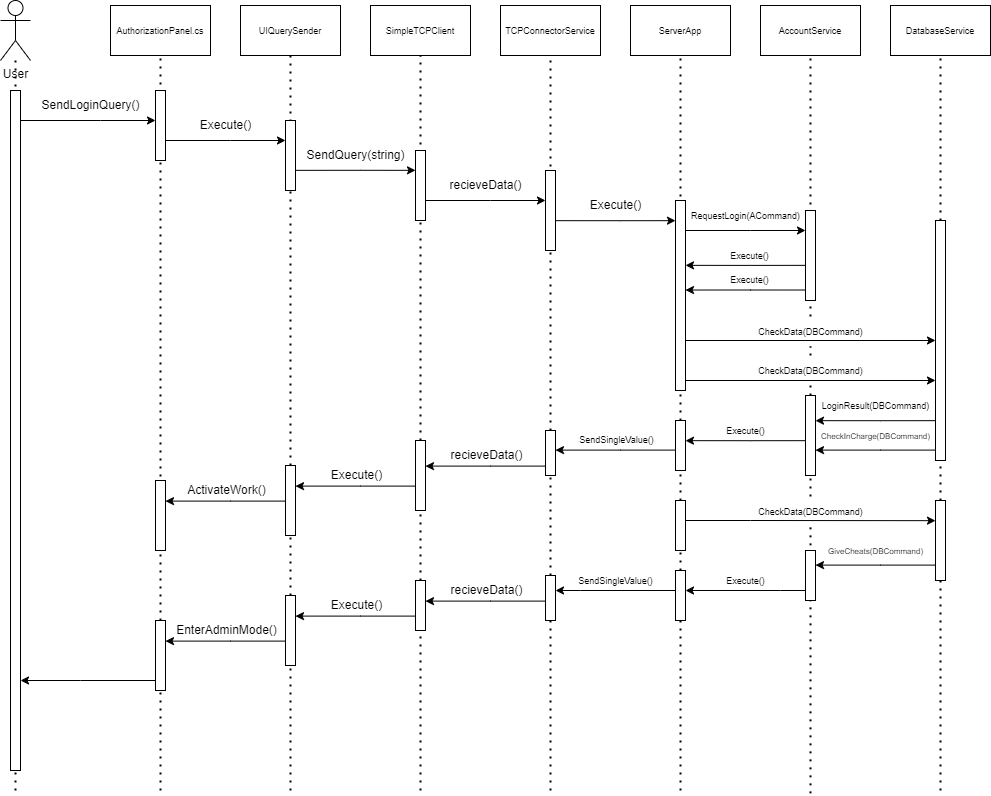


Рисунок 7.4 – Диаграмма последовательностей процесса аутентификации пользователя в системе управления акциями инвестиционного фонда

# 7.4 Моделирование жизненного цикла объектов с помощью диаграммы состояний

Диаграмма состояний (State Machine Diagram) представляет собой мощный инструмент визуального моделирования в UML, который позволяет наглядно отобразить поведение системы в терминах последовательности состояний, через которые проходит объект в ответ на внешние события. Актуальность использования именно этого вида диаграмм обусловлена тем, что процессы дистанционного управления опрыскивателем по своей природе представляют собой конечные автоматы с чётко выраженными состояниями и переходами между ними. В отличие от других UML-диаграмм, диаграмма состояний особенно хорошо подходит для моделирования:

- временных аспектов работы системы;

- реактивного поведения в ответ на внешние события;

- сложной логики принятия решений.

Ниже изображена диаграмма состояний процесса расчёта инструкций к лечению растения (рисунок 7.5).

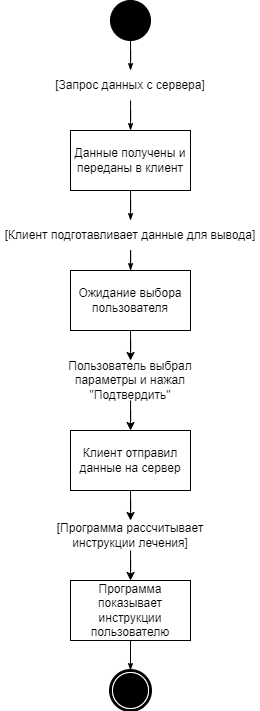


Рисунок 7.5 – Диаграмма состояний процесса покупки акций в системе управления акциями инвестиционного фонда

# 7.5 Моделирование физического размещения системы с помощью диаграммы развертывания

Диаграмма развертывания является важным инструментом в процессе проектирования программного обеспечения, так как она предоставляет наглядное представление физической архитектуры системы. Она показывает, как программные компоненты, называемые артефактами, распределяются по аппаратным устройствам и исполняющим средам, а также описывает их взаимодействие через сетевые соединения. Для программного средства управления акциями инвестиционного фонда диаграмма развертывания необходима для определения инфраструктуры, обеспечивающей функционирование системы. Она помогает разработчикам, системным администраторам и другим заинтересованным сторонам понять, какие аппаратные и программные ресурсы необходимы для работы приложения. Диаграмма позволяет выявить потенциальные узкие места в архитектуре, определить требования к оборудованию и обеспечить масштабируемость и надежность системы. В данном разделе представлена диаграмма развертывания, которая отражает физическое размещение компонентов системы, а также подробное описание ее структуры и процессов.

Диаграмма развертывания программного средства для дистанционного управления опрыскивателем представлена на рисунке 7.6. Она иллюстрирует физическую архитектуру системы, включая аппаратные устройства, исполняющие среды, программные артефакты и их взаимодействие.

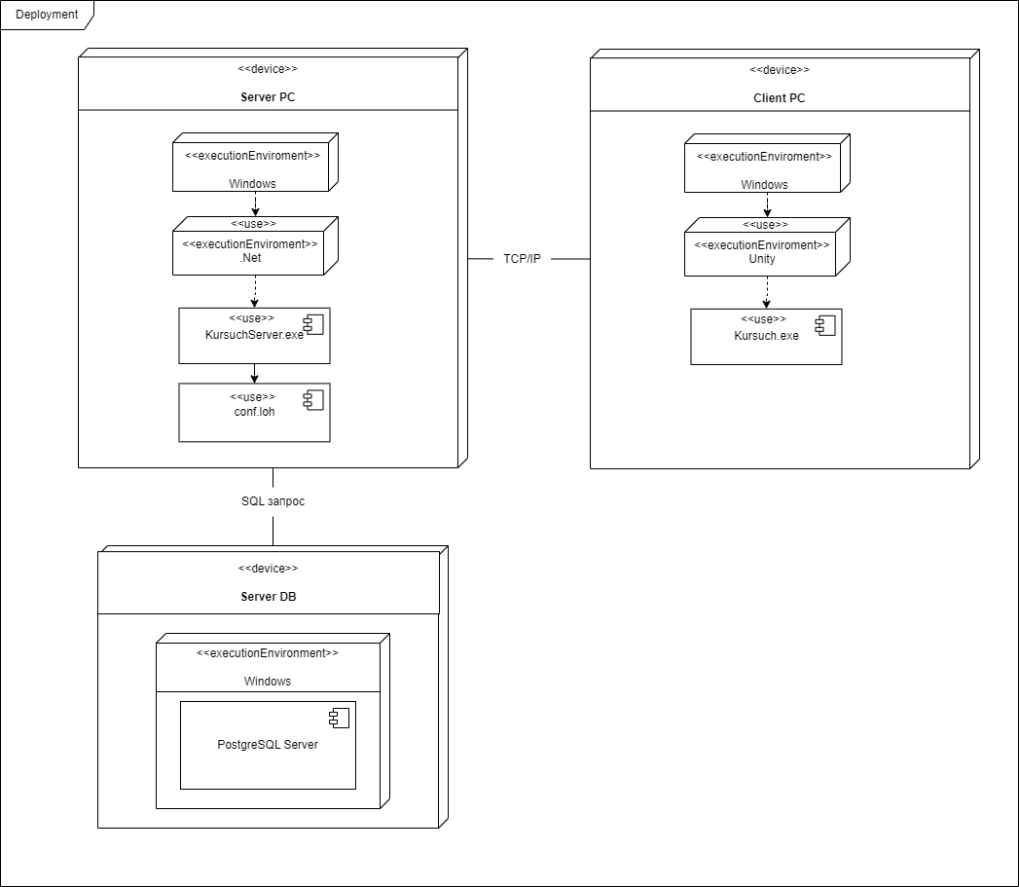


Рисунок 7.6 – Диаграмма развертывания средства для дистанционного управления опрыскивателем

Диаграмма развертывания отражает физическую архитектуру системы и показывает минимальный набор оборудования, необходимый для её функционирования. В состав инфраструктуры входят три основных устройства: машина, на которой развёрнута серверная часть системы; клиентский компьютер с установленным приложением на Unity; а также отдельный сервер баз данных, обеспечивающий хранение и обработку информации. Взаимодействие между клиентом и сервером реализовано с использованием стека протоколов TCP/IP, что гарантирует стабильную передачу данных. Подключение к базе данных осуществляется через библиотеку Npgsql, обеспечивающую эффективное и безопасное взаимодействие с СУБД PostgreSQL.

# 7.6 Вывод по разделу «Моделирование программного средства для управления акциями инвестиционного фонда с использованием UML»

Моделирование программного средства дистанционного управления опрыскивателем растений с использованием языка UML обеспечило комплексное представление о системе, охватывающее её функциональные, структурные, поведенческие и физические аспекты. Применение различных видов UML-диаграмм позволило формализовать архитектуру, упростить этапы проектирования и обеспечить точную координацию между компонентами, что стало надёжной основой для последующей реализации, тестирования и эксплуатации программного продукта.

Диаграмма вариантов использования помогла определить ключевые роли в системе — администратора и оператора — и связать их с основными действиями, доступными через клиентское приложение. Это обеспечило чёткое понимание пользовательских сценариев и позволило выстроить логику интерфейса, соответствующую ожиданиям конечных пользователей. Благодаря учёту взаимосвязей между действиями удалось детализировать поведение системы при различных режимах работы и формализовать требования, предъявляемые к функциям доступа и управления.

Классовые диаграммы клиентской и серверной части отразили структуру системы с акцентом на распределение обязанностей между модулями. Клиентская часть, реализованная в Unity, отвечает за визуализацию и пользовательское взаимодействие, в то время как сервер, реализованный в .NET, обрабатывает сетевые запросы, управляет бизнес-логикой и взаимодействует с базой данных. Отображение классов, их полей, методов и связей обеспечило чёткое соответствие между архитектурой и кодовой реализацией, способствовало модульности, тестируемости и масштабируемости решения.

Диаграмма последовательностей, описывающая процесс авторизации, позволила визуализировать взаимодействие между участниками — от момента ввода данных до обработки на сервере и обратной связи в интерфейсе клиента. Этот сценарий дал возможность оценить динамику сетевых запросов, выявить потенциальные задержки и оптимизировать логику обработки. Благодаря такому анализу обеспечивается устойчивость к сбоям и повышается отзывчивость системы при взаимодействии с пользователем.

Диаграмма состояний, отражающая поведение системы при расчёте зон опрыскивания, показала, как объект (опрыскиватель или сеанс обработки) переходит между состояниями от момента инициализации до завершения обработки. Чёткое определение переходов и условий их выполнения позволило устранить неоднозначности, улучшить предсказуемость поведения и реализовать корректную обработку исключительных ситуаций.

Диаграмма развертывания отразила физическое размещение компонентов: Unity-клиента, серверной части и базы данных. На её основе были определены характеристики взаимодействия между слоями системы, маршруты сетевых соединений, а также требования к аппаратному обеспечению и платформенной независимости. Отдельное внимание было уделено обеспечению надёжного соединения, безопасности передаваемых данных и соответствию системе хранения критически важной информации. Используемые технологии, включая .NET, PostgreSQL, Unity и Npgsql, обеспечили прочную технологическую базу и гибкость масштабирования.

В результате применение UML-диаграмм позволило систематизировать подход к проектированию, устранить дублирование и снизить риски неконсистентности между логикой системы и её реализацией. Визуальное моделирование повысило прозрачность разработки и обеспечило единое понимание структуры всеми участниками проекта. Полученные модели подтвердили, что программное средство для дистанционного управления опрыскивателем обладает высокой степенью надёжности, производительности и адаптивности к дальнейшему расширению.

# 8 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Программное средство для дистанционного управления опрыскивателем растений представляет собой комплексную систему, созданную для автоматизации процессов планирования и реализации операций по обработке сельскохозяйственных участков, контроля параметров опрыскивателя и обеспечения аналитической поддержки операторов. Для эффективного использования этого решения необходимо наличие ясной и понятной документации, которая позволит всем категориям пользователей быстро освоить функционал и выполнять поставленные задачи без лишних рисков. Настоящий раздел разработан таким образом, чтобы представить подробные инструкции по взаимодействию с приложением, оказывая поддержку как администраторам, так и операторам опрыскивателя. Руководство является важной частью проекта, так как оно не только облегчает работу с системой, но и способствует её надежной и корректной эксплуатации в аграрной сфере.

Основная цель данного раздела заключается в том, чтобы описать ключевые возможности программного обеспечения, развернуть последовательность действий для выполнения типовых операций и разъяснить предназначение ролей внутри системы. Администраторы отвечают за настройку и контроль доступа, управление списками растений, заболеваний и препаратов, а также за обобщение статистики о выполненных операциях и обслуживание базы данных. Операторы, в первую очередь, занимаются выбором участка для опрыскивания, вводом координат, заданием радиуса действия и дозировки, запуском вычислительных модулей и визуализацией расчётных зон. Поскольку процессы обработки растений требуют высокой точности и оперативного реагирования на изменения условий, документация уделяет особое внимание надёжности выполнения операций, правильному использованию интерфейса и последовательным действиям, приводящим к ожидаемому результату. Каждый подраздел сопровождается примерами экранных форм и практическими советами по устранению возможных неточностей, что делает руководство удобным инструментом для пользователей с разным уровнем подготовки.

В условиях агротехнической области, где любая ошибка в параметрах обработки может привести к снижению урожайности или избыточному расходу рабочего раствора, руководство по эксплуатации является ключевым элементом. Оно не только помогает оператору быстро освоить интерфейс и логику работы с программой, но и выстраивает единый стандарт взаимодействия с системой, тем самым сокращая количество ошибок и повышая общую эффективность. Благодаря применению передовых технологий, таких как Unity UI для реализации наглядного и интерактивного пользовательского интерфейса и PostgreSQL в связке с Npgsql для стабильного хранения и обработки данных, система обладает понятным дизайном и высокой эксплуатационной надёжностью, что находит отражение в пошаговых инструкциях данного раздела.

# 8.1 Подготовка к эксплуатации

Для начала работы с программным средством дистанционного управления опрыскивателем растений необходимо выполнить подготовительные шаги для правильной настройки и запуска системы. Данный подраздел описывает процесс создания базы данных, запуска серверного и клиентского приложений, авторизации и настройки интерфейса. Эти действия обеспечивают надёжное и безопасное использование системы, что особенно важно в аграрной сфере:

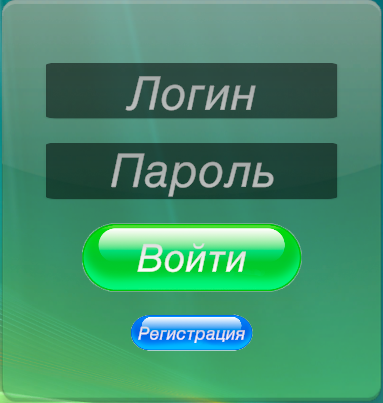
1 Создание базы данных. Программное средство использует PostgreSQL для хранения сведений о растениях, болезнях, препаратах, пользователях и сеансах опрыскивания. Установите PostgreSQL на сервер, создайте новую базу и выполните SQL-скрипт для инициализации таблиц. Это гарантирует сохранение всех необходимых данных.

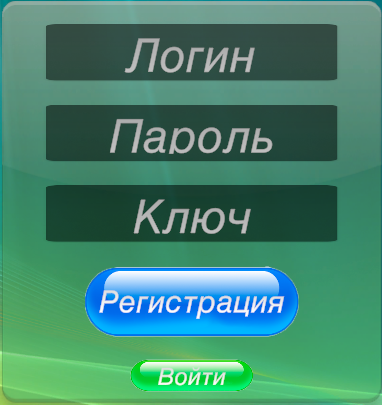
2 Запуск серверного приложения. Консольное C#-приложение запускается на устройстве с .NET 6+. После старта сервер подключается к базе через Npgsql и начинает принимать TCP-запросы от Unity-клиентов. Это обеспечивает работу логики расчёта зон и передачи результатов.

3 Запуск клиентского приложения. Unity-проект на C# (в JetBrains Rider) запускается на Windows или Linux. При старте клиент устанавливает TCP-соединение и отображает интерфейс для авторизации.

4 Авторизация в системе. Для доступа к системе пользователь должен ввести логин и пароль на экране входа. Успешная авторизация открывает интерфейс, соответствующий роли пользователя (администратор или обычный пользователь). Этот шаг гарантирует безопасный доступ и определяет доступные функции. Ниже представлен скриншот, демонстрирующий интерфейс окна авторизации (рисунок 8.1.1).

4.1 В случае, если у пользователя нету учётной записи. Он должен её создать в окне регистрации (Рисунок 8.1.2)





Рисунки 8.1.1 и 8.1.2 – Скриншот окна авторизации и регистрации

5 Окно ошибки. В случае, когда программа выдала ошибку, пользователь увидит окно оповещения с текстом ошибки. (Рисунок 8.2)

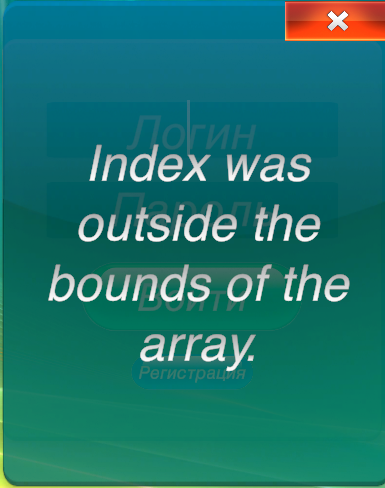


Рисунок 8.2 – Скриншот окна ошибки

6 Панель управления. После авторизации пользователю откроется панель управления, с помощью которой можно перейти в одно из меню программы. (Рисунок 8.3)

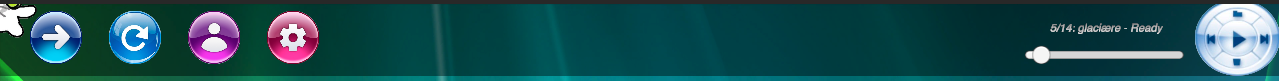


Рисунок 8.3 – Скриншот панели управления

7 Настройка персональных данных. После авторизации пользователь может изменить свои данные, такие как имя логин или пароль, через панель аккаунта в интерфейсе. Ниже представлен скриншот, демонстрирующий интерфейс панели аккаунта (рисунок 8.4).

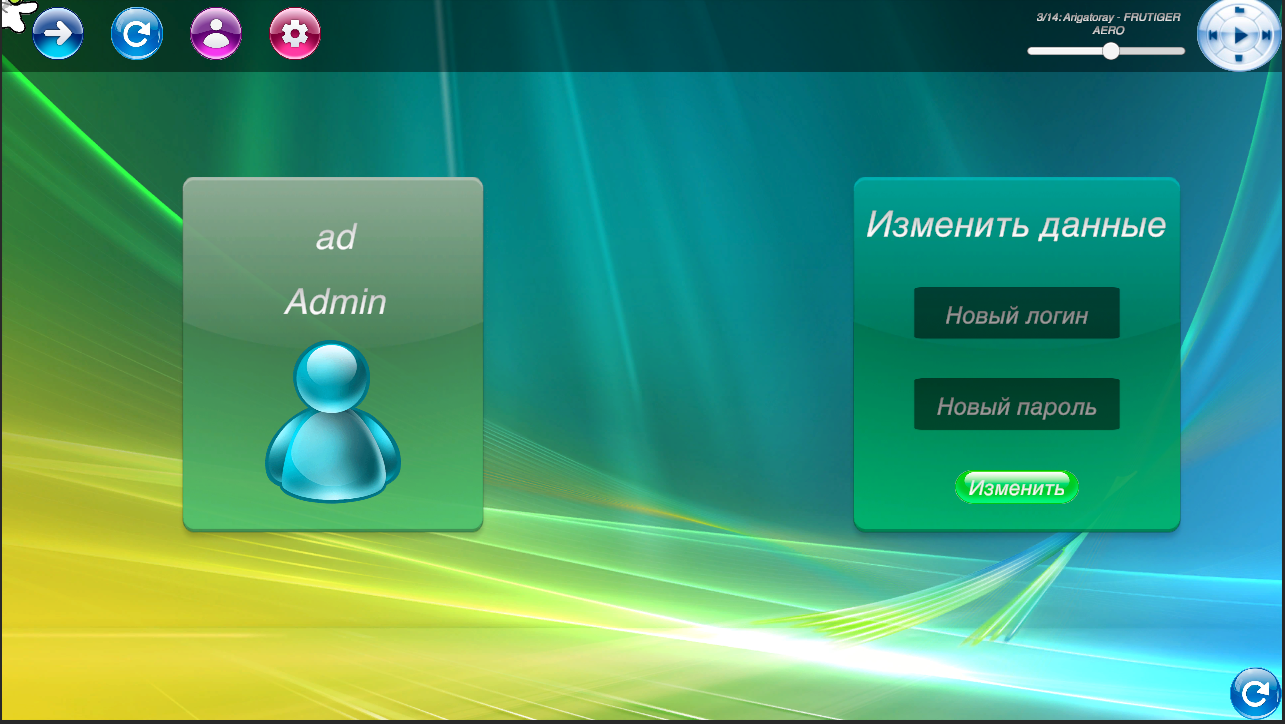


Рисунок 8.4 – Скриншот окна изменения данных

6 Кнопки переподключения находятся в правом нижнем и левом верхнем углах экрана, позволяют пользователю переподключиться к серверу перезапустив клиентское приложение (рисунок 8.5).



Рисунок 8.5 – Скриншот кнопки переподключения

Подготовка к эксплуатации обеспечивает корректную работу программного средства, позволяя пользователям эффективно выполнять задачи, связанные с управлением акциями. Простота настройки и интуитивный интерфейс системы способствуют быстрому освоению, что особенно важно для обеспечения оперативности и точности в финансовых операциях.

# 8.2 Руководство для администратора

Данный подраздел описывает интерфейс и функциональность, доступные пользователю с административными правами. Администратор отвечает за управление учетными записями, акциями, компаниями и настройку системы, что обеспечивает ее стабильную и безопасную работу. В данном разделе представлены основные экраны интерфейса, их элементы и назначение, что позволяет понять, как администратор взаимодействует с системой. Описание ориентировано на предметную область финансов, подчеркивая важность точности и надежности операций.

Панель администратора, представленная на рисунке 8.6.1, она предоставляет доступ к основным функциям управления системой через удобный и структурированный интерфейс. Ниже описаны элементы интерфейса и их назначение:

1 Панель выбора типа записи. В левой части экрана находятся кнопки выбора типа записи, нажав на которые можно добавлять, удалять и изменять соответствующие типы записей базы данных.

2 Записи (в данном случае список болезней). В центральной части экрана расположен список записей с кнопками, позволяющими управлять базой данных.

2.1 Кнопка «Изменить». Позволяет изменить данные конкретной записи в БД, каскадно поменяв её в связанных таблицах (рисунок 8.6.3).

2.2 Кнопка «Удалить». Позволяет удалить данные записи, также каскадно стерев в её в связанных таблицах (рисунок 8.6.3).

2.3 Кнопка «Подтвердить». Позволяет подтвердить добавление новой записи в БД (рисунок 8.6.4).



Рисунок 8.6.1 – Скриншот главного меню администратора



Рисунок 8.6.2 – Скриншот элемента записи

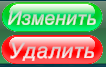


Рисунок 8.6.3 – Скриншот кнопки удалить и изменить на элементе записи



Рисунок 8.6.4 – Скриншот элемента добавления с базу данных

Элементы интерфейса панели администратора организованы в вертикальной компоновке, что обеспечивает удобство навигации. Кнопки имеют четкие подписи, отражающие их назначение. Функциональность, доступная через кнопки, покрывает основные задачи администратора, связанные с управлением системой, что делает меню ключевым элементом административного интерфейса. Добавление и изменение различных типов записей аналогично.

# 8.3 Руководство для пользователя

Данный подраздел описывает интерфейс и функциональность, доступные пользователю с обычными правами. Пользователь может расcчитать лечение растения. В данном разделе представлены основные экраны интерфейса, их элементы и назначение, что помогает понять, как пользователь взаимодействует с системой.

Главное меню, представленное на рисунке 8.7, является основным экраном, отображаемым после успешной авторизации пользователя. Оно служит точкой доступа к функции . Ниже описаны элементы интерфейса и их назначение:

1 Кнопка выбора растения. Позволяет протестировать функционал прототипа, выбрав растение из базы данных.

2 Кнопка выбора болезни. Позволяет протестировать функционал прототипа, выбрав подходящую болезнь из базы данных.

3 Кнопка выбора лекарства. Позволяет протестировать функционал прототипа, выбрав подходящее лекарство из базы данных.

4 Ползунок выбора дозировки лекарства. Позволяет протестировать функционал прототипа, выбрав дозировку лекарства.

5 Кнопка подтверждения выбора. Позволяет начать тестирование прототипа программы запуская процесс расчёта лечения растения.



Рисунок 8.7 – Скриншот главного меню пользователя

# 8.4 Вывод по разделу «Руководство пользователя»

Данный раздел содержит подробное описание процессов и интерфейсов, необходимых для работы с программным средством дистанционного управления опрыскивателем растений. Включённые подразделы, охватывающие подготовку к эксплуатации, работу администратора и оператора, формируют целостное представление о структуре и функциональности системы. Акцент сделан на удобстве использования, надёжности операций и защите данных, что особенно важно в аграрной сфере.

Раздел, посвящённый подготовке к работе, описывает создание базы данных, запуск серверной и клиентской частей, а также начальную авторизацию, что обеспечивает стабильный старт системы. Руководство для администратора охватывает интерфейсы управления справочниками и пользователями, позволяя поддерживать актуальность данных и разграничение прав доступа. Руководство для оператора описывает рабочие панели, используемые для задания параметров обработки и визуализации зон опрыскивания, что упрощает взаимодействие с системой в повседневной работе.

В совокупности содержимое раздела демонстрирует, что руководство пользователя играет важную роль в обеспечении эффективности и безопасности работы программного средства. Чёткое разделение функций, интуитивный интерфейс Unity и корректная обработка пользовательских действий формируют надёжную основу для успешной эксплуатации как администраторами, так и операторами.

# 9 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ РАЗРАБОТАННОЙ СИСТЕМЫ

Тестирование завершает разработку программного средства для дистанционного управления опрыскивателем растений и направлено на подтверждение его функциональности, надёжности и устойчивости при различных условиях эксплуатации. В этом разделе описаны результаты проверок, уделено внимание обработке исключительных ситуаций, некорректных данных и возможных сбоев, возникающих в процессе работы. В агротехнической среде, где точность расчётов, сохранность данных и оперативность реакции имеют решающее значение, тестирование служит ключевым инструментом обеспечения качества и безопасности системы.

Система, реализованная на основе технологий C#, Unity, PostgreSQL, Npgsql, прошла проверку в сценариях как штатного использования, так и экстренных ситуациях. При стандартных операциях проверялось поведение механизма авторизации, приём и корректность команд. В дополнение к этому моделировались ситуации с некорректным вводом—например, передача невалидных данных. В процессе тестирования оценивалось, насколько эффективно система перехватывает исключения, сохраняет непротиворечивость информации в базе данных и сообщает оператору о возникших ошибках.

Одной из ключевых исключительных ситуаций, протестированных в системе, является ввод неправильного логина или пароля при попытке авторизации. Данная ситуация моделирует распространенную ошибку пользователя или потенциальную попытку несанкционированного доступа. При вводе некорректных учетных данных (например, несуществующего логина или неверного пароля) система, возвращает сообщение об ошибке через интерфейс (рисунок 9.1). Это сообщение отображается в диалоговом окне, показывая информацию о внутренней ошибке с сервера, что позволяет персоналу оперативно исправить проблему, а оператору получить точный ответ места проблемы. Система также фиксирует попытку входа в логах обращений к серверу, что позволяет системному администратору сервера отслеживать подозрительную активность.

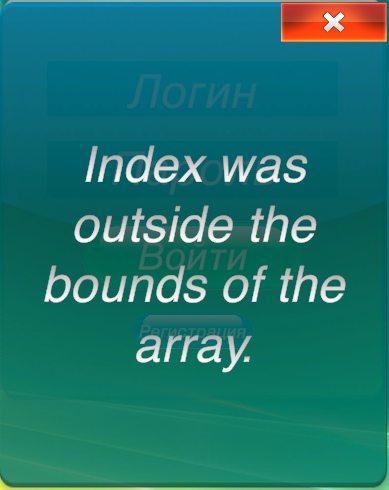


Рисунок 9.1 – Обработка неправильных данных для авторизации

Еще одной протестированной исключительной ситуацией является попытка авторизации с пустыми полями логина и пароля, что моделирует случайный или некорректный ввод со стороны пользователя. При отправке запроса на вход с незаполненными полями система, реализованная с использованием JavaFX (рисунок 9.2), отображает в интерфейсе диалоговое окно с сообщением: «Пожалуйста, заполните все поля». Это уведомление информирует пользователя о необходимости корректного ввода, не позволяя продолжить авторизацию. Проверка пустых полей выполняется на клиентской стороне перед отправкой запроса на сервер, что снижает нагрузку на сервер и ускоряет обработку. Система фиксирует попытку входа в логах через Log4j, записывая временную метку и IP-адрес клиента, что обеспечивает возможность мониторинга. Отсутствие передачи пустых данных на сервер минимизирует риски некорректной обработки и поддерживает целостность системы.

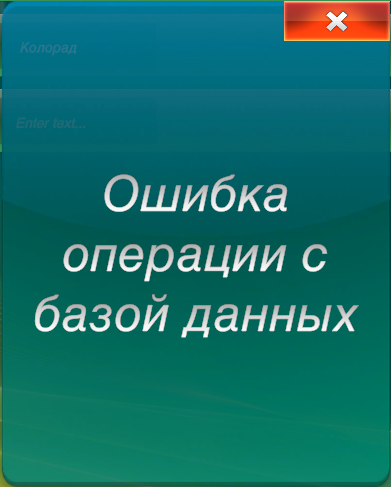


Рисунок 9.2 – Обработка пустых данных для авторизации

Следующей протестированной исключительной ситуацией является отсутствие подключения к серверу, моделирующее сбои в сетевой инфраструктуре или временную недоступность серверного приложения. При попытке клиента установить соединение с сервером через TCP/IP, если сервер недоступен, система отображает диалоговое окно с сообщением: «Ошибка входа» (рисунок 9.3). Клиентская часть системы прекращает попытки отправки запросов, предотвращая перегрузку сети, а информация о сбое записывается в локальные логи с помощью Log4j, включая временную метку и детали ошибки. Сервер, в случае восстановления, автоматически возобновляет обработку подключений без потери данных благодаря использованию Hibernate для транзакционной целостности. Такой подход минимизирует риск повреждения данных, поддерживает стабильность системы и обеспечивает четкую обратную связь, что является критичным для финансовых приложений.

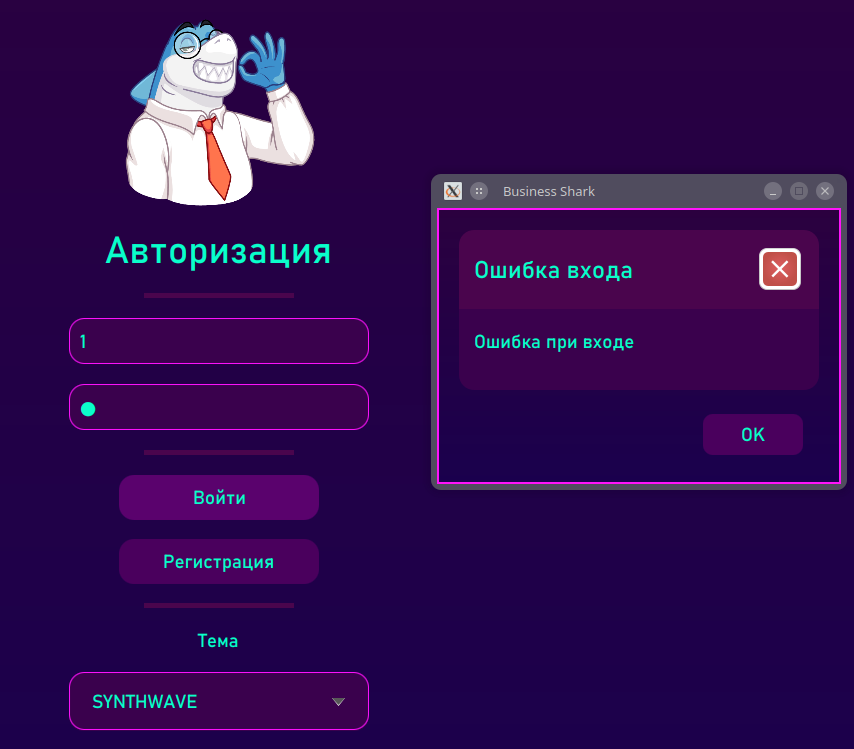


Рисунок 9.3 – Обработка отсутствия подключения к серверу

Еще одной протестированной исключительной ситуацией является ввод текстовой информации в поля, предназначенные исключительно для цифровых данных, например, в поле «Количество акций» на экране управления акциями. При попытке ввести текстовые символы (например, «abc» вместо числа) клиентская часть системы выполняет валидацию ввода на стороне интерфейса и отображает диалоговое окно с сообщением: «Некорректный формат для поля» (рисунок 9.4). Это уведомление информирует пользователя о некорректном формате и предотвращает отправку ошибочных данных на сервер. Валидация использует регулярные выражения для проверки соответствия ввода цифровому формату, что снижает нагрузку на сервер и минимизирует риск некорректной обработки. Такой механизм обработки поддерживает целостность данных и предотвращает сбои, обеспечивая надежность системы.

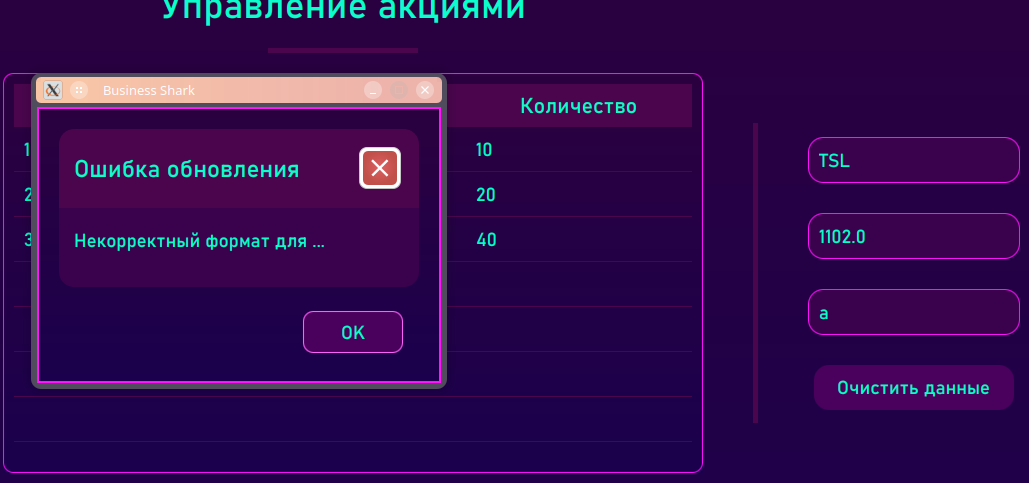


Рисунок 9.4 – Обработка некорректного формата текстового поля

Протестирована ситуация отсутствия подтверждения регистрации, когда новый пользователь, созданный администратором, не завершает процесс активации учетной записи. При регистрации пользователь получает статус «Waiting» в базе данных PostgreSQL, и система ожидает подтверждения. При попытке авторизации с неподтвержденной записью JavaFX-интерфейс отображает сообщение: «Ваш аккаунт ожидает одобрения от администратора» (рисунок 9.5). Инцидент фиксируется в логах через Log4j с указанием времени и логина. Такой механизм повышает безопасность системы и поддерживает ее чистоту.

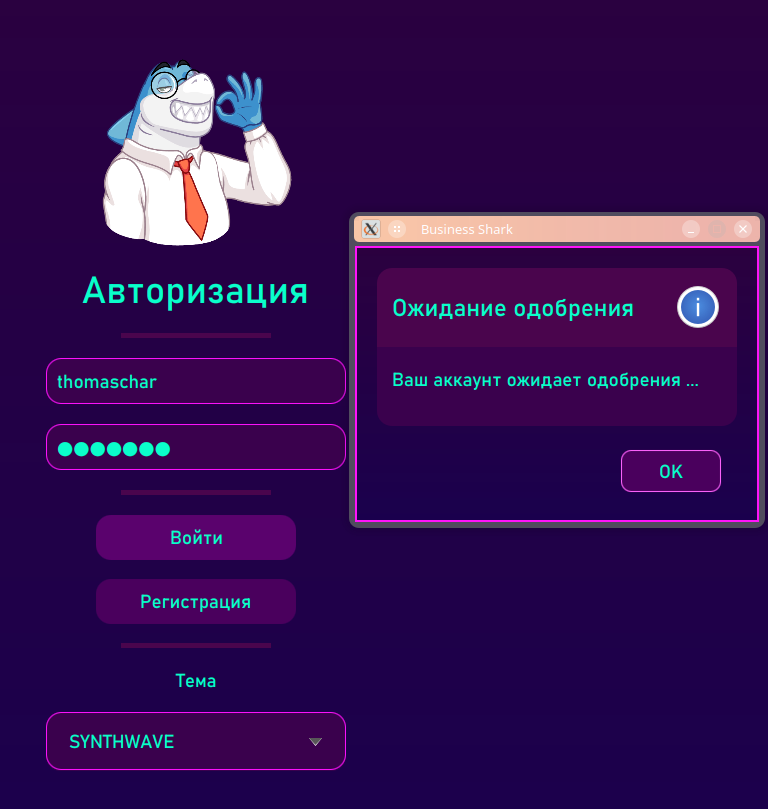


Рисунок 9.5 – Обработка отсутствия подтверждения регистрации

Еще одной протестированной ситуацией является блокировка пользователя, например, из-за подозрительной активности или по решению администратора. При блокировке учетная запись переводится в статус «Blocked» в базе данных. При попытке авторизации заблокированный пользователь получает сообщение в диалоговом окне: «Вы были заблокированы администратором». Система не раскрывает причину блокировки, что снижает риск атак. Заблокированный пользователь не может выполнять операции, администратор может разблокировать учетную запись через интерфейс управления пользователями. Этот механизм защищает систему от несанкционированного доступа и поддерживает доверие пользователей.

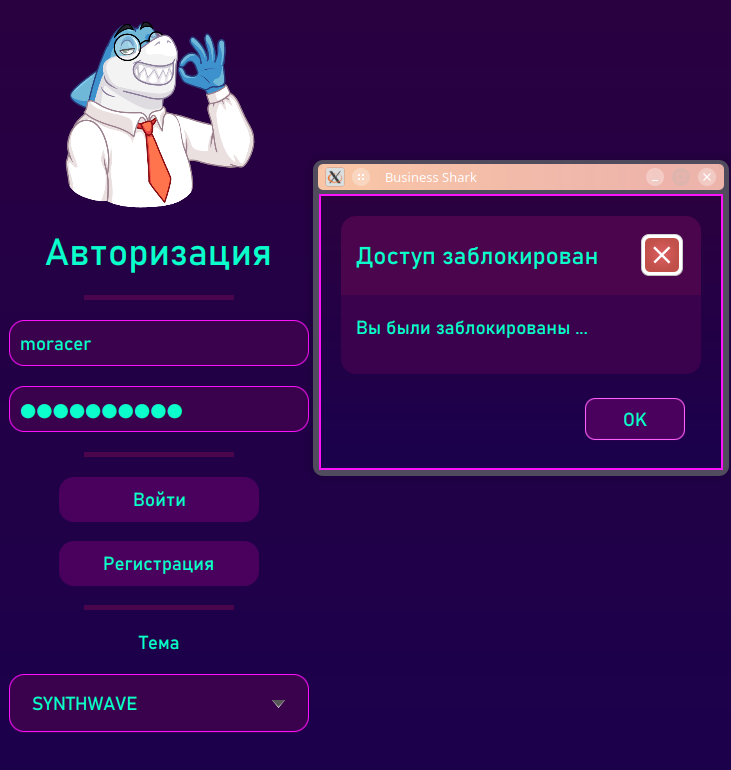


Рисунок 9.6 – Обработка попытки авторизации со стороны заблокированного аккаунта

Была протестирована ситуация попытки продать больше акций, чем имеется в портфолио пользователя, например, попытка продать 100 акций при наличии только 50. При вводе некорректного количества в поле «Кол-во акций на продажу» на экране управления портфолио система выполняет проверку на стороне клиента и отображает диалоговое окно с сообщением: «Вы пытаетесь продать больше, чем есть у вас в портфолио» (рисунок 9.7). Проверка сравнивает введенное значение с данными в базе, предотвращая отправку некорректного запроса на сервер. Инцидент фиксируется в логах с указанием времени, пользователя и количества акций. Такой подход защищает целостность портфеля, предотвращает финансовые ошибки и обеспечивает четкую обратную связь.

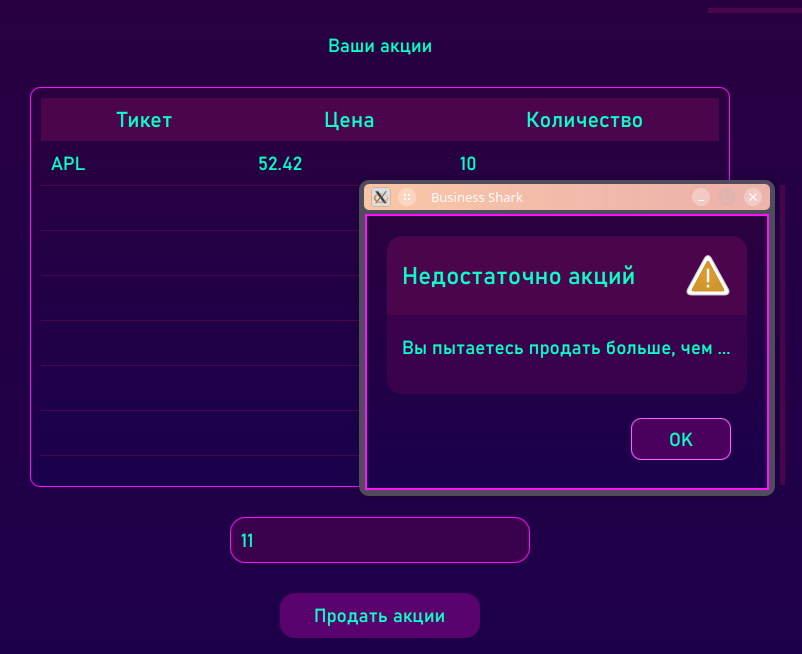


Рисунок 9.7 – Обработка попытки продажи большего количества акций, чем есть в портфолио

Еще одной протестированной ситуацией является попытка купить больше акций, чем доступно в системе, например, запрос на покупку 200 акций при наличии только 100. При вводе количества в поле «Кол-во акций на покупку» система проверяет доступное количество через запрос к базе данных. Если введенное значение превышает доступное, интерфейс отображает сообщение: «Запрошенное количество превышает доступное к покупке» (рисунок 9.8). Проверка выполняется на серверной стороне, гарантируя актуальность данных. Событие логируется с указанием пользователя, тикета акции и времени. Этот механизм предотвращает некорректные транзакции, поддерживая рыночную целостность и минимизируя риск сбоев, что важно для финансовых систем, требующих надежного управления активами.

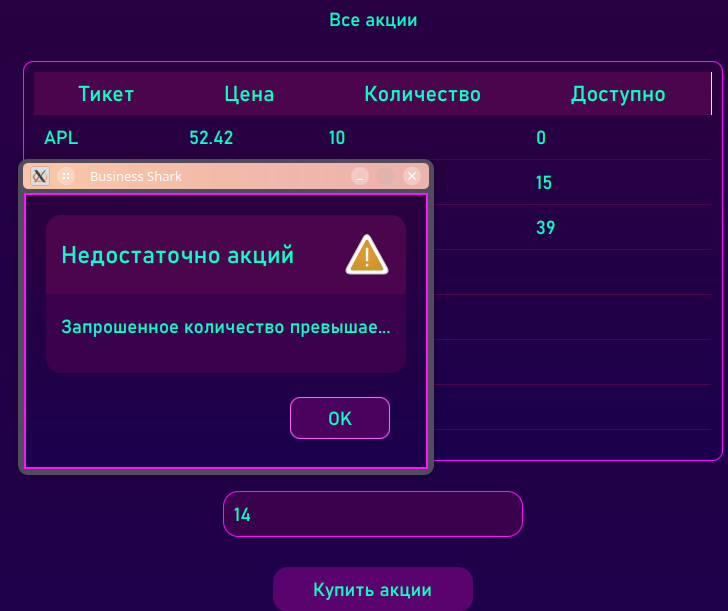


Рисунок 9.8 – Обработка попытки покупки большего количества акций, чем есть в системе

Также протестирована ситуация попытки купить больше акций, чем позволяет баланс пользователя, например, покупка акций на $15,000 при балансе $10,000. При отправке запроса на покупку система сравнивает стоимость акций с балансом пользователя через серверное приложение. Если средств недостаточно, отображается сообщение: «Недостаточно средств на счете» (рисунок 9.9). Проверка использует базу данных для доступа к данным, а инцидент фиксируется в логах с указанием суммы и пользователя. Это предотвращает финансовые нарушения, обеспечивает прозрачность и информирует пользователя о необходимости корректировки.

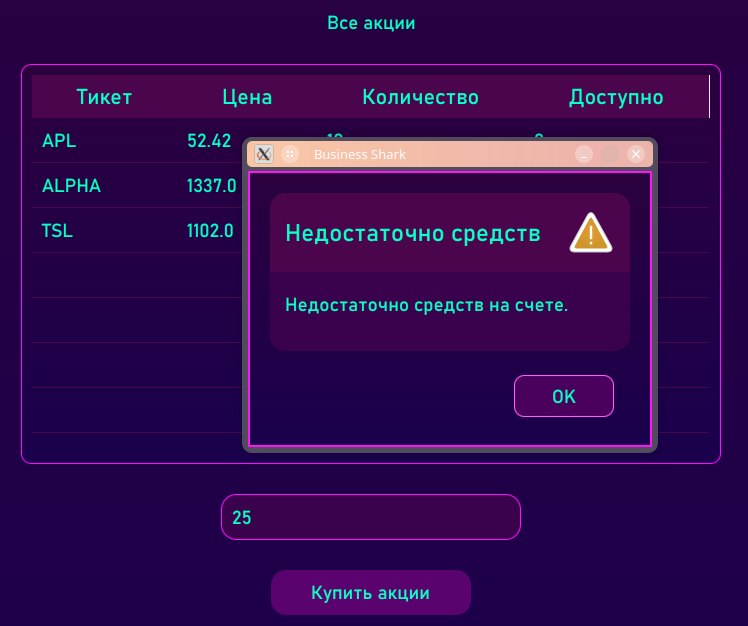


Рисунок 9.9 – Обработка попытки покупки акций на большую сумму, чем есть на балансе

В ходе тестирования была проанализирована работа системы логгирования, реализованная с использованием библиотеки Log4j, которая фиксирует все значимые события, включая исключительные ситуации и ошибки. При возникновении любых операций, таких как авторизация, попытка некорректного ввода или сбой подключения, система записывает в лог-файл подробную информацию: временную метку, имя пользователя (если применимо), тип события и описание ошибки. На рисунке 9.10 можно увидеть пример информации, отображаемой в логах. Логи хранятся на сервере и клиенте (в зависимости от логгера) в текстовом формате, обеспечивая доступ к ним через файловую систему. Этот механизм позволяет отслеживать подозрительную активность, диагностировать сбои и проводить аудит, что повышает безопасность и надежность системы. В финансовой предметной области логгирование критично для обеспечения прозрачности и предотвращения потенциальных угроз.

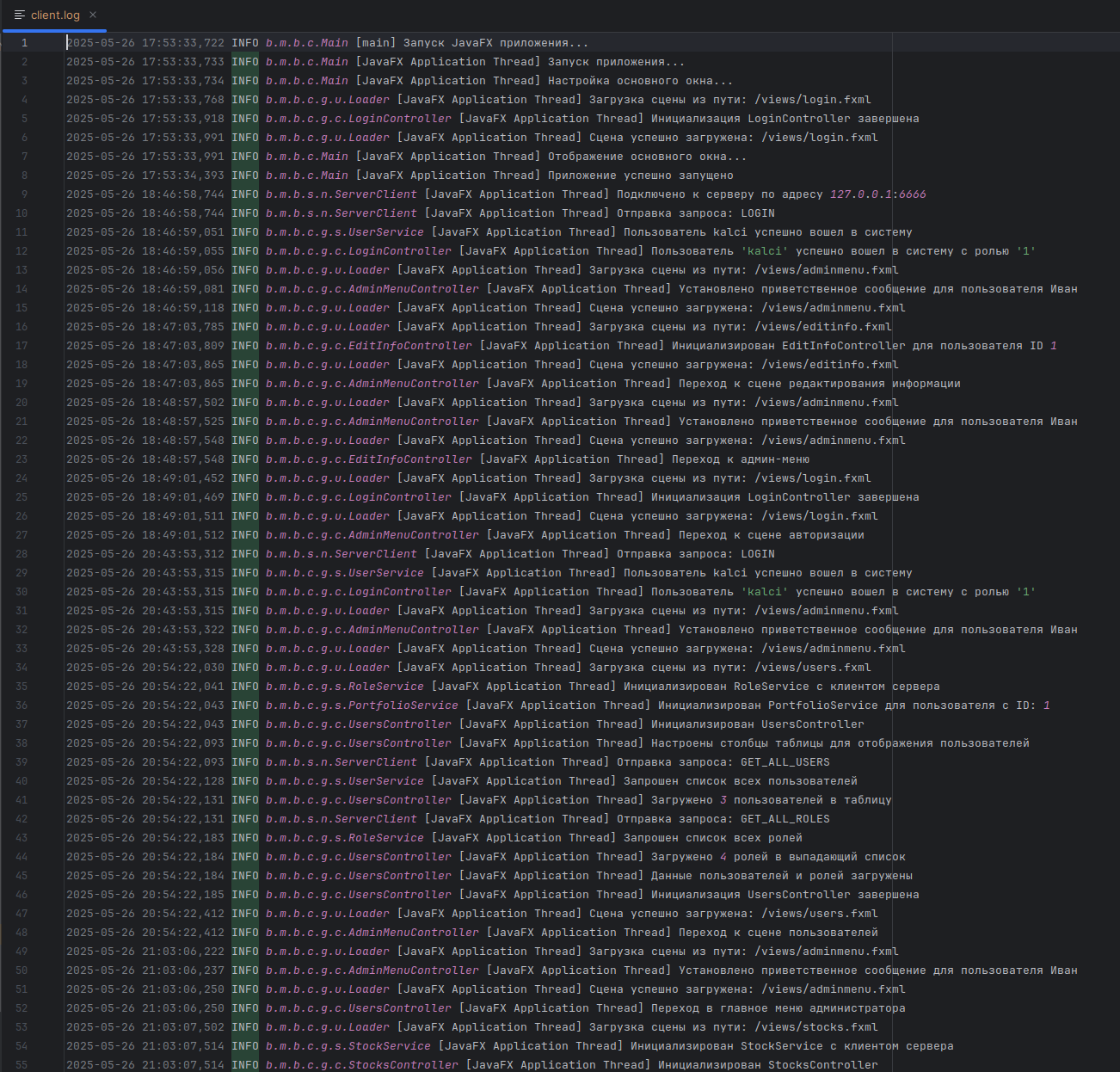


Рисунок 9.10 – Пример логов работы программного средства

Результаты тестирования подтверждают высокую надежность и устойчивость разработанного программного средства для управления акциями инвестиционного фонда к различным исключительным ситуациям. Проведенные тесты охватили ключевые сценарии, такие как некорректный ввод логина или пароля, пустые поля авторизации, отсутствие подключения к серверу, неверный формат данных, неподтвержденная регистрация, блокировка пользователя, попытки некорректных операций с акциями и процесс логгирования. В каждом случае система демонстрировала эффективную обработку ошибок, обеспечивая защиту данных и предоставляя пользователю четкую обратную связь через графический интерфейс. Механизмы валидации, реализованные с использованием Hibernate и регулярных выражений, предотвращают некорректные транзакции, а библиотека Log4j фиксирует события для аудита и диагностики. Тестирование выявило способность системы поддерживать целостность данных, оперативно реагировать на нештатные ситуации и сохранять доверие пользователей. Таким образом, программное средство доказало свою пригодность для эксплуатации, соответствуя требованиям надежности и функциональности, необходимым для управления инвестиционными процессами.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было создано программное средство для дистанционного управления опрыскивателем растений, автоматизирующее процессы мониторинга полевых данных, расчёта зон обработки, отправки команд технике и формирования отчётов. Серверная часть, реализованная на C# в виде консольного приложения, обеспечивает многопоточную обработку запросов, а Unity-клиент, построенный на C# и Unity UI, отвечает за визуализацию интерфейса. Хранение данных в PostgreSQL посредством Npgsql и использование BCrypt.Net для безопасного хэширования паролей гарантируют надёжность и защиту информации. В совокупности получился инструмент, который оптимизирует агротехнологические операции, снижает риски и повышает прозрачность взаимодействий, что критически важно в сфере сельскохозяйственного дистанционного управления.

Первый раздел был посвящён анализу предметной области управления опрыскивателями, в котором рассмотрены ключевые этапы: от выбора культур и учёта заболеваний до подбора препаратов и расчёта маршрутов. Были изучены существующие решения в агроавтоматизации, выявлена необходимость в собственном программном комплексе, способном работать в режиме реального времени с учётом геоданных и параметров обработки. Результатом аналитической части стало обоснование задачи разработки клиент-серверного приложения, способного удовлетворить требования точности, скорости и безопасности.

Во втором разделе сформулированы основные задачи создания многопоточного серверного приложения и Unity-клиента. Описаны требования к функциональности: единый протокол обмена по TCP/IP с использованием текстовых сообщений, эффективные алгоритмы расчёта зон обработки, хранение справочников и сессий в PostgreSQL, а также система авторизации с разграничением прав. Выбран стек технологий – C# и .NET для бизнес-логики, Unity UI для клиентского интерфейса, Npgsql для доступа к базе. Это послужило надежным фундаментом для масштабируемости, производительности и удобства дальнейшей реализации.

Третий раздел посвятили функциональному моделированию с помощью методологии IDEF0. Построена многоуровневая модель процесса расчёта зон: от этапа приёма координатных точек и параметров работы до объединения пересекающихся кругов, вычисления рабочих зон и формирования результатов. В декомпозиции на несколько уровней были явно показаны входящие данные, управляющие сигналы и механизмы (сервер, алгоритм расчёта, отправка ответа), что упростило последующую реализацию и позволило учесть требования по прозрачности и проверке соответствия регламентам агротехнических процедур.

Четвёртый раздел содержит информационную модель IDEF1X, описывающую структуру реляционной базы данных. Нормализация до третьей нормальной формы исключила избыточность и обеспечила целостность данных. Это позволило эффективно хранить и обрабатывать запросы, связанные с проверкой прав доступа, чтением справочников и сохранением результатов расчёта, что является ключевым для устойчивой работы системы.

Пятый раздел посвящён детальному описанию алгоритмов серверной бизнес-логики: процесса авторизации, валидации входных данных, расчёта зон обработки и формирования отчетов. Представлены схемы алгоритмов, наглядно демонстрирующие последовательность шагов: чтение запроса клиента, проверка прав, выбор справочных данных, группировка точек, объединение пересекающихся зон, вычисление объёмов воды и отправка ответа. Такой подход минимизирует вероятность ошибок при реализации и обеспечивает прозрачность всех операций.

Шестой раздел охватывает UML-моделирование системы. На диаграмме вариантов использования показаны роли «Администратор» и «Пользователь», их взаимодействие с системой при настройке справочников и запуске расчётов. Диаграммы классов детализировали структуру клиентской и серверной частей, описали контроллеры, сервисы, модели данных и утилиты. Диаграмма последовательностей раскрыла динамику вызовов при аутентификации и расчёте зон, диаграмма состояний продемонстрировала переходы жизненного цикла сессии расчёта, а диаграмма развёртывания отразила физическую архитектуру: сервер с консольным приложением, клиент-машину с Unity-приложением и сервер базы данных, соединённые через TCP/IP и Npgsql. Все эти модели обеспечили модульность, масштабируемость системы и подтвердили обоснованность архитектурного решения.

Седьмой раздел содержал руководство пользователя, в котором описано начальное развертывание: создание базы данных, запуск серверного и клиентского приложений, регистрация пользователей. Интерфейсы администратора, позволяющие редактировать справочники, управлять учётными записями и настраивать параметры проекта, и интерфейсы оператора, отвечающие за выбор участка, ввод точек и запуск расчёта зон, реализованы с помощью Unity UI и собственных скриптов анимаций UIScaling и UISliding. Это обеспечило интуитивность, безопасность и минимизацию рисков ошибок при эксплуатации.

Восьмой раздел предоставил результаты тестирования, в ходе которого была проверена устойчивость системы к различным исключительным ситуациям: некорректному вводу координат, отсутствию связи с сервером, попыткам неавторизованных действий и обработке ошибок в реальном времени. С помощью Npgsql-транзакций и обработки исключений в C# удалось обеспечить целостность данных и прозрачность операций, что является обязательным в условиях удалённого управления агротехническими задачами.

В результате разработанное программное средство успешно автоматизирует удалённое управление опрыскивателем растений, повышая эффективность расчётов, точность обработки полей и прозрачность взаимодействия между оператором и сервером. Система готова к внедрению в агропромышленных условиях и дальнейшему расширению, полностью соответствуя требованиям производственной сферы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Никитина, Т.В. Основы портфельного инвестирования / Т.В. Никитина. – Москва : Юрайт, 2018. – С. 137–140.

[2] Якушин, Д.И. Управление инвестиционным портфелем / Д.И. Якушин. – Москва : Директ-Медиа, 2022. – 120 с.

[3] A2is. Программы для управления инвестициями [Электронный ресурс]. – Режим доступа: https://a2is.ru/catalog/programmy-dlya-upravleniya-investitsiyami. – Дата доступа: 24.03.2025.

[4] Шилдт, Г. Java. Полное руководство / Г. Шилдт. – Москва : Вильямс, 2018. – 1344 с.

[5] Лузанов, П. PostgreSQL. Профессиональный SQL / П. Лузанов, Е. Рогов, И. Левшин. – Москва : Postgres Professional, 2020. – 320 с.

[6] Бауэр, К. Java Persistence API и Hibernate / К. Бауэр, Г. Кинг, Г. Грегори. – Москва : ДМК-Пресс, 2017. – 652 с.

[7] JavaFX Documentation [Электронный ресурс]. – Режим доступа: https://openjfx.io/. – Дата доступа: 16.04.2025.

[8] Apache Log4j [Электронный ресурс]. – Режим доступа: https://logging.apache.org/log4j/2.x/. – Дата доступа: 17.04.2025.

[9] Черемных, С.В. Моделирование и анализ систем / С.В. Черемных, И.О. Семенов, В.С. Ручкин. – Москва : Финансы и статистика, 2006. – 188 с.

[10] IDEF1X [Электронный ресурс]. – Режим доступа: https://www.cfin.ru/vernikov/idef/idef1x.shtml. – Дата доступа: 22.04.2025.

[11] Project Lombok [Электронный ресурс]. – Режим доступа: https://projectlombok.org/. – Дата доступа: 23.04.2025.

[12] jBCrypt [Электронный ресурс]. – Режим доступа: https://www.mindrot.org/projects/jBCrypt/. – Дата доступа: 23.04.2025.

[13] OpenPDF [Электронный ресурс]. – Режим доступа: https://github.com/LibrePDF/OpenPDF. – Дата доступа: 23.04.2025.

[14] Apache POI [Электронный ресурс]. – Режим доступа: https://poi.apache.org/. – Дата доступа: 23.04.2025.

[15] Гамма, Э. Шаблоны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. – Санкт-Петербург : Питер, 2010. – 368 с.

[16] Фаулер, М. UML. Краткое руководство / М. Фаулер. – Москва : ДМК Пресс, 2004. – 192 с.

[17] Ларман, К. Применение UML и шаблонов / К. Ларман. – Москва : Вильямс, 2006. – 624 с.

# ПРИЛОЖЕНИЕ А (обязательное) Отчет о проверке на заимствования в системе «Антиплагиат»

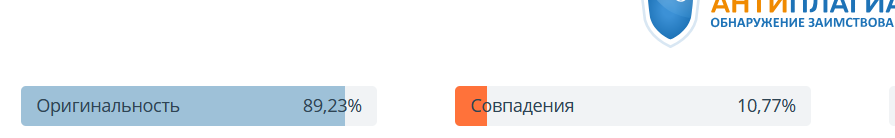


Рисунок А.1 – Проверка на Антиплагиат

# ПРИЛОЖЕНИЕ Б (обязательное) Листинг алгоритмов, реализующих бизнес-логику

Жизненный цикл клиент-серверного взаимодействия (класс ServerClient.java):

package by.mrtorex.businessshark.server.network;

import by.mrtorex.businessshark.server.exceptions.NoConnectionException;

import by.mrtorex.businessshark.server.model.entities.User;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import lombok.Getter;

import lombok.Setter;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.util.ResourceBundle;

/\*\*

\* Класс для управления подключением клиента к серверу.

\* Реализует паттерн Singleton для обеспечения единственного экземпляра клиента.

\*/

public class ServerClient {

private static final Logger logger = LogManager.getLogger(ServerClient.class);

private static ServerClient instance;

private Socket socket;

private ObjectOutputStream out;

private ObjectInputStream in;

@Getter

@Setter

private static User currentUser;

/\*\*

\* Приватный конструктор для инициализации подключения к серверу.

\*

\* @throws NoConnectionException если не удалось подключиться к серверу

\*/

private ServerClient() throws NoConnectionException {

connect(); // Если подключение провалится, выбросится исключение NoConnectionException

}

/\*\*

\* Метод для получения единственного экземпляра класса ServerClient.

\*

\* @return единственный экземпляр ServerClient

\* @throws NoConnectionException если не удалось подключиться к серверу

Продолжение приложения Б

\*/

public static synchronized ServerClient getInstance() throws NoConnectionException {

if (instance == null) {

try {

instance = new ServerClient();

} catch (Exception e) {

logger.error("Ошибка при создании экземпляра ServerClient: {}", e.getMessage());

throw new NoConnectionException("Не удалось создать экземпляр ServerClient");

}

}

return instance;

}

/\*\*

\* Метод для подключения к серверу.

\*

\* @throws NoConnectionException если не удалось подключиться к серверу

\*/

private void connect() throws NoConnectionException {

ResourceBundle bundle = ResourceBundle.getBundle("server");

String serverAddress = bundle.getString("SERVER\_IP");

int serverPort = Integer.parseInt(bundle.getString("SERVER\_PORT"));

try {

socket = new Socket(serverAddress, serverPort);

out = new ObjectOutputStream(socket.getOutputStream());

in = new ObjectInputStream(socket.getInputStream());

logger.info("Подключено к серверу по адресу {}:{}", serverAddress, serverPort);

} catch (IOException e) {

throw new NoConnectionException("Не удалось подключиться к серверу по адресу " + serverAddress + ":" + serverPort);

}

}

/\*\*

\* Метод для отключения от сервера.

\*/

public void disconnect() {

try {

if (in != null) in.close();

if (out != null) out.close();

if (socket != null) socket.close();

instance = null;

logger.info("Отключено от сервера.");

} catch (IOException e) {

logger.error("Ошибка при отключении от сервера: {}", e.getMessage());

}

}

/\*\*

\* Отправляет запрос на сервер и возвращает ответ.

Продолжение приложения Б

\*

\* @param request запрос для отправки

\* @return ответ от сервера

\*/

public Response sendRequest(Request request) {

try {

logger.info("Отправка запроса: {}", request.getOperation());

out.writeObject(request);

out.flush();

return processResponse();

} catch (IOException e) {

logger.error("Ошибка при отправке запроса: {}", e.getMessage());

return null;

}

}

/\*\*

\* Обрабатывает ответ от сервера.

\*

\* @return ответ от сервера

\*/

private Response processResponse() {

try {

return (Response) in.readObject();

} catch (IOException | ClassNotFoundException e) {

logger.error("Ошибка при обработке ответа: {}", e.getMessage());

return null;

}

}

}

Процесс авторизации, регистрации, обновления и удаления пользователей (класс UserController.java):

package by.mrtorex.businessshark.server.controllers;

import by.mrtorex.businessshark.server.exceptions.ResponseException;

import by.mrtorex.businessshark.server.model.entities.User;

import by.mrtorex.businessshark.server.network.Request;

import by.mrtorex.businessshark.server.network.Response;

import by.mrtorex.businessshark.server.serializer.Deserializer;

import by.mrtorex.businessshark.server.serializer.Serializer;

import by.mrtorex.businessshark.server.services.PersonService;

import by.mrtorex.businessshark.server.services.RoleService;

import by.mrtorex.businessshark.server.services.UserService;

import by.mrtorex.businessshark.server.utils.Pair;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.util.Objects;

/\*\*

\* Контроллер для управления операциями с пользователями.

\* Обеспечивает аутентификацию, регистрацию и управление пользователями.

Продолжение приложения Б

\*/

public class UserController {

private static final Logger logger = LogManager.getLogger(UserController.class);

private final UserService userService;

private final PersonService personService;

private final RoleService roleService;

/\*\*

\* Конструктор с внедрением зависимостей сервисов.

\*

\* @param userService сервис работы с пользователями

\* @param personService сервис работы с персональными данными

\* @param roleService сервис работы с ролями

\*/

public UserController(UserService userService, PersonService personService, RoleService roleService) {

this.userService = userService;

this.personService = personService;

this.roleService = roleService;

logger.info("Инициализирован UserController с внешними сервисами");

}

/\*\*

\* Конструктор по умолчанию.

\*/

public UserController() {

this.userService = new UserService();

this.personService = new PersonService();

this.roleService = new RoleService();

logger.info("Инициализирован UserController со стандартными сервисами");

}

/\*\*

\* Выполняет аутентификацию пользователя.

\*

\* @param request запрос с данными для входа

\* @return ответ с результатом аутентификации

\*/

public Response login(Request request) {

try {

User user = (User) new Deserializer().extractData(request);

logger.info("Попытка входа пользователя: {}", user.getUsername());

User existingUser = userService.login(user);

String loggedInUser = Serializer.toJson(existingUser);

logger.info("Успешный вход пользователя: {}", user.getUsername());

return new Response(true, "Аутентификация успешна", loggedInUser);

} catch (ClassCastException e) {

logger.error("Некорректный формат данных пользователя", e);

return new Response(false, "Некорректные данные для входа", null);

} catch (ResponseException e) {

logger.warn("Ошибка аутентификации: {}", e.getMessage());

Продолжение приложения Б

return new Response(false, e.getMessage(), null);

} catch (Exception e) {

logger.error("Неожиданная ошибка при аутентификации", e);

return new Response(false, "Внутренняя ошибка сервера", null);

}

}

/\*\*

\* Регистрирует нового пользователя.

\*

\* @param request запрос с данными нового пользователя

\* @return ответ с результатом регистрации

\*/

public Response register(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof User user)) {

logger.warn("Попытка регистрации с некорректными данными");

return new Response(false, "Некорректные данные пользователя", null);

}

logger.info("Попытка регистрации пользователя: {}", user.getUsername());

User registeredUser = userService.register(user, personService, roleService);

String registeredUserJson = Serializer.toJson(registeredUser);

logger.info("Успешная регистрация пользователя: {}", user.getUsername());

return new Response(true, "Регистрация успешна", registeredUserJson);

} catch (IllegalArgumentException e) {

logger.error("Ошибка формата данных при регистрации", e);

return new Response(false, "Ошибка формата данных", null);

} catch (ResponseException e) {

logger.error("Ошибка регистрации: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Получает список всех пользователей.

\*

\* @return ответ со списком пользователей

\*/

public Response getAllUsers() {

try {

logger.info("Запрос списка всех пользователей");

String usersJson = Serializer.toJson(userService.findAllEntities());

logger.info("Получено {} пользователей", userService.findAllEntities().size());

return new Response(true, "Список пользователей получен", usersJson);

Продолжение приложения Б

} catch (Exception e) {

logger.error("Ошибка получения списка пользователей", e);

return new Response(false, "Ошибка при получении списка пользователей", null);

}

}

/\*\*

\* Удаляет пользователя.

\*

\* @param request запрос с логином пользователя

\* @return результат операции

\*/

public Response deleteUser(Request request) {

try {

String login = (String) new Deserializer().extractData(request);

logger.info("Попытка удаления пользователя: {}", login);

User foundUser = userService.findByUsername(login);

if (foundUser == null) {

logger.warn("Пользователь для удаления не найден: {}", login);

return new Response(false, "Пользователь не найден", null);

}

personService.deleteEntity(foundUser.getPerson());

logger.info("Пользователь успешно удален: {}", login);

return new Response(true, "Пользователь удален", null);

} catch (ClassCastException e) {

logger.error("Некорректный формат логина пользователя", e);

return new Response(false, "Некорректный формат данных", null);

} catch (ResponseException e) {

logger.error("Ошибка удаления пользователя: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Обновляет данные пользователя.

\*

\* @param request запрос с данными для обновления

\* @return результат операции

\*/

public Response updateEntity(Request request) {

try {

Object extractData = new Deserializer().extractData(request);

if (!(extractData instanceof Pair<?,?> pair) ||

!(pair.getKey() instanceof User userToUpdate) ||

!(pair.getValue() instanceof User userThatOperate)) {

logger.warn("Некорректный формат данных для обновления пользователя");

return new Response(false, "Некорректные данные пользователя", null);

Продолжение приложения Б

}

logger.info("Попытка обновления пользователя ID {} (инициатор: {})",

userToUpdate.getId(), userThatOperate.getId());

User existingUserToUpdate = userService.findEntity(userToUpdate.getId());

User existingUserThatOperate = userService.findEntity(userThatOperate.getId());

if (existingUserToUpdate == null || existingUserThatOperate == null) {

logger.warn("Один из пользователей не существует (обновляемый: {}, инициатор: {})",

userToUpdate.getId(), userThatOperate.getId());

return new Response(false, "Один из пользователей не существует", null);

}

userService.updateEntity(userToUpdate, personService);

if (Objects.equals(userToUpdate.getId(), userThatOperate.getId())) {

logger.info("Пользователь обновил свои данные: {}", userToUpdate.getId());

return new Response(true, "Данные пользователя обновлены. Требуется повторный вход.", null);

} else {

logger.info("Администратор обновил данные пользователя: {}", userToUpdate.getId());

return new Response(true, "Данные пользователя обновлены", null);

}

} catch (ResponseException e) {

logger.error("Ошибка обновления пользователя: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Получает данные пользователя.

\*

\* @param request запрос с логином пользователя

\* @return данные пользователя

\*/

public Response readEntity(Request request) {

try {

String username = (String) new Deserializer().extractData(request);

logger.info("Запрос данных пользователя: {}", username);

User user = userService.findByUsername(username);

if (user == null) {

logger.warn("Пользователь не найден: {}", username);

return new Response(false, "Пользователь не найден", null);

}

String userJson = Serializer.toJson(user);

Продолжение приложения Б

logger.info("Данные пользователя получены: {}", username);

return new Response(true, "Данные пользователя получены", userJson);

} catch (ClassCastException e) {

logger.error("Некорректный формат логина пользователя", e);

return new Response(false, "Некорректный формат данных", null);

} catch (ResponseException e) {

logger.error("Ошибка получения данных пользователя: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

}

Процесс покупки и продажи акций (класс PortfolioController.java):

package by.mrtorex.businessshark.server.controllers;

import by.mrtorex.businessshark.server.exceptions.ResponseException;

import by.mrtorex.businessshark.server.model.entities.Stock;

import by.mrtorex.businessshark.server.network.Request;

import by.mrtorex.businessshark.server.network.Response;

import by.mrtorex.businessshark.server.serializer.Deserializer;

import by.mrtorex.businessshark.server.serializer.Serializer;

import by.mrtorex.businessshark.server.services.PortfolioService;

import by.mrtorex.businessshark.server.utils.Pair;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.util.List;

/\*\*

\* Контроллер для управления портфелями пользователей.

\* Обеспечивает операции с акциями пользователей и их счетами.

\*/

public class PortfolioController {

private static final Logger logger = LogManager.getLogger(PortfolioController.class);

private final PortfolioService portfolioService;

/\*\*

\* Конструктор с внедрением зависимости сервиса.

\*

\* @param portfolioService сервис для работы с портфелями

\*/

public PortfolioController(PortfolioService portfolioService) {

this.portfolioService = portfolioService;

logger.info("Инициализирован PortfolioController с внешним сервисом");

}

/\*\*

\* Конструктор по умолчанию.

\*/

public PortfolioController() {

this.portfolioService = new PortfolioService();

Продолжение приложения Б

logger.info("Инициализирован PortfolioController со стандартным сервисом");

}

/\*\*

\* Добавляет акцию в портфель пользователя.

\*

\* @param request запрос с данными акции и пользователя

\* @return результат операции

\*/

public Response addUserStock(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Pair<?,?> data) ||

!(data.getKey() instanceof Pair<?, ?> pair) ||

!(pair.getKey() instanceof Stock stock) ||

!(pair.getValue() instanceof Integer amount) ||

!(data.getValue() instanceof Integer userId)) {

logger.warn("Некорректный формат данных для добавления акции");

return new Response(false, "Некорректный формат данных акции", null);

}

portfolioService.saveEntity(new Pair<>(stock, amount), userId);

logger.info("Добавлена акция ID {} для пользователя ID {}, количество: {}",

stock.getId(), userId, amount);

return new Response(true, "Акция успешно добавлена", null);

} catch (ResponseException e) {

logger.error("Ошибка добавления акции: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Получает акцию из портфеля пользователя.

\*

\* @param request запрос с ID пользователя и акции

\* @return данные об акции

\*/

public Response getUserStock(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Pair<?, ?> data) ||

!(data.getValue() instanceof Integer userId) ||

!(data.getKey() instanceof Integer stockId)) {

logger.warn("Некорректный формат данных для получения акции");

return new Response(false, "Некорректный формат запроса", null);

}

Pair<Stock, Integer> result = portfolioService.findEntity(userId, stockId);

Продолжение приложения Б

String json = Serializer.toJson(result);

logger.info("Получена акция ID {} для пользователя ID {}", stockId, userId);

return new Response(true, "Данные акции получены", json);

} catch (ResponseException e) {

logger.error("Ошибка получения акции: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Обновляет данные акции в портфеле пользователя.

\*

\* @param request запрос с обновленными данными

\* @return результат операции

\*/

public Response updateUserStock(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Pair<?,?> data) ||

!(data.getKey() instanceof Pair<?, ?> pair) ||

!(pair.getKey() instanceof Stock stock) ||

!(pair.getValue() instanceof Integer amount) ||

!(data.getValue() instanceof Integer userId)) {

logger.warn("Некорректный формат данных для обновления акции");

return new Response(false, "Некорректный формат данных", null);

}

portfolioService.updateEntity(new Pair<>(stock, amount), userId);

logger.info("Обновлена акция ID {} для пользователя ID {}, новое количество: {}",

stock.getId(), userId, amount);

return new Response(true, "Акция успешно обновлена", null);

} catch (ResponseException e) {

logger.error("Ошибка обновления акции: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Удаляет акцию из портфеля пользователя.

\*

\* @param request запрос с ID пользователя и акции

\* @return результат операции

\*/

public Response deleteUserStock(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Object[] arr) || arr.length != 2 ||

!(arr[0] instanceof Integer userId) ||

!(arr[1] instanceof Integer stockId)) {

Продолжение приложения Б

logger.warn("Некорректный формат данных для удаления акции");

return new Response(false, "Некорректный формат запроса", null);

}

portfolioService.deleteEntity(userId, stockId);

logger.info("Удалена акция ID {} из портфеля пользователя ID {}", stockId, userId);

return new Response(true, "Акция успешно удалена", null);

} catch (ResponseException e) {

logger.error("Ошибка удаления акции: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

\* Получает все акции пользователя.

\*

\* @param request запрос с ID пользователя

\* @return список акций

\*/

public Response getAllUserStocks(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Integer userId)) {

logger.warn("Некорректный формат ID пользователя");

return new Response(false, "Некорректный ID пользователя", null);

}

List<Pair<Stock, Integer>> stocks = portfolioService.findAllUserStocks(userId);

String json = Serializer.toJson(stocks);

logger.info("Получено {} акций для пользователя ID {}", stocks.size(), userId);

return new Response(true, "Список акций получен", json);

} catch (Exception e) {

logger.error("Ошибка получения списка акций", e);

return new Response(false, "Ошибка при получении списка акций", null);

}

}

/\*\*

\* Получает все связи пользователей и акций.

\*

\* @return список всех связей

\*/

public Response getAllUserStockIds() {

try {

List<Pair<Integer, Integer>> all = portfolioService.findAll();

String json = Serializer.toJson(all);

logger.info("Получено {} связей пользователей и акций", all.size());

return new Response(true, "Список связей получен", json);

} catch (Exception e) {

Продолжение приложения Б

logger.error("Ошибка получения списка связей", e);

return new Response(false, "Ошибка при получении списка связей", null);

}

}

/\*\*

\* Получает баланс счета пользователя.

\*

\* @param request запрос с ID пользователя

\* @return баланс счета

\*/

public Response getAccount(Request request) {

try {

Integer userId = (Integer) new Deserializer().extractData(request);

Double account = portfolioService.getAccount(userId);

String json = Serializer.toJson(account);

logger.info("Получен баланс для пользователя ID {}: {}", userId, account);

return new Response(true, "Баланс получен", json);

} catch (Exception e) {

logger.error("Ошибка получения баланса", e);

return new Response(false, "Ошибка при получении баланса", null);

}

}

/\*\*

\* Устанавливает баланс счета пользователя.

\*

\* @param request запрос с ID пользователя и новым балансом

\* @return результат операции

\*/

public Response setAccount(Request request) {

try {

Object extractedData = new Deserializer().extractData(request);

if (!(extractedData instanceof Pair<?,?> pair) ||

!(pair.getKey() instanceof Integer userId) ||

!(pair.getValue() instanceof Double account)) {

logger.warn("Некорректный формат данных для установки баланса");

return new Response(false, "Некорректный формат данных", null);

}

portfolioService.setAccount(userId, account);

logger.info("Установлен баланс {} для пользователя ID {}", account, userId);

return new Response(true, "Баланс успешно обновлен", null);

} catch (ResponseException e) {

logger.error("Ошибка обновления баланса: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

}

}

/\*\*

Продолжение приложения Б

\* Получает доступное количество акции.

\*

\* @param request запрос с ID акции

\* @return доступное количество

\*/

public Response getStockAvailableAmount(Request request) {

try {

Integer stockId = (Integer) new Deserializer().extractData(request);

int amount = portfolioService.getStockAvailableAmount(stockId);

String json = Serializer.toJson(amount);

logger.info("Получено доступное количество акции ID {}: {}", stockId, amount);

return new Response(true, "Доступное количество получено", json);

} catch (Exception e) {

logger.error("Ошибка получения доступного количества акции", e);

return new Response(false, "Ошибка при получении количества акций", null);

}

}

}

# ПРИЛОЖЕНИЕ В (обязательное) Листинг основных элементов программы

Главный класс серверной части программы (класс Server.java):

package by.mrtorex.businessshark.server.network;

import by.mrtorex.businessshark.server.config.SessionConfig;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.ResourceBundle;

import java.util.concurrent.\*;

/\*\*

\* Основной класс сервера, отвечающий за прием подключений клиентов, мониторинг активности и подключение к БД.

\*/

public class Server {

private static final Logger logger = LogManager.getLogger(Server.class);

private static volatile int clientCount = 0;

private static volatile long lastClientConnectedTime = System.currentTimeMillis();

private static ServerSocket serverSocket;

private static ExecutorService clientExecutor;

private static ScheduledExecutorService monitorExecutor;

private static volatile boolean running = true;

/\*\*

\* Точка входа в приложение.

\*/

public static void main(String[] args) {

Runtime.getRuntime().addShutdownHook(new Thread(Server::shutdown, "Shutdown-Hook"));

startServer();

}

/\*\*

\* Запускает сервер, инициализирует мониторинг и подключение к БД.

\*/

private static void startServer() {

ResourceBundle bundle = ResourceBundle.getBundle("server");

int serverPort;

try {

serverPort = Integer.parseInt(bundle.getString("SERVER\_PORT"));

} catch (NumberFormatException e) {

logger.error("Неверный формат порта сервера в конфигурационном файле", e);

throw new IllegalStateException("Невозможно запустить сервер");

}

Продолжение приложения В

clientExecutor = Executors.newCachedThreadPool();

monitorExecutor = Executors.newSingleThreadScheduledExecutor();

try {

serverSocket = new ServerSocket(serverPort);

logger.info("Сервер запущен на порте {}", serverPort);

startMonitoring();

connectToDatabase();

while (running) {

try {

Socket client = serverSocket.accept();

incrementClientCount();

lastClientConnectedTime = System.currentTimeMillis();

logger.info("Клиент подключился. Текущее количество клиентов: {}", clientCount);

clientExecutor.submit(new ClientThread(client));

} catch (IOException e) {

if (running) {

logger.error("Ошибка при приеме клиента", e);

}

}

}

} catch (IOException e) {

logger.error("Ошибка запуска сервера", e);

} finally {

shutdown();

}

}

/\*\*

\* Запускает мониторинг подключений и завершает сервер при длительном отсутствии клиентов.

\*/

private static void startMonitoring() {

ResourceBundle bundle = ResourceBundle.getBundle("server");

long monitoringInterval;

long shutdownTime;

try {

monitoringInterval = Long.parseLong(bundle.getString("MONITORING\_INTERVAL"));

shutdownTime = Long.parseLong(bundle.getString("SHUTDOWN\_TIME"));

} catch (NumberFormatException e) {

logger.error("Неверный формат интервалов мониторинга", e);

throw new IllegalStateException("Невозможно запустить мониторинг");

}

monitorExecutor.scheduleAtFixedRate(() -> {

long now = System.currentTimeMillis();

if (clientCount == 0 && (now - lastClientConnectedTime) >= shutdownTime) {

logger.info("Нет подключенных клиентов в течение {} мс. Завершение сервера.", shutdownTime);

shutdown();

} else {

Продолжение приложения В

logger.info("Текущее количество клиентов: {}", clientCount);

}

}, 0, monitoringInterval, TimeUnit.MILLISECONDS);

}

/\*\*

\* Потокобезопасное уменьшение счетчика клиентов.

\*/

public static synchronized void decrementClientCount() {

if (clientCount > 0) {

clientCount--;

logger.info("Клиент отключился. Оставшиеся клиенты: {}", clientCount);

} else {

logger.warn("Попытка уменьшить счетчик клиентов ниже нуля");

}

}

/\*\*

\* Потокобезопасное увеличение счетчика клиентов.

\*/

private static synchronized void incrementClientCount() {

clientCount++;

}

/\*\*

\* Асинхронное подключение к базе данных.

\*/

private static void connectToDatabase() {

new Thread(() -> {

logger.info("Попытка подключения к базе данных...");

try {

SessionConfig config = SessionConfig.getInstance();

logger.info("Соединение с БД установлено: {}", config);

} catch (Exception e) {

logger.error("Ошибка при подключении к БД: {}", e.getMessage());

}

}, "DB-Connection-Thread").start();

}

/\*\*

\* Корректное завершение всех ресурсов: сокета, потоков и планировщиков.

\*/

private static synchronized void shutdown() {

if (!running) return;

running = false;

logger.info("Начинается завершение работы сервера...");

try {

if (serverSocket != null && !serverSocket.isClosed()) {

serverSocket.close();

logger.info("Серверный сокет закрыт");

}

} catch (IOException e) {

logger.warn("Ошибка при закрытии серверного сокета", e);

}

Продолжение приложения В

if (monitorExecutor != null && !monitorExecutor.isShutdown()) {

monitorExecutor.shutdownNow();

logger.info("Мониторинг остановлен");

}

if (clientExecutor != null && !clientExecutor.isShutdown()) {

clientExecutor.shutdownNow();

logger.info("Потоки клиентов завершены");

}

logger.info("Сервер завершил работу.");

}

}

Класс серверного потока обработки клиентских подключений (класс ClientThread.java):

package by.mrtorex.businessshark.server.network;

import by.mrtorex.businessshark.server.controllers.\*;

import by.mrtorex.businessshark.server.enums.Operation;

import by.mrtorex.businessshark.server.exceptions.ResponseException;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

/\*\*

\* Поток обработки клиентских подключений.

\* Обрабатывает запросы от клиента и взаимодействует с контроллерами для выполнения операций.

\*/

public class ClientThread implements Runnable {

private static final Logger logger = LogManager.getLogger(ClientThread.class);

private final Socket clientSocket;

private final StockController stockController;

private final UserController userController;

private final CompanyController companyController;

private final RoleController roleController;

private final PortfolioController portfolioController;

/\*\*

\* Конструктор потока с инициализацией контроллеров.

\*

\* @param socket сокет клиента

\*/

public ClientThread(Socket socket) {

this.clientSocket = socket;

stockController = new StockController();

userController = new UserController();

companyController = new CompanyController();

roleController = new RoleController();

portfolioController = new PortfolioController();

Продолжение приложения В

logger.info("Инициализирован новый клиентский поток для сокета {}", socket);

}

/\*\*

\* Основной цикл обработки запросов клиента.

\* Принимает запросы, обрабатывает их и отправляет ответы.

\*/

@Override

public void run() {

try (

ObjectInputStream input = new ObjectInputStream(clientSocket.getInputStream());

ObjectOutputStream output = new ObjectOutputStream(clientSocket.getOutputStream())

) {

boolean keepRunning = true;

while (keepRunning) {

try {

Object obj = input.readObject();

if (!(obj instanceof Request request)) {

logger.warn("Получен некорректный объект вместо запроса от клиента");

Response errorResponse = new Response(false, "Получен некорректный объект запроса", null);

output.writeObject(errorResponse);

output.flush();

continue;

}

Response response = processRequest(request);

if (request.getOperation() == Operation.DISCONNECT) {

keepRunning = false;

}

output.writeObject(response);

output.flush();

} catch (IOException e) {

logger.error("Ошибка соединения с клиентом: {}", e.getMessage());

keepRunning = false;

} catch (ClassNotFoundException e) {

logger.error("Класс запроса не найден: {}", e.getMessage());

keepRunning = false;

} catch (Exception e) {

logger.error("Необработанная ошибка при обработке запроса", e);

keepRunning = false;

}

}

} catch (IOException e) {

logger.error("Ошибка инициализации потоков ввода-вывода клиента", e);

throw new RuntimeException("Ошибка инициализации клиентского потока", e);

} finally {

closeConnection();

}

Продолжение приложения В

}

/\*\*

\* Обрабатывает запрос, вызывая соответствующие методы контроллеров.

\*

\* @param request объект запроса от клиента

\* @return ответ на запрос

\*/

private Response processRequest(Request request) {

try {

return switch (request.getOperation()) {

case CREATE\_STOCK -> stockController.createStock(request);

case READ\_STOCK\_DATA -> stockController.getStockByTicker(request);

case UPDATE\_STOCK -> stockController.updateStock(request);

case DELETE\_STOCK -> stockController.deleteStock(request);

case CREATE\_COMPANY -> companyController.createCompany(request);

case READ\_COMPANY\_DATA -> companyController.getCompanyByName(request);

case UPDATE\_COMPANY -> companyController.updateCompany(request);

case DELETE\_COMPANY -> companyController.deleteCompany(request);

case READ\_USER -> userController.readEntity(request);

case DELETE\_USER -> userController.deleteUser(request);

case UPDATE\_USER -> userController.updateEntity(request);

case CREATE\_USER, REGISTER -> userController.register(request);

case GET\_ALL\_COMPANIES -> companyController.getAllCompanies();

case GET\_ALL\_USERS -> userController.getAllUsers();

case GET\_ALL\_STOCKS -> stockController.getAllStocks();

case GET\_ALL\_ROLES -> roleController.getAllRoles();

case GET\_ALL\_STOCKS\_WITH\_NO\_COMPANY -> stockController.getAllStocksWithNoCompany();

case GET\_COMPANY\_BY\_STOCK -> companyController.getCompanyByStock(request);

case GET\_STOCKS\_BY\_COMPANY -> companyController.getCompanyStocks(request);

case JOIN\_STOCK\_COMPANY -> companyController.addStockToCompany(request);

case SEPARATE\_STOCK\_COMPANY -> companyController.removeStockFromCompany(request);

case ADD\_USER\_STOCK -> portfolioController.addUserStock(request);

case GET\_USER\_STOCK -> portfolioController.getUserStock(request);

case UPDATE\_USER\_STOCK -> portfolioController.updateUserStock(request);

Продолжение приложения В

case DELETE\_USER\_STOCK -> portfolioController.deleteUserStock(request);

case GET\_ALL\_USER\_STOCKS -> portfolioController.getAllUserStocks(request);

case GET\_ALL\_USER\_STOCK\_IDS -> portfolioController.getAllUserStockIds();

case GET\_USER\_ACCOUNT -> portfolioController.getAccount(request);

case SET\_USER\_ACCOUNT -> portfolioController.setAccount(request);

case GET\_STOCK\_AVAILABLE\_AMOUNT -> portfolioController.getStockAvailableAmount(request);

case LOGIN -> userController.login(request);

case DISCONNECT -> new Response(true, "Отключение выполнено успешно", null);

};

} catch (ResponseException e) {

logger.warn("Ошибка при выполнении операции: {}", e.getMessage());

return new Response(false, e.getMessage(), null);

} catch (Exception e) {

logger.error("Внутренняя ошибка сервера при обработке запроса", e);

return new Response(false, "Внутренняя ошибка сервера", null);

}

}

/\*\*

\* Закрывает клиентское соединение и обновляет счетчик подключений сервера.

\*/

private void closeConnection() {

try {

clientSocket.close();

Server.decrementClientCount();

logger.info("Клиентское соединение закрыто");

} catch (IOException e) {

logger.error("Ошибка при закрытии клиентского соединения", e);

}

}

}

Класс запроса, передаваемый от клиента к серверу (класс Request.java):

package by.mrtorex.businessshark.server.network;

import by.mrtorex.businessshark.server.enums.Operation;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NonNull;

import lombok.RequiredArgsConstructor;

import java.io.Serializable;

/\*\*

Продолжение приложения В

\* Запрос, передаваемый от клиента к серверу.

\* Содержит операцию и связанные с ней данные.

\*/

@Data

@AllArgsConstructor

@RequiredArgsConstructor

public class Request implements Serializable {

/\*\*

\* Операция, которую необходимо выполнить.

\*/

@NonNull

private Operation operation;

/\*\*

\* Данные, необходимые для выполнения операции.

\*/

private String data;

}

Класс ответа сервера на запрос клиента (класс Response.java):

package by.mrtorex.businessshark.server.network;

import lombok.AllArgsConstructor;

import lombok.Data;

import java.io.Serializable;

/\*\*

\* Ответ сервера на запрос клиента.

\* Содержит информацию об успешности операции, сообщение и данные.

\*/

@Data

@AllArgsConstructor

public class Response implements Serializable {

/\*\*

\* Статус успешности выполнения операции.

\* true - операция выполнена успешно, false - произошла ошибка.

\*/

private boolean success;

/\*\*

\* Сообщение с описанием результата операции.

\*/

private String message;

/\*\*

\* Дополнительные данные, возвращаемые в ответе.

\*/

private String data;

}

Базовый интерфейс для DAO (Data Access Object) операций (класс DAO.java):

package by.mrtorex.businessshark.server.interfaces;

Продолжение приложения В

import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.SessionFactory;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.util.List;

/\*\*

\* Базовый интерфейс для DAO (Data Access Object) операций.

\*

\* @param <T> тип сущности, с которой работает DAO

\*/

public interface DAO<T> {

Logger logger = LogManager.getLogger(DAO.class);

/\*\*

\* Сохраняет объект в базе данных.

\*

\* @param obj объект для сохранения

\*/

void save(T obj);

/\*\*

\* Обновляет объект в базе данных.

\*

\* @param obj объект для обновления

\*/

void update(T obj);

/\*\*

\* Удаляет объект из базы данных.

\*

\* @param obj объект для удаления

\*/

void delete(T obj);

/\*\*

\* Находит объект по идентификатору.

\*

\* @param id идентификатор объекта

\* @return найденный объект или null, если не найден

\*/

T findById(int id);

/\*\*

\* Получает все объекты данного типа из базы данных.

\*

\* @return список всех объектов

\*/

List<T> findAll();

/\*\*

\* Выполняет операцию в транзакции.

\*

\* @param sessionFactory фабрика сессий Hibernate

\* @param action действие для выполнения

\* @param obj объект, над которым выполняется действие

Продолжение приложения В

\* @throws RuntimeException если произошла ошибка при выполнении транзакции

\*/

default void executeTransaction(SessionFactory sessionFactory,

TransactionConsumer<T> action,

T obj) {

Transaction transaction = null;

try (Session session = sessionFactory.openSession()) {

transaction = session.beginTransaction();

logger.info("Начата транзакция для объекта {}", obj.getClass().getSimpleName());

action.accept(session, obj);

transaction.commit();

logger.info("Транзакция успешно завершена");

} catch (Exception e) {

logger.error("Ошибка выполнения транзакции", e);

if (transaction != null && transaction.isActive()) {

logger.warn("Откат активной транзакции");

transaction.rollback();

}

throw new RuntimeException("Ошибка выполнения транзакции: " + e.getMessage(), e);

}

}

/\*\*

\* Функциональный интерфейс для выполнения действий в транзакции.

\*

\* @param <T> тип объекта, над которым выполняется действие

\*/

@FunctionalInterface

interface TransactionConsumer<T> {

/\*\*

\* Выполняет действие с объектом в контексте сессии Hibernate.

\*

\* @param session сессия Hibernate

\* @param obj объект для обработки

\*/

void accept(Session session, T obj);

}

}

Универсальный интерфейс для сервисов, работающих с сущностями (класс Service.java):

package by.mrtorex.businessshark.server.interfaces;

import java.util.List;

/\*\*

\* Универсальный интерфейс для сервисов, работающих с сущностями.

\*

\* @param <T> тип сущности

\*/

public interface Service<T> {

Продолжение приложения В

/\*\*

\* Находит сущность по ее идентификатору.

\*

\* @param id идентификатор сущности

\* @return найденная сущность или null, если не найдена

\*/

T findEntity(int id);

/\*\*

\* Сохраняет новую сущность.

\*

\* @param entity сущность для сохранения

\*/

void saveEntity(T entity);

/\*\*

\* Удаляет сущность.

\*

\* @param entity сущность для удаления

\*/

void deleteEntity(T entity);

/\*\*

\* Обновляет данные сущности.

\*

\* @param entity сущность с обновленными данными

\*/

void updateEntity(T entity);

/\*\*

\* Возвращает список всех сущностей.

\*

\* @return список сущностей

\*/

List<T> findAllEntities();

}

Интерфейс для проверки корректности объектов (класс Validatable.java):

package by.mrtorex.businessshark.server.interfaces;

/\*\*

\* Интерфейс для проверки корректности объектов.

\*

\* @param <T> тип проверяемого объекта

\*/

public interface Validatable<T> {

/\*\*

\* Проверяет, является ли объект допустимым.

\*

\* @param t объект для проверки

\* @return true, если объект допустим; false в противном случае

\*/

boolean isValid(T t);

}

Продолжение приложения В

Исключение, выбрасываемое при обработке запросов на сервере (класс ResponseException.java):

package by.mrtorex.businessshark.server.exceptions;

import org.apache.logging.log4j.Logger;

import org.apache.logging.log4j.LogManager;

/\*\*

\* Исключение, выбрасываемое при обработке запросов на сервере.

\* Содержит сообщение об ошибке для отправки клиенту.

\*/

public class ResponseException extends RuntimeException {

private static final Logger logger = LogManager.getLogger(ResponseException.class);

/\*\*

\* Создает новое исключение с сообщением об ошибке.

\* Сообщение логируется на уровне ERROR.

\*

\* @param message сообщение об ошибке (будет отправлено клиенту)

\*/

public ResponseException(String message) {

super(message);

logger.error("Создано ResponseException: {}", message);

}

/\*\*

\* Создает новое исключение с сообщением и причиной.

\* Сообщение и причина логируются на уровне ERROR.

\*

\* @param message сообщение об ошибке (будет отправлено клиенту)

\* @param cause исходное исключение

\*/

public ResponseException(String message, Throwable cause) {

super(message, cause);

logger.error("Создано ResponseException: {}", message, cause);

}

/\*\*

\* Возвращает форматированное сообщение об ошибке.

\* Добавляет стандартный префикс к сообщению.

\*

\* @return отформатированное сообщение для клиента

\*/

@Override

public String getMessage() {

return "Ошибка обработки запроса: " + super.getMessage();

}

}

Перечисление операций, поддерживаемых как сервером, так и клиентом (класс Operation.java):

package by.mrtorex.businessshark.server.enums;

import lombok.Getter;

Продолжение приложения В

/\*\*

\* Перечисление операций, поддерживаемых сервером.

\* Группирует операции по функциональным блокам.

\*/

@Getter

public enum Operation {

/\* Операции с акциями \*/

CREATE\_STOCK("Создание акции"),

READ\_STOCK\_DATA("Получение данных акции"),

UPDATE\_STOCK("Обновление акции"),

DELETE\_STOCK("Удаление акции"),

/\* Операции с компаниями \*/

CREATE\_COMPANY("Создание компании"),

READ\_COMPANY\_DATA("Получение данных компании"),

UPDATE\_COMPANY("Обновление компании"),

DELETE\_COMPANY("Удаление компании"),

/\* Операции с пользователями \*/

CREATE\_USER("Создание пользователя"),

READ\_USER("Получение данных пользователя"),

UPDATE\_USER("Обновление пользователя"),

DELETE\_USER("Удаление пользователя"),

/\* Операции получения списков \*/

GET\_ALL\_COMPANIES("Получение списка компаний"),

GET\_ALL\_STOCKS("Получение списка акций"),

GET\_ALL\_USERS("Получение списка пользователей"),

GET\_ALL\_ROLES("Получение списка ролей"),

GET\_ALL\_STOCKS\_WITH\_NO\_COMPANY("Получение свободных акций"),

/\* Операции с отношениями компаний и акций \*/

GET\_STOCKS\_BY\_COMPANY("Получение акций компании"),

GET\_COMPANY\_BY\_STOCK("Получение компании по акции"),

JOIN\_STOCK\_COMPANY("Привязка акции к компании"),

SEPARATE\_STOCK\_COMPANY("Отвязка акции от компании"),

/\* Операции с портфелями пользователей \*/

ADD\_USER\_STOCK("Добавление акции в портфель"),

GET\_USER\_STOCK("Получение акции из портфеля"),

UPDATE\_USER\_STOCK("Обновление акции в портфеле"),

DELETE\_USER\_STOCK("Удаление акции из портфеля"),

GET\_ALL\_USER\_STOCKS("Получение всего портфеля"),

GET\_ALL\_USER\_STOCK\_IDS("Получение ID связей пользователь-акция"),

GET\_USER\_ACCOUNT("Получение баланса пользователя"),

SET\_USER\_ACCOUNT("Установка баланса пользователя"),

GET\_STOCK\_AVAILABLE\_AMOUNT("Получение доступного количества акций"),

/\* Системные операции \*/

LOGIN("Аутентификация пользователя"),

REGISTER("Регистрация пользователя"),

DISCONNECT("Отключение от сервера");

private final String description;

/\*\*

\* Конструктор перечисления.

\* @param description Описание операции на русском языке

\*/

Operation(String description) {

Продолжение приложения В

this.description = description;

}

/\*\*

\* Проверяет, является ли операция операцией чтения.

\* @return true если операция только читает данные

\*/

@SuppressWarnings("unused")

public boolean isReadOperation() {

return this.name().startsWith("GET\_") ||

this.name().startsWith("READ\_");

}

/\*\*

\* Проверяет, является ли операция операцией изменения.

\* @return true если операция изменяет данные

\*/

@SuppressWarnings("unused")

public boolean isModifyOperation() {

return this.name().startsWith("CREATE\_") ||

this.name().startsWith("UPDATE\_") ||

this.name().startsWith("DELETE\_") ||

this.name().startsWith("ADD\_") ||

this.name().startsWith("JOIN\_") ||

this.name().startsWith("SEPARATE\_") ||

this.name().startsWith("SET\_");

}

}

Класс конфигурации Hibernate для работы с базой данных (класс SessionConfig.java):

package by.mrtorex.businessshark.server.config;

import by.mrtorex.businessshark.server.model.entities.\*;

import lombok.Getter;

import org.hibernate.SessionFactory;

import org.hibernate.HibernateException;

import org.hibernate.cfg.Configuration;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

/\*\*

\* Класс конфигурации Hibernate для работы с базой данных.

\* Реализован как синглтон для обеспечения единственной точки доступа к SessionFactory.

\*/

@Getter

public class SessionConfig {

private static final Logger logger = LogManager.getLogger(SessionConfig.class);

private static volatile SessionConfig sessionConfig;

private final SessionFactory sessionFactory;

/\*\*

\* Приватный конструктор для инициализации SessionFactory.

Продолжение приложения В

\*

\* @throws HibernateException если произошла ошибка при создании SessionFactory

\*/

private SessionConfig() throws HibernateException {

try {

logger.info("Инициализация SessionFactory...");

sessionFactory = new Configuration()

.addAnnotatedClass(User.class)

.addAnnotatedClass(Person.class)

.addAnnotatedClass(Role.class)

.addAnnotatedClass(Company.class)

.addAnnotatedClass(Stock.class)

.buildSessionFactory();

logger.info("SessionFactory успешно инициализирована");

} catch (HibernateException e) {

logger.fatal("Ошибка при создании SessionFactory", e);

throw new HibernateException("Не удалось создать SessionFactory", e);

}

}

/\*\*

\* Возвращает экземпляр SessionConfig (реализация синглтона с двойной проверкой блокировки).

\*

\* @return единственный экземпляр SessionConfig

\* @throws IllegalStateException если при создании экземпляра произошла ошибка

\*/

public static SessionConfig getInstance() {

if (sessionConfig == null) {

synchronized (SessionConfig.class) {

if (sessionConfig == null) {

try {

sessionConfig = new SessionConfig();

} catch (HibernateException e) {

throw new IllegalStateException("Не удалось инициализировать SessionConfig", e);

}

}

}

}

return sessionConfig;

}

/\*\*

\* Закрывает SessionFactory при завершении работы приложения.

\*/

@SuppressWarnings("unused")

public void shutdown() {

try {

if (sessionFactory != null && !sessionFactory.isClosed()) {

logger.info("Закрытие SessionFactory...");

sessionFactory.close();

logger.info("SessionFactory успешно закрыта");

}

} catch (HibernateException e) {

logger.error("Ошибка при закрытии SessionFactory", e);

}

}

Продолжение приложения В

}

Утилита для загрузки сцен в приложении (класс Loader.java):

package by.mrtorex.businessshark.client.gui.utils;

import by.mrtorex.businessshark.client.Main;

import by.mrtorex.businessshark.client.gui.enums.ScenePath;

import by.mrtorex.businessshark.client.gui.enums.ThemesPath;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.stage.Stage;

import java.io.IOException;

import java.net.URL;

import java.util.Objects;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

/\*\*

\* Утилита для загрузки сцен в приложении.

\* Позволяет загружать FXML-сцены и управлять стилями.

\*/

public class Loader {

private static final Logger logger = LogManager.getLogger(Loader.class);

/\*\*

\* Загружает сцену из указанного пути с выбросом исключения в случае ошибки.

\*

\* @param stage основная сцена приложения

\* @param scenePath путь к FXML-файлу

\* @throws IOException если произошла ошибка при загрузке сцены

\*/

public static void loadSceneWithThrowException(Stage stage, ScenePath scenePath) throws IOException {

logger.info("Загрузка сцены из пути: {}", scenePath.getPathToFxml());

URL stagePathURL = Loader.class.getResource(scenePath.getPathToFxml());

FXMLLoader loader = new FXMLLoader(stagePathURL);

Parent root = loader.load();

stage.setResizable(true);

Scene scene = new Scene(root);

scene.getStylesheets().add(Objects.requireNonNull(Loader.class.getResource(String.valueOf(ThemesPath.valueOf(Main.themeName).getPathToCss()))).toExternalForm());

scene.getStylesheets().add(Objects.requireNonNull(Loader.class.getResource("/styles/styles.css")).toExternalForm());

stage.setScene(scene);

Продолжение приложения В

logger.info("Сцена успешно загружена: {}", scenePath.getPathToFxml());

}

/\*\*

\* Загружает сцену из указанного пути и обрабатывает возможные ошибки.

\*

\* @param stage основная сцена приложения

\* @param scenePath путь к FXML-файлу

\*/

public static void loadScene(Stage stage, ScenePath scenePath) {

try {

loadSceneWithThrowException(stage, scenePath);

} catch (IOException e) {

logger.error("Ошибка при загрузке сцены: {}", scenePath.getPathToFxml(), e);

AlertUtil.error("Ошибка навигации", "Не удалось перейти к сцене.");

} catch (Exception e) {

logger.error("Неожиданная ошибка при загрузке сцены: {}", scenePath.getPathToFxml(), e);

AlertUtil.error("Неожиданная ошибка", "Произошла ошибка при загрузке сцены.");

}

}

/\*\*

\* Перезагружает сцену для применения темы.

\*/

public static void reloadForTheme() {

loadScene(Main.getPrimaryStage(), ScenePath.LOGIN);

}

}

Главный класс клиентского приложения (класс Main.java):

package by.mrtorex.businessshark.client;

import by.mrtorex.businessshark.client.gui.enums.ScenePath;

import by.mrtorex.businessshark.client.gui.utils.Loader;

import by.mrtorex.businessshark.server.network.ServerClient;

import javafx.application.Application;

import javafx.scene.image.Image;

import javafx.stage.Stage;

import lombok.Getter;

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

import java.io.InputStream;

/\*\*

\* Главный класс клиентского приложения Business Shark.

\* Отвечает за запуск и настройку основного окна приложения.

\*/

public class Main extends Application {

Продолжение приложения В

private static final Logger logger = LogManager.getLogger(Main.class);

/\*\* Название текущей темы приложения \*/

public static String themeName = "SYNTHWAVE";

/\*\* Основная сцена приложения \*/

@Getter

private static Stage primaryStage;

/\*\*

\* Точка входа в JavaFX приложение.

\*

\* @param stage Основная сцена, автоматически создаваемая JavaFX

\* @throws Exception Если произошла ошибка при инициализации приложения

\*/

@Override

public void start(Stage stage) throws Exception {

try {

logger.info("Запуск приложения...");

primaryStage = stage;

configurePrimaryStage();

Loader.loadScene(primaryStage, ScenePath.LOGIN);

logger.info("Отображение основного окна...");

primaryStage.show();

logger.info("Приложение успешно запущено");

} catch (Exception e) {

logger.error("Критическая ошибка при запуске приложения", e);

logger.info("Аварийное завершение работы...");

throw new Exception("Ошибка при запуске приложения", e);

}

}

/\*\*

\* Настраивает основные параметры главного окна приложения.

\*

\* @throws RuntimeException Если не удалось загрузить иконку приложения

\*/

private void configurePrimaryStage() {

logger.info("Настройка основного окна...");

primaryStage.setTitle("Business Shark");

try {

InputStream iconStream = getClass().getResourceAsStream("/images/logo.png");

if (iconStream == null) {

throw new RuntimeException("Не удалось загрузить иконку приложения");

}

Image icon = new Image(iconStream);

primaryStage.getIcons().add(icon);

} catch (Exception e) {

logger.error("Ошибка при загрузке иконки приложения", e);

throw new RuntimeException("Ошибка при загрузке иконки", e);

Продолжение приложения В

}

}

/\*\*

\* Вызывается при завершении работы приложения.

\* Обеспечивает корректное отключение от сервера.

\*/

@Override

public void stop() {

ServerClient client = ServerClient.getInstance();

if (client != null) {

client.disconnect();

}

}

/\*\*

\* Главный метод для запуска приложения.

\*

\* @param args Аргументы командной строки

\*/

public static void main(String[] args) {

try {

logger.info("Запуск JavaFX приложения...");

Application.launch(args);

logger.info("Приложение завершило работу");

} catch (Exception e) {

logger.fatal("Необработанное исключение в главном потоке", e);

}

}

}

Исключение, возникающее при отсутствии соединения (класс NoConnectionException.java):

package by.mrtorex.businessshark.server.exceptions;

/\*\*

\* Исключение, возникающее при отсутствии соединения.

\*/

public class NoConnectionException extends RuntimeException {

/\*\*

\* Конструктор исключения с сообщением.

\*

\* @param message сообщение об ошибке

\*/

public NoConnectionException(String message) {

super(message);

}

}

# ПРИЛОЖЕНИЕ Г (обязательное) Листинг скрипта генерации базы данных

-- Roles Table

CREATE TABLE Roles (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL UNIQUE

);

-- Persons Table

CREATE TABLE Persons (

id SERIAL PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

patronymic VARCHAR(50),

last\_name VARCHAR(50) NOT NULL

);

-- Users Table

CREATE TABLE Users (

id SERIAL PRIMARY KEY,

username VARCHAR(50) NOT NULL UNIQUE,

password\_hash VARCHAR(60) NOT NULL CHECK (LENGTH(password\_hash) = 60),

role\_id INT NOT NULL,

person\_id INT NOT NULL UNIQUE,

FOREIGN KEY (role\_id) REFERENCES Roles(id) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY (person\_id) REFERENCES Persons(id) ON DELETE CASCADE ON UPDATE CASCADE

);

-- Stocks Table

CREATE TABLE Stocks (

id SERIAL PRIMARY KEY,

ticket VARCHAR(5) NOT NULL UNIQUE,

price DOUBLE PRECISION NOT NULL,

amount INT NOT NULL

);

-- Companies Table

CREATE TABLE Companies (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL UNIQUE

);

-- Accounts Table

CREATE TABLE Accounts (

user\_id INT NOT NULL PRIMARY KEY,

account DOUBLE PRECISION CHECK (account >= 0),

FOREIGN KEY (user\_id) REFERENCES Users(id) ON DELETE CASCADE ON UPDATE CASCADE

);

-- Company\_Stock Join Table

CREATE TABLE Company\_Stock (

company\_id INT NOT NULL,

stock\_id INT NOT NULL UNIQUE,

PRIMARY KEY (company\_id, stock\_id),

Продолжение приложения Г

FOREIGN KEY (company\_id) REFERENCES Companies(id) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY (stock\_id) REFERENCES Stocks(id) ON DELETE CASCADE ON UPDATE CASCADE

);

-- User\_Stock Join Table

CREATE TABLE User\_Stock (

user\_id INT NOT NULL,

stock\_id INT NOT NULL,

amount INT NOT NULL,

PRIMARY KEY (user\_id, stock\_id),

FOREIGN KEY (user\_id) REFERENCES Users(id) ON DELETE CASCADE ON UPDATE CASCADE,

FOREIGN KEY (stock\_id) REFERENCES Stocks(id) ON DELETE CASCADE ON UPDATE CASCADE

);

-- Insert Roles

INSERT INTO Roles (name) VALUES

('Admin'),

('User'),

('Waiting'),

('Blocked');

-- Insert Persons

INSERT INTO Persons (first\_name, patronymic, last\_name) VALUES

('Иван', 'Дмитриевич', 'Кольчевский'),

('Алексей', 'Валерьевич', 'Еднач'),

('Артем', 'Андреевич', 'Ковальчук');

-- Insert Users

INSERT INTO Users (username, password\_hash, role\_id, person\_id) VALUES

('kalci', '$2a$10$fbbf337d042d1710336f8u3RA00CdolZNl/dBgMuLjyLTqU2KVRyS', 1, (SELECT id FROM Persons WHERE last\_name = 'Кольчевский' LIMIT 1)), -- Admin role, password: meowmeowmeow:3

('thomaschar', '$2a$10$eb0f08df4490a93668690uTxouIbEf1lfuvULcoK5iaoUN3GKM7m.', 2, (SELECT id FROM Persons WHERE last\_name = 'Еднач' LIMIT 1)), -- User role, password: \*\*\*\*\*\*\*

('moracer', '$2a$10$f2d5cb0c6df9efc95ff3cuAFGKKYHDMAStb.WJGgjs2CQMhBKfv8S', 2, (SELECT id FROM Persons WHERE last\_name = 'Ковальчук' LIMIT 1)); -- User role, password: ilovebmw42

-- Insert Stocks

INSERT INTO Stocks (ticket, price, amount) VALUES

('APL', 52.42, 10),

('ALPHA', 1337, 20),

('TSL', 1102, 40);

-- Insert Companies

INSERT INTO Companies (name) VALUES

('Google'),

('Apple'),

('Tesla');

-- Insert Accounts

INSERT INTO Accounts (user\_id, account) VALUES

Продолжение приложения Г

(2, 1213),

(3, 1233.58);

-- Link Companies and Stocks

INSERT INTO Company\_Stock (company\_id, stock\_id) VALUES

((SELECT id FROM Companies WHERE name = 'Google' LIMIT 1), (SELECT id FROM Stocks WHERE ticket = 'ALPHA' LIMIT 1)), -- ALPHA is Google's stock

((SELECT id FROM Companies WHERE name = 'Apple' LIMIT 1), (SELECT id FROM Stocks WHERE ticket = 'APL' LIMIT 1)), -- APL is Apple's stock

((SELECT id FROM Companies WHERE name = 'Tesla' LIMIT 1), (SELECT id FROM Stocks WHERE ticket = 'TSL' LIMIT 1)); -- TSL is Tesla's stocka's stock

-- Link Users and Stocks

INSERT INTO User\_Stock (user\_id, stock\_id, amount) VALUES

(2, (SELECT id FROM Stocks WHERE ticket = 'ALPHA' LIMIT 1), 5),

(2, (SELECT id FROM Stocks WHERE ticket = 'TSL' LIMIT 1), 1),

(3, (SELECT id FROM Stocks WHERE ticket = 'APL' LIMIT 1), 1),

(3, (SELECT id FROM Stocks WHERE ticket = 'TSL' LIMIT 1), 7);

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Обозначение | | | | | Наименование | | | Дополнитель-ные сведения | | | |
|  | | | | | Текстовые документы | | |  | | | |
|  | | | | |  | | |  | | | |
| БГУИР КР 6-05-0611-01 102 ПЗ | | | | | Пояснительная записка | | | 141 с. | | | |
|  | | | | |  | | |  | | | |
|  | | | | |  | | |  | | | |
|  | | | | | Графические документы | | |  | | | |
|  | | | | |  | | |  | | | |
| ГУИР 506412 001 ПД | | | | | Описание процесса выполнения | | | Формат А4 | | | |
|  | | | | | транзакции с акциями | | |  | | | |
|  | | | | |  | | |  | | | |
| ГУИР 506412 002 ПД | | | | | Схема алгоритма авторизации | | | Формат А4 | | | |
|  | | | | | пользователей в системе и | | |  | | | |
|  | | | | | выбора необходимого действия | | |  | | | |
|  | | | | |  | | |  | | | |
| ГУИР 506412 003 ПЛ | | | | | Плакат Информационная модель | | | Формат А4 | | | |
|  | | | | | системы | | |  | | | |
|  | | | | |  | | |  | | | |
| ГУИР 506412 004 ПЛ | | | | | Плакат Модели представления | | | Формат А4 | | | |
|  | | | | | системы | | |  | | | |
|  | | | | |  | | |  | | | |
| ГУИР 506412 005 ПЛ | | | | | Плакат Результаты работы | | | Формат А4 | | | |
|  | | | | | приложения | | |  | | | |
|  |  |  |  |  | БГУИР КР 6-05-0611-01 102 Д1 | | | | | | |
|  |  |  |  |  |
|  |  |  |  |  |
| Изм. | Л. | № докум. | Подп. | Дата | Программное средство для управлениями акциями инвестиционного фонда  Ведомость курсового  проекта | Лит | | | | Лист | Листов |
| Разраб. | | Цевелюк |  |  |  | Т | |  | 1 | 1 |
| Пров. | | Козлов |  |  | Кафедра ЭИ  гр. 324402 | | | | | |
| Т.контр. | | Козлов |  |  |
| Рец. | |  |  |  |
| Н.контр | |  |  |  |
| Утв. | | Козлов |  |  |
|  | |  |  |  |