МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
вычислительной техники	МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

Программные средства автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов

Студент _____ Трунова Ксения Дмитриевна
(подпись, дата) (ФИО полностью)

Руководитель _____ Балашова И.Ю.
(подпись, дата) (фамилия, инициалы)

Нормоконтролер _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____№_____)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____№____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2026

Федеральное государственное бюджетное образовательное учреждение высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____А.Ю. Козлов
«26» декабря 2025 г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1.Студент____Трунова Ксения Дмитриевна_____гр.__22ВП2___факультета ВТ
<Фамилия, имя, отчество полностью>

2.Тема работы_ Программные средства автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов

Тема утверждена приказом ПГУ № 1583/о от «30» декабря 2025 г.

3.Руководитель работы_к.т.н., доцент кафедры «МО и ПЭВМ» Балашова И.Ю._____

4.Задание на работу (назначение разработки, исходные данные и т. п.)

1) Назначение: программные средства для автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов.
2) Среда выполнения: Windows 11, веб-браузер.
Языковые средства разработки: C#, TypeScript, SQL, UML.
Инструментальные средства: JetBrains Rider, PostgreSQL, Visual Paradigm
Технологии разработки: ASP.NET Core (REST API), React (SPA), Entity
3) Основные функции приложения:
а) регистрация и авторизация сотрудников;
б) классификация входящих заказов по уровню сложности и типу;
в) распределение заказов между сотрудниками с учетом уровня опыта и текущей нагрузки;
г) построение кратчайшего маршрута сборки заказа по складским зонам;
д) управление статусами сотрудников («Свободен», «Занят», «Перерыв»);
е) повышение уровня опыта сотрудника при достижении пороговых значений;
ж) формирование рейтинговой таблицы сотрудников (таблица лидеров).
з) расчет показателей производительности и прогресса выполнения заданных

5. Перечень подлежащих разработке вопросов

1) *Анализ предметной области и постановка задачи;*
2) *Проектирование;*
3) *Реализация;*
4) *Контроль качества программного обеспечения;*
5) *Оформление пояснительной записки.*

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области и требований к системе	10%	07.04.2026 – 13.04.2026	
2	Проектирование программных средств	20%	14.04.2026 – 25.04.2026	
3	Реализация программных средств	30%	27.04.2026 – 11.05.2026	
4	Тестирование программных средств	20%	12.05.2026 – 21.05.2026	
5	Оформление пояснительной записки	20%	22.05.2026 – 02.06.2026	

Дата выдачи задания «26» декабря 2025г.

Руководитель бакалаврской работы _____ 26.12.2025 г.
(подпись, дата)

Задание к исполнению принял студент _____ 26.12.2025 г
(подпись, дата)

Работу к защите допустить
Декан ФВТ _____ Л.Р. Фионова
(подпись, дата)

Реферат

Пояснительная записка содержит 110 листов, 23 рисунка, 15 таблиц, 20 использованных источников и 5 приложений.

ВНУТРИСКЛАДСКАЯ ЛОГИСТИКА, WMS, ЭКСПРЕСС-ДОСТАВКА ПРОДУКТОВ, АВТОМАТИЗАЦИЯ СКЛАДА, МАРШРУТИЗАЦИЯ, БАЛАНСИРОВКА НАГРУЗКИ, REACT, POSTGRESQL, UML, REST API, ТЕСТИРОВАНИЕ.

Объект исследования – процессы внутрискладской логистики в службе экспресс-доставки продуктов питания.

Цель выпускной квалификационной работы – проектирование и разработка программных средств автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов.

Технологии разработки – разработка серверной части выполнялась в среде JetBrains Rider на языке C# с использованием платформы ASP.NET Core (REST API). Клиентская часть реализована в виде одностраничного приложения на React с применением TypeScript. В качестве системы управления базами данных использована PostgreSQL. Взаимодействие с БД осуществлялось через Entity Framework Core. Моделирование бизнес-процессов выполнено с использованием UML.

Среда исполнения – Windows 11, веб-браузер.

Результаты работы – в ходе выполнения выпускной квалификационной работы были спроектированы и разработаны программные средства автоматизации внутрискладской логистики для службы экспресс-доставки продуктов.

Содержание

Введение 7
1 Анализ предметной области и постановка задачи на разработку программных средств автоматизации процессов внутрискладской логистики 9
1.1 Анализ предметной области 9
1.2 Анализ существующих программных средств 11
1.3 Постановка задачи 15
1.4 Анализ требований 16
1.4.1 Анализ функциональных требований 16
1.4.2 Анализ нефункциональных требований 20
1.5 Планирование разработки и оценка бюджета 21
2 Проектирование программных средств автоматизации внутрискладской логистики 24
2.1 Архитектура разрабатываемой системы 24
2.2 Проектирование структур данных 28
2.2.1 Концептуальная модель базы данных 28
2.2.2 Логическая модель базы данных 29
2.3 Проектирование классов программных средств 31
2.4 Алгоритм маршрутизации сборки заказов 33
2.5 Проектирование пользовательского интерфейса 37
3 Реализация программных средств автоматизации внутрискладской логистики 39
3.1 Реализация базы данных 39
3.2 Реализация серверной части приложения 43
3.3 Реализация клиентской части и интерфейса приложения 49
4 Контроль качества программного обеспечения 59
4.1 Подходы к тестированию и метрики оценки качества кода 59

4.2 Тестирование программных средств автоматизации внутрискладской логистики в службе экспресс-доставки 60
4.2.1 Тестирование пользовательской части программного средства автоматизации внутрискладской логистики 60
4.2.2 Тестирование серверной части программного средства автоматизации внутрискладской логистики 63
4.3 Оценка качества программного кода 65
4.3.1 Метрика Чепина 65
4.3.2 Метрика Холстеда 66
4.3.3 Метрика SLOC 68
Заключение 70
Список использованных источников 72
Приложение А (справочное) Глоссарий 75
Приложение Б (обязательное) Листинг приложения 80
Приложение В (рекомендуемое) Графическая часть 100
Приложение Г (рекомендуемое) Функциональное тестирование 102
Приложение Д (рекомендуемое) Листинг модульных тестов 104

Введение

В современном мире рынок экспресс-доставки продуктов питания демонстрирует значительный темп роста, что создает повышенную потребность в эффективных системах управления складом (WMS). WMS (Warehouse Management System) – это программный комплекс для ускорения и улучшения обработки процессов логистики склада, выполнения заказов. Системы управления складом повышают эффективность обработки всех складских процессов, начиная от поступления товаров и заканчивая их отгрузкой, что позволяет улучшить обработку больших объемов товара, исключить дублирующие операции [1, 2].

Одним из ключевых показателей конкурентоспособности является скорость. Именно от времени цикла сборки заказа зависит, как быстро нужные товары отправятся к покупателю и сколько заказов склад сможет обработать за определенное время. При нерациональной сборке сборщик тратит лишнее время на перемещения и поиск товаров, что становится сдерживающим фактором всей операции. Внедрение полноценных WMS-систем требует значительных финансовых вложений и времени, которыми малый и средний бизнес, как правило, не располагает. Для локальных компаний приоритетом является решение ключевой задачи – минимизация времени на сборку заказа при ограниченных затратах.

Объект исследования – процессы внутрискладской логистики в службе экспресс-доставки продуктов питания.

Предмет исследования – алгоритмы автоматизации процессов распределения заказов, маршрутизации сборки и управления нагрузкой персонала в системах автоматизации внутрискладской логистики.

Целью выпускной квалификационной работы является проектирование и разработка программных средств автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов.

Разрабатываемая система должна обеспечить автоматическое распределение заказов между сотрудниками с учетом их опыта и текущей нагрузки, построение кратчайшего маршрута сборки по складу, а также мониторинг производительности персонала.

Для достижения поставленной цели были сформулированы следующие задачи:

– проанализировать предметную область и существующие программные средства для управления складскими процессами;

– спроектировать архитектуру системы, структуры данных и пользовательский интерфейс;

– реализовать серверную и клиентскую части приложения;

– провести контроль качества разработанного программного обеспечения.

В ходе выполнения работы были использованы следующие методы исследования:

– анализ предметной области и бизнес-процессов складской логистики;

– сравнительный анализ существующих программных решений;

– методы системного анализа, включая декомпозицию системы на подпроцессы, анализ вариантов использования, формирование функциональных и нефункциональных требований;

– методы объектно-ориентированного проектирования;

– моделирование бизнес-процессов с использованием UML-диаграмм;

– разработка алгоритмов для распределения заказов и построения маршрутов;

– тестирование разработанного программного продукта.

Успешное решение поставленных задач позволит спроектировать и разработать программные средства автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов.

1 Анализ предметной области и постановка задачи на разработку программных средств автоматизации процессов внутрискладской логистики

1.1 Анализ предметной области

Современный этап развития рынка доставки товаров первой необходимости характеризуется высокой динамикой и ростом конкуренции. В этих условиях локальные компании, особенно малого и среднего бизнеса, сталкиваются с необходимостью повышения эффективности внутрискладских процессов, чтобы оставаться конкурентоспособными.

Внутрискладская логистика представляет собой совокупность операций, выполняемых внутри склада: приемка товара, его размещение, хранение, комплектация заказов и подготовка к отгрузке [2]. При этом именно этап комплектации выступает узким местом всей цепочки, поскольку напрямую определяет время выполнения заказа.

Склад службы экспресс-доставки продуктов представляет собой специфическую среду: товары разделены на зоны по категориям – молочные продукты, хлеб, кондитерские изделия, мясная продукция, овощи и фрукты, напитки, заморозка, бакалея и прочие. В результате маршрут сборщика не является линейным – он представляет собой последовательность переходов между разнородными зонами.

В процессе комплектации заказа сборщик, получая перечень товарных позиций, фактически решает задачу обхода множества точек в пространстве склада. При отсутствии алгоритмической поддержки маршрут формируется интуитивно, и тем самым приводит к неэффективным траекториям движения. На практике это выражается в повторных проходах через одни и те же зоны.

Не менее значимой задачей выступает распределение нагрузки между сотрудниками. Если все заказы направляются одному сотруднику, он не справляется, тогда как остальные простаивают. Корректное распределение

учитывает текущую занятость каждого сборщика, его опыт и сложность конкретного заказа.

Ранжирование персонала по опыту становится критически важным в модели внутрискладской логистики, поскольку новый сотрудник не всегда способен оперативно справиться со сложным заказом, включающим позиции из множества зон [3].

Отдельного внимания заслуживают задачи управления состояниями и анализа производительности сотрудников. При отсутствие централизованного механизма фиксации статусов («свободен», «занят», «перерыв») система распределения заказов не обладает актуальной информацией, что приводит к ошибочным назначениям и конфликтам при выполнении задач. Отсутствие систематического сбора показателей производительности исключает возможность объективной оценки работы сотрудников и внедрения механизмов мотивации.

В совокупности перечисленные факторы формируют комплекс проблем, ключевыми из которых являются:

– неэффективная маршрутизация сборки;

– дисбаланс нагрузки между сотрудниками;

– игнорирование опыта персонала при назначении задач;

– отсутствие механизмов управления состояниями сотрудников в реальном времени;

– недостаточная развитость инструментов мониторинга и анализа производительности.

В этих условиях возникает объективная необходимость проектирования и разработки программных средств автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов.

Разрабатываемая система ориентирована на предприятия малого и среднего бизнеса, для которых критичны как снижение временных затрат на обработку заказов, так и минимизация финансовых вложений во внедрение. Ее

применение целесообразно в ситуациях, характеризующихся недостаточной формализацией процессов, ограниченным числом сотрудников при высоком потоке заказов, а также использованием неэффективных или устаревших инструментов управления.

Таким образом, анализ предметной области позволяет заключить, что повышение эффективности внутрискладской логистики в службах экспресс-доставки напрямую связано с внедрением программных средств, способных обеспечить комплексное совершенствование процессов сборки заказов, распределения нагрузки и управления персоналом.

1.2 Анализ существующих программных средств

В рамках исследования рынка были рассмотрены существующие программные решения, ориентированные на автоматизацию складских процессов и управление персоналом. Для анализа отобраны три системы: «AXELOT WMS», «Consid.WMS», «1С:WMS Логистика. Управление складом». Выбор обоснован российским происхождением продуктов, доступностью технической документации, а также наличием функций управления сборкой заказов и персоналом склада.

«AXELOT WMS» – российская система управления складом, ориентированная на средние и крупные предприятия.

Широкий набор аналитических виджетов является одним из главных преимуществ данной платформы. В режиме реального времени система отслеживает скорость отбора товаров, уровень исполнения заказов, загруженность складских зон. Данные представлены в виде готовых виджетов и аналитических панелей, которые позволяют без необходимости формировать отдельные отчеты [4].

Система формирует виртуальную модель склада для оптимальной маршрутизации. Дополнительно реализована интеграция с видеоаналитики Trassir и голосовой ассистент для терминалов сбора данных (ТДС).

Вместе с тем анализ применимости «AXELOT WMS» в контексте задач малых и средних складских комплексов позволяет выявить ряд существенных ограничений. Прежде всего, ориентация на крупные предприятия фактически предполагает необходимость значительных затрат, включая расходы на внедрение, адаптацию бизнес-процессов и обучение персонала.

В системе не реализованы элементы геймификации, такие как рейтинговые модели, система достижений или визуализация индивидуального прогресса сотрудников. С точки зрения управления человеческими ресурсами это снижает потенциал нематериальной мотивации.

В данной WMS-системе отсутствует гибкое управление статусами, включая фиксацию перерывов, динамическое переключение режимов занятости и контроль переходных состояний. В условиях высокой плотности заказов это может приводить к коллизиям при распределении задач и снижению эффективности алгоритмов балансировки нагрузки.

«Consid.WMS» – российская WMS-система от компании Consid, позиционирующая как решение для малых и средних складов с быстрым внедрением в формате Agile.

Отличительной особенностью «Consid.WMS» является наличие возможности формировать задания с учетом доступа персонала к конкретным зонам и весогабаритных характеристик товаров, причем каждое задание передается на ТСД с автоматически сформированным маршрутом. Доступна облачная версия, не требующая инвестиций в собственную серверную инфраструктуру.

К ограничениям системы следует отнести: отсутствие механизмов геймификации, ограниченные возможности управления статусами сотрудников в реальном времени, а также сложность самостоятельной настройки под специфику небольшого склада без привлечения компании-разработчика [5].

«1С:WMS Логистика. Управление складом» – флагманский продукт компании «1С» в сегменте управления складом, обеспечивающий комплексную автоматизацию всех складских процессов.

Система обеспечивает адресное хранение, интеллектуальные алгоритмы размещения товаров, автоматическое построение оптимальных маршрутов сборки, управление персоналом с контролем KPI и нагрузки, а также развитый мониторинг статуса каждого заказа и сотрудника в режиме реального времени. Глубокая интеграция с экосистемой «1С» является ее ключевым преимуществом для предприятий, уже использующих учетные системы этого производителя.

Вместе с тем данная интеграционная зависимость одновременно является недостатком: внедрение системы вне экосистемы «1С» существенно затруднено. Помимо этого, система ориентирована на крупные предприятия, а стоимость лицензирования и внедрения делает ее недоступной для малого и среднего бизнеса с ограниченным бюджетом. Механизмы геймификации и управление перерывами сотрудников в системе не предусмотрены [6].

Результаты сравнительного анализа всех четырех систем приведены в таблице 1.

Оценка функционала каждой системы проводилась по следующей шкале:

– 0 – функционал полностью отсутствует;
– 1 – функционал полностью реализован.

Таблица 1 – Сравнительный анализ программных средств управления складскими процессами

Критерий оценки	AXELOT WMS	Consid.WMS	1C:WMS Логистика
Автоматическое назначение заказов сотрудникам	1	1	1
Учет опыта сотрудника при распределении	1	1	1
Построение маршрута сборки внутри склада	1	1	1
Мониторинг нагрузки сотрудников в реальном времени	1	1	1
Элементы геймификации и мотивации персонала	0	0	0

Продолжение таблицы 1

Критерий оценки	AXELOT WMS	Consid.WMS	1C:WMS Логистика
Управление перерывами и статусами сотрудников	0	0	0
Легкость адаптации под небольшой склад	0	1	0

По итогам проведенного сравнительного анализа можно сделать вывод, все рассмотренные продукты-аналоги сходны между собой в реализации базовых операций: автоматическое назначение поступающих заказов, построение маршрута комплектации и отслеживание продуктивности работников. Каждая система имеет свои сильные стороны. Однако в полной мере соответствовать всему комплексу сформулированных требований не удалось ни одному из проанализированных продуктов.

Данный факт обосновывает необходимость создания нового специализированного программного средства, функциональность которого должна охватывать:

– автоматизированное распределение заказовых позиций между сборщиками с обязательной привязкой к уровню их опыта;

– расширенное управление статусными состояниями персонала (переход в режим перерыва, блокировка назначений, ручное подтверждение старта сборки);

– геймификационные элементы (рейтинги, система достижений, визуализация прогресса выполнения задач);

– адаптируемость к реалиям небольшого складского хозяйства без существенных затрат на приобретение лицензий.

На основе сформулированных выше требований принято архитектурное решение разрабатывать систему, нацеленную на малые и средние склады в сфере экспресс-доставки продуктов. Ключевые отличия проектируемого решения от существующих рыночных аналогов сводятся к отсутствию лицензионных

отчислений, наличию полноценного блока геймификации и детализированной модели управления статусами сотрудников. Перечисленные особенности обеспечивают оптимальную функциональную комплектацию для автоматизации внутрискладской логистики в службах экспресс-доставки.

1.3 Постановка задачи

Разрабатываемая система предназначена для автоматизации процессов управления заказами и сборщиками на складе службы экспресс-доставки продуктов. Разрабатываемая система ориентирована на склады предприятий малого и среднего бизнеса в сфере экспресс-доставки продуктов. Она должна обеспечивать полный цикл обработки заказа: от его поступления и классификации до подтверждения выполнения сборщиком. Ключевое требование – функционирование в режиме реального времени: назначение заказа должно выполняться незамедлительно, что особенно критично в часы пиковой нагрузки.

Для достижения поставленной цели система должна реализовывать следующие функции:

– классификация входящих заказов по уровню сложности на основе количества товарных позиций и ассортимента;

– балансировка нагрузки между сотрудниками с учетом их текущей загрузки и уровня опыта;

– построение кратчайшего пути маршрутов перемещения сотрудников по складским зонам;

– внедрение механизмов геймификации, включая систему рейтингов и визуализацию прогресса выполнения задач, для стимулирования производительности персонала.

Целевым пользователем разрабатываемой системы является сборщик – рядовой сотрудник склада, осуществляющий физическую комплектацию заказов. Сборщик получает назначенные задания, видит оптимальный маршрут сборки по складским зонам, отмечает выполнение позиций и подтверждает

завершение заказа. Интерфейс должен быть минималистичным, адаптированным для работы с мобильных устройств и не требовать предварительного обучения – это принципиальное требование с учетом высокой текучести кадров, характерной для сферы экспресс-доставки.

1.4 Анализ требований

1.4.1 Анализ функциональных требований

Диаграмма вариантов использования описывает взаимоотношения между группами вариантов использования и действующими лицами, участвующими в процессе, и позволяет на верхнем уровне отобразить связи между актерами и необходимыми сценариями работы системы. Инструмент UML обеспечивает целостное понимание функциональных границ разрабатываемого программного средства, выделяя ключевые роли пользователей и их цели при взаимодействии с системой, что особенно важно для согласования ожиданий заинтересованных сторон на ранних этапах проектирования. Кроме того, она служит основой для дальнейшей детализации требований и проектирования архитектуры системы. Визуализация вариантов использования также упрощает коммуникацию между разработчиками, заказчиками и другими участниками проекта [7, 8].

На основании анализа предметной области и опыта рассмотренных аналогов были сформулированы функциональные требования к разрабатываемой системе, разделенные по функциональным блокам.

Аутентификация и управление учетными записями:

– регистрация и авторизация сотрудника в системе;
– выход из учетной записи;
– просмотр личного профиля.

Управление заказами:

– при создании заказа система автоматически определяет его сложность (простой, средний, сложный) и тип (однотипный, разнотипный);

– расчетное время выполнения заказа вычисляется автоматически на основе состава позиций и применяемых коэффициентов.

Распределение заказов:

– система автоматически назначает заказ свободному сотруднику с подходящим уровнем опыта;

– при назначении учитывается текущая нагрузка сотрудника;

– если все подходящие сотрудники заняты, заказ назначается наименее загруженному из них;

– сотрудник принимает заказ в работу вручную, после чего его статус изменяется на «Занят».

Маршрутизация сборки:

– при создании заказа система автоматически строит кратчайший маршрут обхода складских зон;

– уровень опыта повышается автоматически при достижении установленных пороговых значений выполненных заказов;

– сотрудник может перейти в режим перерыва, в течение которого новые заказы ему не назначаются;

– система отслеживает актуальный статус каждого сотрудника: «Свободен», «Занят», «Перерыв».

Мониторинг и статистика:

– система ведет учет выполненных заказов по каждому сотруднику;

– в профиле сотрудника отображаются показатели OPH (Orders Per Hour – количество заказов в час), LUR (Labour Utilization Rate – коэффициент использования рабочего времени) и прогресс к дневной цели;

– предусмотрена таблица лидеров для сопоставления показателей сотрудников.

Диаграмма вариантов использования, отражающая перечисленные функциональные требования и основные сценарии работы системы, представлена на рисунке 1.

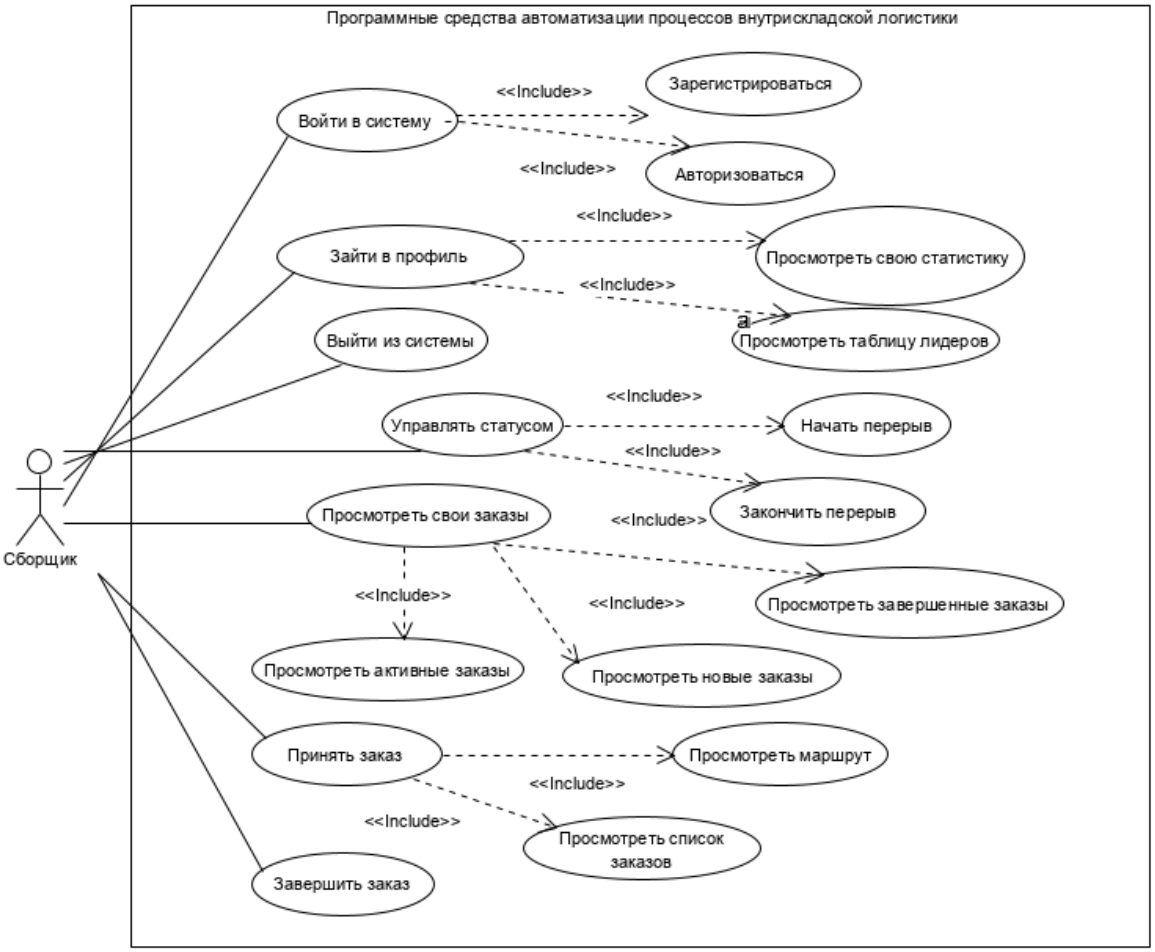

Рисунок 1 – Диаграмма вариантов использования

Сценарий варианта использования «Принять заказ» представлен на рисунке 2.

Наименование: принять заказ
ID: 1
Предусловие: Сотрудник авторизован в системе. У сотрудника есть назначенный заказ в статусе «Назначен».
Действующие лица: сотрудник
Основной поток: **1.** Сотрудник нажимает кнопку «Принять заказ» рядом с выбранным заказом. **2.** Система проверяет, что заказ существует и имеет статус «Назначен». **3.** Система проверяет, нет ли у сотрудника других активных заказов (статус «Принят» или «В сборке»). **4.** Система меняет статус заказа на «Принят». **5.** Система меняет статус сотрудника на «Занят». **6.** Система отображает сообщение: «Заказ принят в работу».
Постусловие: заказ переведен в статус «Принят». Статус сотрудника изменен на «Занят».

Рисунок 2 – Вариант использования «Принять заказ»

Сценарий варианта использования «Завершить заказ» представлен на рисунке 3.

Наименование: завершить заказ
ID: 2
Предусловие: сотрудник авторизован в системе. У сотрудника есть принятый заказ в статусе «Принят».
Действующие лица: сотрудник
Основной поток: 1. Сотрудник нажимает кнопку «Завершить заказ» рядом с выбранным заказом. 2. Система проверяет, что заказ существует и имеет статус «Принят». 3. Система меняет статус заказа на «Собран». 4. Система устанавливает дату завершения заказа. 5. Система увеличивает счетчик выполненных заказов сотрудника. 6. Система уменьшает текущую нагрузку сотрудника. 7. Система отображает сообщение: «Заказ успешно выполнен» и обновленную статистику.
Постусловие: заказ завершен и переведен в статус «Собран».

Рисунок 3 – Вариант использования «Завершить заказ»

Сценарий варианта использования «Уйти на перерыв» представлен на рисунке 4.

Наименование: уйти на перерыв
ID: 3
Предусловие: сотрудник авторизован в системе.
Действующие лица: сотрудник
Основной поток: 1. Сотрудник нажимает кнопку «Начать перерыв». 2. Система проверяет, существует ли сотрудник. 3. Система проверяет, есть ли у сотрудника активные заказы (статус «Принят» или «В сборке»). 4. Система проверяет, не находится ли сотрудник уже на перерыве. 5. Система меняет статус сотрудника на «Перерыв». 6. Система отображает сообщение: «Сотрудник ушел на перерыв» и время начала перерыва.
Постусловие: статус сотрудника изменен на «Перерыв»

Рисунок 4 – Вариант использования «Уйти на перерыв»

В результате анализа функциональных требований были выделены ключевые варианты использования системы, среди которых особое внимание уделено трем основным процессам: приемке заказа сборщиком, фиксации завершения сборки и переходу сотрудника в режим перерыва. Разработанные

спецификации сценариев детально описывают последовательность операций пользователя и соответствующие реакции программного обеспечения.

Полученный перечень требований полностью закрывает задачи небольшого склада экспресс-доставки продуктов и становится базой для проектирования архитектуры, структур данных и пользовательского интерфейса. Это позволяет обеспечить гибкость системы и её адаптацию к изменяющимся бизнес-процессам склада.

1.4.2 Анализ нефункциональных требований

Нефункциональные требования определяют качественные характеристики системы, не связанные напрямую с ее функциями [9]. Разработка серверной части системы автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов осуществляется на платформе ASP.NET Core на языке программирования C# с использованием архитектурного стиля REST API. Клиентская часть реализуется в виде одностраничного приложения (SPA) на основе библиотеки React с применением языка TypeScript. В качестве СУБД используется PostgreSQL. Взаимодействие с базой данных осуществляется посредством фреймворка Entity Framework Core.

Приложение должно корректно функционировать в современных браузерах: Google Chrome (версии 80 и выше), Mozilla Firefox (версии 75 и выше), Apple Safari (версии 13 и выше), Microsoft Edge (версии 80 и выше).

Данное требование обусловлено тем, что разрабатываемая система является веб-приложением, не требующим установки, и предполагает доступ с любого устройства сотрудника – включая мобильные терминалы и персональные смартфоны. Поддержка актуальных версий перечисленных браузеров обеспечивает совместимость с современными стандартами HTML5, CSS3 и JavaScript ES2020+, на которых построена клиентская часть.

1.5 Планирование разработки и оценка бюджета

Грамотное планирование и предварительная оценка затрат представляют собой фундаментальные составляющие успешного создания программного обеспечения. На данном этапе формируется перечень работ, определяются временные рамки их выполнения и распределяются приоритеты между задачами.

Для управления ходом проектных работ был задействован инструмент ProjectLibre – кроссплатформенная система управления проектами с открытым исходным кодом, функционально сопоставимая с профессиональными решениями корпоративного класса. Она позволяет формализовать перечень задач и назначить ответственных исполнителей с указанием их загрузки [10].

После внесения полного перечня стадий создания программного обеспечения в рабочую среду инструмента была получена связная модель проекта, на основании которой стало возможным как календарное планирование с определением дат начала и окончания каждой задачи, так и ресурсно-стоимостной анализ, позволяющий рассчитать суммарные трудозатраты и финансовые расходы.

Визуализацию временной структуры проекта обеспечивает диаграмма Ганта – инструмент, в котором по горизонтальной оси откладывается календарный период, а по вертикальной выстраивается иерархический реестр задач с указанием ответственных исполнителей и длительности выполнения. Диаграмма Ганта наглядно демонстрирует логические взаимозависимости между задачами, выделяет критический путь проекта и позволяет своевременно выявлять потенциальные конфликты в распределении трудовых ресурсов [11].

Полученная диаграмма Ганта, приведённая на рисунке 5, позволяет наглядно оценить общую продолжительность разработки и при необходимости скорректировать сроки выполнения отдельных этапов.

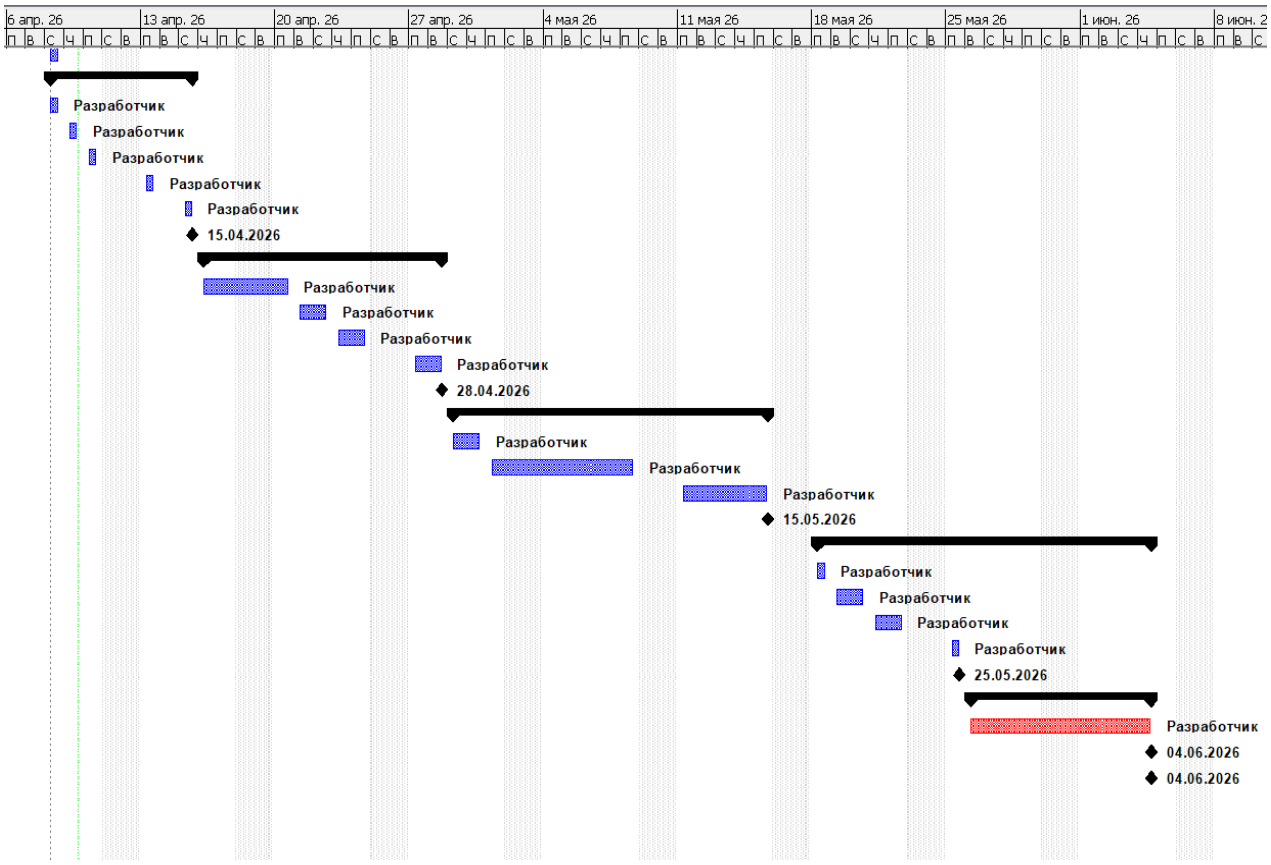

Рисунок 5 – Диаграмма Ганта

В результате декомпозиции проектных работ было выделено пять основных этапов, охватывающих весь жизненный цикл разработки программного обеспечения: анализ требований, проектирование системы, реализацию программных компонентов, тестирование и подготовку итоговой документации. Каждый этап включает набор взаимосвязанных задач, выполнение которых необходимо для последовательного достижения поставленных целей проекта.

Согласно разработанному календарному плану, датой начала работ определено 8 апреля 2026 года, а завершение всех этапов ожидается 4 июня 2026 года. Таким образом, общая продолжительность разработки составляет порядка десяти календарных недель. Сформированный график позволяет обеспечить рациональное распределение трудовых ресурсов между этапами проекта и контролировать соблюдение установленных сроков выполнения работ.

Детализированный реестр задач представлен на рисунке 6.

Название	Начало	Окончание
Начало	08.04.2026, 08:00	08.04.2026, 17:00
Анализ предметной области и постановка задачи на разработку программных средств автома	**08.04.2026, 08:00**	**15.04.2026, 17:00**
Анализ предметной области	08.04.2026, 08:00	08.04.2026, 17:00
Анализ существующих программных средств	09.04.2026, 08:00	09.04.2026, 17:00
Постановка задачи	10.04.2026, 08:00	10.04.2026, 17:00
Анализ функциональных требований	13.04.2026, 08:00	13.04.2026, 17:00
Анализ нефункциональных требований	14.04.2026, 17:00	15.04.2026, 17:00
Конец анализа предметной области и постановки задач	15.04.2026, 17:00	15.04.2026, 17:00
Проектирование программных средств автоматизации внутрискладской логистики	**16.04.2026, 08:00**	**28.04.2026, 17:00**
Проектирование архитектура разрабатываемой системы	16.04.2026, 08:00	20.04.2026, 17:00
Проектирование структур данных	21.04.2026, 08:00	22.04.2026, 17:00
Проектирование пользовательского интерфейса	23.04.2026, 08:00	24.04.2026, 17:00
Проектирование классов программных средств	25.04.2026, 08:00	28.04.2026, 17:00
Конец проектирования	28.04.2026, 17:00	28.04.2026, 17:00
Реализация программных средств автоматизации внутрискладской логистики	**29.04.2026, 08:00**	**15.05.2026, 17:00**
Реализация базы данных	29.04.2026, 08:00	30.04.2026, 17:00
Реализация основного приложения	01.05.2026, 08:00	08.05.2026, 17:00
Реализация интерфейса приложения	11.05.2026, 08:00	15.05.2026, 17:00
Конец реализации разрабатываемой системы	15.05.2026, 17:00	15.05.2026, 17:00
Контроль качества программного обеспечения	**18.05.2026, 08:00**	**04.06.2026, 17:00**
Выбор режима тестирования и метрик кода	18.05.2026, 08:00	18.05.2026, 17:00
Тестирование пользовательской части программных средств автоматизации внутрискладской логистики	19.05.2026, 08:00	20.05.2026, 17:00
Тестирование серверной части программных средств автоматизации внутрискладской логистики	21.05.2026, 08:00	22.05.2026, 17:00
Оценка качества программного кода	25.05.2026, 08:00	25.05.2026, 17:00
Конец контроля качества программного обеспечения	25.05.2026, 17:00	25.05.2026, 17:00
Оформление отчета	**26.05.2026, 08:00**	**04.06.2026, 17:00**
Оформление пояснительной записки	26.05.2026, 08:00	04.06.2026, 17:00
Оформление отчета завершено	04.06.2026, 17:00	04.06.2026, 17:00
Конец проекта	04.06.2026, 17:00	04.06.2026, 17:00

Рисунок 6 – Реестр задач

Для оценки финансовых затрат на реализацию приложения ресурсам были назначены ставки. Используемые ресурсы отражены на рисунке 7.

Название	Тип	E-mail адрес	Максимальные единицы	Стандартная ставка	Начислять	Основной календарь
Разработчик	Работа	trunova.kseniya@gmail.com	100 %	500,00 ₽/час	Пропорционально	Пятидневка

Рисунок 7 – Используемые ресурсы

По итогам стоимостного моделирования совокупные трудозатраты на создание программных средств автоматизации внутрискладкой логистики составили 392 часа, с 08.04.2026 по 04.06.2026, соответственно 42 дня. Полная расчетная стоимость проекта с учетом тарифных ставок задействованного ресурса – 196 000 рублей.

2 Проектирование программных средств автоматизации внутрискладской логистики

2.1 Архитектура разрабатываемой системы

Архитектура программной системы – это совокупность принятых проектных решений, определяющих состав компонентов, способ их организации и характер взаимодействия между ними. Выбор архитектурного подхода определяет сложность разработки, удобство сопровождения и соответствие системы условиям ее эксплуатации [12].

Система представляет собой монолитное веб-приложение с локальным развертыванием всех компонентов: веб-сервер ASP.NET Core, база данных PostgreSQL. Взаимодействие между браузером и сервером осуществляется по протоколу HTTP через петлевой интерфейс (localhost) без выхода во внешнюю сеть. Данная конфигурация соответствует одному из сценариев эксплуатации (например, для малого склада с одним рабочим местом). При необходимости масштабирования на нескольких сотрудников серверная часть и база данных размещаются на выделенном компьютере локальной сети, а клиентские устройства получают доступ к системе по его IP-адресу, что не требует изменений в архитектуре приложения [13]. Такая гибкость позволяет системе адаптироваться к изменяющимся условиям эксплуатации без необходимости переработки архитектуры.

Выбор монолитной архитектуры обусловлен условиями эксплуатации. Система рассчитана на использование в пределах одного склада ограниченным числом сотрудников. Такой подход исключает требования к горизонтальному масштабированию. Применение микросервисной архитектуры в подобных условиях создало бы избыточную операционную сложность без практических преимуществ: потребовались бы отдельная инфраструктура оркестрации, межсервисное взаимодействие и управление несколькими процессами развертывания.

Внутри монолитного приложения реализовано функциональное разделение на две части. Клиентская часть – одностраничное приложение (SPA) на основе библиотеки React с применением языка TypeScript – исполняется в браузере и отвечает за отображение данных и обработку действий пользователя. Серверная часть реализована на платформе ASP.NET Core и предоставляет REST API в формате JSON. Клиентская часть обращается к серверу через HTTP-запросы, сборка статических файлов (HTML, CSS, JavaScript) обслуживается непосредственно сервером ASP.NET Core [12, 13].

Внутри серверной части применяется трехуровневая логическая организация, близкая к шаблону MVC, однако с существенным отличием: уровень представления (View) вынесен в отдельное клиентское приложение React. В результате контроллеры выполняют роль Web API-контроллеров и не формируют HTML-разметку – они принимают запросы, вызывают сервисный слой и возвращают данные в формате JSON [14].

Точкой входа в приложение является файл Program.cs, в котором настраивается веб-хост, регистрируются зависимости (AddScoped для сервисов), подключается база данных через Entity Framework Core, настраивается CORS.

Для наглядного представления физического размещения компонентов системы разработана диаграмма развертывания. Диаграмма приведена на рисунке 8.

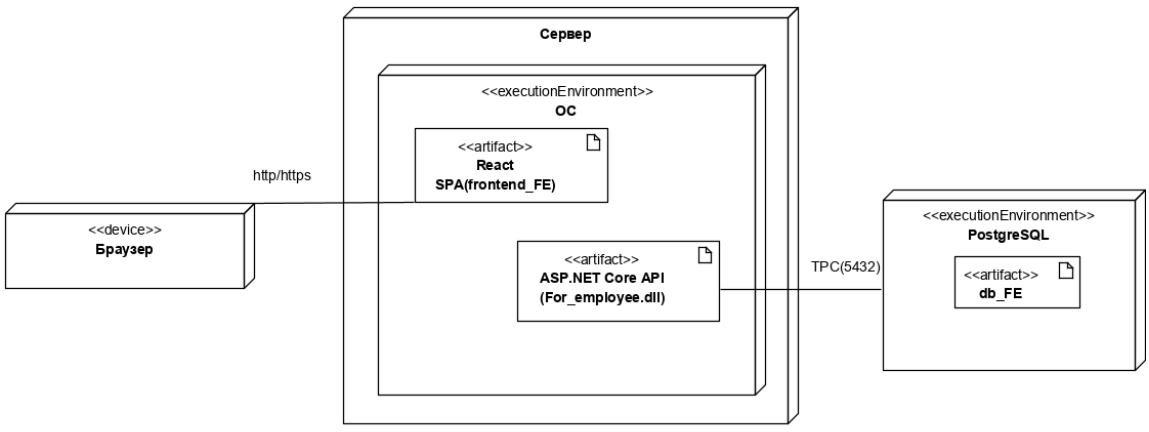

Рисунок 8 – Диаграмма развертывания разрабатываемой системы

На диаграмме показано, что клиентская часть (браузер), серверное приложение ASP.NET Core и СУБД PostgreSQL размещены на одном вычислительном узле. Взаимодействие между браузером и сервером осуществляется по HTTP через localhost, а доступ к базе данных выполняется напрямую из серверной части.

Уровень моделей (Model) представлен классами предметной области: Employee, Order, OrderItem, Product, ProductCategory. Эти классы описывают структуру данных и напрямую связаны с базой данных через класс AppDbContext, реализованный на основе ORM-фреймворка Entity Framework Core. При необходимости связанные данные загружаются совместно – например, при авторизации сотрудник загружается вместе с назначенными ему заказами для формирования полного объекта ответа.

Уровень контроллеров (Controller) включает специализированные классы: AuthController, EmployeesController, OrdersController, ProductsController. Контроллеры принимают HTTP-запросы, выполняют базовую валидацию входных данных, координируют вызовы сервисного слоя и возвращают результаты клиенту.

Сервисный слой (Business Logic Layer) содержит основную бизнес-логику системы. Каждый класс отвечает за строго определенную функциональную область:

– класс OrderService – создание заказа, классификация по типу и сложности, расчет расчетного времени выполнения на основе состава позиций и коэффициентов;

– класс LoadBalancingService – назначение заказов сотрудникам с учетом их текущей нагрузки и уровня опыта, повышение уровень опыта при достижении пороговых значений;

– класс RoutingService – построение кратчайшего маршрута сборки на основе физической планировки склада и зон хранения товаров в конкретном заказе;

– класс GamificationService – расчет показателей производительности OPH и LUR, формирование данных таблицы лидеров, присвоение значков сотрудникам;

– сервис EmployeeService – получение агрегированных данных по сотрудникам, включая связанные заказы.

Контроллеры взаимодействуют с сервисным слоем посредством внедрения зависимостей. Например, при создании заказа OrdersController передает запрос в OrderService, который создает заказ, инициирует назначение сотрудника через LoadBalancingService и строит маршрут через RoutingService.

Уровень доступа к данным (Data Access Layer) реализован с использованием класса AppDbContext, предоставляющего доступ к таблицам базы данных через типизированные коллекции DbSet. Взаимодействие с данными осуществляется посредством LINQ-запросов, а загрузка связанных сущностей выполняется с использованием механизма Include. Контекст базы данных используется сервисным слоем для выполнения операций чтения и модификации данных.

Отдельного внимания заслуживает управление состояниями объектов системы. Статусы заказов («Назначен», «Принят», «Собран») и сотрудников («Свободен», «Занят», «Перерыв») хранятся в базе данных в виде строковых значений. Проверка допустимости переходов между состояниями выполняется на уровне сервисного слоя. Данный подход обеспечивает централизованный контроль бизнес-правил и исключает дублирование логики в контроллерах.

Таким образом, архитектура разрабатываемой системы представляет собой монолитное веб-приложение с локальным развертыванием, трехуровневой логической организацией кода и функциональным разделением на клиентскую (React) и серверную (ASP.NET Core) части. Выбранный подход соответствует условиям эксплуатации системы и обеспечивает достаточную простоту разработки, отладки и сопровождения.

2.2 Проектирование структур данных

2.2.1 Концептуальная модель базы данных

Концептуальная модель данных описывает предметную область на наивысшем уровне абстракции: определяет основные сущности системы и смысловые связи между ними без привязки к конкретной технологии хранения данных. Концептуальная модель служит основой для последующего логического и физического проектирования базы данных [15].

Перед построением концептуальной модели необходимо выделить основные сущности предметной области разрабатываемой системы. Описание сущностей концептуальной модели базы данных представлено в таблице 2.

Таблица 2 – Описание сущностей концептуальной модели базы данных

Название сущности	Идентификатор	Назначение сущности
Сотрудник	Employee	Содержит сведения о сотруднике-сборщике
Заказ	Order	Содержит сведения о заказе
Позиция заказа	OrderItem	Содержит сведения об отдельной товарной позиции в составе заказа
Категория товаров	ProductCategory	Содержит сведения о категории товаров
Товар	Product	Содержит сведения о конкретном товаре

Между выделенными сущностями определены следующие связи, которые представлены в таблице 3.

Таблица 3 – Связи между сущностями концептуальной модели

Наименование связи	Сущность 1	Сущность 2	Тип связи	Описание
Сотрудник – Заказ	Employee	Order	1:N	Один сотрудник может быть назначен на несколько заказов
Заказ – Позиция заказа	Order	OrderItem	1:N	Один заказ содержит несколько товарных позиций

Продолжение таблицы 3

Наименование связи	Сущность 1	Сущность 2	Тип связи	Описание
Категория – Товар	ProductCategory	Product	1:N	Одна категория объединяет несколько товаров

Концептуальная модель данных построена в нотации «сущность – связь» и представлена на рисунке 9.

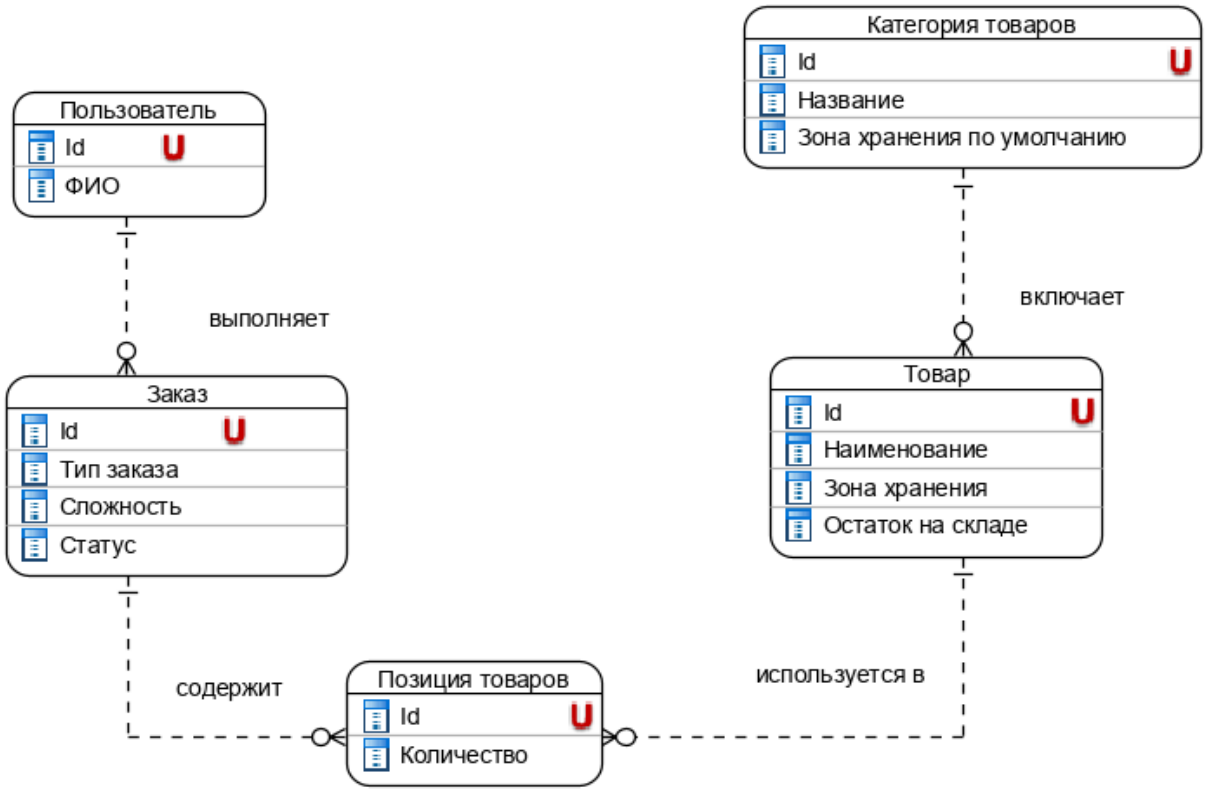

Рисунок 9 – Диаграмма сущность-связь концептуальной модели

Таким образом, концептуальная модель базы данных позволяет четко структурировать основные элементы системы автоматизации внутрискладской логистики. Эта модель становится основой для дальнейшего проектирования логической и физической структуры базы данных.

2.2.2 Логическая модель базы данных

Логическое проектирование базы данных представляет собой промежуточный этап между концептуальным описанием предметной области и

ее физической реализацией в конкретной СУБД. На данном этапе каждая сущность концептуальной модели преобразуется в самостоятельное отношение реляционной схемы, ее свойства становятся атрибутами соответствующего отношения, а смысловые связи между сущностями получают формальное выражение через механизм первичных и внешних ключей. В отличие от концептуальной модели, которая ориентирована на понимание предметной области человеком, логическая модель уже содержит все необходимые сведения для последующей автоматической генерации SQL-скриптов создания таблиц [16].

Отличительной особенностью логической модели по сравнению с концептуальной является необходимость однозначного определения доменов атрибутов: для каждого столбца фиксируется тип данных, допустимость пустого значения и наличие ограничений целостности. Кроме того, на этом этапе уточняются нормальные формы таблиц для минимизации избыточности данных. При этом логическая схема остается независимой от конкретной платформы хранения данных и ориентирована исключительно на требования реляционной модели.

Логическая модель базы данных была построена на основе концептуальной диаграммы. Каждая из пяти сущностей – сотрудник, заказ, позиция заказа, категория товаров и товар – получила соответствующую таблицу с первичными ключами. Связи между сущностями реализованы через внешние ключи с заданными правилами ссылочной целостности: для позиций заказа используется каскадное удаление, для товаров – ограничение удаления, а при удалении сотрудника внешний ключ в заказах обнуляется, чтобы сохранить историю. Такой подход к правилам удаления обеспечивает баланс между целостностью данных и сохранением значимой информации.

Логическая модель базы данных, созданная на основе концептуальной схемы, представлена на рисунке 10.

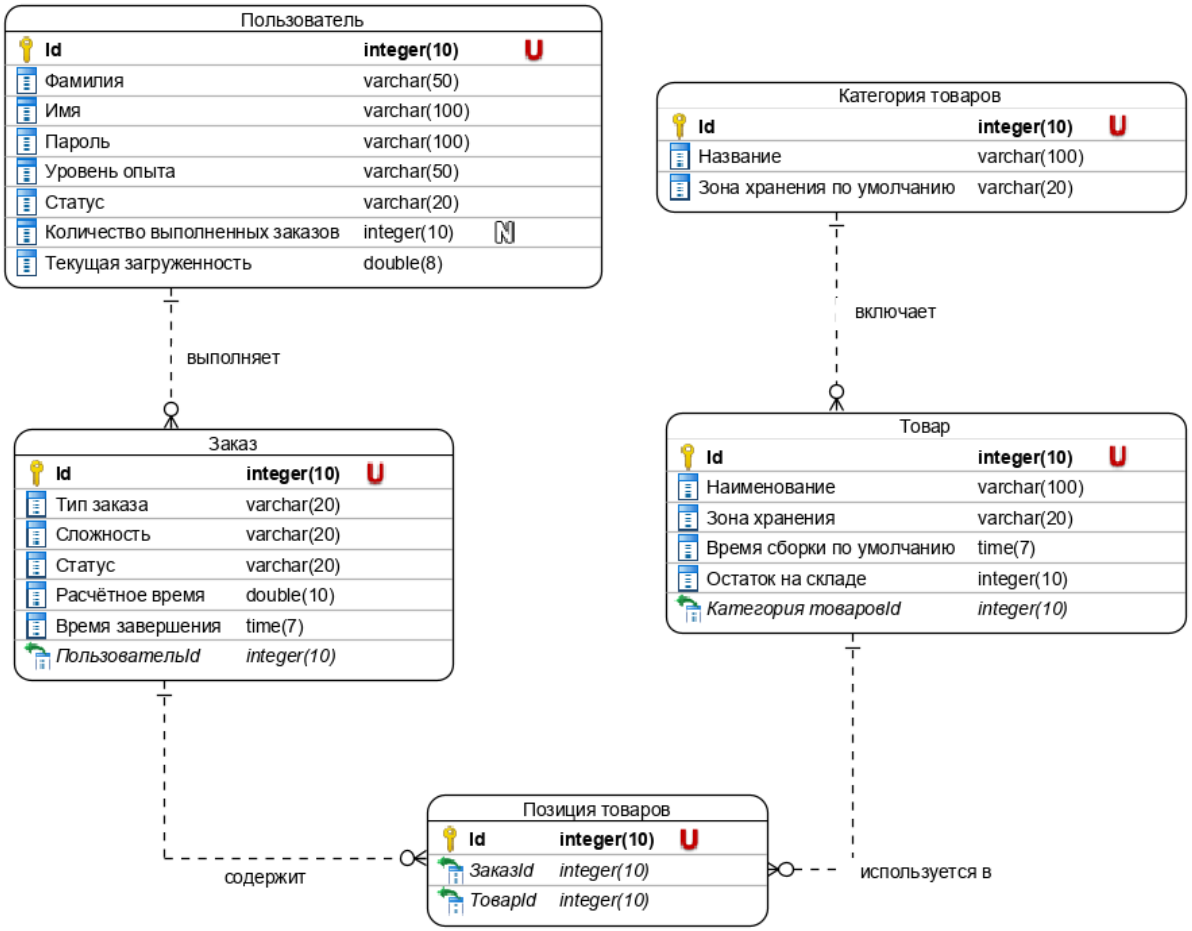

Рисунок 10 – Диаграмма сущность-связь логической модели

Таким образом, полученная логическая схема полностью отражает структуру предметной области системы автоматизации внутрискладской логистики и служит непосредственной основой для физической реализации базы данных средствами PostgreSQL.

2.3 Проектирование классов программных средств

Диаграмма классов – это разновидность диаграмм унифицированного языка моделирования (UML), используемая для визуального представления структуры программной системы: состава классов, их атрибутов, методов и взаимосвязей между ними [17]. В соответствии с архитектурой разрабатываемой системы была спроектирована диаграмма классов (рисунок В.1).

Описание функционального назначения основных классов приведено в таблице 4.

Таблица 4 – Описание основных классов системы

Класс	Назначение
Employee	Модель сотрудника, содержащая атрибуты учетной записи, уровень опыта, статуса и нагрузки, а также навигационное свойство для связанных заказов
Order	Модель заказа, содержащая атрибуты сложности, типа, расчетного времени, статуса и ссылки на исполнителя и список позиций
OrderItem	Модель позиции заказа, содержащая наименование товара, код зоны хранения и нормативное время сборки
Product	Модель товара из справочника, содержащая зону хранения, нормативное время сборки, цену и остаток.
ProductCategory	Модель категории товаров, определяющая зону хранения и нормативное время сборки по умолчанию для всей категории
AppDbContext	Контекст Entity Framework Core. Обеспечивает доступ ко всем таблицам базы данных через типизированные коллекции DbSet. Содержит конфигурацию связей и индексов (Fluent API)
OrderService	Класс создания и классификации заказов, реализующий алгоритмы определения сложности и типа заказа, расчета расчетного времени с применением коэффициентов
LoadBalancingService	Класс балансировки нагрузки, реализующий алгоритм назначения заказа свободному сотруднику с учетом уровня опыта и текущей нагрузки и содержащий логику автоматического повышения уровня опыта
RoutingService	Класс построения маршрута сборки, реализующий алгоритм оптимального обхода складских зон на основе физической планировки склада и рассчитывающий суммарное расстояние маршрута
GamificationService	Класс геймификации, формирующий значок сотрудника, рассчитывает показатели OPH, LUR, прогресс к дневной и недельной цели, формирующий данные для таблицы лидеров
EmployeeService	Класс для получения расширенной информации о сотрудниках с учетом их заказов
AuthService	Класс аутентификации, отвечающий за регистрацию и вход пользователей в систему
AuthController	Контроллер обрабатывает запросы аутентификации и регистрации
OrdersController	Контроллер обрабатывает все операции с заказами
EmployeesController	Контроллер управляет данными сотрудников
ProductsController	Контроллер предоставляет доступ к справочнику товаров

Спроектированная диаграмма классов отражает принцип единственной ответственности: каждый класс реализует строго одну функцию – модели хранят состояние, сервисы реализуют алгоритмы, контроллеры обрабатывают HTTP-запросы. Такая организация кода снижает связность компонентов и упрощает их независимое тестирование.

2.4 Алгоритм маршрутизации сборки заказов

Внутрискладская логистика в службах экспресс-доставки продуктов сталкивается с актуальной проблемой неэффективной маршрутизации сборки заказов. В условиях отсутствия алгоритмической поддержки сборщик формирует маршрут интуитивно, что приводит к увеличению времени выполнения заказа. При высокой нагрузке и ограниченных временных ресурсах это становится критическим фактором, напрямую влияющим на производительность склада и качество обслуживания клиентов.

Традиционные методы оптимизации маршрутов обладают рядом ограничений, которые затрудняют их применение в условиях динамично меняющихся заказов и ограниченных вычислительных ресурсов. Динамическое программирование гарантирует нахождение глобального оптимума, однако имеет факториальную вычислительную сложность, что делает его непригодным для работы в режиме реального времени при большом числе точек. Жадные алгоритмы отличаются высокой скоростью вычислений, но обеспечивают лишь локально оптимальные решения, что не всегда соответствует требованиям к точности маршрутизации. Генетические алгоритмы позволяют находить решения, близкие к оптимальным, однако требуют значительных затрат на настройку параметров и не подходят для малых и средних складов с динамически поступающими заказами.

В разрабатываемой системе применяется двухуровневый подход: на верхнем уровне перебираются все возможные порядки посещения зон хранения, на нижнем – для каждого перехода между соседними точками маршрута кратчайший путь по сеточному графу склада находится с помощью

алгоритма A*. Он относится к классу эвристических методов поиска и представляет собой усовершенствованную версию алгоритма Дейкстры. Его ключевое отличие заключается в использовании эвристической оценки для направленного поиска, чтобы сократить количество исследуемых вершин и ускорить вычисления.

Двухуровневый алгоритм включает следующие шаги.

Шаг 0. Предварительная обработка данных и классификация зон хранения.

Перед построением маршрута выполняется валидация и дедупликация зон хранения: проверяется корректность обозначения каждой зоны. Если в заказе присутствуют несколько товаров из одного ряда, для целей маршрутизации этот ряд учитывается единожды. Дубликаты из входного списка удаляются, чтобы исключить повторные посещения одной и той же точки.

Для каждой валидной зоны определяется категория хранения:

- обычные товары (Normal);
- охлаждаемые товары (Cooling);
- замороженные товары (Freezing).

На основании результата определяется финальная точка маршрута. Если в заказе присутствуют зоны Cooling или Freezing, маршрут завершается у стола Table2. В остальных случаях – у стола Table1.

Шаг 1. Генерация перестановок.

На этом этапе формируются все возможные варианты порядка посещения зон хранения. Количество таких вариантов определяется числом зон и быстро растет с их увеличением. Полный перебор всех допустимых маршрутов на данном шаге гарантирует нахождение оптимального решения на верхнем уровне алгоритма.

Шаг 2. Поиск длины маршрута для каждой перестановки.

Для каждой перестановки вычисляется суммарная длина маршрута. Для поиска кратчайшего пути между соседними зонами в каждой перестановке используется алгоритм A* с манхэттенским расстоянием в качестве эвристической функции.

Шаг 3. Выбор оптимальной перестановки.

Определяется перестановка с минимальной суммарной длиной маршрута.

Для реализации нижнего уровня двухуровневого алгоритма необходимо формализовать представление складского пространства и детально рассмотреть работу алгоритма A*, применяемого для поиска кратчайших путей между соседними точками маршрута.

Складское пространство представляется в виде прямоугольной сетки, где каждая проходимая ячейка является вершиной графа. Формально склад описывается как взвешенный неориентированный граф G = (V, E), где V – множество вершин, соответствующих проходимым ячейкам графа, каждой вершине сопоставлены целочисленные координаты (x, y), E – множество ребер, соединяющих соседние проходимые ячейки по горизонтали или вертикали.

Ячейки, занятые стеллажами или иными физическими препятствиями, исключаются из графа. Зоны хранения товаров, зона приемки заказов Entry и столы комплектации Table1, Table2 отображаются на соответствующие вершины графа.

На вход алгоритма подаются:

– множество зон хранения $Z = \{z_1, z_2, ..., z_n\}$, где каждая зона $z \in Z$ задана своими координатами (x, y) на графе склада;

– стартовая точка – зона приемки заказов Entry с координатами (x_start, y_start);

– конечная точка – стол комплектации с координатами (x_end, y_end), выбираемый на основе температурных категорий зон заказа.

Алгоритм A* – это алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной). В алгоритме A* каждой ситуации S, полученной из исходной ситуации в результате определенной последовательности действий, придается численная оценка:

$$f(S) = g(S) + h(S),$$

где g(S) – реальная текущая стоимость ситуации S (сумма стоимостей действий, которые нужно выполнить, чтобы дойти до данной точки от исходной ситуации);

h(S) – эвристическая функция, которая оценивает стоимость наилучшей последовательности действий, начинающейся с S и заканчивающейся решением.

Работа алгоритма A* включает ряд этапов.

Этап 1. Инициализация.

Граф склада G = (V, E), множество зон хранения $Z = \{z_1, z_2, ..., z_n\}$, стартовая вершина и конечная вершина. Создается открытый список O, который изначально содержит только стартовую вершину. Множество закрытых вершин F изначально пусто.

Этап 2. Основной цикл.

Пока множество O не пусто, выполняются следующие действия:

– на множестве O выбирается ситуация S, для которой значение f(S) минимально.

– если S является целевой вершиной, алгоритм завершается.

– если S не является решением, то она раскрывается путем построения новых ситуаций S', полученных из S путем выполнения возможных действий.

– для каждой порожденной ситуации S' вычисляется ее оценочная функция;

– новые ситуации добавляются в множество O, если они не входят в O или F. Если ситуация уже находится в O или F, ее стоимость обновляется, если новое значение g(S) меньше старого.

Этап 3. Завершение.

Алгоритм завершается, когда достигнута целевая вершина или множество O становится пустым.

Эффективность алгоритма A* во многом определяется качеством эвристической функции h(S). В данной работе используется манхэттенское расстояние, так как:

– склад имеет прямоугольную планировку с системой проходов, по которым перемещение осуществляется только вдоль горизонтальных и вертикальных направлений;

– значение функции никогда не превышает фактическую длину маршрута, что гарантирует нахождение оптимального решения алгоритмом A*.

– для вычисления эвристики требуются только операции вычитания и взятия модуля, что позволяет использовать алгоритм в режиме реального времени;

Манхэттенское расстояние вычисляется по формуле:

$$h(S) = | x_n - x_{goal} | + | y_n - y_{goal} |,$$

где (x_n, y_n) – координаты текущей вершины;

(x_{goal}, y_{goal}) – координаты целевой вершины.

Таким образом, предлагаемый двухуровневый подход позволяет оптимизировать маршруты сборки заказов на складе, сочетая перебор возможных порядков посещения зон с точным поиском кратчайших путей между ними. Использование алгоритма A* с манхэттенской эвристикой обеспечивает вычислительную эффективность и оптимальность решений, что критически важно для работы в условиях высокой нагрузки и ограниченных временных ресурсов.

2.5 Проектирование пользовательского интерфейса

Проектирование пользовательского интерфейса – это этап разработки программного обеспечения, на котором определяется структура, компоновка и визуальное представление элементов взаимодействия с пользователем. Целью проектирования интерфейса является не только обеспечение удобства и эффективности выполнения целевых задач, но и создание интуитивно понятной среды, снижающей риск ошибок и ускоряющей обучение пользователей [18].

Разрабатываемая система ориентирована на сборщика склада, работающего с мобильным устройством или стационарным терминалом в

условиях физической нагрузки. Исходя из этого, к интерфейсу сформулированы следующие ключевые требования:

– минималистичность – на каждом экране отображается только необходимая для текущего действия информация;

– однозначность статусов – текущее состояние заказа и сотрудника должно быть визуально различимо без прочтения текста;

– минимальное количество переходов между экранами – выполнение основных операций не должно требовать более трех последовательных действий.

Структура интерфейса сформирована на основе типовых сценариев работы сотрудника, к которым относятся авторизация, выполнение заказов, а также контроль собственной производительности. Для обеспечения удобства навигации используется единая верхняя панель, предоставляющая доступ к основным разделам и отображающая текущий статус сотрудника. Кроме того, панель может показывать уведомления о новых заказах или важных изменениях, помогая пользователю оставаться в курсе актуальных событий.

Экран авторизации спроектирован как простая форма ввода учетных данных с возможностью перехода к регистрации. После успешного входа пользователь получает доступ к основному функционалу системы.

Экран обработки заказа должен обеспечивать одновременный доступ к информации о маршруте и составе заказа. Визуальное представление склада используется для упрощения ориентации и повышения точности выполнения операций.

Страница профиля сотрудника обеспечивает мониторинг показателей производительности в режиме реального времени. При открытии страница асинхронно загружает два независимых набора данных: статистику текущего сотрудника и общий список персонала, находящихся в рейтинговой таблице.

В целом разработанный интерфейс ориентирован на снижение когнитивной нагрузки и повышение эффективности взаимодействия пользователя с системой за счет продуманной структуры и визуальных решений.

3 Реализация программных средств автоматизации внутрискладской логистики

3.1 Реализация базы данных

Физическая модель базы данных отражает конкретную реализацию структуры данных в выбранной системе управления базами данных (СУБД). Она включает в себя детализированное описание таблиц, их атрибутов, типов данных, первичных и внешних ключей, а также ограничений целостности. При разработке физической модели учитываются специфические требования и ограничения используемой СУБД, такие как типы данных, ограничения на именование объектов и другие особенности.

Физическая модель базы данных была создана в JetBrains Rider с применением Entity Framework Core. Такой подход дает возможность определять структуру хранения данных в коде на C#. Это устраняет риск рассинхронизации между моделью данных и физической схемой при внесении изменений.

Управление эволюцией схемы осуществляется через механизм миграций. Каждое структурное изменение – добавление таблицы, введение нового столбца или модификация индекса – фиксируется в отдельном файле и применяется к базе данных в заданной последовательности. Это позволяет воспроизводить актуальное состояние схемы на любом окружении без необходимости ручного написания SQL-скриптов, а также заполнять базу данных начальными справочными данными.

При настройке связей между сущностями использовались различные стратегии удаления зависимых объектов. Удаление заказа сопровождается каскадным удалением связанных позиций заказа (OrderItem), тогда как удаление товара ограничено – пока он присутствует в составе хотя бы одного заказа, удаление блокируется. Для заказов, исполнитель которых был удален из системы, внешний ключ обнуляется, что позволяет сохранить историю выполненных заказов независимо от изменений в составе персонала.

Для обмена данными между клиентской и серверной частями использовались DTO-модели (Data Transfer Object). Такой подход отделяет внутренние сущности базы данных от объектов, передаваемых через API, и позволяет возвращать клиенту только те поля, которые необходимы для конкретной операции, не раскрывая внутреннюю структуру хранения данных.

В качестве СУБД выбран PostgreSQL. Для администрирования базы данных, визуального контроля структуры таблиц и выполнения диагностических запросов используется инструмент pgAdmin.

Схема базы данных разрабатываемой системы охватывает пять основных сущностей: сотрудник (Employee), заказ (Order), позиция заказа (OrderItem), категория товаров (ProductCategory) и товар (Product).

Физическая ER-диаграмма, отражающая структуру таблиц и связи между ними, приведена на рисунке 11.

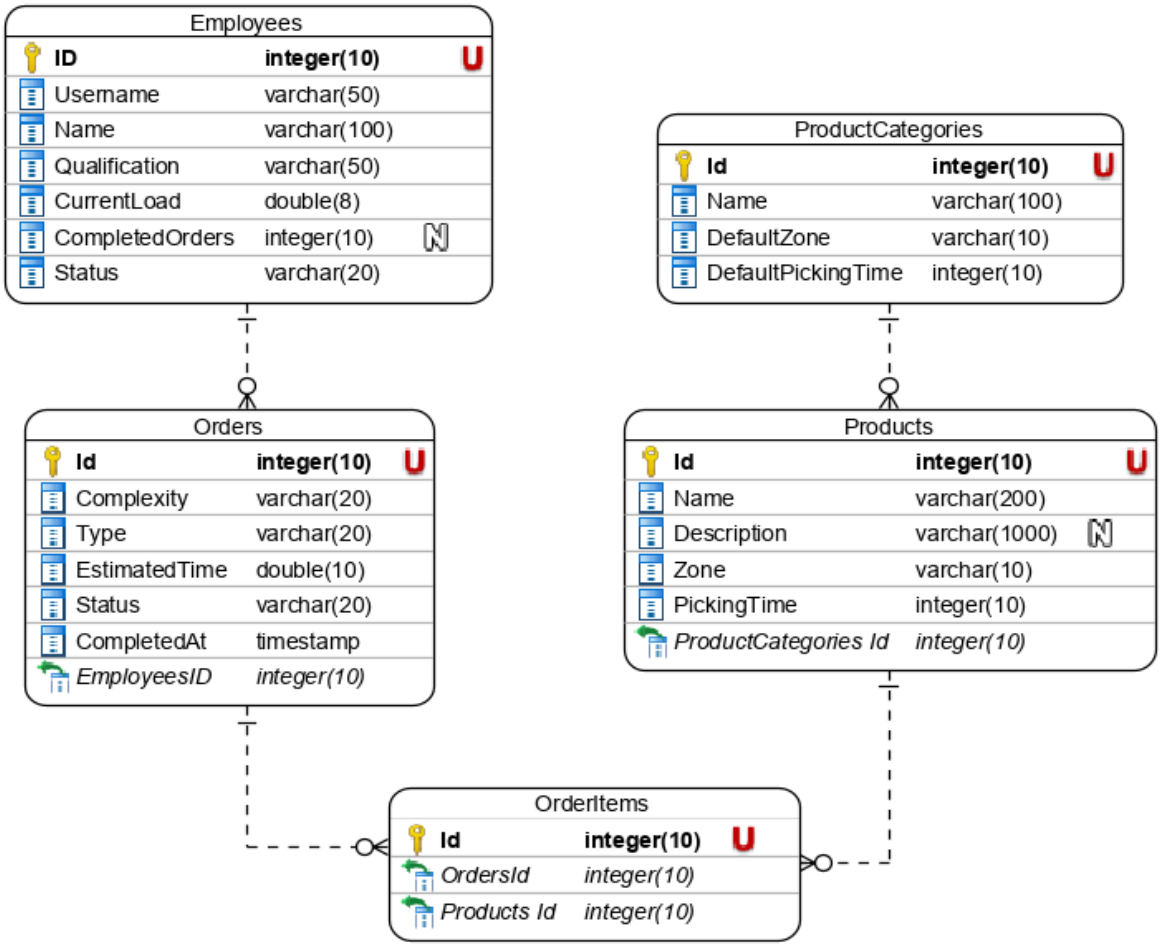

Рисунок 11 – Физическая ER-диаграмма базы данных

Структура базы данных была реализована через конфигурацию контекста Entity Framework Core – класса AppDbContext, унаследованного от DbContext. Доступ к таблицам осуществляется посредством типизированных коллекций DbSet<T>: Orders, OrderItems, Employees, ProductCategories и Products. Детальная настройка каждой сущности выполнена в методе OnModelCreating с использованием Fluent API.

Атрибуты сущности «Сотрудник» описаны в таблице 5.

Таблица 5 – Сущность «Сотрудник» (Employee)

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Идентификатор сотрудника	Id	integer	PK	-	Да
Имя пользователя (логин)	Username	varchar	-	50	Да
Пароль	Password	varchar	-	100	Да
Полное имя сотрудника	Name	varchar	-	100	Да
Уровень опыта	Qualification	varchar	-	50	Да
Текущая нагрузка (мин)	CurrentLoad	double	-	-	Да
Количество выполненных заказов	CompletedOrders	integer	-	-	Нет
Текущий статус	Status	varchar	-	20	Да

Атрибуты сущности «Заказ» описаны в таблице 6.

Таблица 6 – Сущность «Заказ» (Order)

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Идентификатор заказа	Id	integer	PK	-	Да
Уровень сложности	Complexity	varchar	-	20	Да
Тип заказа	Type	varchar	-	20	Да

Продолжение таблицы 6

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Расчетное время сборки (мин)	EstimatedTime	double	-	-	Да
Статус заказа	Status	varchar	-	20	Да
Идентификатор исполнителя	EmployeeId	integer	FK	-	Да
Дата и время завершения	CompletedAt	timestamp	-	-	Да

Атрибуты сущности «Позиция заказа» описаны в таблице 7.

Таблица 7 – Сущность «Позиция заказа» (OrderItem)

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Идентификатор позиции	Id	integer	PK	-	Да
Ссылка на товар	ProductId	integer(10)	FK	10	Да
Ссылка на заказ	OrderId	integer(10)	FK	10	Да

Атрибуты сущности «Категория товаров» приведено в таблице 8.

Таблица 8 – Сущность «Категория товаров» (ProductCategory)

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Идентификатор категории	Id	integer	PK	-	Да
Наименование категории	Name	varchar	-	100	Да
Зона хранения по умолчанию	DefaultZone	varchar	-	10	Да
Нормативное время сборки (мин)	DefaultPickingTime	integer	-	-	Да

В таблице 9 представлены атрибуты сущности «Товар».

Таблица 9 – Сущность «Товар» (Product)

Содержание поля	Системное наименование параметров таблицы	Тип	Ключ	Размер	Обязательность
Идентификатор товара	Id	integer	PK	-	Да
Наименование товара	Name	varchar	-	200	Да
Описание	Description	varchar	-	1000	Нет
Зона хранения	Zone	varchar	-	10	Да
Время сборки (мин)	PickingTime	integer	-	-	Да
Идентификатор категории	CategoryId	integer	FK	-	Да

Разработанная физическая схема данных служит непосредственной основой для реализации серверной части системы и полностью соответствует требованиям, сформулированным на этапе логического проектирования.

3.2 Реализация серверной части приложения

Разработка серверной части системы осуществлялась в среде JetBrains Rider с использованием языка программирования C# и платформы ASP.NET Core. Выбор данной технологии обусловлен ее кроссплатформенностью, высокой производительностью и наличием встроенных механизмов, упрощающих разработку масштабируемых веб-приложений. Существенным преимуществом является поддержка внедрения зависимостей, позволяющая снизить связанность компонентов, а также асинхронная обработка запросов, обеспечивающая устойчивую работу системы при значительных нагрузках.

Архитектура серверной части выполнена в формате REST API и представлена набором контроллеров, каждый из которых отвечает за свою функциональную область. Взаимодействие между компонентами системы иллюстрируется диаграммой компонентов на рисунке 12.

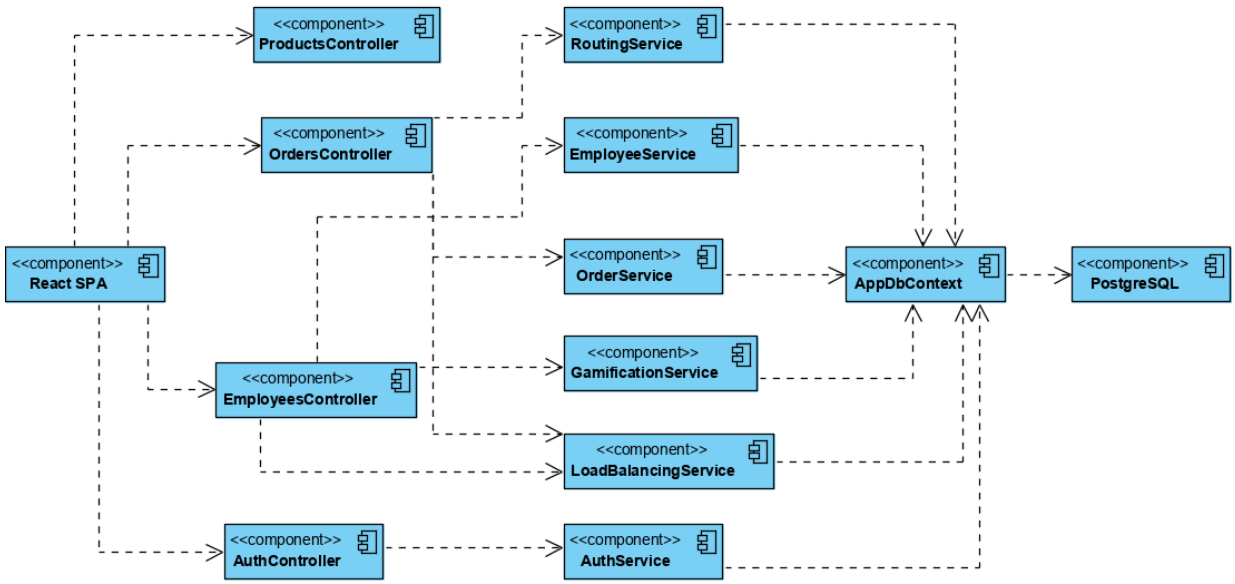

Рисунок 12 – Диаграмма компонентов разрабатываемой системы

Контроллеры серверной части используются преимущественно для обработки HTTP-запросов, получения параметров и возврата результатов клиентской части. Основная прикладная логика сосредоточена в сервисном слое. Такой подход позволяет разделить ответственность между компонентами системы: контроллеры отвечают за маршрутизацию запросов, а сервисы – за выполнение бизнес-правил и обработку данных.

Контроллер AuthController предназначен для регистрации сотрудников и выполнения процедур аутентификации. Обработка соответствующих операций реализована через сервис AuthService, выполняющий проверку регистрационных данных, контроль уникальности имени пользователя и формирование учетной записи сотрудника. При регистрации новому пользователю автоматически назначается начальный уровень опыта «Новичок», устанавливаются начальные параметры нагрузки и рабочий статус. Для хранения учетных данных используется хэширование паролей с применением алгоритма BCrypt. В базе данных сохраняется не исходный пароль, а его хэш-значение. Данный подход снижает риск компрометации учетных записей в случае несанкционированного доступа к данным.

Контроллер OrdersController реализует функционал для управления заказами на всех этапах их обработки. В его задачи входит создание новых заказов, их автоматическая классификация по сложности и типу, а также назначение исполнителей. Кроме того, контроллер отвечает за завершение обработки заказов. Для построения маршрутов сборки и расчета дополнительных параметров, таких как расчетное время выполнения, используются специализированные сервисы RoutingService и LoadBalancingService.

Функционал управления персоналом реализован в контроллере EmployeesController. Этот компонент обеспечивает отслеживание текущих состояний сотрудников, включая переходы между режимами работы, а также автоматическое обновление уровня квалификации по мере выполнения заказов. Кроме того, контроллер предоставляет данные о показателях производительности каждого сотрудника.

Управление персоналом реализовано в EmployeesController. Данный компонент отвечает за контроль текущих состояний сотрудников, включая переключение между режимами работы, обновление уровня опыта по мере накопления завершенных заказов, а также предоставление информации о показателях производительности.

Контроллер ProductsController обеспечивает доступ к справочной информации о товарах. В его задачи входит организация поиска по наименованию и фильтрация по категориям, позволяющая эффективно работать с ассортиментом системы.

Бизнес-логика системы сосредоточена в пяти специализированных сервисах. Обработка заказов сосредоточена в сервисе OrderService, который отвечает за создание заказа, его классификацию, расчет предполагаемого времени выполнения, назначение исполнителя и сопровождение жизненного цикла обработки. После формирования заказа выполняется назначение сотрудника посредством сервиса балансировки нагрузки LoadBalancingService, а построение маршрута передается сервису маршрутизации RoutingService.

При создании заказа автоматически определяются его характеристики. Тип заказа зависит от количества уникальных категорий товаров: если все позиции относятся к одной категории, заказ считается однотипным, в противном случае – разнотипным. В процессе разработки было выявлено, что количество позиций не всегда отражает фактическую сложность обработки. Например, сбор большого числа одинаковых товаров может требовать меньше времени, чем работа с различными товарами из нескольких зон склада. Поэтому при определении сложности учитываются как количество позиций, так и состав заказа.

При определении времени выполнения заказа учитываются несколько факторов: время, необходимое для сбора товаров, перемещение между складскими зонами, а также специфика маршрута. В частности, если заказ содержит товары из температурных зон, в расчет добавляется время на прохождение через шлюзы. Кроме того, на итоговое время влияют коэффициенты сложности заказа. Все параметры, используемые для расчета, вынесены в константы, что позволяет легко вносить корректировки при изменении условий работы склада.

Распределение заказов между сотрудниками выполняется сервисом LoadBalancingService, который учитывает требуемый уровень опыта и текущую нагрузку сотрудника. Процесс выбора исполнителя реализован в виде последовательной фильтрации. Сначала исключаются сотрудники, находящиеся в режиме перерыва, затем анализируются соответствие уровня опыта сотрудника сложности заказа, и в итоге выбирается сотрудник с наименьшей текущей загрузкой.

В системе реализован механизм динамического повышения квалификации сотрудников. После выполнения каждого заказа производится автоматический пересчет показателей эффективности, и при достижении определенных порогов (пятнадцать заказов для перехода с уровня «Новичок» на «Опытный» и тридцать пять заказов для перехода на уровень «Эксперт») система автоматически

обновляет уровень опыта сотрудника. Это позволяет учитывать накопленный опыт работника и использовать его при последующих назначениях заказов.

Формирование маршрутов осуществляется в сервисе RoutingService, который строит оптимальную последовательность посещения зон хранения с учетом реальной структуры складского пространства. Сервис использует двухуровневый алгоритм маршрутизации: на верхнем уровне выполняется полный перебор всех возможных порядков обхода зон, а на нижнем – для каждой пары соседних точек маршрута кратчайший путь по сеточному графу склада вычисляется с помощью алгоритма A*.

Перед построением маршрута сервис проводит предварительную обработку данных: валидацию и удаление дубликатов зон хранения, а также классификацию товаров по температурным категориям (Normal, Cooling, Freezing). На основе категорий определяется конечная точка маршрута – стол комплектации Table1 или Table2.

Складское пространство моделируется как взвешенный неориентированный граф, где вершины представляют проходимые ячейки сетки, а ребра соединяют соседние ячейки по горизонтали или вертикали. Непроходимые ячейки, такие как стеллажи и физические препятствия, исключаются из графа.

Для каждой перестановки зон сервис вычисляет суммарную длину маршрута, последовательно применяя алгоритм A* между соседними точками. Перестановка с минимальной суммарной длиной принимается как оптимальный маршрут и передается сборщику для выполнения.

Для оценки эффективности работы сотрудников используется сервис GamificationService, выполняющий расчет производственных показателей на основе данных о выполненных заказах. В процессе разработки было принято решение отказаться от хранения промежуточных статистических значений в базе данных и вычислять их динамически по фактическим данным. Такой подход позволяет избежать рассогласования информации при изменении состояния заказов.

Одним из основных показателей является OPH – количество заказов, выполненных за час. Расчет выполняется не по фиксированной продолжительности смены, а на основании фактического времени работы: учитывается интервал между первым и последним завершенным заказом за текущий день.

Дополнительно рассчитывается показатель LUR, отражающий степень загрузки рабочего времени. Значение определяется как отношение суммарного расчетного времени выполнения заказов к продолжительности рабочей смены. При вычислении используется предполагаемое время обработки заказов (EstimatedTime), рассчитанное ранее на этапе создания заказа.

В системе реализован механизм отслеживания выполнения дневных и недельных целей. На основании накопленной статистики рассчитывается прогресс выполнения плановых показателей и формируется рейтинг сотрудников. При определении позиции в рейтинге учитывается количество завершенных заказов, что позволяет сравнивать результаты работы сотрудников между собой.

Работа с данными сотрудников реализована через сервис EmployeeService, который обеспечивает получение информации, управление состояниями и выполнение проверок, связанных с изменением режимов работы.

Основная функция сервиса заключается в управлении переходами между состояниями сотрудников. В системе определены три возможных статуса: «Свободен», «Занят» и «Перерыв». Перед тем как сотрудник сможет уйти на перерыв, сервис выполняет две проверки: наличие активных заказов со статусами «Принят» или «В сборке», а также текущий статус сотрудника. Если хотя бы одно из условий не выполняется, переход в режим перерыва блокируется, а пользователь получает соответствующее сообщение об ошибке. Аналогично организовано завершение перерыва: сервис проверяет, что сотрудник действительно находится в статусе «Перерыв», чтобы предотвратить некорректные изменения состояния.

После того как сотрудник завершает выполнение заказа, сервис автоматически запускает процесс пересчета его уровня опыта, используя функционал LoadBalancingService. Все изменения сохраняются в базе данных, а клиентская часть приложения получает актуализированную информацию о текущем уровне опыта сотрудника и общем количестве выполненных им заказов.

Все изменения состояния объектов выполняются через методы серверной части. От прямого изменения статусов на стороне клиента было решено отказаться, поскольку это усложняло контроль допустимых переходов между состояниями.

Для обработки типовых ошибок использовался механизм обработки исключений. При возникновении ошибок пользователю возвращается структурированный ответ с описанием причины. Это позволило упростить отладку взаимодействия между клиентской и серверной частями приложения.

Доступ к данным организован посредством контекста AppDbContext, реализованного с использованием технологии Entity Framework Core. Для выборки и обработки данных применяются LINQ-запросы, а загрузка связанныхсущностей выполняется с использованием механизма Include, который позволяет формировать целостные модели данных для дальнейшей обработки.

Фрагмент кода серверной части представлен в приложении Б. Полный код расположен на сервисе git [19].

3.3 Реализация клиентской части и интерфейса приложения

Клиентская часть системы реализована в виде одностраничного приложения (SPA) на основе библиотеки React с применением языка TypeScript. Маршрутизация между страницами осуществляется на стороне браузера посредством React Router без полной перезагрузки страницы. Все обращения к API выполняются асинхронно через HTTP-клиент, что обеспечивает отзывчивость интерфейса в условиях интенсивного потока заказов.

При проектировании интерфейса учитывалась специфика основного пользователя – сборщика, работающего в условиях физической нагрузки с мобильным устройством или терминалом. В связи с этим интерфейс был ориентирован на минимизацию визуальных элементов, однозначную индикацию состояний и сокращение количества действий, необходимых для выполнения основных операций.

Для обеспечения удобства навигации используется единая верхняя панель управления, которая предоставляет быстрый доступ к основным разделам системы и отображает текущий статус сотрудника (рисунок 13).

Панель управления содержит следующие элементы:

1 – вкладка «Мои заказы»;

2 – текущее время и дата;

3 – индикатор статуса сотрудника;

4 – кнопка для переключения статуса;

5 – выпадающее меню на имени сотрудника;

6 – кнопка «Мой профиль»;

7 – кнопка «Выйти».

Рисунок 13 – Главная панель управления

Вход в систему начинается со страницы авторизации – формы ввода логина и пароля (рисунок 14).

Основные элементы страницы «Вход в систему»:

1 – поле для ввода логина;

2 – поле для ввода пароля;

3 – кнопка для входа в систему;

4 – кнопка для перехода на страницу регистрации;

5 – кнопка для возврата на главную страницу.

Рисунок 14 – Страница авторизации

Страница регистрации предназначена для создания учетной записи нового сотрудника (рисунок 15). Форма включает четыре обязательных поля: имя пользователя (логин), полное имя сотрудника, пароль и его подтверждение. На стороне клиента реализована базовая валидация вводимых данных. При несоблюдении любого из условий регистрация блокируется и пользователю выводится информационное сообщение об ошибке. В случае успешной регистрации данные сотрудника сохраняются, сотруднику автоматически присваивается начальный уровень опыта «Новичок».

Страница регистрации нового сотрудника содержит следующие элементы:

1 – поле для ввода имени пользователя;

2 – поле для ввода полного имени сотрудника;

3 – поле для ввода пароля;

4 – поле для подтверждения пароля;

5 – кнопка для регистрации нового сотрудника;

6 – ссылка для перехода на страницу входа в систему;

7 – ссылка для возврата на главную страницу.

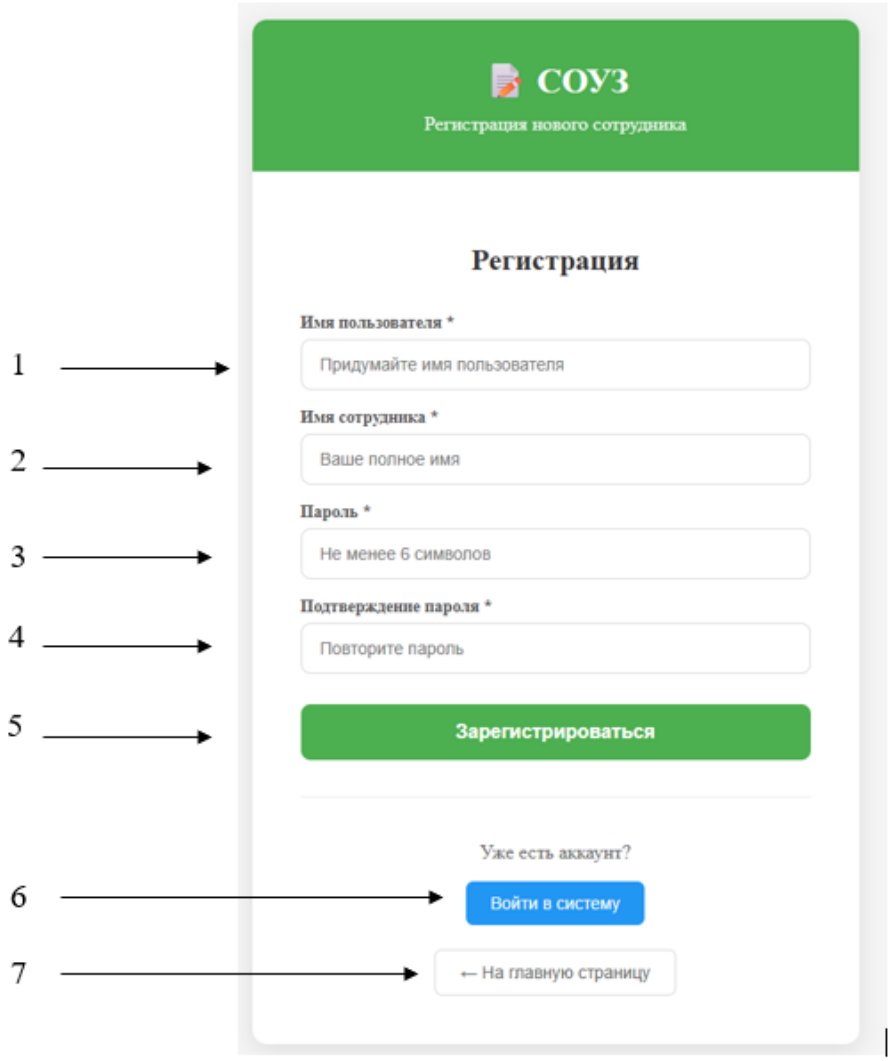

Рисунок 15 – Страница регистрации

Страница создания заказа предназначена для формирования нового заказа путем выбора товарных позиций из справочника по категориям (рисунок 16). Следует отметить, что в реальных условиях эксплуатации функция создания заказов реализуется внешней системой – например, платформой приема заказов

от покупателей, – и поступает в разрабатываемую систему через API. Наличие данной формы в интерфейсе обусловлено необходимостью демонстрации и тестирования алгоритмов распределения и маршрутизации в рамках разработки.

Элементы страницы создания нового заказа включают:

1 – поле для поиска товаров по названию;

2 – панель категорий товаров;

3 – информация о зоне хранения товара;

4 – информация о времени сборки товара;

5 – кнопка для добавления товара в заказ.

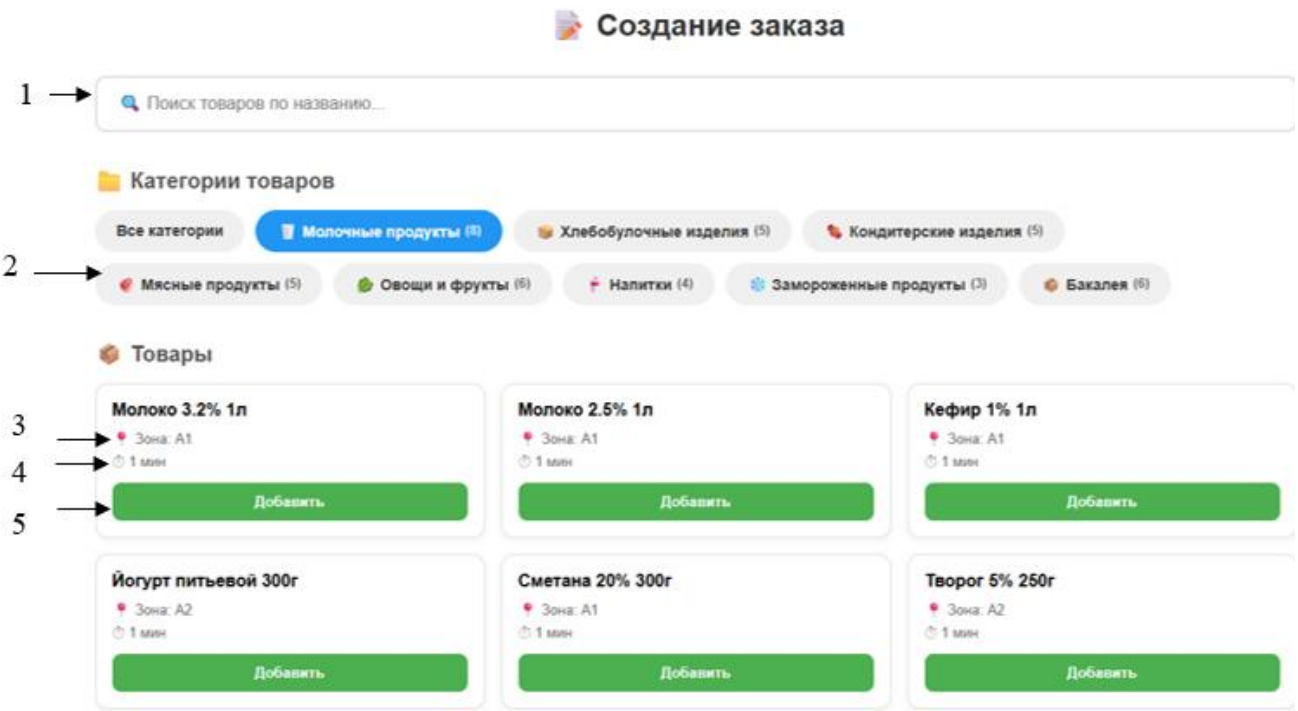

Рисунок 16 – Страница создания заказа

Для мониторинга результатов работы алгоритма распределения реализована страница всех заказов (рисунок 17). Она предоставляет сводную информацию по всем заказам в системе. Данная страница позволяет наглядно убедиться в корректности работы алгоритма балансировки нагрузки: простые заказы назначаются сотрудникам с уровнем «Новичок», а сложные – исполнителям со статусами «Опытный» и «Эксперт».

Элементы страницы создания нового заказа включают:

1 – кнопка для создания нового заказа;

2 – блок информации о заказе (количество товаров, время сборки, тип заказа и сборщик);

3 – информация о статусе заказа;

4 – информация о сложности заказа.

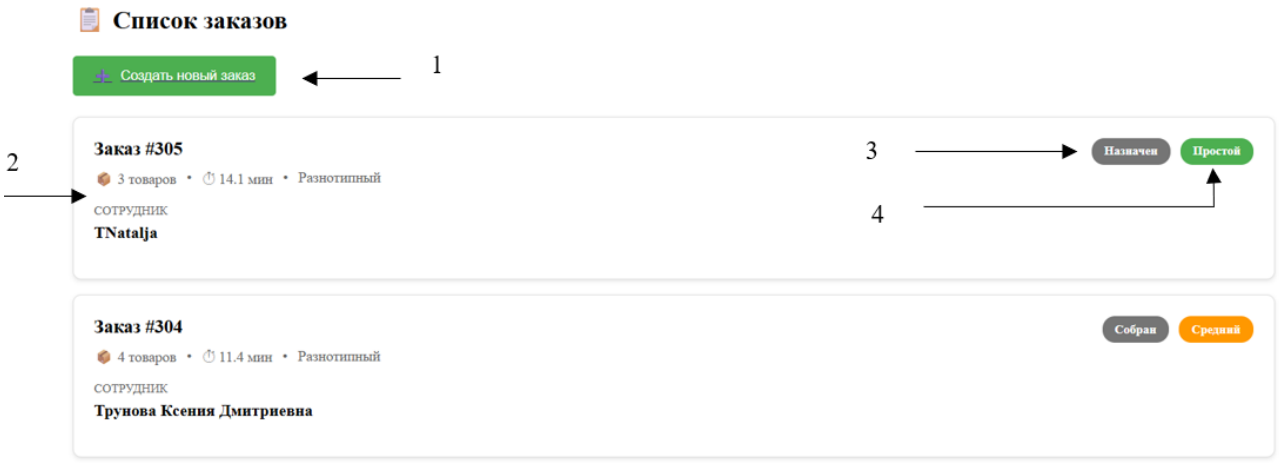

Рисунок 17 – Страница всех заказов

Страница «Мои заказы» разделена на три вкладки: «Новые», «В работе» и «Завершенные» (рисунок 18). Новые назначенные заказы выделяются пульсирующей подсветкой и сопровождаются звуковым уведомлением при получении в фоновом режиме. Для каждого заказа отображается его сложность, тип и расчетное время сборки.

На странице «Мои заказы» представлены:

1 – кнопка для возврата на главную страницу;

2 – кнопка для обновления списка заказов;

3 – информация о количестве новых заказов;

4 – информация о количестве заказов в работе;

5 – информация о количестве завершенных заказов;

6 – уведомление об отсутствии новых заказов.

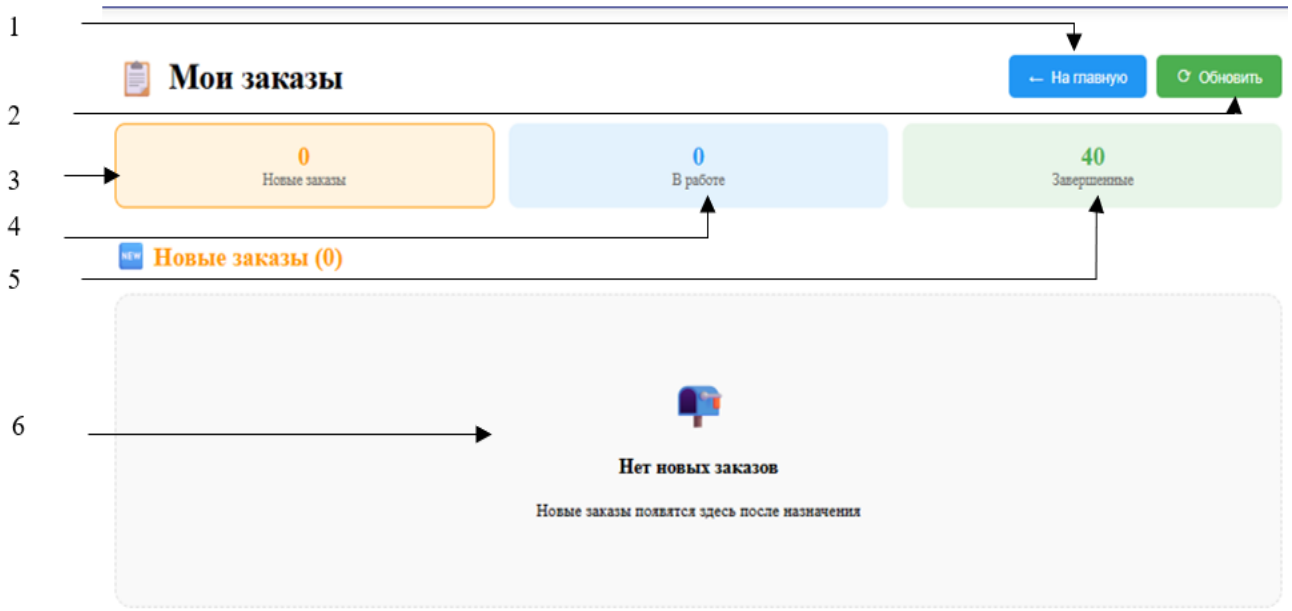

Рисунок 18 – Страница «Мои заказы»

На странице обработки заказа пользователь получает доступ к информации, необходимой для выполнения сборки, что позволяет ему быстро и точно выполнять задачи (рисунок 19). Здесь отображается визуализированный маршрут перемещения по складу, представленный в виде последовательности зон, которые необходимо посетить. Это помогает сотруднику оптимально планировать свои перемещения, сокращая время на поиск нужных товаров и снижая физическую нагрузку. Завершение заказа инициирует обновление его статуса и пересчет текущей загрузки сотрудника.

На странице обработки заказа представлены основные элементы схемы склада и списка заказа.

В схеме склада выделены:

1 – продуктовый стеллаж;

2 – стартовая точка маршрута;

3 –проход между стеллажами;

4 – конечная точка маршрута;

5 – проходной шлюз.

Рисунок 19 – Маршрут заказа

Помимо маршрута, на экране представлен детализированный список товарных позиций (рисунок 20).

В списке заказа представлены:

1 – визуализация маршрута сборки;

2 – кнопка для движения вперед;

3 – кнопка для движения назад;

4 – кнопка для сброса маршрута;

5 – текущая зона сборки;

6 – кнопка для возврата к списку заказов;

7 – кнопка для завершения заказа.

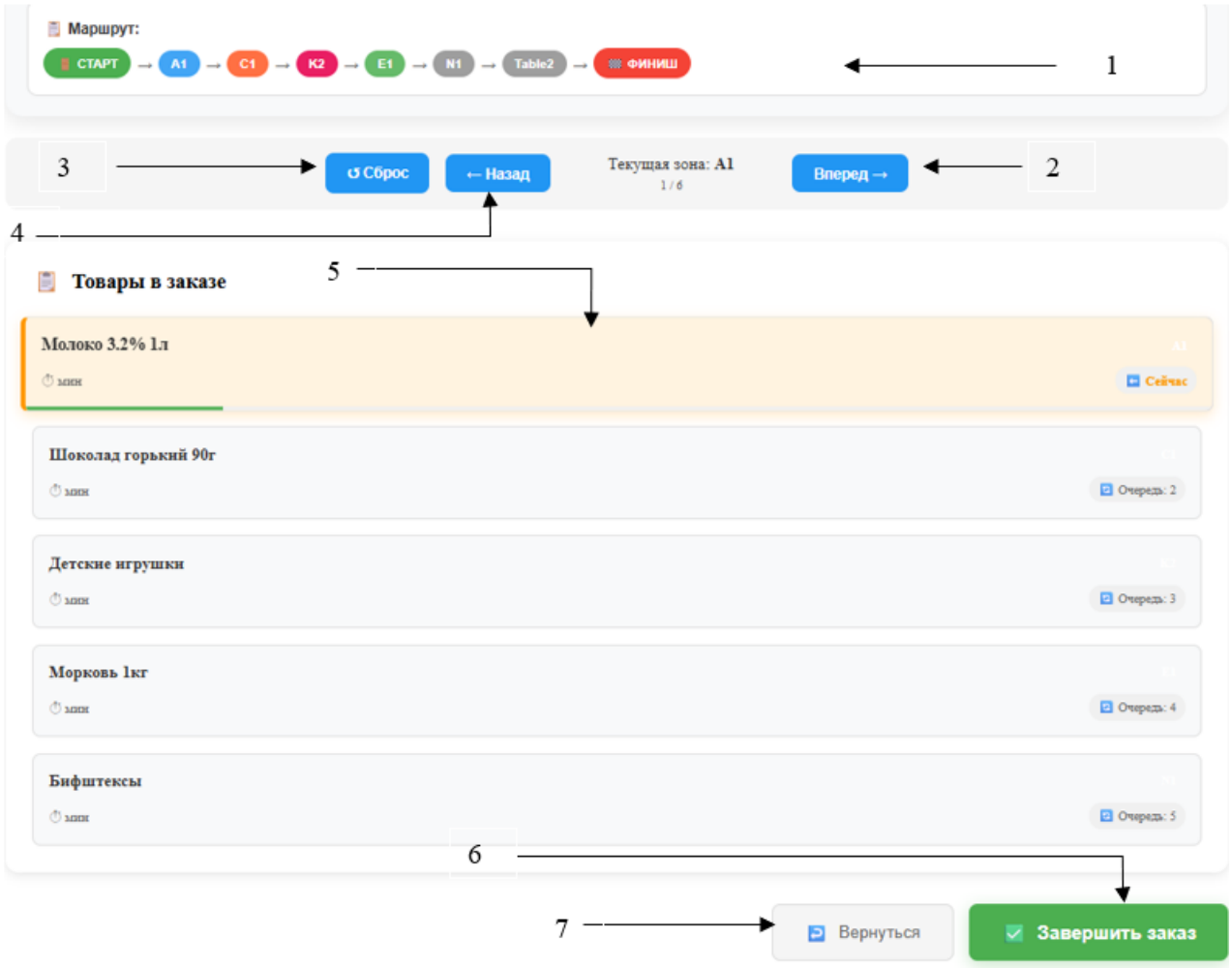

Рисунок 20 – Список заказа

Страница профиля сотрудника предназначена для отображения расширенной аналитики производительности (рисунок 21). На ней представлены ключевые характеристики работы пользователя, включая текущий уровень опыта и позицию в рейтинге, основные показатели эффективности – OPH (количество заказов, выполненных в час) и LUR (коэффициент использования рабочего времени), а также прогресс выполнения дневной и недельной норм. Ниже располагается рейтинговая таблица топ-3 сотрудников, отсортированных по числу выполненных заказов, с указанием квалификации, показателей эффективности и прогресса каждого участника.

Основные элементы страницы «Профиль сотрудника»:

1 – имя сотрудника и его статус занятости;

2 – блок информации о сотруднике, включающий место в рейтинге, уровень опыта и статистику по заказам;

3 – таблица лидерборда с информацией о других сотрудниках.

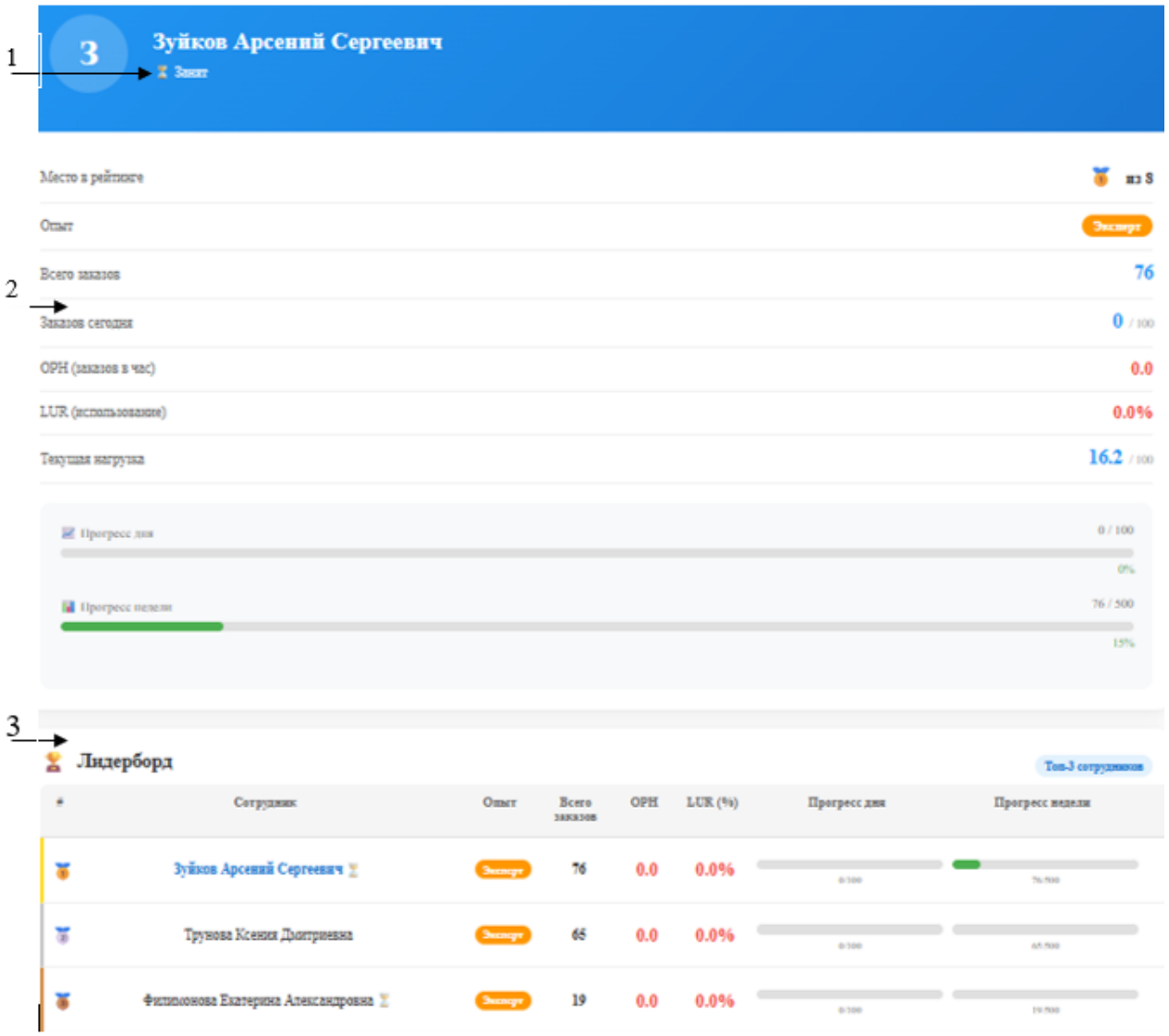

Рисунок 21 – Страница профиля сотрудника

Разработка визуального оформления интерфейса выполнялась с применением встроенных стилей React и CSS-in-JS подхода, что обеспечивает изоляцию стилей компонентов и их независимость от глобального контекста. Навигация между разделами реализована через постоянную верхнюю панель, обеспечивающую быстрый доступ к основным функциям системы и соответствующую общепринятым принципам проектирования пользовательских интерфейсов.

4 Контроль качества программного обеспечения

4.1 Подходы к тестированию и метрики оценки качества кода

Обеспечение качества программного обеспечения является неотъемлемой частью разработки системы автоматизации внутрискладской логистики. Качество программных средств напрямую влияет на надежность работы службы доставки, минимизацию ошибок и повышение удовлетворенности пользователей. В рамках данной работы уделяется особое внимание тестированию и анализу метрик исходного кода, что позволяет гарантировать стабильность и соответствие системы заявленным требованиям.

Для оценки качества программного обеспечения было функциональное и модульные методы тестирования. Функциональное тестирование – проверка соответствия реализованных функций приложения требованиям. Оно позволяет убедиться, что все операции выполняются корректно, а система работает в соответствии с ожиданиями пользователей. Модульное тестирование – проверка отдельных компонентов системы в изоляции от других частей приложения. Это позволяет выявлять ошибки на ранних этапах разработки и обеспечивать стабильную работу модулей.

Для количественной оценки качества кода используются специализированные метрики. Метрика Чепина оценивает логическую сложность программы через анализ количества и вложенности управляющих конструкций. Чем больше вложенность и количество таких конструкций, тем выше сложность кода. Метрика Холстеда основана на анализе количества уникальных операторов и операндов в программе. Она позволяет оценить объем кода, его сложность и усилия, необходимые для разработки и поддержки. Метрика SLOC измеряет количество строк в исходном коде программы, включая комментарии и пустые строки. Эта метрика позволяет оценить объем программного обеспечения и его сложность, что важно для планирования ресурсов и оценки трудозатрат.

4.2 Тестирование программных средств автоматизации внутрискладской логистики в службе экспресс-доставки

4.2.1 Тестирование пользовательской части программного средства автоматизации внутрискладской логистики

Тестирование «белым ящиком» представляет собой подход, при котором анализируется внутренняя структура программного кода. Этот метод позволяет тестировщику изучать логику работы программы, проверять все возможные пути выполнения и выявлять скрытые дефекты. Основное преимущество данного подхода заключается в возможности оптимизации производительности и обеспечения полного покрытия функциональности за счет детального анализа кода.

В процессе проверки клиентской части системы было уделено внимание методу DetermineComplexity, который классифицирует заказы по уровню сложности. Данный метод анализирует два ключевых параметра: количество товарных позиций (itemCount) и однородность заказа (isUniform). На основе этих данных он присваивает заказу одну из трех категорий: «Простой», «Средний» или «Сложный». Исходная реализация метода DetermineComplexity представлена в листинге 1.

Листинг 1 – Реализация метода определения сложности заказа

```
private string DetermineComplexity(Order order,
IReadOnlyDictionary<int, Product> products)
{
    int itemCount = order.Items.Count;
    bool isUniform = DetermineType(order, products) ==
«Однотипный»;

    if (isUniform) // Условие A
    {
        if (itemCount <= 6) // Условие B
            return «Простой»;
        return «Средний»;
    }
    else
    {
        if (itemCount <= 3) // Условие C
```

```
            return «Простой»;
        if (itemCount <= 6) // Условие D
            return «Средний»;
        return «Сложный»;
    }
}
```

Для всесторонней проверки метода был применен структурный подход к тестированию, который включал анализ внутренней логики программы с использованием стратегий покрытия решений, покрытия условий, комбинаторного покрытия условий и графа потока управления.

Покрытие решений – проверка каждого логического условия на истинность и ложность. В частности, анализировались ветвления по параметру isUniform, а также внутренние условия, связанные с количеством товаров (таблица 10).

Таблица 10 – Покрытие решений

№	Условие	Входные данные	Выходные данные
1	a, c, d	isUniform = false item.Count = 10	Сложный

Покрытие условий – тестирование всех возможных исходов для каждого логического выражения. В таблице 11 приведены тестовые случаи для проверки условий.

Таблица 11 – Покрытие условий

№	Решения	Входные данные	Выходные данные
1	a: Да, b: Да	itemCount = 4, isUniform = true	Простой
2	a: Да, b: Нет	itemCount = 8, isUniform = true	Средний
3	b: Нет, c: Нет, d: Да	itemCount = 3, isUniform = false	Средний

Продолжение таблицы 11

4	a: Нет, c: Да	itemCount = 5, isUniform = false	Средний
5	a: Нет, c: Нет, d: Нет	itemCount = 9, isUniform = false	Сложный

Комбинаторное покрытие условий – проверка всех возможных сочетаний логических условий (таблица 12).

Таблица 12 – Комбинаторное покрытие условий

№	Условий	Входные данные	Выходные данные
1	isUniform = Да, itemCount ≤ 6	itemCount = 1, isUniform = true	Простой (рисунок Г.1)
2	isUniform = Да, itemCount > 6	itemCount = 8, isUniform = true	Средний (рисунок Г.2)
3	isUniform = Нет, itemCount ≤ 3	itemCount = 3, isUniform = false	Простой (рисунок Г.3)
4	isUniform = Нет, itemCount > 3, itemCount ≤ 6	itemCount = 5, isUniform = false	Средний (рисунок Г.4)
5	isUniform = Нет, itemCount > 6	itemCount = 9, isUniform = false	Сложный (рисунок Г.5)

Граф потока управления позволяет визуализировать все ветвления и условия, упрощая анализ логики метода и помогая выявить потенциальные ошибки в логике выполнения. Кроме того, он служит основой для проверки полноты тестового покрытия, демонстрируя, какие пути выполнения уже проверены, а какие требуют дополнительного тестирования.

Для наглядного представления логики работы метода был разработан граф потока управления, который иллюстрирует все возможные пути выполнения программы. Этот граф представлен на рисунке 22.

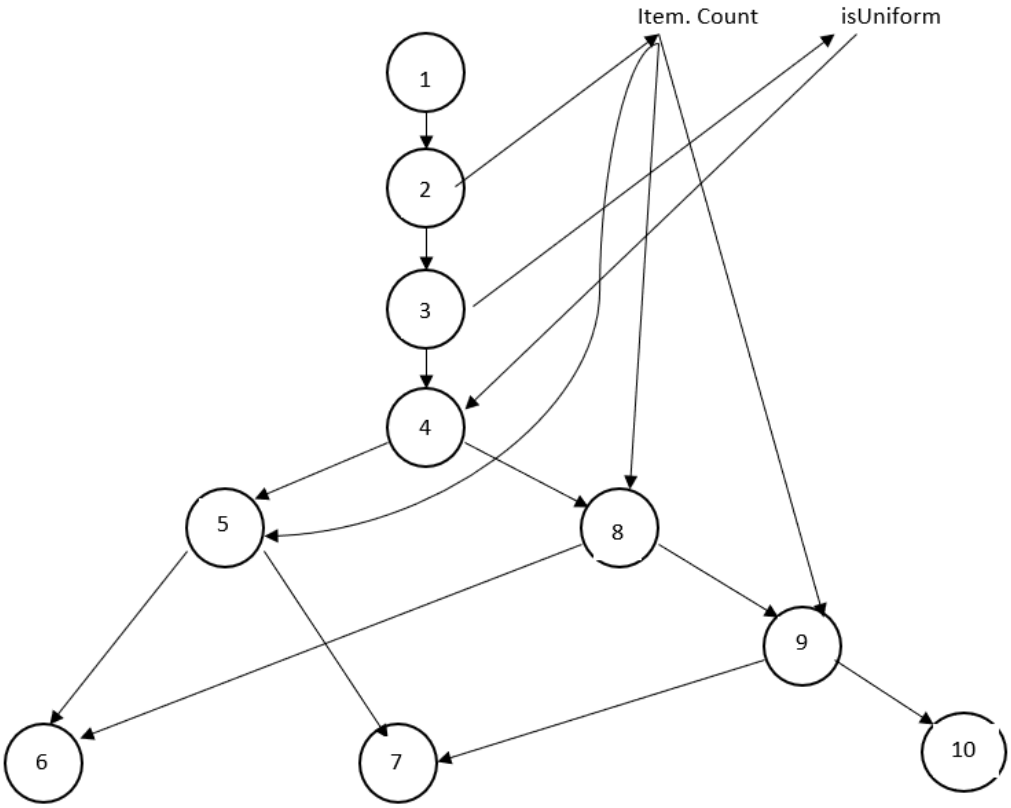

Рисунок 22 – Граф потока управления

Анализ графа подтверждает, что все логические ветвления и условия метода учтены в тестовых случаях, что гарантирует корректность классификации заказов по уровню сложности. Таким образом, структурное тестирование «белым ящиком» обеспечивает высокую надежность реализации метода и минимизирует риск скрытых дефектов.

4.2.2 Тестирование серверной части программного средства автоматизации внутрискладской логистики

В процессе верификации серверной части системы было реализовано модульное тестирование основных компонентов с применением фреймворка xUnit. Тестовые случаи были структурированы в классе AllTests, использующем интерфейс IDisposable для управления ресурсами. Изоляция тестов обеспечивалась за счет использования in-memory базы данных Entity Framework Core, где для каждого теста создавался отдельный экземпляр контекста с уникальным именем базы данных, который удалялся после завершения теста.

Это гарантировало независимость выполнения тестов и стабильность результатов. Всего реализовано восемнадцать тестов, охватывающих основные функциональные подсистемы приложения.

Для сервиса аутентификации было разработано четыре теста, проверяющих корректность регистрации новых сотрудников с автоматическим присвоением начальной квалификации «Новичок», обработку ошибок при несовпадении паролей, успешную авторизацию при вводе корректных учетных данных, а также отказ в доступе при вводе неверного пароля.

Модель сотрудника была проверена тремя тестами, подтверждающими корректность хранения и обработки различных уровней квалификации: «Новичок», «Опытный» и «Эксперт».

Сервисы обработки заказов и балансировки нагрузки были протестированы пятью тестами, охватывающими различные сценарии назначения заказов. В частности, проверялось поведение системы при отсутствии доступных сотрудников, корректное распределение простых заказов сотрудникам с квалификацией «Новичок», сложных заказов – сотрудникам с квалификацией «Эксперт», а также автоматическое повышение квалификации при достижении пороговых значений: переход с уровня «Новичок» на «Опытный» после выполнения пятнадцати заказов и с уровня «Опытный» на «Эксперт» после выполнения тридцати пяти заказов.

Сервис маршрутизации был проверен тремя тестами, обеспечивающими корректность построения маршрутов с учетом структуры склада, температурных зон хранения и выбора соответствующего стола комплектации. Проверялось построение маршрута при отсутствии зон, включение переданной зоны в итоговый маршрут, а также корректность вычисления расстояния между одинаковыми зонами.

Сервис управления сотрудниками был протестирован тремя тестами, проверяющими корректность получения списка всех сотрудников, получение статуса существующего сотрудника, а также обработку запроса при обращении к несуществующему идентификатору (возврат значения null).

Функциональность системы геймификации была проверена в интеграции с механизмом повышения квалификации, где тестировалось изменение уровня сотрудника на основе накопленного опыта и количества выполненных заказов.

Полный код модульных тестов представлен в приложении Д, а результаты их выполнения – на рисунке 23.

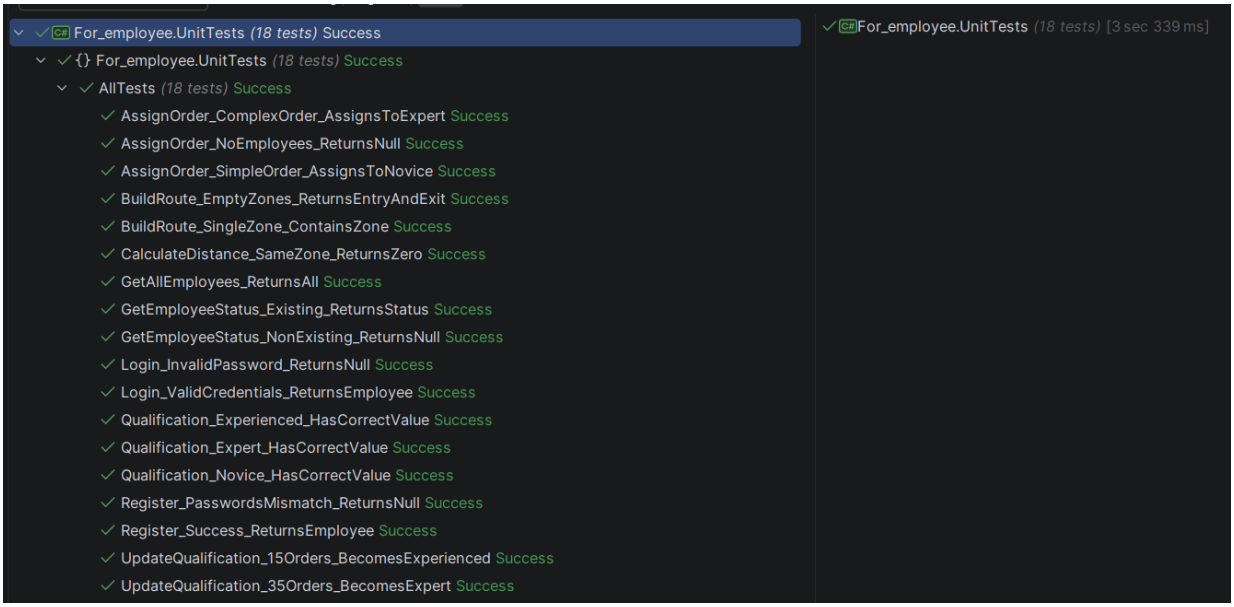

Рисунок 23 – Результат выполнения тестов

Результаты модульного тестирования продемонстрировали корректную реализацию ключевых бизнес-процессов системы. Успешное выполнение всех восемнадцати тестов подтверждает стабильность и надежность разработанных компонентов.

4.3 Оценка качества программного кода

4.3.1 Метрика Чепина

Метрика Чепина служит инструментом для количественной оценки сложности программного кода через анализ его структурных элементов. В рамках исследования системы автоматизации внутрискладской логистики была проведена оценка метода DetermineComplexity, ответственного за классификацию заказов по уровню сложности на основе количества позиций и их однородности.

Для вычисления данной метрики применяется специализированная формула, учитывающая различные типы управляющих конструкций с соответствующими весовыми коэффициентами:

$$Q = 1 \times P + 2 \times M + 3 \times C + 0.5 \times T,$$

где P – количество процедурных вызовов;

M – количество условных операторов;

C – количество операторов выбора;

T – количество неиспользуемых переменных.

Подставив значения параметров в формулу, получаем следующее значение метрики:

$$Q = 1 \times 1 + 2 \times 4 + 3 \times 0 + 0.5 \times 0 = 9$$

Результат расчета метрики Чепина позволяет сделать вывод, что метод обладает невысокой логической сложностью. В его структуре присутствует ограниченное количество управляющих конструкций, а ветвление реализовано простым образом – без использования операторов выбора и циклических конструкций. Это способствует улучшению читаемости кода и упрощает его сопровождение

4.3.2 Метрика Холстеда

Для оценки метрик Холстеда была выбрана функция DetermineComplexity. Данный метод предназначен для определения категории сложности заказа в системе автоматизации внутрискладской логистики на основе двух параметров: количества позиций (itemCount) и их однотипности (isUniform).

Для расчета метрики Холстеда необходимо определить:

- количество уникальных операторов (N1) и операндов (N2);
- общее количество операторов (n1) и операндов (n2).

В таблице 13 представлены операторы функции DetermineComplexity и количество их вхождений.

Таблица 13 – Операторы функции

№	Оператор	Количество вхождений
1	private	1
2	string	1
3	(	5
4	)	5
5	{	3
6	}	3
7	int	1
8	=	2
9	.	2
10	;	7
11	bool	1
12	DetermineType	1
13	==	1
14	if	4
15	<=	3
16	return	5
17	else	1
	n1 = 17	N1 = 46

Операнды функции и число их вхождений приведены в таблице 14.

Таблица 14 – Операнды функции

№	Операнды	Количество вхождений
1	itemCount	4
2	order	2
3	isUniform	2
4	6	2
5	3	1
6	Простой	2
7	Средний	2
8	Сложный	1
9	products	1
10	Однотипный	1
	n2 = 10	N2 = 18

На основе детального анализа исходного кода функции DetermineComplexity были рассчитаны ключевые характеристики программы с применением формул метрики Холстеда. Результаты вычислений представлены в таблице 15.

Таблица 15 – Результаты вычислений по метрике Холстеда

Параметр	Значение
Словарь программы (n)	27
Длина программы (N)	64
Объем программы (V)	304
Уровень сложности (D))	15.3
Уровень трудности (E)	0.67

Полученные значения метрик Холстеда позволяют сделать вывод о том, что функция DetermineComplexity имеет компактную и хорошо структурированную реализацию. Небольшой объем словаря программы и низкий уровень трудности свидетельствуют о высокой читаемости кода и простоте его сопровождения. Умеренный уровень сложности подтверждает, что алгоритм не перегружен избыточными конструкциями, что облегчает его интеграцию в систему автоматизации внутрискладской логистики. Таким образом, анализ метрик Холстеда демонстрирует, что данный метод является эффективным и пригодным для практического применения в разрабатываемой системе.

4.3.3 Метрика SLOC

Для оценки размера исходного кода был выбран метод DetermineComplexity, который отвечает за определение категории сложности заказа в системе автоматизации внутрискладской логистики.

В результате анализа было выявлено следующее распределение строк кода:

- 25 физических строк – общее количество строк в файле;
- 12 логических строк – количество строк, содержащих исполняемый код;
- 2 строки комментариев – документация и пояснения к коду.

Низкие значения данных показателей указывают на компактность реализации алгоритма. Это позволяет сделать вывод о минимальных трудозатратах на разработку, тестирование и дальнейшее сопровождение метода. Компактность кода также способствует его легкой интеграции в систему и упрощает процесс модификации при необходимости.

Заключение

В рамках выпускной квалификационной работы спроектированы и разработаны программные средства автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов питания. Результаты данной работы были опубликованы в статье [20].

На этапе анализа предметной области были исследованы ключевые процессы внутрискладской логистики, выявлены три группы системных проблем: неоптимальная маршрутизация сборки, несбалансированное распределение заказов между сборщиками и отсутствие инструментов оперативного мониторинга производительности персонала. Проведен сравнительный анализ трех существующих программных решений – «AXELOT WMS», «Consid.WMS», «1С:WMS Логистика. Управление складом», – по результатам которого установлена нецелесообразность их применения в условиях малого и среднего бизнеса ввиду высокой стоимости и избыточной функциональности. Сформулированы функциональные и нефункциональные требования к разрабатываемой системе.

Выполнено планирование и оценка стоимости разработки с применением инструмента ProjectLibre. В результате декомпозиции проектных работ выделено двадцать девять самостоятельных заданий. Начало работ определено как 8 апреля 2026 года, завершение – 4 июня 2026 года. Совокупные трудозатраты составили 392 часа, расчетная стоимость проекта – 196 000 рублей.

На этапе проектирования разработаны концептуальная, логическая и физическая модели базы данных, спроектирована архитектура системы на основе разделения серверной и клиентской частей, построены UML-диаграммы вариантов использования, последовательности, классов и компонентов. Определены требования к пользовательскому интерфейсу с учетом специфики работы сборщика склада.

Реализация серверной части выполнена на языке C# с использованием платформы ASP.NET Core Web API. Разработаны три ключевых сервиса:

OrderService, реализующий классификацию заказов по сложности и типу с расчетом нормативного времени сборки, LoadBalancingService, обеспечивающий распределение заказов с учетом квалификации и текущей нагрузки сотрудников и автоматическое повышение квалификационного уровня, и RoutingService, строящий оптимальный маршрут обхода складских зон. Физическая схема базы данных развернута средствами PostgreSQL с применением механизма миграций Entity Framework Core. Клиентская часть реализована в виде одностраничного приложения на React с использованием TypeScript.

Контроль качества разработанного программного обеспечения проводился с применением функционального и модульного методов тестирования по критериям покрытия операторов и покрытия решений. Объектами тестирования выступили метод DetermineComplexity класса OrderService и класс LoadBalancingService. Тестирование осуществлялось с использованием фреймворка xUnit. Дополнительно выполнена оценка метрик качества кода: метрики Чепина, метрики Холстеда и SLOC. По результатам тестирования подтверждена корректность реализованной бизнес-логики и соответствие разработанного программного обеспечения сформулированным требованиям.

Таким образом, в результате выполнения выпускной квалификационной работы разработаны программные средства, обеспечивающие автоматизацию ключевых процессов внутрискладской логистики: алгоритмическое распределение заказов между сборщиками с учетом их опыта и загруженности, построение оптимальных маршрутов сборки и мониторинг производительности персонала. Разработанная система ориентирована на предприятия малого и среднего бизнеса и позволяет сократить временные затраты на сборку заказов, минимизировать ошибки ручного распределения и обеспечить руководство склада актуальными данными о работе персонала.

Список использованных источников

1. Автоматизация склада – WMS // CNews [Электронный ресурс]. – URL: https://www.cnews.ru/book (дата обращения: 14.04.2026).

2. Маликова, Т. Е. Складская логистика: учебник / Т. Е. Маликова. – 2- изд. – Москва: Юрайт, 2025. – 115 с.

3. Гаджинский, А. М. Логистика / А. М. Гаджинский. – Москва: Дашков и К, 2021. – 512 с.

4. AXELOT WMS X: описание, функции и интерфейс // Soware [Электронный ресурс]. – URL: https://soware.ru/products/axelot-wms-x (дата обращения: 14.04.2026).

5. Consid.WMS: WMS система управления складом // Consid [Электронный ресурс]. – URL: https://www.considwms.ru (дата обращения: 14.04.2026).

6. 1С:Предприятие. Конфигурация WMS Логистика. Управление складом, редакция 4.0: сведения о версии 4.0.2 // Фирма «1С» [Электронный ресурс]. – URL: http://downloads.v8.1c.ru/content//WMS4/4_0_2_2/news.htm (дата обращения: 15.04.2026).

7. Пахомова, Ю. Что такое функция и функциональные требования? // Habr [Электронный ресурс]. – 2026. – URL: https://habr.com/ru/articles/1005192/ (дата обращения: 15.04.2026).

8. UML – диаграмма вариантов использования (use case diagram) // Habr [Электронный ресурс]. – URL: https://habr.com/ru/articles/47940/ (дата обращения: 17.04.2026).

9. Пономарева, О. Как правильно формулировать нефункциональные требования // Habr [Электронный ресурс]. – 2025. – URL: https://habr.com/ru/articles/948506/ (дата обращения: 17.04.2026).

10. Чусавитина, Г. Н. Управление образовательными проектами с использованием свободного программного обеспечения ProjectLibre: учебное

пособие / Г. Н. Чусавитина. – Магнитогорск: МГТУ им. Г. И. Носова, 2019. – 166 с.

11. Клаверов, В. Б. Управление проектами. Кейс практического обучения: учебное пособие / В. Б. Клаверов. – Саратов: Ай Пи Эр Медиа, 2018. – 142 с.

12. Гагарина, Л. Г. Проектирование и архитектура программных систем: учебное пособие / Л. Г. Гагарина, А. Р. Федоров, П. А. Федоров. – 2-е изд., испр. и доп. – Москва: ИНФРА-М, 2025. – 334 с.

13. Microsoft Learn. Разработка приложений ASP.NET Core MVC: официальная документация // Microsoft Corporation [Электронный ресурс]. – URL: https://learn.microsoft.com/ru-ru/aspnet/core/mvc/ (дата обращения: 17.04.2026).

14. React Documentation: официальный сайт // Meta Platforms, Inc. [Электронный ресурс]. – 2026. – URL: https://react.dev/ (дата обращения:19.04.2026).

15. Visual Paradigm. Conceptual, Logical and Physical Data Model // Visual Paradigm [Электронный ресурс]. – URL: https://circle.visual-paradigm.com/docs/database-design-engineering/database-designers-guide/conceptual-logical-and-physical-data-model/ (дата обращения: 19.04.2026).

16. Моделирование данных: обзор // Habr [Электронный ресурс]. – URL: https://habr.com/ru/articles/556790/ (дата обращения: 19.04.2026).

17. UML диаграмма классов // GeeksforGeeks [Электронный ресурс]. – URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/ (дата обращения: 20.04.2026).

18. Нильсен, Я. 10 эвристик для дизайна пользовательского интерфейса // Nielsen Norman Group [Электронный ресурс]. – 1994. – URL: https://www.nngroup.com/articles/ten-usability-heuristics/ (дата обращения: 20.04.2026).

19. Трунова, К. Д. Программные средства автоматизации процессов внутрискладской логистики в службе экспресс-доставки продуктов: приложение

к бакалаврской работе // GitHub [Электронный ресурс]. – URL: https://github.com/Ksusha2112/warehouse-logistics-automation (дата обращения: 20.05.2026).

20. Трунова, К. Д. Разработка системы оперативного управления заказами для минимизации времени цикла сборки в службе экспресс-доставки продуктов / И. Ю. Балашова, К. Д. Трунова // Информационные технологии в моделировании и управлении: подходы, методы, решения: сборник статей IX Всероссийской научной конференции с международным участием. – Тольятти: Изд-во Тольяттинского государственного университета, 2026.

Приложение А

(справочное)

Глоссарий

API (Application Programming Interface) – программный интерфейс взаимодействия между приложениями, определяющий набор правил и протоколов для обмена данными между программными компонентами.

ASP.NET Core – кроссплатформенная среда с открытым исходным кодом для создания высокопроизводительных веб-приложений и API-сервисов на языке C#. Поддерживает встроенное внедрение зависимостей и асинхронную обработку запросов.

CORS (Cross-Origin Resource Sharing) – механизм управления доступом к ресурсам веб-сервера с других доменов. Применяется для разрешения обращений клиентской части React к серверному API.

CSS-in-JS – подход к стилизации компонентов в React, при котором стили описываются непосредственно в JavaScript-коде, что обеспечивает их изоляцию и независимость от глобального контекста страницы.

Entity Framework Core – ORM-фреймворк для работы с реляционными базами данных в экосистеме .NET. Позволяет описывать структуру данных в виде классов C# и управлять схемой базы данных через механизм миграций.

HTTP (HyperText Transfer Protocol) – протокол прикладного уровня для передачи данных между клиентом и сервером в веб-приложениях. Используется для взаимодействия клиентской части React с серверным REST API.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на синтаксисе JavaScript. Применяется как формат передачи данных между клиентской и серверной частями разрабатываемой системы.

LUR (Labour Utilization Rate) – коэффициент использования рабочего времени сотрудника, рассчитываемый как отношение суммарного расчетного времени выполненных заказов к фактически отработанному времени.

MVC (Model-View-Controller) – архитектурный паттерн, разделяющий приложение на три компонента: модель данных, представление и контроллер. В разрабатываемой системе уровень представления вынесен в клиентское React-приложение, а контроллеры возвращают данные в формате JSON.

OPH (Orders Per Hour) – показатель производительности сотрудника, определяемый как количество заказов, выполненных за один час рабочего времени. Рассчитывается как отношение числа завершенных заказов к продолжительности рабочего дня (10 часов).

ORM (Object-Relational Mapping) – технология программирования, обеспечивающая автоматическое преобразование данных между реляционной базой данных и объектами языка программирования. В системе реализована посредством Entity Framework Core.

pgAdmin – графическая среда администрирования и выполнения запросов к СУБД PostgreSQL. Используется для управления физической схемой базы данных разрабатываемой системы.

PostgreSQL – свободная объектно-реляционная система управления базами данных с открытым исходным кодом. Применяется в разрабатываемой системе в качестве основного хранилища данных о сотрудниках, заказах и товарах.

ProjectLibre – кроссплатформенная система управления проектами с открытым исходным кодом, используемая для планирования разработки, построения диаграммы Ганта и оценки стоимости проекта.

React – библиотека JavaScript для построения пользовательских интерфейсов, основанная на компонентном подходе. Используется для реализации клиентской части системы в виде одностраничного приложения.

React Router – библиотека маршрутизации для React-приложений, обеспечивающая навигацию между страницами на стороне браузера без полной перезагрузки страницы.

REST API (Representational State Transfer) – архитектурный стиль взаимодействия между клиентом и сервером, основанный на использовании стандартных HTTP-методов для работы с ресурсами. Применяется в серверной части разрабатываемой системы.

SLOC (Source Lines of Code) – метрика оценки объема программного обеспечения, измеряющая количество строк исходного кода. Применяется для оценки трудозатрат на разработку и поддержку программного продукта.

SPA (Single Page Application) – одностраничное веб-приложение, в котором навигация между разделами осуществляется без перезагрузки страницы путем динамического обновления содержимого. Клиентская часть разрабатываемой системы реализована по данному принципу.

СУБД (Система управления базами данных) – совокупность программных средств для создания, хранения и управления структурированными данными. В разрабатываемой системе функции СУБД выполняет PostgreSQL.

TypeScript – строго типизированный язык программирования, являющийся надмножеством JavaScript. Применяется в клиентской части разрабатываемой системы для повышения надежности кода за счет статической типизации.

UML (Unified Modeling Language) – унифицированный язык моделирования для визуального представления архитектуры программных систем. В работе применяются диаграммы вариантов использования, последовательности, классов и компонентов.

WMS (Warehouse Management System) – система управления складом – программный комплекс для автоматизации складских процессов: приемки товаров, их размещения, комплектации заказов и отгрузки.

xUnit – фреймворк для написания и выполнения модульных тестов в экосистеме .NET. Применяется для тестирования сервисов OrderService и LoadBalancingService разрабатываемой системы.

Веб-приложение – клиент-серверное программное решение, доступ к которому осуществляется через браузер посредством передачи данных по протоколу HTTP. Разрабатываемая система реализована как монолитное веб-приложение с локальным развертыванием.

Геймификация – применение игровых механик (значки, рейтинги, прогресс-бары) в неигровом контексте для повышения мотивации пользователей. В разрабатываемой системе реализована через систему значков и таблицу лидеров.

Диаграмма Ганта – инструмент визуального представления календарного плана проекта в виде горизонтальных полос на временно́й шкале. По

вертикальной оси перечислены задачи, по горизонтальной – сроки их выполнения.

Миграция базы данных – механизм Entity Framework Core для управления изменениями в схеме базы данных. Каждое изменение фиксируется в виде отдельного файла и применяется к целевой базе данных в строго определенной последовательности.

Фреймворк – программная платформа, задающая архитектурные принципы и общие правила разработки приложений в определенной предметной области, и предоставляющая набор готовых инструментов и библиотек.

Приложение Б

(обязательное)

Листинг приложения

Листинг Б.1 – Файл AuthController.cs

```
using Microsoft.AspNetCore.Mvc;
using For_employee.DTO;
using For_employee.Modules.Auth;
using For_employee.Modules.Gamification;

namespace For_employee.Controllers
{
    [ApiController]
    [Route(«api/[controller]»)]
    public class AuthController : ControllerBase
    {
        private readonly AuthService _authService;
        public AuthController(AuthService authService)
        {
            _authService = authService;
        }

        [HttpPost(«register»)]
        public async Task<IActionResult> Register([FromBody]
RegisterRequest request)
        {
            try
            {
                var employee = await
_authService.RegisterAsync(request);
                if (employee == null)
                    return BadRequest(new { Message = «Пароли не
совпадают или пользователь уже существует» });

                var response =
_authService.BuildEmployeeResponse(employee); // без badge

                return Ok(response);
            }
            catch (Exception ex)
            {
                return StatusCode(500, new { Message = «Ошибка
сервера», Error = ex.Message });
            }
        }

        [HttpPost(«login»)]
        public async Task<IActionResult> Login([FromBody]
LoginRequest request)
        {
            try
            {
                var employee = await
_authService.LoginAsync(request);
                if (employee == null)
                    return Unauthorized(new { Message = «Неверный
```

```
логин или пароль» });
                var response =
_authService.BuildEmployeeResponse(employee); // без badge

                return Ok(response);
            }
            catch (Exception ex)
            {
                Console.WriteLine($»Ошибка при входе:
{ex.Message}»);
                return StatusCode(500, new { Message = «Ошибка
сервера», Error = ex.Message });
            }
        }

        [HttpGet(«profile/{id}»)]
        public async Task<IActionResult> GetProfile(int id)
        {
            try
            {
                var employee = await
_authService.GetEmployeeByIdAsync(id);
                if (employee == null)
                    return NotFound(new { Message = «Сотрудник не
найден» });

                var response =
_authService.BuildEmployeeResponse(employee);

                return Ok(response);
            }
            catch (Exception ex)
            {
                return StatusCode(500, new { Message = «Ошибка
сервера», Error = ex.Message });
            }
        }

        [HttpGet(«all»)]
        public async Task<IActionResult> GetAllEmployees()
        {
            try
            {
                var employees = await
_authService.GetAllEmployeesAsync();
                return Ok(employees);
            }
            catch (Exception ex)
            {
                return StatusCode(500, new { Message = «Ошибка
сервера», Error = ex.Message });
            }
```

```
        }
    }
}
```

Листинг Б.2 – Файл EmployeesController.cs

```
using Microsoft.AspNetCore.Mvc;
using For_employee.DTO;
using For_employee.Models;
using For_employee.Modules.Employees;
using For_employee.Modules.Gamification;
using For_employee.Modules.LoadBalancing;

namespace For_employee.Controllers
{
    [ApiController]
    [Route(«api/[controller]»)]
    public class EmployeesController : ControllerBase
    {
        private readonly EmployeeService _employeeService;
        private readonly GamificationService _gamificationService;
        private readonly LoadBalancingService _loadService;

        public EmployeesController(
            EmployeeService employeeService,
            GamificationService gamificationService,
            LoadBalancingService loadBalancingService)
        {
            _employeeService = employeeService;
            _gamificationService = gamificationService;
            _loadService = loadBalancingService;
        }

        [HttpGet]
        public async Task<IActionResult> GetAllEmployees()
        {
            var employees = await
_employeeService.GetAllEmployeesAsync();
            return Ok(employees);
        }

        [HttpGet(«leaderboard»)]
        public async Task<IActionResult> GetLeaderboard()
        {
            var employees = await
_employeeService.GetAllEmployeesAsync();
            var leaderboard = await
_gamificationService.GetLeaderboardDataAsync(employees);
            return Ok(leaderboard);
        }

        [HttpPost(«{id}/complete»)]
```

```
        public async Task<IActionResult> Complete(int id)
        {
            try
            {
                var result = await
_employeeService.CompleteOrderAsync(id);
                if (!result.Success)
                    return NotFound(new { message = result.Message
});

                return Ok(new
                {
                    message = result.Message,
                    employeeId = result.EmployeeId,
                    employeeName = result.EmployeeName,
                    qualification = result.Qualification,
                    completedOrders = result.CompletedOrders
                });
            }
            catch (Exception ex)
            {
                return StatusCode(500, new { message = «Ошибка
сервера», error = ex.Message });
            }
        }

        [HttpGet(«{id}/status»)]
        public async Task<IActionResult> GetEmployeeStatus(int id)
        {
            var result = await
_employeeService.GetEmployeeStatusAsync(id);
            if (result == null)
                return NotFound(new { message = «Сотрудник не
найден» });

            return Ok(result);
        }


            var stats = await
_gamificationService.GetEmployeeStatsAsync(employee, employees);
            return Ok(stats);
        }

        [HttpGet(«{id}/dashboard»)]
        public async Task<IActionResult> GetDashboard(int id)
        {
            var result = await
_employeeService.GetEmployeeDashboardAsync(id);

            if (result == null)
                return NotFound(new { message = «Сотрудник не
```

```
найден» });

            return Ok(result);
        }
    }
}
```

Листинг Б.3 – Файл OrderController.cs

```
using Microsoft.AspNetCore.Mvc;
using For_employee.Modules.Orders;
using For_employee.Modules.LoadBalancing;
using For_employee.Modules.Routing;
using For_employee.DTO;

namespace For_employee.Controllers
{
    [ApiController]
    [Route(«api/[controller]»)]
    public class OrdersController : ControllerBase
    {
        private readonly OrderService _orderService;
        private readonly LoadBalancingService _loadService;
        private readonly RoutingService2 _routingService;

        public OrdersController(
            OrderService orderService,
            LoadBalancingService loadService,
            RoutingService2 routingService)
        {
            _orderService = orderService;
            _loadService = loadService;
            _routingService = routingService;
        }

        [HttpGet]
        public async Task<IActionResult> GetAllOrders()
        {
            var result = await _orderService.GetAllOrdersAsync();
            return Ok(result);
        }

        [HttpGet(«{id:int}»)]
        public async Task<IActionResult> GetOrder(int id)
        {
            var result = await
_orderService.GetOrderByIdAsync(id);

            if (result == null)
                return NotFound(new { message = $»Заказ с ID {id}
не найден» });

            return Ok(result);
```

```
        }

        [HttpPost]
        public async Task<IActionResult> CreateOrder([FromBody]
OrderRequest request)
        {
            try
            {
                var result = await
_orderService.CreateOrderFullAsync(request);

                return Ok(result);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(new { message = ex.Message });
            }
            catch (Exception ex)
            {
                return StatusCode(500, new { message = «Ошибка
сервера», error = ex.Message });
            }
        }

        [HttpGet(«my-orders/{employeeId:int}»)]
        public async Task<IActionResult> GetEmployeeOrders(int
employeeId)
        {
            var result = await
_orderService.GetEmployeeOrdersAsync(employeeId);
            return Ok(result);
        }

        [HttpGet(«new-for-employee/{employeeId:int}»)]
        public async Task<IActionResult> GetNewOrders(int
employeeId)
        {
            var result = await
_orderService.GetNewOrdersAsync(employeeId);
            return Ok(result);
        }

        [HttpGet(«{id}/route»)]
        public async Task<IActionResult> GetRoute(int id)
        {
            var result = await
_orderService.GetOrderRouteAsync(id);

            if (result == null)
                return NotFound(new { message = «Заказ не найден»
});
```

```
            return Ok(result);
        }

        [HttpPut(«accept/{orderId:int}»)]
        public async Task<IActionResult> AcceptOrder(int orderId)
        {
            var result = await
_orderService.AcceptOrderAsync(orderId);

            if (!result.Success)
                return BadRequest(new { message = result.Message
});

            return Ok(result);
        }

        [HttpPut(«complete/{orderId:int}»)]
        public async Task<IActionResult> CompleteOrder(int
orderId)
        {
            var result = await
_orderService.CompleteOrderAsync(orderId);

            if (!result.Success)
                return BadRequest(new { message = result.Message
});

            return Ok(result);
        }
        [HttpGet(«check-new/{employeeId:int}»)]
        public async Task<IActionResult> CheckNewOrders(int
employeeId)
        {
            var result = await
_orderService.CheckNewOrdersAsync(employeeId);
            return Ok(result);
        }

        [HttpGet(«completed/{employeeId:int}»)]
        public async Task<IActionResult> GetCompletedOrders(int
employeeId)
        {
            var result = await
_orderService.GetCompletedOrdersAsync(employeeId);
            return Ok(result);
        }

        [HttpGet(«active-count/{employeeId:int}»)]
        public async Task<IActionResult> GetActiveCount(int
employeeId)
        {
            var result = await
```

```
_orderService.GetActiveOrdersCountAsync(employeeId);
            return Ok(result);
        }

        [HttpGet(«status/{status}»)]
        public async Task<IActionResult> GetByStatus(string
status)
        {
            var result = await
_orderService.GetOrdersByStatusAsync(status);
            return Ok(result);
        }

        [HttpGet(«employee/{employeeId:int}/status/{status}»)]
        public async Task<IActionResult>
GetEmployeeOrdersByStatus(int employeeId, string status)
        {
            var result = await
_orderService.GetEmployeeOrdersByStatusAsync(employeeId, status);
            return Ok(result);
        }

        [HttpDelete(«{id:int}»)]
        public async Task<IActionResult> DeleteOrder(int id)
        {
            var result = await _orderService.DeleteOrderAsync(id);

            if (!result.Success)
                return NotFound(new { message = result.Message });

            return Ok(result);
        }

    }
}
```

Листинг Б.4 – Файл EmployeeService.cs

```
using Microsoft.EntityFrameworkCore;
using For_employee.Data;
using For_employee.Models;
using For_employee.DTO;
using For_employee.Modules.LoadBalancing;

namespace For_employee.Modules.Employees
{
    public class EmployeeService
    {
        private readonly AppDbContext _dbContext;
        private readonly LoadBalancingService
_loadBalancingService;

        public EmployeeService(
```

```
            AppDbContext dbContext,
            LoadBalancingService loadBalancingService)
        {
            _dbContext = dbContext;
            _loadBalancingService = loadBalancingService;
        }

        public async Task<List<Employee>> GetAllEmployeesAsync()
        {
            return await _dbContext.Employees.ToListAsync();
        }

        public async Task<EmployeeStatusResponse?>
GetEmployeeStatusAsync(int id)
        {
            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return null;

            var hasActiveOrder = await _dbContext.Orders
                .AnyAsync(o => o.EmployeeId == id && (o.Status ==
«Принят» || o.Status == «В сборке»));

            return new EmployeeStatusResponse
            {
                Id = employee.Id,
                Name = employee.Name,
                Status = employee.Status,
                CurrentLoad = employee.CurrentLoad,
                HasActiveOrder = hasActiveOrder
            };
        }

        public async Task<BreakResponse> StartBreakAsync(int id)
        {
            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return new BreakResponse
                {
                    Success = false,
                    Message = «Сотрудник не найден»
                };

            var hasActiveOrder = await _dbContext.Orders
                .AnyAsync(o => o.EmployeeId == id && (o.Status ==
«Принят» || o.Status == «В сборке»));

            if (hasActiveOrder)
                return new BreakResponse
                {
```

```
                    Success = false,
                    Message = «❌ Нельзя уйти на перерыв, пока
есть активный заказ»
                };

            if (employee.Status == «Перерыв»)
                return new BreakResponse
                {
                    Success = false,
                    Message = «❌ Сотрудник уже на перерыве»
                };

            employee.Status = «Перерыв»;
            await _dbContext.SaveChangesAsync();

            return new BreakResponse
            {
                Success = true,
                Message = «✅ Сотрудник ушел на перерыв»,
                EmployeeId = employee.Id,
                EmployeeName = employee.Name,
                Status = employee.Status,
                BreakStartTime = DateTime.Now
            };
        }

        public async Task<BreakResponse> EndBreakAsync(int id)
        {
            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return new BreakResponse
                {
                    Success = false,
                    Message = «Сотрудник не найден»
                };

            if (employee.Status != «Перерыв»)
                return new BreakResponse
                {
                    Success = false,
                    Message = «❌ Сотрудник не на перерыве»
                };

            employee.Status = «Свободен»;
            await _dbContext.SaveChangesAsync();

            return new BreakResponse
            {
                Success = true,
                Message = «✅ Сотрудник вернулся с перерыва»,
```

```
                EmployeeId = employee.Id,
                EmployeeName = employee.Name,
                Status = employee.Status,
                BreakEndTime = DateTime.Now
            };
        }

        public async Task<BreakStatusResponse?>
GetBreakStatusAsync(int id)
        {
            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return null;

            var hasActiveOrder = await _dbContext.Orders
                .AnyAsync(o => o.EmployeeId == id && (o.Status ==
«Принят» || o.Status == «В сборке»));

            return new BreakStatusResponse
            {
                Id = employee.Id,
                Name = employee.Name,
                IsOnBreak = employee.Status == «Перерыв»,
                Status = employee.Status,
                CanTakeBreak = employee.Status != «Перерыв» &&
!hasActiveOrder
            };
        }

        public async Task<UpdateStatusResponse>
UpdateEmployeeStatusAsync(int id, string newStatus)
        {
            var validStatuses = new[] { «Свободен», «Занят»,
«Перерыв» };
            if (!validStatuses.Contains(newStatus))
                return new UpdateStatusResponse
                {
                    Success = false,
                    Message = «Некорректный статус»
                };

            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return new UpdateStatusResponse
                {
                    Success = false,
                    Message = «Сотрудник не найден»
                };

            if (newStatus == «Перерыв»)
```

```
            {
                var hasActiveOrder = await _dbContext.Orders
                    .AnyAsync(o => o.EmployeeId == id && (o.Status
== «Принят» || o.Status == «В сборке»));

                if (hasActiveOrder)
                    return new UpdateStatusResponse
                    {
                        Success = false,
                        Message = «Нельзя уйти на перерыв, пока
есть активный заказ»
                    };
            }

            employee.Status = newStatus;
            await _dbContext.SaveChangesAsync();

            return new UpdateStatusResponse
            {
                Success = true,
                Message = $»Статус сотрудника изменен на
'{newStatus}'«,
                EmployeeId = employee.Id,
                EmployeeName = employee.Name,
                Status = employee.Status
            };
        }

        public async Task<CompleteOrderResponse>
CompleteOrderAsync(int id)
        {
            var employee = await
_dbContext.Employees.FindAsync(id);
            if (employee == null)
                return new CompleteOrderResponse
                {
                    Success = false,
                    Message = «Сотрудник не найден»
                };

            var lastOrder = await _dbContext.Orders
                .Where(o => o.EmployeeId == id && o.Status ==
«Собран»)
                .OrderByDescending(o => o.Id)
                .FirstOrDefaultAsync();

            if (lastOrder != null)
            {

_loadBalancingService.HandleOrderCompletion(employee, lastOrder);
                await
_loadBalancingService.UpdateEmployeeQualificationAsync(employee);
```

```
                await _dbContext.SaveChangesAsync();
            }

            return new CompleteOrderResponse
            {
                Success = true,
                Message = «Заказ завершен, нагрузка сотрудника
обновлена»,
                EmployeeId = employee.Id,
                EmployeeName = employee.Name,
                Qualification = employee.Qualification,
                CompletedOrders = employee.CompletedOrders
            };
        }

        public async Task<EmployeeDashboardDto?>
GetEmployeeDashboardAsync(int id)
        {
            var employee = await _dbContext.Employees
                .Where(e => e.Id == id)
                .Select(e => new EmployeeDashboardDto
                {
                    Id = e.Id,
                    Name = e.Name,
                    Status = e.Status,
                    CurrentLoad = (decimal)e.CurrentLoad,
                    AssignedOrders = e.Orders.Count(o => o.Status
== «Назначен»),
                    InProgressOrders = e.Orders.Count(o =>
                        o.Status == «Принят» || o.Status == «В
сборке»),
                    CompletedOrders = e.Orders.Count(o => o.Status
== «Собран»),
                    HasNewOrders = e.Orders.Any(o => o.Status ==
«Назначен»),
                    HasActiveOrder = e.Orders.Any(o =>
                        o.Status == «Принят» || o.Status == «В
сборке»),
                    CanTakeBreak = e.Status != «Перерыв» &&
                                   !e.Orders.Any(o =>
                                       o.Status == «Принят» ||
o.Status == «В сборке»)
                })
                .FirstOrDefaultAsync();

            return employee;
        }
    }
}
```

Листинг Б.5 – Файл OrderService.cs

```
using Microsoft.EntityFrameworkCore;
using For_employee.Data;
using For_employee.Models;
using For_employee.DTO;
using For_employee.Modules.LoadBalancing;
using For_employee.Modules.Routing;

namespace For_employee.Modules.Orders
{
    public class OrderService
    {
        private readonly AppDbContext _db;
        private readonly LoadBalancingService _loadService;
        private readonly RoutingService2 _routingService;

        public OrderService(
            AppDbContext db,
            LoadBalancingService loadService,
            RoutingService2 routingService)
        {
            _db = db;
            _loadService = loadService;
            _routingService = routingService;
        }

        public async Task<List<object>> GetAllOrdersAsync()
        {
            return await _db.Orders
                .Include(o => o.Items).ThenInclude(i => i.Product)
                .Include(o => o.Employee)
                .OrderByDescending(o => o.Id)
                .Select(o => new
                {
                    o.Id,
                    o.Type,
                    o.Complexity,
                    o.Status,
                    o.EstimatedTime,
                    ItemsCount = o.Items.Count,
                    EmployeeName = o.Employee != null ?
o.Employee.Name : «Не назначен»
                })
                .ToListAsync<object>();
        }

        public async Task<object?> GetOrderByIdAsync(int id)
        {
            return await _db.Orders
                .Include(o => o.Items).ThenInclude(i => i.Product)
                .Include(o => o.Employee)
                .Where(o => o.Id == id)
```

```
                .Select(o => new
                {
                    o.Id,
                    o.Type,
                    o.Complexity,
                    o.Status,
                    o.EstimatedTime,
                    Items = o.Items.Select(i => new
                    {
                        i.Id,
                        ProductName = i.Product.Name,
                        Zone = i.Product.Zone,
                        i.Quantity
                    }),
                    Employee = o.Employee == null ? null : new
                    {
                        o.Employee.Id,
                        o.Employee.Name,
                        o.Employee.Qualification
                    }
                })
                .FirstOrDefaultAsync();
        }

        public async Task<object>
CreateOrderFullAsync(OrderRequest request)
        {
            if (request?.Items == null || request.Items.Count ==
0)
                throw new ArgumentException(«Заказ должен
содержать хотя бы один товар»);

            var items = request.Items.Select(i => new OrderItem
            {
                ProductId = i.ProductId,
                Quantity = i.Quantity
            }).ToList();

            var order = await CreateOrder(items);

            var employee = await _loadService.AssignOrder(order);

            var zones = order.Items.Select(i =>
i.Product.Zone).Distinct().ToList();
            var route = _routingService.BuildRoute(zones);

            return new
            {
                order.Id,
                order.Status,
                AssignedEmployee = employee?.Name,
                Route = route
```

```
            };
        }

        private async Task<Order> CreateOrder(List<OrderItem>
items)
        {
            var productIds = items.Select(i =>
i.ProductId).ToList();
            var products = await _db.Products
                .Where(p => productIds.Contains(p.Id))
                .ToDictionaryAsync(p => p.Id);

            var order = new Order
            {
                Items = items,
                Status = «Создан»
            };

            order.Type = DetermineType(order, products);

            order.Complexity = DetermineComplexity(order,
products);

            double pickingTimeMinutes = items.Sum(i =>
                products[i.ProductId].PickingTime * i.Quantity);

            var zones = items.Select(i =>
products[i.ProductId].Zone).Distinct().ToList();
            var route = _routingService.BuildRoute(zones);
            int totalDistanceMeters =
_routingService.CalculateTotalDistance(route);

            const double walkingSpeedMps = 0.5;  // 0.5 м/с = 30
м/мин
            double travelTimeMinutes = totalDistanceMeters /
walkingSpeedMps / 60.0;

            bool hasGate = zones.Any(z => z == «Шлюз1» ||
                (z == «E» || z == «F» || z == «G» || z == «N»));

            const double gateTimeSeconds = 30;
            double gateTimeMinutes = hasGate ? gateTimeSeconds /
60.0 : 0;

            double totalEstimatedMinutes = pickingTimeMinutes +
travelTimeMinutes + gateTimeMinutes;


            double complexityCoeff = order.Complexity switch
            {
                «Простой» => 1.0,
                «Средний» => 1.2,
```

```
                «Сложный» => 1.5,
                _ => 1.0
            };

            order.EstimatedTime = Math.Round(totalEstimatedMinutes
* complexityCoeff, 1);

            _db.Orders.Add(order);
            await _db.SaveChangesAsync();

            return order;
        }


        private string DetermineComplexity(Order order,
IReadOnlyDictionary<int, Product> products)
        {
            int itemCount = order.Items.Count;
            bool isUniform = DetermineType(order, products) ==
«Однотипный»;

            if (isUniform)
            {
                // Однотипные заказы
                if (itemCount <= 6)
                    return «Простой»;

                return «Средний»;  // однотипный, 7+ товаров
            }
            else
            {
                // Разнотипные заказы
                if (itemCount <= 3)
                    return «Простой»;
                if (itemCount <= 6)
                    return «Средний»;

                return «Сложный»;  // разнотипный, 7+ товаров
            }
        }

        private string DetermineType(Order order,
IReadOnlyDictionary<int, Product> products)
        {
            if (order.Items.Count == 0)
                return «Однотипный»;

            var categories = order.Items
                .Select(i => products[i.ProductId].CategoryId)
                .Distinct()
                .ToList();
```

```
            return categories.Count == 1 ? «Однотипный» :
«Разнотипный»;
        }

        // 📦 ПРИНЯТЬ ЗАКАЗ
        public async Task<(bool Success, string Message)>
AcceptOrderAsync(int orderId)
        {
            var order = await _db.Orders
                .Include(o => o.Employee)
                .FirstOrDefaultAsync(o => o.Id == orderId);

            if (order == null)
                return (false, «Заказ не найден»);

            if (order.Status != «Назначен»)
                return (false, «Неверный статус»);

            order.Status = «Принят»;

            if (order.Employee != null)
                order.Employee.Status = «Занят»;

            await _db.SaveChangesAsync();

            return (true, «Заказ принят»);
        }
public async Task<(bool Success, string Message)>
AcceptOrderAsync(int orderId)
        {
            var order = await _db.Orders
                .Include(o => o.Employee)
                .FirstOrDefaultAsync(o => o.Id == orderId);

            if (order == null)
                return (false, «Заказ не найден»);

            if (order.Status != «Назначен»)
                return (false, «Неверный статус»);

            order.Status = «Принят»;

            if (order.Employee != null)
                order.Employee.Status = «Занят»;

            await _db.SaveChangesAsync();

            return (true, «Заказ принят»);
        }
```

```
        public async Task<(bool Success, string Message)>
CompleteOrderAsync(int orderId)
        {
            var order = await _db.Orders
                .Include(o => o.Employee)
                .FirstOrDefaultAsync(o => o.Id == orderId);

            if (order == null)
                return (false, «Заказ не найден»);

            if (order.Status != «Принят»)
                return (false, «Нельзя завершить заказ в статусе «
+ order.Status);


            if (order.Employee != null)
            {
                var success = await
_loadService.CompleteOrder(order);
                if (!success)
                    return (false, «Ошибка при обновлении
нагрузки»);
}
```

Приложение В

(рекомендуемое)

Графическая часть

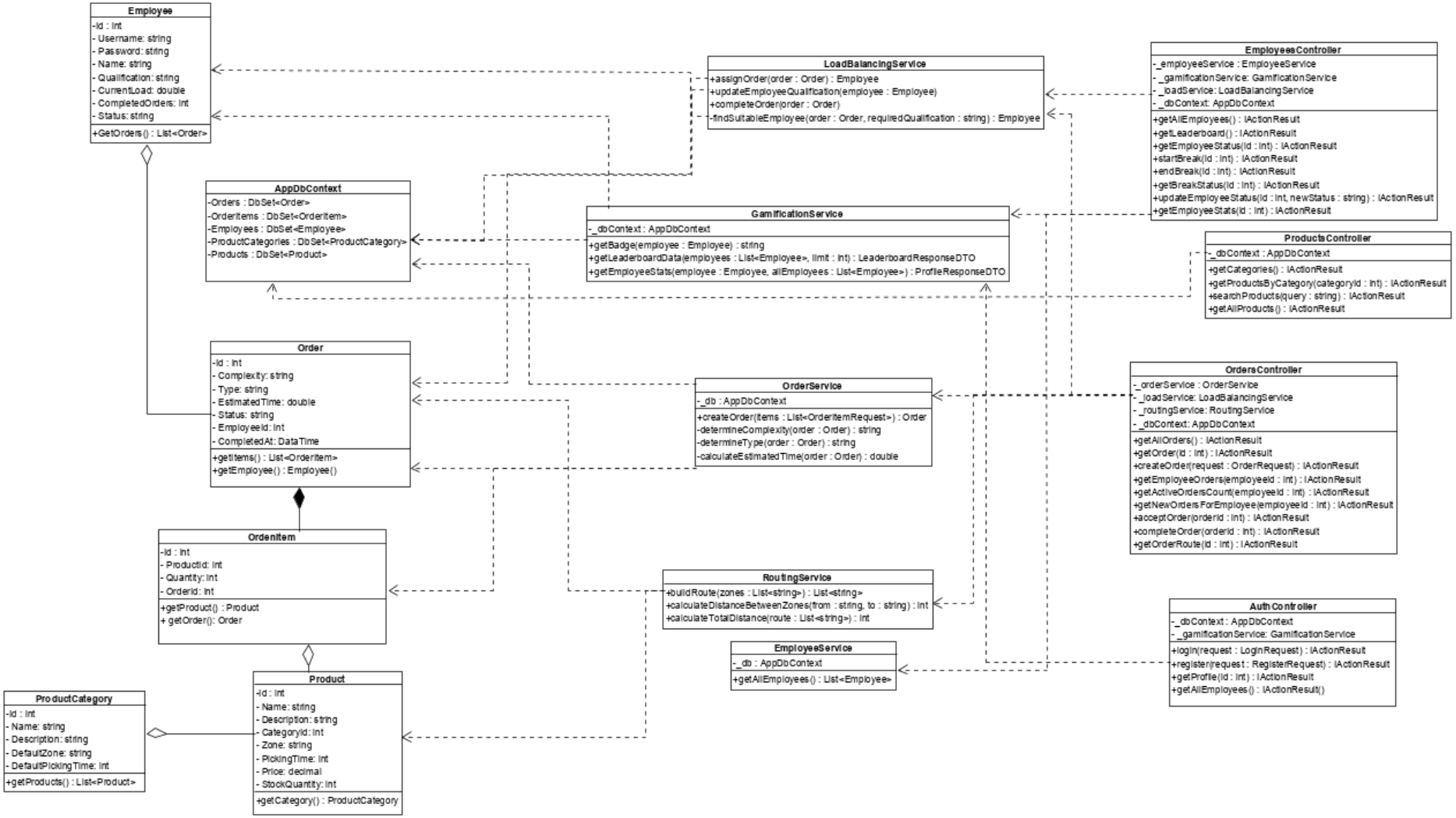

Рисунок В.1 – Диаграмма классов разрабатываемой системы

Приложение Г

(рекомендуемое)

Функциональное тестирование

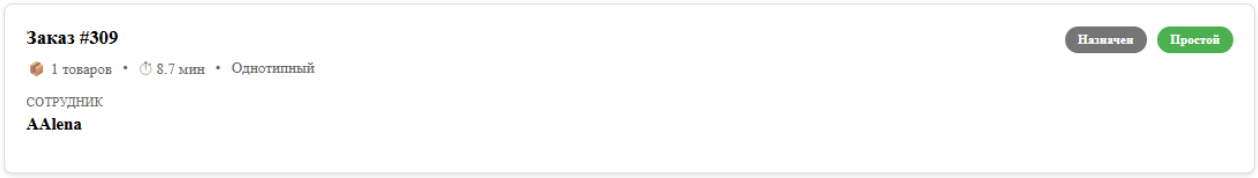

Рисунок Г.1 – Результат тестирования «Простой однотипный заказ»

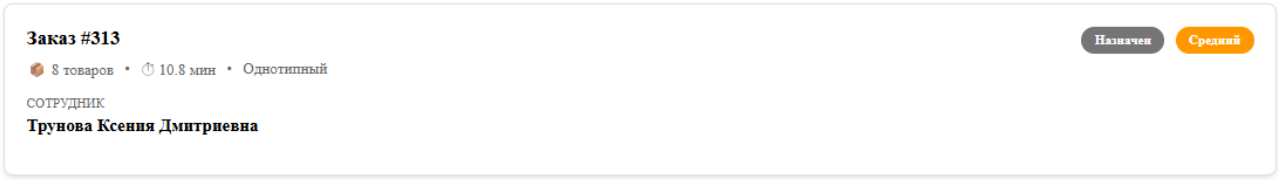

Рисунок Г.2 – Результат тестирования «Средний однотипный заказ»

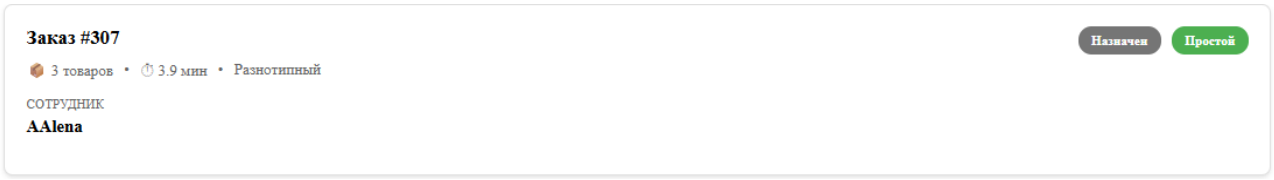

Рисунок Г.3 – Результат тестирования «Простой разнотипный заказ»

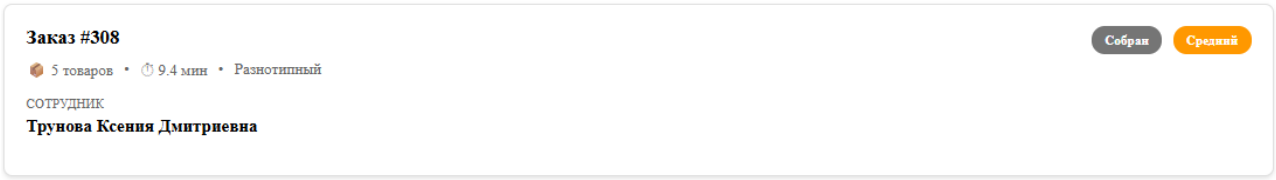

Рисунок Г.4 – Результат тестирования «Средний разнотипный заказ»

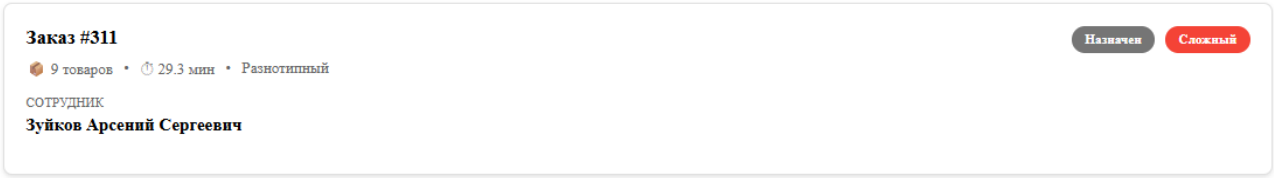

Рисунок Г.5 – Результат тестирования «Сложный заказ»

Приложение Д

(рекомендуемое)

Листинг модульных тестов

```
using For_employee.Data;
using For_employee.DTO;
using For_employee.Models;
using For_employee.Modules.Auth;
using For_employee.Modules.Employees;
using For_employee.Modules.LoadBalancing;
using For_employee.Modules.Orders;
using For_employee.Modules.Routing;
using Microsoft.EntityFrameworkCore;
using Xunit;

namespace For_employee.UnitTests;

public class AllTests : IDisposable
{
    private readonly AppDbContext _context;
    private readonly AuthService _authService;
    private readonly LoadBalancingService _loadBalancingService;
    private readonly RoutingService2 _routingService;
    private readonly OrderService _orderService;
    private readonly EmployeeService _employeeService;

    public AllTests()
    {
        var options = new DbContextOptionsBuilder<AppDbContext>()
            .UseInMemoryDatabase(databaseName:
Guid.NewGuid().ToString())
            .Options;
        _context = new AppDbContext(options);
        _authService = new AuthService(_context);
        _loadBalancingService = new
LoadBalancingService(_context);
        _routingService = new RoutingService2();
        _orderService = new OrderService(_context,
_loadBalancingService, _routingService);
    }

    public void Dispose()
    {
        _context.Database.EnsureDeleted();
        _context.Dispose();
    }

      [Fact]
    public async Task Register_Success_ReturnsEmployee()
    {
        var request = new RegisterRequest
        {
            Username = «testuser»,
            Password = «Test123456»,
            ConfirmPassword = «Test123456»,
            Name = «Тестовый Пользователь»
```

```
        };

        var result = await _authService.RegisterAsync(request);

        Assert.NotNull(result);
        Assert.Equal(«testuser», result.Username);
        Assert.Equal(«Новичок», result.Qualification);
    }

    [Fact]
    public async Task Register_PasswordsMismatch_ReturnsNull()
    {
        var request = new RegisterRequest
        {
            Username = «testuser»,
            Password = «Test123456»,
            ConfirmPassword = «WrongPassword»,
            Name = «Тестовый»
        };

        var result = await _authService.RegisterAsync(request);

        Assert.Null(result);
    }

    [Fact]
    public async Task Login_ValidCredentials_ReturnsEmployee()
    {
        var registerRequest = new RegisterRequest
        {
            Username = «logintest»,
            Password = «MySecretPass123»,
            ConfirmPassword = «MySecretPass123»,
            Name = «Login Test»
        };
        await _authService.RegisterAsync(registerRequest);

        var loginRequest = new LoginRequest
        {
            Username = «logintest»,
            Password = «MySecretPass123»
        };

        var result = await _authService.LoginAsync(loginRequest);

        Assert.NotNull(result);
        Assert.Equal(«logintest», result.Username);
    }

    [Fact]
    public async Task Login_InvalidPassword_ReturnsNull()
    {
```

```
        var registerRequest = new RegisterRequest
        {
            Username = «loginfail»,
            Password = «CorrectPassword123»,
            ConfirmPassword = «CorrectPassword123»,
            Name = «Login Fail»
        };
        await _authService.RegisterAsync(registerRequest);

        var loginRequest = new LoginRequest
        {
            Username = «loginfail»,
            Password = «WrongPassword»
        };

        var result = await _authService.LoginAsync(loginRequest);

        Assert.Null(result);
    }


    [Fact]
    public void Qualification_Novice_HasCorrectValue()
    {
        var employee = new Employee { Qualification = «Новичок» };
        Assert.Equal(«Новичок», employee.Qualification);
    }

    [Fact]
    public void Qualification_Experienced_HasCorrectValue()
    {
        var employee = new Employee { Qualification = «Опытный» };
        Assert.Equal(«Опытный», employee.Qualification);
    }

    [Fact]
    public void Qualification_Expert_HasCorrectValue()
    {
        var employee = new Employee { Qualification = «Эксперт» };
        Assert.Equal(«Эксперт», employee.Qualification);
    }


    [Fact]
    public async Task AssignOrder_NoEmployees_ReturnsNull()
    {
        var order = new Order { Complexity = «Простой» };
        var result = await
_loadBalancingService.AssignOrder(order);
        Assert.Null(result);
    }
```

```
    [Fact]
    public async Task AssignOrder_SimpleOrder_AssignsToNovice()
    {
        var novice = new Employee
        {
            Name = «Новичок»,
            Qualification = «Новичок»,
            Status = «Свободен»
        };
        var experienced = new Employee
        {
            Name = «Опытный»,
            Qualification = «Опытный»,
            Status = «Свободен»
        };

        _context.Employees.AddRange(novice, experienced);
        await _context.SaveChangesAsync();

        var order = new Order { Complexity = «Простой» };
        var result = await
_loadBalancingService.AssignOrder(order);

        Assert.NotNull(result);
        Assert.Equal(«Новичок», result?.Qualification);
    }

    [Fact]
    public async Task AssignOrder_ComplexOrder_AssignsToExpert()
    {
        var novice = new Employee { Name = «Новичок»,
Qualification = «Новичок» };
        var experienced = new Employee { Name = «Опытный»,
Qualification = «Опытный» };
        var expert = new Employee { Name = «Эксперт»,
Qualification = «Эксперт» };

        _context.Employees.AddRange(novice, experienced, expert);
        await _context.SaveChangesAsync();

        var order = new Order { Complexity = «Сложный» };
        var result = await
_loadBalancingService.AssignOrder(order);

        Assert.NotNull(result);
        Assert.Equal(«Эксперт», result?.Qualification);
    }

    [Fact]
    public async Task
UpdateQualification_15Orders_BecomesExperienced()
    {
```

```
        var employee = new Employee
        {
            Name = «Тест»,
            Qualification = «Новичок»,
            CompletedOrders = 15
        };

        _context.Employees.Add(employee);
        await _context.SaveChangesAsync();

        await
_loadBalancingService.UpdateEmployeeQualificationAsync(employee);

        Assert.Equal(«Опытный», employee.Qualification);
    }

    [Fact]
    public async Task UpdateQualification_35Orders_BecomesExpert()
    {
        var employee = new Employee
        {
            Name = «Тест»,
            Qualification = «Опытный»,
            CompletedOrders = 35
        };

        _context.Employees.Add(employee);
        await _context.SaveChangesAsync();

        await
_loadBalancingService.UpdateEmployeeQualificationAsync(employee);

        Assert.Equal(«Эксперт», employee.Qualification);
    }

    [Fact]
    public void BuildRoute_EmptyZones_ReturnsEntryAndExit()
    {
        var result = _routingService.BuildRoute(new
List<string>());
        Assert.Contains(«Entry», result);
        Assert.Contains(«Exit», result);
    }

    [Fact]
    public void BuildRoute_SingleZone_ContainsZone()
    {
        var result = _routingService.BuildRoute(new List<string> {
«A1» });
        Assert.Contains(«A1», result);
    }
```

```
    [Fact]
    public void CalculateDistance_SameZone_ReturnsZero()
    {
        var result =
_routingService.CalculateDistanceBetweenZones(«A1», «A1»);
        Assert.Equal(0, result);
    }


    [Fact]
    public async Task GetAllEmployees_ReturnsAll()
    {
        _context.Employees.AddRange(
            new Employee { Username = «user1», Password = «hash1»,
Name = «User1» },
            new Employee { Username = «user2», Password = «hash2»,
Name = «User2» }
        );
        await _context.SaveChangesAsync();

        var result = await
_employeeService.GetAllEmployeesAsync();

        Assert.Equal(2, result.Count);
    }

    [Fact]
    public async Task GetEmployeeStatus_Existing_ReturnsStatus()
    {
        var employee = new Employee { Username = «test», Password
= «hash», Status = «Свободен» };
        _context.Employees.Add(employee);
        await _context.SaveChangesAsync();

        var result = await
_employeeService.GetEmployeeStatusAsync(employee.Id);

        Assert.NotNull(result);
        Assert.Equal(«Свободен», result.Status);
    }

    [Fact]
    public async Task GetEmployeeStatus_NonExisting_ReturnsNull()
    {
        var result = await
_employeeService.GetEmployeeStatusAsync(999);
        Assert.Null(result);
    }
}
```