

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

**Программные средства обеспечения механизма
единого управления конфигурацией
информационной системы электронного
документооборота**

Студент

Каледа Роман Александрович

(подпись, дата)

(ФИО полностью)

Руководитель

Иванчуков А.Г.

(подпись, дата)

(фамилия, инициалы)

Нормоконтролёр

Такташкин Д.В.

(подпись, дата)

(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от 03.06.2026г. № 12)

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
Вычислительной техники

Кафедра
МО и ПЭВМ

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

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1. Студент Каледа Роман Александрович гр. 22ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Программные средства обеспечения механизма единого управления конфигурацией информационной системы электронного документооборота

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

3. Руководитель работы к.т.н., доцент кафедры «МОиПЭВМ» А. Г. Иванчуков

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

1) *Разработать программные средства, обеспечивающие единый расширяемый механизм администрирования конфигурации функционально разнородных служб информационной системы электронного документооборота*

2) *Среда выполнения приложения: ОС Linux*
Языковые средства разработки: C#, SQLite, UML
Инструментальные средства разработки: JetBrains Rider, Project Libre, Visual Studio 2022, Visual Paradigm, DB Browser for SQLite
Технологии разработки: ООП, Entity Framework Core, Apache Thrift, ASP .NET Core

3) *Основные функции программных средств:*
а) единое управление настройками функционально разнородных служб;
б) расширение состава поддерживаемых служб подключаемыми модулями без модификации основных компонентов;
в) локальное и удалённое администрирование настроек;
г) резервное копирование и восстановление конфигурации с версионированием;
д) согласованное применение настроек с оповещением зависимых служб.

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 114 листов, 26 рисунков, 18 таблиц, 26 использованных источников, 4 приложения.

УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ, МИКРОЯДЕРНАЯ
АРХИТЕКТУРА, ПЛАГИННАЯ АРХИТЕКТУРА, ЭЛЕКТРОННЫЙ
ДОКУМЕНТООБОРОТ, РАСШИРЯЕМОСТЬ, РЕЗЕРВНОЕ КОПИРОВАНИЕ,
C#, .NET, ASP.NET CORE, APACHE THRIFT, ENTITY FRAMEWORK CORE,
SQLITE

Объектом разработки являются программные средства единого управления конфигурацией функционально разнородных служб информационной системы электронного документооборота.

Цель работы – разработка программных средств, обеспечивающих единый расширяемый механизм администрирования конфигурации функционально разнородных служб информационной системы электронного документооборота, единообразный для всех служб и допускающий добавление поддержки новых служб без модификации основных компонентов.

Разработанные программные средства построены на основе микроядерной архитектуры: каждая управляемая служба представлена независимым подключаемым модулем – плагином, реализующим общий интерфейс. Реализованы централизованное хранение конфигурации, локальный и удалённый каналы администрирования, атомарное применение изменений с откатом, событийное оповещение зависимых служб, резервное копирование и восстановление конфигурации с версионированием формата, а также устойчивое фоновое применение настроек.

					ПГУ 09.03.04 – 08БР222.12 ПЗ			
Изм.	Лист	№ докум.	Подп.	Дата	Программные средства обеспечения механизма единого управления конфигурацией информационной системы электронного документооборота	Лит.	Лист	Листов
Разраб.		Каледа Р.А.						
Пров.		Иванчуков А.Г.					4	114
Н. контр.		Такташкин Д.В.				Группа 22ВП2		
Утверд.		Козлов А.Ю.						
					Пояснительная записка			

Содержание

Введение.....	7
1 Анализ предметной области и существующих решений	9
1.1 Анализ предметной области и постановка задачи.....	9
1.2 Анализ существующих решений.....	10
1.2.1 Ansible	11
1.2.2 Spring Cloud Config.....	12
1.2.3 HashiCorp Consul.....	14
1.3 Сравнительный анализ программных продуктов	16
1.4 Планирование разработки и расчет бюджета.....	18
2 Анализ требований к разрабатываемым средствам.....	21
2.1 Анализ функциональных требований	21
2.2 Анализ нефункциональных требований.....	25
3 Проектирование системы	28
3.1 Архитектура системы	28
3.2 Проектирование ядра системы	34
3.3 Проектирование общей инфраструктурной сборки	36
3.4 Проектирование плагинной подсистемы.....	37
3.4.1 Типовая структура плагина.....	38
3.4.2 Проектирование плагина управления службой защиты данных	40
3.4.3 Проектирование плагина управления службой формирования штампов времени	44
3.4.4 Проектирование плагина управления настройками обработки электронных документов.....	47
3.4.5 Проектирование плагина управления подключениями к базе данных	50
3.4.6 Проектирование плагина управления службой повторной обработки информации	52
3.4.7 Особенности остальных плагинов	54
3.5 Проектирование механизма межплагинного взаимодействия	55
3.6 Проектирование подсистемы удалённого администрирования	56
4 Реализация системы	59
4.1 Динамическая загрузка плагинов	61

4.2 Резервное копирование и версионирование формата конфигурации	63
4.3 Иерархические блокировки	68
4.4 Защита фонового применения настроек от устойчивых сбоев	71
5 Тестирование программного продукта	74
5.1 Функциональное тестирование методом белого ящика.....	74
5.1.1 Покрытие операторов	75
5.1.2 Покрытие решений	78
5.1.3 Покрытие условий	79
5.1.4 Покрытие решений и условий	80
5.2 Модульное тестирование.....	82
5.3 Интеграционное тестирование	83
5.4 Оценка качества кода	85
5.4.1 Метрики Холстеда.....	85
5.4.2 Метрика Чепина	89
Заключение.....	91
Список использованных источников.....	94
Приложение А (справочное) Глоссарий	96
Приложение Б (обязательное) Исходный код приложения.....	98
Приложение В (рекомендуемое) Результаты тестирования.....	111
Приложение Г (рекомендуемое) Акт о внедрении.....	113

Введение

Современные информационные системы, как правило, представляют собой совокупность функционально разнородных сервисов, каждый из которых требует регулярной настройки в процессе эксплуатации [1]. Состав настраиваемых параметров у каждого сервиса свой и определяется его назначением, а сами настройки многократно изменяются в течение срока службы системы. По мере развития системы набор её сервисов расширяется, что увеличивает и объём подлежащих администрированию настроек.

Существенной особенностью таких систем является взаимная зависимость сервисов: настройки разных сервисов связаны между собой, а изменение настроек одного сервиса должно своевременно становиться известным остальным заинтересованным сервисам. Поэтому от средств настройки требуется не только изменение отдельных параметров, но и поддержание всей конфигурации системы в согласованном и целостном состоянии.

Распространённая практика, при которой настройки каждого сервиса задаются обособленно – через индивидуальные конфигурационные файлы либо через несвязанные между собой средства администрирования, – в этих условиях приводит к ряду эксплуатационных проблем: отсутствию единой точки управления, разрозненности интерфейсов настройки, повышенной трудоёмкости сопровождения и росту вероятности ошибок при ручном вмешательстве. При взаимозависимости сервисов обособленное хранение настроек особенно затрудняет согласованное внесение изменений и поддержание целостности конфигурации в целом. С увеличением числа сервисов перечисленные проблемы становятся всё более ощутимыми, что определяет потребность в едином механизме управления конфигурацией.

Дополнительную особенность задаче придаёт то, что базовая информационная система уже находится в эксплуатации, а её функциональность необходимо наращивать, не вмешиваясь в основные компоненты. Это предъявляет к разрабатываемым программным средствам повышенные

требования к расширяемости и единообразию: они должны допускать быстрое и удобное расширение состава поддерживаемых сервисов без изменения существующего кода и обеспечивать управление настройками всех сервисов через единые средства администрирования.

Актуальность настоящей выпускной квалификационной работы определяется тем, что с ростом числа взаимозависимых сервисов обособленное управление их настройками становится всё более трудоёмким и подверженным ошибкам, а поддержание конфигурации в согласованном состоянии – всё более сложным. Это обуславливает потребность в централизованном механизме управления конфигурацией сервисов информационной системы.

Целью выпускной квалификационной работы является разработка программных средств, обеспечивающих единый механизм администрирования конфигурации функционально разнородных сервисов информационной системы электронного документооборота, единообразный для всех сервисов и расширяемый без модификации основных компонентов системы.

Для достижения поставленной цели в рамках работы решаются следующие задачи:

- исследование предметной области и существующих средств управления конфигурацией приложений;
- формирование функциональных и нефункциональных требований к разрабатываемым программным средствам;
- проектирование архитектуры разрабатываемых программных средств, обеспечивающей расширяемость без модификации существующих компонентов;
- реализация спроектированных программных средств;
- тестирование разработанных программных средств.

Практическая значимость работы состоит в том, что разрабатываемые программные средства позволяют добавлять поддержку новых сервисов в уже эксплуатируемую систему без её доработки, сокращая трудоёмкость сопровождения и снижая риск ошибок при настройке взаимозависимых сервисов.

1 Анализ предметной области и существующих решений

1.1 Анализ предметной области и постановка задачи

Разрабатываемые программные средства предназначены для управления конфигурацией информационной системы, которая представляет собой совокупность функционально разнородных сервисов, совместно решающих общую прикладную задачу. Каждый сервис выполняет собственную функцию и в процессе эксплуатации требует настройки – задания параметров, определяющих его поведение.

Под конфигурацией сервиса понимается совокупность параметров и настроек, определяющая поведение сервиса в процессе эксплуатации. Перечень параметров каждого сервиса различен и определяется его функциональным назначением: он может охватывать адреса взаимодействующих служб, режимы и условия обработки данных, а также параметры, разделяемые с другими сервисами.

Характерной особенностью рассматриваемой системы является взаимная зависимость сервисов: настройки разных сервисов связаны между собой, поэтому изменение настроек одного сервиса должно становиться известным остальным заинтересованным сервисам, а конфигурация системы в целом должна оставаться согласованной и целостной.

В сложившейся практике настройки каждого сервиса задаются обособленно – через индивидуальные конфигурационные файлы либо через несвязанные между собой средства администрирования. Такой подход приводит к отсутствию единой точки управления, разрозненности интерфейсов настройки, повышенной трудоёмкости сопровождения и росту вероятности ошибок при ручном вмешательстве. При взаимной зависимости сервисов обособленное хранение настроек дополнительно затрудняет согласованное внесение изменений и поддержание целостности конфигурации, в том числе при создании и восстановлении её резервных копий. Дополнительным условием является то,

что базовая система уже находится в эксплуатации, поэтому наращивание её функциональности не должно затрагивать основные компоненты.

Для устранения перечисленных проблем требуется разработать программные средства, обеспечивающие централизованное управление конфигурацией сервисов системы. Разрабатываемые программные средства должны обеспечивать:

- единый механизм управления конфигурацией всех функциональных сервисов системы;
- расширяемость за счёт возможности добавления поддержки новых сервисов, а также обновления и исключения поддержки существующих без модификации остальных компонентов;
- централизованное хранение конфигурации;
- локальный и удалённый каналы администрирования;
- централизованное создание и восстановление резервных копий конфигурации.

Таким образом, ставится задача разработки программных средств, реализующих перечисленные возможности на основе единого расширяемого механизма управления конфигурацией.

1.2 Анализ существующих решений

В настоящее время существует ряд программных продуктов, предназначенных для управления конфигурацией приложений и инфраструктуры [2].

В рамках работы рассмотрены три инструмента, представляющих различные подходы к решению этой задачи: Ansible, Spring Cloud Config и HashiCorp Consul.

Для каждого из них приведены общая характеристика, принцип работы, преимущества и оценка применимости к задаче управления конфигурацией функционально разнородных сервисов.

1.2.1 Ansible

Ansible – это инструмент автоматизации информационной инфраструктуры с открытым исходным кодом, разрабатываемый компанией Red Hat. Инструмент предназначен для автоматизированной настройки операционных систем, развертывания приложений, выполнения административных задач на удаленных узлах и поддержания заданного состояния инфраструктуры.

В основе работы Ansible лежит декларативное описание состояния системы. Администратор формирует так называемые плейбуки – текстовые файлы в формате YAML, в которых перечислены задачи, подлежащие выполнению на управляемых узлах. Применение плейбука осуществляется по push-модели: управляющий узел инициирует подключение к каждому управляемому узлу, как правило, по протоколу SSH, передает необходимые модули и выполняет описанные задачи. Установка дополнительного программного обеспечения на управляемые узлы при этом не требуется. Для группировки управляемых узлов и описания их параметров используется инвентарь – отдельный файл, содержащий перечень узлов и их атрибутов.

В пользу применения Ansible говорит широкая распространенность инструмента, развитое сообщество и обширный набор готовых модулей, позволяющий применять его в разнородных средах. Декларативный формат описания конфигурации читаем и легко поддерживается. Push-модель не требует установки дополнительного агента на управляемые узлы, что упрощает развертывание.

Применительно к задаче управления конфигурацией функционально разнородных сервисов Ansible имеет ряд существенных ограничений. Инструмент рассчитан в первую очередь на администрирование операционных систем и сопутствующей инфраструктуры – управление файлами, пакетами, службами уровня операционной системы. Управление состоянием отдельных компонентов внутри работающего приложения средствами Ansible требует

использования специализированных модулей и собственной разработки сценариев под каждый компонент, что повышает трудоемкость сопровождения и противоречит требованию единообразного механизма управления.

Применение Ansible предполагает наличие у управляющего узла прямого сетевого доступа к управляемым по протоколу SSH с правами на запись в файловую систему и выполнение команд от имени привилегированного пользователя. В условиях, когда управляемая подсистема представляет собой набор процессов в составе приложения, такая модель доступа избыточна и нарушает принцип ограничения прав, принятый в системах с повышенными требованиями к безопасности.

Применение изменений в плейбуках Ansible, как правило, требует перезапуска целевых процессов после обновления конфигурационных файлов. Для подсистем, которые должны сохранять непрерывность работы при изменении настроек, такая модель неприменима. Кроме того, Ansible не предусматривает встроенных средств определения различия между состоянием конфигурации, описанным в источнике, и состоянием, фактически примененным к работающему компоненту – задача обнаружения и устранения подобного расхождения остается на стороне разработчика сценариев.

Таким образом, Ansible эффективен в задачах автоматизации инфраструктуры и системного администрирования, однако не обеспечивает требуемого уровня единообразия и удобства управления конфигурацией функциональных модулей в составе одной информационной системы.

1.2.2 Spring Cloud Config

Spring Cloud Config – это компонент экосистемы Spring Cloud, разрабатываемой компанией VMware. Компонент предназначен для централизованного управления конфигурацией приложений в распределенных программных системах, построенных преимущественно на базе платформы Java.

Архитектурно Spring Cloud Config построен по схеме «сервер конфигурации – клиенты». Сервер конфигурации представляет собой отдельное

приложение, обеспечивающее хранение конфигурационных файлов и предоставляющее REST-интерфейс для их получения. В качестве источника конфигураций используются внешние хранилища – как правило, репозитории системы управления версиями, файловая система или защищенные хранилища секретов. Клиентские приложения при запуске обращаются к серверу по pull-модели и получают актуальные значения параметров. Поддерживается группировка конфигураций по профилям и пометкам, что позволяет хранить параметры для разных сред эксплуатации в одном источнике.

Преимуществами Spring Cloud Config являются глубокая интеграция с экосистемой Spring и связанными механизмами, поддержка версионирования конфигурации за счет использования систем управления версиями, наличие средств шифрования значений, а также возможность динамического обновления параметров в работающем приложении без рестарта при использовании дополнительных компонентов.

Применительно к задаче управления конфигурацией функционально разнородных сервисов Spring Cloud Config имеет ряд ограничений. Инструмент изначально проектировался под платформу Java и микросервисы, построенные на базе Spring. Использование клиентской части в приложениях, реализованных на других технологических стеках, требует разработки альтернативных клиентов и не обеспечивается «из коробки», что снижает применимость инструмента в системах, не построенных на платформе Spring.

Конфигурация в Spring Cloud Config представляется в виде набора пар «ключ – значение» с поддержкой иерархической организации, однако без какой-либо предметной типизации. Контроль допустимости значений конкретных параметров выполняется на стороне приложения-потребителя, что затрудняет согласованную проверку целостности настроек на уровне системы управления конфигурацией. Для задач, в которых параметры разнородных функциональных модулей требуют семантически осмысленных операций – добавления элементов в коллекции, проверки доступности удаленных сервисов, упорядочивания записей по приоритету – указанная модель оказывается недостаточной.

Развертывание Spring Cloud Config предполагает наличие отдельного экземпляра сервера конфигурации в составе инфраструктуры, а также внешнего хранилища, выступающего источником данных. Для задач управления конфигурацией единого приложения с фиксированным составом подсистем такая модель развертывания избыточна.

Кроме того, Spring Cloud Config реализует только pull-модель доставки конфигураций – изменение, внесенное на стороне сервера конфигурации, применяется на стороне клиента при следующем запросе. Возможность инициировать применение настроек со стороны административного интерфейса в режиме реального времени напрямую не поддерживается и требует использования дополнительных средств взаимодействия.

Таким образом, Spring Cloud Config эффективен в задачах централизованного управления конфигурацией микросервисов на платформе Java и не обеспечивает требуемого уровня универсальности, типизации и оперативности управления для задачи единого администрирования функциональных модулей информационной системы.

1.2.3 HashiCorp Consul

В отличие от ранее рассмотренных инструментов, HashiCorp Consul – это не средство управления конфигурацией приложений, а платформа управления сервисами в крупных распределенных системах [3]. Тем не менее, в ее состав входит распределенное хранилище типа «ключ – значение», которое часто используется как централизованный источник настроек в системах разного назначения. Это делает уместным рассмотрение Consul как примера соответствующего класса решений.

Работа Consul строится вокруг кластера серверных узлов, поддерживающих общее согласованное состояние. На стороне приложений ставится агент Consul, который общается с кластером. Конфигурация хранится в иерархии пар «ключ – значение» с поддержкой версионирования, областей видимости и контроля доступа. Получать и обновлять значения можно через

программный интерфейс или утилиту командной строки. Поддерживается механизм слежения за изменениями, чтобы клиентские приложения получали уведомления об обновлениях.

К достоинствам Consul относятся отказоустойчивая архитектура, сохраняющая данные при отказе отдельных узлов, развитая система контроля доступа, шифрование передаваемых данных, а также интеграция с популярными платформами развертывания.

Применительно к задаче управления конфигурацией функционально разнородных сервисов у Consul, как и у других инструментов того же класса, есть ограничения. Платформа рассчитана на работу в распределенной среде и предполагает развертывание отдельного кластера серверных узлов с поддержанием их согласованности. Для управления настройками одного приложения, которое работает в единой среде исполнения, такая инфраструктура избыточна.

Хранилище Consul использует обычную модель «ключ – значение», где значение – это произвольный набор байтов без какой-либо структуры. Никакой встроенной проверки корректности значений на уровне платформы нет, все проверки приходится выполнять на стороне приложений. Связанные операции – например, добавление элемента в упорядоченную коллекцию, изменение приоритета записи или проверка доступности удаленного сервера – на уровне самой платформы не поддерживаются и требуют отдельной реализации в каждом приложении.

Кроме того, Consul только хранит значения и сообщает об их изменениях, но не применяет их к управляемым сервисам. Логiku обработки полученных значений и контроль того, что они действительно применены, приходится писать вручную. Для задачи единого управления настройками функциональных модулей с разной семантикой такого подхода недостаточно.

Помимо этого, Consul не позволяет расширять управляемую систему за счет подключения дополнительных функциональных модулей, поскольку платформа является внешним сервисом по отношению к приложениям. Чтобы

добавить новый функциональный модуль в систему, использующую Consul, все равно придется писать сопутствующий клиентский код вручную.

Таким образом, Consul и другие инструменты класса распределенных KV-хранилищ хорошо подходят для централизованного хранения настроек в крупных распределенных системах, но для задачи единообразного управления разнородными функциональными модулями в составе одной информационной системы их возможностей недостаточно.

1.3 Сравнительный анализ программных продуктов

На основе проведенного обзора был выполнен сравнительный анализ рассмотренных программных продуктов применительно к задаче управления конфигурацией функциональных модулей информационной системы. В качестве критериев сравнения выбраны характеристики, отражающие специфику решаемой задачи. Поскольку каждый из рассмотренных инструментов решает свой класс задач, при сравнении оценивалось не качество инструмента в целом, а его применимость к задаче управления конфигурацией разнородных функциональных модулей в составе единой информационной системы. Результаты сравнения приведены в таблице 1.

Таблица 1 – Сравнительный анализ программных продуктов

Критерий	Ansible	Spring Cloud Config	HashiCorp Consul	Разрабатываемая система
Расширяемость управляемой системы	Не предусмотрена	Не предусмотрена	Не предусмотрена	Через плагины
Применение настроек	Внешнее, по push-модели	Внешнее, по pull-модели	Не выполняется	Внутреннее, по двум каналам
Обнаружение расхождений с фактическим состоянием	На стороне сценариев	На стороне приложения	Не предусмотрено	Встроено
Требование развертывания дополнительной инфраструктуры	Требуется	Требуется	Требуется	Не требуется

Продолжение таблицы 1

Критерий	Ansible	Spring Cloud Config	HashiCorp Consul	Разрабатываемая система
Совместное использование настроек между модулями	Не предусмотрено	Не предусмотрено	Не предусмотрено	Через сборки-медиаторы
Версионирование формата конфигурации с поддержкой миграции	Не предусмотрено	Не предусмотрено	Не предусмотрено	Встроено
Контроль над кодовой базой	Внешний	Внешний	Внешний	Полный

Из приведённой таблицы следует, что ни один из рассмотренных инструментов не покрывает всего набора характеристик, необходимых для решения поставленной задачи.

При выборе подхода к реализации учитывались также происхождение и доступность кодовой базы рассмотренных инструментов. Все они опираются на стороннюю кодовую базу, лицензирование и доступность которой находятся вне контроля разработчика; более того, один из них в последние годы был переведён с открытой лицензии на лицензию, ограничивающую использование при сохранении доступа к исходному коду, и тем самым утратил статус открытого ПО, а условия его использования стали определяться единственным правообладателем.

Сторонние инструменты в качестве основы не рассматривались в том числе по этой причине, хотя при проектировании технически имелась возможность взять за основу, например, Ansible.

На основании результатов сравнительного анализа сделан вывод о необходимости разработки собственного программного решения, независимого от внешней кодовой базы, что особенно важно для системы, эксплуатируемой в составе значимой инфраструктуры с повышенными требованиями к безопасности.

1.4 Планирование разработки и расчет бюджета

Для оценки трудоёмкости и стоимости разработки программных средств выполнено планирование основных этапов работы и расчёт денежных затрат на их выполнение. Средняя предлагаемая заработная плата для работника сферы информационных технологий в Пензенской области по состоянию на 2026 год составляет 70192 рубля в месяц [4]. Соответствующий график динамики заработной платы приведён на рисунке 1.

Динамика ожидаемых и предлагаемых зарплат

Информационные технологии

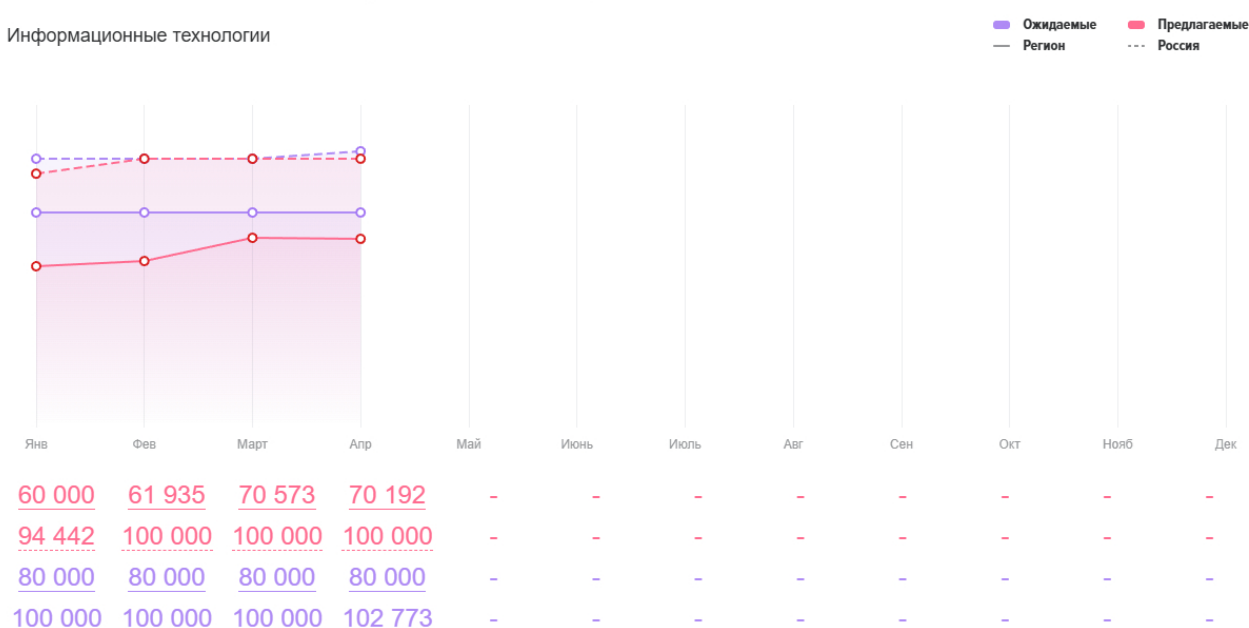


Рисунок 1 – Динамика заработной платы для работников сферы информационных технологий в Пензенской области в 2026 году

Исходя из графика, при 40-часовой рабочей неделе средняя стоимость одного часа работы специалиста составляет приблизительно 439 рублей.

Разработка программных средств единого управления конфигурацией информационной системы электронного документооборота разбита на пять основных этапов. Часть этапов детализирована на отдельные подэтапы. Перечень этапов и подэтапов с указанием их длительности и календарной привязки приведён на рисунке 2.

Название	Продолжи...	Начало	Окончание
Анализ требований	8 дней	02.03.2026, 08:00	11.03.2026, 17:00
Анализ предметной области и постановка задачи	2 дней	02.03.2026, 08:00	03.03.2026, 17:00
Анализ существующих программных продуктов	3 дней	04.03.2026, 08:00	06.03.2026, 17:00
Анализ функциональных требований	2 дней	09.03.2026, 08:00	10.03.2026, 17:00
Анализ нефункциональных требований	1 день	11.03.2026, 08:00	11.03.2026, 17:00
Проектирование системы	12 дней	12.03.2026, 08:00	27.03.2026, 17:00
Проектирование архитектуры системы	3 дней	12.03.2026, 08:00	16.03.2026, 17:00
Проектирования ядра системы	2 дней	17.03.2026, 08:00	18.03.2026, 17:00
Проектирование плагиновой подсистемы	5 дней	19.03.2026, 08:00	25.03.2026, 17:00
Проектирование подсистемы удалённого администрирования	2 дней	26.03.2026, 08:00	27.03.2026, 17:00
Реализация программных средств	19 дней	30.03.2026, 08:00	23.04.2026, 17:00
Реализация ядра системы	4 дней	30.03.2026, 08:00	02.04.2026, 17:00
Реализация плагиновой подсистемы	10 дней	03.04.2026, 08:00	16.04.2026, 17:00
Реализация подсистемы удалённого администрирования	5 дней	17.04.2026, 08:00	23.04.2026, 17:00
Тестирование	10 дней	24.04.2026, 08:00	07.05.2026, 17:00
Составление и оформление документации	7 дней	08.05.2026, 08:00	18.05.2026, 17:00

Рисунок 2 – План разработки программных средств

Для наглядного представления распределения этапов разработки во времени построена диаграмма Ганта, представленная на рисунке 3. Диаграмма Ганта используется как инструмент календарного планирования и позволяет визуализировать последовательность и продолжительность работ при выполнении проекта.

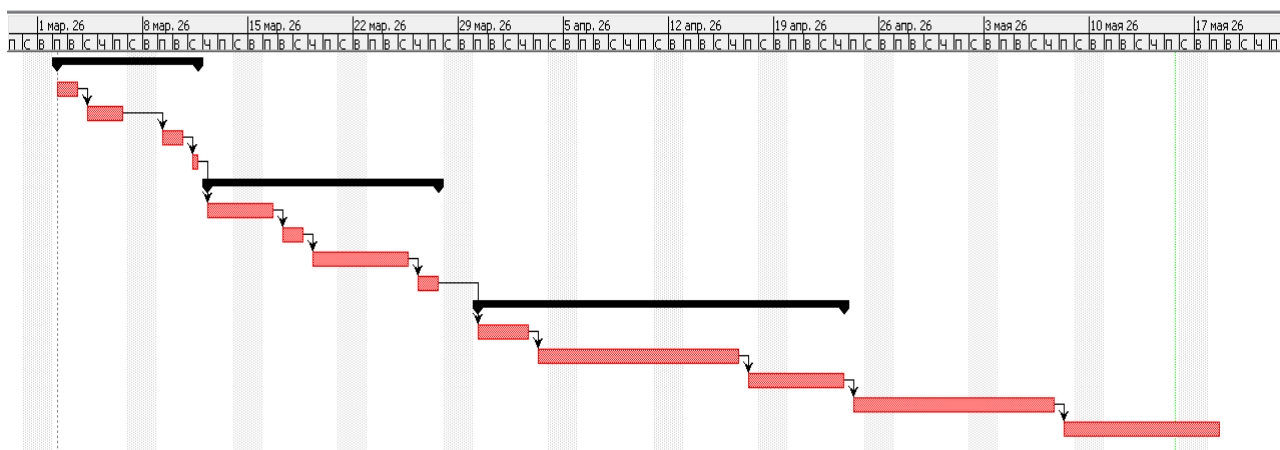


Рисунок 3 – Диаграмма Ганта разработки программных средств

На основе оценки трудоёмкости каждого этапа с применением рассчитанной часовой ставки получена итоговая оценка денежных затрат на разработку. Результаты расчёта приведены в таблице 2.

Таблица 2 – Оценка трудозатрат и стоимости разработки

Этап работы	Трудоёмкость	Часовая ставка	Стоимость
Анализ предметной области и постановка задачи	16 ч	439 руб/ч	7 024 руб
Анализ существующих программных продуктов	24 ч	439 руб/ч	10 536 руб
Анализ функциональных требований	16 ч	439 руб/ч	7 024 руб
Анализ нефункциональных требований	8 ч	439 руб/ч	3 512 руб
Проектирование архитектуры системы	24 ч	439 руб/ч	10 536 руб
Проектирование ядра системы	16 ч	439 руб/ч	7 024 руб
Проектирование плагинной подсистемы	40 ч	439 руб/ч	17 560 руб
Проектирование подсистемы удалённого администрирования	16 ч	439 руб/ч	7 024 руб
Реализация ядра системы	32 ч	439 руб/ч	14 048 руб
Реализация плагинной подсистемы	80 ч	439 руб/ч	35 120 руб
Реализация подсистемы удалённого администрирования	40 ч	439 руб/ч	17 560 руб
Тестирование	80 ч	439 руб/ч	35 120 руб
Составление и оформление документации	56 ч	439 руб/ч	24 584 руб

Таким образом, суммарная трудоёмкость разработки программных средств единого управления конфигурацией информационной системы электронного документооборота составляет 448 часов, а ориентировочная стоимость их разработки – 196 672 рубля.

2 Анализ требований к разрабатываемым средствам

2.1 Анализ функциональных требований

Разрабатываемые программные средства предназначены для централизованного управления настройками набора функционально разнородных служб, образующих информационную систему.

Управление осуществляется администратором системы через локальный или удалённый программный интерфейс. Помимо администратора, с системой взаимодействует внешняя подсистема резервного копирования, обращающаяся к ней для получения и обновления конфигурации в составе общесистемных операций.

К общим функциональным требованиям отнесены требования, описывающие операции системы, не специфичные для отдельной службы:

- предоставление двух каналов администрирования – локального и удалённого, с идентичным набором операций;
- подключение, обновление и исключение поддержки отдельных служб без модификации остальных компонентов системы;
- предоставление интерфейса подсистеме резервного копирования с возможностью получения и обновления конфигурации системы;
- поддержка нескольких версий формата конфигурации с автоматической миграцией данных предшествующих версий к текущему формату при их применении.

Информационная система состоит из набора функционально разнородных служб, каждая из которых обладает собственным составом настроек, определяющим её поведение при эксплуатации. Администратор управляет этими настройками через разрабатываемые программные средства, выполняя для каждой службы операции просмотра, изменения и сброса настроек, а также проверки их состояния. Варианты использования системы приведены на рисунке 4.

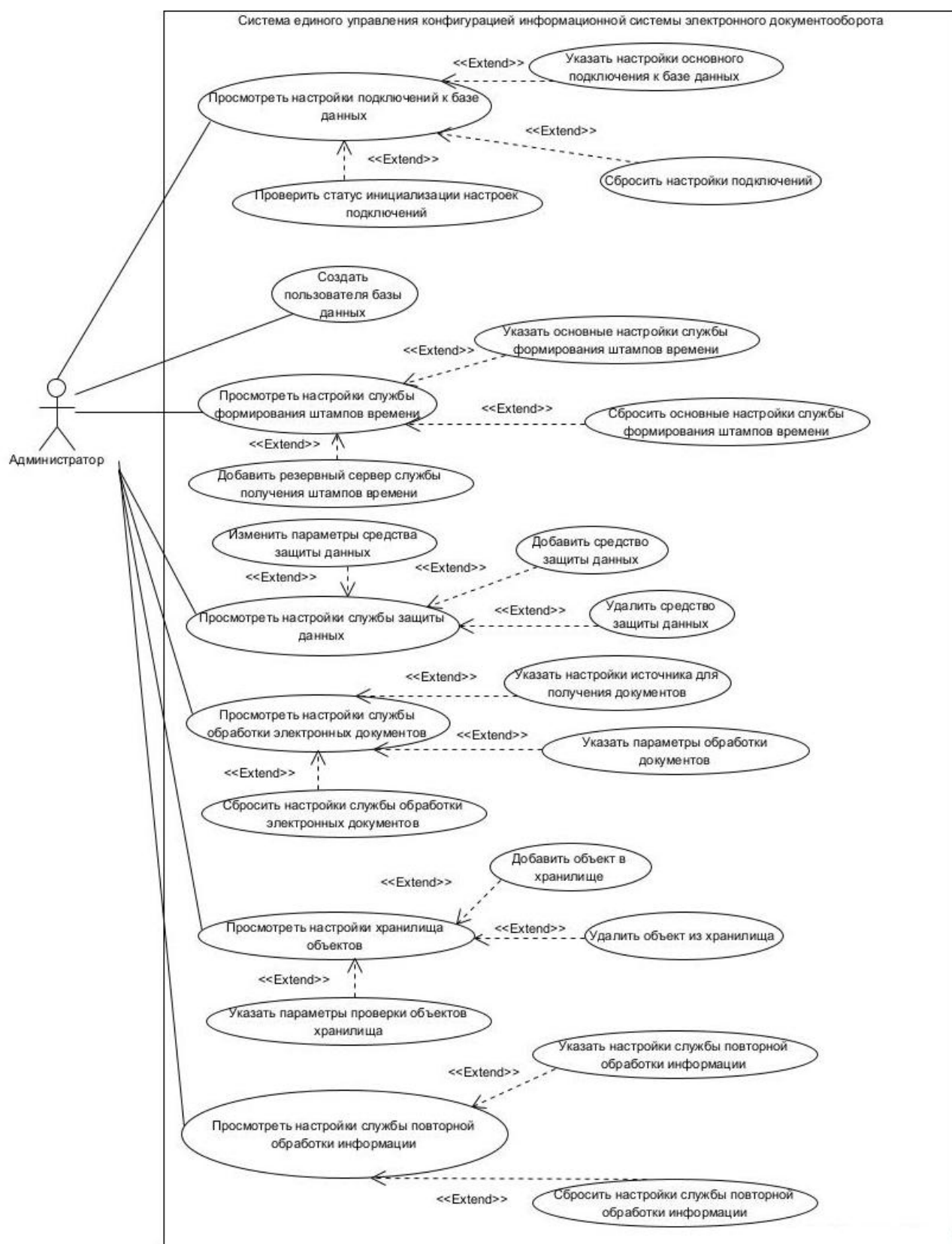


Рисунок 4 – Диаграмма вариантов использования

Службы организованы единообразно и различаются в основном составом параметров, поэтому детальное рассмотрение функциональности приводится на

примере одной из них – управления подключениями к базе данных, обладающего наиболее представительным набором операций.

Поддержка управления подключениями к базе данных выступает в системе централизованным источником настроек подключений, к которым обращаются другие службы в процессе работы.

Различаются основное подключение и набор дополнительных подключений. Параметры подключения включают адрес сервера, порт, имя базы данных и учётные данные доступа.

Основные функциональные требования к поддержке управления подключениями к базе данных включают:

- получение и установку настроек основного подключения к базе данных, обеспечивающие централизованное управление параметрами подключения, используемыми службами-потребителями;
- проверку статуса базы данных, обеспечивающую определение её доступности и актуальности структуры с автоматическим выполнением создания или миграции базы данных при подтверждении со стороны администратора;
- сброс настроек основного подключения, обеспечивающий приведение системы к состоянию, соответствующему отсутствию заданного основного подключения;
- управление составом дополнительных подключений к базе данных, включающее получение списка подключений, добавление новых, изменение настроек ранее добавленных и удаление подключений из состава настроек;
- создание учётной записи пользователя базы данных с использованием учётных данных привилегированного пользователя;
- получение статуса инициализации настроек базы данных.

На основе диаграммы вариантов использования составлены спецификации основных вариантов использования. Спецификация для варианта использования «Указать настройки основного подключения к базе данных» представлена на рисунке 5.

Наименование: указать настройки основного подключения к базе данных
ID: 1
Краткое описание: администратор указывает настройки основного подключения к базе данных с предварительной проверкой её состояния
Главные акторы: администратор
Предусловия: служба управления подключениями к базе данных подключена к системе
Основной поток: <ol style="list-style-type: none"> 1. Администратор задает значения параметров подключения. 2. Администратор инициирует сохранение настроек. 3. Система проверяет статус базы данных и при необходимости выполняет её создание или миграцию структуры до текущей поддерживаемой версии. 4. Система сохраняет настройки в хранилище и применяет их к системе. 5. Система возвращает результат выполнения операции
Альтернативный поток: <ol style="list-style-type: none"> 1. Администратор задает значения параметров подключения с обязательными полями, оставленными незаполненными, либо указывает недоступную базу данных. 2. Система возвращает информацию об ошибке. 3. Сохранение настроек не выполняется
Постусловия: настройки основного подключения сохранены в хранилище и применены к системе

Рисунок 5 – Спецификация варианта использования «Указать настройки основного подключения к базе данных»

Спецификация для варианта использования «Создать пользователя базы данных» представлена на рисунке 6.

Наименование: создать пользователя базы данных
ID: 5
Краткое описание: администратор создает новую учетную запись пользователя базы данных с использованием учетных данных суперпользователя
Главные акторы: администратор
Предусловия: служба управления подключениями к базе данных подключена к системе
Основной поток: <ol style="list-style-type: none"> 1. Администратор задает значения параметров создания пользователя. 2. Администратор инициирует создание пользователя. 3. Система проверяет возможность подключения с использованием учетных данных суперпользователя. 4. Система создает учетную запись нового пользователя в базе данных. 5. Система возвращает результат выполнения операции.
Постусловия: в базе данных создана новая учетная запись пользователя с указанными именем и паролем

Рисунок 6 – Спецификация варианта использования «Создать пользователя базы данных»

Перечисленные функциональные требования определяют состав операций, которые система предоставляет администратору, и порядок их выполнения. Рассмотрение на примере управления подключениями к базе данных раскрывает наиболее представительную часть функциональности. Полученные требования служат основой для дальнейшего проектирования системы.

2.2 Анализ нефункциональных требований

Помимо функциональных требований, на разрабатываемые программные средства накладывается ряд нефункциональных требований [5]. Для данной задачи они являются определяющими: именно нефункциональные требования, а не отдельные операции, в наибольшей степени формируют облик системы и определяют выбор архитектурных решений.

Существенными для рассматриваемой задачи являются следующие нефункциональные требования:

- расширяемость;
- атомарность внесения изменений;
- согласованность данных;
- восстанавливаемость;
- масштабируемость;
- надёжность;
- отказоустойчивость;
- безопасность;
- переносимость.

Ниже приведено содержание каждого из требований.

Расширяемость – система должна допускать добавление поддержки новых служб, а также обновление и исключение поддержки существующих без модификации остальных компонентов. Это требование является ключевым: программные средства встраиваются в уже эксплуатируемую систему, состав служб которой со временем меняется, и расширение не должно затрагивать работающую часть.

Атомарность внесения изменений – изменение конфигурации должно применяться целиком либо не применяться вовсе. Частично применённое состояние недопустимо, поскольку оно нарушает корректность работы зависимых друг от друга служб.

Согласованность данных – конфигурация всех служб должна оставаться взаимно непротиворечивой. Изменение настроек одной службы, затрагивающее другие, должно становиться им известным, чтобы система в целом сохраняла целостное состояние.

Восстанавливаемость – система должна обеспечивать создание целостного снимка конфигурации и последующее восстановление к нему. Снимок должен соответствовать согласованному состоянию на момент его создания.

Масштабируемость – система должна поддерживать горизонтальное масштабирование служб-потребителей настроек. Поскольку такие службы не хранят собственных копий настроек, а получают их централизованно, увеличение числа их экземпляров не порождает задачи синхронизации настроек между экземплярами. Схема масштабирования служб-потребителей приведена на рисунке 7.

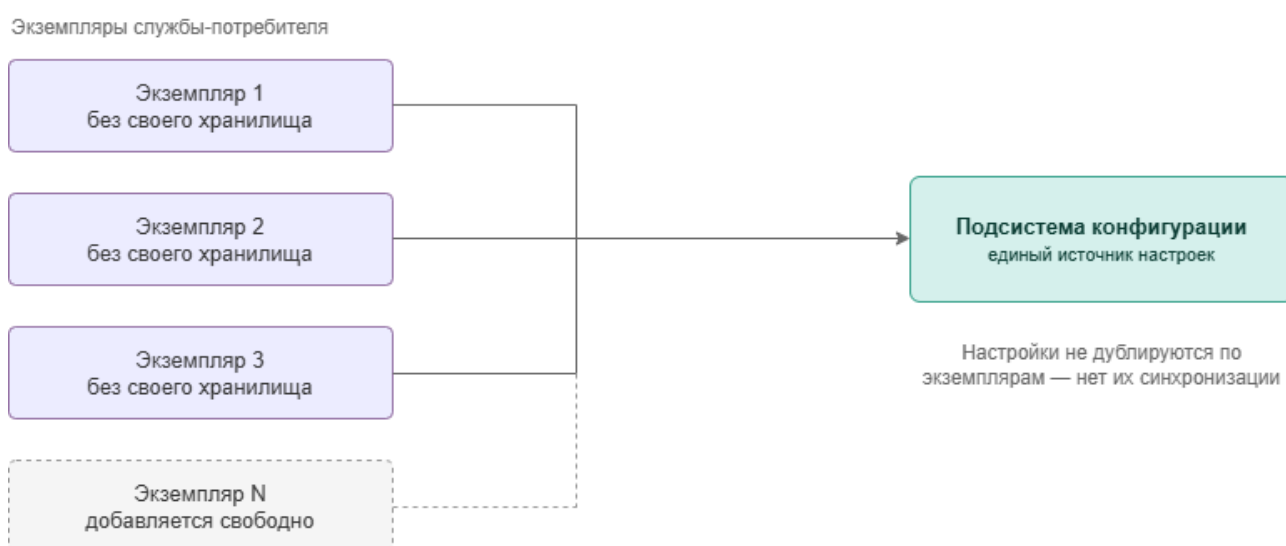


Рисунок 7 – Схема масштабирования служб-потребителей

Надёжность – программные средства должны сохранять корректность работы при длительной эксплуатации, в том числе при поступлении

некорректных входных данных, не приводя систему в неработоспособное состояние.

Отказоустойчивость – система должна сохранять работоспособность и целостность данных при сбоях отдельных компонентов, а также при сбое непосредственно в процессе применения настроек.

Безопасность – должны обеспечиваться защита хранимых настроек и связанных с ними учётных и идентификационных данных, а также разграничение доступа к операциям управления конфигурацией.

Переносимость – программные средства должны работать в целевой программной среде без привязки к специфической инфраструктуре и без необходимости развёртывания дополнительных внешних компонентов.

Перечисленные требования частично конфликтуют между собой: например, расширяемость за счёт изоляции служб противоречит согласованности, требующей совместного использования части данных; параллельная работа с настройками противоречит требованию их непротиворечивости; наконец, централизация хранения, обеспечивающая атомарность и восстанавливаемость, должна уживаться с горизонтальным масштабированием служб-потребителей. Совокупность этих требований и необходимость разрешения противоречий между ними определяют выбор архитектурных решений при проектировании системы.

3 Проектирование системы

3.1 Архитектура системы

Информационная система в целом построена как модульный монолит — единое развёртываемое приложение, составленное из слабо связанных модулей. В рамках единого процесса согласованность и атомарность изменений достигаются сравнительно легко, однако возможность пополнять состав управляемых служб без изменения существующего кода модульный монолит сам по себе не обеспечивает — а именно это требование является для подсистемы управления конфигурацией ведущим. Поэтому для неё архитектурное решение выбиралось отдельно и выводилось непосредственно из её нефункциональных требований.

Нефункциональные требования влияли на выбор архитектуры в разной степени. Их относительный вес при принятии архитектурных решений показан на рисунке 8.

Размер блока и число в правом углу — вес характеристики при выборе архитектуры (шкала 1–10)

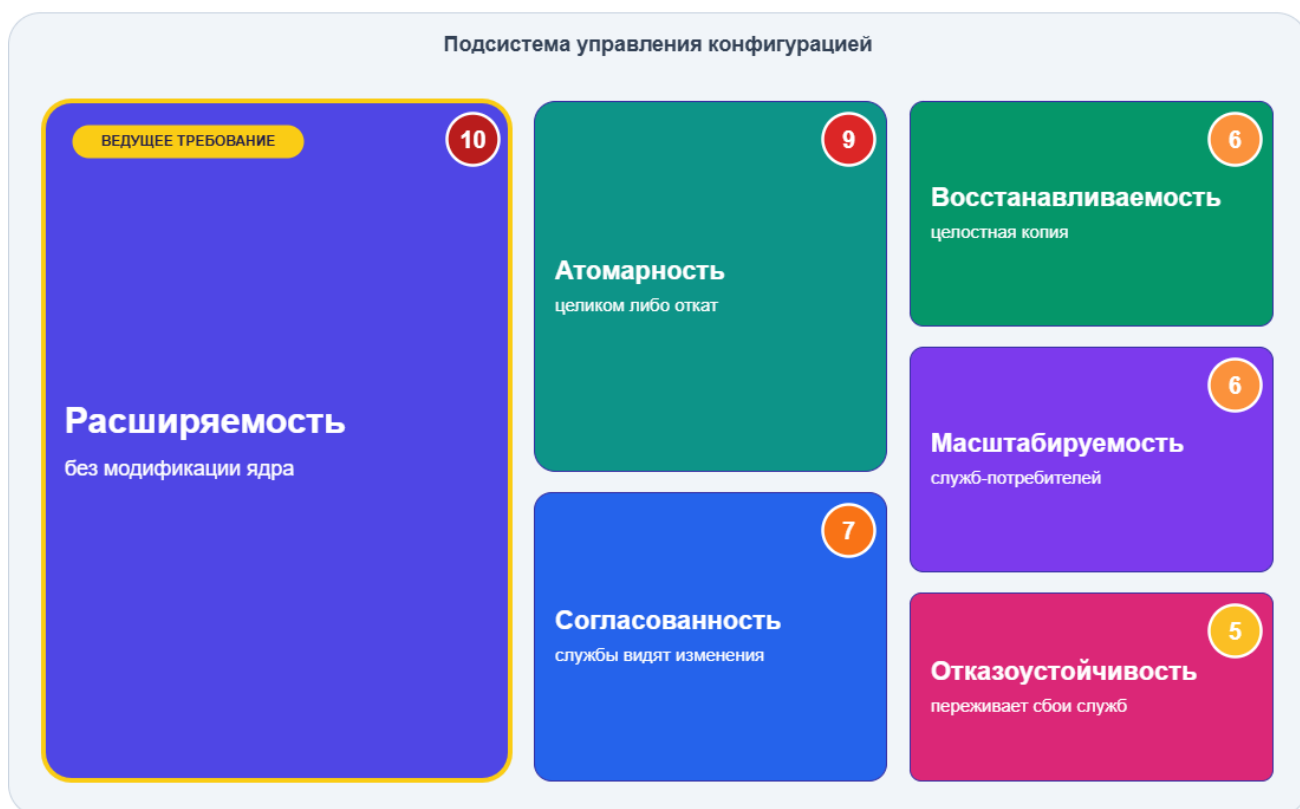


Рисунок 8 – Вес нефункциональных требований при выборе архитектуры

Наибольший вес получило требование расширяемости: именно его не обеспечивает модульный монолит, и ради него подсистема выделяется отдельно. Высокий вес имеют атомарность и согласованность: настройки должны изменяться как единое целое и одинаково восприниматься службами. Восстанавливаемость и масштабируемость оценены средним весом, поскольку достигаются как следствие принятых решений по хранению. Наименьший вес у отказоустойчивости: подсистема обслуживает административную, а не высоконагруженную систему.

Проектирование велось как вывод архитектуры из качественных атрибутов: ведущее требование определяет общий архитектурный стиль, а остальные требования либо удовлетворяются как следствие выбранного стиля, либо обеспечиваются дополнительными тактиками, либо вступают с ним в противоречие, которое разрешается явно. Соответствие нефункциональных требований и принятых архитектурных решений показано на рисунке 9.

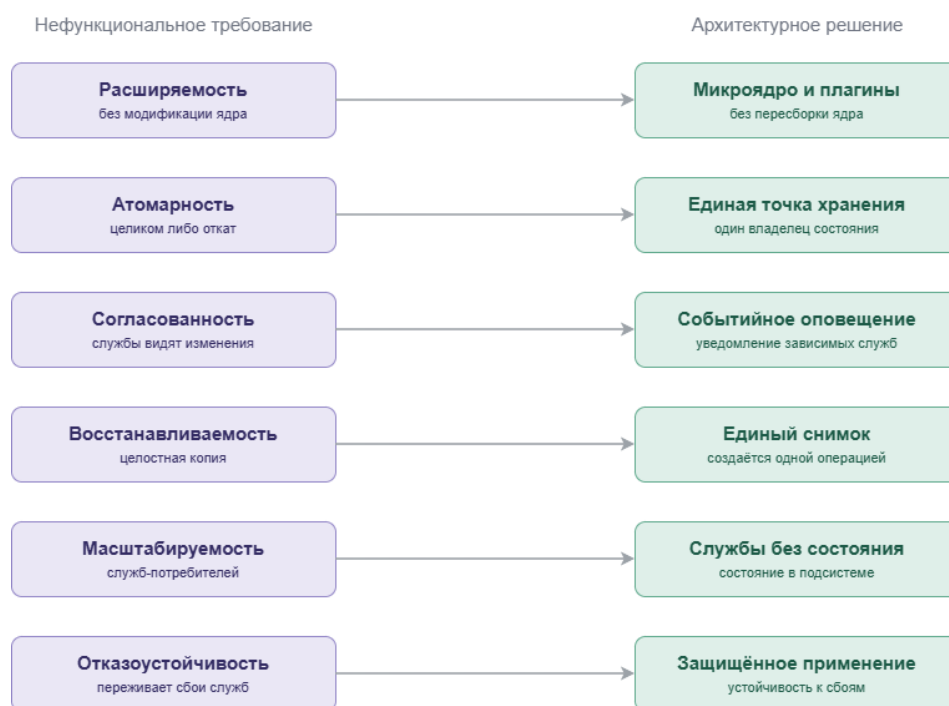


Рисунок 9 – Соответствие нефункциональных требований и архитектурных решений

Расширяемость как ведущее требование определила выбор микроядерной архитектуры: система разделяется на ядро и набор независимых

функциональных модулей – плагинов, реализующих общий программный интерфейс и подключаемых к ядру при запуске [6]. Добавление поддержки новой службы сводится к разработке отдельного плагина и не затрагивает ядро и остальные модули. При этом расширяемость вступает в противоречие с согласованностью: изоляция плагинов, необходимая для независимого подключения, препятствует совместному использованию настроек. Противоречие разрешается выделением общих интерфейсов межплагинного взаимодействия – плагины предоставляют свои возможности друг другу через специально определённые контракты, не образуя прямых зависимостей.

Атомарность и восстанавливаемость определили централизованную организацию хранения: всё состояние конфигурации принадлежит единой подсистеме, выступающей единственным владельцем настроек. Поскольку состояние сосредоточено в одном месте и под единым управлением, изменение применяется целиком либо откатывается, а резервная копия представляет собой единый согласованный снимок, который не может рассинхронизироваться. Напротив, при хранении настроек обособленно в каждой службе резервная копия собирается из независимых частей и легко теряет целостность. Кроме того, при централизованном хранении создание резервной копии сводится к снятию одного снимка с единого источника – это простая операция, тогда как сбор настроек по каждой службе требует их согласования и легко нарушается. Сопоставление двух подходов приведено на рисунке 10.

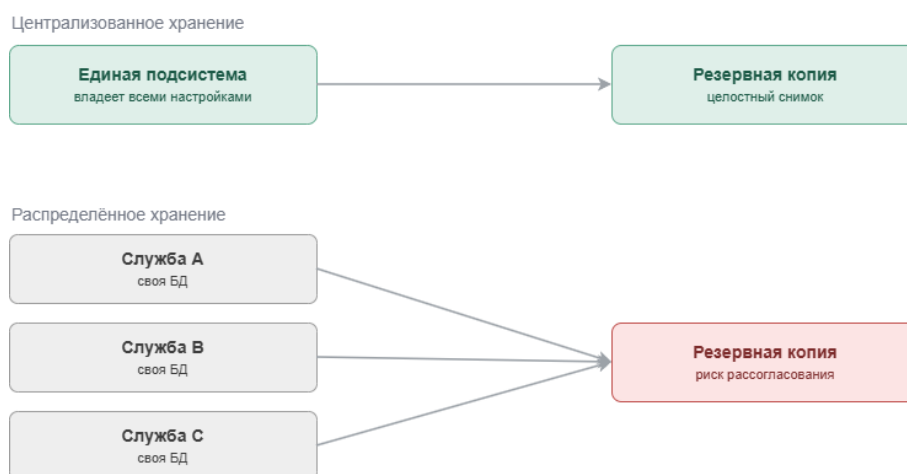


Рисунок 10 – Сопоставление подходов к управлению резервными копиями

Согласованность определила способ оповещения зависимых служб и разграничение параллельных операций.

Так как все настройки проходят через единую подсистему, уведомление об изменении сводится к одному событию из единой точки, а не к распределённому протоколу согласования, в котором пришлось бы определять состав и порядок оповещаемых служб. Параллельные операции администрирования при этом не должны нарушать согласованность, для чего применяется разграничение параллельного доступа к ресурсам подсистемы – на уровне отдельного ресурса и на уровне подсистемы в целом; этим разрешается противоречие между параллелизмом и согласованностью. Схема оповещения приведена на рисунке 11.



Рисунок 11 – Схема оповещения об изменении настроек

Масштабируемость потребовала, чтобы централизация хранения не препятствовала горизонтальному масштабированию служб-потребителей. Противоречие между централизацией, обеспечивающей атомарность, и масштабируемостью разрешается тем, что в единой подсистеме сосредоточена только конфигурация, но не функциональная нагрузка: целевые службы не держат собственных хранилищ настроек и обращаются к единому источнику, благодаря чему увеличение числа их экземпляров не порождает задачи синхронизации настроек между ними.

Отказоустойчивость и надёжность определили способ применения настроек: оно выполняется по двум каналам – инициируемому администратором

и фоновому – единым путём применения; фоновый канал периодически приводит состояние целевой службы в соответствие с сохранёнными настройками и сохраняет устойчивость при длительной недоступности служб. Безопасность определила обращение с защищаемыми учётными данными: они не размещаются в долговременном хранилище. Переносимость определила выбор платформы и средств хранения, не требующих развёртывания дополнительной серверной инфраструктуры в рамках разрабатываемого программного обеспечения.

Систему предполагается реализовать на платформе .NET 6 с использованием языка программирования C# и организовать в виде сервиса, объединяющего ядро и набор подключаемых плагинов, а также вспомогательного модуля удалённого администрирования, обеспечивающего обращение к плагинам по сети [7, 8].

В состав системы входят следующие компоненты:

- ядро, отвечающее за подключение плагинов на этапе старта системы и предоставление интерфейса для резервного копирования конфигурации;
- набор подключаемых плагинов, реализующих управление настройками отдельных служб;
- общая инфраструктурная сборка, содержащая базовые типы, средства синхронизации и общую логику, используемые ядром и плагинами;
- набор сборок межплагинного взаимодействия, содержащих интерфейсы, через которые отдельные плагины предоставляют свои возможности другим плагинам системы;
- модуль удалённого администрирования, обеспечивающий обращение к плагинам по сети;
- встраиваемая база данных, выступающая хранилищем настроек плагинов.

Общая структура подсистемы и связи между её компонентами представлены на рисунке 12.

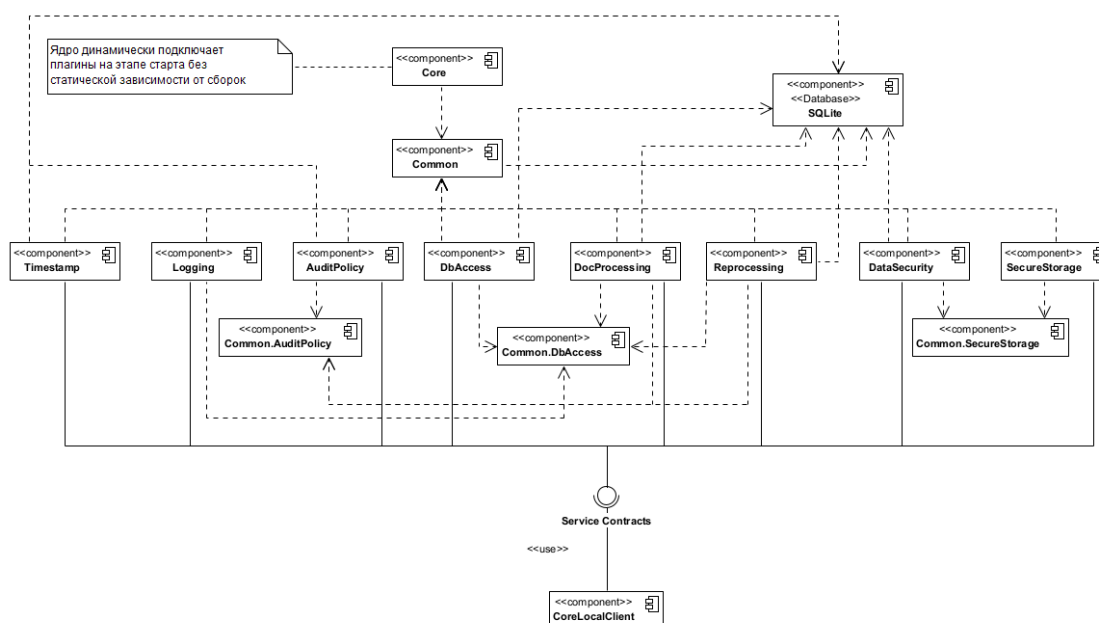


Рисунок 12 – Диаграмма компонентов разрабатываемой подсистемы

На ядро возлагается подключение плагинов на этапе старта системы. Подключение выполняется динамически, без статической зависимости ядра от их сборок, что позволяет пополнять состав плагинов без модификации ядра.

Плагины представляют собой основные функциональные компоненты системы. Каждый плагин управляет настройками отдельной службы: подключений к базе данных, службы формирования штампов времени, службы защиты данных, службы обработки электронных документов, хранилища объектов, службы повторной обработки информации, журналирования событий, а также настроек аудита. Плагины строятся на основе общей инфраструктурной сборки, предоставляющей базовые средства взаимодействия с ядром и общую логику, не зависящую от предметной области. Настройки сохраняются во встраиваемой базе данных SQLite через объектно-реляционную среду Entity Framework Core [9].

Часть плагинов предоставляет свои возможности другим плагином через выделенные сборки межплагинового взаимодействия. Так, плагин подключений к базе данных выступает поставщиком настроек подключений для остальных плагинов, а плагин хранилища объектов предоставляет службе защиты данных

функционал работы с объектами хранилища. При изменении настроек, затрагивающих другие плагины, заинтересованные плагины оповещаются через событие.

Модуль удалённого администрирования обеспечивает обращение к плагинам по сети и реализуется на фреймворке ASP.NET Core [10]; внешние клиенты обращаются к нему по протоколу HTTP [11]. Для обращения к плагинам используется Apache Thrift, обеспечивающий описание интерфейсов в платформонезависимом виде и автоматическую генерацию клиентских и серверных средств [12]. Модуль не имеет статической зависимости от сборок плагинов: обращение выполняется через клиенты, сгенерированные на основе сервисных контрактов каждого плагина.

3.2 Проектирование ядра системы

Ядро системы должно реализовывать две основные функции: подключение плагинов на этапе старта системы и обработку запросов на резервное копирование и восстановление конфигурации, поступающих от внешнего сервиса резервного копирования платформы.

Подсистема подключения плагинов организована вокруг класса `PluginManager`, на который возлагается задача обнаружения, загрузки и инициализации плагинов. Для отдельной загрузки сборок плагинов предполагается использовать класс `PluginLoadContext`, позволяющий каждому плагину использовать собственные версии библиотек-зависимостей без влияния на другие плагины и ядро системы.

Подсистема резервного копирования организована вокруг класса `ConfigurationBackupHandler`. При сохранении конфигурации ядро формирует единую структуру, собирая настройки от каждого плагина; при восстановлении выполняет обратное преобразование. Поддержка изменений формата сохраняемых настроек между версиями системы возлагается на класс `ConfigurationVersions`.

Диаграмма классов ядра представлена на рисунке 13 [13].

Описание классов ядра системы представлено в таблице 3.

Таблица 3 – Описание классов ядра системы

Класс	Назначение
PluginManager	Менеджер для загрузки и управления плагинами
PluginLoadContext	Контекст загрузки сборок для изоляции плагинов
PluginManagerConfig	Конфигурация менеджера плагинов
PluginConfig	Конфигурация плагина
ServiceCollectionExtension	Набор вспомогательных методов, расширяющий интерфейс IServiceCollection
PluginExtensions	Вспомогательные методы для работы с плагинами
ConfigurationBackupHandler	Реализация обработчика операций с резервными копиями
ConfigurationBackupServer	Реализация сервиса для управления резервным копированием настроек
ConfigurationVersions	Версии поддерживаемых конфигураций для резервного копирования
ConfigurationConverter	Вспомогательный класс для работы с моделями, связанными с конфигурацией

Принятые в составе ядра решения – отдельная загрузка сборок плагинов и поддержка миграции формата сохраняемых настроек – обеспечат возможность дальнейшего пополнения состава плагинов и эволюции формата конфигурации без нарушения работоспособности существующих компонентов системы.

3.3 Проектирование общей инфраструктурной сборки

Общая инфраструктурная сборка предназначена для размещения базовых типов, интерфейсов и средств, используемых ядром и плагинами системы. Сборка не будет реализовывать функциональность какой-либо предметной области и содержит общие технические возможности.

В составе общей инфраструктурной сборки предполагается выделить следующие группы:

- базовые интерфейсы плагинов, через которые они подключаются к ядру;
- средства синхронизации для согласованной работы плагинов при одновременном обращении;

- средства резервного копирования – сборщик настроек от плагинов и связанные с ним интерфейсы;
- средства работы с базой данных – общий контекст создания схемы базы данных, базовые интерфейсы хранилищ и поддержка транзакций;
- общие средства протоколирования, валидации и работы со строковыми и временными данными.

Вынесение общих типов и средств в отдельную сборку позволяет избежать дублирования и обеспечить единый подход к типовым техническим задачам во всех компонентах системы.

3.4 Проектирование плагиновой подсистемы

Плагиновая подсистема состоит из восьми плагинов, каждый из которых отвечает за управление настройками отдельной подсистемы. По выполняемой роли плагины разделены на три группы.

К первой группе относятся плагины настройки целевых служб: плагин управления службой защиты данных, плагин управления службой формирования штампов времени, плагин управления настройками обработки электронных документов, плагин управления службой повторной обработки информации и плагин управления службой журналирования событий.

Ко второй группе относятся плагины, предоставляющие свои настройки другим плагином системы: плагин управления подключениями к базе данных и плагин управления настройками аудита. Каждый плагин этой группы сохраняет настройки в собственном хранилище и предоставляет их через соответствующую сборку межплагинового взаимодействия.

Отдельно от двух групп выделен плагин хранилища объектов. Он не хранит собственных настроек и выступает промежуточным слоем к внешнему хранилищу объектов: перенаправляет управляющие запросы и предоставляет другим плагином функционал работы с объектами хранилища через сборку межплагинового взаимодействия.

3.4.1 Типовая структура плагина

В основе каждого плагина лежит типовая структура классов, которая дополняется или сокращается в зависимости от роли плагина. Типовая структура включает классы, обеспечивающие подключение плагина к ядру, приём запросов от администратора, хранение настроек, применение настроек к целевой службе и обращение к настраиваемой службе для выполнения операций.

Состав внедряемых зависимостей класса Handler различается у разных плагинов в зависимости от характера выполняемых операций. Handler может обращаться к Storage для чтения и записи настроек, к Setuper для запуска применения настроек, а также к Provider для прямого обращения к настраиваемой службе.

Диаграмма классов типовой структуры плагина представлена на рисунке 14.

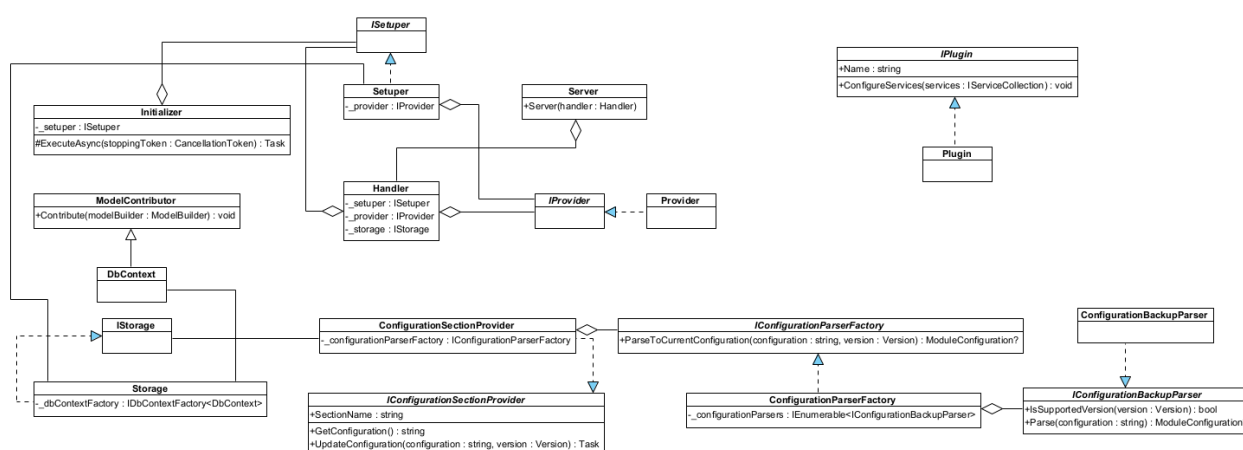


Рисунок 14 – Диаграмма классов типовой структуры плагина

Описание классов типовой структуры плагина представлено в таблице 4.

Таблица 4 – Описание классов типовой структуры плагина

Класс	Назначение
Plugin	Класс, регистрирующий зависимости плагина в контейнере
IPlugin	Интерфейс плагина службы настроек

Продолжение таблицы 4

Класс	Назначение
ConfigurationSectionProvider	Класс для управления секцией конфигурации плагина для резервного копирования
IConfigurationSectionProvider	Интерфейс для управления секцией конфигурации плагина
IConfigurationParserFactory	Интерфейс фабрики анализаторов конфигурации
ConfigurationParserFactory	Фабрика анализаторов для приведения конфигурации к актуальной версии
IConfigurationBackupParser	Интерфейс анализатора конфигурации конкретной версии
ConfigurationBackupParser	Реализация анализатора конфигурации конкретной версии
Handler	Обработчик запросов администратора
Server	Сервер для приёма запросов от администратора
Storage	Хранилище настроек плагина
IStorage	Интерфейс хранилища настроек плагина
DbContext	Контекст для взаимодействия с базой данных
ISetuper	Интерфейс применения настроек плагина
Setuper	Класс применения настроек плагина к настраиваемой службе
Initializer	Фоновый сервис периодического применения настроек
IProvider	Интерфейс провайдера для обращения к настраиваемой службе
Provider	Провайдер для обращения к настраиваемой службе

Использование типовой структуры позволяет обеспечить единый подход к построению плагинов и упростить разработку новых. У плагинов-поставщиков настроек отсутствуют классы Setuper и Initializer, так как они не управляют целевой службой, а только предоставляют свои настройки другим плагинам. У плагина управления хранилищем объектов также отсутствуют Storage, DbContext и ConfigurationSectionProvider.

Взаимодействие классов внутри типовой структуры построено по единой схеме: класс Server принимает запросы администратора и передаёт их обработчику Handler, который в зависимости от запроса сохраняет настройки в Storage, инициирует их применение через Setuper либо обращается к настраиваемой службе напрямую через Provider. Применение настроек при этом

отделено от их хранения: Setuper переносит сохранённую конфигурацию в целевую службу, а фоновый сервис Initializer периодически повторяет это применение, что позволяет восстанавливать согласованность настроек после перезапуска или временной недоступности службы. Такое разделение обязанностей между приёмом запросов, хранением, применением и обращением к службе упрощает сопровождение плагинов и позволяет изменять отдельные их аспекты независимо друг от друга.

Базовые интерфейсы и классы из общей инфраструктурной сборки, входящие в типовую структуру, на диаграммах отдельных плагинов повторно не будут приводиться.

3.4.2 Проектирование плагина управления службой защиты данных

Плагин управления службой защиты данных предназначен для управления настройками служб защиты данных системы и работы с настройками доступа к объектам хранилища.

Плагин построен по типовой структуре с дополнениями, обеспечивающими поддержку нескольких областей применения, нескольких типов поставщиков и работу с настройками доступа к объектам хранилища с использованием функционала из сборки межплагинного взаимодействия. Области применения различаются через перечисление `DataSecurityType`, передаваемое в методах работы с настройками для выбора целевой области применения в системе. Заданные администратором настройки сохраняются в локальной базе данных плагина и применяются к целевой службе через соответствующий провайдер.

Конкретные обработчики запросов администратора и модели данных настроек для отдельных областей применения на диаграмме не приводятся в целях обобщения архитектурного представления плагина.

Диаграмма классов плагина управления настройками службой защиты данных представлена на рисунке 15.

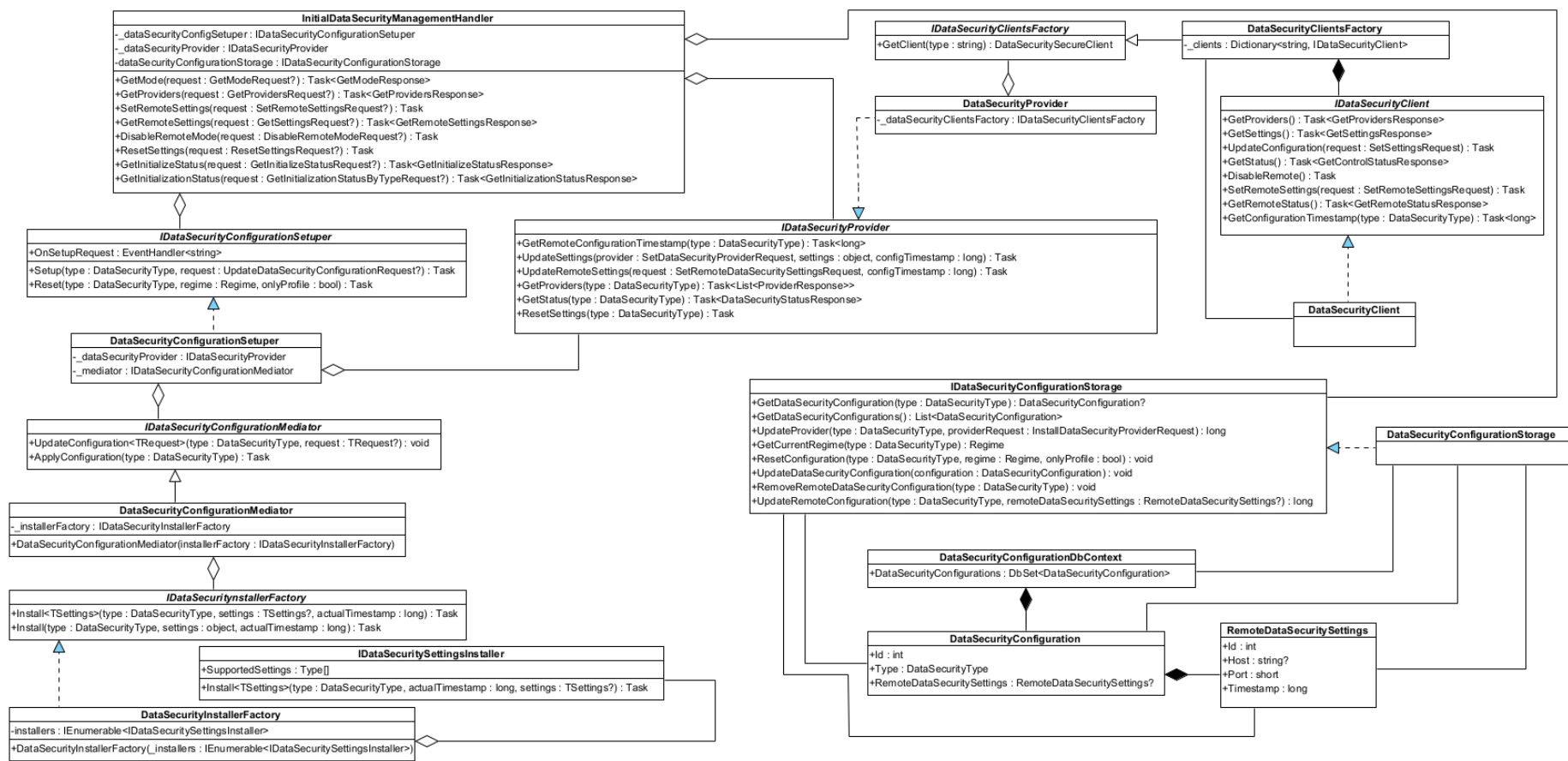


Рисунок 15 – Диаграмма классов плагина управления службой защиты данных

Описание классов плагина управления службой защиты данных представлено в таблице 5.

Таблица 5 – Описание классов плагина управления службой защиты данных

Класс	Назначение
InitialDataSecurityManagementHandler	Обработчик запросов администратора к общим настройкам защиты данных
IDataSecurityConfigurationSetuper	Интерфейс применения настроек защиты данных
DataSecurityConfigurationSetuper	Класс применения настроек защиты данных
IDataSecurityConfigurationMediator	Интерфейс посредника при обращении к базе данных настроек защиты данных
DataSecurityConfigurationMediator	Посредник при обращении к базе данных настроек защиты данных
IDataSecurityInstallerFactory	Интерфейс фабрики установщиков настроек защиты данных
DataSecurityInstallerFactory	Фабрика установщиков настроек защиты данных
IDataSecuritySettingsInstaller	Интерфейс установщика настроек защиты данных
IDataSecurityConfigurationStorage	Интерфейс хранилища настроек защиты данных
DataSecurityConfigurationStorage	Хранилище настроек защиты данных
DataSecurityConfigurationDbContext	Контекст для взаимодействия с базой данных настроек защиты данных
DataSecurityConfiguration	Модель данных настроек защиты данных
RemoteDataSecuritySettings	Модель данных настроек удалённой защиты данных
IDataSecurityProvider	Интерфейс обращения к внешней службе защиты данных
DataSecurityProvider	Провайдер для обращения к внешней службе защиты данных
IDataSecurityClient	Интерфейс клиента службы защиты данных
DataSecurityClient	Клиент службы защиты данных
IDataSecurityClientsFactory	Интерфейс фабрики клиентов служб защиты данных
DataSecurityClientsFactory	Фабрика клиентов служб защиты данных

Для работы с настройками доступа к объектам хранилища в плагине выделена отдельная группа классов. Класс ObjectAccessProvider использует интерфейс IDataSecurityMediator из сборки межплагинового взаимодействия для получения функционала работы с объектами хранилища от плагина хранилища объектов.

Серверную часть интерфейса реализует класс ObjectAccessManagementHandler, обрабатывающий запросы на получение,

добавление, удаление и изменение состояния объектов, а непосредственный обмен с целевой службой выполняется через клиент `ObjectAccessClient`.

Диаграмма классов работы с настройками доступа к объектам хранилища представлена на рисунке 16.

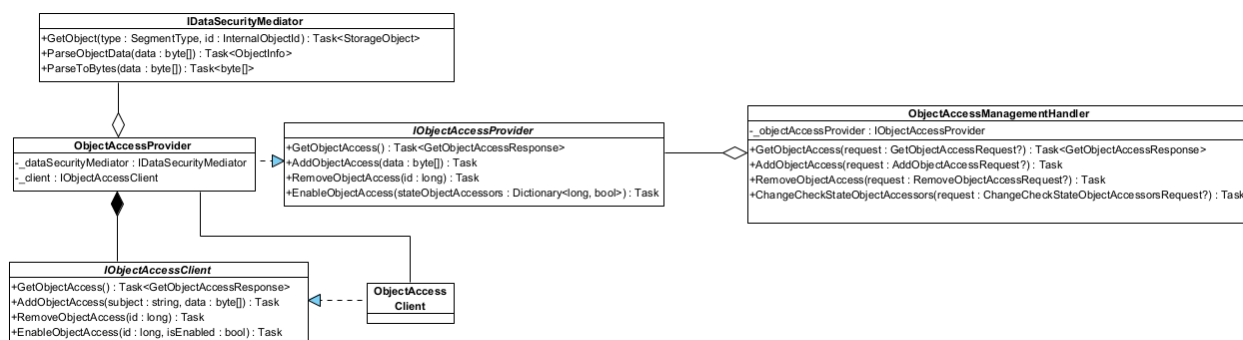


Рисунок 16 – Диаграмма классов работы с настройками доступа к объектам хранилища

Описание классов работы с настройками доступа к объектам хранилища представлено в таблице 6.

Таблица 6 – Описание классов работы с настройками доступа к объектам хранилища

Класс	Назначение
<code>ObjectAccessManagementHandler</code>	Обработчик запросов администратора к настройкам доступа к объектам хранилища
<code>IObjectAccessProvider</code>	Интерфейс работы с настройками доступа к объектам хранилища
<code>ObjectAccessProvider</code>	Класс работы с настройками доступа к объектам хранилища
<code>IObjectAccessClient</code>	Интерфейс клиента работы с настройками доступа к объектам хранилища
<code>ObjectAccessClient</code>	Клиент работы с настройками доступа к объектам хранилища

Такая структура плагина обеспечивает управление настройками нескольких областей применения через единые интерфейсы плагина с возможностью добавления отдельных установщиков, а также работу с настройками доступа к объектам хранилища с использованием функционала из общей инфраструктуры межплагинового взаимодействия.

3.4.3 Проектирование плагина управления службой формирования штампов времени

Плагин управления службой формирования штампов времени предназначен для управления настройками клиентов и серверов службы формирования штампов времени. Плагин построен по типовой структуре с дополнениями, обеспечивающими отдельную работу с настройками клиентов и серверов.

Класс `TimestampManagementHandler` принимает запросы администратора по управлению настройками клиентов службы формирования штампов времени и поддерживает работу с несколькими областями применения, различаемыми через перечисление `DataSecurityType`. `Handler` использует интерфейс `IDataSecurityStorage` из сборки межплагинового взаимодействия для проверки наличия объекта хранилища, получения установленных параметров и проверки статуса инициализации настроек защиты данных.

Настройки клиентов службы формирования штампов времени сохраняются в базе данных и применяются как по запросу администратора, так и в фоновом режиме. Настройки серверов службы формирования штампов времени применяются только в фоновом режиме в зависимости от состояния инициализации настроек защиты данных, поскольку для формирования штампов времени требуется готовность функций защиты данных.

Помимо параметров подключения, управление настройками серверов включает формирование резервного списка серверов с заданием приоритетов: при недоступности основного сервера служба переключается на резервный в соответствии с приоритетом. Сведения об отдельном сервере описываются моделью `TimestampServerSettings` и сохраняются совместно с остальными настройками плагина в базе данных.

Диаграмма классов плагина управления службой формирования штампов времени представлена на рисунке 17.

Описание классов плагина управления службой формирования штампов времени представлено в таблице 7.

Таблица 7 – Описание классов плагина управления службой формирования штампов времени

Класс	Назначение
TimestampManagementHandler	Обработчик запросов администратора к плагину управления службой формирования штампов времени
ITimestampClientConfigurationSetuper	Интерфейс применения настроек клиентов службы формирования штампов времени
TimestampClientConfigurationSetuper	Класс применения настроек клиентов службы формирования штампов времени
ITimestampServerConfigurationSetuper	Интерфейс применения настроек серверов службы формирования штампов времени
TimestampServerConfigurationSetuper	Класс применения настроек серверов службы формирования штампов времени
ITimestampConfigurationStorage	Интерфейс хранилища настроек клиентов службы формирования штампов времени
TimestampConfigurationStorage	Хранилище настроек клиентов службы формирования штампов времени
TimestampConfigurationDbContext	Контекст для взаимодействия с базой данных настроек клиентов службы формирования штампов времени
TimestampConfiguration	Модель данных настроек клиента службы формирования штампов времени
TimestampServerSettings	Модель данных настроек сервера службы формирования штампов времени
ITimestampProvider	Интерфейс обращения к клиентам службы формирования штампов времени
TimestampProvider	Провайдер для обращения к клиентам службы формирования штампов времени
ITimestampServerProvider	Интерфейс обращения к серверам службы формирования штампов времени
TimestampServerProvider	Провайдер для обращения к серверам службы формирования штампов времени
ITimestampThriftClient	Интерфейс для взаимодействия с клиентами службы формирования штампов времени
TimestampThriftClient	Реализация интерфейса для взаимодействия с клиентами службы формирования штампов времени
ITimestampServerThriftClient	Интерфейс для взаимодействия с серверами службы формирования штампов времени
TimestampServerThriftClient	Реализация интерфейса для взаимодействия с серверами службы формирования штампов времени

Такая структура плагина обеспечивает раздельное управление настройками клиентов и серверов службы формирования штампов времени, а также проверку настроек защиты данных через сборку межплагинового взаимодействия.

3.4.4 Проектирование плагина управления настройками обработки электронных документов

Плагин управления настройками обработки электронных документов предназначен для управления настройками сервиса обработки электронных документов и формирования квитанций с результатами проверок. Плагин построен по типовой структуре с дополнениями, обеспечивающими работу с обработкой документов при внешнем и внутреннем обмене и поддержку подключаемых модулей дополнительной проверки.

Для приёма запросов администратора в плагине предусмотрено четыре обработчика – `DocProcessingCommonManagementHandler` для управления общими настройками обработки электронных документов, `ExternalExchangeManagementHandler` и `InternalExchangeManagementHandler` для управления настройками обработки электронных документов в рамках внешнего и внутреннего обмена соответственно, а также `ValidationPluginsManagementHandler` для управления подключаемыми модулями дополнительной проверки.

Общие настройки обработки, а также настройки для внешнего и внутреннего обмена описываются отдельными моделями данных и включают параметры подключения к источнику данных, из которого служба получает документы.

Все настройки сохраняются в базе данных и применяются к службе как по запросу администратора, так и в фоновом режиме.

Диаграмма классов плагина управления настройками обработки электронных документов представлена на рисунке 18.

Описание классов плагина управления настройками обработки электронных документов представлено в таблице 8.

Таблица 8 – Описание классов плагина управления настройками обработки электронных документов

Класс	Назначение
DocProcessingCommonManagementHandler	Обработчик запросов администратора к общим настройкам обработки электронных документов
ExternalExchangeManagementHandler	Обработчик запросов администратора к настройкам обработки электронных документов для внешнего обмена
InternalExchangeManagementHandler	Обработчик запросов администратора к настройкам обработки электронных документов для внутреннего обмена
ValidationPluginsManagementHandler	Обработчик запросов администратора к подключаемым модулям дополнительной проверки
IDocProcessingConfigurationSetuper	Интерфейс применения настроек обработки электронных документов
DocProcessingConfigurationSetuper	Класс применения настроек обработки электронных документов
IDocProcessingConfigurationStorage	Интерфейс хранилища настроек обработки электронных документов
DocProcessingConfigurationStorage	Хранилище настроек обработки электронных документов
DocProcessingConfigurationDbContext	Контекст для взаимодействия с базой данных настроек обработки электронных документов
DocProcessingConfiguration	Модель данных настроек обработки электронных документов
DocProcessingSettings	Модель данных общих настроек обработки электронных документов
DocProcessingSettingsForExternalExchange	Модель данных настроек обработки электронных документов для внешнего обмена
DocProcessingSettingsForInternalExchange	Модель данных настроек обработки электронных документов для внутреннего обмена
DataSourceManagerSettings	Модель данных настроек источника данных
ValidationPluginSettings	Модель данных настроек подключаемого модуля дополнительной проверки
IDocProcessingProvider	Интерфейс обращения к службе обработки электронных документов
DocProcessingProvider	Провайдер для обращения к службе обработки электронных документов

Продолжение таблицы 8

Класс	Назначение
IDocProcessingWorkerClient	Интерфейс клиента службы обработки электронных документов
DocProcessingWorkerClient	Клиент службы обработки электронных документов
IValidationPluginsProvider	Интерфейс работы с подключаемыми модулями дополнительной проверки
ValidationPluginsProvider	Класс работы с подключаемыми модулями дополнительной проверки
IDocProcessingStateChecker	Интерфейс проверки состояния службы обработки электронных документов
IDataSourceConnectionChecker	Интерфейс проверки соединения с источником данных

Такая структура плагина обеспечивает управление настройками обработки электронных документов при внешнем и внутреннем обмене, подключение модулей дополнительной проверки и согласованное применение настроек с настройками подключения к базе данных и аудита через сборки межплагинового взаимодействия.

3.4.5 Проектирование плагина управления подключениями к базе данных

Плагин управления подключениями к базе данных предназначен для управления настройками подключений к базе данных, используемых другими плагинами системы. Плагин является поставщиком настроек для других плагинов системы. В плагине не предусмотрено применение настроек к внешней службе и фоновая инициализация, так как настройки применяются непосредственно при их изменении. Плагин дополнен функционалом предоставления настроек другим плагинам и поддержки миграций базы данных.

Для работы с потребителями настроек предусмотрено два интерфейса – IDbAccessMediator с полным набором операций над настройками подключения и сужающий интерфейс IDbAccessManager, предоставляющий потребителям только возможность получения настроек и подписки на их изменения.

Диаграмма классов плагина управления подключениями к базе данных представлена на рисунке 19.

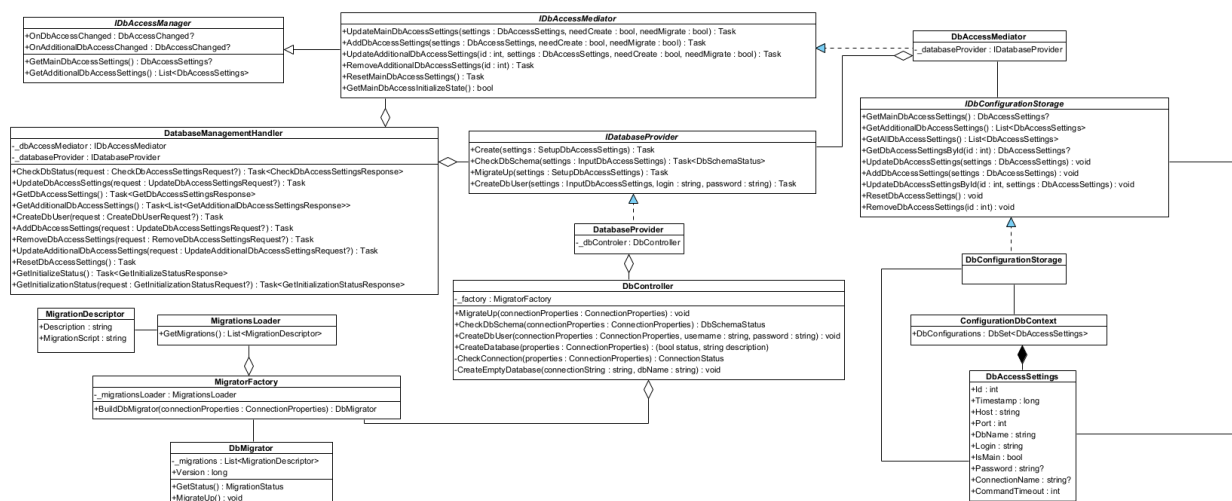


Рисунок 19 – Диаграмма классов плагина управления подключениями к базе данных

Описание классов плагина управления подключениями к базе данных представлено в таблице 9.

Таблица 9 – Описание классов плагина управления подключениями к базе данных

Класс	Назначение
DatabaseManagementHandler	Обработчик запросов администратора к настройкам подключений к базе данных
IDbAccessMediator	Интерфейс посредника для управления настройками подключений к базе данных
IDbAccessManager	Интерфейс получения настроек подключений к базе данных
DbAccessMediator	Посредник для управления настройками подключений к базе данных
IDbConfigurationStorage	Интерфейс хранилища настроек подключений к базе данных
DbConfigurationStorage	Хранилище настроек подключений к базе данных
ConfigurationDbContext	Контекст для взаимодействия с базой данных настроек подключения
DbAccessSettings	Модель данных настроек подключения к базе данных
IDatabaseProvider	Интерфейс провайдера работы с базой данных
DatabaseProvider	Провайдер работы с базой данных
DbController	Контроллер работы с базой данных

Продолжение таблицы 9

Класс	Назначение
MigratorFactory	Фабрика создания миграторов базы данных
DbMigrator	Класс выполнения миграций базы данных
MigrationsLoader	Загрузчик описаний миграций базы данных
MigrationDescriptor	Описание миграции базы данных

Такая структура плагина обеспечивает централизованное управление настройками подключений к базе данных, предоставление их другим плагинам системы через сборку межплагинового взаимодействия и поддержку миграций базы данных.

3.4.6 Проектирование плагина управления службой повторной обработки информации

Плагин управления службой повторной обработки информации построен по типовой структуре. Особенностью плагина является потребление настроек подключения к базе данных и настроек аудита от других плагинов системы для согласованной работы службы повторной обработки информации с базой данных и службой аудита.

Класс `ReprocessingConfigurationSetuper` использует интерфейсы `IDbAccessManager` и `IAuditPolicyManager` из сборок межплагинового взаимодействия для получения настроек подключения к базе данных и настроек аудита, а также подписан на события изменения этих настроек. При получении уведомления об изменении настроек класс инициирует применение новых настроек.

Помимо потребляемых настроек, плагин управляет собственными параметрами службы повторной обработки информации. Эти настройки принимаются от администратора классом `ReprocessingManagementHandler`, сохраняются в базе данных через хранилище плагина и применяются к целевой службе. Текущее состояние отслеживается через запрос статуса инициализации.

Диаграмма классов плагина управления службой повторной обработки информации представлена на рисунке 20.

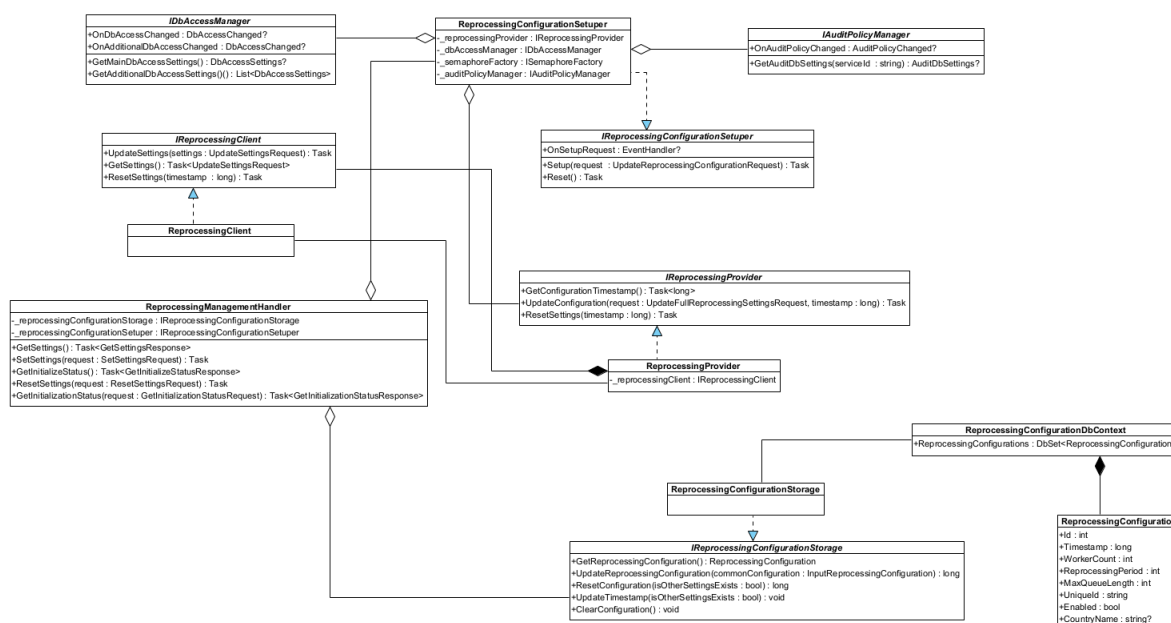


Рисунок 20 – Диаграмма классов плагина управления службой повторной обработки информации

Описание классов плагина управления службой повторной обработки информации представлено в таблице 10.

Таблица 10 – Описание классов плагина управления службой повторной обработки информации

Класс	Назначение
ReprocessingManagementHandler	Обработчик запросов администратора к плагину управления службой повторной обработки информации
IReprocessingConfigurationSetuper	Интерфейс применения настроек службы повторной обработки информации
ReprocessingConfigurationSetuper	Класс применения настроек службы повторной обработки информации
IReprocessingConfigurationStorage	Интерфейс хранилища настроек службы повторной обработки информации
ReprocessingConfigurationStorage	Хранилище настроек службы повторной обработки информации
ReprocessingConfigurationDbContext	Контекст для взаимодействия с базой данных настроек службы повторной обработки информации
ReprocessingConfiguration	Модель данных настроек службы повторной обработки информации
IReprocessingProvider	Интерфейс обращения к службе повторной обработки информации

Продолжение таблицы 10

Класс	Назначение
ReprocessingProvider	Провайдер для обращения к службе повторной обработки информации
IReprocessingClient	Интерфейс клиента службы повторной обработки информации
ReprocessingClient	Клиент службы повторной обработки информации

Такая структура плагина обеспечивает согласованное применение настроек службы повторной обработки информации с настройками других плагинов системы.

3.4.7 Особенности остальных плагинов

В состав плагиновой подсистемы помимо рассмотренных входят ещё три плагина: плагин управления службой журналирования событий, плагин управления настройками аудита и плагин хранилища объектов. По структуре эти плагины либо проще типовой, либо аналогичны уже рассмотренным, в связи с чем отдельные диаграммы для них не приводятся.

Плагин управления службой журналирования событий предназначен для запуска и поддержания работы службы чтения журнала событий. В плагине отсутствует обработчик запросов администратора, так как плагин не предоставляет настроек для управления. Применение настроек выполняется только в фоновом режиме без участия администратора. Плагин не хранит собственных настроек и получает настройки подключения к базе данных от плагина управления подключениями к базе данных через сборку межплагинового взаимодействия.

Плагин управления настройками аудита предназначен для управления настройками подсистемы аудита и предоставления их другим плагином системы. Плагин является поставщиком настроек и построен на основе типовой структуры с теми же отличиями, что и плагин управления подключениями к базе данных: в плагине не предусмотрено применение настроек к внешней службе и фоновая инициализация, так как настройки применяются непосредственно при их

изменении. Плагин дополнен функционалом предоставления настроек другим плагинам через интерфейс `IAuditPolicyManager` для потребителей.

Плагин хранилища объектов предназначен для управления объектами хранилища через взаимодействие с внешним хранилищем объектов. Плагин выступает прослойкой и не хранит собственных настроек, а перенаправляет запросы администратора к внешнему хранилищу. Также плагин реализует интерфейс `IDataSecurityMediator` из сборки межплагинного взаимодействия для предоставления функционала работы с объектами хранилища другим плагинам системы.

3.5 Проектирование механизма межплагинного взаимодействия

Для согласованной работы плагинов системы предусмотрен механизм межплагинного взаимодействия, обеспечивающий обмен настройками и функционалом между плагинами без прямой зависимости между ними. Взаимодействие выполняется через выделенные сборки межплагинного взаимодействия, каждая из которых отвечает за обмен в рамках определённой подсистемы и содержит интерфейсы и общие модели данных для участников взаимодействия.

Для сборок межплагинного взаимодействия, предназначенных для обмена настройками, в сборке предусмотрен интерфейс для плагинов-потребителей с операциями получения настроек и подписки на их изменения. Плагин-поставщик реализует данный интерфейс и предоставляет его потребителям через сборку межплагинного взаимодействия. Для уведомления потребителей об изменении настроек используется событийная подписка через интерфейс. При изменении настроек плагин-поставщик инициирует событие изменения, на которое реагируют подписанные потребители для согласованного применения зависимых настроек.

Для сборок межплагинного взаимодействия, предназначенных для предоставления функционала, предусмотрен интерфейс, реализуемый плагином-

поставщиком функционала и используемый плагинами-потребителями для прямого обращения.

В системе предусмотрено три сборки межплагинного взаимодействия: для настроек подключения к базе данных, для настроек подсистемы аудита и для работы с объектами хранилища и настройками защиты данных. Сборки межплагинного взаимодействия для настроек подключения к базе данных и настроек подсистемы аудита построены по схеме с интерфейсом получения настроек и событийной подпиской. Сборка межплагинного взаимодействия для работы с объектами хранилища и настройками защиты данных содержит интерфейс для предоставления функционала работы с объектами хранилища от плагина хранилища объектов плагину управления службой защиты данных, а также интерфейс для предоставления функционала проверки инициализации настроек защиты данных другим плагинам системы. Конкретные интерфейсы сборок межплагинного взаимодействия и связи плагинов с ними приведены на диаграммах классов плагинов, представленных ранее.

Такой механизм обеспечивает обмен настройками и функционалом между плагинами без нарушения принципа их независимости.

3.6 Проектирование подсистемы удалённого администрирования

Подсистема удалённого администрирования предназначена для предоставления администраторам системы удалённого доступа к управлению конфигурацией системы через стандартизированный сетевой интерфейс. Подсистема будет выступать прослойкой между интерфейсом администратора и ядром системы, преобразуя запросы администратора в формат, поддерживаемый ядром системы.

Подсистему предполагается построить по трёхуровневой схеме [14]. Уровень контроллеров принимает HTTP-запросы администратора и возвращает HTTP-ответы. Уровень провайдеров выполняет преобразование запросов в формат ядра системы и обращается к ядру для выполнения операций. Уровень

клиентов отвечает за непосредственную передачу запросов к ядру системы и получение ответов.

Структура подсистемы рассматривается на примере удалённого администрирования настроек службы формирования штампов времени. Диаграмма классов подсистемы удалённого администрирования настроек службы формирования штампов времени представлена на рисунке 21.

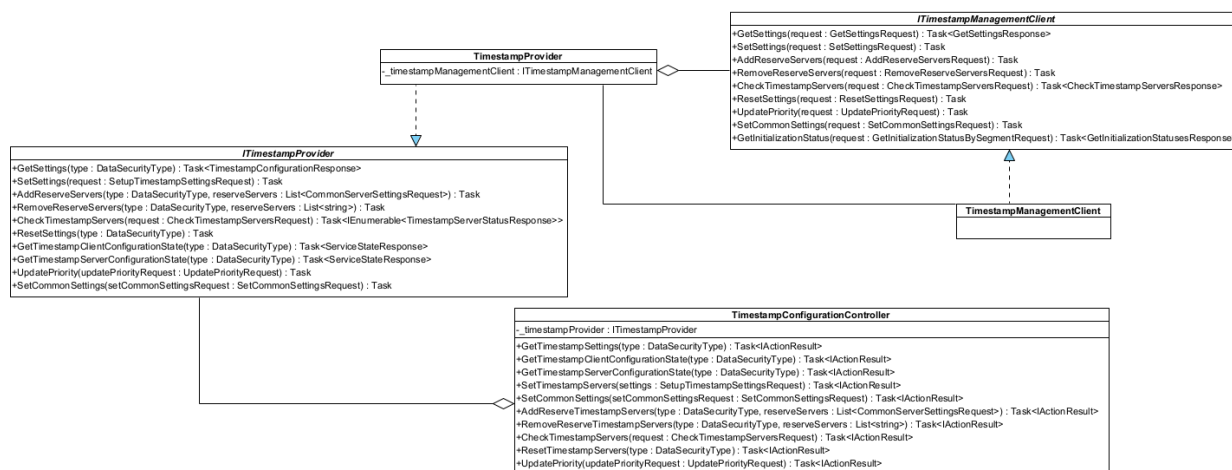


Рисунок 21 – Диаграмма классов удалённого администрирования службы формирования штампов времени

Описание классов подсистемы удалённого администрирования настроек службы формирования штампов времени представлено в таблице 11.

Таблица 11 – Описание классов удалённого администрирования службы формирования штампов времени

Класс	Назначение
TimestampConfigurationController	Контроллер приёма запросов администратора на управление настройками службы формирования штампов времени
ITimestampProvider	Интерфейс провайдера управления настройками службы формирования штампов времени
TimestampProvider	Провайдер управления настройками службы формирования штампов времени
ITimestampManagementClient	Интерфейс клиента ядра системы для управления настройками службы формирования штампов времени

Продолжение таблицы 11

Класс	Назначение
TimestampManagementClient	Клиент ядра системы для управления настройками службы формирования штампов времени

Аналогичные группы классов предполагается реализовать для удалённого администрирования каждой подсистемы, поддерживающей управление через интерфейс администратора.

Такая структура подсистемы удалённого администрирования обеспечивает единый подход к организации удалённого доступа администратора к управлению различными подсистемами.

4 Реализация системы

В состав разработанных программных средств входит несколько подсистем, каждая из которых решает свою задачу в общей архитектуре.

Рабочее место администратора представлено персональным компьютером с веб-браузером. Через браузер по защищённому протоколу HTTPS администратор обращается к веб-интерфейсу системы.

Точкой входа на стороне сервера выступает веб-сервер Nginx [15]. Он принимает входящие HTTPS-соединения и перенаправляет их на frontend-приложение.

Frontend-приложение реализует визуальное представление функций администрирования. Взаимодействие с серверной частью системы осуществляется по протоколу HTTP.

Подсистема удалённого администрирования (CoreLocalClient) выступает прослойкой между frontend-приложением и плагинами ядра системы. На основе HTTP-запросов формируются обращения к плагинам через Apache Thrift, результаты возвращаются в виде HTTP-ответов. Подсистема реализована как фоновый сервис Linux и построена на фреймворке ASP.NET Core [16].

Клиент локального администрирования предоставляет администратору доступ к функциям системы непосредственно на узле, где она развёрнута. Обращение к плагинам ядра выполняется через Apache Thrift.

Ядро системы (Core) отвечает за загрузку плагинов на этапе запуска и за обработку запросов сервиса резервного копирования платформы по сохранению и восстановлению конфигурации. Реализовано в виде фонового сервиса операционной системы Linux, запускаемого через systemd. Использовалась платформа .NET 6 и язык программирования C#.

Плагинная подсистема представляет собой набор подключаемых плагинов, отвечающих за управление настройками отдельных подсистем системы. Каждый плагин принимает запросы администратора через интерфейс Apache Thrift и возвращает результаты обработки.

Часть плагинов дополнительно выступает клиентом целевых служб и передаёт им настройки через тот же протокол. Хранение настроек реализовано во встраиваемой базе данных SQLite, доступ к данным обеспечивается ORM Entity Framework Core [17].

Целевые службы – набор служб, настройками которых управляет система. Развёртываются на одном узле с ядром. Взаимодействие плагинов с целевыми службами выполняется через Apache Thrift.

Использование Apache Thrift в качестве единого протокола внутреннего взаимодействия обеспечивает кроссплатформенность и однородный формат сервисных контрактов, что упрощает интеграцию независимо разрабатываемых компонентов.

Распределение подсистем по узлам развёртывания иллюстрирует рисунок 22.

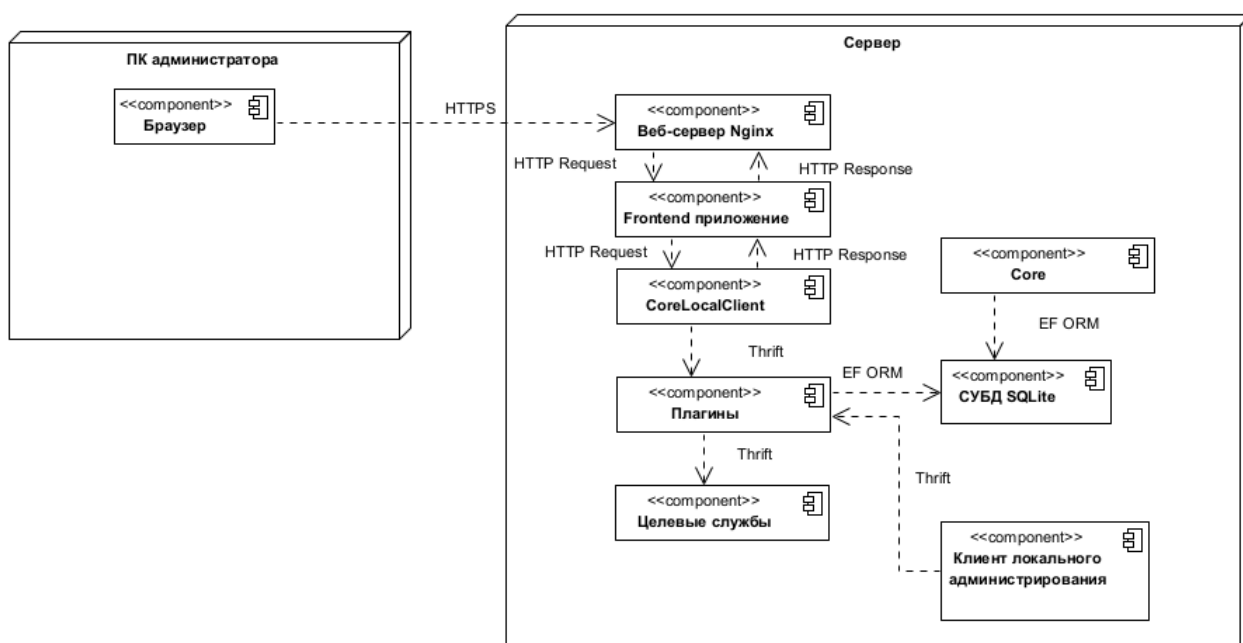


Рисунок 22 – Диаграмма развёртывания системы

Все подсистемы серверной части системы развёртываются на одном узле. Внешним по отношению к серверу узлом является только рабочее место администратора, обращающееся к системе по защищённому протоколу.

4.1 Динамическая загрузка плагинов

Плагинная архитектура системы предполагает динамическое подключение функциональных модулей на этапе запуска без необходимости пересборки ядра. Чтобы это обеспечить, в составе ядра реализован механизм динамической загрузки плагинов, отвечающий за обнаружение, загрузку и инициализацию плагинов из заданной директории.

Каждый плагин представляет собой сборку с расширением `.dll`, размещённую в отдельной поддиректории. Сборка содержит реализацию интерфейса `IPlugin`, а также все зависимости плагина – сторонние библиотеки, на которые он опирается. Интерфейс `IPlugin` задаёт минимальный контракт, который должен реализовать каждый плагин: имя плагина и метод регистрации внутренних сервисов в общем контейнере зависимостей ядра. Намеренно сокращённый состав интерфейса позволяет переносить специфическую для плагина логику внутрь самих регистрируемых сервисов и сохранять контракт в минимальном виде.

Размещение сборок плагинов в общей директории обеспечивается на этапе сборки проекта. В файле проекта каждого плагина задано правило, согласно которому при публикации выходные файлы сборки помещаются в поддиректорию `Plugins`. В результате после публикации всего решения в выходной директории формируется директория `Plugins`, содержащая по поддиректории на каждый плагин. Именно эта директория используется ядром в качестве источника плагинов при запуске.

Динамическая загрузка плагинов реализована в классе `PluginManager`. На этапе запуска он сканирует директорию с плагинами, заданную в конфигурации, и для каждой найденной сборки определяет, требуется ли её подключение. Подключение выполняется в два этапа: сначала сборка загружается в специально созданный контекст загрузки, затем из неё извлекается реализация интерфейса `IPlugin`, и метод регистрации сервисов плагина вызывается с передачей общего контейнера зависимостей.

Изоляция плагинов друг от друга обеспечивается за счёт использования механизма `AssemblyLoadContext` [18]. Для каждого плагина создаётся отдельный контекст загрузки на основе класса `PluginLoadContext`, наследуемого от `AssemblyLoadContext`. В контексте загрузки определена логика разрешения зависимостей сборки через `AssemblyDependencyResolver`. Такое решение предотвращает конфликты версий между плагинами и позволяет опираться на различные версии общих библиотек без взаимного влияния. Контексты создаются с флагом `IsCollectible`, что в перспективе оставляет возможность выгрузки плагина без перезапуска ядра.

Конфигурация загрузки хранится в файле `appsettings.json` ядра и содержит путь к директории с плагинами, а также флаг `ForceLoadAllPlugins`, определяющий режим загрузки. При установленном флаге загружаются все обнаруженные сборки; в обычном режиме загружаются только те плагины, которые перечислены в конфигурации явно. Помимо общей конфигурации, каждый плагин может иметь собственный файл `appsettings.json`, который при подключении плагина подмешивается в общий конфигурационный поставщик ядра. Это позволяет плагину иметь свою область настроек и при этом обращаться к ним через общую инфраструктуру конфигурирования платформы .NET.

Фрагмент кода, выполняющего подключение плагина, приведён в листинге 1.

Листинг 1 – Подключение плагина в составе ядра системы

```
public IPlugin LoadPlugin(string pluginPath)
{
    var loadContext = new PluginLoadContext(pluginPath);
    var assembly = loadContext.LoadFromAssemblyPath(pluginPath);
    var pluginType = assembly.GetType()
        .First(t => typeof(IPlugin).IsAssignableFrom(t));

    var plugin = (IPlugin)Activator.CreateInstance(pluginType!);
    _loadedPlugins[plugin.Name] = (loadContext, plugin);
    return plugin;
}

public static void LoadPlugins(this IServiceCollection services,
    IConfiguration configuration)
```

```

{
    using var pluginManager = new PluginManager(configuration);
    var plugins = pluginManager.LoadConfiguredPlugins();
    foreach (var plugin in plugins)
        plugin.ConfigureServices(services);
}

```

В приведённом фрагменте показана последовательность подключения плагина: создание изолированного контекста загрузки, загрузка сборки из файла, поиск в ней реализации интерфейса `IPlugin`, создание экземпляра плагина и его сохранение в словаре загруженных плагинов. Регистрация сервисов всех загруженных плагинов в общем контейнере зависимостей ядра выполняется через метод-расширение, который вызывается на этапе настройки сервисов приложения.

Такой подход к организации плагиновой подсистемы обеспечивает расширяемость системы без модификации кода ядра, изоляцию плагинов друг от друга на уровне загруженных сборок и единообразие настройки плагинов через стандартные механизмы конфигурирования платформы [19].

4.2 Резервное копирование и версионирование формата конфигурации

Подсистема резервного копирования платформы периодически запрашивает у системы текущее состояние её настроек для включения в резервную копию, а при необходимости восстановления передаёт ранее сохранённое состояние обратно в систему. Для обеспечения такого взаимодействия ядро системы предоставляет внешний интерфейс резервного копирования, по которому сервис резервного копирования может получать и обновлять конфигурацию.

Конфигурация системы складывается из настроек всех подключенных плагинов и общесистемных метаданных, описывающих сохранённый снимок. Состав плагинов определяется на этапе запуска и заранее ядру неизвестен, что исключает централизованное представление всей конфигурации в одной структуре данных. Чтобы решить эту задачу, в системе применяется

агрегирующая схема: ядро собирает конфигурацию по частям, обращаясь к каждому подключённому плагину за его участком настроек, после чего объединяет полученные части в общий документ. При восстановлении выполняется обратное действие: ядро передаёт общий документ каждому плагину, и плагин извлекает из него свой участок настроек для применения.

Серверная часть интерфейса резервного копирования реализована в классе `ConfigurationBackupHandler`. Он принимает запросы на получение конфигурации, её обновление, получение текущей версии и метки времени последнего изменения, а также проверку поддержки переданной версии. Метка времени конфигурации хранится в служебной таблице базы данных ядра.

Сборка конфигурации из участков, предоставляемых плагинами, вынесена в отдельный класс `ConfigurationAggregator`. Для каждого плагина, участвующего в формировании конфигурации, определён интерфейс `IConfigurationSectionProvider`, который реализуется внутри плагина. Реализация интерфейса определяет имя своей секции и обеспечивает её получение и обновление. Класс `ConfigurationAggregator` при сборке конфигурации обходит зарегистрированные реализации этого интерфейса, запрашивает у каждой её участок в формате JSON и помещает результат в общий JSON-документ [20]. При восстановлении общий документ передаётся всем поставщикам секций. Каждый поставщик десериализует из него только те поля, которые объявлены в его модели конфигурации, остальные поля при десериализации игнорируются.

Фрагмент кода агрегатора конфигурации, выполняющего сборку общего документа из участков, приведён в листинге 2.

Листинг 2 – Сборка конфигурации из участков плагинов

```
public string GetServiceConfiguration()
{
    var serviceConfiguration = new JsonObject();

    foreach (var configurationSectionProvider in
        _configurationSectionProviders)
    {
        var rootElement =
            JsonNode.Parse(configurationSectionProvider.GetConfiguration()) as
```

```

JsonObject
    ?? throw new ParseConfigurationException($"При
получении конфигурации секции:
{configurationSectionProvider.SectionName} был получен не
составной объект");

    foreach (var jsonElement in rootElement)
        serviceConfiguration[jsonElement.Key] =
jsonElement.Value is null
            ? JsonValue.Create((object?)null)
            :
JsonValue.Parse(jsonElement.Value.ToJsonString());
}

return serviceConfiguration.ToJsonString();
}

```

В приведённом фрагменте показана сборка общей конфигурации сервиса: агрегатор обходит зарегистрированные поставщики секций, у каждого запрашивает его участок в формате JSON, разбирает его как объект и копирует поля верхнего уровня в общий JSON-документ. После завершения обхода документ сериализуется в строковое представление и возвращается вызывающему коду.

Конфигурация передаётся между системой и сервисом резервного копирования в виде структуры с тремя полями: содержимое в текстовом представлении, версия формата и метка времени. Версия формата необходима для поддержки нескольких поколений структуры конфигурации, что неизбежно при долгосрочной эксплуатации системы. По мере развития плагинов состав и формат хранимых настроек могут меняться, и при восстановлении ранее сохранённой конфигурации требуется привести её к актуальному виду перед применением.

Для решения этой задачи плагин реализует собственный механизм миграции версий формата по типовому паттерну. В рамках паттерна объявляется пара интерфейсов: интерфейс анализатора, описывающий преобразование конфигурации заданной версии в актуальный вид, и интерфейс фабрики, выбирающей соответствующий анализатор по версии из переданной конфигурации. На основе этих интерфейсов в плагине формируется набор анализаторов – по одному на каждую поддерживаемую версию формата, а

фабрика подбирает требуемый при восстановлении. При появлении новой версии формата достаточно добавить новый анализатор и зарегистрировать его в контейнере зависимостей: остальной код плагина остаётся без изменений.

Фрагмент кода фабрики анализаторов и одной из её реализаций из плагина управления настройками подключения к базе данных приведён в листинге 3.

Листинг 3 – Фабрика анализаторов конфигурации и анализатор предшествующей версии

```
internal class DbConfigurationParserFactory :
IDbConfigurationParserFactory
{
    public
    DbConfigurationParserFactory(IEnumerable<IDbConfigurationBackupPar
ser> configurationParsers)
    {
        _parsers = configurationParsers;
    }

    public ModuleConfiguration?
    ParseToCurrentConfiguration(Version version, string configuration)
    {
        var parser = _parsers.FirstOrDefault(p =>
p.IsSupportedVersion(version));

        if (parser == null)
        {
            throw new NotSupportedConfigurationVersion($"Не
зарегистрирован ни один анализатор, " +
                $"поддерживающий конфигурацию версии
{version.Major}.{version.Minor}");
        }

        return parser.Parse(configuration);
    }

    private readonly IEnumerable<IDbConfigurationBackupParser>
    _parsers;
}

internal sealed class DbConfigurationBackupParser :
IDbConfigurationBackupParser
{
    public bool IsSupportedVersion(Version version) =>
        _supportedVersion.Major == version.Major &&
        _supportedVersion.Minor == version.Minor;

    public ModuleConfiguration? Parse(string configuration)
```

```

    {
        var config =
JsonSerializer.Deserialize<Models.V1_0.ModuleConfiguration>(config
uration);
        return config?.ConvertToCurrentVersion();
    }

    private readonly Version _supportedVersion = new() { Major =
1, Minor = 0 };
}

```

В коде показана фабрика анализаторов и один из таких анализаторов – анализатор предшествующей версии формата конфигурации. Список зарегистрированных анализаторов поступает в фабрику средствами контейнера зависимостей.

При преобразовании конфигурации фабрика обходит список и выбирает анализатор, заявивший о поддержке переданной версии; при отсутствии подходящего анализатора выбрасывается исключение о неподдерживаемой версии. Анализатор десериализует переданную строку во внутреннюю модель своей версии и преобразует её к актуальной модели конфигурации через метод преобразования модели. Анализатор актуальной версии устроен аналогично, но без шага преобразования модели.

Чтобы восстановление конфигурации не оставляло систему в промежуточном состоянии при возникновении ошибок, работа каждого плагина с собственным хранилищем выполняется в рамках транзакции на уровне хранилища, а групповая блокировка плагина предотвращает параллельные изменения настроек в процессе восстановления.

При возникновении ошибки в ходе восстановления транзакция отменяется, и хранилище плагина возвращается к состоянию, предшествовавшему операции, а по завершении восстановления групповая блокировка снимается. Тем самым обеспечивается атомарность восстановления: конфигурация либо изменяется целиком, либо остаётся в исходном виде.

Общий ход процедуры восстановления конфигурации в рамках диаграммы деятельности иллюстрирует рисунок 23.

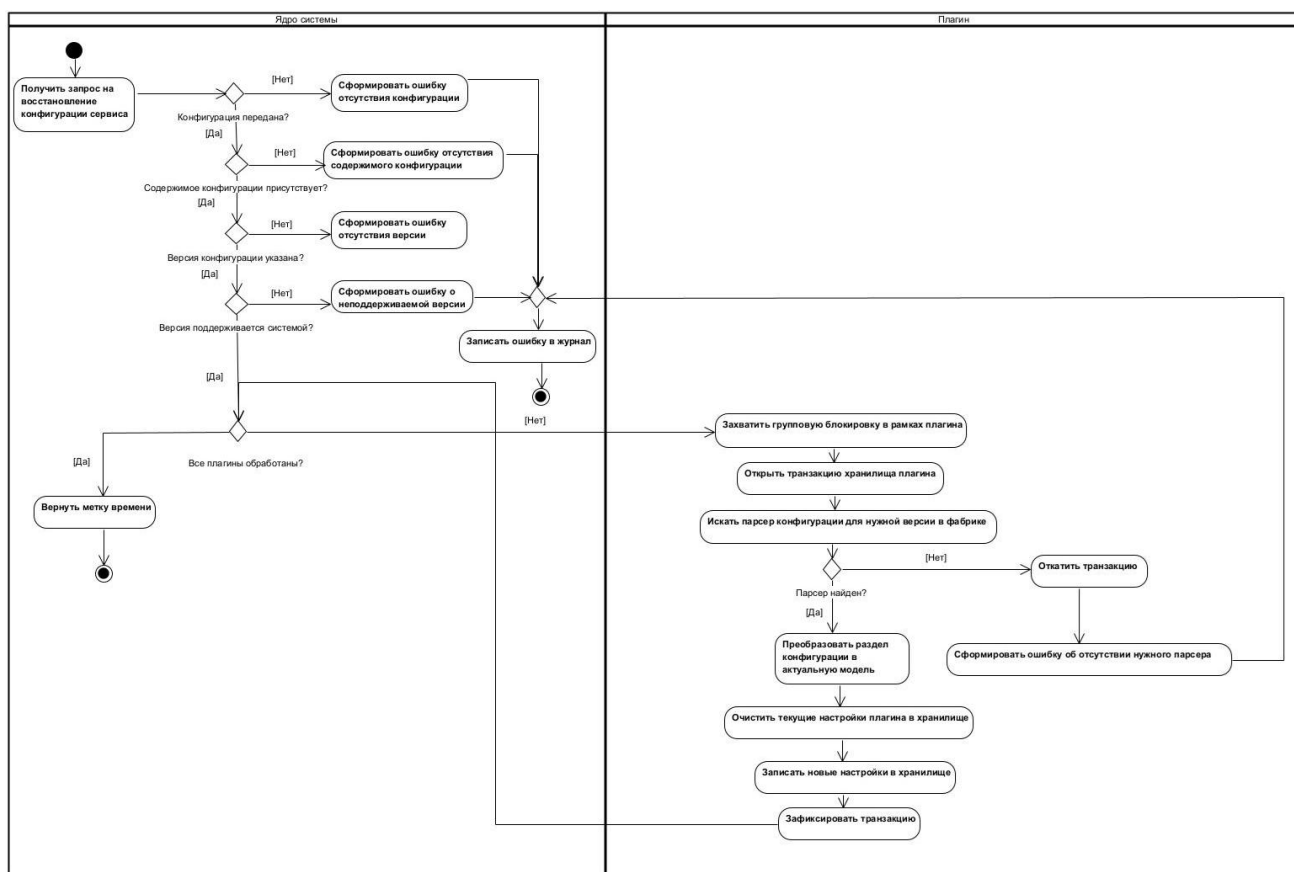


Рисунок 23 – Диаграмма деятельности восстановления конфигурации

Описанный подход позволяет системе сохранять и восстанавливать состояние без зависимости от конкретного состава плагинов, поддерживать развитие формата конфигурации без потери совместимости с ранее сохранёнными снимками и согласованно применять восстановленные настройки за счёт транзакционной обработки и блокировок.

4.3 Иерархические блокировки

Параллельная работа нескольких администраторов с одной и той же подсистемой допустима только при условии, что операции не приводят к рассогласованию данных в хранилище и применённых настроек на целевой службе. Стандартного механизма потоковой синхронизации недостаточно: для одних операций нужно ограничить параллельную модификацию одного и того же ресурса, для других – временно запретить любые изменения внутри плагина.

В системе для решения этой задачи реализованы иерархические блокировки с двумя уровнями детализации – на отдельный ресурс и на группу ресурсов.

Группой считается совокупность ресурсов плагина, объединенных общим префиксом ключа. В качестве префикса обычно выступает имя плагина или его подсистемы, а постфикс ключа обозначает конкретный ресурс внутри группы – например, экземпляр настройки, идентификатор сущности или режим работы. Захват блокировки на один ключ блокирует параллельные операции только над этим ключом, не препятствуя одновременной работе с другими ключами той же группы. Захват блокировки на всю группу блокирует все ключи внутри неё до освобождения групповой блокировки.

Связь между двумя уровнями блокировок реализована через механизм «читатели – писатель». Захват блокировки на отдельный ключ соответствует захвату блокировки чтения у объекта группы, захват блокировки на группу – захвату блокировки записи у того же объекта. Несколько читателей могут работать параллельно, что соответствует параллельному захвату разных ключей. Писатель эксклюзивен и не допускает ни параллельных писателей, ни параллельных читателей, что соответствует эксклюзивности групповой блокировки. Такая модель обеспечивает корректное взаимодействие двух уровней без отдельной логики координации между ними.

Точкой доступа к блокировкам выступает фабрика `ISemaphoreFactory`, предоставляющая два метода: получение блокировки на ключ и захват блокировки на группу. Реализация фабрики – класс `AsyncSemaphoreFactory` – хранит две внутренние коллекции: семафоры по ключам и объекты блокировок «читатели – писатель» по группам. Группа ключа определяется автоматически – как часть строкового представления ключа до символа-разделителя; если разделитель в ключе отсутствует, группа совпадает с самим ключом. При обращении к ещё не использованному ключу или группе соответствующий объект создаётся и помещается в коллекцию; при повторных обращениях возвращается ранее созданный.

Реализация блокировки «читатели – писатель» в классе `AsyncReaderWriterLock` опирается на два семафора и счётчик активных читателей. Один семафор защищает счётчик от гонки при одновременных захватах и освобождениях со стороны нескольких читателей. Второй семафор служит сигналом писателю: его захват означает, что писатель активен и новые читатели должны ждать. Первый захват блокировки чтения занимает этот семафор, чтобы перекрыть запись; последующие захваты увеличивают счётчик. При освобождении блокировки чтения счётчик уменьшается, и при достижении нуля семафор освобождается, открывая доступ писателю или следующим читателям. Захват блокировки записи занимает семафор напрямую и удерживает его до освобождения.

Фрагмент кода фабрики блокировок, реализующего получение блокировки на ключ и на группу, приведён в листинге 4.

Листинг 4 – Получение блокировок на ключ и на группу

```
public async Task<IDisposable> Get(string key)
{
    var prefix = GetPrefix(key);
    var group = _groups.GetOrAdd(prefix, _ => new
AsyncReaderWriterLock());
    var keyLock = _keyLocks.GetOrAdd(key, _ => new
SemaphoreSlim(1, 1));

    var readLock = await group.ReaderLock();
    await keyLock.WaitAsync();

    return new KeyReleaser(keyLock, readLock);
}

public async Task<IDisposable> LockGroup(string prefix)
{
    var group = _groups.GetOrAdd(prefix, _ => new
AsyncReaderWriterLock());
    var writeLock = await group.WriterLock();

    return new GroupReleaser(writeLock);
}
```

В приведённом фрагменте показаны два метода фабрики. Получение блокировки на ключ выполняется в два шага: захват блокировки чтения у объекта группы, к которой относится ключ, и затем захват семафора самого ключа. Захват

блокировки на группу выполняется в один шаг – через блокировку записи у объекта группы. Возвращаемые объекты освобождают захваченные блокировки в порядке, обратном их захвату.

Сценарии применения блокировок различаются по объёму операции. Захват блокировки на ключ используется в операциях, изменяющих один конкретный ресурс плагина – обновление настроек подключения к базе данных, изменение конфигурации экземпляра сущности. Захват блокировки на группу используется при операциях, охватывающих все ресурсы плагина целиком, например при восстановлении конфигурации плагина из резервной копии, когда содержимое хранилища плагина изменяется полностью.

Описанный подход обеспечивает корректную параллельную работу с независимыми ресурсами плагина без избыточной взаимной блокировки и при этом гарантирует эксклюзивность операций, затрагивающих плагины целиком.

4.4 Защита фонового применения настроек от устойчивых сбоев

При управлении настройками подсистемы возможны две ситуации: администратор вносит изменения вручную, либо текущее состояние целевой службы рассогласовывается с сохранёнными настройками без участия администратора (например, после перезапуска службы или восстановления резервной копии). В первой ситуации применение настроек инициируется обработчиком запроса администратора, во второй – система должна обнаружить рассогласование и привести целевую службу в соответствие с сохранёнными настройками. Для решения обеих задач в системе реализован механизм применения настроек, работающий по двум каналам через общий код применения и поддерживаемый фоновым сервисом.

Применение настроек инкапсулировано в установщике настроек плагина. Установщик предоставляет метод применения с одним необязательным параметром – запросом администратора. При вызове с запросом установщик обновляет настройки в собственном хранилище, после чего применяет их на целевую службу. При вызове без запроса установщик обращается к хранилищу

за текущими сохранёнными настройками и применяет их. В обоих случаях применение на целевую службу выполняется единым методом, что обеспечивает идентичное поведение для ручного и фоновых каналов.

Фоновый канал применения реализован как отдельный фоновый сервис плагина, который в цикле обращается к методу применения без запроса. Между итерациями цикла выдерживается пауза, длительность которой управляется отдельным объектом, отвечающим за расчёт паузы. При возникновении ошибки во время применения настроек пауза увеличивается на постоянную величину, пропорциональную числу подряд неуспешных попыток. Чем больше попыток подряд завершилось ошибкой, тем реже фон будет повторять обращение. При успешной итерации счётчик попыток сбрасывается, и пауза восстанавливается.

Увеличение паузы предотвращает ситуацию, когда фон в условиях устойчивой проблемы, например, недоступная целевая служба, совершает обращения слишком часто, нагружая журналы и потребляя ресурсы. Однако одного только увеличения недостаточно: при длительной неработоспособности целевой службы фон будет продолжать попытки бесконечно. Для защиты от такого сценария задано максимальное число попыток подряд. При достижении этого числа фоновое применение переводится в режим ожидания: цикл продолжает работать, но обращения к целевой службе пропускаются до тех пор, пока не поступит внешний сигнал о возобновлении.

Сигналом возобновления выступает событие об изменении настроек, поднимаемое установщиком при ручном вмешательстве администратора. Фоновый сервис подписан на это событие и при срабатывании сбрасывает счётчик попыток, после чего фон возвращается в обычный режим работы. Тем самым после ручного вмешательства администратора система автоматически возвращается к попыткам приведения целевой службы в соответствие с сохранёнными настройками.

Фрагмент кода фонового сервиса плагина, обеспечивающего периодическое применение настроек с защитой от устойчивых сбоев, приведён в листинге 5.

Листинг 5 – Фоновый сервис применения настроек

```
protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
{
    while (!stoppingToken.IsCancellationRequested)
    {
        try
        {
            if (_initializationOptions.Period == TimeSpan.Zero
|| _initializationOptions.Disable)
                continue;

            var serviceIsActive = await
_serviceManager.CheckServiceState(ServiceName);

            if (!serviceIsActive)
            {
                await Task.Delay(5000, stoppingToken);
                continue;
            }

            await _setuper.Setup(null);
            _initializationOptions.ResetPeriod();
        }
        catch (DocProcessingException ex)
        {
            _initializationOptions.UpPeriod();

            const string Message = "При попытке обновления
конфигурации службы обработки электронных документов произошла
ошибка";
            _logger.LogError(ex, Message);
        }

        await Task.Delay(_initializationOptions.Period,
stoppingToken);
    }
}

private void OnSetupRequest(object? sender, EventArgs e) =>
_initializationOptions.ResetPeriod();
```

Описанный подход обеспечивает применение настроек на целевую службу единообразно через ручной и фоновый каналы, защищает фоновый канал от каскадного нарастания нагрузки и журналируемых ошибок при устойчивых сбоях целевой службы. Возврат фона в нормальный режим работы автоматически инициируется ручным вмешательством администратора, что обеспечивает контроль над поведением системы со стороны эксплуатации.

5 Тестирование программного продукта

5.1 Функциональное тестирование методом белого ящика

Для проверки корректности разработанных программных средств применялся метод тестирования «белого ящика» [21]. Его выбор обусловлен тем, что исходный код компонентов системы доступен на этапе тестирования, а значит, поддаётся проверке не только результат работы программы, но и её внутренняя логика – последовательность выполнения операторов, прохождение ветвлений и обработка условий. Дополнительным преимуществом данного метода является возможность количественной оценки полноты тестирования с помощью критериев покрытия операторов, решений и условий.

В качестве объекта тестирования выбран метод фоновой обновления конфигурации службы обработки электронных документов. Данный метод реализует длительно выполняющуюся фоновую операцию, в ходе которой производится ожидание запуска приложения, периодический опрос обслуживаемых сервисов, применение конфигурации и обработка возникающих при этом ошибок. Выбор именно этого метода обусловлен тем, что он содержит несколько уровней ветвления, цикл и обработку исключительных ситуаций, что делает его подходящим для демонстрации всех рассматриваемых критериев покрытия. Структура метода приведена в листинге 6.

Листинг 6 – Структура метода фоновой обновления конфигурации

```
protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
{
    if (!await AppStatusTracker.WaitForAppStartup(_lifetime,
stoppingToken))
        return;

    while (!stoppingToken.IsCancellationRequested)
    {
        await Parallel.ForEachAsync(_services, stoppingToken,
async (service, token) =>
        {
            var options = _initializationOptions[service.Key];
```

```

        if (options.Period == TimeSpan.Zero ||
options.Disable)
            return;

        try
        {
            if (!await
_serviceManager.CheckServiceState(service.Value))
                return;

            await _setuper.Setup();
            options.ResetPeriod();
        }
        catch (DocProcessingException)
        {
            options.UpPeriod();
        }

        await Task.Delay(options.Period, token);
    });
}
}

```

Рассматриваемый метод содержит четыре решения, определяющих ход его выполнения. Первое решение проверяет готовность приложения к работе и определяет, следует ли запускать фоновую операцию. Второе является условием цикла и определяет продолжение периодического обновления конфигурации. Третье решение проверяет, не отключено ли обновление настроек для конкретного сервиса, и является составным: оно объединяет проверку нулевого периода обновления и проверку признака отключения. Четвёртое решение проверяет активность службы перед применением настроек. Перечисленные решения, в том числе составное третье, служат основой для формирования тест-кейсов, обеспечивающих покрытие решений и условий рассматриваемого метода.

5.1.1 Покрытие операторов

В соответствии с критерием покрытия операторов необходимо составить такой набор тестов, при котором каждый оператор программы будет выполнен хотя бы один раз. Перечень операторов метода и число их вхождений приведён в таблице 12.

Таблица 12 – Операторы метода фонового обновления конфигурации

Обозначение	Оператор (кодový фрагмент)	Число вхождений
O1	if (!await AppStatusTracker.WaitForAppStartup(_lifetime, stoppingToken))	1
O2	return;	1
O3	while (!stoppingToken.IsCancellationRequested)	1
O4	await Parallel.ForEachAsync(_services, stoppingToken, async (...) => {...})	1
O5	var options = _initializationOptions[service.Key];	1
O6	if (options.Period == TimeSpan.Zero options.Disable)	1
O7	return;	1
O8	if (!await _serviceManager.CheckServiceState(service.Value))	1
O9	return;	1
O10	await setuper.Setup();	1
O11	options.ResetPeriod();	1
O12	options.UpPeriod();	1
O13	await Task.Delay(options.Period, token);	1

Диаграмма активности метода с указанием операторов приведена на рисунке 24.

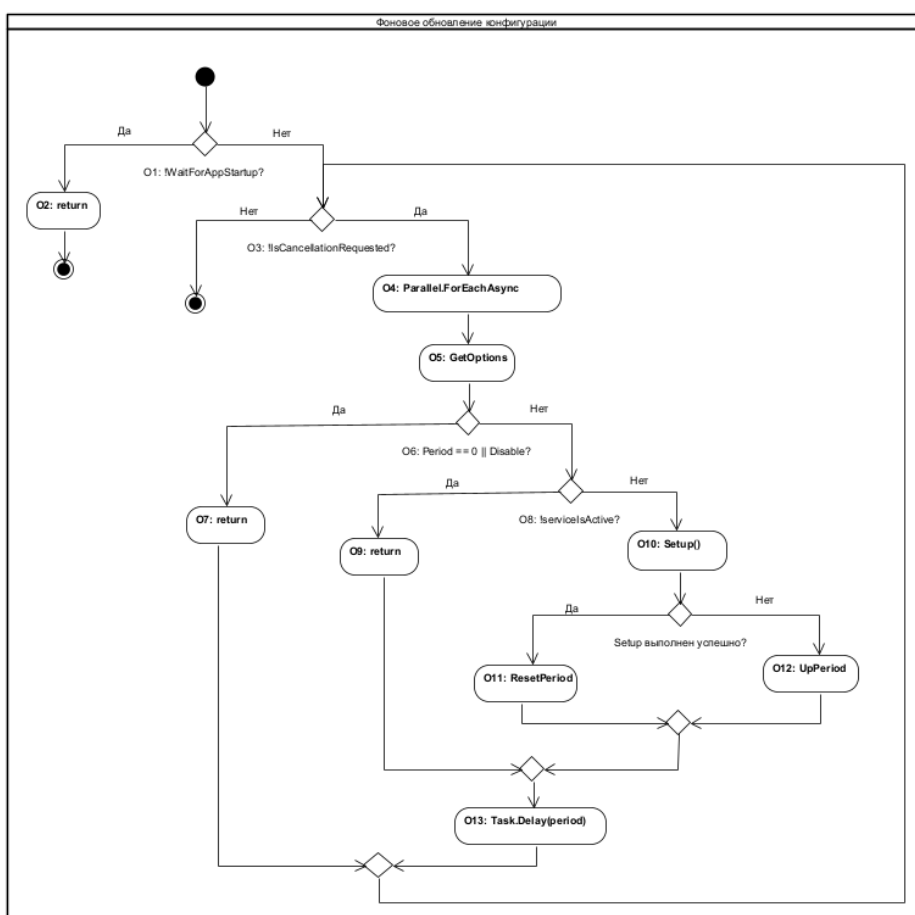


Рисунок 24 – Диаграмма активности метода фонового обновления конфигурации

Для выполнения каждого оператора метода хотя бы один раз тестовые наборы подбирались таким образом, чтобы в совокупности они проходили все ветви метода. Сформированные тестовые наборы приведены в таблице 13.

Таблица 13 – Тест-кейсы для покрытия операторов

№	Операторы	Входные данные	Ожидаемый результат	Фактический результат
1	O1, O2	Приложение не запущено	Фоновая операция не запускается; сообщения о начале обновления в журнал не выводятся	Соответствует ожидаемым результатам (рисунок В.1)
2	O1, O3	Приложение запущено, операция отменена до входа в цикл	В журнал выводится сообщение о запуске фоновой операции; сообщения о транзакциях обновления отсутствуют	Соответствует ожидаемым результатам (рисунок В.2)
3	O1, O3-O7	Период обновления равен нулю либо обновление отключено	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено	Соответствует ожидаемым результатам (рисунок В.3)
4	O1, O3-O6, O8, O9	Служба обработки электронных документов неактивна	В журнал выводится предупреждение о том, что обновление прервано и ожидается запуск службы	Соответствует ожидаемым результатам (рисунок В.4)
5	O1, O3-O6, O8, O10, O11, O13	Служба активна, настройки применены успешно	В журнал выводится сообщение о завершении транзакции обновления и времени следующего запуска	Соответствует ожидаемым результатам (рисунок В.5)
6	O1, O3-O6, O8, O10, O12, O13	При применении настроек возникает ошибка	В журнал выводится сообщение об ошибке при обновлении конфигурации службы	Соответствует ожидаемым результатам (рисунок В.6)

Для выполнения каждого оператора метода хотя бы один раз потребовалось сформировать шесть тестовых наборов. Приведённый набор тестов охватывает как сценарии штатного выполнения фоновой операции, так и случаи её досрочного завершения, пропуска отключённого сервиса, недоступности службы и возникновения ошибки при применении настроек. В результате выполнения тестов каждый оператор метода был задействован, что подтверждает достижимость всех участков кода.

5.1.2 Покрытие решений

В соответствии с критерием покрытия решений, называемого также покрытием ветвей, необходимо составить такой набор тестов, при котором каждое решение программы примет как истинное, так и ложное значение. Тем самым обеспечивается проверка обеих ветвей каждой условной конструкции метода. Сформированные тестовые наборы приведены в таблице 14.

Таблица 14 – Тест-кейсы для покрытия решений

№	Решения	Входные данные	Ожидаемый результат	Фактический результат
1	O1 = да	Приложение не запущено	Фоновая операция не запускается; сообщения о начале обновления в журнал не выводятся	Соответствует ожидаемым результатам (рисунок В.1)
2	O1 = нет, O3 = нет	Приложение запущено, операция отменена до входа в цикл	В журнал выводится сообщение о запуске фоновой операции; сообщения о транзакциях обновления отсутствуют	Соответствует ожидаемым результатам (рисунок В.2)
3	O3 = да, O6 = да	Период обновления равен нулю либо обновление отключено	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено	Соответствует ожидаемым результатам (рисунок В.3)
4	O6 = нет, O8 = да	Служба обработки электронных документов неактивна	В журнал выводится предупреждение о том, что обновление прервано и ожидается запуск службы	Соответствует ожидаемым результатам (рисунок В.4)
5	O8 = нет	Служба активна, настройки применены успешно	В журнал выводится сообщение о завершении транзакции обновления и времени следующего запуска	Соответствует ожидаемым результатам (рисунок В.5)

Для покрытия всех решений метода потребовалось сформировать пять тестовых наборов, в которых каждое решение принимает оба возможных значения. Поскольку покрытие решений гарантирует выполнение каждого оператора, приведённый набор тестов одновременно подтверждает и достижение покрытия операторов, рассмотренного ранее.

5.1.3 Покрытие условий

Для проверки покрытия условий подбирается такой набор тестов, который позволяет получить все возможные результаты для каждого простого условия, входящего в состав решений программы, как минимум по одному разу. При этом одного только покрытия условий недостаточно, поскольку оно не гарантирует выполнение всех операторов метода. Поэтому при построении набора тестов дополнительно учитывается требование, чтобы каждый оператор программы был выполнен хотя бы один раз. Среди решений рассматриваемого метода составным является только третье решение, объединяющее два простых условия – проверку нулевого периода обновления (Об.1) и проверку признака отключения (Об.2). Остальные решения содержат по одному простому условию. Сформированные тестовые наборы приведены в таблице 15.

Таблица 15 – Тест-кейсы для покрытия условий

№	Условия	Входные данные	Ожидаемый результат	Фактический результат
1	O1 = ложь	Приложение не запущено	Фоновая операция не запускается; сообщения о начале обновления в журнал не выводятся	Соответствует ожидаемым результатам (рисунок В.1)
2	O1 = истина, O3 = истина	Приложение запущено, операция отменена	В журнал выводится сообщение о запуске фоновой операции; сообщения о транзакциях обновления отсутствуют	Соответствует ожидаемым результатам (рисунок В.2)
3	Об.1 = истина, Об.2 = ложь	Период равен нулю, признак отключения снят	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено	Соответствует ожидаемым результатам (рисунок В.3)
4	Об.1 = ложь, Об.2 = истина	Период не равен нулю, обновление отключено	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено временно	Соответствует ожидаемым результатам (рисунок В.7)

Продолжение таблицы 15

№	Условия	Входные данные	Ожидаемый результат	Фактический результат
5	O8 = ложь	Служба обработки электронных документов неактивна	В журнал выводится предупреждение о том, что обновление прервано и ожидается запуск службы	Соответствует ожидаемым результатам (рисунок В.4)
6	O8 = истина	Служба обработки электронных документов активна	Предупреждение об ожидании запуска службы в журнал не выводится, выполняется применение настроек	Соответствует ожидаемым результатам (рисунок В.5)

Для покрытия всех условий метода потребовалось сформировать шесть тестовых наборов. Каждое простое условие, участвующее в логике метода, проверено как при истинном, так и при ложном значении, при этом составное условие разложено на два самостоятельных условия. Сформированный набор тестов дополнительно обеспечивает выполнение каждого оператора метода хотя бы один раз, что соответствует требованию, дополняющему данный критерий.

5.1.4 Покрытие решений и условий

В соответствии с этим критерием необходимо составить тесты так, чтобы результаты каждого условия выполнялись хотя бы один раз, результаты каждого решения так же выполнялись хотя бы один раз, и каждый оператор должен быть выполнен хотя бы один раз. Данный критерий можно представить как покрытие маршрутов – каждый тестовый набор соответствует отдельному пути выполнения метода. Сформированные тестовые наборы приведены в таблице 16.

Таблица 16 – Тест-кейсы для покрытия решений и условий

№	Маршрут выполнения	Входные данные	Ожидаемый результат	Фактический результат
1	O1(да) -> O2	Приложение не запущено	Метод завершается; сообщения о начале обновления в журнал не выводятся	Соответствует ожидаемым результатам (рисунок В.1)

Продолжение таблицы 16

2	O1(нет)->O3(нет)	Приложение запущено, операция отменена до входа в цикл	В журнал выводится сообщение о запуске фоновой операции; сообщения о транзакциях обновления отсутствуют	Соответствует ожидаемым результатам (рисунок В.2)
3	O1(нет)->O3(да)->O4->O5->O6.1(да),O6.2(нет)->O7	Период равен нулю, признак отключения снят	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено	Соответствует ожидаемым результатам (рисунок В.3)
4	O1(нет)->O3(да)->O4->O5->O6.1(нет),O6.2(да)->O7	Период не равен нулю, обновление отключено	В журнал выводится сообщение о том, что фоновое обновление настроек обработки электронных документов отключено временно	Соответствует ожидаемым результатам (рисунок В.7)
5	O1(нет)->O3(да)->O4->O5->O6.1(нет),O6.2(нет)->O8(да)->O9	Служба обработки электронных документов неактивна	В журнал выводится предупреждение о том, что обновление прервано и ожидается запуск службы	Соответствует ожидаемым результатам (рисунок В.4)
6	O1(нет)->O3(да)->O4->O5->O6.1(нет),O6.2(нет)->O8(нет)->O10(успех)->O11->O13	Служба активна, настройки применены успешно	В журнал выводится сообщение о завершении транзакции обновления и времени следующего запуска	Соответствует ожидаемым результатам (рисунок В.5)
7	O1(нет)->O3(да)->O4->O5->O6.1(нет),O6.2(нет)->O8(нет)->O10(ошибка)->O12->O13	При применении настроек возникает ошибка обработки электронных документов	В журнал выводится сообщение об ошибке при обновлении конфигурации службы	Соответствует ожидаемым результатам (рисунок В.6)

Для достижения покрытия решений и условий метода потребовалось сформировать семь тестовых наборов. В рамках данного набора каждое простое условие приняло оба значения, каждое решение пройдено как по истинной, так и по ложной ветви, а каждый оператор выполнен хотя бы один раз. В частности, для составного решения проверены оба значения входящих в него условий, а также оба результата решения в целом, что подтверждает отсутствие непокрытых сочетаний.

В ходе тестирования метода фоновое обновление конфигурации методом белого ящика последовательно применены четыре критерия покрытия: покрытие

операторов, покрытие решений, покрытие условий, а также покрытие решений и условий. Для каждого из критериев сформирован отдельный набор тестов, и все тестовые наборы были выполнены успешно. Это подтверждает корректность реализации логики фонового обновления конфигурации службы обработки электронных документов как при штатном выполнении операции, так и при возникновении исключительных ситуаций.

5.2 Модульное тестирование

Для проверки корректности отдельных программных модулей системы выполнялось модульное тестирование. В отличие от тестирования методом белого ящика, рассмотренного ранее, модульное тестирование направлено на проверку поведения отдельных компонентов системы в изоляции от внешних зависимостей.

В качестве инструмента модульного тестирования использовался фреймворк xUnit, предназначенный для разработки тестов в среде .NET [22]. Его выбор обусловлен удобной организацией тестовых сценариев, поддержкой параметризованных проверок и возможностью быстрого выполнения тестов в составе проекта. Для изоляции внешних зависимостей применялась библиотека Moq: обращения к провайдерам, хранилищам и сервисам заменялись объектами-заглушками, что позволяло проверять поведение модуля независимо от реального окружения [23].

Модульные тесты были разработаны для основных модулей системы – плагинов управления настройками служб и контроллеров подсистемы удалённого администрирования. Тестовые сценарии охватывают как успешное выполнение операций, так и ситуации с ошибками, отсутствием данных и некорректными запросами, что позволяет проверить поведение модулей в различных условиях. Исходный код основных модульных тестов приведён в приложении Б. Результаты выполнения модульных тестов приведены на рисунке 25.

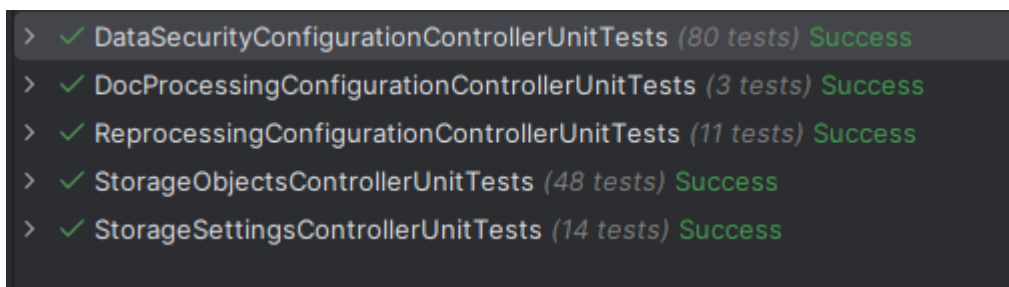


Рисунок 25 – Результаты модульного тестирования

По результатам модульного тестирования подтверждена корректность работы основных плагинов системы в изолированной среде. Разработанные тестовые сценарии охватывают как штатные варианты использования, так и типовые нештатные: отсутствие данных, некорректные входные параметры, неинициализированные зависимости и внутренние ошибки. Это подтверждает надёжность реализованных модулей и их готовность к интеграции в составе системы.

5.3 Интеграционное тестирование

Помимо проверки отдельных модулей в изоляции, выполнялось интеграционное тестирование, цель которого – убедиться в корректности совместной работы компонентов системы, при их взаимодействии через реальные интерфейсы. В отличие от модульных тестов, где зависимости заменяются заглушками непосредственно внутри проверяемого класса, интеграционные тесты обращаются к сервисам по их транспортным интерфейсам и проверяют сквозные сценарии управления конфигурацией.

Тестирование выполняется в специальном окружении разработки, в котором целевые сервисы замещаются их программными имитациями, воспроизводящими поведение реальных сервисов на стороне обработки запросов. Это позволяет проверить полный путь прохождения операции – от клиента управления до обработки на стороне сервиса – без необходимости развёртывания реального целевого окружения. Тесты построены по единому принципу: создаётся клиент соответствующего сервиса, выполняется операция

изменения настроек, после чего настройки считываются обратно и сравниваются с ожидаемыми, что подтверждает их фактическое применение и сохранение.

Тестированием охвачены сценарии управления настройками службы формирования штампов времени, подключений к базе данных, службы обработки электронных документов, аудита, службы повторной обработки информации и службы защиты данных, включая установку поставщиков и профилей для различных режимов работы, а также сброс и удаление настроек. Тесты выполняются в заданном порядке, что позволяет воспроизводить последовательность операций, близкую к реальной эксплуатации.

Запуск интеграционных тестов производится в изолированном окружении на основе Docker-контейнера, что обеспечивает воспроизводимость условий выполнения и независимость от настроек локальной среды [24]. На основе подготовленного Dockerfile собирается образ, содержащий все зависимости проекта и тестовое окружение, после чего из образа запускается контейнер. Внутри контейнер выполняется запуск набора интеграционных тестов средствами платформы .NET, а результаты их прохождения выводятся в консоль. Исходный код интеграционных тестов приведён в приложении Б. Результаты их выполнения приведены на рисунке 26.

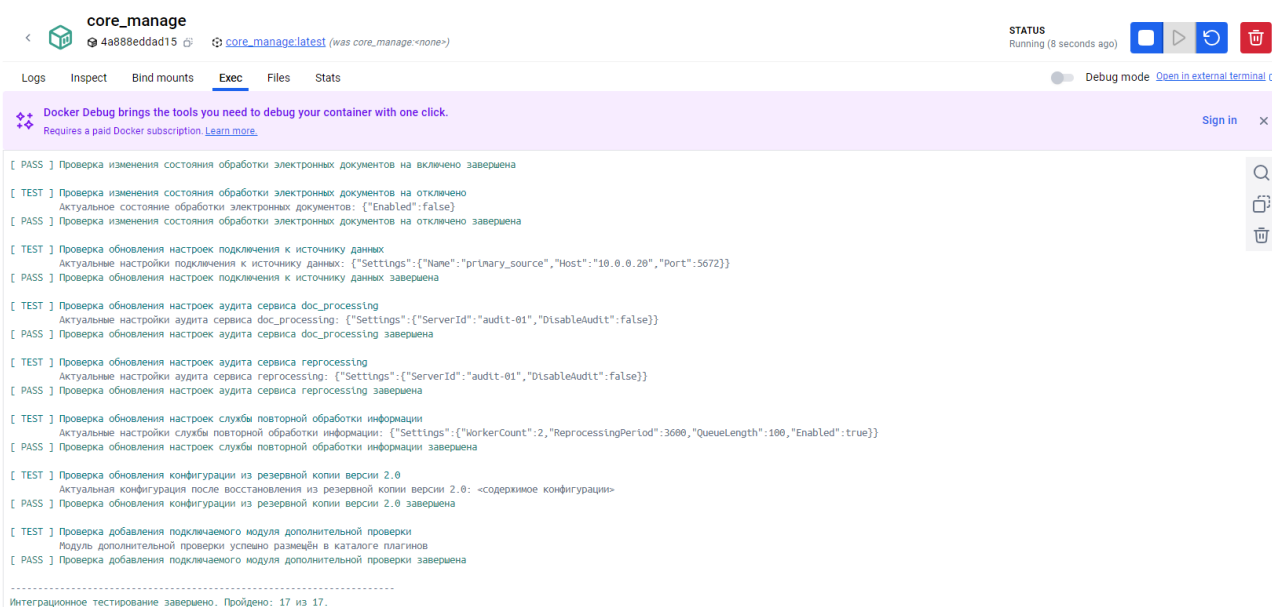


Рисунок 26 – Результаты интеграционного тестирования

По результатам интеграционного тестирования подтверждена корректность совместной работы компонентов системы при управлении конфигурацией. Проверенные сценарии охватывают применение, чтение, сброс и удаление настроек для основных сервисов, а сравнение фактических настроек с ожидаемыми после каждой операции подтверждает их корректное применение и сохранение. Это свидетельствует о согласованности взаимодействия модулей и готовности системы к работе в реальных условиях.

5.4 Оценка качества кода

5.4.1 Метрики Холстеда

Среди подходов к количественной оценке программного кода выделяется группа метрик Холстеда, основанная на анализе используемых в программе операторов и операндов [25]. Данный подход рассматривает исходный код не как набор строк, а как совокупность элементарных лексических единиц, что позволяет судить о насыщенности и логической плотности реализации. На основе подсчёта операторов и операндов вычисляется ряд производных характеристик, отражающих объём программы, её сложность и предполагаемые затраты на разработку и сопровождение. Достоинством такого подхода является объективность и независимость от стиля оформления кода: оценка опирается на формальный лексический состав программы, а не на субъективные суждения.

В качестве объекта анализа выбран метод обновления конфигурации службы обработки электронных документов. Он преобразует поступивший запрос в рабочие настройки службы, согласует текущее и требуемое состояние службы, применяет настройки через её клиент и фиксирует итоговый статус, обрабатывая возможные ошибки. Исходный код метода приведён в листинге 7.

Листинг 7 – Код метода обновления конфигурации службы обработки электронных документов

```
public async Task  
UpdateConfiguration(SetupDocProcessingConfigurationRequest  
request,
```

```

        long timestamp, CancellationToken cancellationToken)
    {
        var settings = request.ConvertToWorkerSettings();
        settings.Timestamp = timestamp;
        var realState = await _client.GetState(cancellationToken);
        var configState = request.ProcessingSettings?.Enabled ?? true;
        if (settings.DataSourceSettings != null)
            settings.DataSourceSettings.Password ??=
_dataSourcePassword.GetValue();
        if (realState && !configState)
        {
            _logger.LogInformation("Выключение обработки электронных
документов");
            await _client.SetState(false, cancellationToken);
        }
        try
        {
            await _client.UpdateSettings(settings, cancellationToken);
            _logger.LogInformation("Настройки службы обработки
электронных документов успешно установлены");
            if (configState)
            {
                if (!realState)
                    await _client.SetState(true, cancellationToken);
                UpdateStatus(ServiceState.ConfiguredAndEnabled, null);
            }
            else
            {
                UpdateStatus(ServiceState.Configured, "Обработка
отключена");
            }
        }
        catch (DocProcessingException ex)
        {
            _logger.LogError("При обновлении настроек службы произошла
ошибка: {message}", ex.Message);
            UpdateStatus(ServiceState.ErrorApplyingSettings,
ex.Message);
            throw;
        }
    }
}

```

Первым шагом расчёта является разбор кода на операторы и операнды с подсчётом числа их повторений. К операторам отнесены ключевые слова языка, знаки операций, разделители и вызовы методов, к операндам – переменные, параметры, имена типов, обращения к членам объектов, а также строковые и иные литералы.

Результаты разбора по операторам сведены в таблицу 17.

Таблица 17 – Операторы метода обновления конфигурации службы обработки электронных документов

№	Оператор	Число вхождений
1	public	1
2	async	1
3	var	3
4	await	4
5	if	4
6	else	1
7	try	1
8	catch	1
9	throw	1
10	=	4
11	??=	1
12	??	1
13	?.	1
14	!=	1
15	&&	1
16	!	2
17	.	19
18	,	9
19	;	15
20	()	18
21	{}	6
22	UpdateConfiguration	1
23	ConvertToWorkerSettings	1
24	GetState	1
25	GetValue	1
26	LogInformation	2
27	SetState	2
28	UpdateSettings	1
29	UpdateStatus	3
30	LogError	1

Общее число операторов составляет $N_1 = 108$ при 30 уникальных операторах ($n_1 = 30$). Результаты разбора по операндам приведены в таблице 18.

Таблица 18 – Операнды метода обновления конфигурации службы обработки электронных документов

№	Операнд	Число вхождений
1	request	2
2	timestamp	1
3	cancellationToken	4
4	settings	5
5	realState	3
6	configState	3

Продолжение таблицы 18

7	Ex	3
8	_client	4
9	_logger	3
10	_dataSourcePassword	1
11	ServiceState	3
12	DataSourceSettings	2
13	Message	2
14	Timestamp	1
15	ProcessingSettings	1
16	Enabled	1
17	Password	1
18	ConfiguredAndEnabled	1
19	Configured	1
20	ErrorApplyingSettings	1
21	SetupDocProcessingConfigurationRequest	1
22	CancellationToken	1
23	Task	1
24	long	1
25	DocProcessingException	1
26	true	2
27	false	1
28	null	2
29	«Выключение обработки электронных документов»	1
30	«Настройки службы обработки электронных документов успешно установлены»	1
31	«Обработка отключена»	1
32	«При обновлении настроек службы произошла ошибка: {message}»	1

Общее число операндов составляет $N_2 = 57$ при 32 уникальных операндах ($n_2 = 32$). По полученным значениям рассчитаны производные характеристики программы по метрике Холстеда. К таким характеристикам относятся словарь и длина программы, её объём и сложность, последовательно вычисляемые по приведённым ниже формулам.

Словарь программы вычисляется по формуле:

$$n = n_1 + n_2 = 30 + 32 = 62.$$

Длина программы вычисляется по формуле:

$$N = N_1 + N_2 = 108 + 57 = 165.$$

Объём программы вычисляется по формуле:

$$V = N \cdot \log_2 n \approx 982.$$

Сложность программы вычисляется по формуле:

$$D = \left(\frac{n_1}{2}\right) \cdot \left(\frac{N_2}{n_2}\right) \approx 27.$$

Оценка интеллектуальных усилий вычисляется по формуле:

$$E = D \cdot V = 26514.$$

Рассчитанные показатели позволяют охарактеризовать метод как обладающий умеренным объёмом и достаточно плотной логической структурой, что объясняется наличием нескольких условных переходов, согласованием состояния службы и обработкой исключений в пределах одной функции. Значения сложности и оценки интеллектуальных усилий указывают на то, что для понимания и изменения метода требуются определённые затраты, однако они остаются в пределах, допускающих дальнейшее сопровождение кода без его обязательной декомпозиции.

5.4.2 Метрика Чепина

Метрика Чепина оценивает сложность программного модуля исходя из характера работы с переменными [26]. В отличие от метрик, опирающихся на объём кода или состав операторов, данный подход анализирует роль переменных в вычислениях и управлении ходом выполнения, что позволяет судить о сложности модуля с точки зрения потоков данных. Основная идея метрики состоит в том, что чем активнее переменные задействованы в управлении логикой и преобразовании данных, тем выше когнитивная нагрузка при сопровождении кода. Расчёт выполняется для того же метода обновления конфигурации службы обработки электронных документов, что и при анализе по метрике Холстеда.

Все переменные метода распределяются по четырём функциональным группам: вводимые переменные, используемые для вычислений и формирования результата (Р); переменные, изменяемые внутри метода (М); управляющие переменные, влияющие на ветвление и порядок выполнения (С); паразитные переменные, не участвующие в вычислениях и не влияющие на логику

работы (Т). Одна переменная может относиться сразу к нескольким группам, если выполняет несколько ролей.

Значение метрики рассчитывается по формуле:

$$Q = a1 \cdot P + a2 \cdot M + a3 \cdot C + a4 \cdot T,$$

где $a1$, $a2$, $a3$, $a4$ – весовые коэффициенты, отражающие степень влияния каждой группы на сложность модуля. Наибольший вес присваивается управляющим переменным, так как они определяют поток управления. В расчёте используются коэффициенты $a1 = 1$, $a2 = 2$, $a3 = 3$, $a4 = 0,5$.

Для рассматриваемого метода переменные распределены следующим образом:

- вводимые переменные: request, timestamp, cancellationToken;
- модифицируемые переменные: settings, realState, configState;
- управляющие переменные: realState, configState, settings;
- паразитные переменные: отсутствуют.

Подстановка полученных значений в формулу даёт:

$$Q = 1 \cdot 3 + 2 \cdot 3 + 3 \cdot 3 + 0,5 \cdot 0 = 18.$$

Полученное значение метрики Чепина показывает, что метод обладает умеренной сложностью с точки зрения работы с переменными. Основной вклад в итоговую оценку вносят управляющие и модифицируемые переменные, что связано с согласованием текущего и требуемого состояния службы и наличием логических проверок, определяющих порядок применения настроек. Это соответствует назначению метода, реализующего не отдельную операцию, а последовательную процедуру подготовки, применения и фиксации конфигурации с обработкой ошибок.

Заключение

В результате выполнения выпускной квалификационной работы были разработаны программные средства, обеспечивающие единый механизм администрирования конфигурации функционально разнородных служб информационной системы электронного документооборота – единообразный для всех служб и расширяемый без модификации основных компонентов. Решение объединяет настройку разнородных служб в рамках единой точки управления и построено на основе микроядерной архитектуры, в которой каждая служба представлена независимым подключаемым модулем. Такой подход обеспечивает добавление, исключение и обновление отдельных модулей без изменения остальных компонентов и единообразный доступ к их настройкам, что позволяет наращивать состав поддерживаемых служб без существенной переработки существующего кода.

Архитектура решения выведена непосредственно из нефункциональных требований, которые для данной задачи являются определяющими. Помимо унифицированного управления конфигурацией, разработанные программные средства предоставляют ряд возможностей:

- централизованное хранение настроек всех управляемых служб в единой точке;
- локальный и удалённый каналы администрирования;
- атомарное применение изменений – целиком либо с откатом – при разграничении параллельного доступа к настройкам;
- событийное оповещение зависимых служб об изменении настроек;
- создание и восстановление целостной резервной копии конфигурации как единого снимка с поддержкой версионирования её формата;
- горизонтальное масштабирование служб-потребителей, не хранящих собственных копий настроек;
- устойчивое фоновое применение конфигурации, сохраняющее работоспособность при временной недоступности служб.

Это повышает управляемость системы, снижает трудоёмкость её сопровождения и уменьшает вероятность ошибок по сравнению с подходом, при котором настройки каждой службы задаются обособленно.

На этапе анализа предметной области были изучены особенности конфигурирования функционально разнородных служб в составе информационной системы, а также рассмотрены существующие средства управления конфигурацией приложений, для каждого из которых выявлены сильные и слабые стороны. По итогам сравнительного анализа была сформулирована постановка задачи и выполнено планирование разработки с оценкой требуемых ресурсов.

На этапе анализа требований были определены функциональные и нефункциональные требования, описывающие как состав реализуемых функций, так и условия, которым должна удовлетворять система в процессе эксплуатации; при этом именно нефункциональные требования и противоречия между ними определили облик системы.

На этапе проектирования архитектура была выведена из нефункциональных требований и включает ядро, плагиновую подсистему, механизм межплагинного взаимодействия и подсистему удалённого администрирования. Для каждой управляемой службы определена структура соответствующего плагина.

На этапе реализации были разработаны ядро, набор плагинов управления настройками отдельных служб и подсистема удалённого администрирования. Реализованы динамическая загрузка плагинов, резервное копирование и версионирование формата конфигурации, разграничение параллельного доступа к настройкам, а также защита фонового применения настроек от устойчивых сбоев.

На этапе тестирования была проведена проверка корректности разработанных программных средств. Выполнено функциональное тестирование методом белого ящика с применением критериев покрытия операторов, решений и условий, модульное тестирование основных плагинов и

интеграционное тестирование, подтвердившее согласованность совместной работы компонентов. Качество программного кода дополнительно оценено с помощью метрик Холстеда и Чепина. Несовпадений между ожидаемыми и фактическими результатами выявлено не было, что подтверждает корректность работы разработанных программных средств.

Корректность и практическую применимость созданного решения подтверждает акт о внедрении, выданный компанией «ИнфоТеКС». Акт приведён в приложении Г.

Таким образом, поставленная цель достигнута, а все задачи выпускной квалификационной работы выполнены в полном объёме.

Список использованных источников

1. Соммервилл, И. Инженерия программного обеспечения / И. Соммервилл. – М.: Вильямс, 2002. – 624 с.
2. Морис, К. Инфраструктура как код / К. Морис. – СПб.: Питер, 2019. – 304 с.
3. Документация HashiCorp Consul [Электронный ресурс]. – URL: <https://developer.hashicorp.com/consul> (дата обращения: 21.04.2026).
4. Статистика заработных плат в сфере информационных технологий в Пензенской области [Электронный ресурс] // hh.ru. – URL: <https://stats.hh.ru/> (дата обращения: 22.04.2026).
5. ГОСТ Р ИСО/МЭК 25010-2015. Системная и программная инженерия. Требования и оценка качества систем и программного обеспечения (SQuaRE). Модели качества систем и программного обеспечения. – М.: Стандартинформ, 2015. – 36 с.
6. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. – СПб.: Питер, 2018. – 352 с.
7. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET / Дж. Рихтер. – СПб.: Питер, 2018. – 896 с.
8. Албахари, Дж. C# 10 и .NET 6. Полное руководство для разработчика / Дж. Албахари. – СПб.: Питер, 2023. – 1216 с.
9. Документация Entity Framework Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/ef/core/> (дата обращения: 28.04.2026).
10. Документация ASP.NET Core [Электронный ресурс]. – URL: <https://learn.microsoft.com/ru-ru/aspnet/core/> (дата обращения: 01.05.2026).
11. RFC 9110. HTTP Semantics [Электронный ресурс]. – URL: <https://www.rfc-editor.org/rfc/rfc9110> (дата обращения: 03.05.2026).
12. Документация Apache Thrift [Электронный ресурс]. – URL: <https://thrift.apache.org/docs/> (дата обращения: 04.05.2026).

13. Фаулер, М. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / М. Фаулер. – СПб.: Символ-Плюс, 2018. – 192 с.
14. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер. – М.: Вильямс, 2018. – 544 с.
15. Документация веб-сервера Nginx [Электронный ресурс]. – URL: <https://nginx.org/ru/docs/> (дата обращения: 08.05.2026).
16. Уорд, Б. Внутреннее устройство Linux / Б. Уорд. – СПб.: Питер, 2021. – 480 с.
17. Документация SQLite [Электронный ресурс]. – URL: <https://www.sqlite.org/docs.html> (дата обращения: 10.05.2026).
18. Троелсен, Э. Язык программирования C# 10 и платформа .NET 6 / Э. Троелсен, Ф. Джепикс. – М.: Диалектика, 2022. – 1312 с.
19. Симан, М. Внедрение зависимостей на платформе .NET / М. Симан, С. ван Деурсен. – СПб.: Питер, 2021. – 552 с.
20. RFC 8259. The JavaScript Object Notation (JSON) Data Interchange Format [Электронный ресурс]. – URL: <https://www.rfc-editor.org/rfc/rfc8259> (дата обращения: 12.05.2026).
21. Майерс, Г. Искусство тестирования программ / Г. Майерс, Т. Баджетт, К. Сандлер. – 3-е изд. – М.: Вильямс, 2019. – 272 с.
22. Документация по фреймворку xUnit.net [Электронный ресурс]. – URL: <https://xunit.net/docs/> (дата обращения: 14.05.2026).
23. Документация по библиотеке Moq [Электронный ресурс]. – URL: <https://github.com/devlooped/moq/wiki> (дата обращения: 16.05.2026).
24. Моуэт, Э. Использование Docker / Э. Моуэт. – М.: ДМК Пресс, 2017. – 354 с.
25. Холстед, М. Х. Начала науки о программах / М. Х. Холстед. – М.: Финансы и статистика, 1981. – 128 с.
26. Орлов, С. А. Технологии разработки программного обеспечения / С. А. Орлов. – СПб.: Питер, 2017. – 480 с.

Приложение А
(справочное)
Глоссарий

Apache Thrift – фреймворк описания интерфейсов и кроссплатформенного межпроцессного взаимодействия с генерацией клиентского и серверного кода.

C# – строго типизированный объектно-ориентированный язык программирования общего назначения.

Docker – платформа контейнеризации, обеспечивающая запуск приложений в изолированном воспроизводимом окружении.

Entity Framework Core – объектно-реляционная система отображения для платформы .NET, обеспечивающая доступ к базе данных через объектную модель.

HTTP (HyperText Transfer Protocol) – протокол прикладного уровня для передачи данных между клиентом и сервером.

JSON (JavaScript Object Notation) – текстовый формат обмена данными, основанный на парах «ключ-значение».

Nginx – веб-сервер, используемый для обработки входящих соединений и проксирования запросов.

REST (Representational State Transfer) – архитектурный стиль взаимодействия клиента и сервера через протокол HTTP с использованием стандартных методов доступа к ресурсам.

SQLite – встраиваемая реляционная система управления базами данных, хранящая данные в едином файле.

xUnit – фреймворк модульного тестирования для платформы .NET.

.NET – программная платформа компании Microsoft для разработки и выполнения приложений.

Приложение Б
(обязательное)
Исходный код приложения

Листинг Б.1 – Класс модульного тестирования

DocProcessingConfigurationControllerUnitTests

```
namespace ConfigManagement.Core.UnitTests;

/// <summary> Модульные тесты контроллера настроек обработки
документов. </summary>
public class DocProcessingConfigurationControllerUnitTests
{
    private readonly DocProcessingConfigurationControllerFactory
    _factory = new();

    [Fact]
    public async void
    GetDataSourceConfiguration_Success_ResultOk()
    {
        var config =
        DocProcessingConfigurationFactory.BuildDataSourceConfiguration();
        _factory.ConfigurationStorageMock
            .Setup(sm =>
            sm.GetDocProcessingConfiguration()).Returns(config);
        var controller = _factory.Build();

        var answer = await
        controller.GetDocProcessingConfiguration();

        var result =
        ControllerAssert.OkObject<DocProcessingConfigurationResponse>(answ
        er);
        Assert.Equal(config.DataSourceManagerSettings,
        result.DataSourceManagerSettings);
    }

    [Fact]
    public async void
    GetDataSourceConfiguration_Fails_ReturnUnprocessable()
    {
        _factory.ConfigurationStorageMock
            .Setup(sm =>
            sm.GetDocProcessingConfiguration()).Throws(new Exception());
        var controller = _factory.Build();

        var answer = await
        controller.GetDocProcessingConfiguration();

        ControllerAssert.UnprocessableEntity(answer);
    }
}
```

Листинг Б.2 – Класс модульного тестирования

ReprocessingConfigurationControllerUnitTests

```

namespace ConfigManagement.Core.UnitTests;
/// <summary> Модульные тесты контроллера службы повторной
обработки информации. </summary>
public class ReprocessingConfigurationControllerUnitTests
{
    private readonly ReprocessingConfigurationControllerFactory
_factory = new();

    [Fact]
    public async void
GetReprocessingConfiguration_Success_ReturnOk()
    {
        var config =
ReprocessingConfigurationFactory.BuildReprocessingConfiguration();
        _factory.StorageMock.Setup(sm =>
sm.GetReprocessingConfiguration()).Returns(config);
        var controller = _factory.Build();

        var answer = await
controller.GetReprocessingConfiguration();

        var result =
ControllerAssert.OkObject<ReprocessingConfiguration>(answer);
        Assert.Equal(config, result);
    }

    [Fact]
    public async Task
UpdateReprocessingConfiguration_Fails_ReturnUnprocessable()
    {
        var request =
ReprocessingConfigurationFactory.BuildUpdateFullReprocessingConfig
urationRequest();
        _factory.SetuperMock
            .Setup(sm =>
sm.Setup(It.IsAny<UpdateReprocessingConfigurationRequest>()))
            .Throws(new Exception());
        var controller = _factory.Build();

        var answer = await
controller.UpdateReprocessingConfiguration(request);

        ControllerAssert.UnprocessableEntity(answer);
    }
}

```

Листинг Б.3 – Класс модульного тестирования StorageObjectsControllerUnitTests

```

namespace ConfigManagement.Core.UnitTests;
/// <summary> Модульные тесты контроллера объектов хранилища.
</summary>
public class StorageObjectsControllerUnitTests
{

```

```

        private readonly StorageObjectsControllerFactory _factory =
new();

        [Theory]
        [InlineData(SegmentType.Primary, ObjectType.Base)]
        [InlineData(SegmentType.Secondary, ObjectType.User)]
        public async void GetObjects_Success_ReturnOk(SegmentType
segment, ObjectType type)
        {
            _factory.ObjectProviderMock
                .Setup(pm => pm.GetObjects(segment, type,
It.IsAny<CancellationToken>()))
                .Returns(Task.FromResult(new
List<ObjectInfoResponse>()));
            var controller = _factory.Build();

            var answer = await controller.GetObjects(segment, type);

            var result =
ControllerAssert.OkObject<List<ObjectInfoResponse>>(answer);
            Assert.NotNull(result);
        }

        [Theory]
        [InlineData(SegmentType.Primary, ObjectType.Base)]
        public async void
AddObject_IncorrectData_ReturnBadRequest(SegmentType segment,
ObjectType type)
        {
            var objectFile =
StorageObjectFactory.BuildEmptyObjectFile();
            var controller = _factory.Build();

            var answer = await controller.AddObject(segment, type,
objectFile);

            ControllerAssert.BadRequest(answer);
            _factory.ObjectProviderMock.Verify(pm =>
                pm.AddObject(segment, type, It.IsAny<byte[]>()),
It.IsAny<CancellationToken>()),
                Times.Never);
        }
    }
}

```

Листинг Б.4 – Класс модульного тестирования StorageSettingsControllerUnitTests

```

namespace ConfigManagement.Core.UnitTests;
/// <summary> Модульные тесты контроллера настроек хранилища
объектов. </summary>
public class StorageSettingsControllerUnitTests
{
    private readonly StorageSettingsControllerFactory _factory =
new();

```

```

[Theory]
[InlineData(SegmentType.Primary)]
[InlineData(SegmentType.Secondary)]
public async void GetSettings_Success_ReturnOk(SegmentType
segment)
{
    var response =
StorageSettingsFactory.BuildSettingsResponse();
    _factory.StorageSettingsProviderMock
        .Setup(pm => pm.GetSettings(segment,
It.IsAny<CancellationToken>()))
        .Returns(Task.FromResult(response));
    var controller = _factory.Build();

    var answer = await controller.GetSettings(segment);

    var result =
ControllerAssert.OkObject<StorageSettingsResponse>(answer);
    Assert.Equal(response, result);
}
}

```

Листинг Б.5 – Класс модульного тестирования

DataSecurityConfigurationControllerUnitTests

```

namespace ConfigManagement.Core.UnitTests;
/// <summary> Модульные тесты контроллера настроек защиты данных.
</summary>
public class DataSecurityConfigurationControllerUnitTests
{
    private readonly DataSecurityConfigurationControllerFactory
_factory = new();

    [Theory]
    [InlineData(DataSecurityType.PrimarySegment)]
    [InlineData(DataSecurityType.Reprocessing)]
    public async void
SetProvider_Success_ReturnOk(DataSecurityType type)
    {
        var request =
DataSecurityConfigurationFactory.BuildSetDataSecurityProviderReque
st(type);
        _factory.DataSecuritySetuperMock
            .Setup(sm => sm.Setup(type,
It.IsAny<UpdateDataSecurityConfigurationRequest>())));
        var controller = _factory.Build();

        var answer = await controller.SetProvider(request);

        ControllerAssert.Ok(answer);
        _factory.DataSecuritySetuperMock.Verify(sm =>

```

```

        sm.Setup(type,
It.IsAny<UpdateDataSecurityConfigurationRequest>()), Times.Once);
    }

    [Fact]
    public async void
GetDataSecurityConfiguration_Fails_ReturnUnprocessable()
    {
        _factory.DataSecurityStorageMock
            .Setup(sm =>
sm.GetDataSecurityConfiguration(It.IsAny<DataSecurityType>()))
            .Throws(new Exception());
        var controller = _factory.Build();

        var answer = await
controller.GetDataSecurityConfiguration(It.IsAny<DataSecurityType>
());

        ControllerAssert.UnprocessableEntity(answer);
    }
}

```

Листинг Б.6 – Класс интеграционного тестирования ManagementIntTests

```

using ConfigManagement.Core.IntTests.Clients;
using ConfigManagement.Core.IntTests.Data;
using ConfigManagement.Core.IntTests.Factory;
using ConfigManagement.Core.IntTests.TestPriority;
using System.Text.Json;

namespace ConfigManagement.Core.IntTests;

/// <summary> Интеграционные тесты управления настройками
сервисов. </summary>
[TestCaseOrderer(
    ordererTypeName:
"ConfigManagement.Core.IntTests.TestPriority.PriorityOrderer",
    ordererAssemblyName: "ConfigManagement.Core.IntTests")]
public class ManagementIntTests
{
    /// <summary> Применение основных настроек службы штампов
времени. </summary>
    [Theory]
    [InlineData(Segment.Primary, "main_server"), TestPriority(1)]
    [InlineData(Segment.Secondary, "new_main_server"),
TestPriority(2)]
    public async void SetCommonSettings(Segment segment, string
uri)
    {
        TestConsole.WriteDescription($"Проверка применения
основных настроек службы штампов времени для {segment}");
    }
}

```

```

        var setSettingsRequest =
TimestampConfigurationFactory.BuildSetCommonSettingsRequest(segment, uri);
        var getSettingsRequest =
TimestampConfigurationFactory.BuildGetSettingsRequest(segment);
        var client =
ManagementClientsFactory.CreateTimestampManagementClient();

        await client.SetCommonSettings(setSettingsRequest);

        var actualConfiguration = await
client.GetSettings(getSettingsRequest);
        Console.WriteLine($"Актуальная конфигурация для {segment}:
{JsonSerializer.Serialize(actualConfiguration)}");

        var mainServer =
actualConfiguration?.Settings?.Servers?.FirstOrDefault(s =>
s.IsMain);
        Assert.NotNull(mainServer);
        Assert.Equal(uri, mainServer.CommonSettings!.Uri);

        TestConsole.WriteDescription($"Проверка применения
основных настроек службы штампов времени для {segment}
завершена");
    }

    /// <summary> Валидация некорректных адресов серверов службы
штампов времени. </summary>
    [Theory, TestPriority(3)]

    [MemberData(nameof(TimestampSettingsTestData.AddIncorrectServersTe
stData), MemberType = typeof(TimestampSettingsTestData))]
    public async void CheckServers_Fails_ThrowException(Segment
segment, List<string> servers)
    {
        TestConsole.WriteDescription($"Проверка некорректных
адресов серверов службы штампов времени для {segment}");

        var request = new CheckTimestampServersRequest { Segment =
segment, Servers = servers };
        var client =
ManagementClientsFactory.CreateTimestampManagementClient();

        var result = await
Assert.ThrowsAsync<TimestampManagementException>(async () => await
client.CheckServers(request));

        Assert.NotNull(result);
        Assert.Equal(TimestampOperationStatus.BadRequest,
result.Status);
    }

```

```

        TestConsole.WriteDescription($"Проверка некорректных
адресов серверов службы штампов времени для {segment} завершена");
    }

    /// <summary> Обновление настроек основного подключения к базе
данных. </summary>
    [Fact, TestPriority(4)]
    public async void UpdateDbAccessSettings_Success_True()
    {
        TestConsole.WriteDescription("Проверка обновления настроек
основного подключения к базе данных");

        const string DbName = "main_db";
        var request =
        DbAccessConfigurationFactory.BuildUpdateDbAccessSettingsRequest(Db
        Name);
        var client =
        ManagementClientsFactory.CreateDatabaseManagementClient();

        await client.UpdateDbAccessSettings(request);

        var actualSettings = await client.GetDbAccessSettings();
        Console.WriteLine($"Актуальные настройки основного
подключения к базе данных:
{JsonSerializer.Serialize(actualSettings)}");

        Assert.NotNull(actualSettings?.Settings);
        Assert.NotNull(request.Settings);
        Assert.Equal(request.Settings.Name,
actualSettings.Settings.Name);
        Assert.Equal(request.Settings.Host,
actualSettings.Settings.Host);
        Assert.Equal(request.Settings.Port,
actualSettings.Settings.Port);

        TestConsole.WriteDescription("Проверка обновления настроек
основного подключения к базе данных завершена");
    }

    /// <summary> Обновление основных настроек обработки
электронных документов. </summary>
    [Fact, TestPriority(5)]
    public async void
UpdateCommonDocProcessingSettings_Success_True()
    {
        TestConsole.WriteDescription("Проверка обновления основных
настроек обработки электронных документов");

        var request =
        DocProcessingConfigurationFactory.BuildUpdateCommonSettingsRequest
        ();
    }

```

```

        var client =
ManagementClientsFactory.CreateDocProcessingManagementClient();

        await client.UpdateCommonSettings(request);

        var actualSettings = await client.GetCommonSettings();
        Console.WriteLine($"Актуальные основные настройки
обработки электронных документов:
{JsonSerializer.Serialize(actualSettings)}");

        Assert.NotNull(actualSettings?.Settings);
        Assert.NotNull(request.Settings);
        Assert.Equal(request.Settings.InputQueueName,
actualSettings.Settings.InputQueueName);
        Assert.Equal(request.Settings.CountWorkers,
actualSettings.Settings.CountWorkers);
        Assert.Equal(request.Settings.TimestampEnabled,
actualSettings.Settings.TimestampEnabled);

        TestConsole.WriteDescription("Проверка обновления основных
настроек обработки электронных документов завершена");
    }

    /// <summary> Переключение состояния обработки электронных
документов. </summary>
    [Theory]
    [InlineData(true), TestPriority(6)]
    [InlineData(false), TestPriority(7)]
    public async void SetDocProcessingState_Success_True(bool
state)
    {
        var label = state ? "включено" : "отключено";
        TestConsole.WriteDescription($"Проверка изменения
состояния обработки электронных документов на {label}");

        var setStateRequest =
DocProcessingConfigurationFactory.BuildSetStateRequest(state);
        var getStateRequest =
DocProcessingConfigurationFactory.BuildGetStateRequest();
        var client =
ManagementClientsFactory.CreateDocProcessingManagementClient();

        await client.SetState(setStateRequest);

        var actualState = await client.GetState(getStateRequest);
        Console.WriteLine($"Актуальное состояние обработки
электронных документов: {JsonSerializer.Serialize(actualState)}");

        Assert.NotNull(actualState);
        Assert.Equal(state, actualState.Enabled);
    }

```

```

        TestConsole.WriteDescription($"Проверка изменения
состояния обработки электронных документов на {label} завершена");
    }

    /// <summary> Обновление настроек подключения к источнику
данных. </summary>
    [Fact, TestPriority(8)]
    public async void
UpdateDataSourceConnectionSettings_Success_True()
    {
        TestConsole.WriteDescription("Проверка обновления настроек
подключения к источнику данных");

        var request =
DataSourceConfigurationFactory.BuildUpdateConnectionSettingsReques
t();
        var client =
ManagementClientsFactory.CreateDataSourceManagementClient();

        await client.UpdateConnectionSettings(request);

        var actualSettings = await client.GetConnectionSettings();
        Console.WriteLine($"Актуальные настройки подключения к
источнику данных: {JsonSerializer.Serialize(actualSettings)}");

        Assert.NotNull(actualSettings?.Settings);
        Assert.NotNull(request.Settings);
        Assert.Equal(request.Settings.Host,
actualSettings.Settings.Host);
        Assert.Equal(request.Settings.Port,
actualSettings.Settings.Port);

        TestConsole.WriteDescription("Проверка обновления настроек
подключения к источнику данных завершена");
    }

    /// <summary> Обновление настроек аудита. </summary>
    [Theory, TestPriority(9)]
    [InlineData("doc_processing")]
    [InlineData("reprocessing")]
    public async void UpdateAuditSettings_Success_True(string
serviceId)
    {
        TestConsole.WriteDescription($"Проверка обновления
настроек аудита сервиса {serviceId}");

        var updateRequest =
AuditConfigurationFactory.BuildUpdateAuditSettingsRequest(serviceI
d);
        var getRequest =
AuditConfigurationFactory.BuildGetAuditSettingsRequest(serviceId);

```

```

        var client =
ManagementClientsFactory.CreateAuditManagementClient();

        await client.UpdateAuditSettings(updateRequest);

        var actualSettings = await
client.GetAuditSettings(getRequest);
        Console.WriteLine($"Актуальные настройки аудита сервиса
{serviceId}: {JsonSerializer.Serialize(actualSettings)}");

        Assert.NotNull(actualSettings?.Settings);
        Assert.NotNull(updateRequest.Settings);
        Assert.Equal(updateRequest.Settings.ServerId,
actualSettings.Settings.ServerId);

        TestConsole.WriteDescription($"Проверка обновления
настроек аудита сервиса {serviceId} завершена");
    }

    /// <summary> Обновление настроек службы повторной обработки
информации. </summary>
    [Fact, TestPriority(10)]
    public async void UpdateReprocessingSettings_Success_True()
    {
        TestConsole.WriteDescription("Проверка обновления настроек
службы повторной обработки информации");

        var request =
ReprocessingConfigurationFactory.BuildSetSettingsRequest();
        var client =
ManagementClientsFactory.CreateReprocessingManagementClient();

        await client.SetSettings(request);

        var actualSettings = await client.GetSettings();
        Console.WriteLine($"Актуальные настройки службы повторной
обработки информации:
{JsonSerializer.Serialize(actualSettings)}");

        Assert.NotNull(actualSettings?.Settings);
        Assert.NotNull(request.Settings);
        Assert.Equal(request.Settings.WorkerCount,
actualSettings.Settings.WorkerCount);
        Assert.Equal(request.Settings.Enabled,
actualSettings.Settings.Enabled);

        TestConsole.WriteDescription("Проверка обновления настроек
службы повторной обработки информации завершена");
    }

    /// <summary> Обновление конфигурации из резервной копии.
</summary>

```

```

[Theory]
[InlineData(2, 0), TestPriority(int.MaxValue - 1)]
public async void UpdateConfiguration_Success_True(int major,
int minor)
{
    var version = new Version { Major = major, Minor = minor
};
    TestConsole.WriteDescription($"Проверка обновления
конфигурации из резервной копии версии
{version.Major}.{version.Minor}");

    var configuration =
ConfigurationBackupFactory.BuildServiceConfigurationByVersion(version);
    var client =
ManagementClientsFactory.CreateConfigurationBackupClient();

    await client.UpdateConfiguration(configuration);

    var actualConfiguration = await client.GetConfiguration();
    Console.WriteLine($"Актуальная конфигурация после
восстановления: {actualConfiguration?.Content}");

    Assert.NotNull(actualConfiguration?.Version);
    Assert.NotNull(actualConfiguration.Content);

    TestConsole.WriteDescription($"Проверка обновления
конфигурации из резервной копии версии
{version.Major}.{version.Minor} завершена");
}

/// <summary> Добавление подключаемого модуля дополнительной
проверки. </summary>
[Fact, TestPriority(int.MaxValue)]
public async void AddValidationPlugin_Success_True()
{
    TestConsole.WriteDescription("Проверка добавления
подключаемого модуля дополнительной проверки");

    var request =
ValidationPluginsConfigurationFactory.BuildAddValidationPluginRequest();
    var pluginName = new FileInfo(request.Path);
    var pluginLocation =
Path.Combine(ValidationPluginsConfigurationFactory.ValidationPluginsDirectory, pluginName.Name);
    var client =
ManagementClientsFactory.CreateValidationPluginsManagementClient();

    await client.AddValidationPlugin(request);

```

```
Assert.True(File.Exists(pluginLocation));

TestConsole.WriteDescription("Проверка добавления
подключаемого модуля дополнительной проверки завершена");
    }
}
```

Приложение В
(рекомендуемое)
Результаты тестирования

```
warn: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:47:20] Приложение не завершило запуск, фоновая операция обновления конфигурации не будет запущена
```

Рисунок В.1 – Журнал при незапущенном приложении

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:48:12] Запуск фоновой операции обновления конфигурации
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:48:12] Получен сигнал отмены, фоновая операция завершена до входа в цикл обновления
```

Рисунок В.2 – Журнал при отмене операции до входа в цикл

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:48:28] Запуск фоновой операции обновления конфигурации
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:48:28] Фоновое обновление настроек обработки электронных документов отключено
```

Рисунок В.3 – Журнал при отключённом обновлении

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:02] Запуск фоновой операции обновления конфигурации
warn: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:02] Служба обработки электронных документов неактивна, обновление прервано, ожидается запуск службы
```

Рисунок В.4 – Журнал при неактивной службе

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:28] Запуск фоновой операции обновления конфигурации
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:28] Применение настроек обработки электронных документов
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:28] Транзакция обновления конфигурации завершена успешно, следующий запуск через 30 с
```

Рисунок В.5 – Журнал при успешном применении настроек

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:50] Запуск фоновой операции обновления конфигурации
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:50] Применение настроек обработки электронных документов
fail: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:49:50] При обновлении конфигурации службы обработки электронных документов произошла ошибка: целевая служба недоступна
```

Рисунок В.6 – Журнал при ошибке применения настроек

```
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:50:15] Запуск фоновой операции обновления конфигурации
info: ConfigManagement.Plugins.DocProcessing.BackgroundConfigurationUpdater[0]
[2026-06-02 19:50:15] Фоновое обновление настроек обработки электронных документов отключено временно
```

Рисунок В.7 – Журнал при временно отключённом обновлении

Приложение Г
(рекомендуемое)
Акт о внедрении

infotecs

Акционерное общество
«Информационные технологии
и коммуникационные системы»
(АО «ИнфоТеКС»)

Юридический адрес: ул. Мишина,
дом 56, стр. 2, этаж 2, пом. IX, комната 29
Москва, 127083

Почтовый адрес: а/я № 80,
улица Отрадная, дом 2Б, стр. 1
Москва, 127273

Тел. (495)737-61-92 Факс (495)737-72-78

ОГРН 1027739185066
ИНН/КПП 7710013769/771401001
<http://www.infotecs.ru>

Акционерное общество «Информационные
технологии и коммуникационные системы»
(АО «ИнфоТеКС»)

127083, Москва, улица Мишина, д. 56, стр. 2,
этаж 2, помещение IX, комната 29

.....08.06.2026..... №.....

Акт о внедрении

Настоящим подтверждаю, что Каледа Роман Александрович является сотрудником компании АО «ИнфоТеКС». Решения, описанные в выпускной квалификационной работе на тему «Программные средства обеспечения механизма единого управления конфигурацией информационной системы электронного документооборота», вошли в состав разрабатываемых технологических средств.

Выдано Каледо Роману Александровичу для предъявления по требованию.

Руководитель обособленного
подразделения в г. Пенза



М.М. Грунтович