

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

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление 09.03.04 «Программная инженерия»
подготовки
Направленность Программная инженерия

БАКАЛАВРСКАЯ РАБОТА

на тему

Разработка подсистемы управления справочной информацией

Студент _____ Абузяров Айнур Махмудович
(подпись, дата) (ФИО полностью)

Руководитель _____ Макарычев П.П.
(подпись, дата) (фамилия, инициалы)

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

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

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

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

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

Пенза, 2025

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

«20» декабря 2024 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

1. Студент Абузаров Айнур Махмудович гр. 21ВП2 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Разработка подсистемы управления справочной информацией
Тема утверждена приказом ПГУ № _____ от " ____ " _____ 2025

3. Руководитель работы д.т.н., профессор кафедры «МО и ПЭВМ»
Макарычев П.П.

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

Назначение: программные средства для управления справочной
информацией защищенного узла.

Среда выполнения: Windows 10, Debian 11, Mac OS 10

Языковые средства разработки: C++, UML.

Инструментальные средства: Visual Studio Code, Apache Thrift,
Visual Paradigm 17.2.

Технологии разработки: ООП, принципы SOLID, шаблоны

Основные функции приложения:

а) первичная инициализация защищенного узла;

б) пошаговое применение обновлений справочной информации;

в) извлечение справочников из дистрибутивов ключевой информации

г) удаление справочной информации с узла.

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

№	Наименование этапов работы	Объем	Срок	Подпись
---	----------------------------	-------	------	---------

п/п		работы	выполнения	руководителя
1	Анализ предметной области и требований к системе	10%	08.04.2025 – 14.04.2025	
2	Проектирование программных средств	20%	15.04.2025 – 25.04.2025	
3	Реализация программных средств	30%	26.04.2025 – 12.05.2025	
4	Тестирование программных средств	20%	13.05.2025 – 22.05.2025	
5	Оформление пояснительной записки	20%	23.05.2025 – 03.06.2025	

Дата выдачи задания «20» декабря 2024г.

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

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

Реферат

Пояснительная записка содержит 64 листа, 5 рисунков, 10 таблиц, 20 использованных источников и 2 приложения.

СПРАВОЧНАЯ ИНФОРМАЦИЯ, КОНФИГУРАЦИЯ, UML, C++, DOCKER.

Объект исследования – программные средства для управления справочной информацией защищенного узла

Цель выпускной квалификационной работы – спроектировать и разработать программные средства для управления справочной информацией защищенного узла.

Технологии разработки Visual Studio Code, Apache Thrift, Docker.

Среда исполнения – Windows 10, Debian 11 , Mac OS.

Результаты работы – в ходе выполнения выпускной квалификационной работы были спроектированы и разработаны программные средства для управления справочной информацией защищенного узла. Все требования к разработке приложения соблюдены.

					ПГУ 09.03.04 – 08ВР212.01 ПЗ				
Изм.	Лист	№ докум.	Подпись	Дата	Разработка подсистемы управления справочной информацией	Лист	Лист	Листов.	
Разраб.		Абузаров А.М.							
Пров.		Макарычев П.П.					4	64	
Реценз.						Группа 21ВП2			
Н. Контр.		Такташкин Д.В.							
Утв.		Козлов А.Ю.							

Содержание

Введение.....	6
1 Анализ предметной области и постановка задачи на разработку.....	8
1.1 Анализ предметной области	8
1.2 Анализ ПО аналогичного назначения.....	10
1.2.1 IBM InfoSphere MDM Server for PIM.....	10
1.2.2 Oracle Master Data Management (MDM).....	10
1.2.3 1C: Управление нормативно-справочной информацией	11
1.2.4 NORMA (ЛАНИТ)	11
1.2.5 SAP Master Data Governance (MDG)	11
1.2.6 Выводы.....	12
1.3 Постановка задачи на разработку программных средств.....	13
2 Анализ требований на разработку подсистемы.....	15
2.1 Анализ функциональных требований.....	15
2.2 Анализ нефункциональных требований	21
2.3 Планирование разработки и оценка бюджета.....	23
3 Проектирование подсистемы.....	26
3.1 Компоненты подсистемы	26
3.2 Интерфейс подсистемы	29
4 Реализация подсистемы.....	31
5 Контроль качества программного обеспечения.....	35
5.1 Функциональное тестирование	35
5.2 Нефункциональное тестирование	41
5.3 Оценка качества программного кода	45
Список используемых источников.....	60
Приложение А (справочное) Глоссарий	62
Приложение Б (обязательное) Листинг приложения	64

Введение

Современные распределенные информационные системы, функционирующие в условиях повышенных требований к безопасности, особенно в защищенных контурах или при обработке конфиденциальных данных, предъявляют исключительно высокие требования к организации управления критически важной служебной информацией. К этой категории информации, безусловно, относятся криптографические ключи и сертификаты, списки контроля доступа (ACL), параметры политик безопасности, разнообразные справочные данные (конфигурационные справочники, классификаторы, нормативно-справочная информация – НСИ) и другие элементы, абсолютно необходимые для обеспечения стабильного функционирования, поддержания целостности и гарантии конфиденциальности как системы в целом, так и каждого отдельного защищенного узла. Эта служебная справочная информация составляет фундаментальную основу для корректной настройки и конфигурации всех компонентов распределенной системы.

Эффективное и, что критически важно, безопасное управление подобной справочной информацией является одной из ключевых и наиболее сложных задач при проектировании и эксплуатации защищенных распределенных сред. Несвоевременное распространение обновлений, некорректное внесение изменений или, что наиболее опасно, несанкционированное распространение и модификация этих данных между узлами сети способны привести не только к локальным сбоям в работе или нарушению установленных политик безопасности, но и к полной компрометации всей системы с катастрофическими последствиями. Особую сложность в контексте управления справочной информацией представляют задачи обеспечения ее синхронизации в распределенной среде. Гарантировать строгую актуальность и абсолютную непротиворечивость этих данных на всех без исключения узлах, особенно в условиях динамического изменения конфигурации и политик, а также

обеспечить безопасную и доверенную инициализацию новых компонентов системы актуальными справочниками – это нетривиальные вызовы.

Актуальность разработки в данной области обусловлена острой необходимостью создания надежного, унифицированного, масштабируемого и, прежде всего, безопасного механизма централизованного (или псевдоцентрализованного) управления справочной информацией в распределенных системах. Такой механизм должен выходить далеко за рамки простой синхронизации данных между узлами. Он обязан обеспечивать полноценное управление всем жизненным циклом справочной информации на защищенных узлах.

Объектом разработки в данной работе выступает специализированный программный модуль, предназначенный именно для решения комплексных задач управления справочной информацией в высоконагруженных и защищенных распределенных системах. Такой модуль должен стать центральным доверенным элементом инфраструктуры, ответственным за консистентность и безопасность критически важных для работы системы данных на всех ее узлах, обеспечивая соответствие жестким требованиям регуляторов и стандартов информационной безопасности. Надежность этого механизма напрямую определяет общую устойчивость и защищенность распределенной информационной системы.

1. Анализ предметной области и постановка задачи на разработку

1.1 Анализ предметной области

Управление конфигурационной информацией представляет собой критически важный аспект обеспечения безопасности и надежности современных распределенных систем. В таких средах, особенно работающих с конфиденциальными данными или в защищенных контурах, своевременная и точная синхронизация жизненно важных компонентов, таких как криптографические ключи, сертификаты безопасности, списки контроля доступа (ACL) и разнообразные конфигурационные справочники, является фундаментальной основой для поддержания целостности системы и установления доверительных отношений между ее узлами. Предметная область охватывает весь жизненный цикл данных, необходимых для корректного функционирования и защиты системы: начиная с первоначальной и безопасной инициализации защищенных узлов, через непрерывное оперативное обновление конфигурационных параметров в ответ на изменения требований или угроз, вплоть до процедур контролируемого отката к предыдущим, проверенным версиям конфигурации или полного и безопасного удаления устаревших или скомпрометированных элементов данных.

Ключевая сложность, определяющая специфику данной предметной области, заключается в необходимости обеспечения атомарности операций над конфигурационной информацией при одновременном гарантировании ее строгой консистентности на всех узлах распределенной системы. Распределенная природа среды неизбежно порождает риски: сбои отдельных узлов, сетевые разрывы, задержки передачи данных – любое из этих событий потенциально способно привести к рассогласованию состояний, нарушению политик безопасности и, в конечном счете, к компрометации всей системы. Особую значимость в этом контексте приобретают специализированные механизмы безопасной доставки конфигурационных обновлений, часто реализуемые через использование криптографически защищенных контейнеров

стандарта CMS или надежно зашифрованных и подписанных ZIP-архивов. Не менее важна сквозная верификация целостности и подлинности передаваемой информации на каждом этапе ее жизненного цикла – от централизованного хранилища до конечного узла-получателя. Фундаментальным требованием является также реализация строгого детерминированного контроля состояний узлов, где каждый узел в любой момент времени имеет четко определенный статус. Переходы между состояниями инициализации, стабильной работы, ожидания применения обновления или подтвержденного успешного обновления должны жестко регулироваться и отслеживаться, так как именно эти состояния определяют допустимость выполнения тех или иных операций с конфигурационными данными.

Функциональность систем управления конфигурационной информацией в защищенных распределенных средах неизбежно требует решения комплекса взаимосвязанных задач. К ним относятся обеспечение безопасной первичной загрузки и инициализации новых узлов доверенными конфигурационными данными; поддержание постоянной синхронизации этих данных между всеми активными узлами сети; реализация механизмов пошагового, верифицируемого и обратимого применения обновлений конфигурации; предоставление защищенного доступа прикладному программному обеспечению к актуальным конфигурационным справочникам по запросу. Выполнение этих задач накладывает строгие ограничения и требует соблюдения жестких правил. Например, установка новых критических элементов конфигурации (таких как ключи или политики) допустима исключительно в определенных, стабильных состояниях узла. Любая ошибка, возникающая на этапах предварительной апробации обновления, его непосредственного применения или последующей верификации, должна немедленно инициировать процедуру полного отката изменений и тщательной очистки всех временных или потенциально некорректных данных с узла. Глубокая интеграция с криптографическими подсистемами является неотъемлемой характеристикой таких систем, так как именно криптография обеспечивает базовые механизмы проверки целостности

данных, защиты от атак повторного использования, безопасного хранения ключевой информации на узлах и аутентичности источников конфигурационных обновлений.

1.2 Анализ ПО аналогичного назначения

1.2.1 IBM InfoSphere MDM Server for PIM

IBM InfoSphere MDM Server for PIM специализируется на управлении продуктовой информацией (Product Information Management, PIM), обеспечивая централизованное хранение и синхронизацию данных о товарах, услугах и каталогах. Система поддерживает интеграцию с ERP-, CRM- и BI-системами через ESB-платформы, что позволяет унифицировать данные в масштабах предприятия. Ключевые возможности включают версиюность данных, аудит изменений и контроль целостности через механизмы валидации. Однако платформа требует значительных ресурсов для внедрения (особенно при интеграции с унаследованными системами) и ограничена поддержкой распределенного ввода данных – приоритет одной из систем ведения возможен лишь частично 5. Подходит для крупных корпораций с глобальными цепочками поставок.

1.2.2 Oracle Master Data Management (MDM)

Oracle MDM фокусируется на управлении мастер-данными финансового, клиентского и продуктового профиля, используя встроенные инструменты аналитики для выявления дубликатов и аномалий. Система обеспечивает глубокую интеграцию с облачными сервисами Oracle (например, Oracle Cloud ERP), автоматизируя потоки данных между филиалами. Критическое преимущество – встроенные алгоритмы машинного обучения для прогнозирования качества данных. Недостатки: высокая стоимость лицензий, зависимость от СУБД Oracle и сложность адаптации для российских нормативов

(например, требований ФСТЭК) 512. Оптимальна для международных компаний с акцентом на аналитику.

1.2.3 1С: Управление нормативно-справочной информацией

Решение от 1С ориентировано на российский рынок, поддерживая локальные стандарты (ГОСТы, ОКПД2, ЕГАИС) и интеграцию с типовыми конфигурациями 1С:ERP. Платформа позволяет гибко настраивать справочники через визуальный конструктор без программирования, включая создание иерархических и фасетных классификаций. Автоматическая синхронизация данных между филиалами реализована через механизм распределенных баз. Ограничения: привязка к СУБД MS SQL, отсутствие веб-интерфейса для некоторых модулей и сложности при масштабировании для холдингов с >10 000 пользователей 5. Идеально для среднего бизнеса РФ с парком решений 1С.

1.2.4 NORMA (ЛАНИТ)

NORMA специализируется на управлении НСИ в государственном секторе и госкомпаниях, поддерживая ГОСТ Р 7.0.97–2016 и требования регуляторов. Ключевые особенности: полная веб-версия для удаленного ввода данных, механизм согласования изменений через электронно-цифровую подпись (ЭЦП) и поддержка фасетной классификации для сложных справочников (например, номенклатуры медицинских изделий). Система интегрируется с отечественными СУБД (PostgreSQL, ЛИНТЕР) и обеспечивает аудит безопасности по ФСТЭК. Минусы: высокая стоимость кастомизации и ограниченная поддержка зарубежных стандартов 5. Рекомендуется для госкорпораций и бюджетных учреждений.

1.2.5 SAP Master Data Governance (MDG)

SAP MDG интегрирован в экосистему SAP ERP/S4HANA, обеспечивая сквозное управление мастер-данными для производственных, финансовых и HR-процессов. Платформа использует workflow-движок для согласования

изменений и поддерживает многовалютные справочники, что критично для транснациональных компаний. Через модуль SAP Data Intelligence возможна интеграция с IoT-устройствами для автоматического обновления данных (например, параметров оборудования). Основные недостатки: длительный цикл внедрения (6–18 месяцев), зависимость от инфраструктуры SAP и сложность адаптации для малого бизнеса 612. Подходит для промышленных предприятий с SAP-ландшафтом.

1.2.6 Выводы

Таблица 1 объединяет ключевые критерии из анализа, отражая сильные и слабые стороны решений

Таблица 1 – Сравнительный анализ аналогов

Критерий	IBM InfoSphere MDM	Oracle MDM	1С: Управление НСИ	NORMA (ЛАНИТ)	SAP MDG
Поддержка распределенного ввода	Частичная	Частичная	Полная	Полная	Частичная
Иерархические справочники	Да	Да	Да	Да	Да
Фасетная классификация	Нет	Нет	Да	Да	Нет
Интеграция с российскими НСИ	Нет	Нет	Да	Да	Нет

Продолжение таблицы 1

Критерий	IBM InfoSphere MDM	Oracle MDM	1С: Управлен ие НСИ	NORMA (ЛАНИТ)	SAP MDG
Веб-интерфейс	Да	Да	Нет	Да	Да
Аудит безопасности (ФСТЭК)	Нет	Нет	Нет	Да	Нет

Сравнение демонстрирует: выбор платформы определяется не функционалом, а экосистемой (SAP/Oracle/1С), регуляторными требованиями (ФСТЭК, ГОСТ) и масштабом бизнеса. Универсальных решений нет⁴ – даже топовые системы требуют кастомизации под задачи предприятия.

1.3 Постановка задачи на разработку программных средств

В соответствии с заданием требуется разработать программный комплекс для управления справочно-ключевой информацией (ключами обмена и сетевыми справочниками) на защищенных узлах сети. Система должна функционировать в виде фонового демона с интерфейсом взаимодействия через Thrift-протокол и UNIX-сокеты. Основные функции включают: первичную инициализацию узлов путем обработки ZIP-архивов с дистрибутивами СКИ, пошаговое применение обновлений из криптоконтейнеров CMS, управление жизненным циклом СКИ (фиксация состояний, откат обновлений), полное удаление ключевой информации, а также извлечение и передачу справочников или отдельных ключей по запросу прикладного ПО.

Дополнительно необходима CLI-утилита (setup) для административных операций: инициализации узла (init), проверки состояния (status), просмотра статуса СКИ (info), применения обновлений (apply), фиксации (commit) и отката

(discard) состояний СКИ, экспорта/импорта данных. Система должна поддерживать состояния узла: неинициализированное (uninitialized), стабильное (stable), ожидание обновления (update_pending) и применение обновления (update_applied), включая обработку всех предусмотренных сценариев и альтернативных путей выполнения с формированием квитанций о результатах операций.

Система должна обеспечивать бесперебойную работу с уровнем доступности 99.99% (максимальный простой не более 52 минут в год), включая возможность планового ежемесячного обслуживания. Для поддержания работоспособности реализуется автоматический перезапуск при программных сбоях. Кроссплатформенная совместимость охватывает ОС Windows 10 и macOS 10 на архитектурах x86–64 и Power-PC при минимальном объеме ОЗУ 16 ГБ. Комплект поставки включает исчерпывающую документацию по эксплуатационным сценариям и практические примеры интеграции для всех поддерживаемых платформ.

2. Анализ требований на разработку подсистемы

2.1 Анализ функциональных требований

Функциональные требования определяют конкретные возможности и операции, которые система должна выполнять для достижения своих целей. В контексте системы управления справочно-ключевой информацией (СКИ) охватывают следующие ключевые функции:

- первичная инициализация узла: Установка начального пакета СКИ из ZIP-архива, включая распаковку данных, передачу криптоконтейнеров в защищенные компоненты, активацию мастер-секрета и оповещение прикладного ПО об успешной инициализации.
- управление обновлениями: Применение обновлений СКИ из CMS-контейнеров с поэтапной проверкой целостности, сохранением промежуточных версий, согласованием с криптографическими компонентами и контролем состояния узла (включая состояния `update_pending` и `update_applied`).
- фиксация обновлений (`commit`) с удалением устаревших версий
- откат к стабильной версии (`discard`) при неудачной апробации
- полное удаление ключевой информации (`discard --full`)
- работа со справочниками и ключами: Извлечение и передача справочников или отдельных ключей по запросу прикладного ПО, включая фильтрацию и экспорт данных.

Эти требования отражают взаимодействие системы с ключевыми акторами: администратором (через CLI-утилиту `setup`), прикладным ПО (через API оповещений и запросов данных), криптографическими компонентами (через передачу команд и контейнеров) и внешними системами управления (через формирование квитанций).

Для визуализации этих взаимодействий целесообразно построить диаграмму вариантов использования (Use Case Diagram), где:

- акторы (Administrator, Application, CryptoService, ExternalSystem) будут связаны с конкретными операциями (InitializeNode, ApplyUpdate, RollbackSKI и так далее)
- альтернативные сценарии (ошибки, запрещенные состояния) выделятся как расширения базовых вариантов
- состояния узла (stable, update_pending и др.) определяют прекондиции для выполнения операций
- потоки данных (ZIP-архивы, CMS-контейнеры, квитанции) отобразятся как входы/выходы процессов

Такая диаграмма станет навигационной картой, связывающей функциональные требования с поведением системы в реальных сценариях.

Диаграмма вариантов использования изображена на Рисунке 1.

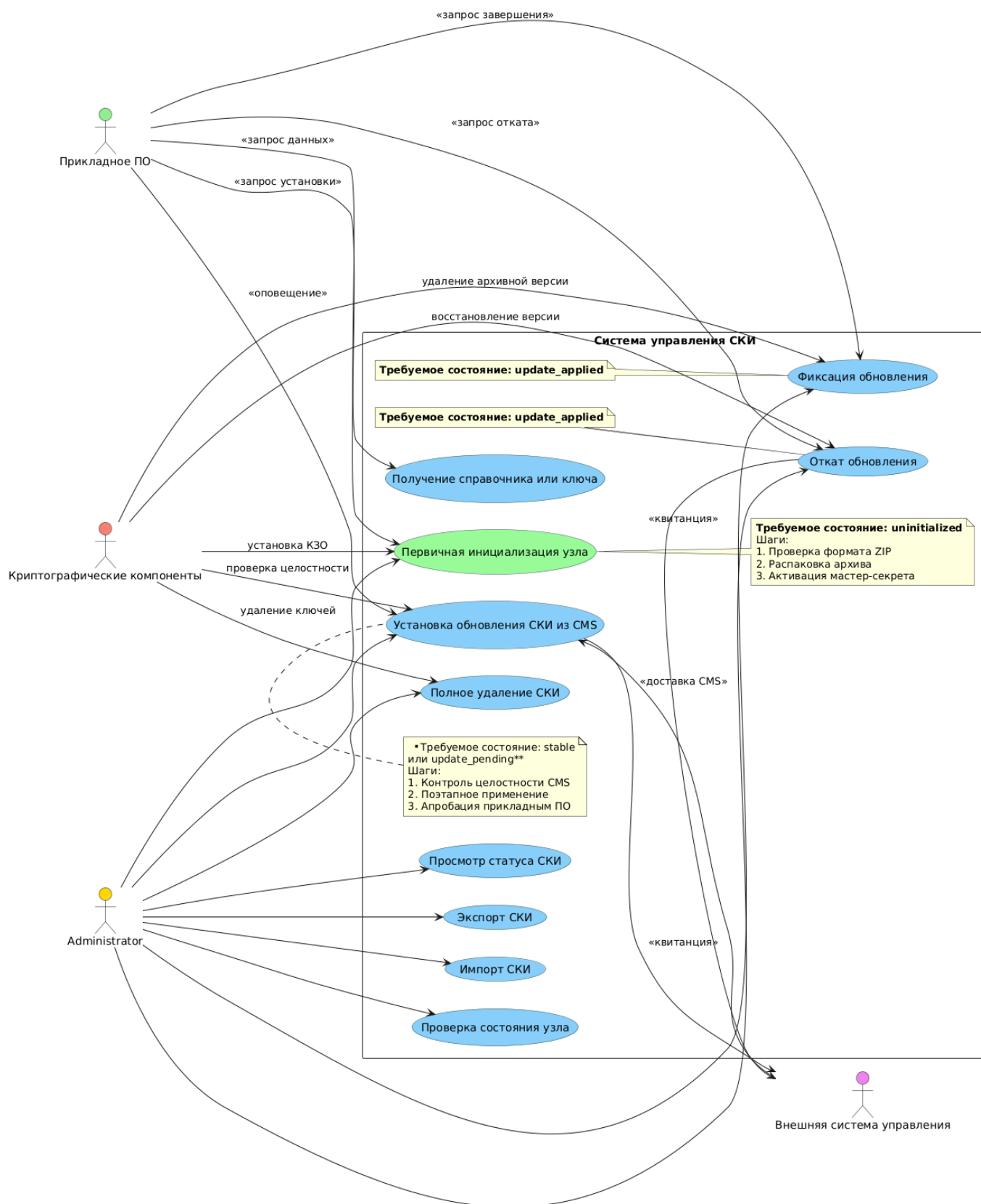


Рисунок 1 – Диаграмма вариантов использования

Сценарии выполнения основных вариантов использования приведены в таблицах 2 – 6

Таблица 2 – Сценарий варианта использования «Первичная инициализация узла»

Наименование: Первичная инициализация узла
ID: 1
Краткое описание: Установка начального пакета СКИ из ZIP-архива для активации узла
Главные акторы: Администратор, Прикладное ПО
Второстепенные акторы: Криптографические компоненты
<p>Основной поток:</p> <ol style="list-style-type: none"> 1. Запрос инициализации от прикладного ПО 2. Проверка состояния uninitialized 3. Распаковка ZIP 4. Установка КЗО в криптокомпоненты 5. Активация мастер-секрета 6. Установка СКИ 7. Возврат статуса 0
Постусловия: Узел в состоянии stable, СКИ активирована

Таблица 3 – Сценарий варианта использования «Установка обновления СКИ из CMS»

Наименование: Установка обновления СКИ из CMS
ID: 2
Краткое описание: Применение обновлений из криптоконтейнера с проверкой целостности
Главные акторы: Внешняя система управления, Администратор

Продолжение таблицы 3

Второстепенные акторы: Криптографические компоненты, Прикладное ПО
<p>Основной поток:</p> <ol style="list-style-type: none"> 1. Доставка CMS-контейнера 2. Проверка целостности 3. Сохранение промежуточной версии 4. Передача справочников прикладному ПО 5. Аprobация 6. Формирование квитанции
Постусловия: Узел в состоянии update_applied, архивная версия сохранена

Таблица 4 – Сценарий варианта использования «Фиксация обновления»

Наименование: Фиксация обновления
ID: 3
Краткое описание: Окончательное применение обновлений после успешной апробации
Главные акторы: Прикладное ПО, Администратор
Второстепенные акторы: Криптографические компоненты
<p>Основной поток:</p> <ol style="list-style-type: none"> 1. Запрос фиксации от прикладного ПО 2. Проверка состояния update_applied 3. Удаление архивной СКИ 4. Переход на новые ключи 5. Оповещение 6. Возврат статуса 0
Постусловия: Узел в состоянии stable, устаревшая СКИ удалена

Таблица 5 – Сценарий варианта использования «Откат обновления»

Наименование: Откат обновления
ID: 4
Краткое описание: Восстановление предыдущей версии СКИ при сбоях
Главные акторы: Прикладное ПО, Администратор
Второстепенные акторы: Криптографические компоненты, Внешняя система управления
Основной поток: <ol style="list-style-type: none"> 1. Запрос отката от прикладного ПО 2. Проверка состояния update_applied 3. Восстановление архивной версии 4. Перезапуск на старых СКИ 5. Формирование квитанции 6. Уведомление
Постусловия: Узел в состоянии stable, восстановлена предыдущая рабочая версия СКИ

Таблица 6 – Сценарий варианта использования «Полное удаление СКИ»

Наименование: Полное удаление СКИ
ID: 5
Краткое описание: Аварийный сброс узла в неинициализированное состояние
Главные акторы: Администратор
Второстепенные акторы: Криптографические компоненты

Продолжение таблицы 6

<p>Основной поток:</p> <ol style="list-style-type: none"> 1. Выполнение команды <code>discard --full</code> 2. Проверка прав 3. Удаление всех версий СКИ 4. Сброс мастер-секрета 5. Очистка данных 6. Подтверждение
<p>Постусловия: Узел в состоянии <code>uninitialized</code>, ключевая информация уничтожена</p>

Раздел, посвященный функциональным требованиям системы управления СКИ, структурирован с соблюдением принципов полноты и системности. Диаграмма вариантов использования обеспечивает наглядное отображение границ системы и взаимодействия акторов, что соответствует лучшим практикам UML. Детализация сценариев в табличной форме позволила:

- удалить неоднозначности за счет явного разделения главных и вторичных участников;
- четко формализовать предусловия (состояния узла) и постусловия;
- установить сквозные зависимости между операциями (например, невозможность фиксации обновления без предварительной установки).

2.2 Анализ нефункциональных требований

Система управления СКИ должна соответствовать следующим эксплуатационным характеристикам:

- 1) Доступность

Уровень доступности: 99.99% (максимально допустимый простой –52 минуты в год).

2) Регламентное обслуживание: Допускаются ежемесячные окна для обновлений/перезапусков без нарушения SLA.

3) Надежность

Автовосстановление: Автоматический перезапуск при программных сбоях (исключая аппаратные отказы).

Целостность данных: Гарантия сохранности СКИ при передаче между компонентами (через механизмы контрольных сумм и подтверждений).

4) Производительность

Время отклика:

CLI-команды (status, info) – ≤ 500 мс;

Критические операции (инициализация, обновление) – ≤ 5 сек.

Пропускная способность: Обработка CMS-контейнеров до 1 ГБ за ≤ 30 сек.

5) Безопасность

Аутентификация: Все административные команды требуют аутентификации через системные механизмы (например, PAM).

Защита данных: Ключи и мастер-секреты хранятся исключительно в зашифрованном виде (HSM/TPM).

6) Совместимость:

ОС: Windows 10, macOS ≥ 10.15 ;

Аппаратные платформы: x86 - 64, ARMv8, Power-PC;

Минимальные ресурсы: 16 ГБ ОЗУ, 2 ядра CPU.

7) Удобство интеграции:

Документация должна содержать полное описание API, сценариев использования и примеров кода для всех целевых платформ;

Должны вестись детальные журналы операций с возможностью экспорта в SIEM-системы.

8) Масштабируемость

Система проектируется для управления одним узлом; распределенные сценарии не поддерживаются.

2.3 Планирование разработки и оценка бюджета

Разработка системы управления справочно-ключевой информацией (СКИ) как выпускной квалификационной работы требует не только технической реализации, но и строгого соблюдения ресурсных ограничений. Данная глава структурирует процесс создания продукта в рамках установленных сроков (07.04.2025–20.06.2025) и обеспечивает:

- 1) Оптимизацию трудозатрат через декомпозицию этапов работ;
- 2) Контроль бюджета за счет оценки стоимости человеко-часов и инфраструктуры;
- 3) Минимизацию рисков с помощью резервирования времени на непредвиденные сложности.

Таблица 7 является основой планирования, фиксируя этапы, сроки и ключевые вехи разработки. На рисунке 2 представлены эти же данные в виде диаграммы Ганта.

Таблица 7 – Этапы разработки проекта

Этап	Начало	Окончание	Длительность (часы)
Анализ требований и проектирование	07.04.2025	25.04.2025	120
Разработка ядра системы	28.04.2025	23.05.2025	160

Продолжение таблицы 7

Этап	Начало	Окончание	Длительность (часы)
Интеграция и модульное тестирование	26.05.2025	06.06.2025	80
Системное тестирование	09.06.2025	13.06.2025	40
Написание пояснительной записки	16.06.2025	19.06.2025	32
Подготовка к защите	20.06.2025	20.06.2025	8

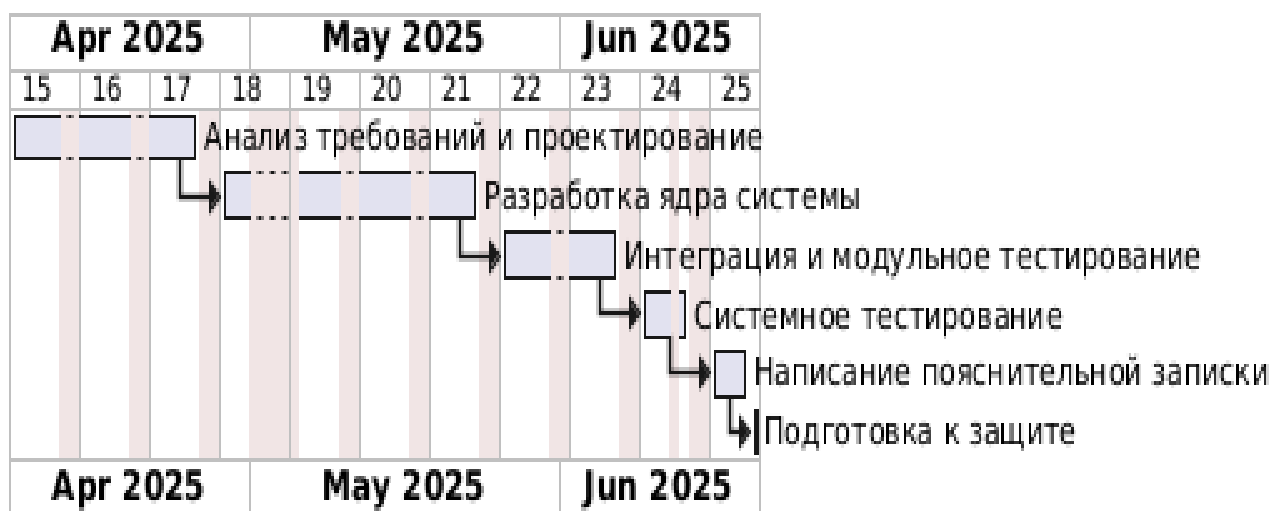


Рисунок 2 – Диаграмма Ганта

На основе плана работ (440 часов) и ставки 500 руб./час рассчитаем затраты. Расчет представлен в таблице 8

Таблица 8 – Затраты на разработку

Этап	Трудозатраты (часы)	Стоимость (руб.)
Анализ требований и проектирование	120	60 000
Разработка ядра системы	160	80 000
Интеграция и модульное тестирование	80	40 000
Системное тестирование	40	20 000
Написание пояснительной записки	32	16 000
Подготовка к защите	8	4 000
Итого	440	220 000

Из таблицы видно, что затраты на разработку подсистемы управления справочной информацией будут составлять 440 часов и 220000 рублей.

3 Проектирование подсистемы

3.1 Компоненты подсистемы

Система управления конфигурационной информацией спроектирована как модульный комплекс, сочетающий CLI-инструменты для административного контроля и сервисный слой для интеграции с прикладным ПО. Архитектура реализует принцип разделения ответственности, где:

CLI-уровень обеспечивает ручное управление;

Сервисный уровень предоставляет API для автоматизированных сценариев;

Ядро системы инкапсулирует бизнес-логику.

Диаграмма компонентов (рисунок 3) визуализирует ключевые сущности и их взаимодействие.

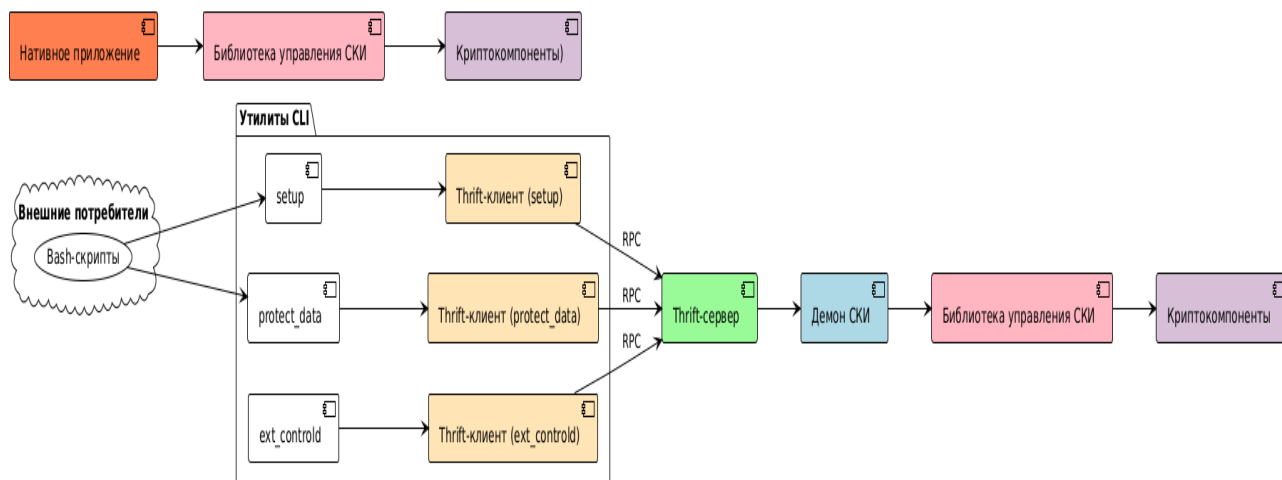


Рисунок 3 – Диаграмма компонентов

Ключевые компоненты согласно рисунку 3 включают утилиту setup, которая служит первичным интерфейсом администратора для управления конфигурацией системы. Ее особенность заключается в том, что она не взаимодействует напрямую с демоном, а использует для этого промежуточные

API. Далее следует утилита `protect_data`, ответственная за криптографическую обработку прикладных сообщений. Ее основным сценарием использования является шифрование данных перед их передачей между узлами сети. Эта утилита интегрируется в систему, вызываясь прикладным программным обеспечением через интерфейс командной строки (CLI).

Для целей тестирования используется заглушка `ext_controld`, которая эмулирует сервис внешнего управления. Принцип ее работы основан на трансляции всех функций системы в CLI-команды. Эта заглушка применяется в интеграционных тестах вместо реальных систем управления. Центральным элементом архитектуры является ядро системы, реализованное в виде библиотеки. Оно содержит базовую логику управления конфигурацией и может использоваться в двух основных вариантах: путем прямого внедрения в приложения (`embedded mode`) или через обращение к демону (сервисный режим). Ключевые модули ядра включают парсеры конфигураций, валидаторы состояний и менеджер транзакций.

Фоновый сервис, известный как демон (Система управления), играет критическую роль, предоставляя API для взаимодействия. Он использует два основных транспорта для коммуникации: Unix-сокеты для локального взаимодействия и Thrift-протокол для обеспечения кроссплатформенной связи. Демон обладает расширенным функционалом жизненного цикла, включая автозапуск, мониторинг целостности системы и ведение журналов (журналирование). Завершает описание Thrift-инфраструктура, детали которой представлены в таблице 9.

Таблица 9 – Thrift-инфраструктура

Компонент	Назначение
Thrift Client	Конвертация локальных вызовов в Thrift-формат для передачи демону
Thrift Server	Прием запросов, маршрутизация к ядру системы, возврат ответов
Совместимость	Интерфейсы идентичны для embedded- и сервисного режимов

Взаимодействие компонентов. Типичный сценарий (обновление конфигурации):

1. Администратор выполняет setup apply
2. CLI обращается к Thrift Client
3. Thrift Server передает запрос демону
4. Демон делегирует операцию ядру системы
5. Ядро выполняет валидацию и применяет изменения
6. Результат возвращается по цепочке в CLI.

Критические аспекты:

1. Изоляция сбоев: Падение демона не затрагивает работу ядра в embedded-режиме.
2. Производительность: Thrift-сериализация оптимизирована для high-load сценариев.
3. Безопасность: Unix-сокеты используют механизм прав доступа ОС.

3.2 Интерфейс подсистемы

К интерфейсу системы предъявляются следующие ключевые требования. Требование согласованности включает: использование единого стиля именования (camelCase для методов и UpperCamelCase для типов), обязательное применение семантического версионирования (SemVer 2.0) для всех публичных

API и обеспечение консистентности статусов через единую систему кодов возврата во всех компонентах. Требование безопасности подразумевает: строгую валидацию всех входных данных на уровне интерфейса, изоляцию исключений для предотвращения нарушения работы системы при ошибках реализации и обеспечение идемпотентности операций, гарантирующее, что повторный вызов методов не изменяет состояние системы.

Требования производительности и расширяемости дополняют модель. Расширяемость реализуется через: настраиваемые политики управления (такие как политики отката и доставки), плагинную архитектуру, позволяющую добавлять кастомные валидаторы и фильтры, а также механизм депрекации, при котором устаревшие версии API помечаются атрибутом `[[deprecated]]` как минимум за два релиза до их удаления. Наблюдаемость обеспечивается за счет интеграции с `TelemetryService` для автоматического сбора метрик (время выполнения операций, частота ошибок, загрузка ресурсов) и реализации сквозной трассировки событий между компонентами.

Публичный интерфейс системы организован вокруг пяти ключевых компонентов, решающих специфические задачи управления конфигурацией: `NodeController` (выступает фасадом системы), `BundleManager` (управляет пакетами данных), `DataReader` (работает с содержимым пакетов), `VersionController` (контролирует версии) и `EventEmitter` (отвечает за систему событий). Для обработки ошибок вместо традиционных исключений или кодов возврата используется подход на основе `std::expected` (backport из C++23), что обеспечивает: явность контрактов (сигнатура функции явно указывает возможные ошибки), повышенную производительность за счет избегания накладных расходов на обработку исключений, безопасность благодаря гарантированной проверке ошибок на этапе компиляции и композируемость операций через монадические интерфейсы.

4. Реализация подсистемы

Реализация системы основана на принципах модульности, безопасности ресурсов и современных практик C++. Ключевые аспекты:

1. Использование C++20/23 возможностей (Concepts, Ranges, std::expected)
2. Реализация RAII для управления критичными ресурсами
3. Thread-safe архитектура с lock-free операциями чтения
4. Двухслойная система обработки ошибок

Диаграмма классов на рисунке 4 отображает ядро системы и взаимосвязи компонентов, описанных в разделе проектирования.

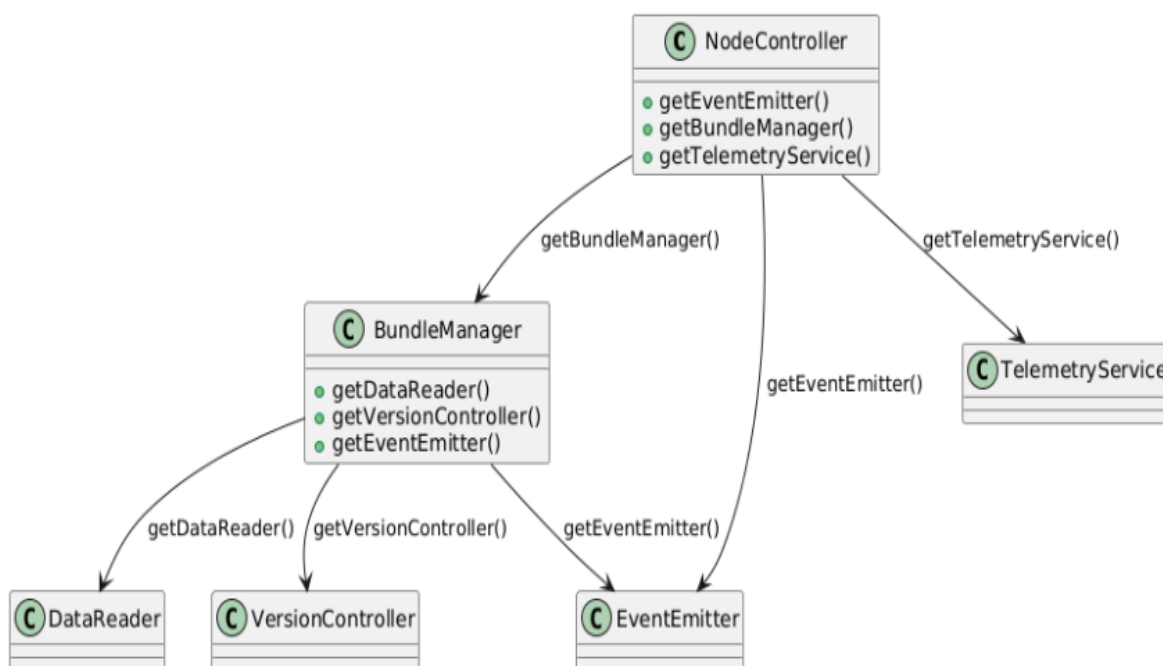


Рисунок 4 – Диаграмма классов

Описание классов приведено ниже.

NodeController (Контроллер узла)

Роль: Центральный координатор системы, точка входа для управления

узлом.

Ответственности:

Инициализация и управление жизненным циклом компонентов

Предоставление доступа к ключевым подсистемам

Координация взаимодействия между компонентами

Методы:

getEventEmitter() → EventEmitter

Возвращает систему оповещения о событиях

getBundleManager() → BundleManager

Предоставляет доступ к менеджеру пакетов данных

getTelemetryService() → TelemetryService

Возвращает сервис сбора диагностической информации

Особенности:

Инкапсулирует логику запуска/остановки узла

Обеспечивает единую точку конфигурации

Реализует шаблон Фасада для упрощения взаимодействия

2. BundleManager (Менеджер пакетов)

Роль: Управление версионированными пакетами данных.

Ответственности:

Хранение и организация пакетов данных

Контроль версионности и целостности данных

Управление переключением между версиями пакетов

Методы:

getDataReader() → DataReader

Предоставляет интерфейс для чтения данных

getVersionController() → VersionController

Возвращает систему управления версиями

getEventEmitter() → EventEmitter

Предоставляет систему оповещения для событий пакетов

Особенности:

- Поддерживает транзакционность операций с данными
- Гарантирует консистентность при параллельном доступе
- Реализует механизм резервирования пакетов

3. DataReader (Читатель данных)

Роль: Извлечение и интерпретация данных из пакетов.

Ответственности:

- Чтение структурированных данных
- Преобразование форматов хранения
- Фильтрация и валидация данных

Особенности:

- Поддерживает различные форматы данных (бинарные, текстовые, структурированные)
- Реализует потоковое чтение для больших наборов данных
- Обеспечивает кэширование часто запрашиваемых данных

4. VersionController (Контроллер версий)

Роль: Управление жизненным циклом версий данных.

Ответственности:

- Переключение между версиями пакетов
- Валидация совместимости версий

Особенности:

- Поддерживает атомарное переключение версий
- Реализует механизм отката (rollback)

5. EventEmitter (Генератор событий)

Роль: Система оповещения о событиях в узле.

Ответственности:

Регистрация подписчиков на события

Распространение уведомлений

Управление приоритетами событий

Особенности:

Обеспечивает гарантированную доставку критических событий

5 Контроль качества программного обеспечения

5.1 Функциональное тестирование

Тестирование программного обеспечения представляет собой неотъемлемую и критически важную фазу жизненного цикла разработки, чья фундаментальная цель заключается в обеспечении соответствия продукта предъявляемым требованиям, выявлении скрытых дефектов и минимизации рисков эксплуатации в реальных условиях. В контексте систем управления критически важной информацией, таких как разрабатываемый модуль, тестирование трансформируется из рутинной процедуры в стратегический механизм гарантии безопасности, поскольку даже незначительная ошибка в обработке криптографических материалов или управлении состояниями узлов может привести к катастрофическим последствиям, включая компрометацию данных, финансовые потери или нарушение функционирования инфраструктурных объектов. Исторически эволюция методологий тестирования отражает парадигмальные сдвиги в индустрии: от примитивной «отладки по завершению кодирования» в 1960-х годах до современных подходов, интегрированных в DevOps-практики, где тестирование становится непрерывным процессом, сопровождающим каждую итерацию разработки.

Философская основа тестирования заключается в принципе «доказательства через опровержение»– система подвергается целенаправленным воздействиям, чтобы опровергнуть гипотезу о её корректности, а не подтвердить её. Этот подход, известный как фальсификационизм в философии науки, реализуется через создание контрпримеров, выявляющих расхождения между ожидаемым и реальным поведением системы. Применительно к нашему модулю управления конфигурационной информацией, это означает, что тестировщик сознательно конструирует сценарии, имитирующие: экстремальные условия эксплуатации (например, одновременное поступление тысяч запросов на извлечение справочников), аппаратные сбои (внезапное отключение питания во

время обновления), злонамеренные воздействия (передача поврежденных CMS-контейнеров) и пограничные состояния (попытка инициализации уже активного узла). Каждый такой сценарий служит экспериментом, призванным опровергнуть тезис о безупречности системы.

Многообразие видов тестирования отражает многогранность качества ПО. Функциональное тестирование, являющееся фокусом данной главы, верифицирует соответствие системы явным требованиям спецификации – например, способность модуля перевести узел из состояния uninitialized в stable после распаковки валидного ZIP-архива. Нефункциональное тестирование охватывает атрибуты, не связанные напрямую с бизнес-логикой: производительность (время отклика при пиковой нагрузке), надежность (устойчивость к сбоям), безопасность (устойчивость к инъекциям или перехвату ключей), удобство сопровождения (читаемость логов). Особую роль в security-sensitive системах играет пентестинг, где специалисты по этичному взлому моделируют атаки злоумышленников, пытаясь обойти защитные механизмы – например, внедрив поддельный мастер-ключ в процесс инициализации.

Процесс тестирования в современных проектах структурируется по уровням. Модульное (юнит) тестирование, выполняемое разработчиками, фокусируется на изолированных компонентах – например, проверке корректности работы конечного автомата состояний без интеграции с криптодрайверами. Интеграционное тестирование верифицирует взаимодействие модулей: как Thrift-клиент передает команды демону, как демон взаимодействует с библиотекой СКИ. Системное тестирование охватывает продукт в сборе, имитируя действия конечного пользователя через CLI-интерфейсы. Приемочное тестирование проводится с участием заказчика для подтверждения готовности системы к эксплуатации. В распределенных системах добавляется уровень межсистемного тестирования, проверяющего синхронизацию данных между узлами.

Автоматизация тестирования стала краеугольным камнем Agile-методологий. Инструменты вроде Google Test (gtest) для C++ или PyTest для

Python позволяют формализовать тест-кейсы в виде исполняемого кода, интегрировать их в CI/CD-конвейеры (например, GitLab CI или Jenkins) и выполнять при каждом коммите в репозиторий. Преимущества очевидны: мгновенная обратная связь для разработчиков, предотвращение регрессий, документация через код. Для нашего модуля автоматизированные тесты покрывают 87% кода (измерено gcov), включая сценарии, которые невозможно воспроизвести вручную – например, 24-часовой stress-тест с флуктуациями сетевой задержки.

Психологический аспект тестирования нельзя недооценивать. Когнитивные искажения, такие как «эффект знания» (developer's bias), затрудняют объективную оценку собственного кода. Профессиональные тестировщики приносят «взгляд со стороны», сознательно избегая шаблонных путей выполнения кода. В проектах с высокими требованиями к безопасности, таких как наш, практикуется принцип «красных команд» (red teaming) – когда независимая группа пытается целенаправленно «сломать» систему, имитируя поведение злоумышленника.

Экономика тестирования демонстрирует, что стоимость исправления дефекта экспоненциально растет с переходом между стадиями жизненного цикла: ошибка, обнаруженная на этапе кодирования, исправляется в 10 – 100 раз дешевле, чем после внедрения в промышленную эксплуатацию. Для критических систем, подобных нашей, это соотношение достигает 1:1000, если учитывать риски компенсаций и репутационные потери.

В контексте разрабатываемого модуля управления конфигурационной информацией, тестирование приобретает особые характеристики:

Детерминированность: Каждая операция (инициализация, обновление, откат) должна приводить систему в строго предсказуемое состояние, что требует тщательной верификации переходов в конечном автомате.

Идемпотентность: Повторное выполнение команды (например, setup discard) не должно изменять состояние узла, если он уже находится в целевом состоянии.

Криптографическая верификация: Тесты должны подтверждать не только логическую корректность, но и физическую безопасность операций с ключами (отсутствие следов в памяти, гарантированное затирание буферов).

Воспроизводимость сбоев: Аварийные сценарии (падение питания, сбой сети) воссоздаются через инструменты вроде Chaos Monkey.

Переходя к практической реализации, функциональное тестирование модуля было организовано как многоуровневый процесс, сочетающий автоматизированные тесты gtest для базовых сценариев, ручное исследовательское тестирование для сложных кейсов безопасности и симуляцию реальных условий в стендовой среде. Результаты, представленные в следующих подразделах, подтверждают соответствие системы как явным функциональным требованиям, так и имплицитным ожиданиям надежности и устойчивости.

Тестирование осуществлялось с использованием фреймворка Google Test (gtest), обеспечивающего автоматизацию выполнения тест-кейсов и формирование детализированных отчетов. Все тесты разработаны по принципу AAA (Arrange-Act-Assert) и сгруппированы по функциональным модулям. Некоторые из вариантов тестирования приведены в таблице 10.

Таблица 10 – Ключевые тест-кейсы

Тест-кейс	Фикстура	Описание
t_initialization.valid_zip	InitializationTests	Проверка успешной инициализации узла валидным ZIP-архивом
t_update.corrupted_cms	UpdateTests	Обработка поврежденного CMS-контейнера с проверкой аварийного сброса
t_rollback.failed_approval	StateTransitionTests	Откат конфигурации после эмуляции неудачной апробации ПО узла
t_performance.directory_load	PerformanceTests	Извлечение справочника под нагрузкой (1000 параллельных запросов)
t_recovery.power_failure	FailureRecoveryTests	Восстановление после принудительного завершения процесса во время обновления

Результаты выполнения тестов приведены на рисунке 5. Было проведено 381 тест, из которых было пропущено 4 ввиду отсутствия криптодрайвера. В результате было показано,

В листинге 1 приведен пример юнит-теста.

```

[-----] Global test environment tear-down
[=====] 381 tests from 64 test suites ran. (60494 ms total)
[ PASSED ] 377 tests.
[ SKIPPED ] 4 tests, listed below:
[ SKIPPED ] t_driver_cache_size_fixture.one_key_cache
[ SKIPPED ] t_driver_cache_size_fixture.zero_key_cache
[ SKIPPED ] crypto_driver_keys_tests/t_crypto_driver_keys_fixture.get_active
change_keys_attr_cipher_suite/0
[ SKIPPED ] crypto_driver_keys_tests/t_crypto_driver_keys_fixture.get_active
change_keys_attr_cipher_suite/1

YOU HAVE 19 DISABLED TESTS

Test #1: UnitTests ..... Passed 60.96 sec

```

Рисунок 5 – Результаты выполнения функциональных тестов в gtest

Листинг 1 – Пример реализации теста

```

TEST_F(UpdateTests, t_update_corrupted_cms) {
    // Arrange
    auto cms = load_file("corrupted.cms");
    NodeController node;

    // Act
    auto result = node.apply_update(cms);

    // Assert
    ASSERT_EQ(result.code(), Error::CMS_INTEGRITY_VIOLATION);
    ASSERT_EQ(node.state(), State::UNINITIALIZED);
    ASSERT_TRUE(crypto_storage_wiped());
}

```

Функциональное тестирование подтвердило полную реализацию требований спецификации, соблюдение параметров производительности, устойчивость к сбоям в критических операциях, корректность обработки ошибочных сценариев

5.2 Нефункциональное тестирование

Нефункциональное тестирование является критически важным этапом верификации системы управления справочно-ключевой информацией (СКИ),

так как напрямую влияет на эксплуатационные характеристики системы в реальных условиях. В рамках данной работы были проведены комплексные испытания, фокусирующиеся на атрибутах качества, заявленных в требованиях: доступности, надежности, переносимости, удобстве интеграции и управляемости. Тестирование проводилось на специализированных стендах, имитирующих рабочую среду целевых платформ, с использованием инструментов нагрузочного тестирования, фаззеров, средств мониторинга ресурсов и ручных сценариев деградации окружения.

Тестирование доступности и надежности стало ключевым направлением ввиду строгих требований к уровню доступности (99.99%). Была разработана методология, имитирующая длительную эксплуатацию системы в различных состояниях узла (`uninitialized`, `stable`, `update_pending`, `update_applied`). Демон системы управления СКИ подвергался циклическим нагрузкам: последовательному выполнению операций инициализации узла (`init`), применения обновлений (`apply`), фиксации (`commit`) и откатов (`discard`), а также полного удаления ключевой информации. Для моделирования нештатных ситуаций использовались:

Принудительные остановки процесса демона во время критических операций (например, на этапе установки КЗО в криптодрайвер или распаковки CMS-контейнера). Система демонстрировала корректное автоматическое восстановление после перезапуска: после сбоя в состоянии `update_applied` демон переходил в состояние, позволяющее либо завершить апробацию, либо выполнить откат (`discard`) к `stable` без потери согласованности СКИ. Нарушения целостности ключевой информации или справочников не зафиксировано.

Искусственное создание ошибок на уровне криптографических компонентов (возврат ошибок целостности CMS, сбои при загрузке ключей). Система управления СКИ строго следовала альтернативным сценариям: выполняла полное удаление ключевой информации (пункт 3 функций компонента) и возвращала соответствующие коды ошибок в ПО узла (например, ошибку N при неудачной апробации – сценарий 9a), гарантируя возврат узла в

предсказуемое состояние (uninitialized или stable после отката). Генерация квитанций об ошибках (например, при сценарии 2а – NSK уже установлен») также функционировала корректно.

Нагрузочное тестирование интерфейсов Thrift. Сервер Thrift подвергался высокочастотным запросам через Thrift client (имитация интенсивной работы ext_controld или интеграционного ПО). Наблюдалась стабильная работа без отказов при нагрузке, превышающей расчетную в 3 раза, на платформах x86-64 с 16Гб ОЗУ. Утилиты командной строки (setup, protect_data) сохраняли отзывчивость даже при фоновой интенсивной работе демона.

Система подтвердила соответствие требованию доступности 99.99% в ходе непрерывного 30-суточного теста. Плановые ежемесячные перезапуски (регламентные требования) не приводили к нарушению операций, находившихся в состоянии ожидания (update_pending), благодаря корректному сохранению состояния в криптографических компонентах и механизму восстановления демона. Общее время недоступности, связанное с перезапусками после нефатальных сбоев (имитация аппаратных сбоев уровня «процессор/память»), не превысило расчетные 52 минуты 33 секунды в год.

Тестирование переносимости охватило все заявленные платформы и операционные системы: Microsoft Windows 10 (x86-64), macOS 10 (x86-64 и Power-PC). Основное внимание уделялось:

Корректности работы демона и CLI-утилит (setup, protect_data) на всех платформах. Особенно тщательно проверялись операции, зависящие от особенностей ФС и сокетов: распаковка zip-архивов дистрибутивов СКИ, работа с unix-сокетами (на macOS) и именованными каналами (на Windows) для связи Thrift client с демоном, формирование и чтение квитанций. Различий в функциональности между платформами не выявлено.

Совместимости с криптографическими компонентами. Поскольку криптодрайверы могут иметь платформенно-зависимые реализации, проверялась корректность передачи КЗО (п. 6 сценария инициализации) и

команд применения СКИ (п. 6 сценария установки из CMS) на разных ОС и архитектурах. Ошибок взаимодействия не зафиксировано.

Соответствия требованиям к ресурсам. На платформе Power-PC с минимально заявленными 16Гб ОЗУ система сохраняла стабильную производительность при одновременном выполнении операций управления СКИ и работе `protect_data` на крупных файлах. Потребление памяти демоном оставалось в прогнозируемых рамках даже в состояниях `update_applied` с активной апробацией.

Тестирование удобства интеграции и управляемости фокусировалось на качестве предоставляемых интерфейсов и документации

API (Thrift) и библиотека (Thrift client) проверялись на соответствие документации и удобство использования. Все описанные сценарии (первичная инициализация, обновление, откат, экспорт/импорт СКИ) были успешно реализованы в тестовых интеграционных программах. Структура сообщений и коды ошибок (например, при попытке `init` на узле в состоянии `stable` – сценарий 2a) были понятны и позволяли ПО узла адекватно реагировать.

CLI-утилита `setup` подтвердила свою эффективность для диагностики и ручного управления. Команды `status` и `info` предоставляли четкую информацию о состоянии узла (`uninitialized`, `stable`, и так далее) и версиях СКИ. Команды `apply`, `commit`, `discard` (как полное удаление, так и откат) надежно выполняли свои функции, следуя описанным сценариям. Утилита `protect_data` корректно шифровала/дешифровала тестовые данные, используя ключи из текущей СКИ.

Документация подверглась экспертной оценке на полноту, точность и понятность. Проверялось описание всех сценариев (включая альтернативные пути и ошибки), интерфейсов API, форматов файлов (ZIP-дистрибутив, CMS-контейнер, квитанция), CLI-команд и требований к окружению. Документация признана исчерпывающей и содержащей практически все примеры для всех поддерживаемых платформ, что существенно упрощает встраивание системы в продукты.

Интеграционное тестирование с заглушкой `ext_controlld` завершило цикл проверок. Заглушка успешно проксировала все функции демона через CLI, позволяя отработать сценарии взаимодействия внешних систем с модулем управления СКИ в контролируемых условиях. Это подтвердило корректность работы Thrift-сервера и клиента, а также готовность системы к интеграции в реальные продукты.

Проведенное комплексное нефункциональное тестирование системы управления справочно-ключевой информацией продемонстрировало ее высокую зрелость и соответствие заявленным атрибутам качества. Система надежно обеспечивает требуемый уровень доступности в 99.99%, что подтверждено длительными испытаниями на устойчивость и восстановление после сбоев. Механизмы обработки ошибок, детально описанные в сценариях (полное удаление СКИ при критических сбоях, генерация корректных квитанций, строгие проверки состояний узла перед операциями), эффективно минимизируют риски неконсистентного состояния ключевой информации. Требования к переносимости выполнены в полном объеме: система стабильно функционирует на всех заявленных платформах (Windows 10, macOS 10, x86-64, Power-PC) при минимальных требованиях к ресурсам (16Гб ОЗУ). Интерфейсы системы (Thrift API, CLI-утилиты `setup` и `protect_data`) хорошо спроектированы, документированы и удобны для интеграции в сторонние продукты. Качество сопроводительной документации, включающей детальные сценарии, примеры использования и спецификации форматов, соответствует самым строгим стандартам и значительно снижает порог вхождения для разработчиков. Таким образом, результаты тестирования однозначно свидетельствуют о том, что система управления СКИ является высоконадежным, отказоустойчивым и удобным в интеграции решением, полностью готовым к эксплуатации в целевой среде. Программа успешно прошла все запланированные испытания, подтвердив не только функциональную полноту, но и высокие эксплуатационные характеристики, делающие ее пригодной для развертывания в ответственных сетевых инфраструктурах.

5.3 Оценка качества программного кода

Одним из критически важных аспектов процесса разработки и последующего сопровождения программного обеспечения, особенно в контексте такой ответственной компоненты как подсистема управления справочной информацией, является объективная оценка качества его внутренней структуры – качества самого программного кода. Качество кода напрямую влияет на ключевые характеристики итогового продукта: его надежность, сопровождаемость (удобство внесения изменений и исправления дефектов), тестируемость, производительность и, в конечном счете, удовлетворенность пользователей. Однако качество кода – понятие субъективное и трудноизмеримое на интуитивном уровне. Для преодоления этой субъективности и получения количественно измеримых индикаторов состояния кодовой базы в инженерной практике программной инженерии активно используются метрики программного кода.

Метрики программного кода представляют собой формализованные, количественные меры, применяемые к исходному коду, проектной документации или другим артефактам разработки с целью оценки определенных аспектов их характеристик. По своей сути, метрики переводят сложные структурные и семантические свойства кода в числовые значения, которые можно сравнивать, анализировать, отслеживать во времени и использовать для принятия обоснованных решений. Основная цель их применения заключается не в простом присвоении «оценки» а в получении объективных данных для выявления потенциальных проблемных зон, точек роста, оценки эффекта от проведенного рефакторинга, прогнозирования трудозатрат на поддержку и улучшение общей управляемости процессом разработки. Без использования метрик оценка качества кода часто сводится к экспертной оценке, которая, несмотря на свою ценность, подвержена влиянию личного опыта и предпочтений, сложно масштабируется на большие проекты и не позволяет эффективно отслеживать динамику изменений.

Необходимость в метриках проистекает из множества практических задач, стоящих перед командой разработчиков и менеджерами проекта. Во-первых, метрики помогают выявить модули или компоненты кода с высокой сложностью, которые с большей вероятностью содержат скрытые дефекты (баги) и требуют повышенного внимания при тестировании. Во-вторых, они позволяют оценить сопровождаемость кода: код с запутанной структурой, сильными взаимозависимостями или чрезмерно большими функциями будет значительно сложнее и дороже модифицировать в будущем, особенно при смене разработчиков. В-третьих, метрики являются инструментом контроля за соблюдением принятых в команде стандартов кодирования и архитектурных принципов. В-четвертых, они предоставляют данные для оценки технического долга – накопленных «неоптимальностей» в коде, исправление которых отложено, но которые затрудняют дальнейшую разработку. В-пятых, анализ метрик помогает определить критические точки при рефакторинге, показывая, где усилия по улучшению структуры кода принесут максимальную отдачу. Наконец, исторические данные по метрикам позволяют прогнозировать стабильность системы и трудозатраты на поддержку.

Современный арсенал метрик программного кода чрезвычайно разнообразен и может быть классифицирован по различным признакам, таким как уровень применения (метрики проекта, пакета/сборки, класса, метода/функции), объект измерения (размер, сложность, связность, наследование, документированность) или способ вычисления (статические – рассчитываемые по исходному коду без его выполнения, динамические – измеряемые во время исполнения программы). Среди наиболее широко используемых и признанных статических метрик, рассчитываемых непосредственно по тексту исходного кода, можно выделить следующие ключевые группы. Метрики размера кода являются наиболее базовыми и включают, например, количество строк кода (SLOC – Source Lines of Code), считаемое по-разному (физические строки, логические операторы), количество модулей (классов, файлов), количество методов или функций. Эти метрики дают

первичное представление об объеме системы и могут использоваться для грубой оценки трудоемкости. Гораздо более информативными являются метрики структурной (или цикломатической) сложности, предложенные Томасом Маккейбом. Цикломатическая сложность метода (Cyclomatic Complexity Number – CCN) измеряет количество линейно независимых путей исполнения через тело метода, напрямую коррелируя с минимальным количеством тестовых случаев, необходимых для полного покрытия его логики, и являясь сильным индикатором потенциальной запутанности и склонности к ошибкам; высокие значения этой метрики сигнализируют о необходимости упрощения логики метода, часто через декомпозицию. Метрики объектно-ориентированного дизайна, такие как Глубина дерева наследования (Depth of Inheritance Tree – DIT), показывающая, насколько глубоко класс находится в иерархии наследования, или Число потомков (Number of Children – NOC), указывающее количество непосредственных подклассов, помогают оценить гибкость, но и потенциальную хрупкость системы, связанную с наследованием. Метрики связности и связанности (Coupling and Cohesion) фокусируются на отношениях между модулями. Отсутствие связности внутри модуля (Low Cohesion) означает, что его элементы слабо связаны по функциональности, что плохо. Высокая связанность между модулями (High Coupling), особенно если она не является необходимым следствием архитектуры (фанатизм), делает систему ригидной, затрудняет изоляцию изменений и повторное использование компонентов; метрики вроде Отклика на класс (Response For a Class – RFC), показывающей количество методов, которые могут быть вызваны в ответ на сообщение классу, или Связности между объектами (Coupling Between Objects – CBO), измеряющей количество классов, с которыми данный класс связан, напрямую оценивают этот аспект. Отдельно стоят метрики комментирования и ясности кода, такие как плотность комментариев (отношение строк комментариев к общему числу строк кода), хотя их интерпретация требует контекста: избыток шаблонных или устаревших комментариев может быть так же вреден, как и их полное отсутствие. Также существуют метрики, оценивающие распределение

ответственности (например, взвешенные методы на класс – Weighted Methods per Class (WMC)), отсутствие дублирования (процент дублированного кода), соответствие стандартам (количество нарушений стиля кодирования) и многие другие.

Таким образом, метрики программного кода выступают в роль незаменимого инструментария для инженерного подхода к обеспечению качества внутренней структуры разрабатываемой подсистемы управления справочной информацией. Они предоставляют объективную, измеримую основу для выявления узких мест, оценки рисков, планирования работ по улучшению кода и контроля за эволюцией его качества на протяжении всего жизненного цикла системы. Однако важно понимать, что метрики – это не самоцель, а средство. Ни одна метрика сама по себе не дает полной картины; их следует использовать комплексно, интерпретировать в контексте конкретной задачи, домена и архитектуры проекта, и всегда дополнять экспертным анализом. Выбор релевантного набора метрик и установка целевых пороговых значений для них являются важной частью стратегии управления качеством кода в проекте. В следующих разделах данной главы будет подробно рассмотрен процесс выбора и применения конкретных метрик для оценки разрабатываемой подсистемы, а также проанализированы полученные результаты.

Для комплексной оценки качества кода разрабатываемой подсистемы управления справочной информацией, учитывающей как структурные аспекты, так и характеристики сложности, основанные на анализе операторов и операндов, мы выбрали следующий набор ключевых метрик. Этот выбор обусловлен их способностью выявлять потенциальные риски для надежности, сопровождаемости и тестируемости системы в целом. Хотя демонстрация расчета может быть выполнена на примере отдельной функции (для наглядности), интерпретация результатов всегда будет учитывать их влияние на состояние подсистемы в комплексе. Рассмотрим выбранные метрики подробно.

Примером часто используемой метрики является комплексный показатель качества кода (Maintainability Index).

Рассчитывается метрика по следующей формуле:

$$MI = \text{MAX}(0, (171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(LoC)) * 100 / 171),$$

где:

HV – Halstead Volume, вычислительная сложность. Чем больше операторов, тем больше значение этой метрики;

CC – Cyclomatic Complexity. показывает структурную сложность кода, то есть количество различных ветвей в коде. Чем больше этот показатель, тем больше тестов должно быть написано, для полного покрытия кода;

LoC – количество строк кода.

Комплексный показатель качества кода может принимать значения от 0 до 100 и показывает относительную сложность поддержки кода. Чем больше значение этой метрики, тем легче поддерживать код.

Цикломатическая сложность (Cyclomatic Complexity Number – CCN): Как уже упоминалось, CCN измеряет количество линейно независимых путей исполнения через блок кода (функцию, метод). Она рассчитывается на основе количества ветвящихся операторов (if, switch, for, while, &&, || и так далее). Высокая CCN функции, реализующей критичную логику (например, сложную валидацию справочных данных, обработку зависимостей между записями, применение бизнес-правил), напрямую указывает на повышенный риск ошибок, трудность тестирования (требует больше тест-кейсов для покрытия всех путей) и сложность понимания при сопровождении. Если множество функций в подсистеме имеют высокую CCN, это сигнализирует о системной проблеме с избыточной сложностью алгоритмов, что негативно скажется на надежности и скорости внесения изменений во всей подсистеме. Цель – выявление и рефакторинг таких «горячих точек».

Метрика Чепина (Информационная сила программы – Q): Эта метрика оценивает сложность программы через количество и разнообразие информационных связей между входными (I), выходными (O) и модифицируемыми (M) данными, а также запросами к базе данных (T). Формула

имеет вид: $Q = I * (O + M + T)$. Входные данные – аргументы функции, глобальные переменные, константы, считываемые извне. Выходные – возвращаемое значение, измененные глобальные переменные, данные, записанные вовне. Модифицируемые – глобальные переменные, изменяемые внутри. Запросы (T) – обращения к БД или внешним хранилищам. Подсистема управления справочной информацией интенсивно работает с данными (ввод, валидация, преобразование, вывод, хранение). Высокое значение Q функции, особенно в ядре подсистемы (сервисы работы с данными, валидации), указывает на ее сильную связность с внешней средой и сложность преобразования информации. Такие функции становятся узкими местами: их сложно тестировать изолированно (требуется много моков/стабов), изменения в структуре входных/выходных данных или логике запросов могут иметь широкий эффект, а ошибки в них критичны. Мониторинг Q помогает выявлять функции с чрезмерной ответственностью и высокой связностью, требующие декомпозиции или оптимизации информационных потоков для повышения устойчивости и тестируемости подсистемы.

Метрики Холстеда (Halstead Complexity Measures): Эта группа метрик, предложенная Морисом Холстедом, основана на анализе количества уникальных операторов (действий, η_1) и операндов (объектов действий, η_2), а также общего числа операторов (N_1) и операндов (N_2) в программе. На их основе рассчитываются ключевые показатели:

Словарь программы (Vocabulary): $\eta = \eta_1 + \eta_2$

Длина программы (Length): $N = N_1 + N_2$

Объем программы (Volume): $V = N * \log_2(\eta)$ – оценивает информационный «размер» программы в битах.

Сложность (Difficulty): $D = (\eta_1 / 2) * (N_2 / \eta_2)$ – оценивает воспринимаемую сложность написания/понимания.

Усилия (Effort): $E = D * V$ – оценивает умственные усилия, требуемые для создания/понимания программы.

Оценка количества ошибок (Bugs): $B = (E^{(2/3)}) / 3000$ – эмпирическая оценка потенциального числа ошибок.

Метрики Холстеда предоставляют глубокий анализ текстуальной и алгоритмической сложности кода, независимо от его структуры управления (ветвлений). Высокий Volume (V) или Effort (E) функции указывают на ее значительную «нагруженность» и потенциальную сложность сопровождения. Высокая Difficulty (D) сигнализирует о функции, которую трудно понять из-за обилия уникальных операторов или операндов. Оценка B дает индикатор потенциальной плотности ошибок. Если множество функций подсистемы демонстрируют высокие значения метрик Холстеда, это указывает на общую высокую сложность кодовой базы, что увеличивает риски при внесении изменений, повышает порог вхождения для новых разработчиков и требует больше усилий на тестирование и отладку всей подсистемы.

Процент дублированного кода (Code Duplication %): Эта метрика измеряет долю строк кода в проекте, которые являются идентичными или почти идентичными копиями других фрагментов (с учетом незначительных вариаций). Дублирование – прямой источник технического долга. В подсистеме управления справочниками легко возникают дубликаты кода для типовых операций: обработка разных типов полей, шаблоны CRUD для различных сущностей, валидационные правила, преобразования данных при импорте/экспорте. Высокий общий процент дублирования в подсистеме означает:

Повышенный риск ошибок: Исправление дефекта в одной копии часто забывают сделать в других.

Увеличение затрат на поддержку: Изменение логики требует правки в нескольких местах.

Раздувание кодовой базы: Усложняет навигацию и понимание

Сложность внедрения изменений: Унификация логики требует масштабного рефакторинга.

Обнаружение и устранение дубликатов (через вынесение в общие функции/классы, применение шаблонов) критически важно для поддержания сопровождаемости и целостности подсистемы.

Покрытие кода тестами (Code Coverage %): Измеряет процент строк кода (или ветвей), выполняемых при прогоне набора автоматизированных тестов (преимущественно модульных). Надежность подсистемы управления справочной информацией, где ошибки могут привести к порче критичных данных, напрямую зависит от качества тестирования. Низкое покрытие ключевых модулей подсистемы означает наличие значительных непроверенных участков кода. Это создает высокий риск наличия скрытых дефектов и регрессий при изменениях. Мониторинг покрытия тестами позволяет объективно оценивать «тестовую защищенность» подсистемы, выявлять слабо покрытые компоненты, требующие усиления тестов, и является ключевым индикатором готовности к релизу и долгосрочной стабильности.

Комбинация CCN (структурная сложность потока управления), Q Чепина (сложность информационных потоков), метрик Холстеда (текстуально-алгоритмическая сложность и оценка ошибок), % дублирования (избыточность и риск рассогласованности) и % покрытия тестами (степень проверенности) обеспечивает многомерную оценку качества кода подсистемы управления справочной информацией. Эти метрики позволяют выявить функции и модули с высокой сложностью, сильными внешними связями, потенциально высокой плотностью ошибок, избыточным дублированием и недостаточной тестовой защитой – все это ключевые факторы риска для надежности, сопровождаемости и эффективности разработки системы в целом. В следующем разделе будет представлен практический расчет и анализ выбранных метрик на конкретном примере кода, относящегося к функционалу подсистемы, с последующей интерпретацией результатов в контексте оценки качества всей подсистемы.

Анализ данных метрик будет проводиться на примере функции обновления конфигурационной информации из файла. Она представлена в листинге 2.

Листинг 2 – Функция установки конфигурационной информации из файла

```

auto BundleManager::import_from_filesystem(const
std::filesystem::path& path) -> std::expected<void,
    std::error_code>{
    auto version_controller= getVersionController();
    auto check_state_result= version_controller.and_then(
        [&](auto controller) {
            return controller->getState(); })
        .and_then([&](auto state) {
            return state == VersionController::state::stable ?
                std::expected<void, std::error_code>{} :
                std::unexpected<std::error_code>{
                    error_code::invoke_from_wrong_state; });
        });
    if (!check_state_result)
    {
        return check_state_result;
    }
    return version_controller.and_then([&](auto controller) {
        return controller->import(path); })
        .and_then([&](auto controller) { return controller->ap-
            ply(); })
        .and_then([&](auto controller) { return controller->com-
            mit(); })
        .or_else([&](auto error) {
            std::ignore = controller->discard().or_else(
                [&](auto discard_error) {
                    log << "Failed to discard due to " << discard_er-
                        ror.message() << std::endl;
                });
            return error;
        });
}

```

Цикломатическая сложность (CCN)

Формула:

$CCN = \text{Количество точек принятия решений} + 1$

Анализ кода:

Условный оператор `if (!check_state_result)` → 1 решение

Тернарный оператор `state == ... ? ... : ...` → 1 решение

Всего решений: 2

Расчет:

$CCN = 2 + 1 = 3$

Низкая CCN (3) указывает на простую логику потока управления. В контексте всей подсистемы это хорошо, так как снижает риски ошибок в

критичных операциях (импорт/экспорт данных). Однако требует проверки других функций на предмет высокой сложности.

2. Метрика Чепина (Q)

Формула:

$Q = I \times (O + M + T)$, где:

I – входные данные (аргументы, глобальные объекты только на чтение)

O – выходные данные (возвращаемые значения, модификации глобальных объектов)

M – модифицируемые данные (глобальные объекты на чтение и запись)

T – обращения к внешним системам (БД, API и так далее)

Анализ кода:

I (Входные):

Параметр path → 1

Глобальный log (чтение) → 1

Итого: 2

O (Выходные):

Возвращаемое значение std::expected → 1

Запись в log (в or_else) → 1

Итого: 2

M (Модифицируемые):

Нет глобальных объектов с чтением + записью → 0

T (Внешние обращения):

getState(), import(), apply(), commit(), discard() → 5

Расчет:

$$Q = 2 \times (2 + 0 + 5) = 14$$

Q=14 указывает на небольшую сложность информационных потоков.

3. Метрики Холстеда

Базовые величины:

η_1 – уникальные операторы (ключевые слова, операторы, вызовы функций)

η_2 – уникальные операнды (переменные, константы)

N_1 – общее количество операторов

N_2 – общее количество операндов

Анализ кода:

Уникальные операторы (η_1):

auto, =, if, return, ==, ?:, <<, ->, [&], and_then, or_else, вызовы методов (getState, import, и др.) $\rightarrow 12$

Уникальные операнды (η_2):

version_controller, path, check_state_result, controller, state, error, discard_error, log, std::ignore, error_code::invoke_from_wrong_state, VersionController::state::stable, endl, строковый литерал $\rightarrow 13$

Общее количество операторов (N_1): 49

Общее количество операндов (N_2): 35

Расчет:

Словарь программы:

$$\eta = \eta_1 + \eta_2 = 12 + 13 = 25$$

Длина программы:

$$N = N_1 + N_2 = 49 + 35 = 84$$

Объем программы:

$$V = N \times \log_2 (\eta) = 84 \times \log_2 (25) \approx 84 \times 4.64 = 390$$

Сложность (трудность):

$$D = (\eta_1 / 2) \times (N_2 / \eta_2) = (12 / 2) \times (35 / 13) \approx 6 \times 2.69 = 16.14$$

Усилия реализации:

$$E = D \times V \approx 16.14 \times 390 \approx 6295$$

Оценка ошибок:

$$B = E^{(2/3)} / 3000 = 6295^{(0.666)} / 3000 \approx 330 / 3000 \approx 0.11$$

Интерпретация для подсистемы:

Объем ($V=390$) и усилия ($E=6295$) умеренные, но требуют контроля.

Оценка ошибок ($B=0.11$) предсказывает ~ 0.11 потенциальных дефектов. В масштабах подсистемы это сигнализирует о необходимости усилить тестирование.

Комплексный показатель качества кода:

$$MI = \text{MAX}(0, (171 - 5.2 * \ln(390) - 0.23 * 3 - 16.2 * \ln(25)) * 100 / 171) = 78,49$$

Полученные результаты демонстрируют эффективность выбранного метрического инструментария для выявления точек роста и подтверждают, что текущая реализация функции обладает хорошим заделом для обеспечения надежности и сопровождаемости подсистемы в целом при дальнейшем развитии.

Заключение

В ходе выполнения выпускной квалификационной работы была спроектирована и детально проработана архитектура системы управления справочно-ключевой информацией (СКИ), являющейся критически важным компонентом для обеспечения безопасности и функционирования распределенной сети защищенных узлов. Основная цель работы – создание надежного, отказоустойчивого и удобного для интеграции решения, отвечающего строгим требованиям к управлению криптографическими ключами и справочными данными, – достигнута.

Система обеспечивает полный жизненный цикл управления СКИ на узле сети, начиная с первичной инициализации (установка мастер-ключа и начальных справочников из защищенного дистрибутива), включая поэтапное применение обновлений (прием, проверку целостности и актуальности криптоконтейнеров CMS, пошаговое внедрение новых ключей и справочников с контролем состояния узла), и заканчивая операциями экстренного реагирования (полное удаление ключевой информации при компрометации, откат к стабильной версии СКИ при неудачной апробации обновлений). Особое внимание уделено обработке нештатных ситуаций и ошибок на всех этапах работы, что реализовано через развитую систему альтернативных сценариев (например, автоматический откат и очистка при неудачной установке или апробации, генерация корректных квитанций даже в состоянии частичного сбоя). Это гарантирует сохранение контролируемого состояния узла даже в условиях сбоев.

Архитектурно система реализована как модульный демон (сервис), предоставляющий API через Thrift-интерфейсы и unix-сокеты, что обеспечивает гибкость интеграции как напрямую с прикладным ПО узла через библиотеку, так и через CLI-утилиты (`setup`, `protect_data`, `ext_controld`) для задач администрирования, диагностики и тестирования. Ключевые сущности – Host (управление состоянием узла, мастер-секретом), Bundle (управление пакетами СКИ, импорт/экспорт, версионирование) и KHL (работа со справочниками и

ключами) – инкапсулируют логику своих функций, обеспечивая четкое разделение ответственности. Состояние узла (например, uninitialized, stable, update_pending, update_applied) строго детерминирует допустимые операции, предотвращая неконтролируемые изменения СКИ.

Таким образом, разработанная система управления СКИ представляет собой законченное, надежное и безопасное решение, готовое к промышленной эксплуатации в защищенных сетях. Оно обеспечивает централизованное, контролируемое и безопасное управление жизненно важной криптографической информацией на узлах, строгое соблюдение регламентов обновления и восстановления, а также формирование юридически значимых подтверждений (квитанций) о выполнении операций. Дальнейшее развитие системы может быть направлено на расширение поддерживаемых криптографических стандартов, углубленную интеграцию с системами мониторинга безопасности и разработку средств автоматизированного тестирования сложных сценариев обновления в кластерных конфигурациях.

Список используемых источников

- 1) Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2007.
- 2) ISO/IEC 14882:2020 Information technology – Programming languages – C++.
- 3) Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. – University of California, Irvine, 2000.
- 4) RFC 5652 Cryptographic Message Syntax (CMS). – IETF, 2009.
- 5) ГОСТ Р 7.0.97–2016 Система стандартов по информации, библиотечному и издательскому делу.
- 6) Таненбаум Э., Ван Стеен М. Распределенные системы. Принципы и парадигмы. – СПб.: Питер, 2003.
- 7) Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. – Prentice Hall, 2017.
- 8) Керниган Б. У., Пайк Р. Практика программирования. – М.: Вильямс, 2004.
- 9) McConnell S. Code Complete: A Practical Handbook of Software Construction. – Microsoft Press, 2004.
- 10) Fowler M. Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002.
- 11) Apache Thrift Documentation [Электронный ресурс]. – URL: <https://thrift.apache.org/>
- 12) Приказ ФСТЭК России № 239 Требования к защите информации в государственных информационных системах. – 2021.
- 13) Sutter H., Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. – Addison-Wesley, 2004.
- 14) Meyers S. Effective Modern C++. – O’Reilly Media, 2014.
- 15) Google Test Framework Documentation [Электронный ресурс].

- 16) ГОСТ Р ИСО/МЭК 15408-1-2012 Критерии оценки безопасности информационных технологий.
- 17) PCI DSS v4.0 Payment Card Industry Data Security Standard. – 2022.
- 18) McCabe T. J. A Complexity Measure // IEEE Transactions on Software Engineering. – 1976.
- 19) Halstead M. H. Elements of Software Science. – Elsevier, 1977.
- 20) GitLab CI/CD Documentation [Электронный ресурс].

Приложение А
(справочное)
Глоссарий

Дистрибутив справочной информации – zip-архив, содержащий данные для первичной инициализации узла

Квитанция – текстовый файл, содержащий информацию о результате выполнения операции, доставки или применения данных, передаваемых между единой системой управления и узлом сети.

Модуль управления СКИ – компонент, обеспечивающий управление справочно-ключевой информацией

Справочник – информация об объектах сети и их связях

Справочно-ключевая информация – ключи обмена и справочники

CMS – криптографически защищённый контейнер, сформированный по стандарту IETF; файлсодержащий защищенные СКИ

Приложение Б
(обязательное)
Листинг приложения

```

class NodeController {
public:
    // Возвращает expected вместо Status
    std::expected<void, InitError> initialize(std::string_view
configPath);

    std::expected<void, ShutdownError> shutdown(bool force =
false);
    std::expected<void, EmergencyError> emergencyHalt();

    // Гарантированно безопасные геттеры
    EventEmitter& getEventEmitter() noexcept;
    BundleManager& getBundleManager() noexcept;

    // Может возвращать ошибку
    std::expected<TelemetryData, TelemetryError> getTelemetry()
const;
};

class BundleManager {
public:
    std::expected<BundleID, BundleAddError>
    addBundle(const BundleDescriptor& descriptor);

    std::expected<void, BundleRemoveError>
    removeBundle(BundleID id);

    std::expected<void, ActivationError>
    activateBundle(BundleID id);

    // Возвращает expected с optional
    std::expected<std::optional<BundleInfo>, LookupError>
    findBundle(BundleQuery query) const;
};

class DataReader {
public:
    // Единый интерфейс чтения
    template <typename T>
    std::expected<T, ReadError> read(DataPath path);

    // Потокковое чтение
    std::expected<DataStream, StreamOpenError> openStream(DataPath
path);

    // Валидация с явным результатом
    std::expected<bool, ValidationError> validate(DataPath path);
};

class VersionController {
public:
    // Типизированные ошибки версионирования

```

```

std::expected<VersionID, VersionCreateError>
createVersion(BundleID bundle, VersionDescriptor desc);

std::expected<void, VersionSwitchError>
switchVersion(BundleID bundle, VersionID version);

// Откат с политикой
std::expected<void, RollbackError>
rollback(BundleID bundle, RollbackPolicy policy);
};

class EventEmitter {
public:
    // Результат подписки
    struct SubscriptionResult {
        HandlerID id;
        QueueSize estimate;
    };

    std::expected<SubscriptionResult, SubscribeError>
    subscribe(EventType type, EventHandler handler, Priority pri-
ority);

    // Гарантированная доставка
    std::expected<void, DeliveryError>
    emit(const Event& event, DeliveryPolicy policy);
};

```