

**Национальный исследовательский ядерный университет «МИФИ»**



**Институт интеллектуальных кибернетических систем  
КАФЕДРА КИБЕРНЕТИКИ (№ 22)**

Направление подготовки 09.03.04 Программная инженерия

**Пояснительная записка**

к учебно-исследовательской работе студента на тему:

**Исследование и реализация сферического коня в вакууме  
на основе теоретико-множественного подхода**

Группа Б12-345

Студент \_\_\_\_\_ Иванов А. Б.

Руководитель \_\_\_\_\_  
Петров В. Г.

Научный консультант \_\_\_\_\_

Оценка руководителя 12 \_\_\_\_\_

Оценка комиссии \_\_\_\_\_

Члены комиссии

_____	_____
_____	_____
_____	_____
_____	_____

**Москва 2024**

**Национальный исследовательский ядерный университет «МИФИ»**



**Институт интеллектуальных кибернетических систем  
КАФЕДРА КИБЕРНЕТИКИ (№ 22)**

**Задание на УИР**

Студенту гр. Б12-345 Иванову Александру Борисовичу

**ТЕМА УИР**

**Исследование и реализация сферического коня в вакууме  
на основе теоретико-множественного подхода**

**ЗАДАНИЕ**

№ п/п	Содержание работы	Форма отчетности	Срок исполнения	Отметка о выполнении Дата, подпись
<b>1.</b>	<b>Аналитическая часть</b>			
1.1.	Анализ и сравнение архитектур построения серверных приложений			
1.2.	Анализ и сравнение паттернов проектирования модулей системы			
1.3.	Анализ инструментов применяемых для реализации системы			
<b>2.</b>	<b>Теоретическая часть</b>			
2.1.	Разработка моделей микросервисов			
<b>3.</b>	<b>Инженерная часть</b>			
3.1.	Проектирование ... (системы, подсистемы, модуля...)			
3.2.	Использовать методологию проектирования....			
3.3.	Разработать архитектуру для... (с учетом требований к...)			
3.4.	Результаты проектирования оформить с помощью.... При проектировании использовать язык... (например, IDEF, или UML)			
<b>4.</b>	<b>Технологическая и практическая часть</b>			
4.1.	Реализовать... (систему, подсистему, модуль...)	Исполняемые файлы, исходный текст		
4.2.	Протестировать... с помощью...			
4.3.	Разработать тестовые примеры для...	Исполняемые файлы, исходные тексты тестов и тестовых примеров		

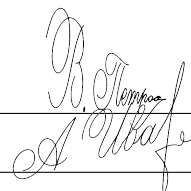
4.4.	Реализация должна иметь форму/обладать качествами...			
4.5.	Ожидаемым результатом является программная система/программный комплекс/программное обеспечение... со следующими отличительными характеристиками...			
4.6.	При реализации использовать технологию/платформу...			
5.	Оформление пояснительной записки (ПЗ) и иллюстративного материала для доклада.	Текст ПЗ, презентация	15.12.2021	

## ЛИТЕРАТУРА

1. Сычёв М. С. История Астраханского казачьего войска: учебное пособие. — Астрахань : Волга, 2009.
2. Соколов А. Н., Сердобинцев К. С. Гражданское общество: проблемы формирования и развития (философский и юридический аспекты): монография / под ред. В. М. Бочарова. — Астрахань : Калининградский ЮИ МВД России, 2009.
3. Гайдаенко Т. А. Маркетинговое управление: принципы управленческих решений и российская практика. — 3-е изд, перераб. и доп. — М. : Эксмо: МИРБИС, 2008.

Дата выдачи задания:  
01.09.2020

Руководитель  
Студент



Петров В. Г.  
Иванов А. Б.

## Реферат

Общий объем основного текста, без учета приложений — 32 страниц, с учетом приложений — 40. Количество использованных источников — XX. Количество приложений — X.

КЛЮЧЕВОЕ СЛОВО 1, КЛЮЧЕВОЕ СЛОВО 2, ....

Целью данной работы является ...

В первой главе проводится обзор и анализ ...

Во второй главе описываются использованные и разработанные/модифицированные методы/модели/алгоритмы ....

В третьей главе приводится описание программной реализации и экспериментальной проверки ....

В приложении А описаны основные требования к форматированию пояснительных записок к дипломам и (магистерским) диссертациям.

В приложении В представлена общая структура пояснительной записки.

В приложении Г приведены некоторые дополнительные комментарии к использованию данного шаблона.

# Содержание

<b>Введение</b>	<b>5</b>
<b>1 Аналитическая часть</b>	<b>6</b>
1.1 Анализ проблемной области и архитектур построения серверных приложений . . . . .	6
1.1.1 Микросервисная архитектура . . . . .	7
1.1.2 Монолитная архитектура . . . . .	9
1.1.3 Сравнение монолитной и микросервисной архитектур . . . . .	9
1.2 Анализ и сравнение паттернов проектирования модулей системы . . . . .	10
1.2.1 Луковая архитектура . . . . .	13
1.2.2 Hexagonal Architecture . . . . .	14
1.2.3 Чистая архитектура . . . . .	15
1.3 Анализ инструментов применяемых для реализации системы . . . . .	16
1.3.1 Взаимодействие микросервисов . . . . .	16
1.3.2 Хранение данных . . . . .	20
1.3.3 Авторизация . . . . .	21
1.4 Выводы . . . . .	23
1.5 Постановка задачи научно исследовательской работы . . . . .	23
<b>2 Теоретическая часть</b>	<b>24</b>
2.1 Модель микросервиса . . . . .	24
2.2 Модели модулей системы . . . . .	25
2.2.1 Модель сервиса аутентификации . . . . .	25
2.2.2 Модель сервиса узи . . . . .	26
2.2.3 Модель Gateway API . . . . .	27
<b>3 Результаты проектирования ...</b>	<b>28</b>
3.1 Использование методики «такой-то» для проектирования программных систем «такого-то типа» . . . . .	28
3.2 Общая архитектура системы ... . . . .	28
3.3 Архитектура подсистемы 1... . . . .	28

3.4	Архитектура подсистемы $N$ ...	28
3.5	Проектирование протокола взаимодействия подсистем $X$ и $Y$	28
3.6	Выводы	28
<b>4</b>	<b>Реализация и экспериментальная проверка ...</b>	<b>29</b>
4.1	Выбор инструментальных средств	29
4.2	Состав и структура реализованного программного обеспечения	29
4.3	Основные сценарии работы пользователя	30
4.4	Разработка тестовых примеров	30
4.5	Сравнение реализованного программного обеспечения с существующими аналогами	31
4.6	Выводы	31
	<b>Заключение</b>	<b>32</b>
	<b>Приложения</b>	<b>33</b>
<b>A</b>	<b>Основные правила форматирования</b>	<b>33</b>
<b>B</b>	<b>Общая структура пояснительной записки</b>	<b>35</b>
<b>Г</b>	<b>Правила использования шаблона</b>	<b>38</b>
Г.1	Титульные листы	39
Г.1.1	Титульные листы LaTeX	39
Г.1.2	Титульные листы из PDF	39

## Введение

Введение всегда содержит краткую характеристику работы по следующим аспектам:

- актуальность:
  - кто и почему в настоящее время интересуется данной проблематикой (в т.ч. для решения каких задач могут быть полезны исследования в данной области),
  - краткая история вопроса (в формате год-фамилия-что сделал),
  - нерешенные вопросы/проблемы;
- новизна работы (что нового привносится данной работой);
- оригинальная суть исследования;
- содержание по главам (по одному абзацу на главу).

Общий объем введения должен не превышать 1,5 страниц (для ПЗ к УИРам может быть чуть меньше).

# 1. Аналитическая часть

## 1.1 Анализ проблемной области и архитектур построения серверных приложений

Традиционная архитектура на основе клиент-серверной модели является одной из наиболее распространенных архитектур для создания серверных приложений. Эта модель предполагает разделение приложения на две составляющие - клиентскую и серверную. Клиентская составляющая отвечает за интерфейс взаимодействия с пользователем, а серверная - за обработку запросов и предоставление данных. Основным преимуществом клиент-серверной модели является возможность создания распределенной архитектуры, которая позволяет создавать приложения с большими объемами данных или с большим количеством пользователей. Также этот подход позволяет легко масштабировать и обновлять серверное приложение без влияния на клиентскую составляющую. Одним из недостатков клиент-серверной модели является ее зависимость от сети. Отказ сети или недоступность сервера может привести к невозможности доступа к приложению. Еще одним недостатком является необходимость настройки и поддержки серверной инфраструктуры, что может требовать отдельных затрат и увеличить стоимость разработки и поддержки серверного приложения. Для реализации клиент-серверной модели на практике используются различные технологии и инструменты, как проприетарные, так и открытые. Например, для создания серверных приложений могут использоваться языки программирования, такие как Java, Python, Ruby, PHP, а для создания клиентских приложений - HTML, CSS, JavaScript. Существует также множество фреймворков, библиотек и инструментов, которые предназначены для упрощения разработки серверных приложений на основе клиент-серверной модели. Например, Node.js является популярным фреймворком для создания серверных приложений на JavaScript, а Django и Flask - для Python. Также используются такие средства, как базы данных, предназначенные для хранения данных приложения, или сервисы и инструменты для автоматизации и управления развертыванием и масштабированием серверных приложений. Кроме того, существуют такие платформы, как Amazon Web Services и Microsoft Azure, которые предоставляют в облаке готовые инструменты для создания и развертывания серверных приложений на основе клиент-серверной модели.



### 1.1.1 Микросервисная архитектура

Микросервисная архитектура является относительно новым подходом к разработке серверных приложений. Она характеризуется модульностью, то есть каждый модуль является отдельным сервисом, который может быть развернут и масштабирован отдельно от других модулей. Микросервисная архитектура рассматривает приложение как совокупность небольших сервисов, которые представляют отдельные функции приложения. Каждый сервис обычно работает независимо от других сервисов, используя API для обмена данными между другими сервисами. Основное преимущество микросервисной архитектуры заключается в ее модульности. Это дает возможность быстро отвечать на изменения требований и легко масштабировать приложение по мере необходимости. Другим преимуществом микросервисной архитектуры является возможность использования различных технологий и языков программирования для различных сервисов. Это позволяет использовать наилучшее решение для каждой отдельной задачи. Микросервисная архитектура может быть также более устойчивой и надежной, поскольку имеется возможность аварийного отключения отдельных сервисов, позволяя уменьшить влияние отказа других компонентов на работу приложения в целом. Существует несколько типов микросервисных архитектур, которые различаются по тому, как организованы и взаимодействуют между собой микросервисы. К одному из распространенных типов микросервисных архитектур относится оркестрованная архитектура [8] – это такая архитектура, в которой существует централизованный компонент или слой, называемый оркестратором, который координирует действия и взаимодействие других микросервисов в системе. Оркестратор отвечает за управление жизненным циклом запросов от клиентов, контроль последовательности выполнения операций и обработку бизнес-логики интеграции между сервисами. Оркестрованная архитектура обеспечивает высокий уровень контроля и координации в распределенных системах, позволяя эффективно управлять сложными процессами и задачами. В то же время, она может создавать единую точку отказа, увеличивая зависимость системы от работы оркестратора. Ещё одним из распространённых типов микросервисных архитектур является event-driven архитектура – архитектура, в которой микросервисы взаимодействуют через отправку и прием событий. Каждый сервис может публиковать события, на которые другие сервисы могут, соответственно, подписываться и реагировать. К её преимуществам относятся: гибкость (компоненты могут быть легко добавлены или удалены без влияния на другие части системы), масштабируемость (каждый компонент может масштабироваться независимо для обеспечения высокой произ-

водительности), отказоустойчивость (отказ одного компонента не блокирует работу всей системы). Однако тут возникает сложность в обеспечении согласованности данных между различными сервисами, управлении событиями и обработке ошибок. Также стоит упомянуть про подход API Gateway – данный тип архитектуры использует централизованный шлюз, который является единой точкой входа для всех запросов клиентов к микросервисам или другим системам. Шлюз может выполнять несколько функций, к которым относятся: аутентификация, авторизация, маршрутизация запросов, преобразование протоколов и форматов данных, кеширование, балансировка трафика и другие. Данный подход позволяет сделать архитектуру приложения более гибкой, безопасной и легкой в управлении, а также позволяет централизованно управлять всеми взаимодействиями клиентов с микросервисами, упрощая развертывание и поддержку системы.

Микросервисная архитектура предусматривает разбиение приложения на небольшие независимые сервисы, каждый из которых выполняет строго определённые задачи. Эти сервисы взаимодействуют через сети с использованием API или брокеров сообщений.

#### **Преимущества микросервисной архитектуры:**

- **Гибкость масштабирования:** отдельные сервисы масштабируются независимо, что оптимизирует использование ресурсов.
- **Свобода выбора технологий:** команды могут использовать разные языки программирования и инструменты для реализации отдельных сервисов.
- **Лучшая управляемость:** изолированные сервисы упрощают внесение изменений и поддержку.
- **Повышенная устойчивость:** сбой одного сервиса не влияет на работу других.
- **Параллельная разработка:** команды могут работать над разными сервисами одновременно.

#### **Недостатки микросервисной архитектуры:**

- **Сложность проектирования:** требует тщательного планирования взаимодействия между сервисами.
- **Проблемы с согласованностью данных:** переход к eventual consistency добавляет сложности.
- **Высокие инфраструктурные затраты:** управление множеством сервисов требует дополнительных инструментов, таких как Kubernetes.
- **Увеличение задержек:** сетевое взаимодействие между сервисами медленнее, чем вызовы в памяти.

- **Сложности диагностики:** распределённый характер системы затрудняет выявление и исправление ошибок.

### 1.1.2 Монолитная архитектура

Монолитная архитектура представляет собой подход, при котором всё приложение разрабатывается как единый, неделимый блок. В нём объединены пользовательский интерфейс, бизнес-логика и уровень работы с данными. Такой подход исторически был основным в разработке ПО.

#### **Преимущества монолитной архитектуры:**

- **Простота разработки:** начальный этап разработки требует меньше усилий на планирование структуры.
- **Единая кодовая база:** централизованное хранение и управление кодом упрощает процесс разработки и интеграции изменений.
- **Лёгкость развертывания:** однотипные развертывания минимизируют риск несовместимости.
- **Высокая производительность:** отсутствие сетевого взаимодействия между компонентами ускоряет их взаимодействие.
- **Поддержка транзакционности:** встроенные механизмы работы с транзакциями позволяют обеспечить согласованность данных.

#### **Недостатки монолитной архитектуры:**

- **Сложность масштабирования:** увеличение нагрузки требует масштабирования всего приложения, даже если это касается лишь одного его компонента.
- **Ограничения технологического выбора:** единая технологическая платформа ограничивает возможности внедрения новых инструментов.
- **Рост сложности кода:** по мере увеличения объёма функциональности поддержка системы становится сложнее.
- **Долгое развертывание:** внесение малейших изменений требует пересборки и перезапуска приложения.
- **Уязвимость к сбоям:** сбой в одном компоненте приводит к отказу всей системы.

### 1.1.3 Сравнение монолитной и микросервисной архитектур

Для более наглядного понимания различий между монолитной и микросервисной архитектурами представим их основные аспекты в таблице:

Аспект	Монолитная архитектура	Микросервисная архитектура
Сложность разработки	Низкая	Высокая
Масштабируемость	Ограниченная	Гибкая
Устойчивость к сбоям	Низкая	Высокая
Технологический выбор	Ограниченный	Гибкий
Развертывание	Простое	Сложное
Поддержка транзакционности	Простая	Сложная
Задержки взаимодействия	Минимальные	Увеличенные
Затраты на инфраструктуру	Низкие	Высокие

Таблица 1.1 – Сравнение монолитной и микросервисной архитектур

**Выводы:** Монолитная архитектура лучше подходит для проектов с ограниченным функционалом и небольшими командами, где важны простота и быстрота разработки. Микросервисы же оправданы в сложных, масштабируемых системах с распределёнными командами и высокими требованиями к отказоустойчивости.

## 1.2 Анализ и сравнение паттернов проектирования модулей системы

Связанность (англ. Coupling) и согласованность (англ. Cohesion) являются фундаментальными концепциями в области разработки программного обеспечения, которые оказывают значительное влияние на читаемость, поддержку, тестируемость и переиспользование кода. В данной работе исследуются эти понятия, их виды, а также взаимосвязь между ними с целью формирования рекомендаций по их применению для повышения эффективности разработки.

### Связанность (Coupling)

Связанность характеризует степень зависимости одного программного модуля от других. Чем выше уровень связанности, тем сильнее изменения в одном модуле отражаются на других. Стремление к низкой связанности (low coupling) является ключевым принципом проектирования, позволяющим минимизировать количество зависимостей между модулями и повысить их автономность.

Существует несколько типов связанности, классифицируемых по степени их зависимости:

- **Content coupling** (содержательная связанность): возникает, когда один модуль полагается на внутреннюю реализацию другого модуля, например, доступ к его локальным данным. Изменения в одном модуле требуют изменений в другом.
- **Common coupling** (общая связанность): модули используют общие данные, например, глобальные переменные. Изменение этих данных затрагивает все модули, ко-

торые с ними работают.

- **External coupling** (внешняя связанность): несколько модулей зависят от общего внешнего ресурса, такого как сервис или API. Изменения в этом ресурсе могут привести к некорректной работе модулей.
- **Control coupling** (управляющая связанность): один модуль управляет поведением другого через передачу управляющих сигналов, например, флагов, изменяющих поведение функций.
- **Stamp coupling** (штампованная связанность): модули обмениваются сложными структурами данных, однако используют лишь их часть. Это может приводить к побочным эффектам из-за неполной или изменяемой информации.
- **Data coupling** (связанность по данным): наименее инвазивный тип, при котором модули взаимодействуют исключительно через передачу конкретных данных. Такой подход минимизирует взаимозависимость.

## Согласованность (Cohesion)

Согласованность характеризует степень внутренней связанности функциональных элементов модуля. Высокая согласованность подразумевает, что все компоненты модуля ориентированы на выполнение одной конкретной задачи или функции, что улучшает читаемость и управляемость кода.

Типы согласованности включают:

- **Functional cohesion** (функциональная согласованность): модуль объединяет весь необходимый функционал для выполнения одной задачи.
- **Sequential cohesion** (последовательная согласованность): результаты одной функции используются в качестве входных данных для другой.
- **Communication cohesion** (коммуникационная согласованность): все элементы модуля работают с одними и теми же данными.
- **Procedural cohesion** (процедурная согласованность): элементы модуля выполняются в определенной последовательности, необходимой для достижения результата.
- **Temporal cohesion** (временная согласованность): функции модуля сгруппированы на основе их выполнения в определенный момент времени (например, при обработке ошибок).
- **Logical cohesion** (логическая согласованность): элементы модуля объединены логически общей функцией, но могут отличаться по своей природе (например, обработ-

ка различных типов ввода).

- **Coincidental cohesion** (случайная согласованность): элементы модуля не связаны логически, что делает модуль хаотичным и сложным для понимания.

## Взаимосвязь между Coupling и Cohesion

Связанность и согласованность находятся в сложной взаимозависимости. Уменьшение связанности обычно способствует увеличению согласованности и наоборот. Например, избыточная согласованность может привести к росту зависимости между модулями, что увеличивает связанность. Таким образом, ключевым аспектом является нахождение оптимального баланса, который обеспечит поддержку высокого уровня согласованности при минимально возможной связанности.

## Примеры применения

Рассмотрим два варианта компоновки модулей:

- Низкая согласованность:

```
./internal
  ./model
    product.go
    user.go
  ./factory
    product_factory.go
    user_factory.go
  ./repository
    product_repository.go
    user_repository.go
```

В данном случае функционал рассредоточен, что усложняет понимание структуры приложения и взаимодействия его компонентов.

- Высокая согласованность:

```
./internal
  ./product
    product.go
    product_factory.go
```

```
product_repository.go
./user
user.go
user_factory.go
user_repository.go
```

Здесь модули структурированы по функциональным областям, что повышает согласованность и упрощает работу с кодом.

Применение принципов низкой связанности и высокой согласованности позволяет создавать поддерживаемые, масштабируемые и понятные программные системы. Их использование способствует не только повышению качества кода, но и ускорению разработки, что особенно важно в современных условиях высокой конкуренции и быстрого изменения требований.

### 1.2.1 Луковая архитектура

Луковая архитектура была предложена Джеффри Палермо в 2008 году как расширение гексагональной архитектуры. Её цель — облегчить поддержку приложений за счёт разделения аспектов и строгих правил взаимодействия слоёв.

#### Основная идея

Луковая архитектура предполагает многослойную организацию системы, где каждый слой зависит только от внутреннего слоя. Центральным слоем всегда является независимый слой, который представляет собой сердцевину архитектуры. Остальные слои располагаются вокруг него, создавая структуру, похожую на лук.

#### Ключевые принципы:

- Приложение строится вокруг независимой объектной модели.
- Внутренние слои определяют интерфейсы, внешние — реализуют их.
- Зависимости направлены к центру.
- Код предметной области изолирован от инфраструктуры.

#### Слои

**Domain Model:** содержит основные сущности и поведение предметной области, например, валидацию данных.

**Domain Services:** реализует бизнес-логику, которая не связана с конкретными сущностями, например, расчёт стоимости заказа.

**Application Services:** оркестрирует бизнес-логику, например, обработку запроса на создание заказа.

**Infrastructure:** реализует интерфейсы и адаптеры для работы с базами данных, внешними API и другими ресурсами.

### Преимущества

- Независимость предметной области от инфраструктуры.
- Гибкость замены внешних слоёв.
- Высокая тестируемость.

### Недостатки

- Сложность обучения.
- Необходимость создания множества интерфейсов.

### 1.2.2 Hexagonal Architecture

**Hexagonal Architecture** (также известная как Ports and Adapters) была предложена Alistair Cockburn. Её цель — устранить зависимость бизнес-логики от внешних систем и сделать приложение более гибким и тестируемым.

#### Основная идея

Hexagonal Architecture представляет приложение в виде шестиугольника, где каждая сторона представляет интерфейс (порт) для взаимодействия с внешними системами (адаптерами). Центр архитектуры — это бизнес-логика, изолированная от деталей реализации.

#### Ключевые принципы:

- Бизнес-логика полностью изолирована от инфраструктуры.
- Взаимодействие с внешним миром происходит через порты и адаптеры.
- Внешние системы подключаются через адаптеры, соответствующие определённым портам.



## Слои

**Core (Business Logic):** центральная часть архитектуры, содержащая доменные сущности и бизнес-правила.

**Ports:** интерфейсы, через которые осуществляется взаимодействие между Core и внешними системами.

**Adapters:** реализация портов для подключения конкретных технологий, таких как базы данных, веб-сервисы и т.д.

## Преимущества

- Гибкость: легко заменять внешние системы, не затрагивая бизнес-логику.
- Тестируемость: изолированная бизнес-логика упрощает написание модульных тестов.
- Чёткое разделение обязанностей.

## Недостатки

- Сложность проектирования и реализации.
- Дополнительные накладные расходы на создание портов и адаптеров.

### 1.2.3 Чистая архитектура

**Чистая архитектура**, предложенная Робертом Мартином, основывается на идеях Onion и Hexagonal архитектур. Её ключевая цель — изолировать бизнес-логику от деталей реализации.

## Базовые принципы

- Использование принципов SOLID.
- Разделение системы на слои.
- Независимость бизнес-логики от инфраструктуры.
- Тестируемость бизнес-логики без внешних зависимостей.

## Слои

**Domain:** содержит общие бизнес-правила, структуры и интерфейсы.

**Application:** реализует конкретные сценарии использования, например, обработку запросов.

**Presentation:** отвечает за взаимодействие с пользователем через REST API, CLI и т.д.

**Infrastructure:** содержит детали реализации, такие как доступ к базе данных или внешние сервисы.

### Пересечение границ

В чистой архитектуре зависимости направлены внутрь. Внешние слои реализуют интерфейсы, определённые внутренними слоями. Это достигается за счёт принципа инверсии зависимостей (DIP).

### Преимущества

- Независимость от фреймворков и инфраструктуры.
- Простота тестирования.
- Переносимость и возможность разделения на микросервисы.

### Недостатки

- Требуется строгое разделение бизнес-правил.
- Возможность излишнего усложнения структуры.

### Вывод

**Чистая архитектура** представляет собой эволюцию луковой и гексагональной архитектур, сохраняя их принципы, но делая больший акцент на SOLID и изоляции бизнес-логики. Она подходит для сложных систем, требующих высокой гибкости и тестируемости, но может быть избыточной для небольших приложений.

## 1.3 Анализ инструментов применяемых для реализации системы

### 1.3.1 Взаимодействие микросервисов

Взаимодействие между микросервисами имеет критическое значение для обеспечения их эффективной работы и достижения бизнес-целей. Существует два основных подхода к взаимодействию: синхронное и асинхронное, каждый из которых служит своим целям и задачам.

Синхронное взаимодействие необходимо в ситуациях, когда требуется немедленная обратная связь. Например, когда клиент получает данные из базы данных или выполняет операции, которые требуют актуальной информации в реальном времени. В таких сценариях задержка в получении ответа может нарушить логическую последовательность

действий, и важно обеспечить, чтобы данные были актуальны на момент выполнения запроса.

Асинхронное взаимодействие, в свою очередь, используется в случаях, когда немедленный ответ не требуется. Это позволяет микросервисам отправлять запросы и продолжать выполнять свои задачи, не дожидаясь ответа. Например, в случаях обработки фоновых задач или выполнения операций, которые могут быть выполнены позже, асинхронные подходы позволяют разгружать сервисы и поддерживать их работу без блокировок. Это особенно выгодно в сценариях с высокой нагрузкой, когда множество запросов могут приходить одновременно.

## Синхронное взаимодействие

Микросервисы могут синхронно общаться между собой с использованием различных технологий и протоколов. В этом контексте рассмотрим четыре популярных метода взаимодействия: HTTP/REST API, gRPC, GraphQL и WebSocket.

В большинстве случаев для микросервисов, которые обмениваются фиксированными данными, использование GraphQL может быть излишним, поскольку этот подход предназначен для динамического запроса данных. В сценариях, где структуры данных заранее известны и не требуют изменения, REST API будет более простым и понятным решением. То же касается и WebSocket: двунаправленный стриминг данных может быть избыточным для многих бизнес-приложений, где достаточно стандартного запроса и ответа.

Таким образом, основные методы, которые стоит рассмотреть для синхронного взаимодействия микросервисов, — это HTTP/REST API и gRPC.

**HTTP/REST API** HTTP/REST API — это один из самых распространенных способов синхронного взаимодействия между микросервисами. Каждый микросервис предоставляет набор эндпоинтов, к которым другие сервисы могут обращаться для выполнения операций и получения данных. Используя стандартные методы HTTP (GET, POST, PUT, DELETE), микросервисы обмениваются сообщениями с четко определенными правилами и структурой.

### Преимущества использования HTTP/REST

- **Простота:** REST API легко реализовать и документировать. Он основан на понятных стандартах HTTP и может использоваться практически на любой платформе.
- **Читаемость:** Структура URL и использование HTTP-методов делают интерфейс API

интуитивно понятным и удобным для работы разработчиков.

- **Совместимость:** REST API может легко взаимодействовать с разными языками программирования и платформами, что делает его универсальным решением для микросервисной архитектуры.

**gRPC** gRPC, разработанный Google, представляет собой высокопроизводительный фреймворк для удаленных вызовов процедур (RPC), который поддерживает множество языков программирования. Он использует HTTP/2 для передачи данных, что обеспечивает преимущества, такие как многопоточность и меньшие задержки при передаче информации.

#### Преимущества использования gRPC

- **Производительность:** Использование HTTP/2 позволяет gRPC эффективно обрабатывать множество параллельных запросов, что делает его более производительным, чем традиционные REST API.
- **Статическая типизация:** gRPC использует Protocol Buffers для описания структуры данных, что позволяет разработчикам строго определять, какой тип данных будет передаваться. Это обеспечивает дополнительную безопасность и позволяет избежать ошибок при взаимодействии между сервисами.
- **Автогенерация кода:** Удобство разработки достигается благодаря автоматической генерации клиентского кода на разных языках, что ускоряет процесс создания микросервисов.

#### Сравнение HTTP/REST API и gRPC

Характеристика	HTTP/REST API
Протокол	Основан на HTTP/1.1 или HTTP/2
Структура данных	Ограничена JSON/XML
Типизация	Нестатическая типизация (JSON)
Производительность	Может иметь более высокую задержку
Поддержка потоковой передачи	Ограниченная
Автогенерация клиента	Обычно требует ручного создания клиентского кода
Простота и читаемость	Прост в реализации, легко понимается
Универсальность	Широко используется и поддерживается, совместим с многими
Тестирование и отладка	Легче тестировать с инструментами для работы с HTTP (напр

Таблица 1.2 – Сравнение HTTP/REST API и gRPC

**Заключение** Нашей системе не требуется возможность динамической типизации, поэтому выбор сделать в сторону gRPC.

## Асинхронное взаимодействие

Асинхронное взаимодействие между микросервисами позволяет увеличить производительность и гибкость распределенных систем. В этом подходе микросервисы могут обмениваться данными без необходимости дожидаться ответа, что снижает задержки и повышает общую эффективность системы. Рассмотрим основные методы асинхронного взаимодействия.

### Основные методы асинхронного взаимодействия

- **Очереди сообщений:** Использование систем обмена сообщениями, таких как RabbitMQ, Apache Kafka или Redpanda, позволяет отправлять сообщения между микросервисами без прямого связывания. Один сервис может отправлять сообщения в очередь, а другой — извлекать их и обрабатывать по мере возможности. Это гарантирует, что сервисы могут работать независимо друг от друга и не блокируют друг друга в случае высокой нагрузки.
- **Событийно-ориентированная архитектура:** В этой архитектуре микросервисы реагируют на события, происходящие в системе. События могут генерироваться различными компонентами и служить сигналами для других микросервисов о том, что произошло что-то важное (например, изменение состояния, завершение задачи и т. д.). Это позволяет строить более гибкие и масштабируемые системы.
- **HTTP-события (Webhooks):** Использование вебхуков позволяет микросервису отправлять HTTP-запросы в другие сервисы при наступлении определённых событий. Это простой способ интеграции, позволяющий уведомлять другие службы о произошедших изменениях или событиях.

Выбор в сторону очередей сообщений также обуславливается необходимостью наличия механизма Dead Letter Queue (DLQ), который позволяет обрабатывать ошибки при отправке и получении сообщений. Кроме того, мы стремимся автоматизировать хранение сообщений, что делает использование очередей сообщений предпочтительным решением.

### Описание систем очередей сообщений

- **Apache Kafka:** - Kafka — это распределенная платформа для потоковой передачи данных, которая обеспечивает высокую пропускную способность и низкую задержку. Она работает по принципу публикации и подписки, позволяя множеству клиентов читать и писать сообщения. - **Назначение:** Идеально подходит для обработки потоков данных в реальном времени и хранения больших объемов событий с воз-

возможностью их долговременного хранения.

- **Redpanda:** - Redpanda — это высокопроизводительная, совместимая с Kafka, распределенная система сообщений, оптимизированная для работы с потоками данных в реальном времени. - **Назначение:** Обеспечивает низкую задержку и простоту настройки, что делает её предпочтительной для решений, требующих высокой производительности.
- **Apache Pulsar:** - Pulsar — это распределенная система потоковой передачи данных, которая поддерживает многопоточность и управление потоками. Она предоставляет гибкий подход к очередям сообщений и событиям. - **Назначение:** Отличается поддержкой долгосрочного хранения сообщений и сложных сценариев работы с геораспределёнными данными.

**Сравнение систем очередей сообщений** Ниже представлено сравнение преимуществ и недостатков Apache Kafka, Redpanda и Apache Pulsar.

Система	Преимущества	Недостатки
Apache Kafka	- Высокая надежность	- Сложность настройки и администрирования
	- Масштабируемость	- Требуется дополнительных ресурсов
	- Долговременное хранение данных	- Зависимость от Zookeeper
Redpanda	- Низкая задержка	- Меньшая экосистема поддержки
	- Простота настройки	
	- Совместимость с Kafka	
Apache Pulsar	- Гибкость	- Сложность архитектуры и настройки
	- Поддержка геораспределённых данных	- Меньшая популярность по сравнению с конкурентами
	- Высокая масштабируемость	

Таблица 1.3 – Сравнение систем очередей сообщений

**Заключение** Асинхронное взаимодействие, организованное через очереди сообщений, является эффективным способом повышения производительности и устойчивости микросервисов. Учитывая возможности по снижению задержки и простоты настройки, мы выбрали Redpanda как предпочтительное решение для организации асинхронного взаимодействия между микросервисами.

### 1.3.2 Хранение данных

**Реляционные базы данных (RDBMS):** Реляционные базы данных, такие как MySQL и PostgreSQL, предлагают структурированный способ хранения данных с использованием таблиц, что позволяет легко реализовать связи между ними. Они подходят для большинства приложений, требующих согласованности и целостности данных.

## Описание реляционных баз данных

1. **MySQL:** - **Описание:** MySQL — это популярная реляционная база данных с открытым исходным кодом, которая используется во множестве веб-приложений и сервисов. Она известна своей производительностью, простотой настройки и широким сообществом.

- **Назначение:** Широко используется для веб-приложений, особенно тех, которые требуют быстрого доступа к данным.

2. **PostgreSQL:** - **Описание:** PostgreSQL — это мощная объектно-реляционная база данных с открытым исходным кодом, которая поддерживает различные типы данных и расширенные функции, такие как работа с JSON и поиск по полнотекстовому содержимому.

- **Назначение:** Подходит для сложных и крупных приложений, требующих расширенной функциональности обработки данных.

**Сравнение MySQL и PostgreSQL** Ниже представлено сравнение преимуществ и недостатков MySQL и PostgreSQL.

Система	Преимущества	Недостатки
MySQL	- Высокая производительность	- Ограниченная поддержка сложных запросов
	- Простота настройки	- Менее гибкая работа с типами данных
	- Широкое сообщество и поддержка	- Нет полной поддержки ACID
PostgreSQL	- Поддержка сложных запросов и индексов	- Меньшая производительность
	- Расширенные функции для работы с данными	- Сложнее в настройке и администрировании
	- Полная поддержка ACID и транзакций	

Таблица 1.4 – Сравнение MySQL и PostgreSQL

## Заключение

Учитывая наши требования и специфику приложений, мы выбираем PostgreSQL как основное решение для хранения данных в нашей архитектуре.

### 1.3.3 Авторизация

В современной веб-разработке авторизация пользователей является важной частью обеспечения безопасности и управления доступом к ресурсам. Существует несколько подходов к авторизации, каждый из которых имеет свои особенности, преимущества и недостатки. Два самых популярных метода авторизации — это использование сессий и JSON Web Tokens (JWT). В этом обзоре мы рассмотрим основные аспекты каждого из этих подходов.

**Подход 1: Авторизация с использованием сессий** Авторизация с использованием сессий предполагает классический подход к управлению состоянием пользователя. Когда пользователь выполняет вход в систему, сервер создает сессию и сохраняет информа-

цию о пользователе, а также уникальный идентификатор сессии (обычно в виде куки) в браузере.

#### **Преимущества**

- **Безопасность:** Сессии могут быть безопаснее, так как данные о пользователе хранятся на сервере, и доступ к ним ограничен.
- **Управление сессиями:** Сервер может контролировать время жизни сессии и управлять активными сессиями (например, отключать их по запросу пользователя).
- **Простота реализации:** Для простых приложений реализация сессий может быть проще и привычнее для разработчиков.

#### **Недостатки**

- **Сложности с масштабированием:** В распределенных системах требуется дополнительная архитектура для хранения сессий (например, база данных или кэш), что может усложнить разработку.
- **Зависимость от состояния:** Поскольку сессии хранят состояние на сервере, это может ограничить возможность использования микросервисной архитектуры.

#### **Подход 2: Авторизация с использованием JWT**

JSON Web Tokens (JWT) — это подход к авторизации, основанный на использовании токенов. Когда пользователь входит в систему, сервер генерирует токен, который содержит зашифрованную информацию о пользователе и отправляет его клиенту. Клиент хранит этот токен и включает его в заголовки запросов при доступе к защищённым ресурсам.

#### **Преимущества**

- **Безсостояние:** JWT не требуют хранения состояния на сервере. Токен самостоятельно хранит информацию о пользователе, что упрощает масштабирование и работы в распределенных системах.
- **Безопасность:** Токены можно подписывать и шифровать, что добавляет уровень защиты данных.
- **Гибкость:** JWT можно использовать для разных типов приложений, включая веб-приложения, мобильные приложения и API.

#### **Недостатки**

- **Управление сроком действия:** JWT имеют фиксированный срок действия, и их нужно будет обновлять. Отзыв токена может быть сложнее реализовать.
- **Размер:** Токены могут быть крупнее, чем идентификаторы сессий, что может отрицательно сказаться на производительности при частом использовании в заголовках



запросов.

### **Заключение**

JWT более удобный и простой в реализации из-за того что не требуется хранение состояния на сервере.

## **1.4 Выводы**

1. Выполнен анализ архитектурных стилей построения систем, с учетом требований к динамической расширяемости отдельных модулей системы, выбрана микросервисная архитектура с применением API Gateway.
2. Проанализированы основные архитектурные паттерны построения приложений. Clean Architecture за счет преобразования SOLID, разбиения на слои и преимуществ Dependency inversion выбрана как основа написания микросервисов.
3. Проанализированных основные инструменты для разработки системы, произведены сравнения аналогов
  - для синхронного взаимодействия между микросервисами выбран gRPC, за счет строгой типизации контрактов и производительности за счет сериализации данных.
  - для асинхронного взаимодействия между микросервисами выбран подход с брокером сообщений, за счет универсальности и отказоустойчивости. В качестве реализации выбрана RedPanda, Kafka совместимое API с наибольшей производительностью.
  - для хранения структурированных данных выбрана PostgreSQL, за счет развитой надежности и удобства
  - для хранения неструктурированных данных выбрано NoSQL хранилище S3 Minio.
  - в качестве схемы авторизации выбрана схема с JWT токенами, которая позволяет не хранить активное состояние на серверной стороне.

## **1.5 Постановка задачи научно исследовательской работы**

1. Проектирование и разработка системы серверного приложения системы «Интеллектуальный ассистент врача УЗИ» на основе выбранных инструментов и подходов.
2. Тестирование и оценка разработанной системы и ее производительности. Устранение недостатков при наличии.

## 2. Теоретическая часть

### 2.1 Модель микросервиса

К каждому микросервису выделяются общие функциональные требования:

- При необходимости выполнять CRUD операции с базой данных
- При необходимости писать сообщения в логброкер сообщений
- При необходимости читать сообщения из логброкера сообщений

К каждому микросервису выделяются общие нефункциональные требования:

- Соответствовать чистой архитектуре
- Иметь высокий cohesion и низкие coupling

Исходя из этих требований, типовая архитектура нашего сервиса будет соответствовать приведенной.

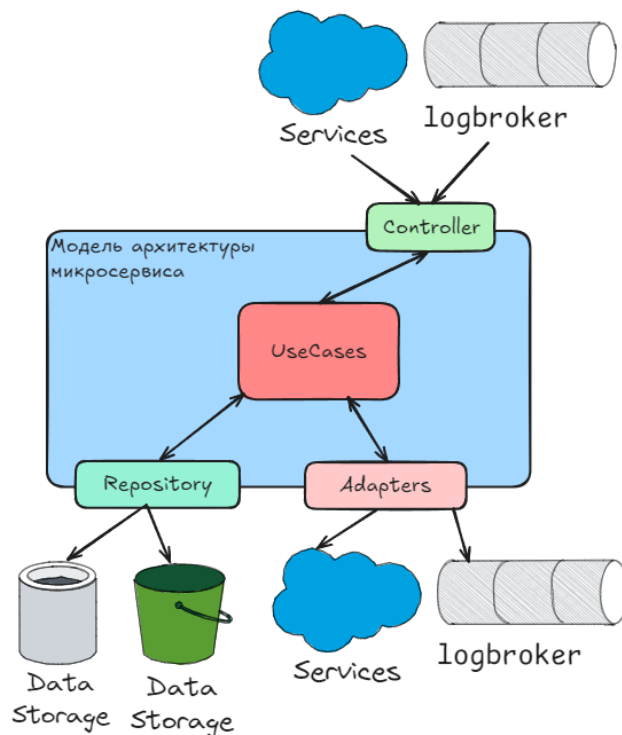


Рисунок 2.1 – Модель типового микросервиса

## 2.2 Модели модулей системы

### 2.2.1 Модель сервиса аутентификации

Микросервис аутентификации осуществляет регистрацию новых пользователей, и обновление токенов у уже существующих. Для существующих пользователей он выдает новую пару JWT tokens(access token + refresh token) по refresh token или логину и паролю.

Функциональные требования к сервису аутентификации:

- Возможность регистрировать новых пользователей в сервисе
- Подписывать JWT ключи для авторизации
- Обменивать логин пароль или refresh JWT ключ на новый refresh JWT ключ

Тогда для соответствия требованиям, модель базы данных:

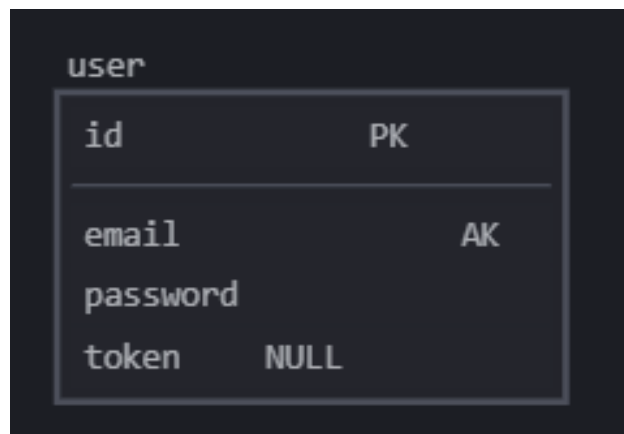


Рисунок 2.2 – Модель базы данных сервиса аутентификации

Для обмена refresh token на новую пару ключей, в токен необходимо зашивать ID пользователя. В соответствии с требованиями, модель сервиса авторизации:

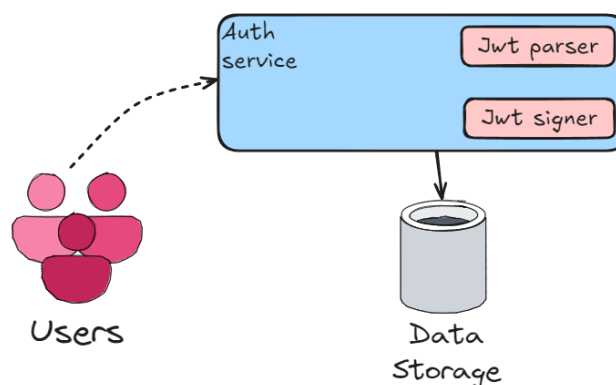


Рисунок 2.3 – Модель сервиса аутентификации

### 2.2.2 Модель сервиса узи

Для начала нужно определить вид данных которые должен хранить узи.

Узи представляет из себя последовательный набор кадров, конкретной щитовидной железы. На щитовидной железе могут быть обнаружены злокачественные образования, называемыми узлами. Узел злокачественного образования может быть запечатлен на нескольких кадрах, такое отображения узла на конкретном кадре называется **сегментом**, узлов может быть любое количество. Каждый узел и сегмент в отдельности характеризуется классификационными признаками. На данный момент имеется 3 классификационных признака: вероятности принадлежности узлов к классам **TI-RADS**. TI-RADS, описывает стадию злокачественного образования, существует 3 стадии: TI-RADS23, TI-RADS4, TI-RADS5.

Функциональные требования к сервису узи:

- Принимать единый набор композицию кадров узи, разбивать и сохранять ее по кадрам.
- После разбиения узи по кадрам, оповещать через брокер сообщений, сервис с ml моделью, о доступности обработки изображения
- Предоставлять возможность совершать CRUD операции над узлами, сегментами, узи, ti-rads, images.
- Сохранять результат сегментации и классификации нейронной моделью посредством логброкера сообщений.

Тогда для соответствия требованиям, модель базы данных:

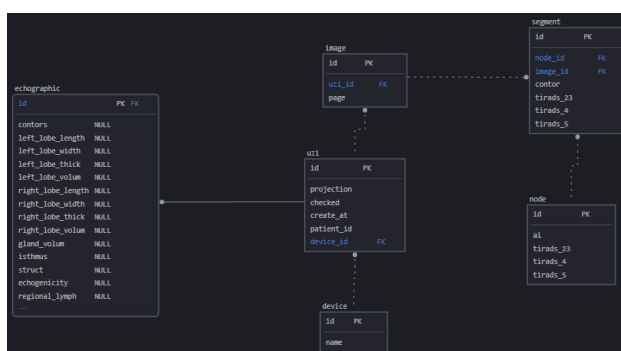


Рисунок 2.4 – Модель базы данных сервиса аутентификации

В соответствии с требованиями, модель сервиса узи:

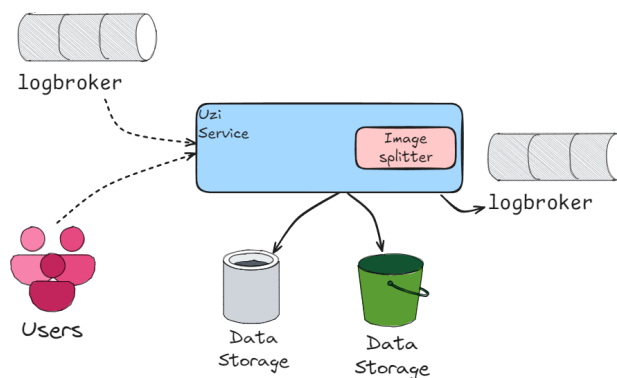


Рисунок 2.5 – Модель сервиса УЗИ

### 2.2.3 Модель Gateway API

Сервис gateway API, является точкой входа для всех пользователей системы. Сервис делает запросы в дочерние микросервисы.

Функциональные требования к сервису Gateway API:

- Если дочерний микросервис реализует публичный для пользователя контракт взаимодействия, то сервис должен реализовывать доступ к этому контракту для пользователя
- Предоставлять пользователю по запросу кадры узи
- Запускать пайплайн обработки узи, посредством сообщения в логброкер

В соответствии с требованиями, модель сервиса Gateway:

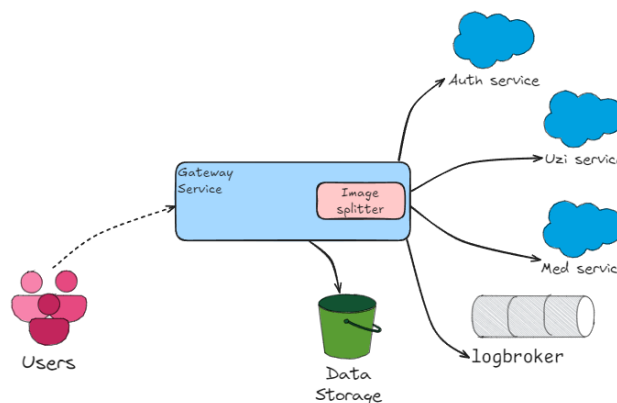


Рисунок 2.6 – Модель сервиса gateway

### 3. Результаты проектирования ...

В этой главе описывается, что и как было спроектировано. При необходимости, описывается использованная методика проектирования. Сюда же относится описание внешних и внутренних программных интерфейсов, а также форматы и структуры входных и выходных данных.

#### 3.1 Использование методики «такой-то» для проектирования программных систем «такого-то типа»

...

#### 3.2 Общая архитектура системы ...

...

#### 3.3 Архитектура подсистемы 1...

...

#### 3.4 Архитектура подсистемы $N$ ...

...

#### 3.5 Проектирование протокола взаимодействия подсистем $X$ и $Y$

...

#### 3.6 Выводы

Следует перечислить, какие инженерные результаты были получены, а именно: какие программные системы, подсистемы или модули были спроектированы. Следует не только назвать полученные архитектуры, но и отметить их отличительные особенности.

## 4. Реализация и экспериментальная проверка ...

В этой главе описывается, что и как было запрограммировано, отлажено, протестировано, и что в результате получилось. Большинство работ должны содержать приведенные ниже разделы. Но нужно учитывать, что точный состав этой главы, как и других глав, зависит от специфики работы.

Фрагменты программного кода в тексте необходимо выделять при помощи команды `\verb`. Многострочные листинги должны оформляться при помощи пакета `listings`.

Пример:

```
# let s x y z = x z (y z);;
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
# let k x y = x;;
val k : 'a -> 'b -> 'a = <fun>
# let i = s k k;;
val i : '_a -> '_a = <fun>
```

Листинг 4.1 иллюстрирует использование выносных листингов. Листинг 4.2 показывает пример включения внешнего файла в качестве листинга, в данном случае — выносного.

### 4.1 Выбор инструментальных средств

В этом разделе обосновывается выбор инструментальных средств; одним из критериев выбора могут быть какие-либо требования к разрабатываемой системе, и если этих требований много, они могут быть выделены в отдельный раздел, или же в приложение. Этот пункт не пишется, если в аналитической главе был раздел, посвященный сравнительному анализу и выбору инструментальных средств.

### 4.2 Состав и структура реализованного программного обеспечения

Нужно охарактеризовать реализованное ПО: является ли оно настольной программой для Windows, или веб-приложением в форме сайта/веб-сервиса, или модулем/подключаемой библиотекой, или .... Также нужно перечислить, из чего оно состоит: какие

---

#### Листинг 4.1 – Выносной листинг

```
List myList = new List();
Element myElement = new Element();
myList.Append(myElement);
```

---

#### Листинг 4.2 – Листинг из файла HelloWorld.scala

---

```
object HelloWorld {  
  def main(args: Array[String]) {  
    println("Hello, world!")  
  }  
}
```

---

исполняемые файлы и их назначение, конфигурационные файлы, файлы баз данных, требования к программному и аппаратному окружению, и т.п.

Если реализованное приложение достаточно обширно, этот раздел может быть разделен на несколько: один с общим описанием, и по одному на подсистемы самого верхнего уровня.

### 4.3 Основные сценарии работы пользователя

Нужно помнить, что пользователем может быть не только «менеджер» или «человек в белом халате», но и другой программист. Последнее относится, в первую очередь, к реализованным библиотекам. Для «обычных» приложений нередко бывают пользователи нескольких категорий — например, обычный пользователь и администратор. Для каждой категории нужно описать, как выполняются основные функции, предпочтительно, с помощью серии скрин-шотов. Однако считается плохим тоном вставлять длинную вереницу из скрин-шотов: если их много, большую часть нужно выносить в приложение. Для *этого* раздела нормальной является плотность скрин-шотов из расчета: 1 страница скрин-шотов на 1-2 страницы текста.

### 4.4 Разработка тестовых примеров

Описываются наиболее характерные тестовые примеры, для прогона на интеграционных тестах. (Да, использование unit-тестирования — это почти всегда хорошо, основное исключение составляют работы, в которых используемый инструментарий по какой-либо причине в принципе исключает такую возможность. Например, что-нибудь вроде Mathematica.)

В этом же разделе могут приводиться и результаты тестирования, включая таблицы и графики. Результаты тестирования могут быть вынесены в отдельный раздел, если много текстового материала и/или использована (не совсем) стандартная методика тестирования (описание которой также нужно привести).

**Замечание.** В ПЗ (как УИРа, так и ВКР) следует избегать ситуаций, когда значительную



часть основного содержания составляют страницы с иллюстрациями и таблицами, особенно, если такие страницы следуют подряд. В основном тексте следует оставлять лишь самые основные таблицы и рисунки, а остальное — выносить в приложение.

#### **4.5 Сравнение реализованного программного обеспечения с существующими аналогами**

В сравнении должно быть отражено, чем полученное ПО выгодно (и невыгодно) отличается от прочих ближайших аналогов. Практика показывает, что аналоги есть всегда. А если нет аналогов, значит есть частичные решения, которые реализуют какие-то части функционала вашей системы. Тут тоже может быть относительно много таблиц и графиков.

#### **4.6 Выводы**

Следует перечислить, какие практические результаты были получены, а именно: какое программное или иное обеспечение было создано. В число результатов могут входить, например, методики тестирования, тестовые примеры (для проверки корректности/оценки характеристик тех или иных алгоритмов) и др. По каждому результату следует сделать вывод, насколько он отличается от известных промышленных аналогов и исследовательских прототипов.

## Заключение

В заключении в тезисной форме необходимо отразить результаты работы:

- аналитические (что изучено/проанализировано);
- теоретические;
- инженерные (что спроектировано);
- практические (что реализовано/внедрено).

Примерная формула такая: по каждому указанному пункту приводится по 3-5 результатов, каждый результат излагается в объеме до 5 фраз или предложений.

Также есть смысл привести предполагаемые направления для будущей работы.

Общий объем заключения не должен превышать 1,5 страниц (1 страницы для УИРов).

## А. Основные правила форматирования

Текст пояснительной записки должен готовиться для печати на листах формата А4, использоваться должен шрифт с засечками (Roman; обычно — Times Roman или Times New Roman), 12 или 14 кегль. Размеры полей:

- верхнее: 20 мм.
- нижнее: 20 мм.
- левое: 10 мм.
- правое: 25 мм.

Нумероваться должны все страницы, начиная с первой (титульной), однако сами номера следует проставлять на страницах, начиная со страницы реферата. Номер следует проставлять внизу страницу (в центре).

Заголовки оформляются тем же шрифтом, что и основной текст (т.е., соответственно, Times Roman или Times New Roman). Для заголовков первого уровня размер шрифта может быть больше размера шрифта основного текста (обычно 14-16).

Все разделы текста: реферат, оглавление, введение, три главы основного содержания, список литературы, заключение, приложения — должны снабжаться содержательным заголовком и начинаться с новой страницы; сами заголовки следует при этом центрировать (заголовки параграфов и пунктов выравниваются по ширине). Следует обратить внимание, что заголовки всех разделов, кроме трех основных глав, регламентированы; заголовки трех основных глав должны быть содержательными и отражать суть соответствующей главы. Названия типа «Аналитическая часть» и «Теоретическая глава» — *недопустимы*.

Текст пояснительной записки может содержать рисунки и таблицы. Все рисунки и таблицы должны снабжаться номерами и подписями:

- нумерация рисунков и таблиц должна быть сквозная (но отдельная, т.к. для рисунков своя, для таблиц — своя);
- в случае большого количества иллюстраций/таблиц, допускается «вложенная» нумерация (т.е. таблицу/рисунок можно снабжать составным номером в формате

⟨номер главы⟩.⟨номер внутри главы⟩;

- подрисуночная подпись должна располагаться снизу по центру;

- название таблицы следует помещать над таблицей слева, без абзацного отступа в одну строку с ее номером через тире (ГОСТ 7.32-2001, п.6.6.1).

Здесь перечислены не все, а лишь основные требования к оформлению. Прочие требования — см. соответствующие ГОСТы.

Для того чтобы избежать больших отступов в списках, которые по умолчанию добавляют окружения `itemize` и `enumerate`, следует использовать `compactitem` (для маркированных списков) и `compactenum` (для нумерованных списков) из пакета `paralist`. Например:

- это;
- не нумерованный;
- список;
- без лишних промежутков.

И для нумерованных списков:

- 1) нумерованные списки;
- 2) пакета `paralist`;
- 3) еще и удобно настраивать;
- 4) (например, менять формат номера).

или

- а) это другой;
- в) нумерованный;
- г) список;
- д) без лишних промежутков;
- е) и с буквенной нумерацией.

А если хочется нумерацию сделать англоязычной, то нужно использовать окружение `otherlanguage` (таким образом: `\begin{otherlanguage}[numerals=latin]{russian}`)

- а) это другой;
- в) нумерованный;
- г) список;
- д) без лишних промежутков;
- е) и с буквенной нумерацией.

**Замечание.** По неизвестным причинам, переключения не происходит, хотя должно.

## В. Общая структура пояснительной записки

1. Титульный лист
2. Лист с подписями (только для ВКР)
3. Задание (в данном примере используется задание на диплом)
4. Отчет из Антиплагиата<sup>1</sup>
5. Реферат (всегда на отдельной стр.)
6. Оглавление. Начинается с новой страницы.
7. Введение
  - 7.1. Актуальность
  - 7.2. Новизна
  - 7.3. Оригинальная суть исследования
  - 7.4. Содержание ПЗ по главам (тезисно)
8. Аналитическая глава. Пишется в стиле *аналитического обзора*
9. Теоретическая и инженерная глава. Описываются использованные, доработанные и разработанные модели, алгоритмы, методы, и т.п. Кроме того, тут формулируется архитектура системы.
10. Инженерная глава. В этой главе следует отразить результаты проектирования, что, в общем случае, включает в себя следующие пункты:
  - 10.1. Описание используемой методики проектирования
  - 10.2. Общая архитектура системы
  - 10.3. Архитектура подсистемы [таких подразделов может быть несколько штук, по одному на каждую подсистему или модуль, требующую детальное рассмотрение]
  - 10.4. Проектирование внешних и внутренних интерфейсов/протоколов взаимодействия
11. Практическая глава. Описывается реализация, включая выбор инструментальных средств<sup>2</sup>. Типовое содержание:
  - 11.1. Состав и структура реализованного ПО

---

<sup>1</sup>Обычно, допускается до 30% заимствованного текста для работ бакалавров и до 20% – для работ магистров; см. соответствующие нормативные документы

<sup>2</sup>В тех случаях, когда (а) этот выбор имеет существенное значение для всей работы и (в) он не был, по каким-либо причинам, проделан в аналитической главе

- 11.2. Выбор инструментальных средств
- 11.3. Основные сценарии работы различных категорий пользователей
- 11.4. Результаты тестирования (разработка тестовых примеров, таблицы и графики результатов прогона)
- 11.5. Сравнение с существующими аналогами
- 12. Заключение
- 13. Список литературы
- 14. Приложения

Кроме того, в ПЗ могут включаться и такие разделы, как словарь терминов, список сокращений и др. В зависимости от предпочтений автора, могут помещаться как в начале ПЗ (до оглавления), так и в конце (после заключения, но до приложений).

**Замечания:**

1. На каждый элемент из списка литературы должна быть хотя бы одна ссылка в тексте.
2. Список литературы должен быть оформлен согласно ГОСТ [1; 2].
3. Минимальное количество источников для УИРов — 15–20 (для работ с большой аналитической и теоретической частью нормальное количество — 25–30 и более), для дипломов — соответственно, 30–35 и 35–60. Эти цифры существенны, т.к. «недобор», как правило, свидетельствует о не выполнении аналитической части работы и, следовательно, недостаточном владении предметом.
4. При подготовке РСПЗ рекомендуется вставлять уже наработанные к моменту подготовки РСПЗ материалы. Однако, в любом случае, каждый раздел должен начинаться с аннотации, заключенной в окружение \annotation. В пояснительной записке к диплому аннотации не нужны.
5. Между заголовком главы и первым разделом рекомендуется поместить один-два абзаца связанного текста с кратким содержанием (планом) главы.
6. Общее число и объем приложений не ограничивается. Объем ПЗ *без* приложений — 25–40 стр. для УИРов, и не менее 60–100 стр. для дипломов. Объем ПЗ не может быть меньше указанных размеров. Это означает, что студент не выполнил работу, или, как минимум, не удосужился подготовить адекватную ПЗ. Превышать верхние пределы также не желательно, в некоторых комиссиях это может восприниматься негативно; однако, в целом, небольшое превышение допустимо, если проделана действительно большая работа и получено много результатов (например, экспери-

ментальных, или получены нетривиальные аналитические или теоретические результаты).

7. ГОСТ требует, чтобы нумерация страниц начиналась с первой, титульной, страницы. При этом на самой титульной странице номер не печатается. В данном случае, номера также не проставляются на листах задания, а также на листе с подписями (для ВКР).

## **Список литературы**

1. ГОСТ Р 7.1-2003 Система стандартов по информации, библиотечному и издательскому делу. Библиографическая запись. Библиографическое описание. Общие требования и правила составления. — М. : Стандартинформ, 2004.
2. ГОСТ Р 7.0.5-2008 Система стандартов по информации, библиотечному и издательскому делу. Библиографическая ссылка. Общие требования и правила составления. — М. : Стандартинформ, 2008.

## Г. Правила использования шаблона

Настоящий шаблон все еще несколько несовершенен в плане оформления: например, неправильная нумерация приложений, и еще несколько нюансов. В последующих версиях это будет исправляться.

Ниже описана структура исходных текстов шаблона (и, соответственно, структура исходных текстов ПЗ).

Кодировка всех файлов — UTF8, и для сборки PDF документов следует использовать команду `xelatex`. При этом при работе через Sublime Text + LaTeXTools следует использовать вариант сборщика Basic Builder.

В репозитории несколько «головных» файлов, предназначенных для генерации документов на разных стадиях выполнения проекта.

- < >-1-task.tex — для генерации бланка задания;
- < >-2-rspz.tex — для генерации отчета с титульным листом для РСПЗ;
- < >-3-pz.tex — для генерации отчета с титульными листами для ПЗ.

Задача головных файлов — «склеить» вместе разные части ПЗ. Каждая часть (реферат, введение, каждая содержательная глава, заключение, библиография, приложения) выделяется в отдельный файл.

- thesis-abstract.tex — содержит аннотацию;
- thesis-intro.tex — содержит введение;
- thesis-chapter1.tex — текст первой главы;
- thesis-chapter2.tex — текст второй главы;
- thesis-chapter3.tex — текст третьей главы;
- thesis-bibl.tex — список литературы (только подключение к проекту);
- biblio.bib — собственно библиография (в формате BibTeX);
- thesis-conclusion.tex — заключение;
- thesis-appendix1.tex — первое приложение;
- thesis-appendix2.tex — второе приложение;

Другие файлы, используемые для настройки шаблона и определения параметров проекта:

- thesis-macro.tex — содержит определения различных макрокоманд, которые ча-



сто используются в конкретной работе, например, определения окружения для теорем, некоторые часто используемые формулы, и т. п.;

0-0-project-members.tex — информация о проекте: руководитель, студент, тема и т. п.;

0-1-task-data.tex — информация о задании;

\_content.tex — склеенное основное содержимое отчета, которое используется для генерации ПЗ и РСПЗ;

Головные файлы следует менять только в том случае, если требуется сгенерировать документ, изначально не предусмотренный в данном шаблоне. Если требуется добавить новые файлы к основному содержимому проекта — новый раздел отчета или приложения, следует вносить изменения в \_content.tex.

## Г.1 Титульные листы

Существует два варианта генерации титульных листов:

- использование листов, сверстанных в  $\text{\LaTeX}$  (используется по умолчанию);
- подстановка пустых бланков из PDF-файлов.

### Г.1.1 Титульные листы $\text{\LaTeX}$

Проект содержит определения титульных листов, описанные в виде .tex файлов. На данный момент требуется заполнение данных студента вручную. Позже будет реализована автоматическая подстановка данных проекта при инициализации репозитория.

### Г.1.2 Титульные листы из PDF

При возникновении проблем с использованием титульных листов  $\text{\LaTeX}$  возможно включить в документ бланки титульных листов из PDF-файлов. Для этого нужно раскомментировать соответствующие команды **includepdf** в начале документа.

Для того, чтобы  $\text{\LaTeX}$  при компиляции автоматически «подхватил» задание, его нужно сохранить в формате pdf (например, с помощью виртуального принтера), поместить в ту же папку /title и назвать task.pdf. Точно также следует поступить с титульной страницей (title.pdf). При оформлении ПЗ для ВКР следует дополнительно поместить в папку /title pdf-версию листа с подписями, назвав файл title-dep22.pdf. После этого нужно раскомментировать команду

```
\includepdf[ ... ]{title/title-dep22.pdf}
```

в начале головного файла.

Образцы и Word-шаблоны титульных листов для (РС)ПЗ к УИРам, НИРам, практикам и ВКР доступны в репозитории

<https://gitlab.com/skibcsit/thesis-titles>.

**Замечание.** В шаблоне используется пакет `hyperref`, который делает две вещи: все перекрестные ссылки «кликабельны», а также выделены (красной) рамочкой. Эти рамки *не выводятся на печать*. Вместо цветных рамок, возможны другие способы выделения ссылок (см. документацию пакета).