Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: ЖЦРПО

ОТЧЕТ по лабораторной работе № 3 на тему ИССЛЕДОВАНИЕ АРХИТЕКТУРНОГО РЕШЕНИЯ

Студенты: П.С. Хвесько
Т.Д. Таврель
Д.В. Василевич

Преподаватель: Д.А. Жалейко

МИНСК 2025

Часть 1. Проектирование архитектуры

1. Определение типа приложения

Разрабатываемая система представляет собой клиент-серверное приложение для передачи данных через протоколы TCP и UDP. Она позволяет пользователям:

- Загружать и скачивать файлы;
- Выполнять команды, такие как отправка и получение сообщений. Система реализована как многопоточное приложение, где сервер принимает запросы от клиентов, обрабатывает их и отправляет ответы. Это обеспечивает высокую производительность и возможность одновременного взаимодействия множества пользователей.

Система реализована как **клиент-серверное приложение**.

2. Выбор стратегии развёртывания

Наиболее подходящей стратегией развертывания является модель клиент-сервер. В данном подходе:

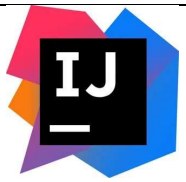
- **Серверная часть** (реализованная на Kotlin) отвечает за обработку входящих соединений, управление передачей файлов и выполнение команд.
- **Клиентская часть** (также написанная на Kotlin) предоставляет интерфейс для взаимодействия с сервером, позволяя пользователям отправлять команды и получать данные.

Основные компоненты системы:

- **Клиентская часть:**
  - *Kotlin -приложение*: для взаимодействия с сервером. для доступа к платформе.
  - *Стабильное интернет-соединение*: необходимо для корректного взаимодействия с сервером.
- **Приложение:**
  - *Фронтенд:* технологии Kotlin Compose/Desktop-Compose, используется для создания визуального интерфейса и интерактивных элементов.
  - *Бэкенд:* Kotlin для реализации серверной логики, обработки данных и управления бизнес-процессами.

- **Среда разработки:**
  - *IntelliJ IDEA:* для разработки на Kotlin

3. Обоснование выбора технологии

	**IntelliJ IDEA** — это высокоэффективная интегрированная среда разработки (IDE) для Java, Kotlin, используемая в проекте для создания клиент-серверного приложения, обеспечивающего передачу данных через протоколы TCP и UDP. Интуитивный интерфейс, интеллектуальное автозавершение кода и мощные инструменты рефакторинга значительно повышают производительность разработчиков, позволяя быстро и эффективно реализовывать функционал, необходимый для обработки команд и передачи файлов.
	**Kotlin** — это современный статически типизированный язык программирования, который полностью совместим с Java и активно используется в проекте для разработки клиент-серверного приложения. Он известен своим лаконичным и чистым синтаксисом, что облегчает чтение и написание кода. Kotlin предлагает множество удобных функций, таких как расширения, корутины для асинхронного программирования и безопасные по отношению к null типы, что способствует более надежной и эффективной разработке сетевых приложений для передачи данных через протоколы TCP и UDP.
	**TCP (Transmission Control Protocol)** — это надежный, ориентированный на соединения протокол передачи данных, используемый в нашем проекте для обеспечения стабильной и безопасной передачи информации между клиентом и сервером. Он гарантирует, что все данные будут доставлены в правильном порядке и без потерь, что критически важно для обмена командами и файлами.
	**UDP (User Datagram Protocol)** — это протокол передачи данных без установления соединения, используемый в нашем проекте для обеспечения быстрой и эффективной передачи информации между клиентом и сервером. Он подходит для сценариев, где важна скорость, а не абсолютная надежность.

Указание показателей качества (методология FURPS)

Functionality (Функциональные требования):

- Поддержка команд для загрузки и скачивания файлов.
- Обработка команд, таких как ECHO и TIME.
- Надежная передача данных через TCP и UDP.
- Поддержка мультимедийных материалов (видео, аудио, изображения).

Usability (Удобство использования):

- Интуитивно понятный и единообразный интерфейс.
- Быстрая и удобная навигация.
- Легкость в использовании команд для взаимодействия с сервером.
- Минимальное количество кликов для доступа к ключевому функционалу.

Reliability (Надёжность):

- Высокая готовность системы (90–99% времени безотказной работы).
- Возможность восстановления данных в случае сбоев.
- Обработка ошибок и исключений для устойчивой работы.
- Обеспечение отказоустойчивости: при отказе отдельных компонентов система продолжает работать с ограниченным функционалом.

Performance (Производительность):

- Средняя задержка отклика не превышает 2 секунд.
- Эффективное использование системных ресурсов.
- Поддержка одновременного доступа большого количества пользователей.
- Масштабируемость системы для роста нагрузки.

Supportability (Сопровождаемость):

- Возможность добавления нового функционала без значительных изменений существующего кода.
- Централизованное логирование и обработка ошибок.

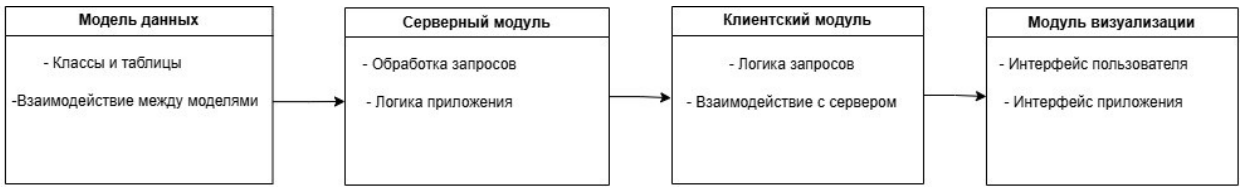
Обозначение путей реализации сквозной функциональности

Сквозная функциональность охватывает аспекты, затрагивающие несколько модулей системы. Для их реализации будут использованы следующие подходы:

- **Логирование и аудит:**
  Внедрение логирования для отслеживания активности пользователей и выявления ошибок..
- **Обработка исключений:**
  Создание единой системы для обработки ошибок и исключений.
- **Безопасность:**
  Реализация мер защиты от потенциальных угроз, таких как атаки через сеть.

Изображение структурной схемы приложения

Ниже представлена структурная схема «Архитектуры To Be», отображающая основные функциональные блоки и слои системы: **Рис. 1. Структурная схема приложения (To Be):**

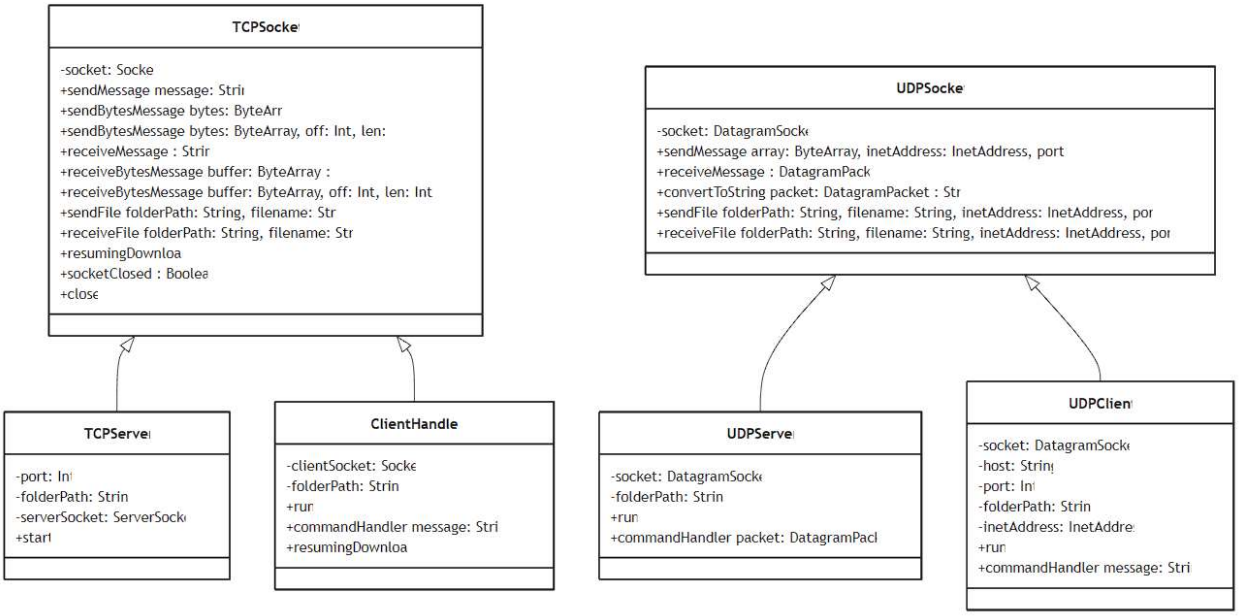


Примечание: Данная схема иллюстрирует основные слои приложения – от взаимодействия с пользователем до хранения данных.

Часть 2. Анализ архитектуры («As is»)

На данном этапе проведён анализ существующей архитектуры системы, сформированной в ходе первого Sprint Review. С использованием инструмента обратной инженерии (например, IBM Rational Rose) была сгенерирована диаграмма классов, отображающая структуру ключевых компонентов.

Рис. 2. Диаграмма классов (As is):



Примечание: Диаграмма демонстрирует основные модели системы:

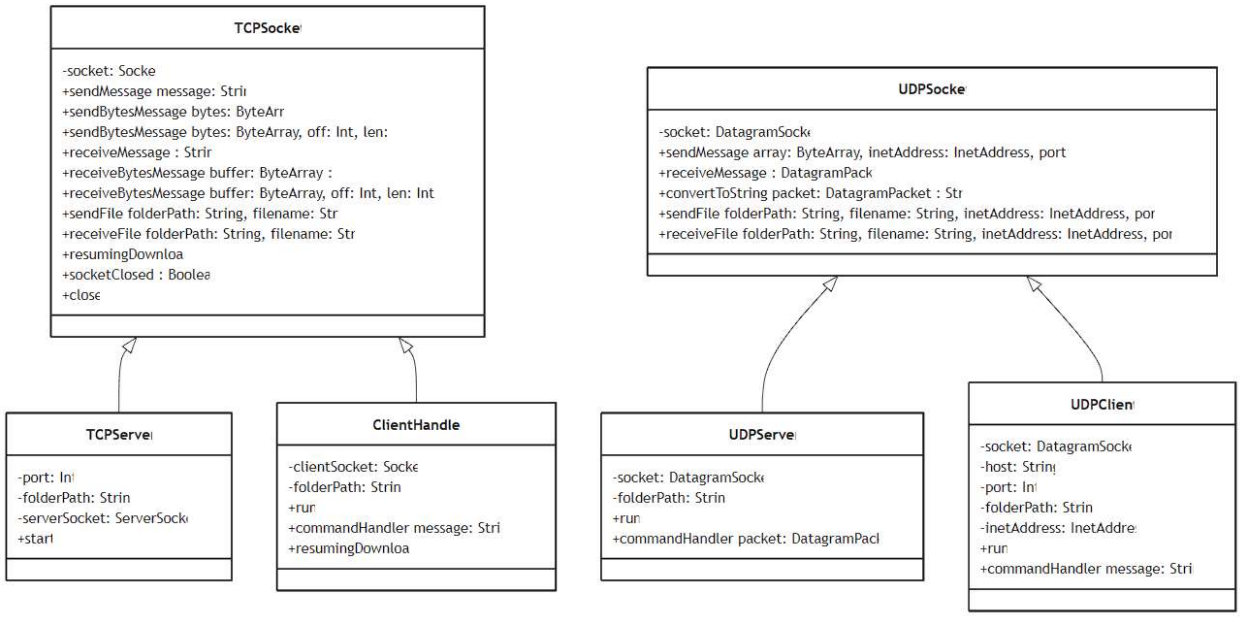
- **User** – отвечает за данные пользователей.
- **UserProfile** – хранит дополнительную информацию о пользователях.
- **Course** – представляет курсы, доступные на платформе.
- **Lesson** – содержит информацию об уроках, входящих в курсы.
- **Test** и **Question** – обеспечивают функциональность проведения тестирования.

Связи между классами отражают их отношения, выявленные в исходном коде.

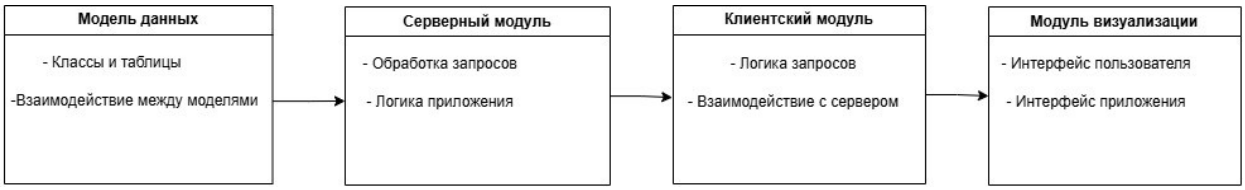
Часть 3. Сравнение и рефакторинг

Ниже представлены две блок-схеммы, отражающие текущее («As Is») и проектируемое («To Be») состояния системы обучения. Первая иллюстрирует

целевой вариант архитектуры и процессов, вторая – текущее (фактическое) положение дел в уже разработанных модулях.



Блок-схема «As Is»



Блок-схема «To Be»

1. Сравнение архитектур «As is» и «To be»

- **Модульность и разделение ответственности:**
  As is: В текущей архитектуре может наблюдаться недостаточное разделение между модулями, что приводит к смешиванию логики и усложняет поддержку. Взаимодействие между различными компонентами может быть неструктурированным.
  To be: Предлагаемая архитектура чётко разделяет ответственность между модулями: модуль данных (таблицы и взаимодействие), серверный модуль (обработка запросов и бизнес-логика), клиентский модуль (логика запросов и взаимодействие с сервером), и модуль визуализации (интерфейс пользователя). Это упрощает масштабирование и поддержку системы.

- **Сквозная функциональность:**
  As is: Функциональности, такие как обработка запросов и взаимодействие с сокетами, могут быть реализованы фрагментарно, что приводит к дублированию кода и усложнению логики.
  To be: В проектируемой архитектуре централизация функций, таких как обработка запросов через TCP/UDP сокеты, обеспечивается через чётко определённые модули, что способствует лучшему управлению и снижению дублирования кода.
- **Применение шаблонов проектирования:**
  As is: В существующей архитектуре могут отсутствовать проверенные шаблоны проектирования, что затрудняет тестирование и расширение системы.
  To be: В новой архитектуре используются современные подходы, такие как разделение на серверный и клиентский модули, а также чёткое определение взаимодействия через TCP и UDP сокеты, что улучшает тестируемость и расширяемость системы.

2. Анализ причин отличий

- **Эволюция системы:**
  Архитектура «As is» формировалась в ходе быстрого прототипирования, где приоритетом была скорость разработки, а не соблюдение принципов модульности и сквозной функциональности.
- **Ограниченные сроки:**
  На ранних этапах разработки не уделялось достаточное внимание внедрению единых решений для сквозных аспектов, что привело к фрагментарной реализации функций
- **Отсутствие чётких стандартов:**
  Недостаток использования единых архитектурных стандартов и шаблонов проектирования способствовал появлению рассогласованностей в структуре кода.

3. Пути улучшения архитектуры (Рефакторинг)

Для повышения качества архитектурного решения рекомендуется:

- **Улучшить модульность и разделение ответственности:**
  Улучшить модульность и разделение ответственности:
  Провести рефакторинг, выделив отдельные модули для данных, серверной логики, клиентских запросов и визуализации. Внедрить

четкое разделение между TCP и UDP компонентами.

- **Централизовать сквозные функции:**
  Внедрить единый модуль для обработки запросов и управления сокетами, чтобы избежать дублирования кода и упростить поддержку.
- **Использовать шаблоны проектирования:**
  Применять шаблоны, такие как Service и Repository, для повышения тестируемости и снижения связности между модулями, что облегчит масштабирование системы.
- **Оптимизировать производительность:**
  Реализовать кэширование для запросов к серверу и оптимизировать взаимодействие с сокетами для повышения скорости обработки запросов.
- **Стандартизировать код и документировать архитектуру:**
  Разработать стандарты кодирования и ведение документации по архитектурным решениям, чтобы обеспечить единообразие разработки и ускорить интеграцию новых участников в проект.

Вывод

В результате проведённого исследования были выявлены основные различия между существующей («As is») и проектируемой («To be») архитектурами. В проектируемой архитектуре достигнуто лучшее разделение ответственности, централизовано реализована сквозная функциональность и заложены основы для масштабируемости и поддержки системы. Анализ существующего решения выявил слабые места, связанные с высокой связностью компонентов и фрагментарной реализацией критически важных функций. Предложенные пути рефакторинга, включая применение шаблонов проектирования и централизованное управление сквозными аспектами, позволят повысить надёжность, производительность и сопровождаемость системы.