# Обновлённый и расширенный дорожный план проекта (Roadmap)

## 0.3.0-SNAPSHOT — Event-driven архитектура (выполнено)

**Цель:** Развязать сетевой слой и UI через событийную архитектуру. Это упрощает поддержку и расширение, соблюдая принцип *KISS* (“проще значит лучше”).

**Реализовано в 0.3.0:**  
- **EventBus:** Введён интерфейс EventBus и простая реализация SimpleEventBus. Этот шиной событий пользуется клиент для передачи сообщений между компонентами без прямых вызовов. (Реализовано, используется в коде).  
- **Базовые события:** Созданы классы событий: MessageReceivedEvent, MessageSendEvent, UserConnectedEvent, UserDisconnectedEvent. Они представляют приход сообщения, запрос на отправку, подключение пользователя и отключение пользователя соответственно. (Выполнено).  
- **ChatClientConnector:** Переписан на работу через EventBus. Теперь он подписывается на событие отправки сообщения (MessageSendEvent) и при его получении отправляет сообщение на сервер. Также при получении сообщения от сервера создает и публикует событие MessageReceivedEvent для UI. (Выполнено; в коде подключение через eventBus.subscribe() реализовано).  
- **ChatClientUI:** Теперь не обращается к сокету напрямую. Вместо этого при нажатии «Send» публикует событие MessageSendEvent с текстом. Также подписывается на MessageReceivedEvent и обновляет текстовое поле чата при получении новых сообщений. (Выполнено, UI декомпозирован от сетевого кода).

*Замечания:* Данная версия успешно избавилась от жёсткой связи между UI и сетью. Код стал проще тестировать и изменять, следуя *Single Responsibility Principle* (класс UI отвечает только за интерфейс, а Connector – за сеть). Библиотека **replines** используется для командной обработки (CommandHandler), что соответствует паттерну «Команда» и также вписывается в событийную модель.

## 0.3.1-SNAPSHOT — Системные события и список пользователей (частично выполнено)

**Цель:** Расширить событийную модель для системных оповещений (вход/выход пользователей) и отображать активных пользователей. Сделать вывод таких сообщений особым стилем в UI.

**Сделано в 0.3.1:**  
- **События входа/выхода:** При подключении клиента ChatClientConnector публикует событие UserConnectedEvent, а при отключении – UserDisconnectedEvent. Сервер при входе пользователя рассылает системное сообщение (через общий Message с Type.SYSTEM: *«[SERVER] X entered the chat»*). Аналогично при выходе. Это обеспечивает уведомление всех клиентов о присоединении/отсоединении участников. (Реализовано: сервер рассылает, клиент получает как системное сообщение).  
- **Обработка на клиенте:** События UserConnectedEvent/UserDisconnectedEvent пока напрямую UI не обрабатываются (они создаются, но в UI нет отдельного подписчика на них). Тем не менее, приходящие системные сообщения отображаются в общем окне чата с префиксом [SYSTEM]. (Частично реализовано: функционально оповещения есть, но UI обрабатывает их как обычные сообщения).

**Что не реализовано (планируется):**  
- **Особый стиль для системных сообщений:** Пока сообщения типа SYSTEM выводятся тем же стилем, что и обычные. Планируется отображать их другим цветом и курсивом, чтобы визуально отличать события (например, вход/выход) от пользовательских реплик. Для этого можно использовать JTextPane или HTML-вью, либо добавлять в JTextArea текст, помеченный стилем. (*Предложение:* определить в Message флаг или тип для системных сообщений и при добавлении в чат применять к ним иной формат шрифта).  
- **Список активных пользователей:** Отсутствует отображение текущих участников чата. Предлагается вести на клиенте список пользователей и обновлять его при событиях входа/выхода. Например, при событии UserConnectedEvent(username) добавлять имя в список, при UserDisconnectedEvent(username) – удалять. Первоначальный список можно получать от сервера при подключении (сервер может посылать новый тип сообщения с текущим списком пользователей либо последовательность UserConnectedEvent для всех уже находящихся в чате). Реализация этой функции запланирована.  
- **Улучшение серверной логики:** Сейчас сервер просто рассылает строковые оповещения о входе/выходе. В будущем можно унифицировать это с EventBus-системой (например, ввести на сервере собственный EventBus или особый класс сообщений для системных событий). Пока это не критично (KISS: простой broadcast работает), но при расширении функционала может понадобиться более структурированная передача информации (например, для передачи списка пользователей сразу).

*Замечания:* Функционал 0.3.1 частично готов: уведомления о присоединениях/выходах работают (видны как [SYSTEM] ... сообщения). Однако UI можно улучшить, чтобы эти уведомления были более заметны. Также стоит реализовать отображение активных пользователей — это повысит информативность интерфейса. В остальном архитектура событий зарекомендовала себя хорошо: новые типы событий добавляются без ломки существующего кода (соответствует *Open/Closed Principle* — система расширяется событиями, не требуя модификации базовых классов).

## 0.4.0-SNAPSHOT — Передача файлов (планируется)

**Цель:** Добавить возможность отправлять файлы через чат. Пользователи смогут пересылать изображения, документы и пр. Файлы будут передаваться по сети с отображением прогресса передачи.

**Планируемая реализация:**  
- **Новые события:** Ввести события FileSendEvent и FileReceivedEvent. По аналогии с текстовыми сообщениями: при выборе файла для отправки UI публикует FileSendEvent (содержит файл или его путь), а сетевой слой перехватывает и отправляет файл адресатам; получив файл, клиент публикует FileReceivedEvent для UI.  
- **Протокол передачи:** Использовать тот же TCP-сокет для пересылки файлов. Проще всего сначала реализовать отправку файла целиком как байтовый поток через существующий ObjectOutputStream. Например, упаковать файл в объект сообщения (Message с типом FILE и содержимым в виде массива байт или потока). *KISS-реализация:* для начала можно ограничить размер файла или читать файл полностью в память, отправляя одним объектом (простота в ущерб памяти). Позднее, для больших файлов, потребуется передача по частям.  
- **Отображение прогресса:** В UI при начале отправки генерировать индикатор (progress bar). По мере передачи данных можно обновлять прогресс. Процент прогресса можно вычислять, если заранее известен размер файла. Например, отправлять сначала метаданные (имя файла, размер), затем поток данных. Клиент, получая файл, также отображает прогресс загрузки, либо по завершении информирует о получении и сохраняет файл.  
- **Обработка на сервере:** Сервер будет получать файл от отправителя и пересылать всем либо адресату. Так как чат пока общедоступный (нет приватных сообщений), предполагается, что файл рассылается всем участникам, аналогично текстовому сообщению. Синхронность: чтобы не блокировать прием других сообщений, можно файл передавать параллельно (например, временно не читать новые объекты, пока не передан текущий файл), или лучше – пересылать по частям и одновременно принимать другие мелкие сообщения? Здесь можно начать с простого: блокирующая передача, раз у нас модель с потоками на каждого клиента. Это проще (KISS) и для небольшого числа пользователей приемлемо.  
- **UI при получении:** Когда FileReceivedEvent получен, клиент может автоматически сохранить файл (например, в temp или в выбранную папку) и показать сообщение типа “[SYSTEM] Получен файл *filename* (сохранён в ...)” или же отображать превью (если изображение). Это можно улучшать постепенно.

*Замечания:* Данная функциональность увеличит нагрузку на сеть и память, поэтому **важно** не нарушить принцип *DRY* при добавлении кода передачи файлов. Следует по возможности переиспользовать логику отправки/приема (например, расширить класс Message или создать общий интерфейс MessageTransporter). Также, чтобы не усложнять преждевременно (*YAGNI*), можно сначала реализовать передачу файлов без шифрования и предварительной оптимизации – просто убедиться, что механизм работает, а затем улучшать (например, прогрессбар, большие файлы, потоковая передача).

## 0.5.0-SNAPSHOT — Голосовые сообщения (планируется)

**Цель:** Реализовать передачу голосовых сообщений, позволяя пользователям отправлять короткие аудиозаписи через чат. Это расширит способы коммуникации (аналогично голосовым сообщениям в мессенджерах).

**Планируемая реализация:**  
- **События:** Добавить VoiceSendEvent и VoiceReceivedEvent для отправки/получения голосовых сообщений. Их обработка схожа с FileEvent, но специфична для аудио.  
- **Запись звука:** Использовать Java Sound API (например, TargetDataLine) для захвата аудио с микрофона. При нажатии кнопки «Записать» клиент записывает звук (PCM WAV или другой формат) в буфер/файл. Для простоты можно ограничиться короткими записями (например, до 30 секунд) и сохранять их во временный файл.  
- **Отправка аудио:** После завершения записи публикуется VoiceSendEvent с содержимым аудио (файл или байтовый массив). Сетевой слой передает аудио аналогично файлу – либо как отдельный тип сообщения, либо reuse FileSendEvent с пометкой, что это аудио. Поскольку аудио обычно небольшое, можно переслать целым куском (если 30с записи ~ несколько сотен KB). Для *более продвинутого подхода*, обсуждаемого в 0.6.x, можно рассмотреть UDP для потоковой передачи, но на этапе 0.5.x можно отправлять как файл (через TCP) для надёжности и простоты.  
- **Получение и воспроизведение:** Клиент, получив VoiceReceivedEvent (с аудио-данными), сохраняет аудиофайл и сразу воспроизводит его (Java Sound API, Clip или SourceDataLine). UI может добавить в чат специальный элемент (например, кнопка «▶️ Play», или автоматически проигрывать). Проигрывание должно быть в отдельном потоке, чтобы не блокировать UI.  
- **UX аспекты:** Ввести ограничение на длину сообщения, кнопку записи/остановки. Отобразить в чате, что пользователь отправил голосовое (например, текст “[Voice message]” или иконку). Возможно, реализовать для получателя возможность прослушать повторно (хранить локально полученный файл).

*Замечания:* Голосовые сообщения – сложнее, так как вовлекают работу с аппаратурой (микрофон, динамики) и потенциально требуют *реального времени*. Принцип *KISS* подсказывает: на первом шаге лучше реализовать запись и отправку *по завершении записи*, не стремясь к прямому стримингу аудио. Затем, если понадобится мгновенная голосовая связь, можно будет подумать о UDP. Кроме того, важно не дублировать код файловой отправки: можно обобщить передачу бинарных данных (файл/аудио) через единый механизм, сделав, например, базовый класс или метод для отправки блоба данных. Это поддержит *DRY* и упростит дальнейшее сопровождение.

## 0.6.0-SNAPSHOT — Новый протокол передачи данных (планируется)

**Цель:** Улучшить формат обмена сообщениями и протокол связи. Уйти от Java-сериализации (ObjectOutputStream) в сторону текстового протокола (JSON) или иных оптимальных форматов. Также рассмотреть использование UDP для передачи потоковых данных (голос) и WebSocket для универсальности.

**Запланированные изменения:**  
- **JSON вместо Serializable:** Реализовать сериализацию сообщений в JSON (например, с помощью библиотеки Jackson или Gson). Каждое сообщение (текст, системное, файл, аудио) можно представить в виде JSON-объекта с полями: type, from, content (и доп. полями для файлов, например filename). Сервер будет читать строку JSON, парсить в объект Message (или обрабатывать без создания объекта, в зависимости от реализации). Клиенты – аналогично.  
- *Преимущества:* JSON читаем человеком, протокол независим от языка (позволит в будущем писать клиенты на других языках, или интегрироваться с вебом). Проще отлаживать (можно видеть содержимое сообщений).  
- *Недостатки:* Требует добавить парсинг/форматирование, чуть усложняет код. Скорость JSON достаточная для чата, но для аудио/файлов можно комбинировать: JSON для метаданных, а бинарные данные отправлять отдельно (например, Base64 или в сыром виде по другому каналу).  
- **Обновление архитектуры клиента/сервера:** Придётся заменить ObjectInput/OutputStream на, скажем, BufferedReader/Writer для строк JSON. Возможно, стоит реализовать слой **Transport** – отдельный класс, инкапсулирующий логику кодирования/декодирования сообщений. Это пример принципа *Dependency Inversion* – чтобы остальной код (EventBus, обработчики) не зависел от конкретного способа пересылки.  
- **TCP vs UDP:** Оставить TCP для надежной доставки важной информации (текст, файлы). Для голосовых сообщений **можно** использовать UDP, чтобы снизить задержки. Реализация: ввести отдельный сокет UDP на клиента и сервер, по которому отправлять аудио-пакеты. Альтернатива – продолжать слать аудио как файл по TCP (проще, но может быть задержки). Решение может быть гибким: сделать настройку, какой транспорт использовать для аудио.  
- *Паттерн Стратегия:* Можно применить для выбора транспорта – например, создать интерфейс VoiceTransport с реализациями VoiceViaTCP и VoiceViaUDP, переключаемыми в зависимости от настроек или автоматически. Это позволит легко менять способ доставки, соблюдая *Open/Closed Principle*.  
- **WebSocket (опционально):** Рассмотреть перевод протокола на WebSocket (особенно, если планируется веб или мобильный клиент). WebSocket объединяет преимущества: работает поверх TCP, удобен для браузеров. Возможный план: после 1.0.0, реализовать сервер на Spring Boot с поддержкой STOMP/WebSocket, а текущий толстый клиент оставить как есть или переключить на WebSocket. Пока 0.6.0 фокусируется на JSON поверх существующего сокета, WebSocket можно оставить на потом (указано как опция в плане).

*Замечания:* Переход на новый протокол — серьезное изменение, требующее тщательного тестирования. Здесь важно не нарушить работающий функционал (**не “сломать” то, что уже работает**, следуя принципу *Open/Closed и Liskov* — новые изменения не должны менять ожидаемое поведение старого кода). Может быть разумно сделать 0.6.x веткой, параллельно поддерживая старый протокол, и переключить клиентов и сервер на новый формат после отладки. Также, внедряя JSON, мы готовимся к масштабированию и поддержке мобильных/веб-клиентов в будущем (что упоминается после 1.0.0).

## 0.7.0-SNAPSHOT — Шифрование переписки (планируется)

**Цель:** Обеспечить конфиденциальность сообщений. Добавить возможность шифровать сообщения (минимум текст, можно также файлы) с помощью симметричного шифрования (AES). Пользователь должен иметь опцию включить/выключить шифрование.

**Основные шаги:**  
- **Выбор алгоритма:** Использовать алгоритм AES-256 для шифрования сообщений. Ключ шифрования может быть предустановлен (например, фиксированный для всех клиентов на время сессии) или вводиться пользователем (лучше – генерировать при старте чата и обмениваться через безопасный канал или предварительно известный пароль). Для простоты можно начать с фиксированного ключа, зашитого в приложении, или вводимого в UI (KISS для начальной реализации).  
- **EncryptionEvent:** Ввести событие (например, EncryptionToggleEvent) или параметр конфигурации, позволяющий включать/отключать шифрование. Когда шифрование включено, **все исходящие сообщения** перед отправкой шифруются, а входящие – расшифровываются.  
- **Реализация на клиенте и сервере:** - На клиенте: перед отправкой текста (в ChatClientConnector) проверять флаг “encryptionEnabled”. Если true, шифровать содержимое сообщения (например, в Base64 строку после шифрования) и помечать сообщение как зашифрованное. Сообщения типа FILE или VOICE тоже можно шифровать (для файлов – шифровать байты; для аудио – аналогично файлам).  
- На сервере: в идеале сервер не должен расшифровывать содержимое, а просто транслировать как есть (концепция end-to-end шифрования). То есть сервер получит зашифрованные данные и отправит всем в том же виде. Клиенты-получатели сами расшифруют. Таким образом, даже сервер не знает содержания (повышает конфиденциальность). Это самый простой путь, т.к. не требует хранения ключей на сервере. Альтернативно, можно шифровать на уровне транспортного соединения (SSL/TLS), но в рамках проекта проще именно на уровне сообщений.  
- **UI/UX:** Добавить в интерфейс галочку или настройку «🔒 Шифрование: Вкл/Выкл». При активном шифровании, можно отображать значок замка рядом с сообщениями или надпись “[Encrypted]” для ясности. Если кто-то не включил шифрование, его клиент будет видеть бессмысленный шифртекст (поэтому, возможно, надо договориться, что все клиенты используют одинаковый ключ или централизованно включать для всех). В рамках простой реализации – можно считать, что все участники используют одну настройку (например, обмениваются ключом заранее).  
- **Ключи и безопасность:** Для учебного/первичного варианта не будем усложнять протоколом обмена ключами (YAGNI – лишняя сложность сейчас). Но важно: хранить ключ безопасно, не логировать. При рефакторинге можно выносить шифрование в отдельный модуль/класс (паттерн **Strategy**: шифратор/дешифратор, который можно подменить или отключить).

*Замечания:* Шифрование добавляет накладные расходы: сообщения станут двоичными или base64, увеличатся в размере. Но AES быстро работает для текстов и небольших файлов – это не критично. Важно соблюдать *Single Responsibility*: вынести логику шифрования из основных классов (например, в утилитный класс EncryptionService), чтобы не размазывать код шифрования повсюду. Также стоит помнить про *Open/Closed*: хорошо, если добавление новых алгоритмов или отключение шифрования не потребует модификации существующего кода (например, через интерфейс Encryptor).

## 0.8.0-SNAPSHOT — Авторизация и права доступа (планируется)

**Цель:** Добавить авторизацию пользователей в чате. Ввод логина/пароля при подключении, валидация на сервере. Также ввести базовое разграничение прав (например, администратор/пользователь) для демонстрации управляемости.

**Основные шаги:**  
- **События входа/выхода:** Добавить LoginEvent / LogoutEvent. При старте приложение сначала отображает диалог входа (вместо простого ввода имени как сейчас). Пользователь вводит имя и пароль. Эти данные отправляются на сервер, который проверяет корректность. Если успех – рассылается UserConnectedEvent как раньше, и клиент заходит в чат. Если нет – сервер отправляет отказ (например, новый тип сообщения ErrorMessage), и клиент показывает ошибку и повторный запрос логина.  
- **Хранение учетных записей:** Для простоты, можно захардкодить несколько пар “логин-пароль” или хранить их в конфиг-файле на сервере. В идеале – использовать базу данных или файл (SQLite или просто properties). Но на 0.8.x можно и в памяти держать. *KISS:* начать с простого списка пользователей на сервере.  
- **Реализация на сервере:** Сервер расширяется логикой проверки: при новом подключении вместо автоматического входа, ждет от клиента сообщение с типом LOGIN (с полями username/password). Проверяет. Если ок – добавляет этого клиента в список активных и отправляет ему подтверждение + текущую историю или список участников. Если не ок – может закрыть соединение или сообщить об ошибке и ждать повторный запрос.  
- **Сессии и права:** Каждому подключенному соответствует сессия (в простейшем случае – сам факт соединения с привязанным именем). Можно пометить некоторых пользователей как админов (например, если username == "admin"). Права доступа могут выражаться в командах – например, команда /kick <user> доступна только администратору, которая отключает указанного пользователя. Реализовать базово: в CommandHandler проверять, что текущий пользователь админ, если выполняет административную команду.  
- **Обновление UI:** Диалог входа (JOptionPane или отдельное окно) – реализовать ввод пароля с маскировкой. В основном окне чата можно отображать имя вошедшего пользователя (например, в заголовке окна уже есть). Если реализуются админ-функции, админа можно помечать (например, звездочка возле его имени в списке пользователей, когда появится).

*Замечания:* Авторизация добавляет сложность, поэтому важно не нарушить *KISS* – можно реализовать минимально работоспособную схему. Например, пока без регистрации (только заранее известные учетные данные). Обмен паролем лучше проводить по шифрованному соединению или шифровать пароль хотя бы тем же AES при передаче (чтобы не слать в открытом виде). Это пересекается с предыдущим этапом шифрования: если он реализован, то логин уже будет защищён. Стоит также помнить принцип *YAGNI* – не реализовывать сверхсложную систему прав, а ограничиться тем, что понадобится (например, 1-2 роли).

## 0.9.0-SNAPSHOT — Контакты и история сообщений (планируется)

**Цель:** Улучшить пользовательский опыт: хранить историю чата и предоставлять функции контактов и, возможно, приватных чатов. Использовать локальную базу данных для сохранения истории сообщений.

**Основные функции:**  
- **Локальная история (SQLite):** Интегрировать SQLite (через JDBC или встроенную библиотеку) в клиентское приложение для хранения истории. Каждый раз при получении нового Message сохранять его (автор, содержание, время) в локальную БД. При запуске клиента после логина можно загружать последние N сообщений из истории и отображать, чтобы пользователь видел предыдущий контекст чата. Это особенно полезно, если чат работает постоянно и пользователь переподключается.  
- **Команда /history:** Добавить команду (через CommandHandler) для вывода определённой истории, например /history 20 – показать последние 20 сообщений. Либо при инициализации UI автоматически подгружать историю. Реализовать событие HistoryEvent для передачи исторических сообщений UI.  
- **Контакты и приватные сообщения:** Ввести понятие списка контактов – возможно, в рамках нашего группового чата это список всех пользователей (как адресная книга). Группы – неявно сейчас одна общая “комната”. Можно расширить: реализовать приватные сообщения командой типа /w <user> <text> (whisper). Для этого в сообщении нужно указывать получателя. Можно добавить поле to в класс Message. Если to не null, сервер отправляет сообщение только этому пользователю (а не всем). Контакты могут храниться как список известных пользователей; можно сохранять в SQLite или простым списком.  
- **UI для контактов:** Добавить панель или список пользователей (полученный в 0.3.1/0.8.0) – это фактически список контактов онлайн. Можно позволить двойным щелчком по пользователю открыть приватный диалог (новое окно чата с этим пользователем, P2P). Однако это уже значительное расширение – возможно, за пределами объёма 0.9.0, если придерживаться YAGNI. На минимальном уровне, “контакты” могут означать просто отображение списка всех участников и возможность отправить личное сообщение через команду.  
- **Группы:** Упоминаются в плане, но, вероятно, подразумеваются групповые чаты. Это сложнее – нужно поддерживать несколько комнат. Возможно, пока просто отметить как потенциальную фичу на будущее (создание новых комнат/каналов).

*Замечания:* Хранение истории улучшает UX, но нужно учитывать синхронизацию: локальная история у каждого своя и может пропустить сообщения, если клиент офлайн. Можно расширить сервер, чтобы по запросу выдавал историю (например, хранить последние 100 сообщений сервер-side). Но в 0.9.x проще реализовать локально (каждый клиент сохраняет что видел). При разработке важно соблюсти *SRP*: например, вынести работу с БД в отдельный класс HistoryRepository, не мешая его с UI или сетевым кодом. Это облегчит поддержку.

## 1.0.0 — Первый стабильный релиз (запланировано)

**Цель:** Свести воедино все реализованные функции, отладить и выпустить стабильную версию чата. Обеспечить удобство запуска (для пользователя – минимальные усилия).

**Основные вехи к 1.0.0:**  
- **Полноценный GUI:** Приложение клиента должно иметь удобный графический интерфейс, включающий отображение чата, списка пользователей, поля ввода, кнопок для файлов/голоса, настроек (шифрование, и пр.). Все функции, реализованные ранее, интегрированы в UI. Например, кнопка «Отправить файл», кнопка «Записать голос», индикаторы шифрования, меню «История» и т.д.  
- **Системные сообщения:** Отображаются с особым форматированием (цвет, курсив), как запланировано в 0.3.1.  
- **Передача файлов:** Работает надежно, протестирована на разных типах файлов, отображается прогресс.  
- **Голосовые сообщения:** Запись и прослушивание работают без сбоев; возможно, улучшен формат (например, отправка более сжатого аудио, Opus/OGG вместо PCM, если потребуется).  
- **Шифрование:** Пользователь может включить, и при включенном режиме вся передача защищена. Документация для пользователя, что это обеспечивает (например, предупреждение, что нужно у всех включить).  
- **История:** Последние сообщения видны при запуске, история хранится.  
- **Упаковка клиента:** Клиентское приложение будет упаковано так, чтобы пользователь мог запускать без установки JDK вручную. Возможно, использовать jpackage или сборку в самодостаточный исполняемый файл (native image). Цель – **“без необходимости ставить Java”** – значит, в поставку включить JRE или сделать exe.  
- **Запуск сервера:** Сервер поставляется в виде исполняемого JAR (server.jar) или скрипта, который запускает его “одной командой”. Например, java -jar server.jar --port 12345. Документация должна описывать запуск.  
- **Документирование и тестирование:** К 1.0.0 написать краткое руководство (README) для пользователя. Написать юнит-тесты на основные компоненты (сеть, шифрование, команда parser и т.д., многое уже можно протестировать из-за отделения логики). Прогнать интеграционные тесты (клиент-сервер взаимодействие).

*Замечания:* Версия 1.0.0 должна включать все минимально необходимое: групповой чат с GUI, передачу файлов и аудио, систему событий, безопасность (шифрование, авторизация) и сохранение истории. Очень важно уделить время качеству кода – к этому моменту проект вырос, и нужно убедиться, что он не превратился в ком сложно поддерживаемого кода. Следует провести рефакторинг, устранить дублирование, проверить соответствие принципам *SOLID*. Возможно, привлечь сторонний код-ревью или прогнать статические анализаторы.

## После 1.0.0 — дальнейшие идеи

После достижения стабильной версии, можно рассматривать следующие направления развития:

* **1.1.0 — Мобильный клиент:** Разработка мобильного приложения (Android/iOS) для чата. Вероятно, понадобится сервер с WebSocket API или HTTP+long polling. Можно написать отдельный фронт на Flutter или Kotlin (Android). Для этого сервер может быть переработан на базе Spring Boot/WebSocket чтобы поддерживать стандартные протоколы.
* **Переход на Spring Boot:** Переписать сервер на Spring Boot, используя встроенный WebSocket STOMP или SockJS. Это облегчит интеграцию с веб и мобилками, а также добавит возможности масштабирования (можно развернуть на сервере, кластеризовать при необходимости). Минус – увеличение требований (нужен контейнер или деплой). Решение зависит от целей проекта.
* **Новые фичи:** Видео-звонки (более сложный уровень — потребует WebRTC, что за рамками данного проекта), боты/интеграции, улучшенный UI/UX (например, эмодзи, реакции на сообщения), и пр. Но эти функции уже вне изначального плана и должны рассматриваться отдельно, придерживаясь *YAGNI* – внедрять по мере необходимости и наличия ресурсов.

*Выполненный функционал отмечен выше как «выполнено» или «частично».* Каждая следующая версия логически наращивает предыдущую. Важно двигаться итеративно: реализовать и стабилизировать одну версию прежде, чем браться за следующую, что, судя по плану, и предполагается. Теперь перейдём к обзору архитектурных решений и качеству кода.

# Свод основных архитектурных решений проекта

* **Event-driven архитектура через EventBus:** Одно из ключевых решений – внедрение шины событий (паттерн *Observer* / Publisher-Subscriber). UI и сетевой код общаются не напрямую, а через публикацию и подписку на события. Это даёт слабую связанность: можно модифицировать или заменять UI или транспорт независимо. Такое решение упрощает добавление новых типов событий (например, FileEvent, VoiceEvent), следуя принципу *OCP* (расширяемость без изменения существующего кода). Решение уже показало свою эффективность в версии 0.3.x. Реализация SimpleEventBus потокобезопасна (использует ConcurrentHashMap и CopyOnWriteArrayList для подписчиков), что важно, ведь события могут приходить из сетевых потоков. **Замечание:** Стоит убедиться, что при завершении работы или отключении компонентов они отписываются от событий (во избежание утечек). Сейчас EventBus живёт пока работает приложение, поэтому критичного нет, но в будущем (например, при динамическом создании окон) это может понадобиться.
* **Разделение на модули (Maven multi-module):** Проект структурирован на три модуля: common (общие классы, например, Message и события), server (серверная часть) и client (клиентская часть). Такое решение соответствует принципу многослойной архитектуры: общие модельные классы вынесены, чтобы исключить дублирование (*DRY*). Клиент и сервер оба зависят от common, но не напрямую друг от друга. Это облегчает управление зависимостями – например, common можно собрать в отдельный .jar и использовать на обеих сторонах, гарантируя, что формат сообщений一致ен. **Замечание:** В текущей версии common довольно небольшая, но со временем будет расти (туда можно положить утилиты, модели, возможно, код для JSON-сериализации). Важно поддерживать обратную совместимость при изменении common (например, добавляя новые типы сообщений, не ломать существующие).
* **Паттерн «Команда» для пользовательского ввода:** В консольном клиенте и GUI реализована обработка команд вида /help, /exit, /clear. Используется библиотека replines-core, которая предоставляет ReplRunner и CommandRegistry. Это фактически реализация паттерна *Command*: каждая команда регистрируется с действием. Это решение позволяет легко расширять набор команд (например, добавить /history, /w <user> и т.д.) без большого переписывания логики парсинга – достаточно зарегистрировать новую команду и написать обработчик. **Замечание:** Сейчас команды exit, help, clear реализованы. Следует убедиться, что команды не конфликтуют с вводимым текстом (например, что обычное сообщение, начинающееся с /, не воспринимается как команда – в коде это учтено, т.к. проверяется if (text.startsWith("/"))). Возможно, в будущем стоит иметь команду для отправки файлов или других действий тоже через этот механизм, или, как минимум, следить, чтобы UI-элементы (кнопки) дублировали функциональность команд для удобства пользователя.
* **Потоковая модель и работа с сетью:** Сервер реализован на блокирующих сокетах: каждый новый клиент обслуживается в отдельном потоке (new Thread(new ClientHandler(socket)).start()). Клиент тоже использует отдельный поток для ожидания сообщений от сервера (см. ChatClientConnector.connect() открывает Thread("ChatClient-Listener")). Это *простая и понятная модель* многопоточности (соответствует KISS для сетевого взаимодействия). Она хорошо работает для небольшого числа пользователей. Решение отказаться от NIO/Selector в пользу простых потоков оправдано на данном этапе (реализация быстрее и проще). **Замечание:** Использование Concurrent коллекций на сервере для списка клиентов (ConcurrentHashMap.newKeySet) – верный шаг для потокобезопасности. Однако при рассылке сообщений *есть потенциальная проблема:* одновременная отправка разными потоками одному клиенту может привести к тому, что два ObjectOutputStream запишут в сокет вперемешку. Текущее решение с clients.removeIf(...) внутри broadcast не синхронизировано, и два потока могут параллельно итератор проходить. Это может вызвать непредсказуемое поведение. *Решение:* синхронизировать блок рассылки (например, оборачивать broadcast(msg) в synchronized или использовать очередь сообщений). Пока, видимо, таких коллизий не возникало (сообщения приходят последовательно), но при активах двух отправителей одновременно возможно. Это стоит иметь в виду и при необходимости доработать.
* **Форматирование и UI-вывод:** Решено было использовать Swing (JFrame, JTextArea и т.п.) для простоты (альтернатива могла быть JavaFX, но он потребовал бы дополнительные сложности с потоками JavaFX Application Thread). Swing выбран, вероятно, как более знакомый и легковесный вариант. Это оправдано *KISS*-соображениями: быстрее реализовать чат на Swing. В то же время, Swing позволяет достаточно гибко кастомизировать вывод. Решение использовать простой JTextArea для вывода чата – минимально достаточное. **Замечание:** Если требуется раскрашивать/форматировать текст (для системных сообщений или, скажем, выделения имен), JTextArea напрямую не поддерживает стили. Но можно использовать JTextPane или StyledDocument. Возможно, имеет смысл перейти на JTextPane для зоны чата, что позволит менять цвет/шрифт отдельных сообщений. Это затронет лишь UI, остальная архитектура не пострадает (благодаря изоляции через EventBus). Кроме того, **необходимо** учитывать, что доступ к Swing-компонентам должен происходить из EDT (Event Dispatch Thread). В коде правильно делается: вызовы обновления UI обёрнуты в SwingUtilities.invokeLater (например, при публикации MessageReceivedEvent, добавлении текста). Нужно продолжать следить за этим, иначе возможны проблемы синхронизации UI.
* **Использование Object Streams vs. JSON (текущее решение):** На данный момент обмен осуществляется сериализованными объектами Message через ObjectOutputStream. Это очень быстро позволило запустить чат (минимум кода — достаточно чтобы класс Message implements Serializable). Решение оправдано для стадии 0.3.x – *быстрое получение результата* (что соответствует принципу *YAGNI* — не реализовывать сложный протокол раньше времени). Однако, как планируется, это будет изменено на JSON (0.6.0). Сейчас важно, что данное решение наложило некоторые ограничения: клиент и сервер должны иметь совместимые версии классов (иначе десериализация сломается), и оно затрудняет подключение других типов клиентов. Но как временное решение — вполне приемлемо. **Замечание:** При переходе на JSON, стоит предусмотреть обратную совместимость или одновременно обновить все компоненты. Также надо учесть, что сериализация объектов хранит много метаинформации (имена классов и пр.), что может быть избыточно. JSON решит это.
* **Обработка ошибок и устойчивость:** В текущей реализации сделано минимально: при ошибке сети (например, Exception в клиентском потоке чтения) на клиенте публикуется UserDisconnectedEvent и на сервере удаляется клиент. Клиент при отправке в sendMessage ловит IOException и публикует системное сообщение об ошибке. Это решение позволяет не крашить приложение при потере связи. **Замечание:** Можно улучшить, например: при отключении сервера клиент мог бы показывать диалог "Connection lost", а не просто писать в чат "[SYSTEM] ...". Также логирование: сейчас System.out используется для debug (в консоли принты). Стоит перейти на logging framework (например, логировать на сервере подключения/ошибки). Это не было критично на начальных этапах, но к релизу 1.0.0 желательно иметь аккуратный лог.
* **Расширяемость и поддержка новых функций:** Благодаря введённой архитектуре (EventBus, модульность, разделение обязанностей), проект довольно готов к расширениям из roadmap. Добавление новой функциональности (файлы, аудио) будет происходить через добавление новых событий и обработчиков, не затрагивая уже рабочий код отправки текстовых сообщений. Это большая победа архитектурного решения, которое соответствует принципам *OCP* и *DIP* (мы зависим от абстракций событий, а конкретные реализации – новые классы – подключаются без изменений старого кода). Пример: для отправки файлов можно создать класс FileMessage extends Message или добавить поле в Message, но лучше, чтобы старый код умел игнорировать или обрабатывать новые типы. Можно использовать instanceof или type field – это расширение, а базовые механизмы останутся прежними.
* **Использование записей (Java record) и современных возможностей:** Заметим, что некоторые события реализованы как record (например, MessageReceivedEvent, MessageSendEvent). Это удобное решение – обеспечивает неизменяемость и компактный код. Также проект нацелился на Java 17 (судя по pom.xml source/target 17). Это хорошо, т.к. позволяет использовать новые API и синтаксические возможности. **Замечание:** Надо помнить, что Java 17 требует соответствующей JRE у пользователя; но если мы упакуем приложение с JRE, проблем не будет. Использование record – плюс для *KISS/DRY*: меньше шаблонного кода (геттеры, конструкторы).
* **Legacy-код и поддержка:** В проекте присутствует класс ChatClient\_legacy – старый способ без EventBus. Решение оставить его, видимо, чтобы сохранять пример или тест. Архитектурно, основной код уже ушёл от этой реализации. **Замечание:** По принципу *YAGNI/DRY*, возможно, этот класс уже не нужен и может быть удалён, чтобы не захламлять кодовую базу (если в нём нет особой ценности). Каждая оставленная неиспользуемая часть увеличивает поддержку. Если он служит для отладки, можно пометить как deprecated. В идеале, к релизу 1.0 весь legacy-код убрать либо выделить в отдельный модуль/пример.

# Анализ проекта: принципы KISS, DRY, YAGNI, SOLID и паттерны

Проект развивается с учётом ключевых инженерных принципов. Расмотрим их применение и области для улучшения:

* **KISS (Keep It Simple, Stupid — «Делай проще»):**  
  На текущий момент архитектура стремится к простоте. Использованы простые решения там, где это уместно: стандартные Java-сокеты и потоки вместо сложных фреймворков, Swing для GUI вместо более тяжелого UI-фреймворка, Java-сериализация на первых этапах вместо немедленной имплементации собственного протокола. Это позволило быстро получить работоспособный чат. Код довольно прозрачен: один класс отвечает за одну функцию (сервер, клиент, UI, ивент-бас).  
  **Где можно упростить/уточнить:** Есть пара мест, которые выглядят немного избыточно. Например, ApplicationContext содержит поле context самого же типа, дублируя себя (в коде ctx.setContext(ctx)). Возможно, это не нужно – можно всегда передавать сам объект, а внутри хранить ссылку на себя не имеет особого смысла. Также ChatUiCallback интерфейс определён, но фактически не используется – вероятно, остаток старой идеи. Такие вещи лучше удалять, чтобы не усложнять восприятие (соответствует KISS – убрать лишнее). В остальном, проект следует принципу простоты: каждая новая возможность сначала делается максимально прямолинейно (например, передача файла – сначала цельным куском, без преждевременного усложнения). Это хороший подход.
* **DRY (Don’t Repeat Yourself — «Не повторяйся»):**  
  Разделение на модуль common уже предотвращает повторение определений классов сообщений и событий между клиентом и сервером. EventBus один на всё приложение (в каждом клиенте). Также вынесение команд обработчика в CommandHandler позволяет использовать его и в консоли, и в GUI, не дублируя код команд. Видно стремление к устранению дублирования: например, и UI, и Console клиент используют общие события и формат сообщений.  
  **Что можно улучшить по DRY:** Есть небольшое дублирование между ChatClientConsole и ChatClientUI: оба создают ChatClientConnector, оба подписываются на сходные события (MessageReceivedEvent), и оба имеют свой цикл чтения ввода (консоль – Scanner, UI – TextField action). Хотя их природа разная, стоит вынести общую логику (например, подключение к серверу, создание EventBus, CommandHandler) в одно место. Возможно, можно реорганизовать так: ChatClientConsole и ChatClientUI наследуются от общего абстрактного класса или используют один помощник. Но это не критично – кода там не слишком много, и он разный (UI vs консоль).  
  Другая деталь: сервер при каждом сообщении повторяет логику рассылки (цикл по clients). Если в будущем появятся разные комнаты или индивидуальная отправка, можно инкапсулировать эту логику рассылки в метод, чтобы не повторять. Пока же дублирования особо нет.  
  Также, если будем реализовывать и FileSendEvent, и VoiceSendEvent, вероятно, код их передачи будет очень похож. Нужно будет постараться абстрагировать общий код передачи бинарных данных (например, метод sendBytes(bytes, type) или объединить эти события). Это предотвратит дублирование в сетевом слое.
* **YAGNI (You Aren’t Gonna Need It — «Вам это не понадобится»):**  
  Судя по плану, разработка идёт итерационно, добавляя функции по необходимости. На момент 0.3.x не реализовывались заранее вещи из будущих версий. Это соответствует YAGNI – не писать код “впрок”. Кодовая база относительно компактна и не содержит явно ненужных на данном этапе функций. Например, авторизация не попытались встроить в 0.3.x, её отложили до 0.8.x, когда действительно будет нужна.  
  **Где наблюдаются отклонения:** Можно отметить, что ввели UserConnectedEvent/UserDisconnectedEvent и интерфейс EventBus.subscribe позволяет подписываться на них, но UI пока их не использует. Это небольшой случай опережения: подготовили события, хотя функционал (список пользователей) ещё не реализован. Однако их наличие не мешает, а закладывает основу – тут можно поспорить с YAGNI, но в рамках проектирования событий можно считать допустимым шагом на опережение. Другой пример – ChatUiCallback интерфейс, как уже сказано, выглядит как “незадействованный запасной путь”. Если он более не понадобится (а с EventBus он и не нужен), стоит удалить.  
  В остальном YAGNI соблюдается: например, *не* стали сразу делать сложную многокомнатную структуру чата или не внедрили сразу базу данных – это всё запланировано на будущее, по мере роста требований. Это правильный подход, позволяющий не обременять код лишней сложностью преждевременно.
* **SOLID:**  
  Рассмотрим, насколько код и проект соответствуют пяти принципам SOLID:
* **S (Single Responsibility Principle, принцип единственной ответственности):** В целом, разделение обязанностей выполнено хорошо. Классы имеют чёткие роли: ChatClientConnector – связь с сервером, ChatClientUI – интерфейс, ChatServer – основной сервер, SimpleEventBus – шина событий, CommandHandler – обработчик команд. Это делает код понятным и модульным. Могут быть места, где класс делает чуть больше, чем одна ответственность (например, ChatClientConnector ивенты публикует и сетевой ввод обрабатывает – но это всё относится к “сетевому слою клиента”, так что нормально). UI класс чуть смешивает логику UI и регистрацию команд/EventBus – но регистрация событий для UI несложна и ожидаема.
* **O (Open/Closed Principle, принцип открытости/закрытости):** Проект растёт через добавление новых классов и событий, в основном не требуя модификации существующих. Например, чтобы добавить новый тип сообщения, мы добавляем новый Event и обрабатываем его, не меняя код EventBus или базового Message (если сделаем расширение Message, можно тоже не менять существующий код, используя полиморфизм). Некоторые части, правда, придётся модифицировать при новых фичах – например, переход на JSON затронет ChatClientConnector и ChatServer (но это смена фундаментальной технологии, допустимо). В основном же архитектура расширяема: новые команды, новые типы Event – всё добавляется. Это показывает, что OCP в значительной мере соблюдается.
* **L (Liskov Substitution Principle, принцип подстановки Барбары Лисков):** В проекте почти нет наследования (кроме, возможно, JFrame для ChatClientUI, что стандартно). Использование интерфейсов (EventBus) и реализаций (SimpleEventBus) – там LSP выполняется, т.к. SimpleEventBus полностью соответствует контракту EventBus. Если будут наследоваться какие-то классы (например, расширенный Message), важно, чтобы их можно было везде использовать вместо базовых. Пока проблем нет, но надо иметь в виду при расширении классов: не вводить в подклассах поведение, которое нарушит ожидания. Например, если сделать EncryptedMessage extends Message, он должен всё равно работать там, где ждут Message.
* **I (Interface Segregation Principle, принцип разделения интерфейсов):** Интерфейсов немного, и они узкие по назначению. EventBus разбит на два метода (subscribe, publish) — трудно разбить его ещё, да и не нужно. Интерфейс ChatUiCallback был очень узким (два метода), но по сути он не используется. В общем, нет “толстых” интерфейсов, которые заставляли бы реализовывать лишнее. Все интерфейсы компактны, ISP не нарушается.
* **D (Dependency Inversion Principle, принцип инверсии зависимостей):** DIP предполагает, что высокоуровневые модули не зависят от низкоуровневых реализаций, а и те, и другие зависят от абстракций. В нашем случае EventBus – это абстракция, на которую зависят и UI, и сетевой код, вместо того чтобы UI зависел напрямую от конкретной реализации сети. Это большой плюс к DIP. Другой пример – CommandHandler формирует CommandRegistry и предоставляет ReplRunner – UI и Console зависят от интерфейса ReplRunner (через его методы), а не от конкретной реализации парсинга. На сервере, однако, пока нет таких слоёв абстракции – ChatServer напрямую работает с сокетами. В будущем, при переходе на WebSocket или внедрении другой транспортной системы, стоит тоже выделить интерфейс, например, ClientConnection (с методами send, close), и делать разные реализации для разных протоколов. В целом, DIP соблюдается в самых критичных местах (UI vs Network), что уже хорошо.
* **Применённые шаблоны проектирования и возможные паттерны для улучшения:**
* **Observer/Publisher-Subscriber:** Как отмечалось, EventBus – собственная реализация этого паттерна. Это центральный шаблон, держащий архитектуру. Он применён уместно и сделан достаточно гибко.
* **Command:** Использован для реализации пользовательских команд (через CommandRegistry). Шаблон Command позволил отделить команды от основной логики ввода.
* **Singleton:** Явно не используется, и это хорошо – глобальных синглтонов нет, EventBus создаётся явно и передаётся, что облегчает тестирование (можно подменить при необходимости). Если бы был какой-то глобальный объект (например, логгер), можно было бы применить Singleton, но здесь нет необходимости.
* **Factory:** Пока не используется. Возможно, потребуется при создании сообщений разных типов из JSON (например, фабрика, которая по полю type создает нужный объект Message или Event). Это может появиться в 0.6.x при переходе на новый протокол. Фабричный метод или абстрактная фабрика может пригодиться, чтобы инкапсулировать логику построения объектов (например, MessageFactory.createFromJson(String json)).
* **Strategy:** Может быть применён для выбора протокола (TCP/UDP) или алгоритма шифрования. Например, объект MessageTransportStrategy с методами send(Message)/receive() – и две реализации: ObjectStreamStrategy и JsonSocketStrategy. Тогда можно переключать протокол, не меняя остальной код. Пока это не сделано, но при рефакторинге 0.6.0 стоит обдумать. Также Strategy упоминался для голосового транспорта (TCP vs UDP).
* **MVC (Model-View-Controller) / MVP / MVVM:** В контексте UI можно отметить, что элементы этих паттернов частично есть: модель чата (состояние, пользователи) и представление (Swing GUI) связаны через EventBus (что можно считать контроллером событий). Однако нет чёткого выделения слоя модели. Можно улучшить, например, введя класс ChatModel (состояние: список сообщений, список пользователей) и связывать UI с моделью через события (UI обновляется когда модель меняется). Сейчас UI сам хранит, например, список сообщений просто в текстовом поле. Для текущего объёма это нормально, но при усложнении (история, группы) стоит рассмотреть паттерн MVC/MVVM для лучшей организации.
* **Adapter:** Если планируется интеграция с другим API (например, использование Spring WebSocket), можно написать адаптер, чтобы его сообщения конвертировались в наш формат событий. Пока не актуально, но знать о такой возможности полезно.
* **Decorator:** Может пригодиться для функциональностей вроде шифрования – вместо раздувания класса сообщений, можно декорировать поток или сообщения. Например, сделать EncryptedOutputStream обёрткой над реальным OutputStream, чтобы на лету шифровать данные. Либо декорировать объект Message перед отправкой. Но это усложнит понимание, возможно проще явно вызывать шифрование.
* **Template Method:** В серверном коде, если появятся разные способы обработки (скажем, BaseServer с шаблонным методом handleMessage, реализуемым в наследниках для разных протоколов). Пока с этим можно не заморачиваться.

**Итоговое впечатление:** Проект развивается поступательно и уже демонстрирует неплохое соответствие базовым принципам проектирования. Благодаря событиям (основа архитектуры) расширение функциональности не приводит к лавинообразным изменениям кода – это хороший признак. Принципы *KISS* и *YAGNI* помогли сосредоточиться на нужных вещах и получить рабочий продукт на ранней стадии, избегая over-engineering. В дальнейшем, новые возможности следует добавлять аккуратно, следя за *DRY* (чтобы не копировать код) и применяя нужные паттерны по мере необходимости, но не ради паттернов, а ради упрощения поддержки. Например, если код обработки файлов/аудио станет повторяться, вынести его в общий класс; если UI станет сложно управлять, рассмотреть MVC; если добавится много видов сообщений, возможно, использовать фабрику и полиморфизм.

Проект ориентирован на дальнейшее развитие, и заложенные архитектурные решения (event bus, модульность, команды) создают прочную основу. В то же время, всегда следует быть готовым к рефакторингу: архитектура – не догма. Если какой-то подход перестанет справляться (например, производительности потоков хватать не будет или EventBus станет перегружен событиями), нужно будет улучшать или менять решение. Судя по плану, команда к этому готова, так как на каждом этапе оцениваются новые технологии (JSON, WebSocket, мобильные клиенты). Главное – сохранять баланс между внедрением новых возможностей и поддержанием кода простым и чистым. Это позволит успешно достичь версии 1.0.0 и далее развивать проект.