Concord - Software Design Document

Authors:
Yousef Noufal
Gunnar Johansson
Simon Johansson
Jonathan Alm
Alexander Lisborg

Intro to application

The system is a client & server chat application where multiple clients can:

- Join, create, and leave chat channels.
- The server manages the chat channels and handles client connections.
- The client application provides a user interface for interacting with the chat system.

Vision

Our vision is to create a robust and scalable chat application that facilitates secure and efficient communication. The application aims to provide an intuitive user experience, while ensuring security, performance, and adaptability for future development. Ideas for developing the application further is:

1. Sending Files
2. add voice- & maybe video-channels

Also a long term goal is to give the user a full customizable interface which we know many programmers want.

Summary of how the system works

Our program consists of Server, Client & UI. Firstly we establish a connection from the client to the server. Then when the user interacts with our interface for ex: presses buttons or sends a message. The UIController calls on our Client methods which in return passes it to the server through the socket with our Message class.

If the user for example wants to join a channel, our UI calls on Client method which sends a JoinChannelCommand class to the server containing channel name and password, server then receives these messages in our ClientHandler, where we pattern match on the class type and handle it. When the ClientHandler has completed/validated the request it sends back a response message to the client or an error message containing a String of what went wrong which we print to the user for ex “Invalid Password” if trying to join a channel with incorrect password.

Design

- Scalability: The server should handle multiple client connections simultaneously.
- Security: Messages are encrypted using AES encryption.
- Usability: The client UI should be intuitive and responsive.

System Overview

The system follows a client*server architecture. The main components are:

- Server: Manages chat channels and client connections and chat history.
- Client: Provides a user interface for users to interact with the chat system.
- Communication: Uses sockets for communication between the client and server.
- Messages: classes sent between Client & Server for easy communication.

Important Classes for functionality

Class: Server

Responsibilities:

- Listen for incoming client connections.
- Manage chat channels.
- Handle client requests.

Significant methods:

- startListening(): Starts the server to listen for client connections.
- stop(): Stops the server.
- createChannel(String channelName, String password): Creates a new chat channel.
- getChannel(String channelName): Retrieves a chat channel by name.

Important Attributes:

- Map<String, ChatChannel>: map of active chat channels.
- boolean isRunning: boolean for if server is active.

Class: ClientHandler

Responsibilities:

- Handle communication with a single client.
- Process client messages.

Significant methods:

- run(): Listens for incoming messages from the client.
- joinChannel(String channelName, String password): Joins the client to a chat channel.
- leaveChannel(String channelName): Removes the client from a chat channel.
- sendMessage(Message message): Sends a message to the client.

Class: Client

Responsibilities:

- Facade for managing client side operations.

Significant methods:

- createChannel(String channelName, String password): Sends a request to create a new channel.
- joinChannel(String channelName, String password): Sends a request to join a channel.
- leaveChannel(): Sends a request to leave the current channel.
- sendMessage(String message): Sends a message to the current channel.

Class: ClientCommunicationManager

Responsibilities:

- Handle communication between the client and server.

Significant methods:

- run(): Continuously reads messages from the server.
- sendMessageToServer(ServerMessage message): Sends a message to the server.

Class: UIController

Responsibilities:

- Handle user interactions.
- Update the UI based on server responses.

Significant methods:

- update(UIMessage message): Updates the UI based on the received message.
- createChannel(): Handles the creation of a new channel.
- joinChannel(): Handles joining an existing channel.
- showChatArea(): Displays the chat area in the UI.

Class: MessageVisitorUI

Responsibilities:

- Process UI messages.

Significant methods:

- handle(DisplayError e): Handles error messages.
- handle(DisplayMessage m): Handles display messages.
- handle(UpdateChannels u): Handles channel updates.

Class: EncryptionLayer

Responsibilities:

- Encrypt and decrypt messages using AES.

Significant methods:

- generateKey(): Generates a new AES key.
- encrypt(Object encryptedData, SecretKey key): Encrypts data.
- decrypt(String encryptedData, SecretKey key): Decrypts data.

Class: StandardView

Responsibilities:

- Display the main user interface
- Handle user interactions

Significant methods:

- startArea(): initializes the area of the view.
- showChatArea(): Displays the chat area.
- appendChatText(String txt): appends text to the chat area.
- add*ButtonListener(ActionsListener listener): Adds buttons for our user interface like leave/join/create channel.

Design Patterns Implemented:

- For our messages we've implemented a **visitor pattern** in order to pattern match on different types of messages. This way we can easily handle the different types of messages and separate the server & client messages. Also easily scalable when implementing new features.
- For the View module we use **Decorator & factory pattern**, the purpose of the decorator pattern is to add functionality to objects dynamically (ViewDecorator & its subclasses wrap an IView object to add certain functionality). As well as **observer pattern** between client and our controller
- We use an **observer pattern** to handle state changes in the Client * UI. This way whenever we change states for ex leaves a channel, We notify our observers (UI) that we need to update our channel list. We also have a **visitor pattern** setup in the UI in order to pattern match on the state changes and handle those accordingly.

- We use a **facade pattern** in the client module, Client is a facade for the client*side operations, hiding the complexities of the ClientCommunicationManager & ClientChannelRecord, i.e provides a simplified interface for the client*side operations.
- We also implement an **abstract factory** in order to show the message objects correctly with the needed formatting.
- We also have **Singleton** to create a single instance of Server Class.

Testing/Testability

Is a critical aspect of our development process, ensuring that the system functions as expected/correctly without faults & check that it meets specific requirements. Our approach to achieve a good testing of the program is:

1. **Modular design**: The software is divided into independent modules, which makes it easier to test our different subsystems isolated and check for specific test cases and behavior of the subsystem.
2. **Mocking**: Dependencies are mocked to simulate different scenarios and control the test environment.
3. **Logging**: Logging information on what's going on in the program (serverside) which makes debugging/finding issues easier for when unexpected crashes happen.
4. **Clear Documentation**: Thorough documentation of the classes functionality, interfaces and dependencies to help with understanding and testing the application.

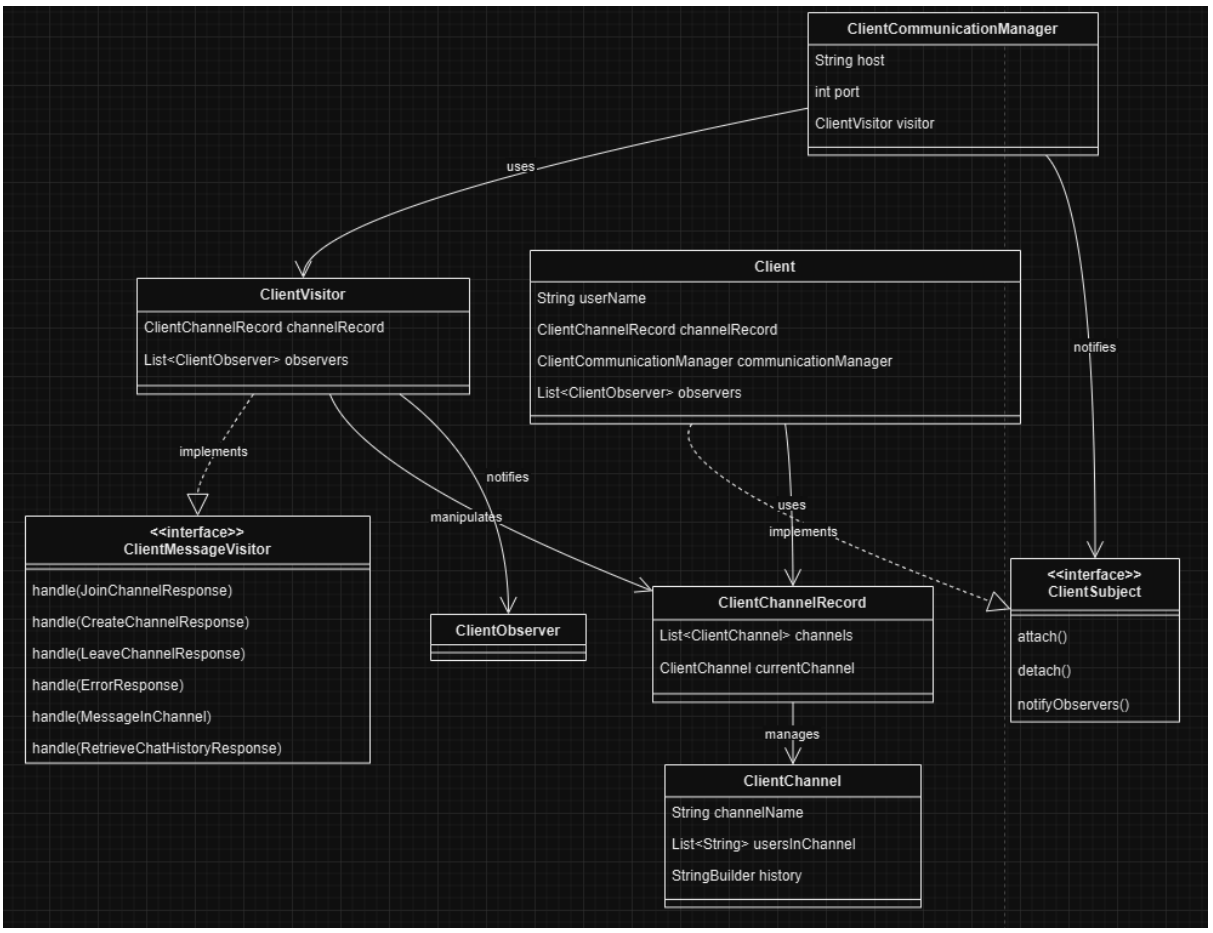
By following these steps we hope to achieve high test coverage, finding bugs/incorrect behaviors earlier and overall a more maintainable codebase.

UML Diagram

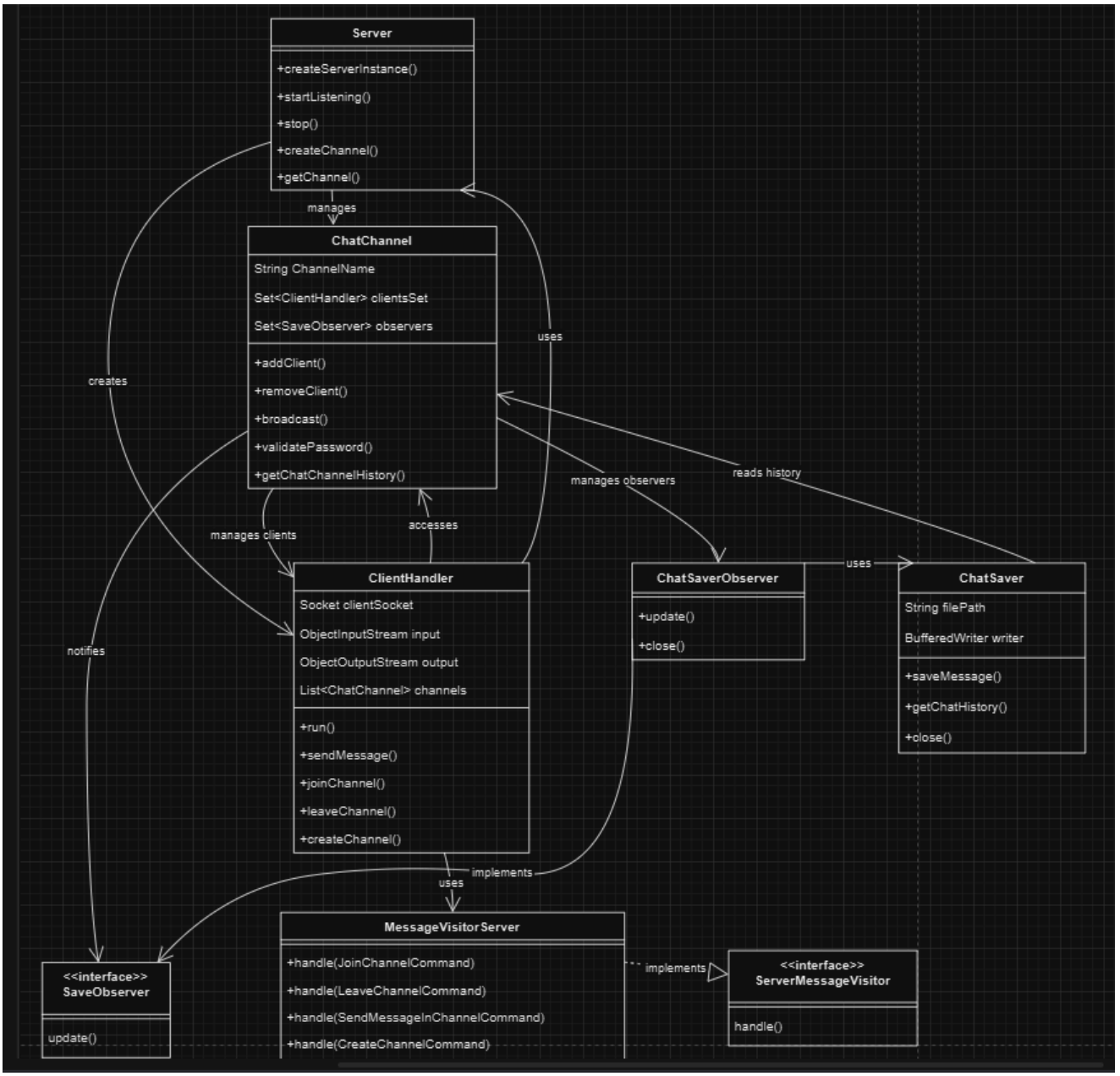
For Displaying our UML diagram we´ve decided to keep it somewhat simplified and then we have the extensive UML diagram below with the link to it.

Our Project consists of 3 main packages - Model, View & Controller.
Our model Consists of 3 Sub packages - Client, Server & Messages.

Client:

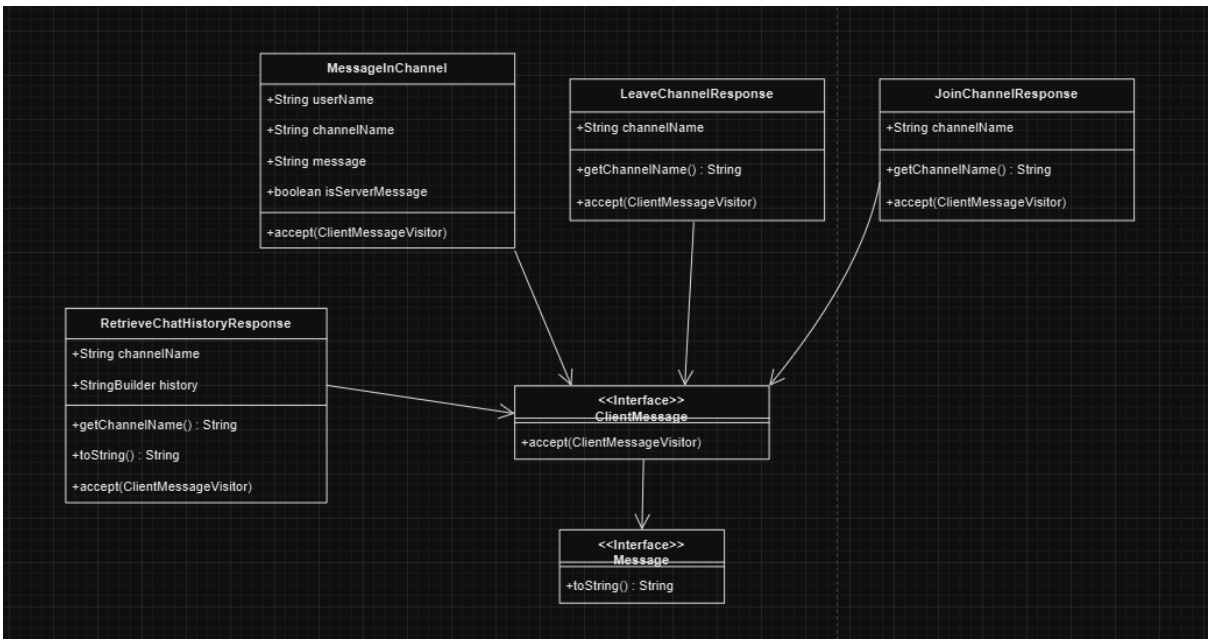

ClientCommunicationManager
String host
int port
ClientVisitor visitor
uses
notifies
ClientVisitor
ClientChannelRecord channelRecord
List<ClientObserver> observers
Client
String userName
ClientChannelRecord channelRecord
ClientCommunicationManager communicationManager
List<ClientObserver> observers
implements
notifies
manipulates
uses
implements
<<interface>>
ClientMessageVisitor
handle(JoinChannelResponse)
handle(CreateChannelResponse)
handle(LeaveChannelResponse)
handle(ErrorResponse)
handle(MessageInChannel)
handle(RetrieveChatHistoryResponse)
ClientObserver
ClientChannelRecord
List<ClientChannel> channels
ClientChannel currentChannel
<<interface>>
ClientSubject
attach()
detach()
notifyObservers()
manages
ClientChannel
String channelName
List<String> usersInChannel
StringBuilder history

Server:

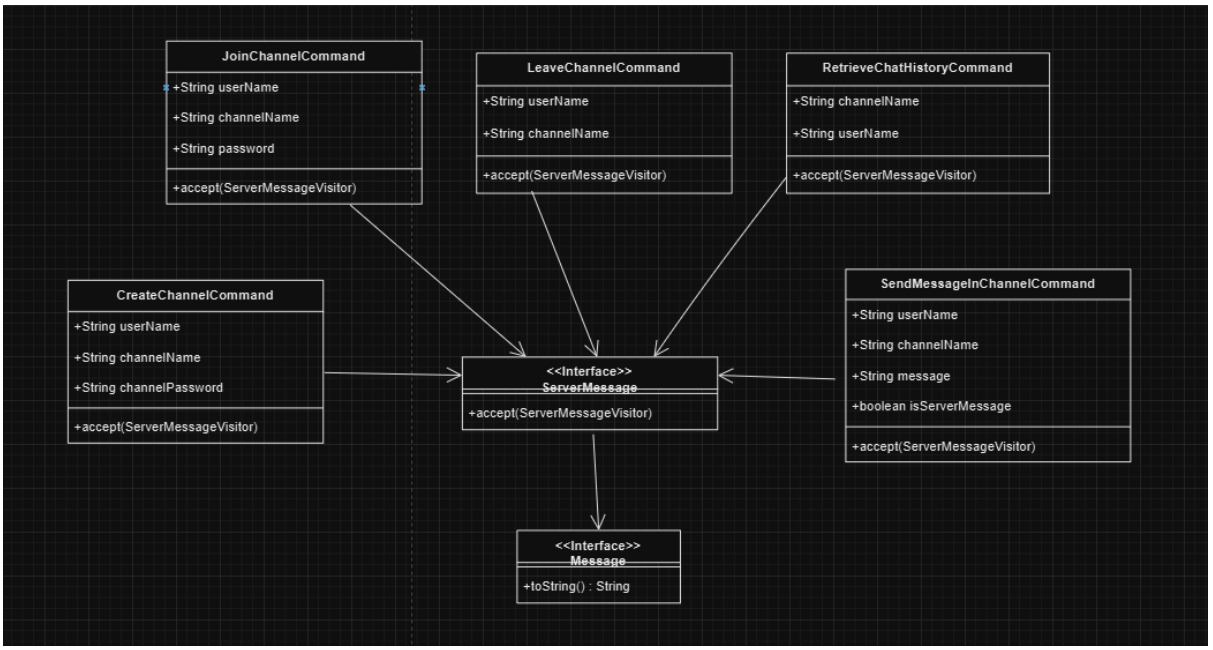


Messages:

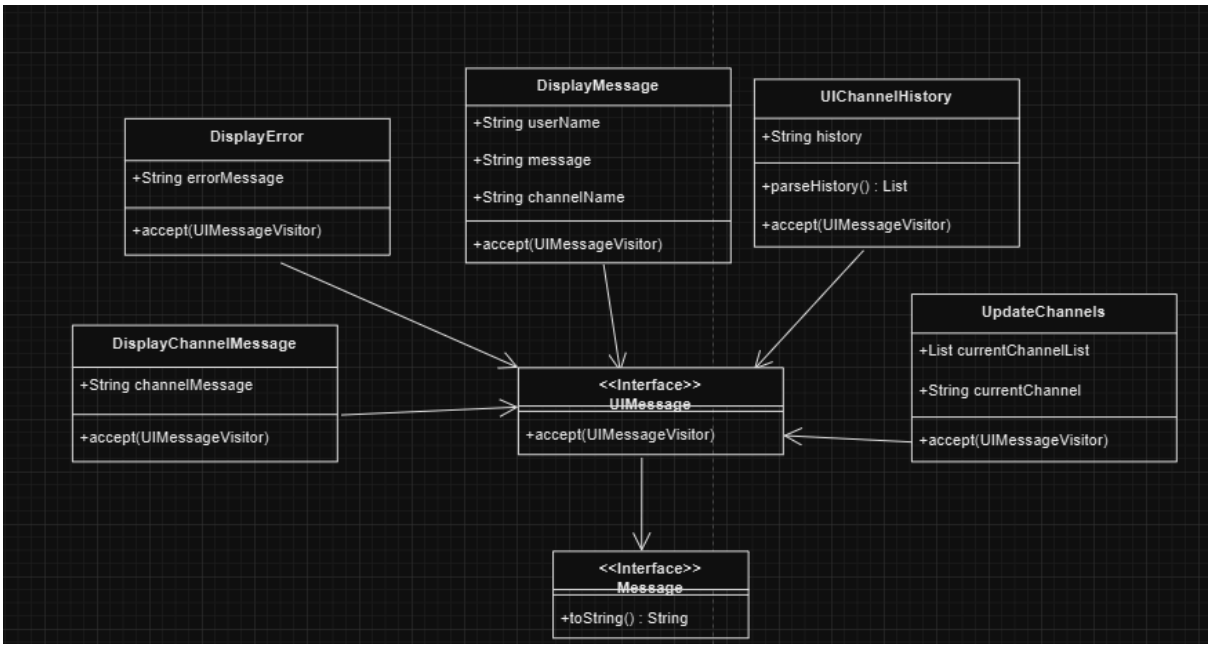
ClientMessages:



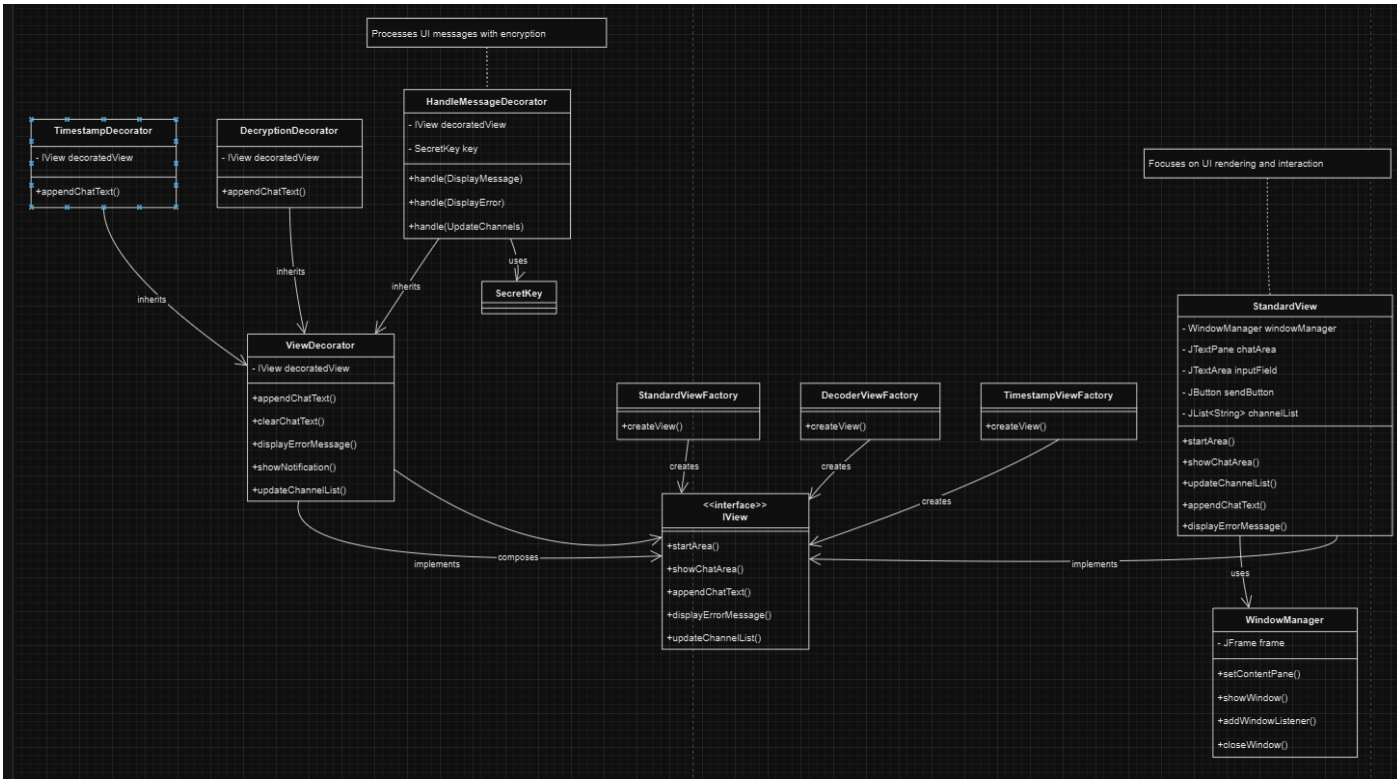
ServerMessage:



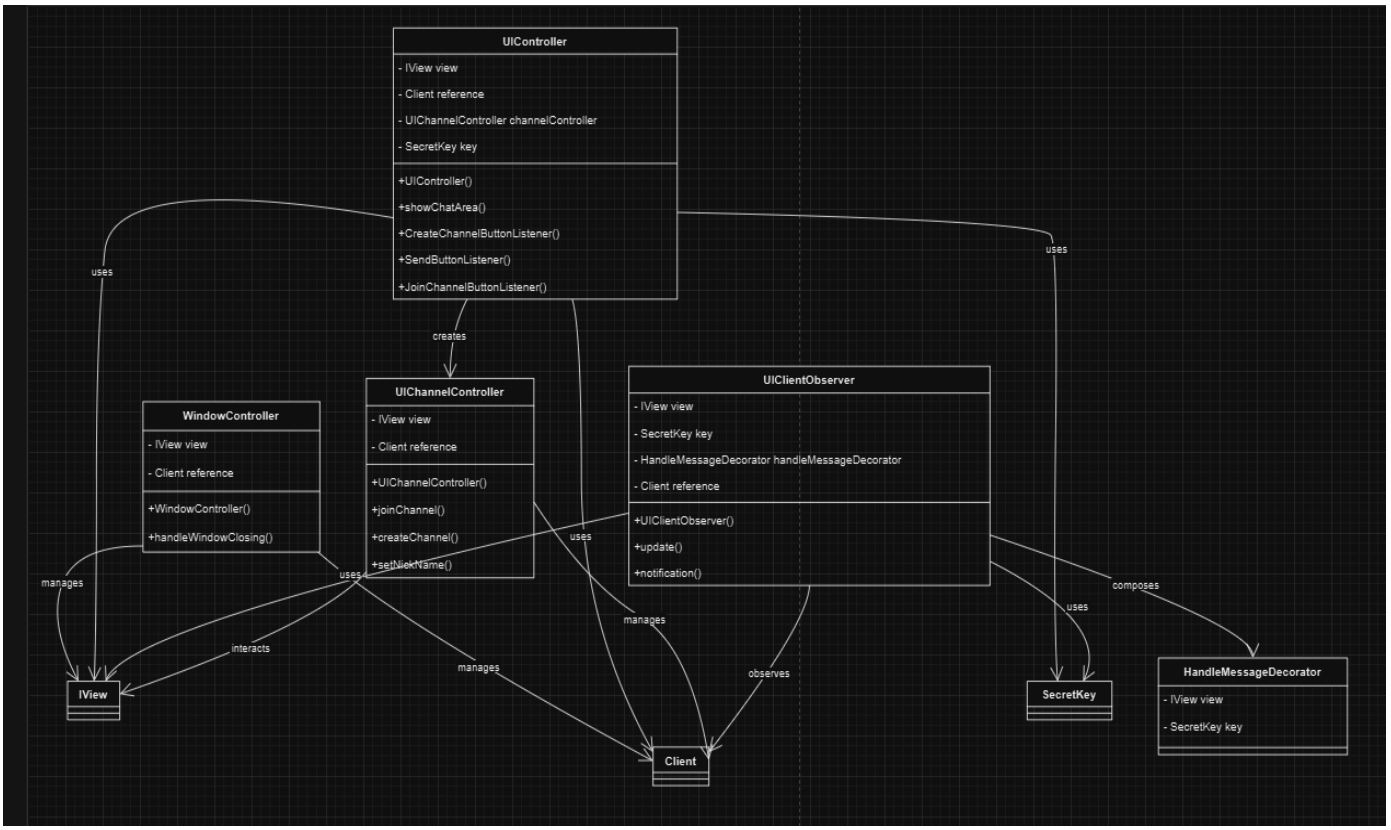
UIMessage:



View:

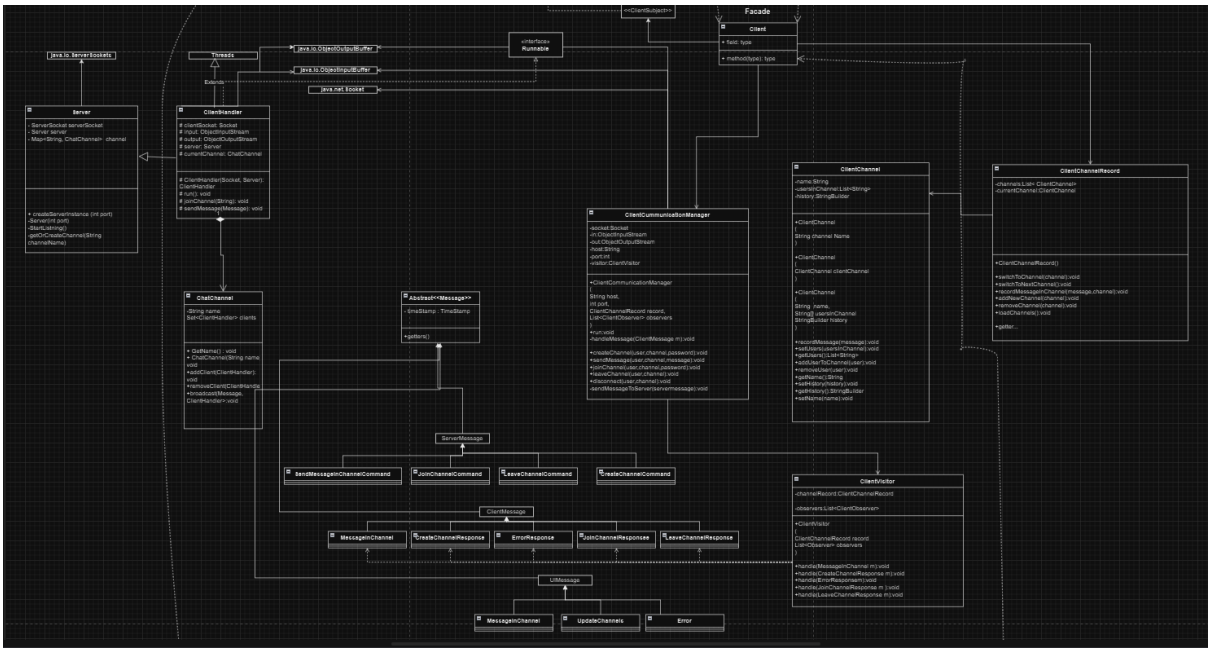


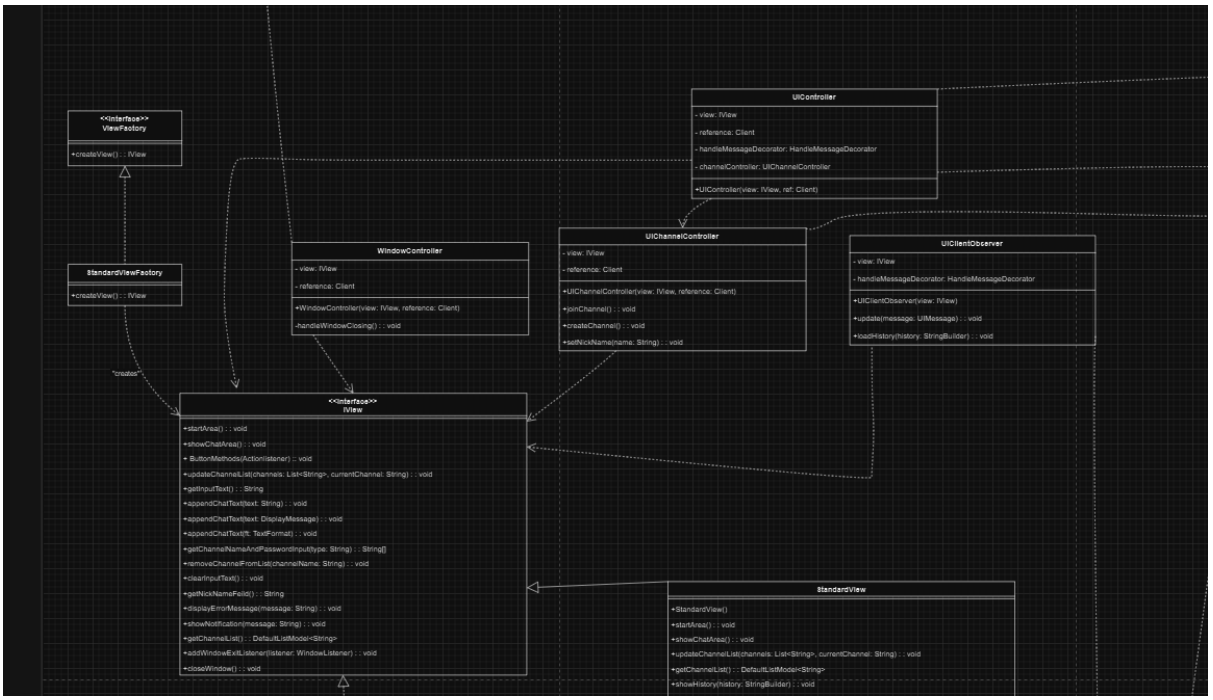
Controller:



Full-Scale UML Diagram:
https://drive.google.com/file/d/1JFjcGW4i9rPGsPwExs3kspswlE4KX0_4/view?usp=sharing if pictures below aren't clear enough.

Full scale UML





Software Design Document