I. Project goal and functionality

The goal of the project is to create a client-server chat application with support for simultaneous connections using threads and an administrator panel. Clients connect via a graphical user interface which allows them to set their username and send messages to the chat. The administrator uses the server-side console and can ban or unban clients from the chat. The server broadcasts received messages to all connected clients. Chat functionality may be further extended by adding timestamps to the messages or filtering certain words. The project is written in Java 8 with no external libraries. Swing is used for GUI components.

II. Class description

ChatServer	Accepts clients, starts AdminConsole
ClientHandler	Runnable, handles client threads
ClientMain	Starts client GUI
ClientGUI	Swing UI for users
AdminConsole	Admin console running in server
BanManager	Singleton for storing banned users
Message	Data class for chat messages
MessageAdapter	Adapter for message - raw string
MessageDecorator	Decorator base + concrete decorators
Protocol	Message protocol constant definitions

Table 1 Class names and descriptions

III. Design patterns used

1. Singleton pattern

General usage: Ensures that only one instance of a class exists in the application. The instance is usually globally accessible.

Use case in project: **BanManager** stores all banned usernames. Server components (client handlers, admin console) all access the same singleton instance for the purpose of guaranteeing consistent ban state across threads.

Fig. 1 Class diagram of the singleton pattern

2. Adapter pattern

General usage: Adapters convert one interface to another so that components with incompatible formats can work together.

Use case in project: Network communication uses raw protocol strings, while the application uses structured **Message** objects. **MessageAdapter** translates between them: - **fromRaw()** parses a network string. **toRaw()** serializes a Message for sending. This allows protocol changes without modifying business logic.

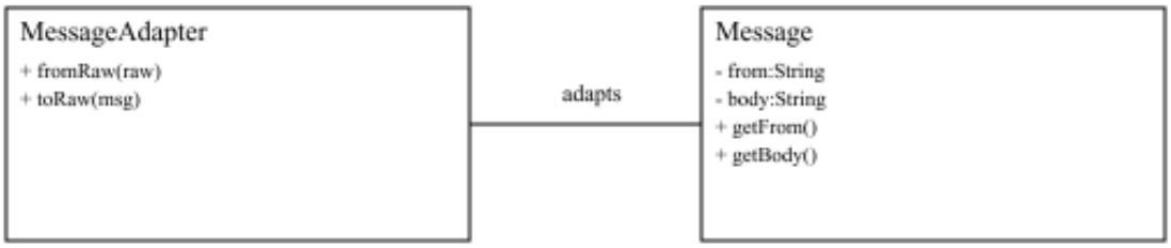

Fig. 2 Class diagram of the adapter pattern

3. Decorator pattern

General usage: Allows behavior to be added dynamically to an object by “wrapping” it inside one or more decorator objects. Each decorator keeps the original interface.

Use case in project: Incoming messages are wrapped in decorators: - **CensorDecorator** removes forbidden words. **TimestampDecorator** prepends a timestamp. This enables message transformations without modifying the **Message** class.

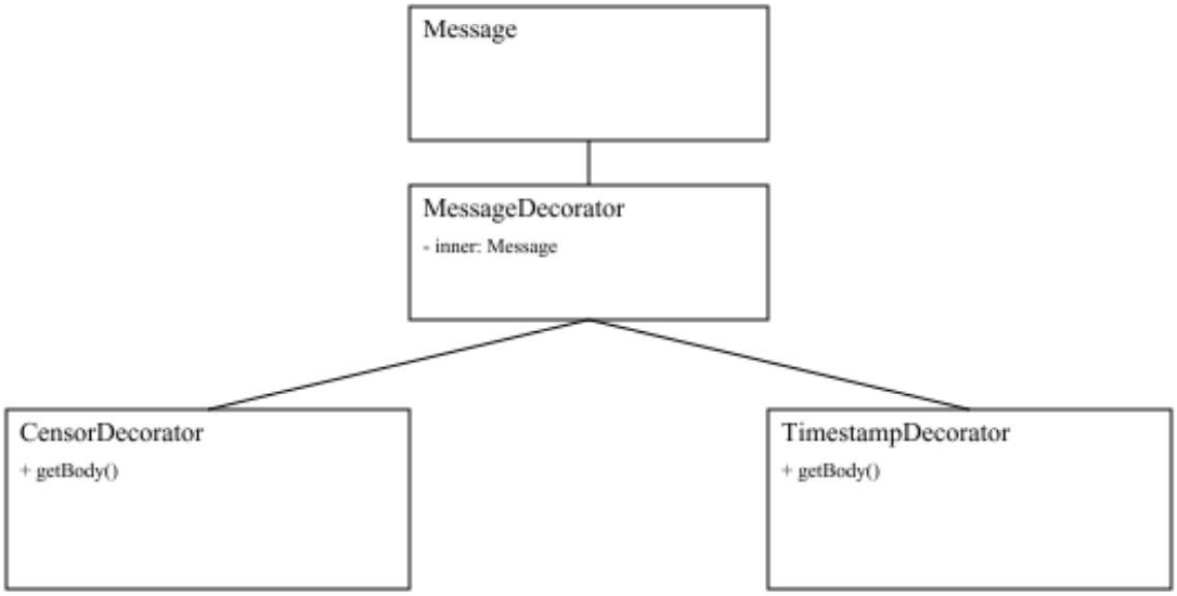

Fig. 3 Class diagram of the decorator pattern

IV. Results

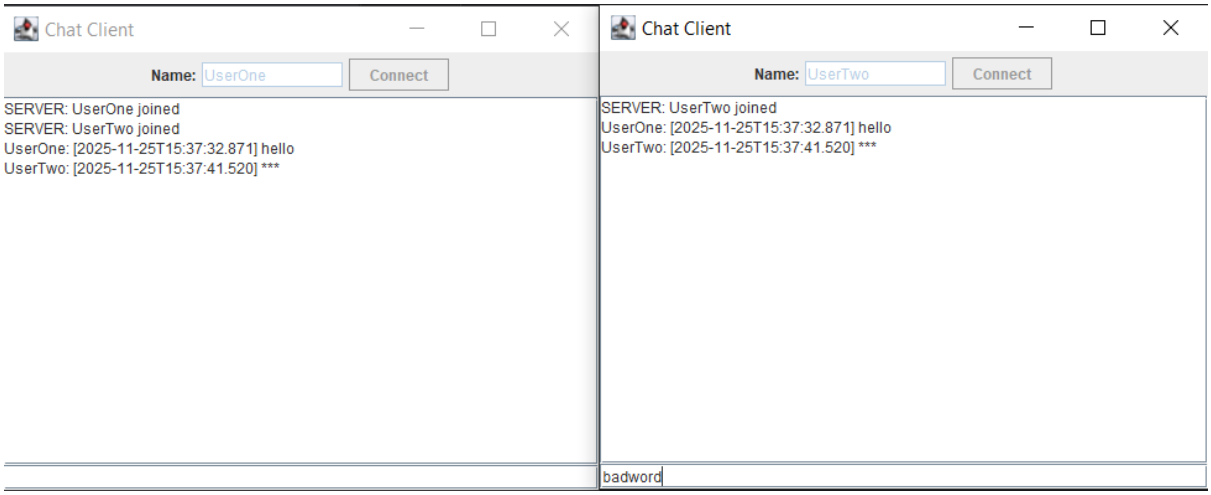

Fig. 4 Demonstration of multi-client chat with censorship of words and timestamps via decorator

```
Server listening on port: 1151
Admin console running. Type 'help' for commands.
admin> ban UserTwo
Banned: UserTwo
admin> listbans
[usertwo]
admin> quit
Shutting down server (admin requested).
```

Fig. 5 Demonstration of the admin console