Programowanie obiektowe Java ***PROJEKT***	
Politechnika Świętokrzyska	
Temat:	System do zarządzania zespołem
Autor:	Jakub Francuz
Grupa:	2ID12B

1. Cel projektu:

Celem projektu było stworzenie aplikacji graficznej do komunikacji między członkami zespołu ‘Team Manager’ z wykorzystaniem klient-serwer (localhost) oraz biblioteki graficznej Swing. Całość została napisana w języku Java wraz z praktykami programowania obiektowego.

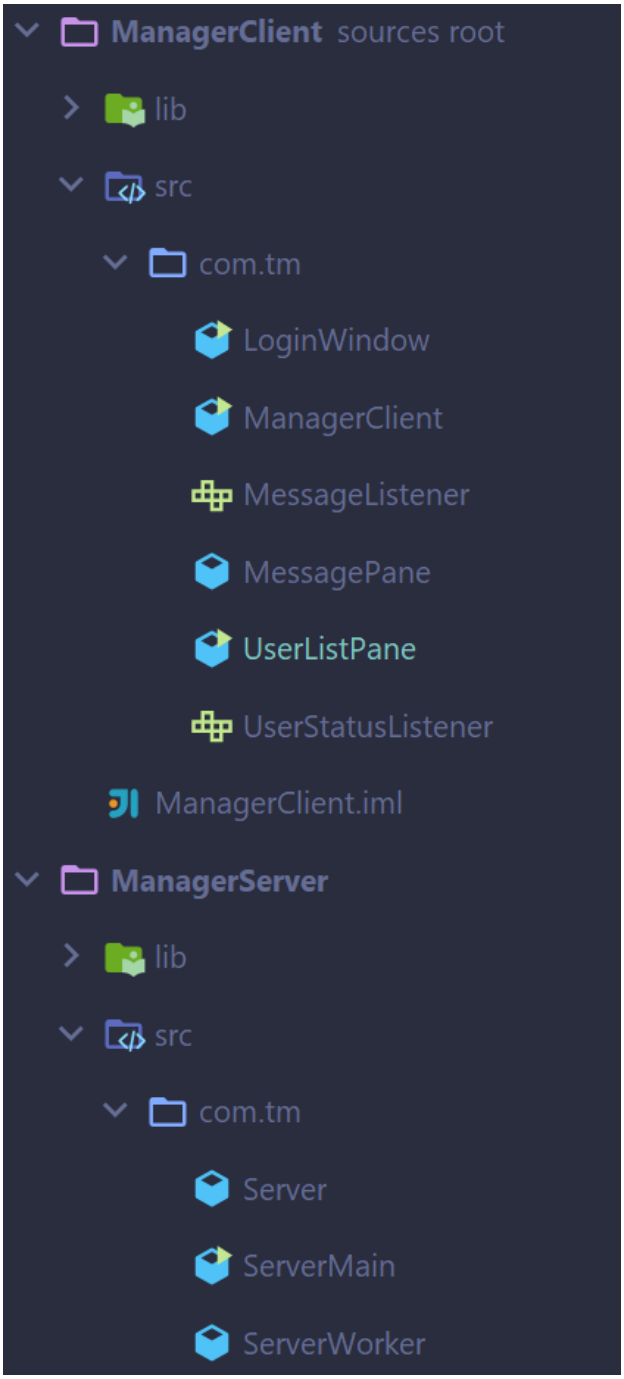
Program przygotowany jest aby działać zarówno jako CLI (command line interface) jak i GUI (graphic user interface). Powodem jest paradygmat zastosowany przy projektowaniu aplikacji – ‘dziel i zwyciężaj’. Najpierw okodowane zostały podstawowe funkcjonalności i prototypowane za pomocą linii poleceń. CLI zostało dla możliwości dalszego rozwijania aplikacji, a to po pierwsze pomogłoby w testowaniu kolejnych funkcjonalności, a także umożliwiło implementację tych bardziej złożonych (z natury CLI zapewnia dużo większą swobodę i efektywność interakcji z programem).

Postępy projektowe miały być dokumentowane za pomocą commitów na platformie w chmurze wspierającej system rozproszonej kontroli wersji GIT. Postawiłem tutaj na bardziej popularny serwis Github. Treści commitów (jak i komentarzy w kodzie) są w języku angielskim. Umożliwia to bardziej elastyczne zaznajomienie z terminami z IT, gdzie próżno szukać polskich semantycznych odpowiedników. Link do repozytorium to:

https://github.com/jqbFrnzs/javaPJ

Jako środowisko programistyczne obrałem Intellij IDEA, które ma na rynku najbardziej przychylne opinie co do intuicyjności przy pracy nad projektami w języku JAVA.

2. Struktura projektu:



Aplikacja podzielona jest na 2 moduły: *ManagerClient* i *ManagerServer*.

Oba moduły należą do paczek com.tm (tm – TeamManager)

Jak nazwa wskazuje, jeden zawiera komponenty dla strony klienta, drugi dla serwera aplikacji.

ManagerServer:

Klasa *ServerMain* – główny 'entry point' dla części serwerowej aplikacji

Klasa *ServerWorker* – implementacje funkcjonalności i logiki dla serwera

Klasa *Server* – umożliwia śledzenie i zarządzanie połączeniami z serwerem

Klasa *Constants* – zawiera stałą: numer portu, z której korzystają 3 klasy

ManagerClient:

Klasa *ManagerClient* – podstawowa klasa wejściowa dla części klienta

Klasa *LoginWindow* – zajmuje się GUI okna logowania

Klasa *UserListPane* – przedstawia graficznie listę użytkowników

Klasa *MessagePane* – okno do pisania wiadomości i komunikacji end-to-end

Interfejs *MessageListener* – metoda abstrakcyjna dla MessagePane

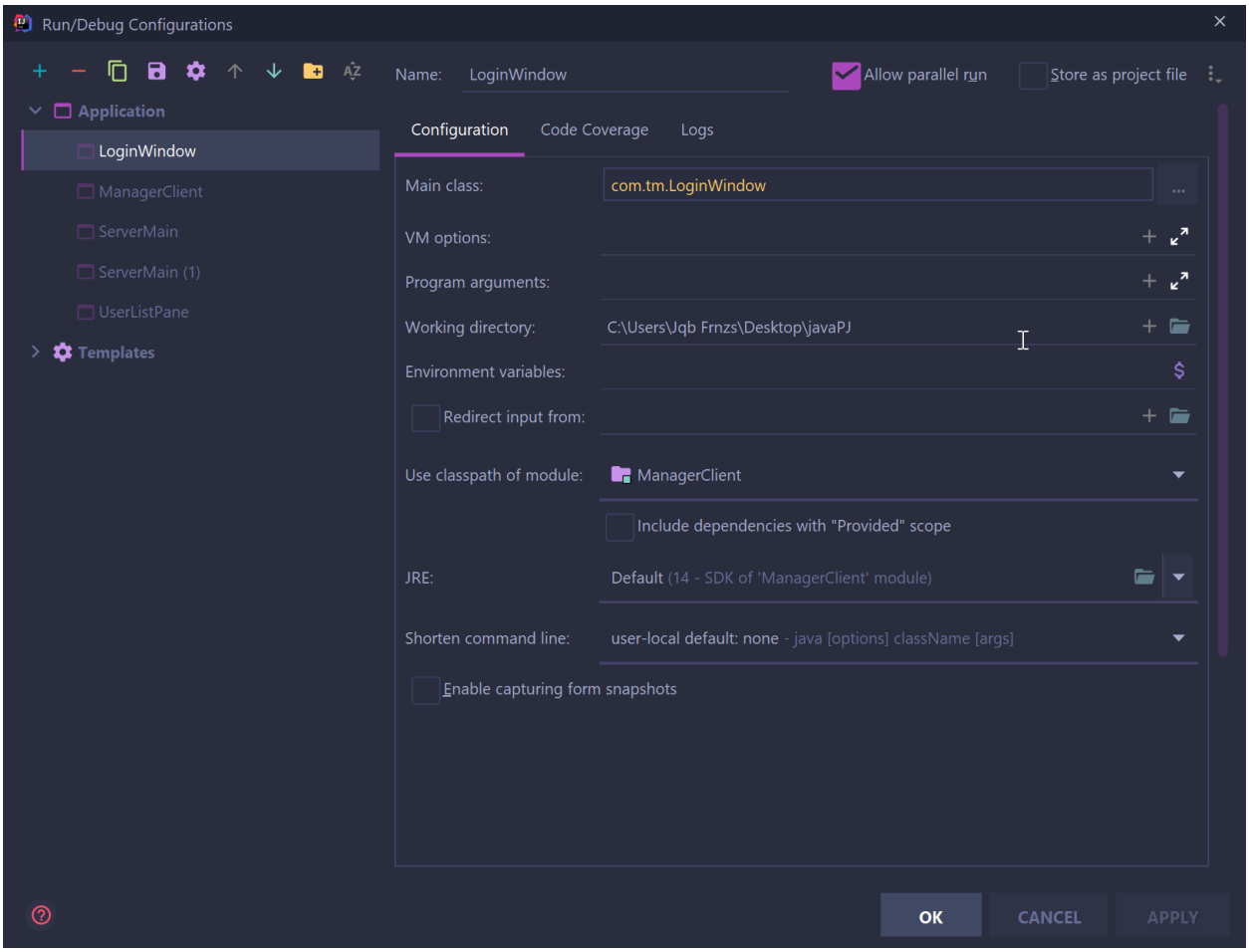
Interfejs *UserStatusListener* – metody abstrakcyjne dla UserListPane

3. Działanie programu:

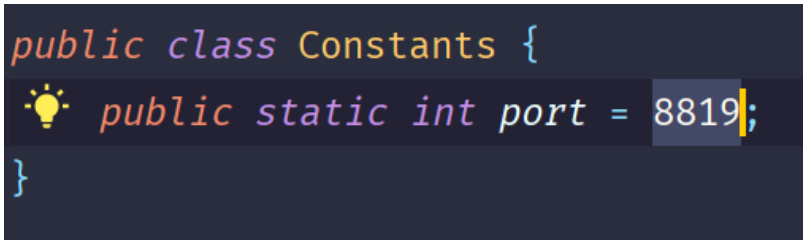
1. Przed uruchomieniem program trzeba upewnić się, że jest włączona opcja uruchomienia współbieżnego dla klasy LoginWindow, aby można było uruchomić wiele instancji logowania dla localhosta.

Dla Intellij IDEA:

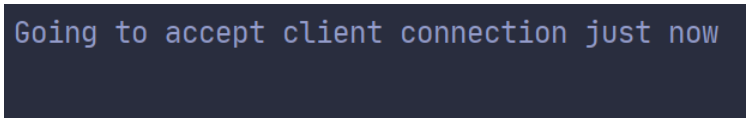
*Zakładka **Run -> Edit Configurations…***



3. Określamy numer portu serwera, który znajduje się w klasie **Constants** (domyślnie 8819):

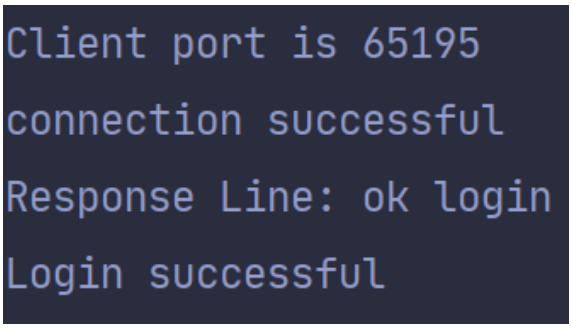


3. *Następnie uruchamiamy klasę* ***ServerMain****, powinien pojawić się w konsoli dla klasy komunikat:*

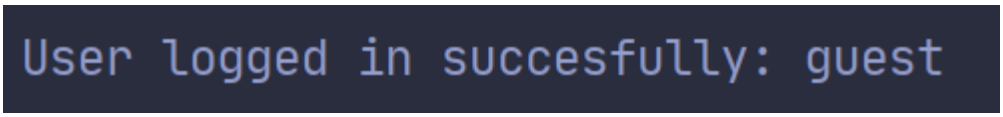


Właśnie została uruchomiona instancja serwera.

4. Uruchamiamy klasę kliencką **ManagerClient**, w konsoli tej klasy ukazuje się:



W klasie **ServerMain** za to komunikat:



*Instancja klienta działa, jej entry point (main) loguje automatycznie użytkownika **guest**. Konsola dla tej klasy wyświetla dalsze informacje dla kolejnych zalogowanych użytkowników. Jesteśmy gotowi na logowanie manualne poszczególnych użytkowników.*

LOGOWANIE GUI

5a. Uruchamiamy klasę **LoginWindow,** pojawia się okno logowania:

wpisujemy jedno z zestawów danych (zakodowane na sztywno, aktualnie brak podpiętej bazy danych):

1)

Login: adam

Hasło: adam

2)

Login: jqb

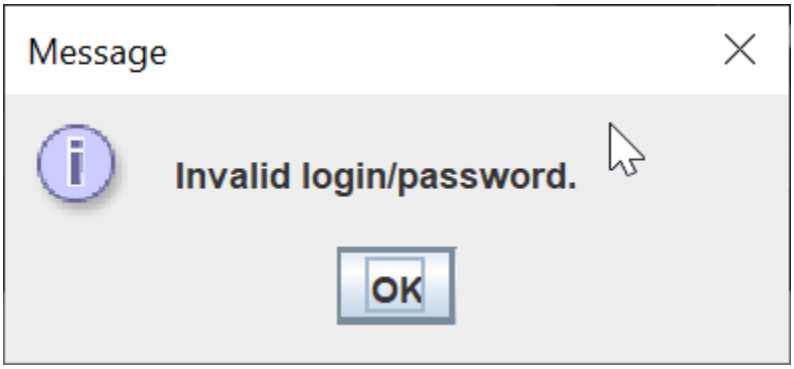
Hasło: jqb

3)

Login: tomek

Hasło: tomek

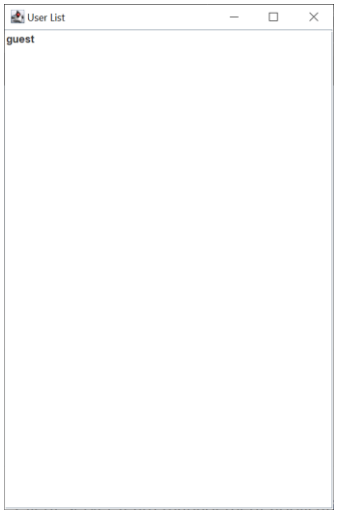
W przypadku wprowadzenia niepoprawnych danych wyświetla się baner:



Oraz komunikat w konsoli klasy **LoginWindow**:

```
Response Line: error login
```

W przypadku poprawnego logowania pojawia się lista zalogowanych użytkowników:



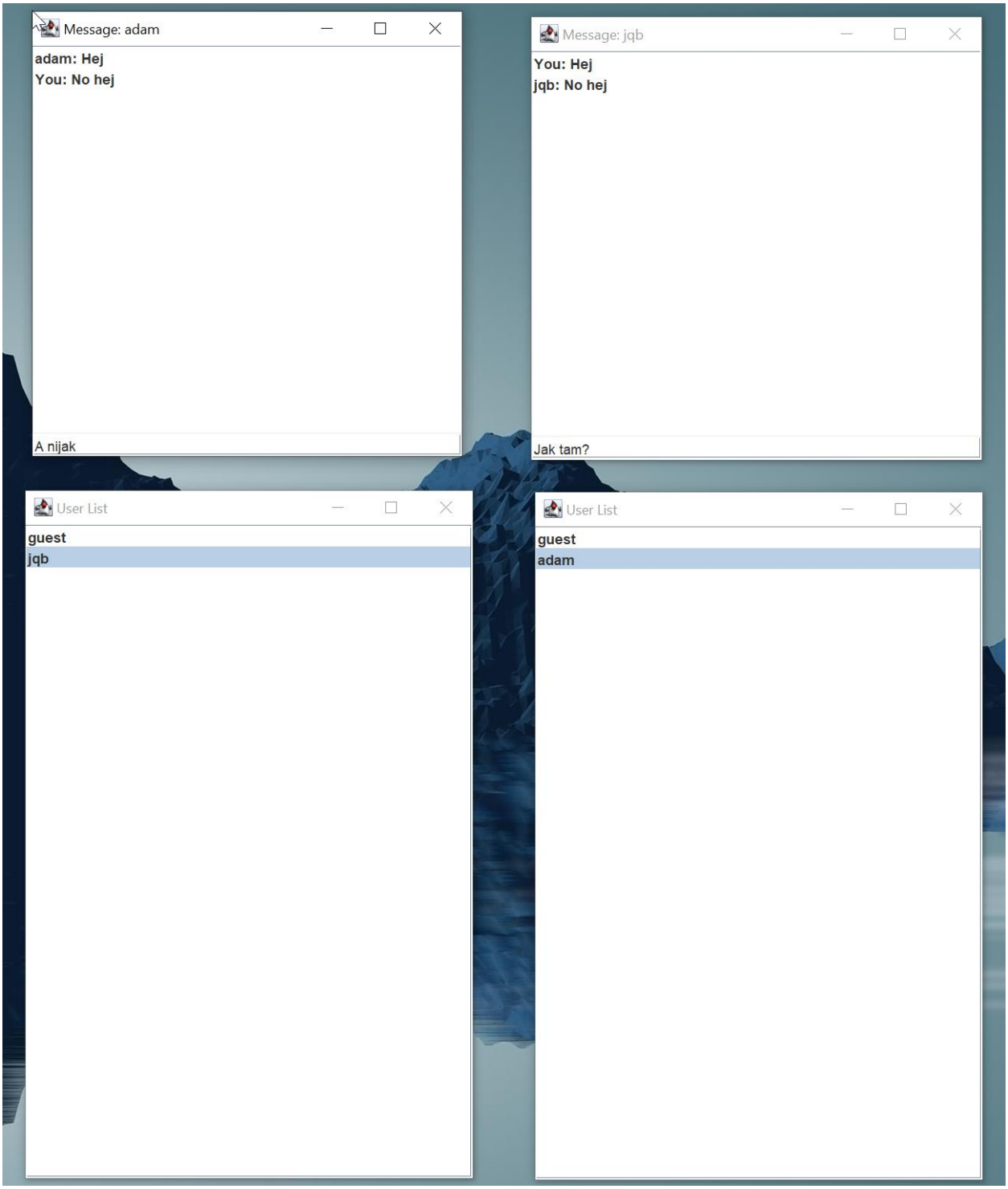
W konsoli **ServerMain**:

```
User logged in succesfully: adam
```

W konsoli **ManagerClient**:

```
ONLINE: adam
```

6a. Wysyłanie wiadomości:



W celu wysłania wiadomości dwukrotnie klikamy na login użytkownika z poziomu okna **‘User List’**,

Pojawia się okno z etykietą **‘Message: nazwaUżytkownika’.** Wiadomość wpisujemy w dolnym pasku (input field), a wysyłamy za pomocą wciśnięcia klawisza **ENTER.**

7a. Aby się wylogować z konta danego użytkownika należy nacisnąć domyślny przycisk przeznaczony do zamykania okna **‘User List’** (krzyżyk)

Wtedy w konsoli **ServerMain** pojawia się komunikat:

```
Client disconnected, waiting for another connection . . .
```

LOGOWANIE CLI

5b. Uruchamiamy dowolny emulator konsoli (np. powershell), a następnie za pomocą telnetu logujemy się na localhosta za pomocą wprowadzonego portu (domyślnie 8819):

```
PS C:\Users\Jqb Frnzs> telnet localhost 8819
```

Naciskamy ENTER

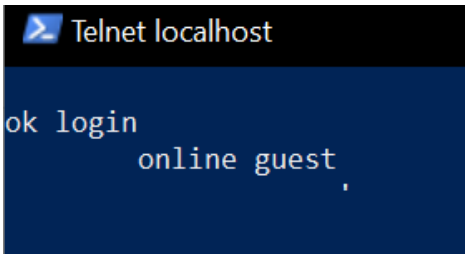
W konsoli **ServerMain** pojawia się komunikat:

```
Accepted connection from Socket[addr=/0:0:0:0:0:0:0:1,port=49572,localport=8819]
Going to accept client connection just now
```

6b. Logowanie opiera się o wpisanie słowa kluczowego **login**, następnie spacja, nazwaUżytkownika, spacja i hasłoUżytkownika, np.:

login jqb jqb

Przy poprawnym zalogowaniu w emulatorze wyświetla się:



Wyświatlają się powiadomienia kto jest online (tutaj *guest* z instancji początkowej **ManagerClient**)

W konsoli **ServerMain**:

```
User logged in succesfully: jqb
```

W konsoli **ManagerClient**:

```
ONLINE: jqb
```

7b. Wysyłanie wiadomości do konkretnego użytkownika:

Składnia opiera się o słowo kluczowe: **msg**

Instrukcja jest cięta na 3 tokeny: komenda (tutaj **msg**), potem adresat, a na końcu treść.

Wysłanie od użytkownika **adam** do **jqb**:

```
ok login
        online guest
                    online jqb
                               msg jqb Hej, jak tam?
```

Odebranie powyższej wiadomości od **adama** przez **jqb**:

```
ok login
        online guest
                    online adam
                               msg adam Hej, jak tam?
```

8b. Dołączanie do pokoju, aby wysyłać wiadomość rozgłoszeniową (broadcast message) do każdego użytkownika, który też dołączył do tego pokoju:

Słowo kluczowe **join** oraz znak **#** przed nazwą pokoju (bez spacji pomiędzy) :

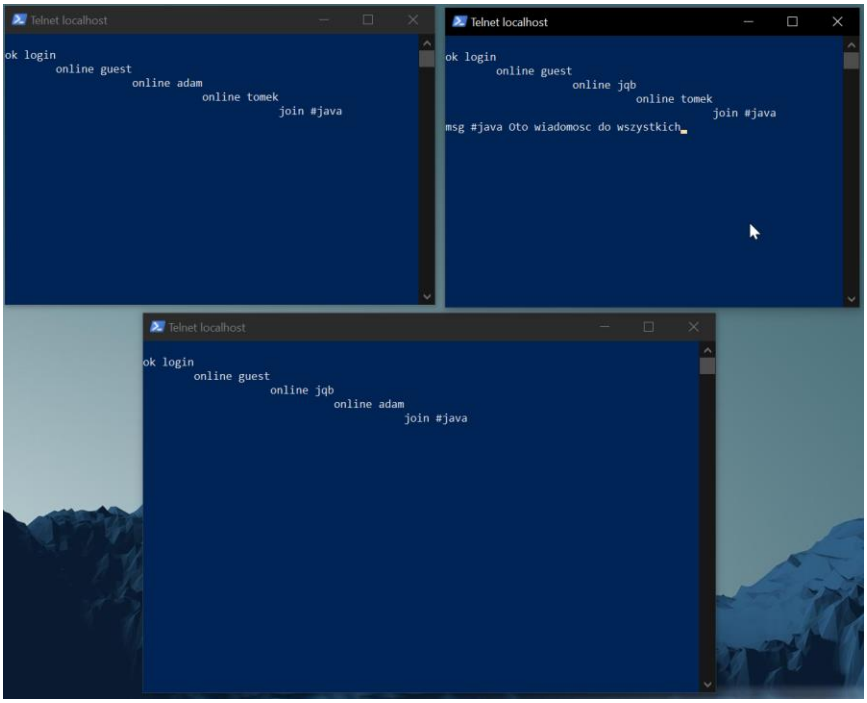
```
join #Hello
```

Pokoje nie istnieją wcześniej, są tworzone w czasie rzeczywistym przez użytkowników. Gdy jeden użytkownik dołączy do pokoju, a on nie istnieje, zostaje automatycznie utworzony.

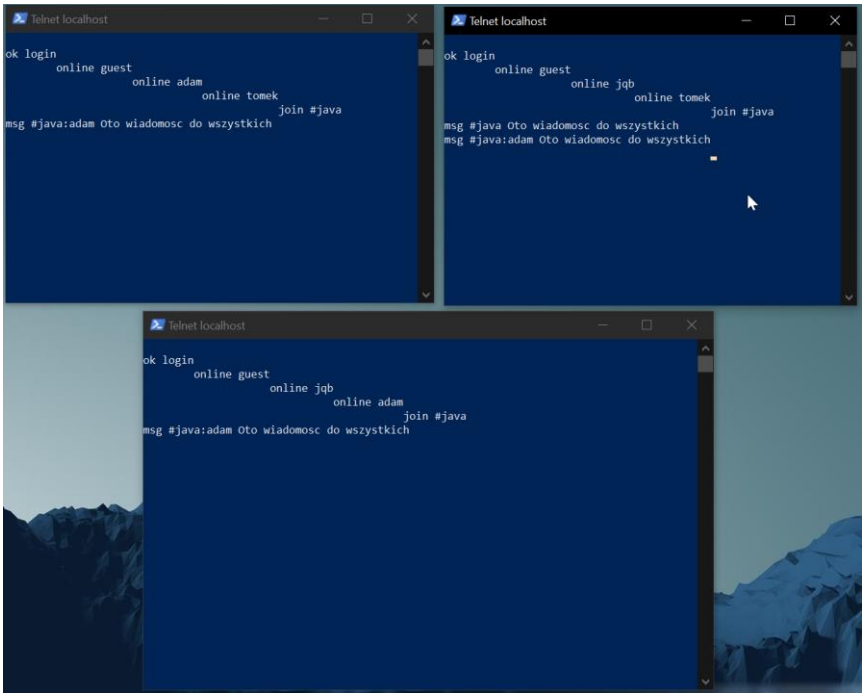
9b. Wysyłanie wiadomości do wszystkich członków pokoju:

Składnia komendy: ***msg #****nazwaPokoju treść wiadomości*

Przed wysłaniem:



Po wysłaniu:



10b. Opuszczanie pokoju przez użytkownika:

Składnia komendy: ***leave*** ***#****nazwaPokoju*

```
leave #java
```

Po opuszczeniu pokoju przez użytkownika, wiadomości do tego pokoju wysyłane, nie będą się mu wyświetlać.

10b. Wylogowanie się poszczególnych użytkowników:

Wykorzystywane są 2 komendy: **quit**

```
quit


Connection to host lost.
PS C:\Users\Jqb Frnzs>
```

lub **logoff**

```
                    logoff

Connection to host lost.
PS C:\Users\Jqb Frnzs>
```

Po wylogowaniu w konsoli **ManagerClient** wyświetla się komunikat:

```
OFFLINE: tomek
OFFLINE: jqb
```

4. Analiza kodu:

Klasa ServerWorker:

```
private void handleClientSocket() throws IOException, InterruptedException {
    InputStream inputStream = clientSocket.getInputStream();
    this.outputStream = clientSocket.getOutputStream();

    BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
    String line;
    while ( (line = reader.readLine()) != null) {
        String[] tokens = StringUtils.split(line);
        if (tokens != null && tokens.length > 0) {
            String cmd = tokens[0];
            if ("logoff".equals(cmd) || "quit".equalsIgnoreCase(cmd)) {
                handleLogoff();
                break;
            } else if ("login".equalsIgnoreCase(cmd)) {
                handleLogin(outputStream, tokens);
            } else if ("msg".equalsIgnoreCase(cmd)) {
                // split ONLY 3 tokens so you can send space-separated messages (user-user)
                String[] tokensMsg = StringUtils.split(line, separatorChars: null, max: 3);
                handleMessage(tokensMsg);
            } else if ("join".equalsIgnoreCase(cmd)) {
                handleJoin(tokens);
            } else if ("leave".equalsIgnoreCase(cmd)) {
                handleLeave(tokens);
            } else {
                String msg = "unknown " + cmd + "\n";
                outputStream.write(msg.getBytes());
            }
        }
```

Tworzony jest strumień wejścia i wyjścia dla klienta. Następnie za pomocą BufferReadera jest czytany strumień wejścia. Następnie czytana jest linia z konsoli. Dzielona na tokeny i za pomocą instrukcji warunkowych dokonywana jest weryfikacja poprawności komend takich jak **msg, login, join, leave, quit i logoff.** Pierwszym tokenem tokens[0] jest zawsze komenda np. login. Maksymalna ilość tokenów dla wprowadzanej z CLI instrukcji to 3 (3 słowa) np. *msg jqb Czesc.*

```java
private void handleLeave(String[] tokens) {
    if (tokens.length > 1) {
        String topic = tokens[1];
        topicSet.remove(topic);
    }
}
```

Obsługuje komendę leave, usuwa pokój (topic) z instancji użytkownika, jeśli go opuścił.

```java
public boolean isMemberOfTopic(String topic) {
    // checks if user is member of topic
    return topicSet.contains(topic);
}
```

Sprawdza czy użytkownik jest członkiem danego pokoju, tak aby wysyłać wiadomość rozgłoszeniową do poprawnej grupy docelowej.

```java
private void handleJoin(String[] tokens) {
    // adds topic to a user
    if (tokens.length > 1) {
        String topic = tokens[1];
        topicSet.add(topic);
    }
}
```

Obsługuje komendę join, dołącza pokój do instancji użytkownika.

```java
private void handleMessage(String[] tokens) throws IOException {
    String sendTo = tokens[1];
    String body = tokens[2];
    // checks if first char is '#'
    boolean isTopic = sendTo.charAt(0) == '#';

    List<ServerWorker> workerList = server.getWorkerList();
    for (ServerWorker worker : workerList) {
        // sends broadcast message if user is member of certain #topic
        if (isTopic) {
            if(worker.isMemberOfTopic(sendTo)) {
                String outMsg = "msg " + sendTo + ":" + login + " " + body + "\n";
                worker.send(outMsg);
            }
        } else {
            if (sendTo.equalsIgnoreCase(worker.getLogin())) {
                String outMsg = "msg " + login + " " + body + "\n";
                worker.send(outMsg);
            }
        }
    }
}
```

Obsługuję komendę **msg**, tokens[1] to adresat, tokens[2] to treść wiadomości. Sprawdza czy adresat ma w pierwszym znaku **'#'**, jeśli tak, jest to wiadomośc rozgłoszeniowa do pokoju (topic). W pętli sprawdzana jest cała lista zalogowanych użytkowników w poszukiwaniu tych z podanym w komendzie pokojem – do nich wysyłana jest wiadomość rozgłoszeniowa, w przeciwnym przypadku wysyłana jest zwyczajna, bezpośrednia wiadomość do konkretnego użytkownika za pomocą zwykłej składni **msg nazwaUżytkownika treść**.

```java
private void handleLogoff() throws IOException {
    server.removeWorker( serverWorker: this);
    List<ServerWorker> workerList = server.getWorkerList();

    // send other online users current user's status
    String onlineMsg = "offline " + login + "\n";
    for(ServerWorker worker : workerList) {
        // do not send ONLINE message to oneself
        if (!login.equals(worker.getLogin())) {
            worker.send(onlineMsg);
        }
    }
    clientSocket.close();
}
```

Obsługuje komendę **logoff**, usuwa użytkownika, który ją wpisał z listy zalogowanych użytkowników.

```java
public String getLogin() {
    return login;
}
```

Zwraca login pojedynczego użytkownika (instancji).

```java
private void send(String msg) throws IOException {
    if (login != null) {
        // do not send ONLINE message if clientSocket is opened but user not connected
        try {
            outputStream.write(msg.getBytes());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Obsługuję wysyłanie wiadomości do użytkowników np. komunikat o statusie ONLINE innego użytkownika.

Klasa Server:

```java
@Override
public void run() {
    try {
        ServerSocket serverSocket = new ServerSocket(serverPort);
        while(true) {
            System.out.println("Going to accept client connection just now");
            Socket clientSocket = serverSocket.accept();
            System.out.println("Accepted connection from " + clientSocket);
            ServerWorker worker = new ServerWorker( server: this, clientSocket);
            workerList.add(worker);
            worker.start();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Obsługuje ServerSocket, wysyła komunikaty odnośnie połączeń klienckich z serwerem. Dodaje użytkownika do listy ONLINE.

```java
public void removeWorker(ServerWorker serverWorker) {
    workerList.remove(serverWorker);
}
```

Usuwa użytkownika z listy ONLINE.

Klasa Server:

```
public class ServerMain {
    public static void main(String[] args) {
        Server server = new Server(Constants.port);
        server.start();
    }
}
```

Główny entry-point programu. Uruchamia całą infrastrukturę serwera. Korzysta z klasy ze stałą posiadającą numer lokalnego portu dla serwera.

Klasa UserListPane:

```
userListUI.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount() > 1) {
            String login = userListUI.getSelectedValue();
            MessagePane messagePane = new MessagePane(client, login);

            JFrame f = new JFrame( title: "Message: " + login);
            f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            f.setSize( width: 500,  height: 500);
            f.getContentPane().add(messagePane, BorderLayout.CENTER);
            f.setVisible(true);
        }
    }
});
```

Obsługuje podwójne kliknięcie na nazwę użytkownika w liście użytkowników GUI, po którym wyświetla się okno z pisaniem wiadomości.

```java
// add user to the model
@Override
public void online(String login) { userListModel.addElement(login); }

// removes user from the model
@Override
public void offline(String login) { userListModel.removeElement(login); }
```

Funkcje obsługują dodawanie/usuwanie z listy użytkowników, po tym jak oni się wylogują.

Klasa MessagePane:

```java
inputField.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            String text = inputField.getText();
            client.msg(login, text);
            listModel.addElement("You: " + text);
            inputField.setText("");
        } catch (IOException e1) {
            e1.printStackTrace();
        }
    }
});
```

Nasłuchuje na pole z wysyłaniem wiadomości, następnie wyświetla wysłaną wiadomość w polu rozmowy oraz zeruje input-field, aby można było wprowadzić kolejną wiadomość.

```java
@Override
public void onMessage(String fromLogin, String msgBody) {
    if (login.equalsIgnoreCase(fromLogin)) {
        String line = fromLogin + ": " + msgBody;
        listModel.addElement(line);
    }
}
```

Obsługuje wyświetlanie wiadomości od nadawcy do odbiorcy w panelu rozmowy.

Klasa ManagerClient:

```java
public void addUserStatusListener(UserStatusListener listener) { userStatusListeners.add(listener); }
public void removeUserStatusListener(UserStatusListener listener) { userStatusListeners.remove(listener); }

public void addMessageListener(MessageListener listener) { messageListeners.add(listener); }
public void removeMessageListener(MessageListener listener) { messageListeners.remove(listener); }
```

Dodawanie/usuwanie nasłuchu statusu/wiadomości.

Klasa LoginWindow

```java
JPanel p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
p.add(loginField);
p.add(passwordField);
p.add(loginButton);
```

Stworzenie panelu logowania, dodanie przycisku, inputów dla loginu i hasła.

5. Wnioski i uwagi

- projekt pozwolił utrwalić fundamenty programowania obiektowego takie jak: enkapsulacja, dziedziczenie czy polimorfizm oraz zrozumieć istotę połączenia klient-serwer

- miałem okazję bliżej zapoznać się, z systemem kontroli wersji GIT oraz ekosystemem Github, które są niezastąpione przy pracy nad bardziej rozbudowanymi projektami

- commity GITa są szczególnie przydatne przy powrocie do pracy nad projektem po dłuższej przerwie, dlatego powinny być precyzyjne i nieobszerne

- Swing okazał się bardzo dobrą opcją jako biblioteka graficzna do prostego zastosowania