



Politechnika Krakowska
im. Tadeusza Kościuszki



Wydział Inżynierii
Elektrycznej i Komputerowej

POLITECHNIKA KRAKOWSKA

WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I KOMPUTEROWEJ

ZAJĘCIA PROJEKTOWE Z PROGRAMOWANIA W JĘZYKU JAVA

TEMAT PROJEKTU:

**Gry „Poker” oraz „Remik” w oparciu o architekturę klient – serwer w
języku Java**

GRUPA PROJEKTOWA:

P2 w składzie:

Kwaśniewski Jakub, Kwak Krzysztof, Starzyk Konrad

PROWADZĄCY PROJEKT:

dr. inż. Sławomir Bąk

DATA WYKONANIA:

13 czerwca 2024



SPIS TREŚCI

Wstęp	3
Struktura projektu	4
Pakiety	8
Serwer i klient – komunikacja, obsługa wielu stołów i zwrot wyniku z obiektu Callable	12
Wygląd i działanie okien aplikacji, GUI (logowanie, rejestracja, menu, lobby gry)	17
Działanie gier (poker i remik)	26
Rankingi	39
Baza danych	42
Podsumowanie	43



WSTĘP

Tematem projektu była implementacja w języku Java karcianych gier wieloosobowych - klasycznego **Pokera** w odmianie Texas Hold'em oraz **Remika** – w oparciu o architekturę klient – serwer. Założenia techniczne projektu były następujące:

- Nielimitowana (w rozsądnym zakresie) liczba równoległe trwających rozgrywek / połączonych klientów
- Wykorzystanie **bazy danych** jako formy przechowywania informacji
- Praca z wątkami (*Thread*, *Callable* etc.)
- Łączność klient-serwer oparta o obiekty typu **Socket** i **ServerSocket**
- Graficzny interfejs użytkownika w oparciu o bibliotekę **JavaFX**

Do realizacji projektu wykorzystaliśmy poznane podczas zajęć oraz własnych poszukiwań narzędzia i funkcjonalności dostarczane przez JDK i zewnętrzne biblioteki. Aplikacja działa w oparciu o **JDK w wersji 21** i korzysta z narzędzia do zarządzania projektem **Apache Maven**. Całość kodu była tworzona w IDE dla języka Java od firmy JetBrains, **IntelliJ IDEA**.



STRUKTURA PROJEKTU

Główny wątek serwera uruchamiany jest w klasie **GameServer** – nawiązywane jest statyczne połączenie z bazą danych, z którego korzystają różne wątki. Następnie serwer oczekuje na połączenia od klientów (każde z nich jest obsługiwane za pomocą **ClientHandlera**, odbierającego i przetwarzającego **Packety** – o nich więcej później), którzy są dodawani do listy **handlers** i tworzą nowy wątek. Najpierw omówimy grę **Poker**.

```
public void start(int port) throws IOException {  
    executorService = Executors.newFixedThreadPool( nThreads: 100);  
    serverSocket = new ServerSocket(port);  
    taskManager = new TaskManager();  
    System.out.println("----- URUCHOMIONO SERWER -----\\n");  
    System.out.println("ADRES IP SERWERA: " + serverSocket.getInetAddress());  
    System.out.println("PORT: " + serverSocket.getLocalPort());  
    while (true) {  
        handlers.add(new ClientHandler(serverSocket.accept()));  
        handlers.getLast().start();  
    }  
}
```

Fig. 1 Start serwera

Klasa wewnętrzna **TaskManager** odpowiada za obsługę rozpoczętych przez graczy rozgrywek. Wykorzystuje ona inną klasę, **FutureTaskCallback** (dziedziczącą z klasy **FutureTask<>**), w celu przeprowadzenia gry i zwrócenia listy zwycięzców serii rozdań i załadowania wyników rozgrywki do bazy danych.



```
private static class TaskManager {  
    2 usages  
    ExecutorService executor;  
    2 usages  
    private List<PokerGame> tasks;  
    1 usage  
    private List<FutureTaskCallback> callbackTasks;  
  
    1 usage  pizzaogkebab  
    public TaskManager(){  
        executor = Executors.newCachedThreadPool();  
        tasks = new ArrayList<>();  
        callbackTasks = new ArrayList<>();  
    }  
  
    1 usage  pizzaogkebab  
    public void startTask(PokerGame pokerGame){  
        tasks.add(pokerGame);  
        FutureTaskCallback callback = new FutureTaskCallback(pokerGame);  
        executor.execute(callback);  
    }  
}
```

Fig. 2 Klasa TaskManager

ClientHandler stanowi pośrednik pomiędzy klientem (klasa **Client**) a serwerem po stronie serwera, każdy działa na osobnym wątku.



```
public class ClientHandler extends Thread {  
    24 usages  
    private Player player;  
    8 usages  
    private Socket clientSocket;  
    3 usages  
    private ObjectOutputStream out;  
    3 usages  
    private ObjectInputStream in;  
    6 usages  
    private String UUID;  
  
    pizzaogkebab  
    @Override  
    public String toString() { return player.getPlayerData(); }  
    1 usage pizzaogkebab  
    public ClientHandler(Socket socket) {...}  
    pizzaogkebab +2  
    public void run() {...}  
    1 usage pizzaogkebab +1  
    public void stopConnection() {...}  
    1 usage pizzaogkebab +2  
    private void parseRequest(Packet request) {...}  
    31 usages pizzaogkebab  
    public void sendPacket(Packet packet) {...}  
    pizzaogkebab  
    public Player getPlayer() { return player; }  
}
```

Fig. 3 Klasa ClientHandler



Klasa **Client** steruje JavaFX'owymi kontrolerami poszczególnych okien, otrzymując pakiety instrukcji z serwera, i przesyła do serwera pakiety zawierające dane o działaniach użytkowników.

```
public class Client extends Thread {  
    5 usages  
    private Socket clientSocket;  
    public ObjectOutputStream out;  
    3 usages  
    private ObjectInputStream in;  
    3 usages  
    private LoginController lc;  
    11 usages  
    private PokerLobbyController plbc;  
    11 usages  
    private RummyLobbyController rlbc;  
    2 usages  
    private MenuController mc;  
    2 usages  
    private RegisterController rc;  
    13 usages  
    private PokerTableController ptc;  
    3 usages  
    private PokerRankingController prc;  
    3 usages  
    private RemikRankingController rrc;  
    1 pizzaogkebab +1  
    public void run() {...}  
    6 usages 1 Konrad Starzyk  
    public void stopConnection() throws IOException {...}  
    1 usage 1 pizzaogkebab +2  
    private void parseResponse(Packet response) {...}  
    1 usage 1 Konrad Starzyk  
    private void goToPokerRanking(HashMap<String,Integer> RankingMap){...}  
    1 usage 1 Konrad Starzyk  
    private void goToRemikRanking(HashMap<String,Integer> RankingMap){...}
```

Fig. 4 Klasa Client (pola i część metod)



PAKIETY

Do komunikacji pomiędzy klientem i serwerem używane są pakiety reprezentowane przez klasę **Packet**. Klasa ta służy do organizacji danych, które mają być przesyłane przez klienta lub serwer. Dane obejmują różne informacje takie jak wiadomości tekstowe, liczby czy serializowane obiekty. Pakiety umożliwiają łatwe grupowanie, a następnie odczytywanie przesyłanych danych.

Klasa, z której dziedziczą wszystkie pozostałe rodzaje pakietów jest klasa **Packet**. Zawiera konstruktor oraz pola:

- **desc** typu **String** – opis służący głównie do debugowania i sprawdzania poprawności pakietów
- **type** typu **PacketType** – rodzaj przesyłanego pakietu

```
public class Packet implements Serializable {  
    private PacketType type;  
    private String desc;  
  
    public Packet(PacketType type, String data) {  
        this.type = type;  
        this.desc = data;  
    }  
  
    public PacketType getType() {  
        return type;  
    }  
  
    public String getDesc() {  
        return desc;  
    }  
}
```




W projekcie zdefiniowany został typ enumeracyjny **PacketType**, który przetrzymuje wszystkie rodzaje pakietów. Rodzaje używane są przez serwer oraz klienta do identyfikacji pakietu i następnego wywołania odpowiedniej funkcji.

```
public enum PacketType {  
    ACK, LOGIN, REGISTER, GAME, CREATEGAME, JOINGAME, GAME_READY_STATUS,  
    REMIK, RANKING  
}
```

Klasy reprezentujące pakiety dziedziczące po Packet:

- **LoginPacket** – wysyłany przy próbie logowania oraz wylogowania z konta. Rozszerza Packet o pola login, password, player oraz status służący do określenia wyniku zapytania użytkownika do serwera.
- **RegisterPacket** – wysyłany przy próbie założenia konta. Rozszerza Packet o pola login, password, email, date, salt, player zawierające dane użytkownika oraz status służący do określenia wyniku zapytania użytkownika do serwera.
- **CreateGamePacket** – wysyłany w momencie próby utworzenia nowego lobby gry Poker lub Remik. Rozszerza Packet o pola UUID zawierające unikalny kod utworzonej gry oraz gameType definiujący rodzaj gry.
- **JoinGamePacket** – wysyłany przy próbie dołączenia do lobby. Rozszerza Packet o pola UUID zawierające unikalny kod gry do której użytkownik chce dołączyć, oraz gameType definiujący rodzaj gry.



- **GameReadyPacket** – wysyłany przy zmianie statusu gotowości w lobby przez użytkownika. Rozszerza packet o pola UUID zawierające unikalny kod gry, w której lobby znajduje się użytkownik oraz player definiujący gracza, który wysła konkretne żądanie.
- **GamePacket** – wysyłany podczas każdego ruchu wykonanego w grze. Rozszerza Packet o pola Status definiujące rodzaj działania jakie powinien podjąć serwer/klient, move_type definiujące rodzaj ruchu użytkownika, pola liczbowe zawierające aktualny stan konta gracza oraz pola zawierające karty gracza.
- **RankingPacket** – wysyłany podczas próby uzyskania rankingu z gry Poker lub Remik. Rozszerza Packet o pole status definiujące rodzaj gry której ranking chcemy zobaczyć oraz rankingMap zawierające ranking.

Aby możliwe było przesyłanie obiektów przez sockety każdy z przesyłanych obiektów musi implementować interfejs Serializable. Interfejs ten umożliwia zapisanie obiektów do strumienia aby następnie można było przesłać go przez sieć. Obiekt konwertowany jest przez wysyłającego (serwer lub klienta) do sekwencji bajtów, następnie zostaje wysłany i konwertowany przez odbiorcę z powrotem na obiekt.



Przykładowy wygląd pakietu dziedziczącego po Packet:

```
public class GameReadyPacket extends Packet {
    private String UUID;
    private Player player;
    public enum Status {
        READY,
        NOT_READY,
    }
    public enum GameType {
        POKER,
        RUMMY,
    }
    private GameType gameType;
    private Status status;
    public GameReadyPacket(String data, Status status) {
        super(PacketType.GAME_READY_STATUS, data);
        this.status = status;
    }
    public GameReadyPacket(String data, GameType gameType, Player player,
String uuid, Status status) {
        super(PacketType.GAME_READY_STATUS, data);
        this.player = player;
        this.UUID = uuid;
        this.status = status;
        this.gameType = gameType;
    }
    public GameReadyPacket(String data, String uuid, Status status) {
        super(PacketType.GAME_READY_STATUS, data);
        this.UUID = uuid;
        this.status = status;
    }
    public GameReadyPacket(String data, GameType gameType, String uuid,
Status status) {
        super(PacketType.GAME_READY_STATUS, data);
        this.UUID = uuid;
        this.status = status;
        this.gameType = gameType;
    }
}
```



SERWER I KLIENT – KOMUNIKACJA

Komunikacja między serwerem i klientem odbywa się na zasadzie ciągłego, wzajemnego nasłuchiwanie serializowanych pakietów, które po rozpakowaniu są odpowiednio interpretowane, zgodnie z zawartością. Funkcja **ClientHandler.parseRequest()** odpowiada za przetwarzanie przez serwer pakietów odebranych od klienta.

```
public void run() {  
    Packet request;  
    while (true) {  
        try {  
            request = (Packet) in.readObject();  
            if (request == null) break;  
            parseRequest(request);  
        } catch (EOFException e) {  
            System.err.println("Klient zamknął połączenie: " + clientSocket.getInetAddress());  
            break;  
        } catch (IOException | ClassNotFoundException e) {  
            System.err.println("Nie udało się odczytać pakietu");  
            GameServer.removeNick(this.player.getPlayerData());  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Fig. 5 Funkcja run() klasy ClientHandler odpowiedzialna za przechwytywanie pakietów



Pakiety od serwera do klienta odbierane są w klasie **Client**. Każdy przyjęty pakiet jest przetwarzany przez funkcję **Client.parseResponse()** która zgodnie z jego zawartością wywołuje odpowiednie funkcje w kontrolerach okienek.

```
public void run() {  
    try {  
        sendPacket(new Packet(PacketType.ACK, data: "hello"));  
        Packet response;  
        while (clientSocket.isConnected()) {  
            try {  
                response = (Packet) in.readObject();  
                if (response == null) break;  
                parseResponse(response);  
            } catch (ClassNotFoundException e) {  
                throw new RuntimeException(e);  
            } catch (SocketException e) {  
                System.err.println("Połączenie z serwerem zostało zamknięte.");  
                break;  
            }  
        }  
        stopConnection();  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Fig. 6 Funkcja run() klasy Client czekająca na odpowiedź z serwera



OBSŁUGA WIELU STOŁÓW (ROZGRYWEK)

Serwer umożliwia prowadzenie kilku rozgrywek w tym samym czasie dzięki uruchamianiu każdej rozgrywki na nowym wątku. Zajmuje się tym wspomniana wcześniej klasa **TaskManager** – posiada ona własny **ExecutorService** i listę wątków **PokerGame** typu Callable, zwracających listę wygranych graczy po zakończeniu rozgrywki. Ich wynik jest przechwytywany przez obiekt **FutureTaskCallback**, który po zakończeniu gry dodaje odpowiedni wpis w bazie danych.

```
public void startTask(PokerGame pokerGame){  
    tasks.add(pokerGame);  
    FutureTaskCallback callback = new FutureTaskCallback(pokerGame);  
    executor.execute(callback);  
}
```

Fig. 7 Metoda startTask()



```
public class PokerGame implements Callable<HashMap<ClientHandler, Integer>> {
    2 usages
    String id;
    5 usages
    int playersReady;
    43 usages
    List<ClientHandler> players;
    15 usages
    List<Player> playersData;
    6 usages
    ArrayList<Karta> tableCards;
    no usages
    private Talia deck = new Talia( czyJokery: false);
    9 usages
    final Lock lock = new ReentrantLock();
    4 usages
    final Condition next = lock.newCondition();
    27 usages
    Integer currentBid;
    19 usages
    Integer moneyPool;
    12 usages
    List<ClientHandler> recentWinners = new ArrayList<>();
    2 usages  📌 pizzaogkebab
    public void handlerRaiseBetween(ClientHandler ch, Integer r) {...}
    1 usage  📌 kwakczuszka +1
    public void handlerRaise(ClientHandler ch, Integer r) {...}
    1 usage  📌 pizzaogkebab +1
    public void handlerCall(ClientHandler ch) {...}
    1 usage  📌 kwakczuszka +1
    public void handlerFold(ClientHandler ch) {...}
    no usages  📌 kwakczuszka +1
    public void updateMoneyPool() {...}
    1 usage  📌 kwakczuszka +1
    public boolean canProceed() {...}
    3 usages  📌 kwakczuszka +1
    private ArrayList<ClientHandler> Showdown() {...}
```

Fig. 8 Pola i część metod klasy PokerGame



ZWRACANIE WYNIKÓW PO ZAKOŃCZONEJ ROZGRYWKIE (CALLABLE)

Każdy wątek **PokerGame** po zakończeniu działania zwraca obiekt typu **HashMap<ClientHandler, Integer>** - mapę zawierającą pary ClientHandler'ów, czyli handlersy klientów z co najmniej jedną wygraną w danej serii rozdania, i liczbę ich wygranych. Na tej podstawie, do bazy danych przesyłane są wyniki graczy.

```
public class FutureTaskCallback extends FutureTask<HashMap<ClientHandler, Integer>> {
    1 usage  ± pizzaogkebab
    public FutureTaskCallback(Callable<HashMap<ClientHandler, Integer>> callable) { super(callable); }

    ± pizzaogkebab *
    public void done(){
        if (isCancelled()) System.out.println("gra przerwana ;{");
        else{
            try {
                HashMap<ClientHandler, Integer> chMap = get();
                for (Map.Entry<ClientHandler, Integer> set : chMap.entrySet()) {
                    System.out.println("wynik z wątku wynikowego: " + set.getKey() + ": " + set.getValue());
                    Statement st = connection.createStatement();
                    String sql = "UPDATE pokerRanking SET Points = Points + " + set.getValue() + " + " + set.getValue() + " + " +
                                "WHERE UserID = " + set.getKey().getPlayer().getPlayerID() + " ";
                    st.executeUpdate(sql);
                }
            } catch (InterruptedException | ExecutionException | SQLException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Fig. 9 Klasa FutureTaskCallback



WYGLĄD I DZIAŁANIE POSZCZEGÓLNYCH OKIEN APLIKACJI

LOGOWANIE

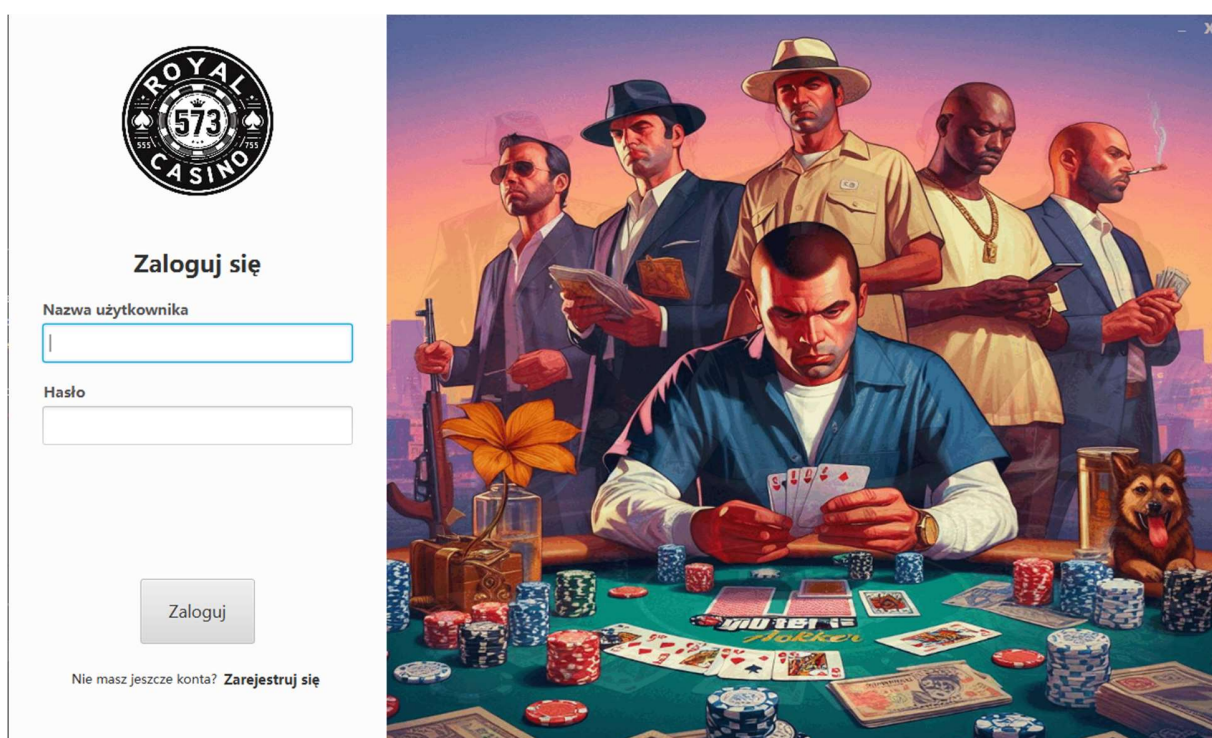


Fig. 10 Ekran logowania

Standardowo w celu zalogowania się do gry podać musimy podać login i hasło użytkownika. Przy rejestracji, poza loginem i hasłem, podajemy również adres e-mail oraz datę urodzenia (w myśl Ustawy z dnia 19 listopada 2009 r. o grach hazardowych, **osoby nieletnie nie mogą brać w nich udziału**).



REJESTRACJA

Fig. 11 Okno rejestracji

Podczas rejestracji i logowania nasze oko ucieszy pokaz slajdów złożony z kilku wygenerowanych przez model DALL-E3 grafik, utrzymanych w konwencji gry hazardowej.

PROCES REJESTRACJI I LOGOWANIA

Po wprowadzeniu danych, sprawdzenia ich unikalności i weryfikacji wieku w procesie rejestracji następuje utworzenie obiektu Callable **PassHash** – dla podanego hasła generuje on salt (ciąg bajtów, nieco więcej o nim później) i tworzy unikalny hash hasła z dodanym salt'em w



oparciu o algorytm **PBKDF2WithHmacSHA1** (Password-based-Key-Derivative-Function v2 with hash-based message authentication code SHA1) i kodowanie **Base64** w celu łatwiejszego przechowywania ciągu bajtów w bazie danych. Po wykonaniu tych zadań zwracana jest para [hash hasła, salt] i następuje dodanie rekordu do bazy danych.

```
@Override
public Pair<String, String> call() throws Exception {
    SecureRandom random = new SecureRandom();
    byte[] salt = new byte[16];
    random.nextBytes(salt);
    KeySpec spec = new PBEKeySpec(prehash.toCharArray(), salt, iterationCount: 65536, keyLength: 128);
    SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    byte[] hash = factory.generateSecret(spec).getEncoded();
    return new Pair<>(Base64.getEncoder().encodeToString(hash), Base64.getEncoder().encodeToString(salt));
}
```

Fig. 12 Metoda call() obiektu PassHash, w której następuje enkrypcja danych

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
FutureTask<Pair<String, String>> future = (FutureTask<Pair<String, String>>)
    executorService.submit(new PassHash(password));
this.password = future.get().getKey();
this.salt = future.get().getValue();
```

Fig. 13 Generowanie hashy w obiekcie RegisterPacket

Podczas logowania, z bazy pobierane są hash saltowanego hasła i salt a od użytkownika wprowadzone hasło. Statyczna metoda **PassHash.comparePasswd()** sprawdza poprawność wprowadzonego przez użytkownika hasła, dodając do niego pobrany z bazy salt, hashując całość i porównując ze sobą hashe.



```
public static boolean comparePasswd(String psd, String salt, String hash)
    throws NoSuchAlgorithmException, InvalidKeySpecException {
    byte[] slt = Base64.getDecoder().decode(salt);
    KeySpec spec = new PBEKeySpec(psd.toCharArray(), slt, iterationCount: 65536, keyLength: 128);
    SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    byte[] hash1 = factory.generateSecret(spec).getEncoded();
    return Base64.getEncoder().encodeToString(hash1).equals(hash);
}
```

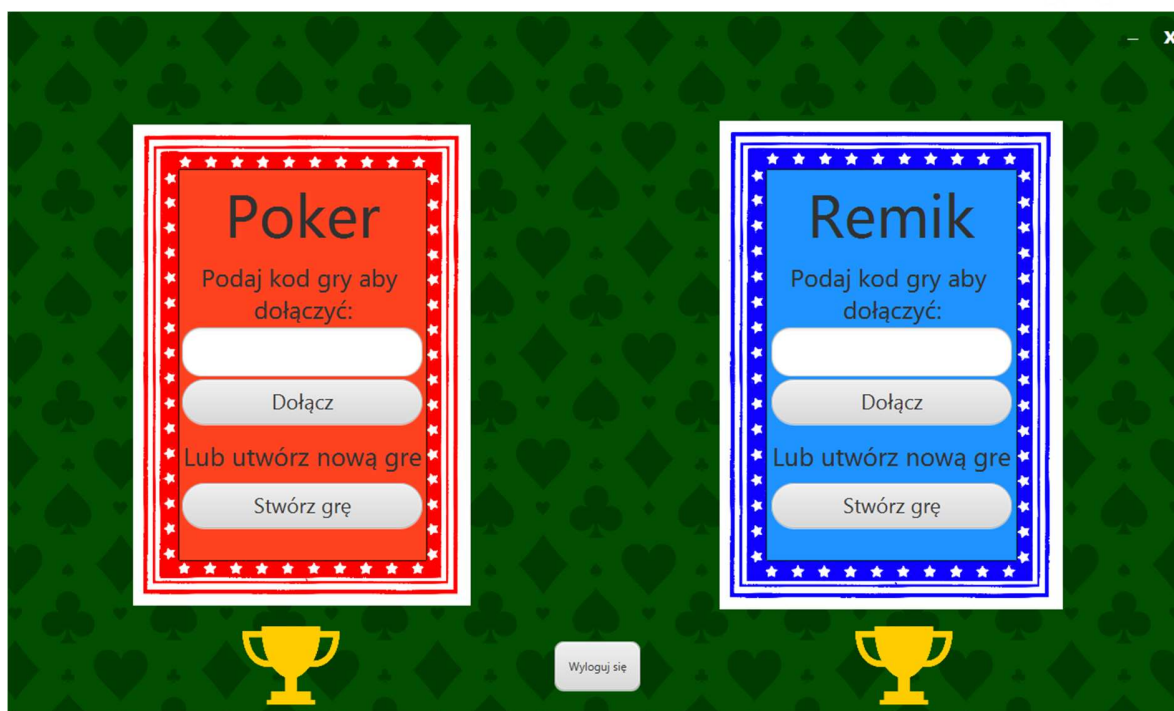
Fig. 14 Metoda `comparePasswd()` klasy `PassHash`

Salt (pseudolosowa sekwencja bajtów) wykorzystywany jest po to, aby zagwarantować unikalność hashy również przy identycznych hasłach więcej niż jednego użytkownika – szanse na uzyskanie w zastosowanym przez nas procesie dwa razy takiego samego hashu są w zasadzie zerowe.

MENU GŁÓWNE

Po zalogowaniu klient jest przenoszony do menu, które podzielone jest na dwa główne obszary dotyczące **pokera** i **remika**. Każdy z tych obszarów zawiera możliwość stworzenia nowego lobby gry lub dołączenia od istniejącej gry za pomocą unikatowego kodu generowanego przy tworzeniu lobby, jak i również sprawdzenia aktualnego rankingu dziesięciu najlepszych graczy danej gry, któremu odpowiadają ikony pucharów, widoczne pod kartami. Poza tymi dwoma obszarami jest jeszcze możliwość zminimalizowania lub zamknięcia aplikacji oraz wylogowanie z konta.

Wygląd menu głównego:



Kontroler *MenuController.java* przypisany jest do pliku *Menu.fxml* i zarządza akcjami wykonywanymi z interfejsie graficznym.

Najważniejsze metody:

- Tworzenie lobby pokera:

```
@FXML
public void createPokerGame() {
    String uniqueID = UUID.randomUUID().toString();
    System.out.println(uniqueID);
    Packet p = new CreateGamePacket("game creation", uniqueID,
    CreateGamePacket.GameType.POKER);
    Main.client.sendPacket(p);
}
```

Metoda ta na początku generuje unikalny kod niezbędny do dołączenia do lobby, który inni gracze podają podczas dołączania. Następnie tworzony jest i wysłany pakiet *CreateGamePacket* zawierający: wygenerowany wcześniej kod oraz identyfikator typu gry *POKER*.



- Dołączanie do lobby remika:

```
@FXML
public void joinRummyGame() {
    String uuid = gameId1.getText();
    if (uuid.isEmpty()) {
        System.out.println("error");
    } else {
        JoinGamePacket packet = new JoinGamePacket("JOIN", uuid,
JoinGamePacket.GameType.RUMMY, JoinGamePacket.Status.JOIN);
        Main.client.sendPacket(packet);
    }
}
```

Pobierany jest kod podany w *TextField* przeznaczonym dla danej gry, w tym przypadku *gameID1*, a następnie jeśli tworzony jest pakiet typu *JoinGamePacket* zawierający: podany kod dołączenia, identyfikator typu gry *RUMMY* oraz identyfikator statusu *JOIN*.

- Wylogowanie:

```
@FXML
public void LogOut() {
    Main.client.sendPacket(new LoginPacket("Logging out",
LoginPacket.Status.LOGOUT));
}
```

Po kliknięciu przycisku „Wyloguj się”, klient wysyła pakiet *LoginPacket* ze statusem *LOGOUT*.



Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Inżynierii
Elektrycznej i Komputerowej



Wydział Inżynierii
Elektrycznej i Komputerowej

Wydział Inżynierii
Elektrycznej i Komputerowej

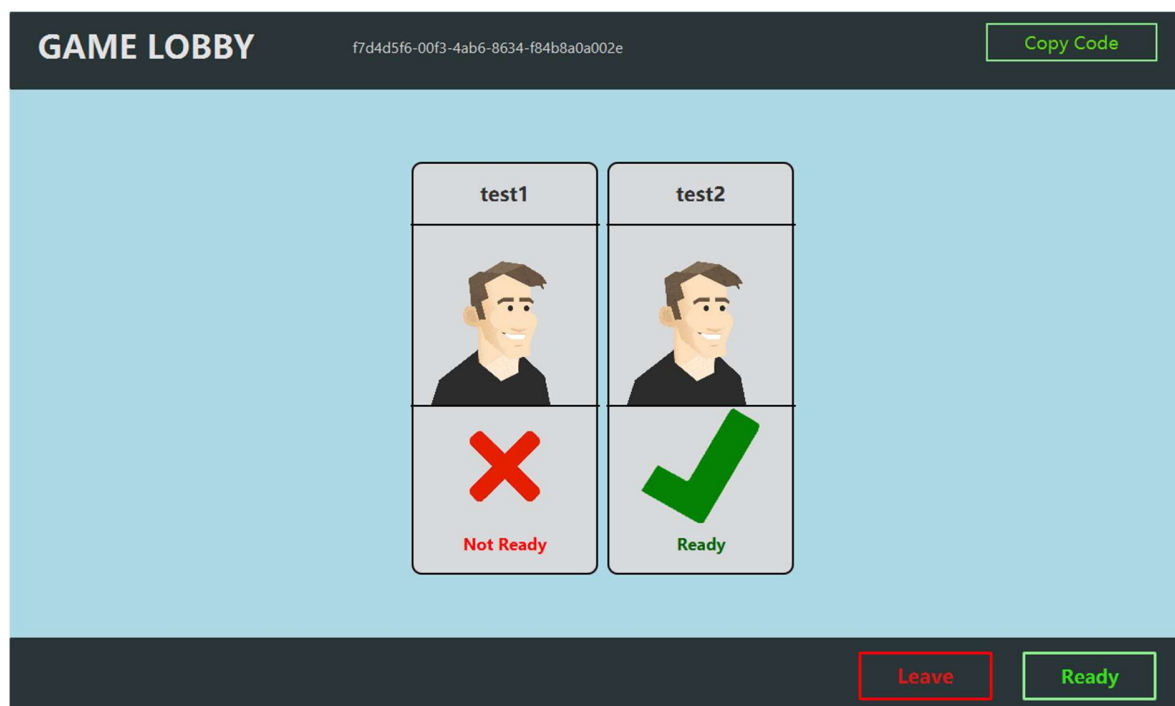
•Przejsięcie do rankingu pokera:

```
@FXML
public void goToPokerRanking() {
    Main.client.sendPacket(new
RankingPacket("Switching to Poker Ranking",
RankingPacket.Status.POKER));
}
```

Metoda ta powoduje wysłanie pakiety *RankingPacket* zawierającego status *POKER*, który powoduje przeniesienie do widoku rankingu pokera.

LOBBY

Poszczególne lobby działają na tej samej zasadzie dla obu gier, jedną wizualną różnicą jest inny motyw kolorystyczny odpowiadający kartom znajdującym się na poprzednim menu, czyli dla pokera czerwony, a dla remika niebieski.





Ilość wyświetlanych graczy i odpowiadających ich awatarów **zwiększa się dynamicznie**, w zależności od liczby osób, które połączyły się z danym lobby.

```
public void refreshPlayerContainer() {
    playersContainer.getChildren().clear();
    for (Map.Entry<Integer, Player> p : this.players.entrySet()) {
        Integer playerId = p.getKey();
        Player player = p.getValue();
        System.out.println("Lc: " + player.getPlayerData());
        try {
            FXMLLoader fxmlLoader = new FXMLLoader();

            fxmlLoader.setLocation(getClass().getResource("/com/example/casino/MiniPlayer.fxml"));
            VBox playerBox = fxmlLoader.load();
            MiniPlayerController miniPlayerController =
            fxmlLoader.getController();
            miniPlayerController.setData(player.getPlayerData(),
            player.isReady());
            playersContainer.getChildren().add(playerBox);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Każdy gracz w lobby ma możliwość zmiany swojego statusu gotowości poprzez dwa przyciski: *Leave* i *Ready*; znajdujące się w prawym dolnym rogu aplikacji.

```
public void changeStatus(boolean isReady, Player player) {
    if (isReady) {
        this.players.get(player.getPlayerID()).setReady(true);
    } else {
        this.players.get(player.getPlayerID()).setReady(false);
    }
}
```




Limit graczy w lobby to pięć osób, natomiast gra rozpoczyna się w momencie, kiedy wszyscy gracze w lobby, ale muszą być połączone co najmniej dwie osoby. Sprawdzane jest to w klasie **ClientHandler**, która wysyła do graczy pakiet **GameReadyPacket**, ze statusem **READY**. Pakiet ten jest wysyłany jako **multicast** do graczy w lobby i jeśli wszyscy gracze wyślą pakiet informujący o tym że są gotowi, rozgrywka rozpoczyna się:

```
if (status.equals(GameReadyPacket.Status.READY)) {  
    System.out.println(++GameServer.pokerGames.get(uuid).playersReady);  
    player.setReady(true);  
    GameServer.pokerGames.get(uuid).broadcast(new GameReadyPacket("ready",  
gameType,  
        player, uuid, GameReadyPacket.Status.READY));  
    if (GameServer.pokerGames.get(uuid).playersReady > 1 &&  
pokerGames.get(uuid).playersReady == pokerGames.get(uuid).players.size()) {  
        GameServer.addNewGameThread(GameServer.pokerGames.get(uuid));  
    }  
} else {  
    GameServer.pokerGames.get(uuid).playersReady--;  
    player.setReady(false);  
    GameServer.pokerGames.get(uuid).broadcast(new GameReadyPacket("notready",  
gameType,  
        player, uuid, GameReadyPacket.Status.NOT_READY));  
}
```



REMIK

Pula kart rozgrywki to dwie talie kart, czyli łącznie 104 kart, które są tasowane wewnątrz jednej dużej talii *cardsInDeck*, która implementowana jest przy użyciu *LinkedList*. Tworzona jest także dodatkowa początkowa pusta *LinkedList* *discardedCards*, wykorzystywana później jako stos kart odrzuconych.

```
LinkedList<RemikCard> cardsInDeck = new LinkedList<>();
LinkedList<RemikCard> discardedCards = new LinkedList<>();

public RemikDeck() {
    discardedCards = new LinkedList<>();
    for (int i = 0; i < 2; i++) { // bo 2 talie
        for (String suit : RemikCard.suits) {
            for (String rank : RemikCard.ranks) {
                RemikCard temp = new RemikCard(suit, rank);
                cardsInDeck.add(temp);
            }
        }
    }
    Collections.shuffle(cardsInDeck);
}
```



Rozgrywka rozpoczyna się od rozdania po 13 kart dla każdego gracza. Karty na ręce gracza są sortowane na względem specjalnej zmiennej *sortValue*, która jest obliczana na podstawie koloru, jak i figury karty:

```
final static String[] suits = {"Hearts", "Diamonds", "Spades", "Clubs"};
final static int[] suitsSortValues = {0, 26, 13, 39};
final static String[] ranks = {"Two", "Three", "Four", "Five", "Six",
"Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King", "Ace"};
final static int[] values = {2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 1};
final static int[] sortValues = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1};
private int sortValue;
private int calculateSortValue(String rank, String suit) {
    int sortValue = 0;
    for (int i = 0; i < ranks.length; i++) {
        if (ranks[i].equals(rank)) {
            sortValue += sortValues[i];
        }
    }

    for (int i = 0; i < suits.length; i++) {
        if (suits[i].equals(suit)) {
            sortValue += suitsSortValues[i];
        }
    }

    return sortValue;
}
```

W ten sposób każda karta ma przypisaną jednoznacznie wartość i nawet kiedy wystąpią powtórki danej karty, to **rozłożenie kart na ręce będzie zawsze takie samo**.



W trakcie działania gry co każdą wykonaną akcję wywoływana jest metoda *routine()*, która:

- Sprawdza czy talia jest pusta, jeśli tak to karty ze stosu kart odrzuconych są dodawane do talii, a następnie przetasowywane:

```
if (deck.isDeckEmpty()) {  
    deck.refillDeckFromDiscardedCards();  
}  
public void refillDeckFromDiscardedCards() {  
    cardsInDeck.addAll(discardedCards);  
    discardedCards.clear();  
    Collections.shuffle(cardsInDeck);  
}
```

- Wyświetlana jest odpowiednia karta na szczycie talii kart odrzuconych:

```
String topCard;  
ImageView topImage = deckTop;  
if (!deck.getDiscardedCards().isEmpty()) {  
    topCard = getImagePath(deck.getDiscardedCards().getLast());  
} else {  
    topCard = "/images/cards/back.png";  
}  
topImage.setImage(new Image(getClass().getResourceAsStream(topCard)));
```

- Poinformowanie gracza o statusie rozgrywki, np. czy ma teraz dobrać czy odrzucić kartę:

```
if (playersTurn) {  
    if (drawTime) {  
        infoDisplay.setText("Dobierz kartę z talii lub stosu kart odrzuconych:");  
    } else {  
        infoDisplay.setText("Odrzuć kartę lub wyłóż karty:");  
    }  
} else {  
    infoDisplay.setText("Oczekiwanie na zagranie innych graczy...");  
}
```

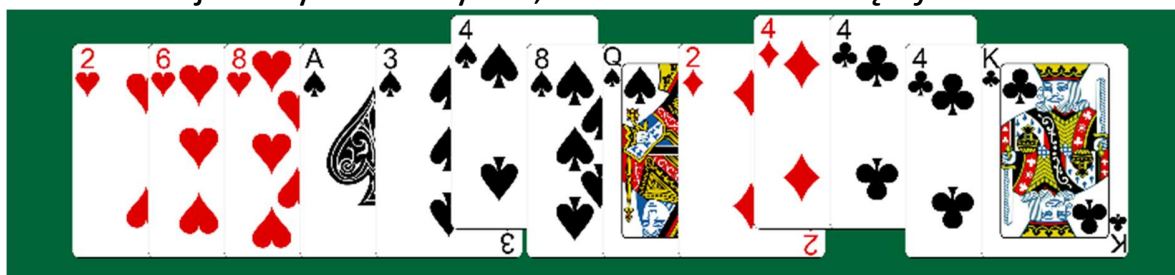


- Wywołanie metody *updateButtonsVisibility()*, która w zależności od etapu rozgrywki pozwala wykonać konkretne akcje poprzez blokowanie bądź odblokowanie odpowiednich przycisków:

```
updateButtonsVisibility();  
private void updateButtonsVisibility() {  
    if (playersTurn) {  
        if (drawTime) {  
            drawCardButton.setDisable(false);  
            takeFromTopButton.setDisable(false);  
            confirmDiscardButton.setDisable(true);  
            tryToLayOffButton.setDisable(true);  
        } else {  
            drawCardButton.setDisable(true);  
            takeFromTopButton.setDisable(true);  
            confirmDiscardButton.setDisable(false);  
            tryToLayOffButton.setDisable(false);  
        }  
    } else {  
        drawCardButton.setDisable(true);  
        takeFromTopButton.setDisable(true);  
        confirmDiscardButton.setDisable(true);  
        tryToLayOffButton.setDisable(true);  
    }  
}
```



Karta, na której ma zostać wykonana akcja po kliknięciu na nią wizualnie przenosi się do góry, w celu zwiększenia widoczności, natomiast w tablicy *boolean* odpowiadającej stanowi czy dana karta jest wybrana czy nie, wartość zmienia się z *false* na *true*:



```
@FXML
private synchronized void pcClicked(MouseEvent event) {
    Object source = event.getSource();
    if (source instanceof ImageView) {
        ImageView clickedCard = (ImageView) source;
        int cardIndex = getCardIndex(clickedCard);
        if (cardIndex != -1) {
            toggleCardSelection(cardIndex, clickedCard);
        }
    }
}

private int getCardIndex(ImageView cardView) {
    if (cardView == pc1) return 1;
    if (cardView == pc2) return 2;
    if (cardView == pc3) return 3;
    if (cardView == pc4) return 4;
    if (cardView == pc5) return 5;
    if (cardView == pc6) return 6;
    if (cardView == pc7) return 7;
    if (cardView == pc8) return 8;
    if (cardView == pc9) return 9;
    if (cardView == pc10) return 10;
    if (cardView == pc11) return 11;
    if (cardView == pc12) return 12;
    if (cardView == pc13) return 13;
    if (cardView == pc14) return 14;
    return -1;
}
```



Tura gracza dzieli się na dwa etapy:

- Dobieranie karty – z talii, bądź szczytu stosu kart odrzuconych:

```
@FXML
private void drawCard(ActionEvent event) {
    RemikCard drawnCard = deck.dealOne();
    player1.addCard(drawnCard);
    player1.sortCardsBySortValue();
    displayCards(player1);
    cardsLeft.setText("Cards left in deck: " + deck.cardsLeftInDeck());

    drawTime = !drawTime;
    routine();
}

@FXML
private void takeFromTop(ActionEvent event) {
    if (!deck.getDiscardedCards().isEmpty()) {
        player1.addCard(deck.discardedCards.removeLast());
        player1.sortCardsBySortValue();
        displayCards(player1);
        cardsLeft.setText("Cards left in deck: " + deck.cardsLeftInDeck());
        drawTime = !drawTime;
        routine();
    }
}
```



- Odrzucenie karty (i dodanie jej na szczyt stosu kart odrzuconych), **opcjonalnie poprzedzone wyłożeniem sekwensu, lub dołożeniem kart do istniejącego sekwensu na stole** (dokładnie opisane w dalszej części sprawozdania):

```
private void discardSelectedCard() {  
    for (int i = 0; i < pcChosenArray.length; i++) {  
        if (pcChosenArray[i]) {  
            removeCardFromHand(i + 1, true);  
            break;  
        }  
    }  
}  
  
@FXML  
private void confirmDiscard(ActionEvent event) {  
    if (onlyOneCardUp()) {  
        discardSelectedCard();  
        drawTime = !drawTime;  
        routine();  
    }  
}
```




Gracz może wyłożyć karty na stół układając je w jednym z dwóch możliwych wariantów sekwensów, lecz każdy sekwens musi się składać co najmniej z 3 kart:

- Wszystkie karty tego samego koloru, lecz ich figury występują po sobie np.: 3,4,5 lub 10, walet, dama:

```
private boolean isSameSuitWithIncrementingValues(ArrayList<RemikCard> cards)
{
    cards.sort(Comparator.comparingInt(RemikCard::getSortValue));
    String suit = cards.get(0).getSuit();
    for (int i = 1; i < cards.size(); i++) {
        if (!cards.get(i).getSuit().equals(suit) ||
cards.get(i).getSortValue() != cards.get(i - 1).getSortValue() + 1) {
            return false;
        }
    }
    return true;
}
```

- Figury wszystkich kart są figurami tego samego rodzaju, ale każda jest innego koloru.

```
private boolean isDifferentSuitsWithSameValue(ArrayList<RemikCard> cards) {
    int value = cards.get(0).getValue();
    HashSet<String> suits = new HashSet<>();
    for (RemikCard card : cards) {
        if (card.getValue() != value || !suits.add(card.getSuit())) {
            return false;
        }
    }
    return true;
}
```



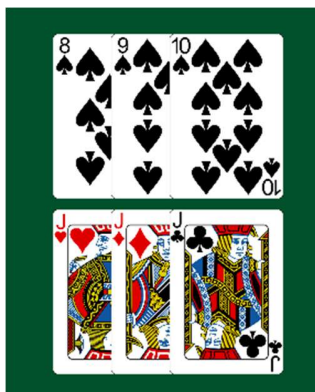
Politechnika Krakowska
im. Tadeusza Kościuszki
ul. J. J. 1
31-064 Kraków

Programowanie w języku Java - semestr IV



Wydział Inżynierii
Elektrycznej i Komputerowej

Elektrycznej i Komputerowej
Wydział Inżynierii



```
@FXML
public void tryToLayOff(ActionEvent event) {

    ArrayList<RemikCard> chosenCards = new ArrayList<>();

    // dodaj wszystkie wybrane karty do listy "chosenCards"
    for (int i = 0; i < pcChosenArray.length; i++) {
        if (pcChosenArray[i]) {
            if (i < player1.getCardsOnHand().size()) {
                chosenCards.add(player1.getCardsOnHand().get(i));
            }
        }
    }

    // sprawdzenie warunków: czy co najmniej 3 karty i jeden z 2 rodzajow
    // sekwensow
    if (chosenCards.size() >= 3 &&
        (isSameSuitWithIncrementingValues(chosenCards) ||
         isDifferentSuitsWithSameValue(chosenCards))) {
        // Jeśli pasuje, usuń wybrane karty
        for (int i = chosenCards.size() - 1; i >= 0; i--) {
            RemikCard card = chosenCards.get(i);
            int cardIndex = player1.getCardsOnHand().indexOf(card);
            if (cardIndex >= 0) {
                removeCardFromHand(cardIndex + 1, false);
            }
        }
        // podział kart na rzędy
        ArrayList<ArrayList<RemikCard>> rowsOfCards =
splitCardsIntoRows(chosenCards);

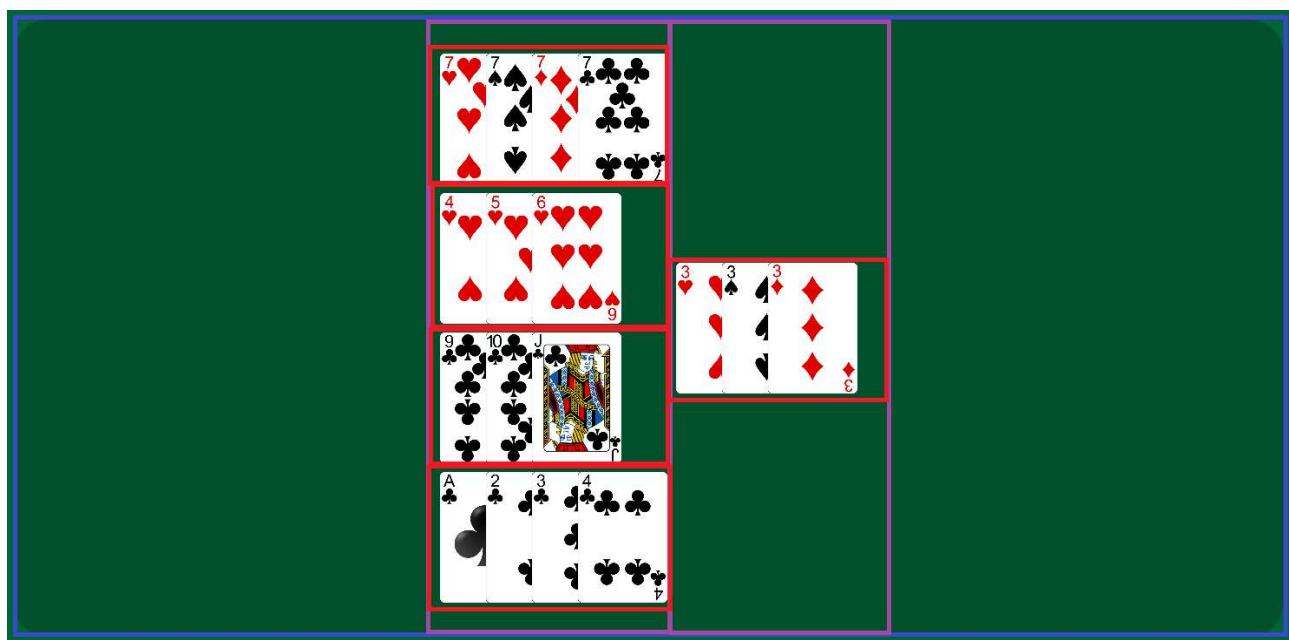
        // wyświetlanie karty na middleBoard
        for (ArrayList<RemikCard> row : rowsOfCards) {
            displayRowOfCards(row);
        }
        routine();
    } else {
        System.out.println("Nie");
    }
}
```



Metoda sprawdzająca czy wybrane przez gracza spełniają wszystkie warunki niezbędne do wyłożenia danych kart na stół. **Jeśli warunki zostały spełnione usuwa je z ręki gracza (ale nie dodaje do stosu kart odrzuconych) i dodaje na środek stołu.**

Karty na środku stołu są dodawane dynamicznie na bazie *HBox* i *VBox*:

- Cały stół to jeden duży *HBox* (zaznaczony na niebiesko na obrazku).
- W zależności od ilości wyłożonych sekwensów tworzone są *VBox*'y, gdzie każdy mieści maksymalnie 4 sekwensy (zaznaczone na różowo na obrazku).
- Każdy sekwens jest osobnym *HBox*'em zawarty w odpowiednim *VBox* (zaznaczone na czerwono na obrazku).





Metoda układająca karty w opisany powyżej sposób:

```
private void displayRowOfCards(ArrayList<RemikCard> row) {  
    // Sortowanie kart w rzędzie po sortValue  
    row.sort(Comparator.comparingInt(RemikCard::getSortValue));  
    HBox cardRow = new HBox();  
    cardRow.setSpacing(-50); // odstęp między kartami  
  
    for (RemikCard card : row) {  
        String imagePath = getImagePath(card);  
        ImageView temp = new ImageView();  
        temp.setFitHeight(150);  
        temp.setFitWidth(103);  
        temp.setImage(new Image(getClass().getResourceAsStream(imagePath)));  
        temp.setUserData(card);  
  
        // mechanika klikania na kartę na środku  
        temp.setOnMouseClicked(event -> {  
            displayCardsInHBox(cardRow);  
            tryToAddCardToHBox(cardRow);  
        });  
        cardRow.getChildren().add(temp);  
    }  
    addRowToMiddleBoard(cardRow);  
}
```

Metoda zarządzająca VBox'ami i dodająca nowe sekwensy – zawarte w HBox'ach do odpowiednich VBox:

```
private void addRowToMiddleBoard(HBox cardRow) {  
    if (middleBoard.getChildren().isEmpty() ||  
        !(middleBoard.getChildren().get(middleBoard.getChildren().size() - 1)  
            instanceof VBox)) {  
        VBox vbox = new VBox();  
        vbox.setSpacing(10); // odstęp między vboxami  
        vbox.setAlignment(Pos.CENTER); // wysrodkowanie  
        middleBoard.setSpacing(10); // odstęp między vboxami  
        middleBoard.getChildren().add(vbox);  
    }  
    VBox currentVBox = (VBox)  
middleBoard.getChildren().get(middleBoard.getChildren().size() - 1);  
  
    if (currentVBox.getChildren().size() == 4) {  
        VBox newVBox = new VBox();  
        newVBox.setSpacing(10);  
        newVBox.setAlignment(Pos.CENTER);  
        middleBoard.getChildren().add(newVBox);  
        currentVBox = newVBox;  
    }  
    currentVBox.getChildren().add(cardRow);  
}
```



Gracz poza wykładaniem zebranych kart w sekwensy, może dokładać pojedyncze karty do istniejących już sekwensów, lecz **może to jedynie zrobić jeżeli już istniejący sekwens na stole po dodaniu nowej karty dalej spełnia jeden z dwóch wymaganych warunków:**

```
private void tryToAddCardToHBox(HBox cardRow) {  
    if (!drawTime) {  
        ArrayList<RemikCard> cardsInRow = new ArrayList<>();  
        for (Node node : cardRow.getChildren()) {  
            if (node instanceof ImageView) {  
                RemikCard card = getCardFromImageView((ImageView) node);  
                if (card != null) {  
                    cardsInRow.add(card);  
                }  
            }  
        }  
  
        // czy jedna karta wybrana  
        int selectedCardIndex = -1;  
        for (int i = 0; i < pcChosenArray.length; i++) {  
            if (pcChosenArray[i] && i < player1.getCardsOnHand().size()) {  
                selectedCardIndex = i;  
                break;  
            }  
        }  
        // ...  
    }  
}
```



```
//...
    if (selectedCardIndex != -1) {
        RemikCard selectedCard =
player1.getCardsOnHand().get(selectedCardIndex);

        // sprawdzenie czy po dodaniu warnki bylyby dalej spelnione
        cardsInRow.add(selectedCard);
        if (isSameSuitWithIncrementingValues(cardsInRow) ||
isDifferentSuitsWithSameValue(cardsInRow)) {
            cardsInRow.remove(selectedCard); //

            // usuwa karte z reki
            removeCardFromHand(selectedCardIndex + 1, false);
            pcChosenArray[selectedCardIndex] = false;

            // dodaj karte do middleBoard
            String imagePath = getImagePath(selectedCard);
            ImageView temp = new ImageView();
            temp.setFitHeight(150);
            temp.setFitWidth(103);
            temp.setImage(new
Image(getClass().getResourceAsStream(imagePath)));

            temp.setUserData(selectedCard);
            temp.setOnMouseClicked(event -> {
                displayCardsInHBox(cardRow);
                tryToAddCardToHBox(cardRow);
            });

            cardRow.getChildren().add(temp);

            // sortowanie kart w poszczegolnych hboxach
            sortCardsInHBox(cardRow);

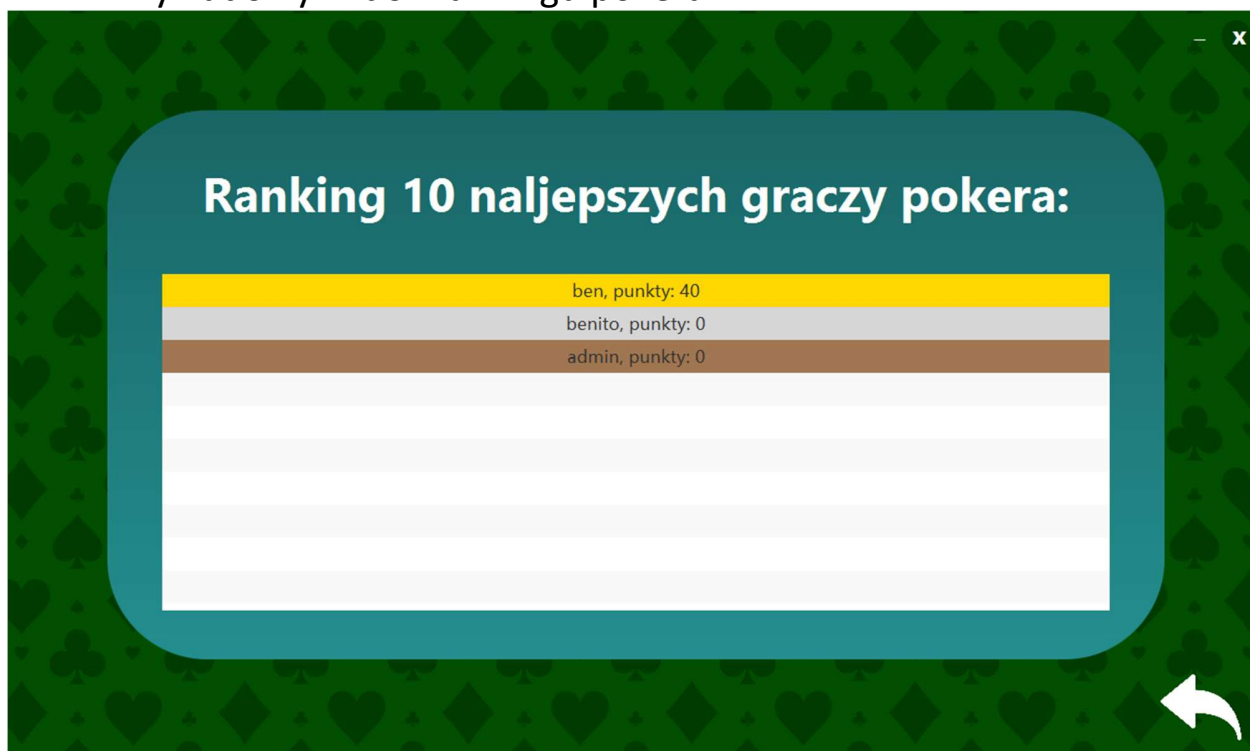
            System.out.println("Dodano karte: " + selectedCard);
        } else {
            cardsInRow.remove(selectedCard);
            System.out.println("Nie można dodać karty: " + selectedCard);
        }
    }
}
```



RANKINGI

Każda z gier ma osobny ranking zapisany w bazie danych w odpowiadającej danej grze tabeli. W specjalnym panelu rankingu dla każdej gry możemy sprawdzić ranking aktualnie dziesięciu najlepszych graczy.

Przykładowy widok rankingu pokera:



ben, punkty: 40
benito, punkty: 0
admin, punkty: 0



W momencie kliknięcia ikonki pucharaka, odpowiadającej rankingowi wysyłane jest zapytanie do bazy danych, które zwraca dziesięciu najlepszych graczy z danej gry:

```
Statement st= connection.createStatement();
ResultSet rs = st.executeQuery("SELECT users.username, pokerRanking.Points
FROM pokerRanking JOIN users USING(UserID) ORDER BY pokerRanking.Points DESC
LIMIT 10");
```

Otrzymany wynik jest zapisywany do *HashMap*'y zawierającej *String* oraz *Integer*, czyli odpowiednio nazwa użytkownika oraz jego punkty rankingowe danego gracza:

```
HashMap<String,Integer> remikRankingMap = new HashMap<>();

try {
    Statement st= connection.createStatement();
    ResultSet rs = st.executeQuery("SELECT users.username,
rummyRanking.Points FROM rummyRanking JOIN users USING(UserID) ORDER BY
rummyRanking.Points DESC LIMIT 10"); //do zmiany na remik

    while (rs.next()) {
        String username = rs.getString("Username");
        int points = rs.getInt("Points");
        remikRankingMap.put(username, points);
    }
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

Następnie po dodaniu wyników do *HashMap*'y wysyłany jest do klienta pakiet *RankingPacket* zawierający identyfikator statusu danej gry, w tym przypadku *REMIK*, oraz *HashMap*ę.

```
sendPacket(new RankingPacket("Switching to Remik Ranking",
RankingPacket.Status.REMIK, remikRankingMap));
```




Klient po odebraniu tego pakietu przełącza się na scenę z rankingiem np. *RemikRanking.fxml* i inicjalizuje *ListView* z pobranymi danymi:

```
public void initTop10List() {
    top10remik.getItems().clear();
    ArrayList<Map.Entry<String, Integer>> entryList = new
ArrayList<>(remikRankingMap.entrySet());
    entryList.sort((entry1, entry2) ->
entry2.getValue().compareTo(entry1.getValue()));

    for (Map.Entry<String, Integer> entry : entryList) {
        String entryString = entry.getKey() + ", punkty: " +
entry.getValue();
        top10remik.getItems().add(entryString);
    }

    top10remik.setCellFactory(lv -> {
        return new ListCell<String>() {
            @Override
            protected void updateItem(String item, boolean empty) {
                super.updateItem(item, empty);
                setText(item);
                if (getIndex() == 0) {
                    setId("first-cell");
                } else if (getIndex() == 1) {
                    setId("second-cell");
                } else if (getIndex() == 2) {
                    setId("third-cell");
                } else if (getIndex() % 2 == 0) {
                    setId("even-cell");
                } else if (getIndex() % 2 == 1) {
                    setId("odd-cell");
                } else {
                    setId(null);
                }
            }
        };
    });
}
```



BAZA DANYCH

Projekt oparty jest o darmową, zdalną bazę danych MySQL, dostarczaną przez Filess.io. Połączenie z nią nawiązywane jest przez sterownik **JDBC**. Serwer tworzy jedno, statyczne połączenie, z którego później różne klasy korzystają w różnych miejscach.

```
public static String connectionUrl = "jdbc:mysql://i3m.h.filess.io:3307/JavaProject_fourthtalk";  
11 usages  
public static Connection connection;  
  
static {  
    try {  
        connection = DriverManager.getConnection(GameServer.connectionUrl,  
            user: "JavaProject_fourthtalk", password: "26c741dadf126863995c714674f8a4c681c4dfb3");  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    }  
}
```

Baza zawiera trzy tabele – users, pokerRanking i rummyRanking, pierwsza służy do obsługi logowania i rejestracji, a pozostałe przechowują rankingi graczy w poszczególnych grach.



PODSUMOWANIE

Efektywność Architektury Klient-Serwer:

Architektura oparta na socketach okazała się skuteczna do implementacji gier wieloosobowych. Zapewniała ona prostą, lecz efektywną komunikację między klientem a serwerem, co jest kluczowe w przypadku aplikacji wymagających niskich opóźnień.

Wyzwania związane z Synchronizacją:

Zastosowanie mechanizmów synchronizacji stanu gry na serwerze było kluczowe. Implementacja odpowiednich algorytmów synchronizujących rozgrywkę w czasie rzeczywistym stanowiła jedno z większych wyzwań, wymagając dokładnego przetestowania i optymalizacji.

Stabilność i Bezpieczeństwo:

W trakcie realizacji projektu zauważyliśmy, jak ważne jest zapewnienie stabilności i bezpieczeństwa połączeń. Użycie odpowiednich mechanizmów autoryzacji oraz zarządzanie połączeniami klienckimi zwiększyło bezpieczeństwo i niezawodność systemu.