

**Wydział Inżynierii
Elektrycznej i Komputerowej**

POLITECHNIKA KRAKOWSKA

**WYDZIAŁ INŻYNIERII ELEKTRYCZNEJ I
KOMPUTEROWEJ**

**PROJEKT Z PRZEDMIOTU "PROGRAMOWANIE W
JĘZYKU JAVA"**

PROJEKT "MONOPOLY"

Prowadzący przedmiot:

dr inż. S Bąk

Wykonał:

Kamil Jagielski
Mateusz Jankowski

1. Wstęp.

Poniższy raport poświęcony będzie omówieniu funkcjonalności oraz implementacji kodu projektu "Monopoly".

Lista narzędzi oraz środowisk wykorzystanych do realizacji projektu:

- IntelliJ Idea
- Github

W ramach zajęć projektowych zrealizowaliśmy następujące kroki:

- Komunikacja sieciowa
- Wielowątkowość
- Połączenie z bazą danych
- Rozgrywka sieciowa

Celem naszego projektu było odwzorowanie kultowej gry "Monopoly" stosując umiejętności pozyskane na laboratoriach z przedmiotu "Programowanie w Języku Java".

2. Opis projektu.

"Monopoly" to popularna gra planszowa, której celem jest osiągnięcie finansowej dominacji poprzez kupowanie, wynajmowanie i handel nieruchomościami, prowadząc innych graczy do bankructwa. Gra została stworzona na początku XX wieku, a jej pierwowzorem była gra "The Landlord's Game" wynaleziona przez Elizabeth Magie w 1903 roku. Charles Darrow, który jest często przypisywany za stworzenie "Monopoly", opatentował ją w 1935 roku i sprzedał prawa firmie Parker Brothers.

Rozgrywka odbywa się na planszy złożonej z pól przedstawiających różne nieruchomości, które gracze mogą kupować za wirtualne pieniądze. Na planszy znajdują się również pola z szansami losowymi oraz pola specjalne, takie jak więzienie, parking czy podatek. Gracze poruszają się po planszy zgodnie z wynikiem rzutów kostką. W trakcie gry budują domy i hotele na

swoich nieruchomościach, co zwiększa czynsz, który muszą płacić inni gracze lądujący na tych polach.

"Monopoly" jest znane z możliwości prowadzenia negocjacji i zawierania umów między graczami, co dodaje grze strategicznej głębi. Gra jest krytykowana za to, że może trwać bardzo długo i za to, że często wynik rozgrywki może być zdeterminowany przez losowe rzuty kostką.

W grze "Monopoly" plansza składa się z różnych typów pól, z których każde pełni unikalną funkcję i wpływa na strategię gry. Oto omówienie tych pól:

Nieruchomości: To pola, które gracze mogą kupić, budować na nich domy i hotele, oraz pobierać czynsz od innych graczy, którzy na nich lądują. Nieruchomości są zazwyczaj podzielone na grupy kolorystyczne, a posiadanie wszystkich nieruchomości w jednej grupie pozwala na budowę na nich i zwiększa wartość czynszu.

Dworce: Są cztery dworce na planszy. Gracz może kupić dworzec i pobierać czynsz od innych graczy. Czynsz wzrasta z każdym kolejnym posiadanym dworcem.

Zakłady użyteczności publicznej: Na planszy znajdują się dwa zakłady użyteczności publicznej – wodociągi i elektrownia. Gracze, którzy posiadają te pola, mogą pobierać czynsz, który jest obliczany na podstawie rzutu kostką.

Karty Szansa i Kasa Społeczna: To pola, na których gracze losują karty ze specjalnymi instrukcjami, które mogą wpłynąć na ich finanse, ruch lub inne aspekty gry. Karty mogą oferować nagrody pieniężne, mandaty, ruch na inne pole planszy lub inne niespodzianki.

Podatek dochodowy i podatek luksusowy: Lądowanie na tych polach powoduje konieczność zapłacenia określonej kwoty do banku.

Więzienie: To pole, na którym gracz może trafić z kart Szansa lub Kasa Społeczna, z pola "Idź do więzienia" lub wyrzucając odpowiednią kombinację kostek. Gracz w więzieniu musi wykonać określone czynności, aby wyjść na wolność, takie jak zapłacenie kaucji, użycie karty "Wyjdź z więzienia" lub wyrzucenie odpowiedniej kombinacji kostek.

Idź do więzienia: Pole to zmusza gracza do przeniesienia się bezpośrednio do więzienia, bez przechodzenia przez pole "Start" i bez odbierania wynagrodzenia.

Darmowy parking: To neutralne pole, na którym gracz nie podejmuje żadnych działań. W niektórych domowych wersjach gry pole to może mieć dodatkowe zasady, takie jak zbieranie wszystkich pieniędzy z podatków.

Start: Każdy gracz przechodzący przez to pole otrzymuje ustaloną kwotę pieniędzy (zwykle 200 dolarów), co jest kluczowym elementem strategii, szczególnie na początku gry.

W naszym projekcie używamy oryginalnej planszy "Monopoly" lecz zrezygnowaliśmy z możliwości handlu, w celu uproszczenia rozgrywki. Grać naraz może 4 graczy.

3. Kod projektu.

Klasa GameManager:

```
package Server;

import BoardSpaces.*;
import Utilities.Player;
import javafx.util.Pair;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

public class GameManager {

    /*
    start game ->
    notify listeners that game started ->
    start new GameManager for chosen lobby ->
    create 'game scene' for all clients in the lobby that gets data from thats lobbys GameManager
    */

    private static int gameId = 0;

    private HashMap<String, Player> players;

    private ArrayList<Board> boardSpaces;

    private ArrayList<StreetSet> streetSets;

    private String currentPlayerLogin;

    private Player currentPlayer;

    public GameManager(ArrayList<String> usersLogins) {

        this.players = new HashMap<>();

        this.boardSpaces = new ArrayList<>();

        this.streetSets = new ArrayList<>();

        for(String login : usersLogins){

            players.put(login, new Player(usersLogins.size()));

        }

        if (players.size() < 2 || players.size() > 4) throw new IllegalArgumentException("Invalid number of players");

        //random number from 0 to players.size() - 1

        int randomPlayer = (int) (Math.random() * players.size());
```

```

        currentPlayerLogin = usersLogins.get(randomPlayer);

        currentPlayer = players.get(currentPlayerLogin);

        gameId++;
    }

    public boolean nextTurn() {

        if (!currentPlayer.hasRolled()) return false;

        ArrayList<String> playerLogins = new ArrayList<>(players.keySet());

        int currentIndex = playerLogins.indexOf(currentPlayerLogin);

        int nextIndex = (currentIndex + 1) % playerLogins.size();

        currentPlayerLogin = playerLogins.get(nextIndex);

        currentPlayer = players.get(currentPlayerLogin);

        System.out.println("Current player: " + currentPlayerLogin);

        return true;
    }

    public Pair<String, Integer> handleSpaceEvent(int currentSpaceID, Player player) {

        if (boardSpaces.get(player.getCurrentSpace()).getType() == SpaceType.STREET ||

            boardSpaces.get(player.getCurrentSpace()).getType() == SpaceType.RAILROAD ||

            boardSpaces.get(player.getCurrentSpace()).getType() == SpaceType.UTILITY) {

            return handlePropertyEvent(currentSpaceID, player);

        } else {

            return handleEvent(currentSpaceID, player);

        }

    }

    public Pair<String, Integer> handlePropertyEvent(int currentSpaceID, Player player) {

        if (boardSpaces.get(currentSpaceID).getType() == SpaceType.STREET) {

            BoardSpaceStreet street = (BoardSpaceStreet) boardSpaces.get(currentSpaceID);

            if (street.getOwner() == null) {

                if (player.getMoney() >= street.getBuyoutCost()) {

                    player.buyProperty(currentSpaceID, street.getBuyoutCost());

                    street.setOwner(player);

                    return new Pair<>("buyStreet", street.getBuyoutCost());

                } else {

                    return new Pair<>("noBuyStreet", 0);

                }

            } else if (street.getOwner() != player) {

                int rent = street.getRent();

                player.addMoney(-rent);

                street.getOwner().addMoney(rent);

                if (player.getMoney() < 0) {

                    return new Pair<>("bankrupt", 0);

                }

                return new Pair<>("payRent", rent);

            } else {

                return new Pair<>("ownProperty", 0);

            }

        }

    }

```

```

} else if (boardSpaces.get(currentSpaceID).getType() == SpaceType.RAILROAD) {

    BoardSpaceRailroad railroad = (BoardSpaceRailroad) boardSpaces.get(currentSpaceID);

    if (railroad.getOwner() == null) {

        if (player.getMoney() >= railroad.getBuyoutCost()) {

            player.buyProperty(currentSpaceID, railroad.getBuyoutCost());

            railroad.setOwner(player);

            return new Pair<>("buyRailroad", railroad.getBuyoutCost());

        } else {

            return new Pair<>("noBuyRailroad", 0);

        }

    } else if (railroad.getOwner() != player) {

        int rent = railroad.getRent();

        player.addMoney(-rent);

        railroad.getOwner().addMoney(rent);

        if (player.getMoney() < 0) {

            return new Pair<>("bankrupt", 0);

        }

        return new Pair<>("payRent", rent);

    } else {

        return new Pair<>("ownProperty", 0);

    }

} else {

    BoardSpaceUtility utility = (BoardSpaceUtility) boardSpaces.get(currentSpaceID);

    if (utility.getOwner() == null) {

        if (player.getMoney() >= utility.getBuyoutCost()) {

            player.buyProperty(currentSpaceID, utility.getBuyoutCost());

            utility.setOwner(player);

            return new Pair<>("buyUtility", utility.getBuyoutCost());

        } else {

            return new Pair<>("noBuyUtility", 0);

        }

    } else if (utility.getOwner() != player) {

        int rent = utility.getRent(player.getCurrentRoll(), 0);

        player.addMoney(-rent);

        utility.getOwner().addMoney(rent);

        if (player.getMoney() < 0) {

            return new Pair<>("bankrupt", 0);

        }

        return new Pair<>("payRent", rent);

    } else {

        return new Pair<>("ownProperty", 0);

    }

}

}

}

public Pair<String, Integer> handleEvent(int currentSpaceID, Player player) {

```

```

if (boardSpaces.get(currentSpaceID).getType() == SpaceType.EVENT) {

    BoardSpaceEvent event = (BoardSpaceEvent) boardSpaces.get(currentSpaceID);

    switch (event.getSpaceEventType()) {

        case CHEST:

            return new Pair<>("chest", 0);

        case INCOME_TAX:

            player.addMoney(-60);

            return new Pair<>("incomeTax", 60);

        case CHANCE:

            return new Pair<>("chance", 0);

        case GO_TO_JAIL:

            player.goToJail();

            return new Pair<>("goToJail", 0);

        case LUXURY_TAX:

            player.addMoney(-100);

            return new Pair<>("luxuryTax", 100);

        default:

            return new Pair<>("error", 0);

    }

} else {

    return new Pair<>("error", 0);

}

}

public Player getCurrentPlayer() {

    return players.get(currentPlayerLogin);

}

public String getCurrentPlayerLogin() {

    return currentPlayerLogin;

}

public HashMap<String, Player> getPlayers() {

    return players;

}

private void populateStreets() {

    streetSets.add(new StreetSet("Brown", 2, 50));

    streetSets.add(new StreetSet("Light Blue", 3, 50));

    streetSets.add(new StreetSet("Purple", 3, 100));

    streetSets.add(new StreetSet("Orange", 3, 100));

    streetSets.add(new StreetSet("Red", 3, 150));

    streetSets.add(new StreetSet("Yellow", 3, 150));

    streetSets.add(new StreetSet("Green", 3, 200));

    streetSets.add(new StreetSet("Dark Blue", 2, 200));

}

private void populateSpaces() {

    // Space 0

    boardSpaces.add(new BoardSpaceEvent(

```

```

        new BoardSpace(1, "Go", SpaceType.EVENT),

        SpaceEventType.GO
    ));

    // Space 1
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(2, "Mediterranean Avenue", SpaceType.STREET),
        streetSets.get(0),
        60,
        streetSets.get(0).getHouseCost(),
        new ArrayList<>(Arrays.asList(2, 10, 30, 90, 160, 250))
    ));

    // Space 2
    boardSpaces.add(new BoardSpaceEvent(
        new BoardSpace(3, "Community Chest", SpaceType.EVENT),
        SpaceEventType.CHEST
    ));

    // Space 3
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(4, "Baltic Avenue", SpaceType.STREET),
        streetSets.get(0),
        60,
        streetSets.get(0).getHouseCost(),
        new ArrayList<>(Arrays.asList(4, 20, 60, 180, 320, 450))
    ));

    // Space 4
    boardSpaces.add(new BoardSpaceEvent(
        new BoardSpace(5, "Income Tax", SpaceType.EVENT),
        SpaceEventType.INCOME_TAX
    ));

    // Space 5
    boardSpaces.add(new BoardSpaceRailroad(
        new BoardSpace(6, "Reading Railroad", SpaceType.RAILROAD),
        200,
        new ArrayList<>(Arrays.asList(25, 50, 100, 200))
    ));

    // Space 6
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(7, "Oriental Avenue", SpaceType.STREET),
        streetSets.get(1),
        100,
        streetSets.get(1).getHouseCost(),
        new ArrayList<>(Arrays.asList(6, 30, 90, 270, 400, 550))
    ));

    // Space 7
    boardSpaces.add(new BoardSpaceEvent(
        new BoardSpace(8, "Chance", SpaceType.EVENT),

```



```
        SpaceEventType.CHANCE
    ));

    // Space 8
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(9, "Vermont Avenue", SpaceType.STREET),
        streetSets.get(1),
        100,
        streetSets.get(1).getHouseCost(),
        new ArrayList<>(Arrays.asList(6, 30, 90, 270, 400, 550))
    ));

    // Space 9
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(10, "Connecticut Avenue", SpaceType.STREET),
        streetSets.get(1),
        120,
        streetSets.get(1).getHouseCost(),
        new ArrayList<>(Arrays.asList(8, 40, 100, 300, 450, 600))
    ));

    // Space 10
    boardSpaces.add(new BoardSpaceEvent(
        new BoardSpace(11, "Jail", SpaceType.EVENT),
        SpaceEventType.JAIL
    ));

    // Space 11
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(12, "St. Charles Place", SpaceType.STREET),
        streetSets.get(2),
        140,
        streetSets.get(2).getHouseCost(),
        new ArrayList<>(Arrays.asList(10, 50, 150, 450, 625, 750))
    ));

    // Space 12
    boardSpaces.add(new BoardSpaceUtility(
        new BoardSpace(13, "Electric Company", SpaceType.UTILITY),
        150,
        new ArrayList<>(Arrays.asList(4, 10))
    ));

    // Space 13
    boardSpaces.add(new BoardSpaceStreet(
        new BoardSpace(14, "States Avenue", SpaceType.STREET),
        streetSets.get(2),
        140,
        streetSets.get(2).getHouseCost(),
        new ArrayList<>(Arrays.asList(10, 50, 150, 450, 625, 750))
    ));

    // Space 14
```

```

boardSpaces.add(new BoardSpaceStreet(

    new BoardSpace(15, "Virginia Avenue", SpaceType.STREET),

    streetSets.get(2),

    160,

    streetSets.get(2).getHouseCost(),

    new ArrayList<>(Arrays.asList(12, 60, 180, 500, 700, 900))

));

// Space 15

boardSpaces.add(new BoardSpaceRailroad(

    new BoardSpace(16, "Pennsylvania Railroad", SpaceType.RAILROAD),

    200,

    new ArrayList<>(Arrays.asList(25, 50, 100, 200))

));

// Space 16

boardSpaces.add(new BoardSpaceStreet(

    new BoardSpace(17, "St. James Place", SpaceType.STREET),

    streetSets.get(3),

    180,

    streetSets.get(3).getHouseCost(),

    new ArrayList<>(Arrays.asList(14, 70, 200, 550, 750, 950))

));

// Space 17

boardSpaces.add(new BoardSpaceEvent(

    new BoardSpace(18, "Community Chest", SpaceType.EVENT),

    SpaceEventType.CHEST

));

// Space 18

boardSpaces.add(new BoardSpaceStreet(

    new BoardSpace(19, "Tennessee Avenue", SpaceType.STREET),

    streetSets.get(3),

    180,

    streetSets.get(3).getHouseCost(),

    new ArrayList<>(Arrays.asList(14, 70, 200, 550, 750, 950))

));

// Space 19

boardSpaces.add(new BoardSpaceStreet(

    new BoardSpace(20, "New York Avenue", SpaceType.STREET),

    streetSets.get(3),

    200,

    streetSets.get(3).getHouseCost(),

    new ArrayList<>(Arrays.asList(16, 80, 220, 600, 800, 1000))

));

// Space 20

boardSpaces.add(new BoardSpaceEvent(

    new BoardSpace(21, "Free Parking", SpaceType.EVENT),

    SpaceEventType.FREE_PARKING

```

```

));

// Space 21
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(22, "Kentucky Avenue", SpaceType.STREET),
    streetSets.get(4),
    220,
    streetSets.get(4).getHouseCost(),
    new ArrayList<>(Arrays.asList(18, 90, 250, 700, 875, 1050))
));

// Space 22
boardSpaces.add(new BoardSpaceEvent(
    new BoardSpace(23, "Chance", SpaceType.EVENT),
    SpaceEventType.CHANCE
));

// Space 23
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(24, "Indiana Avenue", SpaceType.STREET),
    streetSets.get(4),
    220,
    streetSets.get(4).getHouseCost(),
    new ArrayList<>(Arrays.asList(18, 90, 250, 700, 875, 1050))
));

// Space 24
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(25, "Illinois Avenue", SpaceType.STREET),
    streetSets.get(4),
    240,
    streetSets.get(4).getHouseCost(),
    new ArrayList<>(Arrays.asList(20, 100, 300, 750, 925, 1100))
));

// Space 25
boardSpaces.add(new BoardSpaceRailroad(
    new BoardSpace(26, "B. & O. Railroad", SpaceType.RAILROAD),
    200,
    new ArrayList<>(Arrays.asList(25, 50, 100, 200))
));

// Space 26
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(27, "Atlantic Avenue", SpaceType.STREET),
    streetSets.get(5),
    260,
    streetSets.get(5).getHouseCost(),
    new ArrayList<>(Arrays.asList(22, 110, 330, 800, 975, 1150))
));

// Space 27
boardSpaces.add(new BoardSpaceStreet(

```

```
        new BoardSpace(28, "Ventnor Avenue", SpaceType.STREET),

        streetSets.get(5),

        260,

        streetSets.get(5).getHouseCost(),

        new ArrayList<>(Arrays.asList(22, 110, 330, 800, 975, 1150))

    ));

    // Space 28
    boardSpaces.add(new BoardSpaceUtility(

        new BoardSpace(29, "Water Works", SpaceType.UTILITY),

        150,

        new ArrayList<>(Arrays.asList(4, 10))

    ));

    // Space 29
    boardSpaces.add(new BoardSpaceStreet(

        new BoardSpace(30, "Marvin Gardens", SpaceType.STREET),

        streetSets.get(5),

        280,

        streetSets.get(5).getHouseCost(),

        new ArrayList<>(Arrays.asList(24, 120, 360, 850, 1025, 1200))

    ));

    // Space 30
    boardSpaces.add(new BoardSpaceEvent(

        new BoardSpace(31, "Go to Jail", SpaceType.EVENT),

        SpaceEventType.GO_TO_JAIL

    ));

    // Space 31
    boardSpaces.add(new BoardSpaceStreet(

        new BoardSpace(32, "Pacific Avenue", SpaceType.STREET),

        streetSets.get(6),

        300,

        streetSets.get(6).getHouseCost(),

        new ArrayList<>(Arrays.asList(26, 130, 390, 900, 1100, 1275))

    ));

    // Space 32
    boardSpaces.add(new BoardSpaceStreet(

        new BoardSpace(33, "North Carolina Avenue", SpaceType.STREET),

        streetSets.get(6),

        300,

        streetSets.get(6).getHouseCost(),

        new ArrayList<>(Arrays.asList(26, 130, 390, 900, 1100, 1275))

    ));

    // Space 33
    boardSpaces.add(new BoardSpaceEvent(

        new BoardSpace(34, "Community Chest", SpaceType.EVENT),

        SpaceEventType.CHEST

    ));
```

```

// Space 34
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(35, "Pennsylvania Avenue", SpaceType.STREET),
    streetSets.get(6),
    320,
    streetSets.get(6).getHouseCost(),
    new ArrayList<>(Arrays.asList(28, 150, 450, 1000, 1200, 1400))
));

// Space 35
boardSpaces.add(new BoardSpaceRailroad(
    new BoardSpace(36, "Short Line", SpaceType.RAILROAD),
    200,
    new ArrayList<>(Arrays.asList(25, 50, 100, 200))
));

// Space 36
boardSpaces.add(new BoardSpaceEvent(
    new BoardSpace(37, "Luxury Tax", SpaceType.EVENT),
    SpaceEventType.LUXURY_TAX
));

// Space 37
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(38, "Park Place", SpaceType.STREET),
    streetSets.get(7),
    350,
    streetSets.get(7).getHouseCost(),
    new ArrayList<>(Arrays.asList(35, 175, 500, 1100, 1300, 1500))
));

// Space 38
boardSpaces.add(new BoardSpaceEvent(
    new BoardSpace(39, "Chance", SpaceType.EVENT),
    SpaceEventType.CHANCE
));

// Space 39
boardSpaces.add(new BoardSpaceStreet(
    new BoardSpace(40, "Boardwalk", SpaceType.STREET),
    streetSets.get(7),
    400,
    streetSets.get(7).getHouseCost(),
    new ArrayList<>(Arrays.asList(50, 200, 600, 1400, 1700, 2000))
));
}

public void startGame() {
    populateStreets();
    populateSpaces();
}

```

```
public int getGameID() {  
    return gameID;  
}  
  
}
```

Klasa GameManager w projekcie Monopoly zarządza rozgrywką, kontrolując stan gry oraz interakcje między graczami i planszą. Konstruktor klasy przyjmuje listę loginów graczy, tworzy instancje graczy, sprawdza ich liczbę (od 2 do 4), a następnie losuje gracza rozpoczynającego grę.

Funkcja nextTurn() zmienia aktualnego gracza na następnego w kolejce. Funkcje handleSpaceEvent() i handlePropertyEvent() obsługują wydarzenia na planszy, takie jak kupowanie nieruchomości, płacenie czynszu lub podatków, czy trafienie do więzienia.

Metody populateStreets() i populateSpaces() inicjalizują zestawy ulic oraz pola na planszy, definiując ich właściwości, takie jak koszt zakupu, czynsz i typ pola. Funkcja startGame() uruchamia grę, wywołując te metody.

Klasa zarządza również identyfikacją gry poprzez zmienną gameID, która jest inkrementowana przy każdej nowej grze.

Omówienie wyszczególnionych wymagań projektu:

GUI:

```
// Login scene
LoginLayout loginLayout = new LoginLayout();
Scene loginScene = new Scene(loginLayout, v: 500, v1: 300);

// Register scene
RegisterLayout registerLayout = new RegisterLayout();
Scene registerScene = new Scene(registerLayout, v: 500, v1: 300);

// Main menu scene
MainMenuLayout mainMenuLayout = new MainMenuLayout();
Scene mainMenuScene = new Scene(mainMenuLayout, v: 500, v1: 300);

// Create lobby scene
CreateLobbyLayout createLobbyLayout = new CreateLobbyLayout();
Scene createLobbyScene = new Scene(createLobbyLayout, v: 500, v1: 300);

// List lobbies scene
ListLobbiesLayout listLobbiesLayout = new ListLobbiesLayout();
Scene listLobbiesScene = new Scene(listLobbiesLayout, v: 500, v1: 300);

// Lobby scene
LobbyLayout lobbyLayout = new LobbyLayout();
Scene lobbyScene = new Scene(lobbyLayout, v: 500, v1: 300);

// Game scene
FXMLLoader loader = new FXMLLoader(getClass().getResource("/monopoly.fxml"));
AnchorPane root = loader.load();
GameController gameController = loader.getController();
Scene gameScene = new Scene(root);

primaryStage.setScene(loginScene);
primaryStage.setTitle("Monopoly Game");
primaryStage.show();
```

W aplikacji Monopoly różne widoki interfejsu użytkownika są definiowane i konfigurowane jako odrębne sceny. Najpierw tworzona jest scena logowania, która korzysta z LoginLayout i ma wymiary 500x300 pikseli. Następnie definiowana jest scena rejestracji z RegisterLayout o tych samych wymiarach.

Dalsze sceny obejmują główne menu, które wykorzystuje MainMenuLayout, a także scenę tworzenia lobby z CreateLobbyLayout, obie o wymiarach 500x300 pikseli. Kolejno, scena listy lobby z ListLobbiesLayout oraz scena samego lobby z LobbyLayout, również mają te same wymiary.

```

// List lobbies scene - Refresh button
listLobbiesLayout.getRefreshLobbiesButton().setOnAction(e -> {
    // Mateusz Jankowski +1
    Thread refreshThread = new Thread(new Runnable(){
        // Mateusz Jankowski +1
        @Override
        public void run() {
            try {
                dataOut.writeUTF(str: "listLobbies");
                dataOut.flush();

                ArrayList<String> listOfLobbies = new ArrayList<>();
                int listOfLobbiesSize = dataIn.readInt();
                for (int i = 0; i < listOfLobbiesSize; i++) {
                    String lobby = dataIn.readUTF();
                    listOfLobbies.add(lobby);
                    System.out.println("Lobby: " + lobby);
                }

                Platform.runLater(() -> {
                    listLobbiesLayout.clearLobbies();
                    listOfLobbies.forEach(listLobbiesLayout::addLobby);

                    Alert alert = new Alert(Alert.AlertType.INFORMATION);
                    alert.setTitle("Success");
                    alert.setHeaderText("Lobbies refreshed");
                    alert.setContentText("Lobbies have been refreshed");
                    alert.showAndWait();
                });
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    });
    refreshThread.start();
});

```

W pokazanym fragmencie kodu obsługiwany jest przycisk odświeżania listy lobby. Po kliknięciu przycisku uruchamiany jest nowy wątek, który komunikuje się z serwerem w celu pobrania zaktualizowanej listy lobby.

Wątek wysyła żądanie "listLobbies" do serwera, a następnie odczytuje rozmiar listy lobby. Dla każdego elementu listy odczytywana jest nazwa lobby, która jest dodawana do listy. Informacja o nazwie lobby jest również wypisywana w konsoli.

Po zakończeniu pobierania danych, na głównym wątku aplikacji (JavaFX) jest wywoływana metoda `Platform.runLater`. W tej metodzie interfejs użytkownika jest aktualizowany — najpierw lista lobby jest czyszczona, a następnie dodawane są nowe elementy. Na końcu użytkownikowi wyświetlane jest powiadomienie o sukcesie operacji odświeżania.

W przypadku wystąpienia wyjątku `IOException` błąd jest obsługiwany poprzez wypisanie śladu stosu błędów.

Połączenie z bazą danych SQL:

```
protected static boolean loginUser(String login, String password, User.Users users) {

    if (login == null || password == null) return false;
    if (!users.users.containsKey(login)) return false;
    User user = users.users.get(login);
    if (!user.getPassword().equals(password)) return false;
    if (user.isLoggedIn()) return false;

    try (Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/monopolydb", "root", "");
        PreparedStatement statement = connection.prepareStatement("UPDATE user SET isLoggedIn = true WHERE login = ?")) {
        statement.setString(1, login);
        statement.executeUpdate();
        users.fetchUsers();
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Metoda `loginUser` w aplikacji `Monopoly` służy do uwierzytelniania użytkowników. Proces rozpoczyna się od sprawdzenia, czy login i hasło nie są puste. Następnie metoda weryfikuje, czy podany login istnieje w bazie użytkowników. Po potwierdzeniu istnienia użytkownika, sprawdzane jest, czy podane hasło jest poprawne oraz czy użytkownik nie jest już zalogowany.

Jeśli wszystkie te warunki są spełnione, metoda nawiązuje połączenie z bazą danych `monopolydb` i wykonuje zapytanie SQL, które aktualizuje stan logowania użytkownika na "zalogowany". Po pomyślnym wykonaniu zapytania, metoda odświeża listę użytkowników, wywołując `users.fetchUsers()`, i zwraca `true`, sygnalizując sukces. W przypadku wystąpienia błędów SQL, metoda drukuje stos śladu błędu i zwraca `false`, informując o niepowodzeniu operacji.

```

protected static String registerUser(String login, String password, String nickname, User.Users users) {
    if (users.users.containsKey(login)) return "exists";
    try (Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/monopolydb", "root", "")) {
        PreparedStatement statement = connection.prepareStatement("INSERT INTO user (login, password, nickname, isLoggedIn) VALUES (?, ?, ?, false)") {
            statement.setString(1, login);
            statement.setString(2, password);
            statement.setString(3, nickname);
            statement.executeUpdate();
            users.fetchUsers();
            return "success";
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return "error";
    }
}

```

Metoda `registerUser` w aplikacji `Monopoly` umożliwia rejestrację nowych użytkowników. Proces rozpoczyna się od sprawdzenia, czy login już istnieje w bazie użytkowników. Jeśli login jest zajęty, metoda zwraca komunikat "exists".

Jeżeli login jest dostępny, metoda nawiązuje połączenie z bazą danych `monopolydb` i wykonuje zapytanie SQL, które wstawia nowego użytkownika do tabeli `user` z podanymi danymi: loginem, hasłem i pseudonimem. Stan logowania nowego użytkownika jest ustawiany na "false". Po pomyślnym dodaniu użytkownika, metoda aktualizuje listę użytkowników, wywołując `users.fetchUsers()`, i zwraca "success", sygnalizując sukces rejestracji. W przypadku wystąpienia błędu SQL, metoda drukuje stos śladu błędu i zwraca "error", informując o niepowodzeniu operacji.

Wielowątkowość:

```
public void startLobbyListener(DataInputStream dataIn, DataOutputStream dataOut, LobbyLayout lobbyLayout,
                               Stage primaryStage, Scene gameScene, GameController gameController) {

    ↳ Mateusz +1
    lobbyListener = new Thread(new Runnable(){
        ↳ Mateusz +1
        @Override
        public void run() {
            try {
                while (true) {
                    String command = dataIn.readUTF();
                    System.out.println("Command received client lobby listener: " + command);
                    if (command.equals("refreshLobbyUsers")) {
                        System.out.println("Refreshing lobby users");
                        ArrayList<String> listOfUsers = new ArrayList<>();
                        int listOfUsersSize = dataIn.readInt();
                        for (int i = 0; i < listOfUsersSize; i++) {
                            String user = dataIn.readUTF();
                            listOfUsers.add(user);
                            System.out.println("User: " + user);
                        }

                        Platform.runLater(() -> {
                            lobbyLayout.clearUsers();
                            listOfUsers.forEach(lobbyLayout::addUser);
                        });
                    } else if (command.equals("startGame")) {
                        System.out.println("Starting game");
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
}
```

Ta metoda uruchamia wątek nasłuchujący na komunikaty z serwera dotyczące aktualizacji użytkowników w lobby oraz rozpoczęcia gry. Jeśli otrzyma komunikat "refreshLobbyUsers", aktualizuje listę użytkowników w interfejsie graficznym aplikacji. Jeśli otrzyma komunikat "startGame", rozpoczyna grę.

```

Thread leaveLobbyThread = new Thread(new Runnable() {
    ⚡ Mateusz Jankowski +1
    @Override
    public void run() {
        try {
            dataOut.writeUTF(str: "stopLobbyListener");
            dataOut.writeUTF(str: "leaveLobby");

            dataOut.writeUTF(currentLobbyName);
            dataOut.writeUTF(currentUserLogin);
            System.out.println("\n");
            System.out.println("Current lobby: " + currentLobbyName);
            System.out.println("Leaving user: " + currentUserLogin);

            boolean removedSuccess = dataIn.readBoolean();
            if (!removedSuccess){
                Platform.runLater(() -> {
                    Alert alert = new Alert(Alert.AlertType.ERROR);
                    alert.setTitle("Error");
                    alert.setHeaderText("Leaving lobby failed");
                    alert.setContentText("Error occurred during leaving lobby, could not remove player");
                    alert.showAndWait();
                });
                startLobbyListener(dataIn, dataOut, lobbyLayout, primaryStage, gameScene, gameController);
                return;
            }

            currentLobbyName = null;

            Platform.runLater(() -> {
                primaryStage.setScene(mainMenuScene);
            });

        } catch (IOException ex) {
            startLobbyListener(dataIn, dataOut, lobbyLayout, primaryStage, gameScene, gameController);
            ex.printStackTrace();
        }
    }
});
leaveLobbyThread.start();

```

Ta sekcja kodu reaguje na naciśnięcie przycisku opuszczania lobby w interfejsie graficznym. Po kliknięciu, uruchamia wątek, który wysyła komunikaty do serwera informujące o opuszczeniu lobby i zatrzymuje nasłuchiwanie na zmiany w lobby. Jeśli użytkownik nie zostanie pomyślnie usunięty z lobby, wyświetla komunikat o błędzie. Po pomyślnym opuszczeniu lobby, zmienia scenę interfejsu graficznego na główny menu.

W pierwszej sekcji, jest uruchamiany nowy wątek (lobbyListener) w metodzie startLobbyListener(), który nasłuchuje na komunikaty z serwera w pętli nieskończonej.

W drugiej sekcji, po naciśnięciu przycisku opuszczania lobby, jest tworzony nowy wątek (leaveLobbyThread) wewnątrz metody run(), który wysyła komunikaty do serwera dotyczące opuszczenia lobby oraz zatrzymuje nasłuchiwanie na zmiany w lobby. Dodatkowo, korzysta z metody Platform.runLater(), aby zmienić interfejs graficzny po stronie klienta.

Dzięki zastosowaniu wielowątkowości, aplikacja może jednocześnie nasłuchiwać na komunikaty z serwera oraz reagować na interakcje użytkownika, co pozwala na płynne działanie interfejsu użytkownika bez blokowania wątku głównego.

Komunikacja Klient-Serwer:

```

Mateusz
public static void main(String[] args) { launch(args); }

Mateusz +1
@Override
public void start(Stage primaryStage) throws IOException {
    InetAddress ip = InetAddress.getByName(HOST);
    Socket socket = new Socket(ip, PORT);

    System.out.println("Connected to server: " + socket);

    DataOutputStream dataOut = new DataOutputStream(socket.getOutputStream());
    DataInputStream dataIn = new DataInputStream(socket.getInputStream());
}
```

Metoda start rozpoczyna pracę aplikacji i ustanawia połączenie z serwerem. Najpierw pobierany jest adres IP serwera za pomocą InetAddress.getByName(HOST). Następnie tworzony jest nowy socket, który łączy się z serwerem na określonym porcie (PORT).

Po pomyślnym nawiązaniu połączenia na konsoli wypisywana jest informacja o udanym połączeniu.

Tworzone są dwa strumienie danych: DataOutputStream do wysyłania danych do serwera oraz DataInputStream do odbierania danych z serwera, które korzystają z odpowiednich strumieni wejścia i wyjścia socketu.

Całość pozwala na dwukierunkową komunikację z serwerem, co jest kluczowe dla funkcjonalności aplikacji.

```
└─ Mateusz +1
public static void main(String[] args) {
    try {
        ServerSocket server = new ServerSocket(PORT);
        Socket socket;

        System.out.println("Server started on port " + PORT + "...");

        while (true) {
            socket = server.accept();
            System.out.println("New client request received: " + socket);

            DataInputStream dataIn = new DataInputStream(socket.getInputStream());
            DataOutputStream dataOut = new DataOutputStream(socket.getOutputStream());

            System.out.println("Creating a new handler for this client...");

            ClientHandler client = new ClientHandler(socket, i, dataIn, dataOut);

            Thread thread = new Thread(client);

            System.out.println("Adding this client to active client list");

            clients.put(i, client);

            thread.start();

            i++;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Serwer jest inicjalizowany przy użyciu `ServerSocket` na podanym porcie. Następnie w pętli nieskończonej serwer oczekuje na nowe połączenia od klientów. Gdy połączenie zostanie zaakceptowane, tworzy nowy socket dla klienta.

Dla każdego nowego klienta tworzony jest strumień wejścia `DataInputStream` oraz strumień wyjścia `DataOutputStream`, co pozwala na komunikację między serwerem a klientem.

Następnie tworzony jest nowy obiekt `ClientHandler`, który obsługuje komunikację z klientem. Każdy `ClientHandler` jest uruchamiany w nowym wątku, co pozwala na równoczesne obsługiwanie wielu klientów.

Nowy klient jest dodawany do aktywnej listy klientów i uruchamiany jest wątek obsługujący jego komunikację.

W przypadku wystąpienia wyjątku, błąd jest wypisywany na konsolę.

```
@Override
public void run() {
    String clientCommand;
    initializeCommands();
    while (true) {
        try {
            // receive the string with the command from the client
            clientCommand = dataIn.readUTF();
            System.out.println("Command received: " + clientCommand);

            // map the command to the appropriate function
            Command command = commandMap.get(clientCommand);
            System.out.println("Command: " + command);

            if (command != null) {
                // execute the command
                command.execute(dataIn, dataOut, clientID);
            }
        } catch (SocketException | EOFException e) {
            System.out.println("Client " + clientID + " disconnected");
            break;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

1 usage  Mateusz
private static void initializeCommands() {
    commandMap.put("login", new LoginCommand());
    commandMap.put("register", new RegisterCommand());
    commandMap.put("createLobby", new CreateLobbyCommand());
    commandMap.put("joinLobby", new JoinLobbyCommand());
    commandMap.put("listLobbies", new ListLobbiesCommand());
    commandMap.put("logout", new LogoutCommand());
    commandMap.put("exit", new ExitCommand());
    commandMap.put("leaveLobby", new LeaveLobbyCommand());
    commandMap.put("startGame", new StartGameCommand());
    commandMap.put("stopLobbyListener", new StopLobbyListenerCommand());
    commandMap.put("rollDice", new RollDiceCommand());
    commandMap.put("endTurn", new EndTurnCommand());
}
```

Metoda run() jest częścią wątku, który nasłuchuje i obsługuje komendy wysyłane przez klientów. Głównym zadaniem tej metody jest odbieranie komend od klientów, mapowanie ich na odpowiednie funkcje oraz ich wykonanie.

Po inicjalizacji komend w metodzie initializeCommands(), pętla while (true) nasłuchuje ciągle na nowe komendy. Gdy nadejdzie komenda od klienta, jest ona odczytywana i przypisywana do zmiennej clientCommand. Następnie sprawdzane jest, czy istnieje odpowiednia

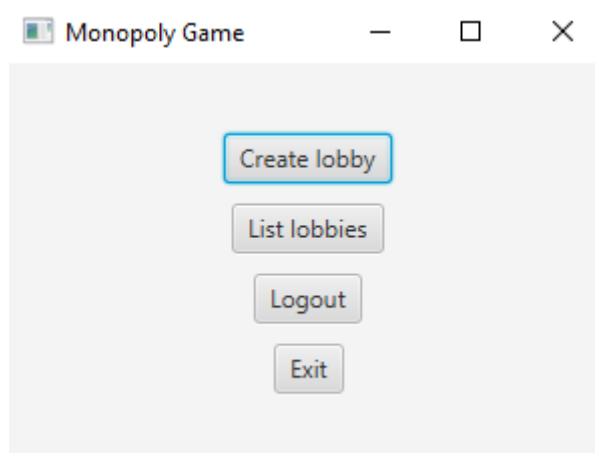
komenda zmapowana do tej otrzymanej. Jeśli tak, to jest ona wykonywana poprzez wywołanie metody `execute()` na obiekcie reprezentującym daną komendę.

Jeśli wystąpi błąd podczas odczytywania komendy (np. przerwanie połączenia przez klienta), wychwytywany jest odpowiedni wyjątek (`SocketException` lub `EOFException`), wypisywany jest komunikat o odłączeniu klienta i pętla zostaje przzerwana, co kończy działanie wątku obsługującego danego klienta.

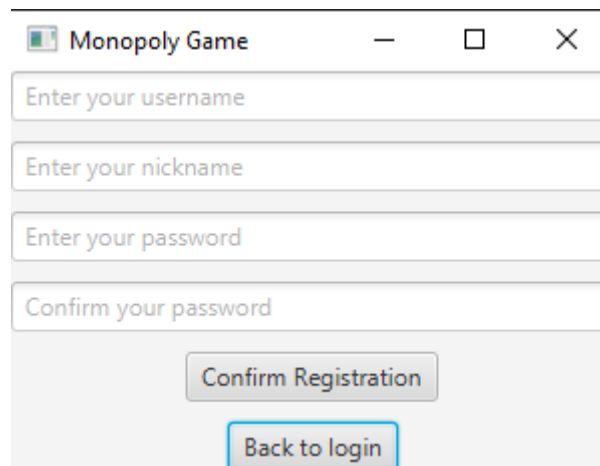
Problemy oraz trudności napotkane w trakcie wykonywania projektu:

- Na początku w serwerze do czytania wiadomości klienta użyliśmy `while` loopa z `switchem` i `case`'ami, ale wraz z powiększającym się projektem zdecydowaliśmy się przerobić system na bardziej optymalny używając `HashMap`y.
- Przed implementacją bazy danych używaliśmy zwykłego pliku tekstowego do którego zapisywaliśmy i zczytywaliśmy dane używając `buffered reader`.
- Ponieważ zdecydowaliśmy się użyć oryginalnej planszy, tworzenie jej okazało się zbyt trudne klasycznymi metodami deklarowania elementów w programie więc zdecydowaliśmy się użyć aplikacji `SceneBuilder`

4. Zrzuty ekranu z aplikacji.



Menu



Monopoly Game

Enter your username

Enter your nickname

Enter your password

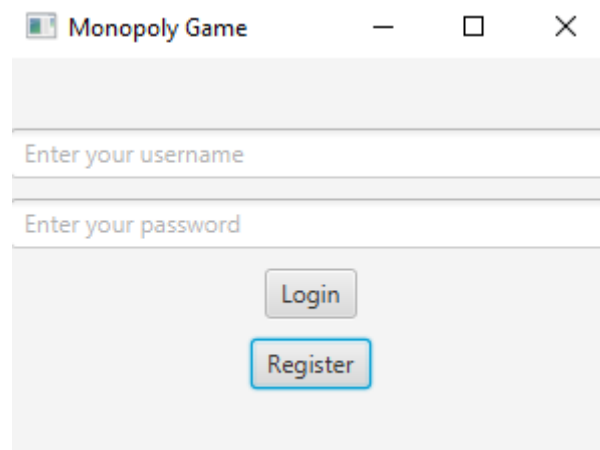
Confirm your password

Confirm Registration

Back to login

This screenshot shows the registration interface of the Monopoly Game. It features four text input fields for 'Enter your username', 'Enter your nickname', 'Enter your password', and 'Confirm your password'. Below these fields are two buttons: 'Confirm Registration' and 'Back to login'. The 'Back to login' button is highlighted with a blue border.

Ekran rejestracji



Monopoly Game

Enter your username

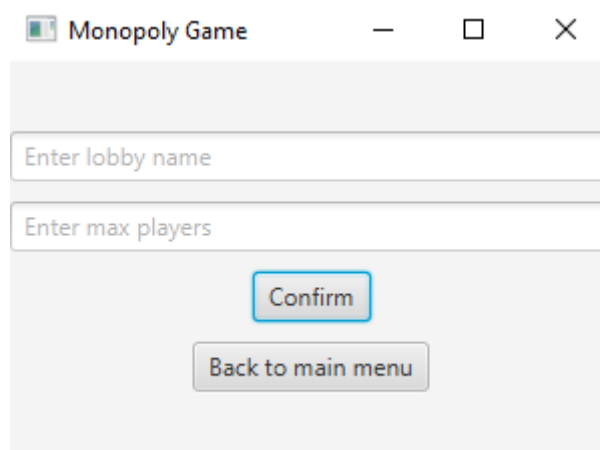
Enter your password

Login

Register

This screenshot shows the login interface of the Monopoly Game. It features two text input fields for 'Enter your username' and 'Enter your password'. Below these fields are two buttons: 'Login' and 'Register'. The 'Register' button is highlighted with a blue border.

Ekran logowania



Monopoly Game

Enter lobby name

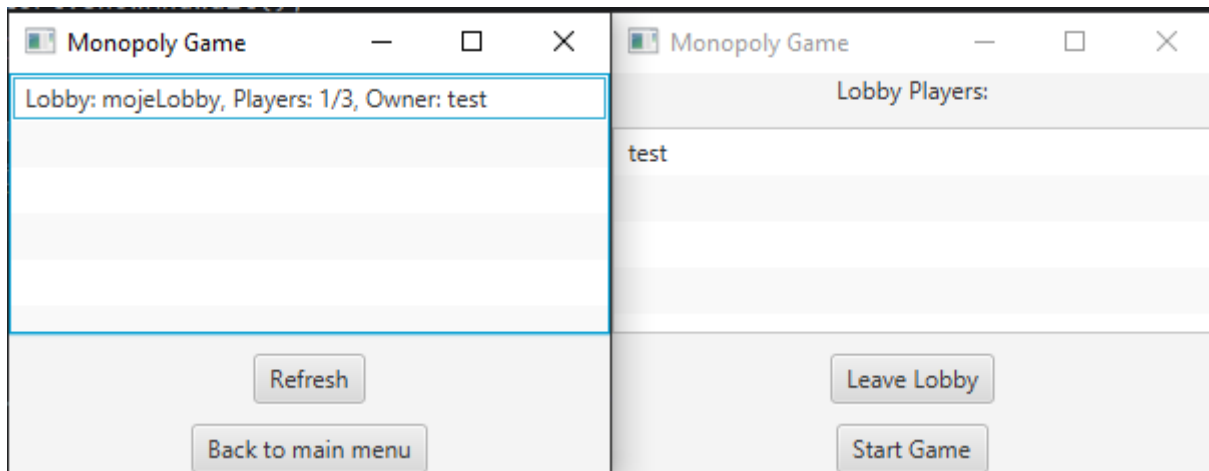
Enter max players

Confirm

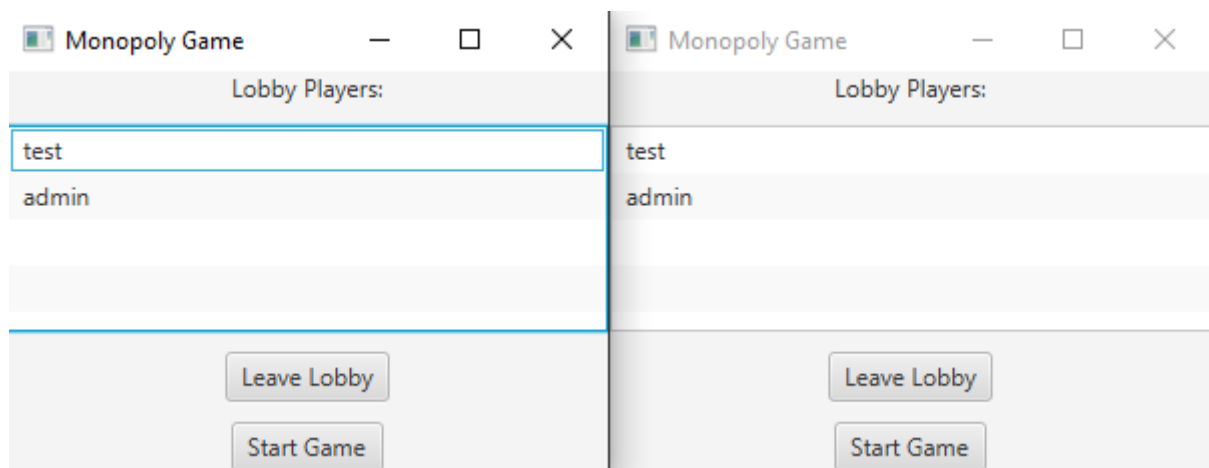
Back to main menu

This screenshot shows the lobby creation interface of the Monopoly Game. It features two text input fields for 'Enter lobby name' and 'Enter max players'. Below these fields are two buttons: 'Confirm' and 'Back to main menu'. The 'Confirm' button is highlighted with a blue border.

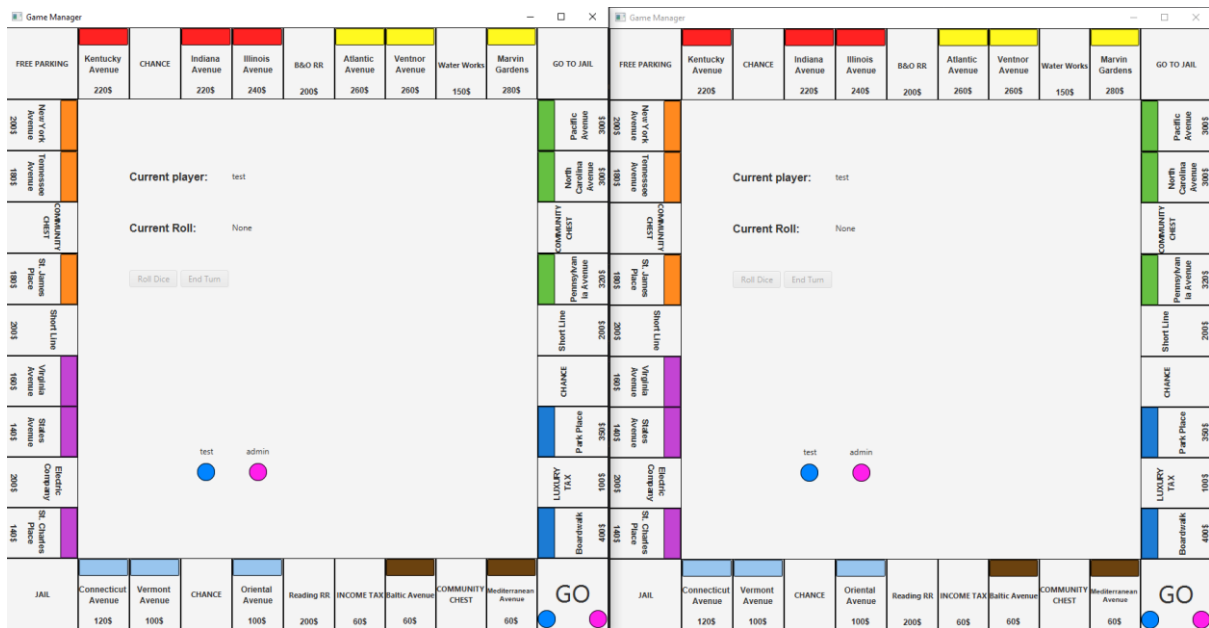
Tworzenie lobby



Lista lobby



Lista lobby



Wygląd planszy do gry (dla dwóch różnych klientów)

5. Wnioski.

Projekt „Monopoly” zrealizowany w ramach zajęć projektowych wprowadza funkcjonalność popularnej gry planszowej przy użyciu narzędzi takich jak IntelliJ Idea oraz GitHub. Nasza implementacja koncentruje się na kluczowych aspektach rozgrywki, takich jak komunikacja sieciowa, wielowątkowość oraz połączenie z bazą danych, umożliwiając jednoczesną grę czterech graczy w trybie sieciowym.

Podczas realizacji projektu, kluczowym celem było odwzorowanie podstawowych zasad gry „Monopoly” z użyciem umiejętności nabytych podczas laboratoriów z przedmiotu „Programowanie w Języku Java”. Decyzja o zrezygnowaniu z opcji handlu nieruchomościami była podyktowana chęcią uproszczenia rozgrywki, co pozwoliło skupić się na innych, bardziej technicznych aspektach projektu.

Analizując różne typy pól w grze „Monopoly”, można zauważyć, że każde z nich wnosi unikalną wartość strategiczną i wpływa na dynamikę gry. W naszym projekcie zachowaliśmy oryginalną planszę oraz zasady dotyczące ruchu graczy, budowy domów i hoteli oraz pobierania czynszu.

Projekt pokazał, że możliwe jest przeniesienie klasycznej gry planszowej do środowiska cyfrowego, zachowując jej istotne cechy i zasady. Użycie technologii sieciowych oraz baz danych pozwoliło na stworzenie dynamicznej i interaktywnej wersji gry, która może być łatwo rozszerzana i modyfikowana w przyszłości.

Podsumowując, projekt „Monopoly” jest udanym przykładem zastosowania nowoczesnych technologii do realizacji kompleksowego zadania programistycznego, łącząc aspekty techniczne z elementami klasycznej gry planszowej. Przyszłe iteracje projektu mogą uwzględniać dodatkowe funkcje, takie jak handel nieruchomościami, aby jeszcze bardziej zbliżyć się do pełnego doświadczenia gry „Monopoly”.

6. Literatura.

- Wiadomości z zajęć/wykładów.
- "TCP/IP Sockets in Java: Practical Guide for Programmers" – Kenneth L. Calvert, Michael J. Donahoo
- "Database Systems: The Complete Book" – Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom

- "Java Concurrency in Practice" – Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea
- https://monopoly.fandom.com/wiki/Main_Page