KITron - dokumentacja

Autorzy:
Karolina Biela
Agata Bogacz
Barbara Chraścik
Jakub Jungiewicz
Mikołaj Kozieł
Franciszek Kubis

server

I. Serwer

Za włączenie serwer odpowiedzialna jest klasa Main:

```java
package server.main;

import java.io.IOException;

public class Main {

    public static void main(String[] args) throws IOException {
        Server server = new Server();
        server.process( poolSize: 20);
    }
}
```

W argumencie podajemy, ile wątków ma być na raz odpowiedzialna za obsługę wiadomości TCP ze strony klientów.

```java
public void process(int poolSize) throws IOException {

    System.out.println("SERVER");
    System.out.println("Waiting for players...");
    ExecutorService executorService = Executors.newFixedThreadPool(poolSize);

    try {
        serverSocket = new ServerSocket(portNumber);

        while(true){

            clientSocket = serverSocket.accept();
            System.out.println("Client " + playerIdGiver.incrementAndGet() + " connected");
            Player player = new Player(playerIdGiver.get(), clientSocket);

            players.add(player);

            Runnable clientListener = new ClientHandler( server: this, player, clientSocket);
            executorService.execute(clientListener);

        }

    } catch (IOException e) {
        e.printStackTrace();
    }
    finally{
        if (serverSocket != null){
            serverSocket.close();
        }
    }
}
```

Metoda process czeka na kolejnych klientów i tworzy każdemy nowy socket po czym włącza wątek odpowiedzialny za nasłuchiwanie, obsługę wiadomości od klienta oraz dodaje klienta do listy graczy na serwerze.
Dodatkowo w klasie Serwer znajdują się metody odpowiedzialne odpowiednio za: dodanie pokoju, znalezienie pokoju po nazwie, znalezienie pokoju po ID, dołączenie do istniejącego pokoju oraz opuszczenie pokoju.

```
public synchronized int addRoom(String name, int maxPeople, Player player) {

    if(getRoomId(name) == -1){
        Room room = new Room(STANDARD_BOARD_WIDTH,STANDARD_BOARD_HEIGHT,maxPeople,name);
        rooms.add(room);
        new Thread(room).start();
        room.join(player);
        System.out.println("Player " + player.getName() + " added new Room " + name + " " + maxPeople);
        return 0;
    }
    return 1;
}
```

Tworzy nowy pokój z planszą o stałych wymiarach, podaje w konstruktorze również maksymalną ilość ludzi jaka może dołączyć do pokoju oraz nazwę pokoju.

```
private Room getRoomByName(String name) {
    for(Room room : rooms) {
        if(room.getName().equals(name)) {
            return room;
        }
    }
    return null;
}
```

```
public int getRoomId(String name) {
    for(int i = 0; i < rooms.size(); i++){
        if(rooms.get(i).getName().equals(name))
            return i;
    }
    return -1;
}
```

getRoomByName i getRoomId szukają pokoju odpowiednio po nazwie i po identyfikatorze.

```
public synchronized int joinRoom(Player player, String roomName){

    Room room = getRoomByName(roomName);

    if(room != null && !room.isRoomActive())
    {
        room.join(player);
        System.out.println("Player " + player.getName() + " joined Room " + roomName);
        return 0;
    }
    return 1;
}
```

Metoda joinRoom odpowiedzialna jest za dodanie gracza do pokoju.

```
public synchronized int leaveRoom(Player player){
    for(Room room : rooms) {
        if(room.containsPlayer(player)){
            room.leave(player);
            System.out.println("Player " + player.getName() + " left the room");
            return 0;
        }
    }
    return 1;
}
```

Metoda leaveRoom wyrzuca danego gracza z pokoju w którym obecnie przebywa.

II. ClientHandler

Odpowiedzialnością klasy ClientHandler jest obsługa requestów (wiadomości TCP) od klienta. Wiadomości obsługiwane to: “left”, “right”, “up”, “down”, “hostRoom”, “joinRoom”, “leaveRoom”, “initPlayer” i “roomList”.

```
System.out.println("Player playing: - " + (player.getPlayerState() == PlayerState.PLAYING));

if(player.getPlayerState() == PlayerState.PLAYING) {
    switch (messageList[0]) {
        case "left":
            if(player.getDirection() != Direction.LEFT && player.getDirection() != Direction.RIGHT){
                player.setDirection(Direction.LEFT);
                player.addToPath(player.getPosition());
                System.out.println("left");
            }
            break;
        case "right":
            if(player.getDirection() != Direction.LEFT && player.getDirection() != Direction.RIGHT){
                player.setDirection(Direction.RIGHT);
                System.out.println("right");
                player.addToPath(player.getPosition());
            }
            break;
        case "down":
            if(player.getDirection() != Direction.DOWN && player.getDirection() != Direction.UP){
                player.setDirection(Direction.DOWN);
                System.out.println("down");
                player.addToPath(player.getPosition());
            }
            break;
        case "up":
            if(player.getDirection() != Direction.DOWN && player.getDirection() != Direction.UP){
                player.setDirection(Direction.UP);
                System.out.println("up");
                player.addToPath(player.getPosition());
            }
            break;
    }
    out = new PrintWriter(clientSocket.getOutputStream(), autoFlush: true);
    out.println("moved");
}
else{
```

Poruszanie się w którąś stronę odsyła klientowi wiadomość “moved”.

Pozostałe wiadomości:

```
else{

    int i;
    response = "wrong command";
    //wiadomosc bedize miec postac albo
    // hostRoom nazwaPokoju maxIloscGraczy
    // joinRoom nazwa pokoju
    // leaveRoom
    // initPlayer imieGracza

        switch (messageList[0]){
            case "hostRoom":
                i = server.addRoom(messageList[1], Integer.parseInt(messageList[2]), player);
                if(i == 0) response = "hostRoom success";
                else response = "hostRoom fail";
                break;
            case "joinRoom":
                i = server.joinRoom(player, messageList[1]);
                if(i == 0) response = "joinRoom success";
                else response = "joinRoom fail";
                break;
            case "leaveRoom":
                i = server.leaveRoom(player);
                if(i == 0) response = "leaveRoom success";
                else response = "leaveRoom fail";
                break;
            case "initPlayer":
                //check if there is player with this name
                if(messageList.length == 3){
                    player.init(messageList[1]);
                    response = "init OK";
                }
                break;
            case "roomList":
                response = createRoomList();
    }

    out = new PrintWriter(clientSocket.getOutputStream(),  autoFlush: true);
    out.println(response);
}
```

Obsługujemy odpowiednią wiadomość i odsyłamy odpowiednią odpowiedz.

```
private String createRoomList() {
    StringBuilder roomList = new StringBuilder();

    for(Room room: server.rooms){
        roomList.append(room.getName());
        roomList.append(",");
        roomList.append(room.getUserNumber());
        roomList.append(",");
        roomList.append(room.getMaxPlayers());
        roomList.append(';');
    }
    return roomList.toString();
}
```

Metoda tworzy listę pokoi i robi z nich Stringa, którego później odsyła do klienta.

III. Player

Klasa player opisuje stan gracza oraz to, w którym kierunku się porusza w danym momencie, jakie ma imię, jaki kolor, jak szybko sie porusz, jak duży jest jego ślad, czy jest żywy w grze, jaka ma aktualnie pozycje i czy ma aktualnie jakiś bonus np. nieśmiertelność.

```
private final Socket socket;
private int id;
private String color;
private String name;
private Direction direction;
private Direction newDirection;
private boolean initialized = false;
private Path path;
private PlayerState playerState;
private Point position;
private int speed = 2;
private int size = 3;
private boolean alive = true;
private boolean immortal = false;

Player(int id, Socket socket){
    this.id = id;
    this.path = new Path();
    this.playerState = PlayerState.IDLE;
    this.socket = socket;

}
```

Najważniejsze metody klasy Player:

- clearPath - czyści ścieżkę gracza,
- init - inicjuje gracza z odpowiednimi zmiennymi,
- findNewPosition, porusza gracza w odpowiednim kierunki i znajduje mu nowy punkt

```
public void clearPath(){
    this.path = new Path();
}

public void init(String name){
    if(!initialized){
        this.name = name;
        initialized = true;
        System.out.println("Initialized player: " + name);
    }

}

public Point findNewPosition() {

    int oldX = position.getX();
    int oldY = position.getY();

    Point point = null;

    switch(direction) {
        case DOWN:
            point = new Point(oldX, y: oldY+speed, state: "end");
            break;
        case UP:
            point = new Point(oldX, y: oldY-speed, state: "end");
            break;
        case LEFT:
            point = new Point( x: oldX-speed,oldY, state: "end");
            break;
        case RIGHT:
            point = new Point( x: oldX+speed,oldY, state: "end");
            break;
    }

    return point;
}
```

Pozostałe metody to zwykłe gettery i/lub settery.

Dodatkowo w pakiecie serwer.main znajdują się dwa enumy.

```
package server.main;

public enum Direction {
    UP, DOWN, LEFT, RIGHT
}
```

```
package server.main;

public enum PlayerState {
    PLAYING, WAITTING, IDLE
}
```

IV. ROOM

Pokój posiada plansze, maksymalna ilość graczy, timer oraz listę graczy i kolorów, które im przydzielamy, dodatkowym elementem miał być powerUpSpawner, który miał za zadanie dodawać powerUpy na planszy.

```
//room properties
private Board board;
private int maxPlayers;
private boolean roomActive = false;
private String name;
private int alive;
private Timer timer;
private List<Player> players = new ArrayList<>();
private List<String> colors = Arrays.asList("#9D00FF", "#FF00FF","#00FFFF","#00FF00","#FF0000","#FFFF00","#ff0099","#6e0dd0");
private PowerUpSpawner powerUpSpawner;


//sockets
private static int multicastPort = 4446;
private DatagramChannel channel;
private NetworkInterface multicastInterface = null;
private DatagramChannel multicastChannel = null;
private InetSocketAddress serverAddress = new InetSocketAddress( hostname: "239.1.1.1",  port: 5000);
MulticastSocket multicastSocket;
InetAddress group;
```

```
public Room(int width, int height, int maxPlayers, String name){
    this.board = new Board(width, height);
    this.maxPlayers = maxPlayers;
    this.name = name;
    timer = new Timer();

    try {
        multicastSocket = new MulticastSocket();
        group = InetAddress.getByName("224.0.113.0");
    } catch (IOException e) {
        e.printStackTrace();
    }

    powerUpSpawner = new PowerUpSpawner(board);


    try {
        multicastInterface = NetworkInterface.getNetworkInterfaces().nextElement();
        multicastChannel = DatagramChannel.open(StandardProtocolFamily.INET)
                .setOption(StandardSocketOptions.SO_REUSEADDR,  value: true)
                .bind(new InetSocketAddress( port: 5000))
                .setOption(StandardSocketOptions.IP_MULTICAST_IF, multicastInterface);
        multicastChannel.configureBlocking(false);
        MembershipKey groupKey = multicastChannel.join(Inet4Address.getByName("239.1.1.1"), multicastInterface);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

W pokoju posiadamy timer, który co odpowiedni odcinek czasowy będzie robił krok na planszy i poruszał graczy oraz sprawdzał kolizje. W konstruktorze dodatkowo tworzymy multicastSocket na który będziemy rozsyłać do wszystkich graczy w pokoju stan mapy.

Dodawanie playera do pokoju oraz opuszczanie pokoju przez playera:

```
public synchronized void join(Player player){

    players.add(player);
    player.setPlayerState(PlayerState.WAITING);

    player.setColor(colors.get(players.size()));
}

public synchronized void leave(Player player){
    players.remove(player);
    player.setPlayerState(PlayerState.IDLE);

}

public boolean isRoomActive() { return roomActive; }

public String getName() { return name; }
```

Metoda run czeka, aż wszyscy dołączą do pokoju po czym rozpoczyna grę i wysyła wszystkim, że gra się zaczęła. Włącza timera który co 60 ms aktualizuje planszę.

```
@Override
public void run() {


    System.out.println("Room init");
    this.timer = new Timer();
    board.refreshBoard();

    while(players.size() != maxPlayers){
        try {
            System.out.println(players.size());
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }


    System.out.println("Game started in: " + this.name);

    startGame();
    roomActive = true;

    sendToAllStartMessage();

    timer.schedule(new processTask( room: this),  delay: 0,  period: 60);
    //new Thread(powerUpSpawner).start();

    System.out.println("koniec room");
}
```

Wystartowanie gry - zmiana stanu wszystkich graczy i położenie graczy na planszy.

```
private void startGame() {
    for(Player player : players){
        player.setPlayerState(PlayerState.PLAYING);
        player.setAlive(true);
    }
    alive = players.size();
    putPlayersOnBoard();
}
```

Losujemy pozycje oraz kierunek dla graczy na poczatku:

```
private void putPlayersOnBoard() {
    for(Player player: players){
        player.clearPath();

        List<Point> startPoints = new ArrayList<>();

        int x;
        int y;

        Random r = new Random();
        do{

            x = r.nextInt( bound: (board.getWidth() - 15) + 1) + 8;
            y = r.nextInt( bound: (board.getHeight() - 15) + 1) + 8;

        } while(startPoints.contains(new Point(x, y,  state: "start")));

       Point point = new Point(x, y,  state: "start");
       board.drawPlayer(point, player);

        player.setPosition(point);
        player.addToPath(point);
        Random rand = new Random();
        int direction = rand.nextInt( bound: 4);

        switch (direction){
            case 0:
                player.setDirection(Direction.UP);
                break;
            case 1:
                player.setDirection(Direction.DOWN);
                break;
            case 2:
                player.setDirection(Direction.LEFT);
                break;
            case 3:
                player.setDirection(Direction.RIGHT);
                break;

        }
```

Wysyłanie obecnego stanu planszy do wszystkich graczy
Stan planszy ma postac:
idGracza-kolorgracza-punktStartowy-punktySkrętówGracza-punktKońcowy;
itd.

```
private void sendUpdate() {

    System.out.println("Sending Package UDP");
    DatagramPacket sendPacket;

    //buffer = ByteBuffer.wrap(parsePlayerList().getBytes();

    //ByteBuffer buffer = ByteBuffer.wrap(parsePlayerList().getBytes());
    byte[] buffer = (parsePlayerList()).getBytes();
    try {

        sendPacket = new DatagramPacket(buffer, buffer.length, group, Room.multicastPort);
        multicastSocket.send(sendPacket);
    } catch (IOException e) {
        e.printStackTrace();
    }

}
```

Z pozostałych metod najważniejszą jest update - metoda która znajduje graczom nowe pozycje i sprawdza na planszy kolizje.

```
public boolean containsPlayer(Player player) { return players.contains(player); }

private void update() {
    for(Player player : players) {

        if(player.isAlive()){

            Point newPosition;
            newPosition = player.findNewPosition();

            if(board.checkCollision(player.getPosition(), newPosition, player)){
                player.setPosition(newPosition);
            }
            else{
                player.setAlive(false);
                alive--;
            }
        }
    }
}

public Player findWinner(){
    for(Player player: players){
        if(player.isAlive())
            return player;
    }
    return null;
}

public int getUserNumber() { return players.size(); }

public int getMaxPlayers() { return maxPlayers; }
```

V. Board

W klasie board znajduje się logika sprawdzenia kolizji, jest ona dosyć skomplikowana i na pewno dałoby się ją nieco uprościć. W dokumentacji zawrzemy sprawdzenie kolizji ze względu na jeden kierunek z czterech (UP)

Ustawiamy odpowiednie pola klasy Board w konstruktorze:

```
private int[][] board;
private int width;
private int height;
private Point collisionPoint;

Board(int width, int height){
    this.board = new int[width][height];
    this.width = width;
    this.height = height;
}


public int getWidth() { return  width; }

public int getHeight() { return height; }
```

Logika sprawdzania kolizji i wyznaczania punktów według których będziemy zaznaczać zaznaczone miejsca na planszy po kolejnym korku:

```
public boolean checkCollision(Point oldPosition, Point newPosition, Player player){

    Direction direction = player.getDirection();

    Point leftTop = null;
    Point rightBottom = null;

    boolean collision;

    switch(direction){

        case UP:
            leftTop = new Point(newPosition.getX(), newPosition.getY(), state: "collision");
            rightBottom = new Point( x: oldPosition.getX()+player.getSize()-1, y: oldPosition.getY()-1, state: "collision");

            if(leftTop.getY() < 0){
                leftTop = new Point(newPosition.getX(), y: 0, state: "collision");
                player.setAlive(false);
                System.out.println("up");
            }
            else {
                collision = checkArrayMarkedSectionsUp(leftTop, rightBottom);
                if(!collision){
                    int i = 1;

                    while(board[collisionPoint.getX()][collisionPoint.getY() + i] !=0){
                        i++;
                    }

                    int x = leftTop.getX();
                    int y = collisionPoint.getY() + i;

                    leftTop = new Point(x, y, state: "collision");

                    if(!player.isImmortal())
                        player.setAlive(false);

                }
            }
            break;
```

Kolejne case-y to kolejne kierunki (na pewno dałoby się to jakoś uprościć, niestety czas nie pozwolił) - koniec metody wygląda następująco:

```
markSection(leftTop, rightBottom, player);

return player.isAlive() || player.isImmortal();
```

Podobnie jest w przypadku metod checkArrayMarkedSecions<kierunek>
Wstawiam jedną z czterech metod, która robi to samo co pozostałe tylko różni się delikatnie implementacją, również uważam, że dałoby się to na pewno jakoś uprościć.

```
//w gore
private boolean checkArrayMarkedSectionsUp(Point leftTop, Point rightBottom) {

    for(int j=rightBottom.getY()-1 ; j >= leftTop.getY(); j--){
        for(int i=leftTop.getX(); i<rightBottom.getX(); i++){
            if(board[i][j] > 0){
                collisionPoint = new Point(i, j,  state: "collision");
                System.out.println("collision");
                return false;
            }

        }
    }
    return true;
}
```

Metoda markSection zaznacza, które pola planszy zostały zajęte w kolejnym kroku.
Metoda refreshBoard zeruje plansze.

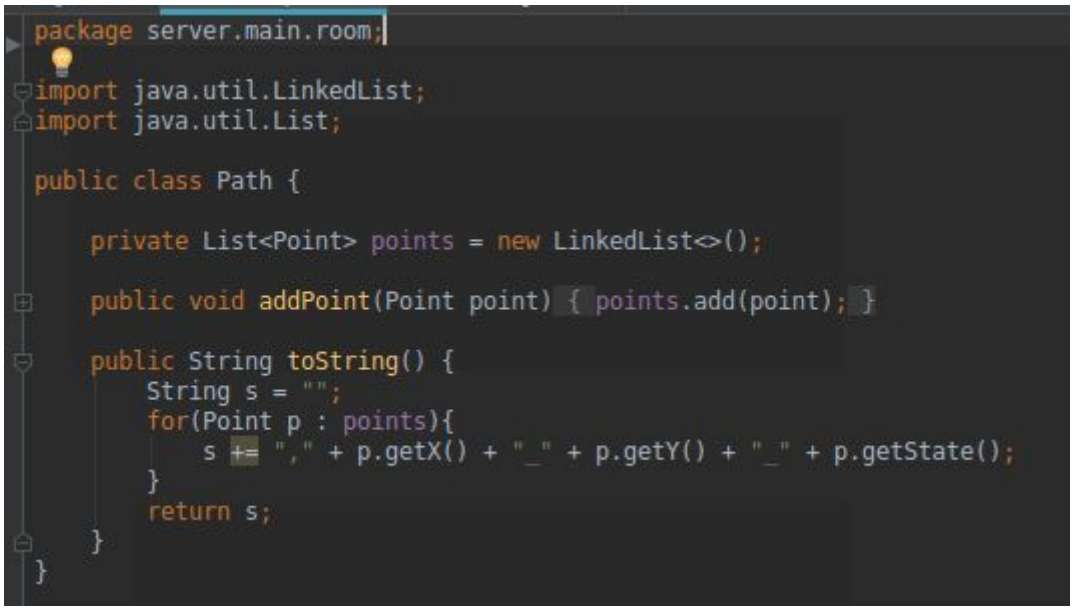
```
public void markSection(Point leftTop, Point rightBottom, Player player){
    for(int i=leftTop.getX(); i<= rightBottom.getX(); i++){
        for(int j=leftTop.getY(); j <= rightBottom.getY(); j++){
            board[i][j] = player.getId();
        }
    }
    System.out.println("LeftTop: x " + leftTop.getX() + " y: " + leftTop.getY());
    System.out.println("RightBottom: x " + rightBottom.getX() + " y: " + rightBottom.getY());
    System.out.println(player.getName());

}

public void refreshBoard(){
    for(int i = 0; i < this.width; i++){
        for(int j = 0; j< this.height; j++){
            board[i][j] = 0;
        }
    }
}
```

Dodatkowo w klasie Board znajdują się metody odpowiedzialne za powerUpy, których implementacji nie udało nam się dokończyć:

```java
public void cleanPowerUp(Point point, int size) {
    for(int i = point.getX(); i< point.getX()+size; i++){
        for(int j = point.getY(); i< point.getY()+size; j++){
            board[i][j] = 0;
        }
    }
}

public void drawPlayer(Point point, Player player){
    for(int i = point.getX(); i<point.getX()+player.getSize(); i++){
        for(int j = point.getY(); j<point.getY()+player.getSize(); j++){
            board[i][j] = player.getId();
        }
    }
}

public void addPowerUp(PowerUp powerUp) {
    int x = powerUp.getPosition().getX();
    int y = powerUp.getPosition().getY();
    int size = powerUp.getSize();
    switch(powerUp.getPowerUpKind()){
        case IMMORTALITY:
            drawRectangleOnBoard(x,y, size,  value: -1);
            break;
        case SPEEDDOWN:
            drawRectangleOnBoard(x,y, size,  value: -2);
            break;
        case SPEEDUP:
            drawRectangleOnBoard(x,y, size,  value: -3);
            break;
    }
}

private void drawRectangleOnBoard(int x,int y, int size, int value) {
    for(int i = x; i < x+size; i++){
        for(int j = y; j < y+size; j++){
            board[i][j] = value;
        }
    }
```

VI. Path

Klasa Path jest odpowiedzialna za tworzenie ścieżki gracza i robienie z niej stringa, który jest potem rozsyłany do graczy w postaci stanu gry.



```
package server.main.room;

import java.util.LinkedList;
import java.util.List;

public class Path {

    private List<Point> points = new LinkedList<>();

    public void addPoint(Point point) { points.add(point); }

    public String toString() {
        String s = "";
        for(Point p : points){
            s += "," + p.getX() + "_" + p.getY() + "_" + p.getState();
        }
        return s;
    }
}
```

VII. Point

Klas punktu - co tu dużo mówić, po prostu zwykły punkt:

```java
public class Point {

    private int x;
    private int y;
    private String state;

    public Point(int x, int y, String state){
        this.x = x;
        this.y = y;
        this.state = state;
    }

    public int getY() { return y; }

    public int getX() { return x; }

    public boolean equals(Object o) {
        if (o == null) {
            return false;
        }
        if (!(o instanceof Point)) {
            return false;
        }
        return (x == ((Point) o).x && y == ((Point) o).y);
    }

    public String getState() { return state; }
}
```

game

I. Spliter

Klasa odpowiedzialna za parsowanie wiadomości wysyłanych przez serwer do klienta (połączenie UDP). Wiadomość miała następującą formę:

id,nazwaGracza,kolorHexa,x_y_state,x2_y2_state2,
....;id2,nazwaGracza2,kolorHexa2,x_y_state,...:
xBonusu_yBonusu_bonus;....

Przykładowa wiadomość skopiowana w trakcie działania programu:

1,,#FF00FF,303_41_start,519_41_end;2,,#00FFFF,35_125_start,35_161_end,51_161_end,51_173_end,63_173_end,63_185_end,73_185_end,73_193_end,85_193_end,85_201_end,105_201_end,105_207_end,121_207_end,121_213_end,137_213_end,137_207_end,155_207_end,155_209_end;

Ze względu na brak czasu na zaimplementowanie do końca bonusów część wiadomości po : nie miała znaczenia, jednak parser poprawnie obsługiwał przewidzianą w przyszłości pełną wersję wiadomości.

Poniższa metoda odpowiadała za rozdzielenie informacji dotyczących tras wykonanych przez poszczególnych graczy na podstawie znaków specjalnych (przecinki, podkreślenia,

średniki). Zwracała ArrayListę zawierającą obiekt typu PlayerInfo opisany w kolejnym punkcie.

```
public ArrayList<PlayerInfo> parse(String msg){

    String [] colonSplittedString = msg.split(":");

    //~~~~~~~~~~~~~~~~~~~~~~~do : mamy colonSplittedString[0]~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    String [] semicolonSplittedString = colonSplittedString[0].split(";");

    ArrayList<PlayerInfo> playerInfoArrayList = new ArrayList<>();
    for (int i = 0; i < semicolonSplittedString.length; i++){
        playerInfoArrayList.add(new PlayerInfo());
    }

    for (int i = 0; i < semicolonSplittedString.length; i++){
        String [] commaSplittedString = semicolonSplittedString[i].split(",");

        playerInfoArrayList.get(i).setId(commaSplittedString[0]);
        playerInfoArrayList.get(i).setPlayer(commaSplittedString[1]);
        String color = commaSplittedString[2];
        Color parsedColor = Color.web(color);
        playerInfoArrayList.get(i).setColor(parsedColor);

        for (int j = 3; j < commaSplittedString.length; j++) {
            String[] pointSplittedString = commaSplittedString[j].split("_");

            PointPlayer point = new PointPlayer(Integer.parseInt(pointSplittedString[0]), Integer.parseInt(pointSplittedString[1]));
            Stage stage = Stage.valueOf(pointSplittedString[2]);

            playerInfoArrayList.get(i).getPoints().add(point);
            playerInfoArrayList.get(i).getStages().add(stage);
        }


    }
```

Druga część parsera rozdzielała informacje dotyczące bonusów.

```
public ArrayList<Bonus> parseBonus(String msg) {
    String [] colonSplittedString = msg.split(":");

    ArrayList<Bonus> bonusInfoArrayList = new ArrayList<>();
    String [] semicolonSplittedString = colonSplittedString[1].split(";");

    for (int i = 0; i < semicolonSplittedString.length; i++){
        bonusInfoArrayList.add(new Bonus());
    }

    for (int i = 0; i < semicolonSplittedString.length; i++) {
        String[] commaSplittedString = semicolonSplittedString[i].split("_");

        PointPlayer point = new PointPlayer(Integer.parseInt(commaSplittedString[0]), Integer.parseInt(commaSplittedString[1]));

        bonusInfoArrayList.get(i).setPoint(point);
        bonusInfoArrayList.get(i).setBonus(commaSplittedString[2]);


    }

      for (int i = 0; i < bonusInfoArrayList.size(); i++){
          System.out.println("X: " + bonusInfoArrayList.get(i).getPoint().getX());
          System.out.println("Y: " + bonusInfoArrayList.get(i).getPoint().getY());
          System.out.println("Bonus: " + bonusInfoArrayList.get(i).getBonus());
      }


    return bonusInfoArrayList;
}
```

II. PlayerInfo

W celu ułatwienia przechowywania informacji potrzebnych do rysowania tras graczy na planszy, utworzona została klasa PlayerInfo. Obiekty tej klasy posiadały id, nazwę gracza, kolor linii, tablice na miejsca skrętu (miejsca skrętu to punkty postaci x, y) oraz na stany niefortunnie nazwane “stages” w wyniku wady słuchu.

```
public class PlayerInfo {
    private String id;
    private String player;
    private Color color;
    private ArrayList<PointPlayer> points;
    private ArrayList<Stage> stages;


    public PlayerInfo() {
        this.points = new ArrayList<>();
        this.stages = new ArrayList<>();
    }
```

III. Bonus

Klasa odpowiedzialna za przechowywanie informacji o punkcie gdzie ma znaleźć się dany bonus oraz o rodzaju bonusu.

```
public class Bonus {
    private PointPlayer point; // = new ArrayList<>();

    private String bonus;
```

IV. Map

Klasa odpowiada za podstawowy widok planszy. Posiada setCanvas(), która ustawia wielkość planszy, kolor i rysuje na niej budynek D17 za pomocą funkcji drawD17().

```
public class Map {

    public Map(){}

    public Canvas setCanvas(){
        Canvas canvas = new Canvas( width: 600,  height: 440);
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.setFill(Color.BLACK);
        gc.fillRect( x: 0,  y: 0, canvas.getWidth(), canvas.getHeight());
        drawD17(gc);
        return canvas;
    }
```

```
    public void drawD17(GraphicsContext gc) {

        gc.setStroke(Color.WHITE);
        gc.setLineWidth(5.0);
        gc.beginPath();
        gc.lineTo( x1: 260,  y1: 305);
        gc.lineTo( x1: 260,  y1: 310);
        gc.lineTo( x1: 50,  y1: 250);
        gc.lineTo( x1: 50, y1: 200);
        gc.lineTo( x1: 50, y1: 200);
        gc.lineTo( x1: 260, y1: 220);
        gc.lineTo( x1: 260, y1: 240);
        gc.moveTo( x0: 300,  y0: 240);
        gc.lineTo( x1: 300,  y1: 230);
        gc.lineTo( x1: 400,  y1: 230);
        gc.lineTo( x1: 450,  y1: 350);
        gc.lineTo( x1: 300,  y1: 315);
        //gc.lineTo();
        gc.stroke();


    }
```

Funkcja niepodłączona
readingPixels() umożliwa czytanie białych pixeli z plaszy (tutaj obrys budynku D17), w celu stworzenia z nich “ścian” które w przypadku wjechania w nie powodują śmierć gracza.

```
public int[][] readingPixels(Canvas canvas, int[][] board) {
    SnapshotParameters params = new SnapshotParameters();
    params.setFill(Color.BLACK);

    PixelReader reader = canvas.snapshot(params, image: null).getPixelReader();

    double canvasWidth = canvas.getWidth();
    double canvasHeight = canvas.getHeight();
    System.out.println(canvasWidth + " " + canvasHeight);
    for (int x = 0; x < canvasWidth; x++)
        for (int y = 0; y < canvasHeight; y++) {
            Color col=reader.getColor(x, y);
            Color D17=Color.WHITE;
            if ( D17.equals(col)) board[x][y] =88;
            else board[x][y]=0;
        }

    return board;
}
```

V. DrawPixels

Odpowiada za narysowanie na planszy śladu drogi każdego gracza, używając danych przesyłanych od serwera, parsowanych za pomocą klasy Spliter.

VI. ControlsThread

Klasa odpowiada za czytanie klawiszy i przesyłanie wiadomości o ruchu gracza do serwera.

VII. MapReceiver

Klasa odpowiada za odbieranie wiadomości UDP wysyłanych przez serwer dotyczących aktualnych tras pokonanych przez graczy oraz położenia bonusów.

```java
@Override
public void run() {

	InetAddress group = null;
	MulticastSocket multicastSocket = null;
	byte[] buf = new byte[8192];

	try {
		multicastSocket = new MulticastSocket(port);
		group = InetAddress.getByName("224.0.113.0");
		multicastSocket.joinGroup(group);
	} catch (IOException e) {
		e.printStackTrace();
	}

	if(multicastSocket != null){
		while(true){
			try {
				DatagramPacket datagramPacket = new DatagramPacket(buf, buf.length);
				multicastSocket.receive(datagramPacket);
				String response = new String(datagramPacket.getData(), 0, datagramPacket.getLength());
				game.getCanvas(response);
				System.out.println(response);

			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}
}
```

gui

I. Menu

Klasa odpowiedzialna za wyświetlanie okienka Menu wraz z przyciskami: New Game, Instructions, Highscore oraz End Game, przenoszących do odpowiednich okienek.

II. IPServer

Klasa odpowiedzialna za wyświetlanie okienka IPServer, w którym podajemy IP naszego serwera, po czym otwierane jest okienko Login, w którym tworzony jest socket TCP do łączności z serwerem.

III. Login

Klasa odpowiedzialna za wyświetlanie okienka Login, w którym podajemy nazwę swojego Playera. Po wciśnięciu przycisku submit wysyłany jest do serwera komunikat “InitPlayer”.

```
public void sendName(TextField text){
    try {
        String imie = text.getText();
        String msg = "initPlayer ".concat(" ".concat(imie));

        //
            if (imie.equals("")) {
                AlertView alert = new AlertView(owner,  text: "Please enter your name!");

            } else {
                out.println(msg);

                String response = null;
                response = in.readLine();
                System.out.println(response);

                if(response.contains("init OK")) {
                    owner.close();
                    String rooms = null;
                    rooms = setRooms();
                    RoomsView pokoje = new RoomsView(rooms, socket);
                    pokoje.showRoomsView();
                }
            }

    } catch (IOException e1) {
        e1.printStackTrace();
    }
}
```

Przed stworzeniem okienka RoomsView jest wysyłana do serwera wiadomość “roomList”, po czym odbieramy wiadomość zawierającą informację na temat pokojów utworzonych wcześniej. Ma ona postać: *nazwa_pokoju1, liczba_osob_zapisanych_do_pokoju1, max_liczba_osob_w_ pokoju1; nazwa_pokoju2, liczba_osob_zapisanych_do_pokoju2, max_liczba_osob_w_ pokoju2; …,* która jest następnie przekazywana jako argument do funkcji tworzącej RoomsView.

```
public String setRooms(){
    out.println("roomList");
    String line = null;
    while(line == null) {
        try {
            line = in.readLine();
        } catch (IOException e1) {

        }
    }
    System.out.println(line);
    return line;
}
```

IV. RoomsView

Klasa odpowiedzialna za wyświetlanie okienka RoomsView, w którym wybieramy pokój, do którego chcemy dołączyć. Przy tworzeniu okienka na podstawie Stringu wcześniej uzyskanego od serwera a przekazanego do konstruktora tej klasy jako argument, tworzona jest lista pokoi wcześniej utworzonych w systemie.

```
public ListView<String[]> setList() throws IOException {
    ListView<String[]> list = new ListView<>();

    items = FXCollections.observableArrayList();
    items = createList(items);
    list.setItems(items);

    list.setCellFactory(param -> updateItem(item, empty) -> {
            super.updateItem(item, empty);

            if (empty || item == null || item[0] == null) {
                setText(null);

            } else {
                setText(item[0]);

            }
    });

    list.getSelectionModel().selectedItemProperty()
            .addListener(new ChangeListener<String[]>() {
                @Override
                public void changed(ObservableValue<? extends String[]> observable,
                                    String[] oldValue, String[] newValue) {
                    printRow(newValue);
                }
            });

    return list;
}
```

```
public ObservableList<String[]> createList(ObservableList<String[]> items)

    String[] rooms = line.split( regex: ";");
    int i = rooms.length, m = 0;
    String[][] roomsInfo = new String[i][3];
    for(String x: rooms){
        String[] tmp = x.split( regex: ",");
        roomsInfo[m] = tmp;
        m++;
    }

    for(String[] room: roomsInfo){
        items.add(room);
    }

    return items;
}
```

Istnieje możliwość odświeżenia tej listy za pomocą przycisku “Refresh”. W tym przypadku ponownie wysyłana jest wiadomość “RoomList” do serwera, oraz aktualizowana jest lista. W przypadku wybrania pokoju, informacje na jego temat ukazują się w prawej górnej części okienka. Jest to możliwe dzięki tej funkcji:

```
public void printRow(String[] row){
    nazwa.setText("Name: "+ row[0]);
    nazwa.setTextFill(Color.WHITE);
    nazwa.setStyle("-fx-font-size: 24;");
    ilosc.setText("Players: " + row[1] + "/" + row[2] );
    ilosc.setTextFill(Color.WHITE);
    ilosc.setStyle("-fx-font-size: 24;");
```

W przypadku wybrania pokoju i wciśnięciu przycisku “Choose” wysyłana do serwera jest wiadomość “joinServer”. W przypadku odebraniu wiadomości zwrotnej zawierającej “success” wywoływana jest funkcja ProgreesMaking(), która tworzy okienko Waiting do czasu uzyskania od serwera wiadomości “StartGame”. Wtedy to następuję zamknięcie obu okienek oraz otworzenie okienka Game.

```
public void ProgressMaking() throws IOException {
    Waiting pForm = new Waiting();
    pForm.Waiting(socket);
    Task<Void> task = () -> {
            //while(response.equals(null)) {
                String response = in.readLine();
                if (response.contains("startGame")) {
                    return null;
                }
            //}
        /*   for(int i = 0; i < 10000; i++){
                System.out.println(i);
            }
            */
            return null ;
    };
    pForm.activateProgressBar(task);
    task.setOnSucceeded(event -> {
        pForm.getDialogStage().close();
        owner.close();
        Game actualGame = null;
        try {
            actualGame = new Game(socket);
        } catch (IOException e) {
            e.printStackTrace();
        }
        actualGame.showActualGame();
    });
    pForm.getDialogStage().show();

    Thread thread = new Thread(task);
    thread.start();
}
```

Istnieje również możliwość stworzenia nowego pokoju, który będzie później pojawiał się na liście dostępnych pokoi opisanej wcześniej. Można tego dokonać poprzez wciśnięcie przycisku “+”. Następuje wtedy otwarcie okienka NewRoom.

V. NewRoom

Klasa odpowiedzialna za wyświetlanie okienka NewRoom, w którym podaje dane nowego pokoju, a następnie ten pokój tworzymy. Po wpisaniu do TextField nazwy pokoju oraz wybraniu liczby maksymalnej liczby playerów w pokoju z ChoiceBox i naciśnięciu przycisku “CreateRoom” wywoływana jest funkcja SendRoom()

```
public void sendRoom(Stage rooms){
    if (nameField.getText().equals("")) {
        AlertView alert = new AlertView(owner,  text: "Please enter room name!");
    }
    else {
        out.println("hostRoom " + nameField.getText() + " " + cb.getSelectionModel().getSelectedItem());
        String msg = null;
        try {
            msg = in.readLine();
        } catch (IOException e1) {
            e1.printStackTrace();
        }
        if (msg.contains("success")) {
            try {
                ProgressMaking(rooms);
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            AlertView alert = new AlertView(owner,  text: "Shit got down");
        }
        owner.close();
    }
}
```

Wysyła ona do serwera wiadomość “hostRoom” wraz z informacjami o pokoju wybranymi przez nas. Jeśli odpowiedź zwrotna zawiera “success” wywoływana jest funkcja ProgressMaking() analogiczna do funkcji o tej samej nazwie w klasie RoomsView. W przypadku innej odpowiedzi zwrotnej tworzony jest AlertView z odpowiednim tekstem.

VI. Game

Klasa odpowiedzialna za wyświetlanie okienka Game, odpowiedzialnego za wyświetlanie przebiegu naszej gry. Jest to zrealizowane za pomocą obiektu typu Canvas, który później w miarę napływania informacji z serwera o kolejnych ruchach playerów jest aktualizowany.

```
public void showActualGame(){
    TCPHandler tcpHandler = new TCPHandler(scene, socket);
    Thread tcpThread = new Thread(tcpHandler);
    tcpThread.start();

    final ImageView imv = new ImageView();
    final Image image2 = new Image(Main.class.getResourceAsStream( name: "stylesheets/images/logo.png"));
    imv.setImage(image2);


   canvas= initCanvas();

    HBox hbox = setButtonHbox();
    hbox.setAlignment(Pos.BOTTOM_CENTER);
    root.getChildren().addAll(imv,canvas, hbox);

    MapReceiver mapReceiver = new MapReceiver( game: this);
    Thread thread = new Thread(mapReceiver);
    thread.start();
    // actualGame();
}
```

Okienko posiada dwa przyciski: “End Game” otwierające okienko GameOver oraz “Return” otwierające okienko Menu.

Funkcje odpowiedzialne za aktualizowanie obiektu typu Canvas:

```
public Canvas initCanvas(){
    Map map =new Map();
   final Canvas canvas = map.setCanvas();
    return  canvas;
}

public Canvas getCanvas(String data){

    DrawPixels drawPixels = new DrawPixels();
    canvas = drawPixels.setRoad(data,canvas);

    return  canvas;
}
```

VII. GameOver

Klasa odpowiedzialna za wyświetlanie okienka GameOver. Okienko zawiera dwa przyciski: “Menu” otwierające okienko Menu oraz “ChangeRoom” otwierające okienko RoomsView.

VIII. Instruction

Klasa odpowiedzialna za wyświetlanie okienka Instruction, w którym ma być wyświetlana instrukcja.

IX. Highscore

Klasa odpowiedzialna za wyświetlanie okienka Highscore, w którym mają być wyświetlane najlepsze wyniki zdobyte podczas rozgrywki.

X. AlertView

Klasa odpowiedzialna za wyświetlanie okienka AlertView z tekstem podanym jako argument w funkcji wywołania.

```
public AlertView(Stage owner, String text){
    Alert.AlertType type  = Alert.AlertType.INFORMATION;
    Alert alert = new Alert(type,  contentText: "");
    alert.initModality(Modality.APPLICATION_MODAL);
    alert.initOwner(owner);
    alert.getDialogPane().setContentText(text);
    alert.getDialogPane().setHeaderText(null);
    alert.showAndWait()
            .filter(response -> response == ButtonType.OK)
            .ifPresent(response -> System.out.println("The alert was approved"));
```

XI. Waiting

Klasa odpowiedzialna za wyświetlanie okienka Waiting z animowanym ProgressIndicator zbindowanym do taska podanego w funkcji wywołania.

Css

Do zaprogramowania wyglądu okienek użyliśmy Cascade Style Sheets. Zostały one dodane do Gui za pomocą instrukcji getStylesheets. Tak wygląda na przykładzie okienka menu:

```
public Menu(){
    new JFXPanel();
    owner = new Stage(StageStyle.DECORATED);
    root = new VBox();
    scene = new Scene(root, widthScene, heightScene);
    scene.getStylesheets().add
            (Menu.class.getResource( s: "stylesheets/default.css").toExternalForm());
    scene.getStylesheets().add
            (Menu.class.getResource( s: "stylesheets/menu.css").toExternalForm());
    setStageProperty();
    setHBoxProperty();
}
```

I. Plik default

W pliku default.css mamy domyślny wygląd przycisków oraz tekstu. Przyciski są przezroczyste z białą ramką i białym tekstem, po najechaniu zmieniają się na białe z czarnym tekstem. Tekst natomiast domyślnie jest biały. Mamy tu także dodaną czcionkę ARCADECLASSIC której używamy w całej aplikacji. W pliku default również określamy tło dla wszystkich okienek w aplikacji

```css
@font-face {
    -fx-font-family: 'ARCADECLASSIC';
    src: url('ARCADECLASSIC.ttf');
}

.root {
    -fx-background-image: url("images/space.png");
}

.text{
    -fx-text-fill: white;
    -fx-font-family: "ARCADECLASSIC";
}
.label{
    -fx-font-family: "ARCADECLASSIC";
    -fx-alignment: top-center;
    -fx-text-fill: white;
}
.button{
    -fx-background-color: transparent;
    -fx-font-size: 30px;
    -fx-border-color: white;
    -fx-text-fill: white;
}
.button:hover{
    -fx-background-color: white;
    -fx-text-fill: black;
}
```

Kolejne pliki są do nadpisywania niektórych właściwości z pliku default oraz do określania cech obiektów które pojawiają się tylko w danych okienkach.

II. gameOver.css

W pliku gameOver.css mamy zmienione kolory przycisków na czerwony i zielony

III. gameView.css

W pliku gameView.css mamy jedynie zaprogramowany odstęp pomiędzy przyciskami

IV. highscore.css

W pliku highscore.css mamy określony wygląd tabeli wyników za pomocą obiektu table-view oraz zmianę wielkości czcionki dla tytułu okienka.

V. instrucions.css

W plik instrucions.css jest pusty, stworzony do modyfikacji tekstu i obrazów dla instrukcji gry.

VI. login.css

W pliku login.css mamy zaprogramowany wygląd pola do wpisywania imienia za pomocą obiektu text-field

VII. menu.css

W pliku menu.css mamy nadpisaną wielkość przycisków oraz dla każdego mamy zmieniony kolor co widać na zdjęciu poniżej. Nazwy przycisków np. #score są przypisywane w klasie menu do obiektu button za pomocą setId().

```
    -fx-text-fill: #ffe136;
}
#play:hover{
    -fx-background-color: #ffe136;
    -fx-text-fill: black;
}

#insrtuction{
    -fx-border-color: #328fff;
    -fx-text-fill: #328fff;
}

#insrtuction:hover{
    -fx-background-color: #328fff;
    -fx-text-fill: black;
}

#score{
    -fx-border-color: #72ff4b;
    -fx-text-fill: #72ff4b;
}

#score:hover{
    -fx-background-color: #72ff4b;
    -fx-text-fill: black;
}

#end{
    -fx-border-color: #ff0b0b;
    -fx-text-fill: #ff0b0b;
}
```

```
Button highBttn = new Button( text: "HIGHSCORE");
highBttn.setId("score");
highBttn.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent e) {
        Highscore highscore = new Highscore();
        try {
            highscore.showHighscore();
        } catch (IOException e1) {
            e1.printStackTrace();
        }

        owner.close();
    }
});

Button endGame = new Button( text: "QUIT");
endGame.setId("end");
endGame.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent e) { owner.close(); }
```

VIII. newRoom.css

W pliku newRoom.css mamy zaprogramowany wygląd obiektu choice-box w którym wybieramy ilość graczy oraz obiekt text-field.

IX. roomsView.css

W pliku roomsView.css programujemy wygląd listy pokoi przy użyciu list-cell oraz zmianę koloru przycisku newRoom który służy do dodania nowego pokoju i z tego powodu jego kolor został ustawiony na czerwony
Ostatnim plikiem jest plik Waiting.css w który została zmieniona jedynie wielkość czcionki