



Państwowa Wyższa Szkoła Zawodowa w Tarnowie
Wydział Politechniczny
Informatyka

Projekt - Technologie obiektowe i komponentowe

Gra logiczna „Statki kosmiczne”

Wykonawcy:

Ernest Bieś

Konrad Czechowski

Dawid Kwaśny

Prowadzący:

mgr inż. Rafał Jędryka

Tarnów, 2020

Spis treści

1. Opis projektu gry.....	3
2. Zasady gry.....	3
3. Założenia projektowe.....	4
3.1. Architektura projektu.....	4
3.2. Zapytania HTTP obsługiwane przez serwer.....	5
4. Struktura projektu.....	8
4.1. Struktura projektu – serwer.....	9
4.2. Struktura projektu – klient.....	32
4.3. Algorytm generowania rozmieszczenia statków.....	35
5. Lista wykorzystanych technologii oraz bibliotek.....	39
6. Określenie scenariuszy użytkownika i przypadków użycia.....	41
7. Identyfikacja funkcji.....	42
8. Diagramy UML.....	45
8.1. Diagramy aktywności.....	45
8.2. Diagramy klas.....	49
8.3. Diagramy przypadków użycia.....	50
8.4. Diagramy sekwencji.....	51
8.5. Diagramy stanów.....	53
8.6. Diagramy pakietów.....	54
9. Opis klienta.....	55
9.1. Środowisko programistyczne.....	55
9.2. Wygląd interfejsu graficznego klienta.....	55
9.3. Instrukcja obsługi.....	58
10. Proces debugowania.....	59

1. Opis projektu gry

Celem projektu było stworzenie gry logicznej **"Statki kosmiczne"** będącej połączeniem dwóch popularnych gier logicznych "Statki" oraz "Saper" w oparciu o architekturę **klient-serwer**.

Gra „Statki kosmiczne” składa się z dwóch aplikacji klienta i serwera. Wszystkie operacje związane z utworzeniem gier użytkowników, zapamiętaniem ich stanów, generowaniem rozmieszczenia statków na planszy są realizowane przez serwer gry. Natomiast klient pełni jedynie funkcję graficznej interpretacji informacji uzyskanych z serwera. Stany gier użytkowników oraz zakończone gry, na podstawie których tworzony zostaje ranking, zapisywane są w bazie danych. Komunikacja między serwerem, a klientem realizowana jest przy wykorzystaniu zapytań REST. Każdy ruch gracza jest monitorowany na serwerze.

2. Zasady gry

Założeniem gry jest odnalezienie wszystkich umieszczonych na planszy statków w jak najmniejszej liczbie kroków, korzystając z informacji ile statków znajduje się w pobliżu odsłoniętego pola na planszy. Po rozpoczęciu gry na planszy o wymiarach 9x9 utworzonej z kwadratów są losowo rozmieszczone statki kosmiczne o różnych wielkościach:

Dwa Transportowce o wymiarze 4

Trzy Samoloty kosmiczne o wymiarze 3

Trzy Wahadłowce o wymiarze 2

Dwa Szturmowce o wymiarze 1

Statki nie są widoczne dla gracza. W przypadku odkrycia pola, które nie zawiera żadnego statku widoczna jest liczba, która wskazuje na ilu polach sąsiadujących z odkrytym polem znajdują się statki. Wobec powyższego gracz musi podejmować swoją decyzję dotyczącą pola, które w następnej kolejności odkryje na podstawie logicznego myślenia, gdyż tym samym zwiększa swoje szanse na odnalezienie statków w pobliżu.

3. Założenia projektowe

3.1. Architektura projektu

Projekt opiera się na architekturze **klient-serwer**.

Klient:

Zadaniem klienta jest wyświetlanie aktualnej planszy gry w statki, liczbę sprawdzonych pól planszy (kroków), wybór pola do sprawdzenia przez gracza oraz przesyłanie informacji dotyczących do serwera w celu sprawdzenia, czy na wskazanym polu znajduje się statek, czy też pole jest puste (tzw. pudło). Klient przesyła również informacje dotyczące użytkownika, który uruchomił grę oraz pobiera stan gry danego użytkownika. Klient pełni funkcję graficznej interpretacji informacji uzyskanych z serwera.

Serwer:

Komunikacja między serwerem, a klientem jest realizowana przy wykorzystaniu zapytań REST. Serwer generuje nową grę dla użytkownika o podanej nazwie, przechowuje stany gier wszystkich użytkowników, generuje losowy rozkład statków na planszy korzystając z odpowiedniego algorytmu dla nowej gry, sprawdza pole planszy wskazane przez użytkownika, celem ustalenia jaki statek został trafiony i przesyła tę informację do klienta, aktualizuje stany gier dla wszystkich użytkowników. Przesyła do klienta status gry danego użytkownika.

Baza danych:

W bazie danych przechowywane są wszystkie stany gier użytkowników. Do poprawnego działania aplikacji wymagany jest serwer bazodanowy PostgreSQL. Aplikacja aktualnie działa na serwerze zewnętrznym. Informacje o połączeniu z serwerem zawarte są w pliku **resources/application.properties**.

Aktualna konfiguracja bazy danych:

```
#PostgreSQL
spring.datasource.platform=postgres
spring.datasource.url=jdbc:postgresql://techify.tk:59453/spaceships
spring.datasource.username=spaceships
spring.datasource.password=spaceships
```

3.2. Zapytania HTTP obsługiwane przez serwer

Serwer obsługuje następujące zapytania HTTP:

URL USŁUGI 1	127.0.0.1:8080/api/newgame
TYP ŻĄDANIA	POST
PARAMETRY WEJŚCIOWE [PARAMETR]	user, pass
PARAMETRY WYJŚCIOWE [JSON]	{ „steps”:0, „code”:„NEWGAME”, „shipName”:”, „board”:”, „masts”:0, }
KOD ODPOWIEDZI SERWERA	200 – utworzono nową grę użytkownika user 403 – nieprawidłowe dane logowania
KOMENTARZ	user – nazwa użytkownika pass – hash hasła użytkownika Utworzenie nowej gry użytkownika user „code” - kod stanu gry: - „NEWGAME” - utworzono nową grę użytkownika

URL USŁUGI 2	127.0.0.1:8080/api/getgame
TYP ŻĄDANIA	POST
PARAMETRY WEJŚCIOWE [PARAMETR]	user, pass
PARAMETRY WYJŚCIOWE [JSON]	{ „steps”:„8”, „code”:„LOADGAME”, „shipName”:”, „board”:„X X X X 3 X X”, „type”:0, }
KOD ODPOWIEDZI SERWERA	200 – OK 403 – nieprawidłowe dane logowania
KOMENTARZ	user – nazwa użytkownika pass – hash hasła użytkownika Wczytanie stanu gry użytkownika user

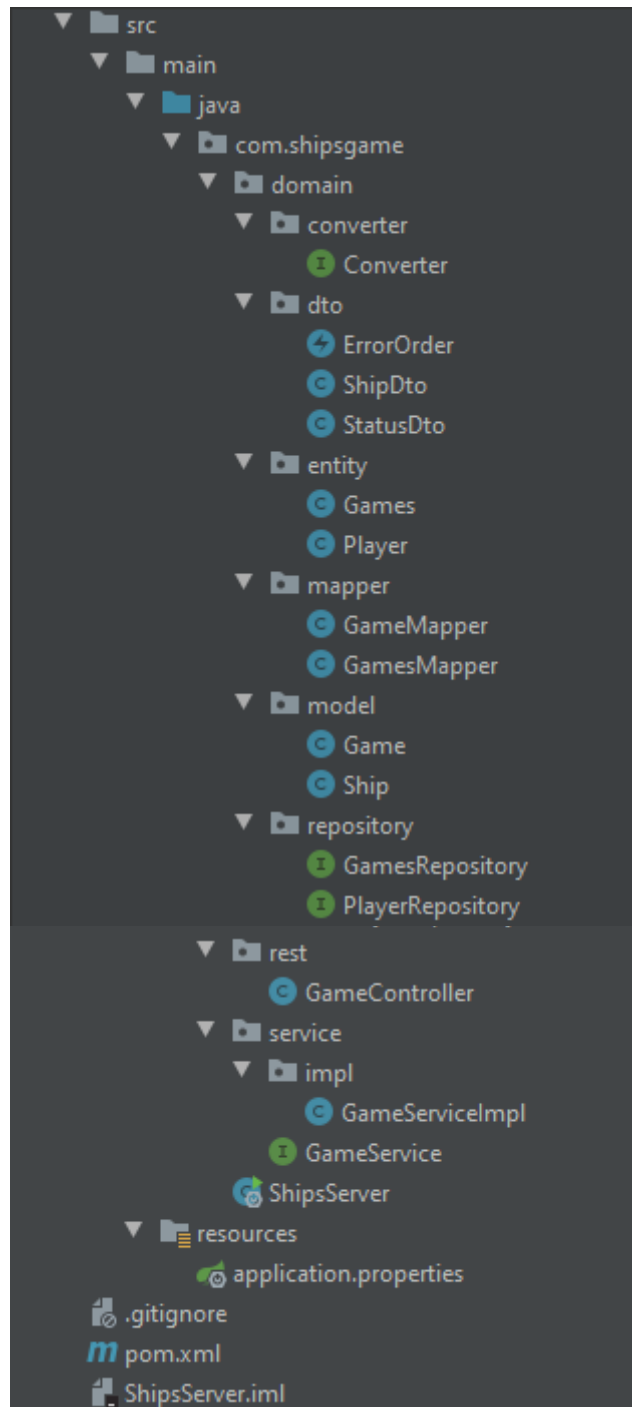
	„steps” - liczba kroków „code” - kod stanu gry: - „LOADGAME” – gra została wczytana „shipName” - nazwa trafionego statku „board” - stan planszy „type” - liczba masztów trafionego statku
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

URL USŁUGI 3	127.0.0.1:8080/api/shotGame
TYP ŻĄDANIA	POST
PARAMETRY WEJŚCIOWE [PARAMETR]	user, pass, shot
PARAMETRY WYJŚCIOWE [JSON]	<pre>{ „steps”:11, „code”:”HIT”, „shipName”:”Trójmasztowiec”, „board”:”X X X 3 X X”, „type”:3, }</pre>
KOD ODPOWIEDZI SERWERA	200 – OK 403 – nieprawidłowe dane logowania 404 – niepoprawny strzał
KOMENTARZ	user – nazwa użytkownika pass – hash hasła użytkownika shot – pozycja do sprawdzenia na planszy „code” - kod stanu gry: - „HIT” - pozycja trafiona - „MISS” - brak trafienia - „CHECKED” - pole ponownie sprawdzane - „SHOTDOWN” - statek zestrzelony - „ENDGAME” - wszystkie statki zestrzelone (koniec gry)

URL USŁUGI 4	127.0.0.1:8080/api/login
TYP ŻĄDANIA	POST
PARAMETRY WEJŚCIOWE [PARAMETR]	user, pass
PARAMETRY WYJŚCIOWE [STRING]	true
KOD ODPOWIEDZI SERWERA	200 - OK 401 – nieprawidłowe dane logowania
KOMENTARZ	user – nazwa użytkownika pass – hash hasła użytkownika

URL USŁUGI 5	127.0.0.1:8080/api/getrank
TYP ŻĄDANIA	GET
PARAMETRY WEJŚCIOWE [PARAMETR]	-
PARAMETRY WYJŚCIOWE [STRING]	"user 40 user1 43"
KOD ODPOWIEDZI SERWERA	200 – OK 404 – brak zasobów do wyświetlenia
KOMENTARZ	name – nazwa użytkownika score – wynik

4. Struktura projektu



Rys. Struktura projektu – serwer

4.1. Struktura projektu – serwer

Pakiet: domain

Pakiety wchodzące w skład pakietu: converter, dto, entity, mapper, model, repository

Pakiet: converter

Interfejs: Converter

Opis: Interfejs dla mapperów

Pakiet: dto

Klasy: ErrorOrder, ShipDto, StatusDto

Opis: Data Transfer Object (DTO) – klasy odpowiedzialne za przesyłanie obiektów.

Pakiet: entity

Klasy: Games, Player

Opis: Encje przechowywane w bazie danych.

Pakiet: mapper

Klasy: GameMapper, GamesMapper

Opis: Mapperzy, które odpowiedzialne są za konwersję (mapper GameMapper – konwersja z Games na Game, mapper GamesMapper – konwersja z Game na Games).

Pakiet: model

Klasy: Game, Ship

Opis: Modele wykorzystywane podczas gry.

Pakiet: repository

Klasy: GamesRepository, PlayerRepository

Opis: GamesRepository – pozwala na wykonywanie zapytań SQL z wykorzystaniem encji Games. PlayerRepository – pozwala na wykonywanie zapytań SQL z wykorzystaniem encji Player.

Pakiet: rest

Klasy: GameController

Opis: Obsługa zapytań HTTP.

Pakiet: service

Klasy: GameService, impl/GameServiceImpl

Opis: Logika biznesowa.

4.1.1. Klasa ShipsServer

Główna klasa serwera zawierająca metodę main().

Kod źródłowy:

```
package com.shipsgame;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ShipsServer {

    public static void main(String[] args) {
        SpringApplication.run(ShipsServer.class, args);
    }

}
```

4.1.2. Interfejs Converter

Interfejs Converter to Interfejs dla mapperów.

Kod źródłowy:

```
package com.shipsgame.domain.converter;

@FunctionalInterface
public interface Converter<F, T> {
    T convert(F from);
}
```

4.1.3. Klasa ErrorOrder

Klasa odpowiedzialna za obsługę błędów.

Kod źródłowy:

```
package com.shipsgame.domain.dto;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(value= HttpStatus.NOT_FOUND, reason="Error order")
```

```
public class ErrorOrder extends RuntimeException {
    public ErrorOrder(String message){
        super(message);
    }
}
```

4.1.3. Klasa ShipDto

Klasa odpowiedzialna za przesyłanie obiektów (DTO).

Kod źródłowy:

```
package com.shipsgame.domain.dto;

import java.io.Serializable;

public class ShipDto implements Serializable {
    private static final long serialVersionUID = 100L;

    private Integer type;

    private String name;

    private String[] positions;

    public ShipDto() {
    }

    public ShipDto(Builder builder) {
        this.name = builder.name;
        this.type = builder.type;
        this.positions = builder.positions;
    }

    public String[] getPositions() {
        return positions;
    }

    public Integer getType() {
        return type;
    }

    public String getName() {
        return name;
    }

    public static final class Builder {
        private Integer type;
        private String name;
        private String[] positions;
    }
}
```

```

    public Builder() {
    }

    public Builder type(Integer type) {
        this.type = type;
        return this;
    }

    public Builder name(String name) {
        this.name = name;
        return this;
    }

    public Builder positions(String[] positions) {
        this.positions = positions;
        return this;
    }

    public ShipDto build() {
        return new ShipDto(this);
    }
}

```

4.1.4. Klasa StatusDto

Klasa StatusDto wykorzystywana jest do przesyłania informacji z serwera do klienta. Zawiera informacje dotyczące liczby wykonanych przez użytkownika kroków (steps), nazwę kodu wykonanych operacji lub stanu gry (code), nazwę statku kosmicznego (shipName) na którym była wykonana operacja oraz jego typ (type), a także przechowuje stan planszy (board).

Kod źródłowy:

```

package com.shipsgame.domain.dto;

public class StatusDto {
    private int steps; //number of user steps
    private String code; //code of game (for e.g. new game, end game)
    private String shipName; //name of the spaceship
    private String board; //current board status
    private int type; //type of spaceship

    public StatusDto() {
    }
}

```

```

public StatusDto(String code, String shipName, int type, int steps, String board){
    this.code = code;
    this.shipName = shipName;
    this.type = type;
    this.steps = steps;
    this.board = board;
}

public StatusDto(Builder builder){
    this.code = builder.code;
    this.shipName = builder.shipName;
    this.type = builder.type;
    this.steps = builder.steps;
    this.board = builder.board;
}

public int getSteps() {
    return steps;
}

public String getCode() {
    return code;
}

public String getShipName() {
    return shipName;
}

public int getType() {
    return type;
}

public String getBoard() {
    return board;
}

public static final class Builder {
    private String code;
    private String shipName;
    private Integer type;
    private Integer steps;
    private String board;

    public Builder(){
    }

    public Builder code(String code) {
        this.code = code;
        return this;
    }
}

```

```

        public Builder shipName(String shipName) {
            this.shipName = shipName;
            return this;
        }

        public Builder type(Integer type) {
            this.type = type;
            return this;
        }

        public Builder steps(Integer steps) {
            this.steps = steps;
            return this;
        }

        public Builder board(String board) {
            this.board = board;
            return this;
        }

        public StatusDto build() {
            return new StatusDto(this);
        }
    }
}

```

4.1.5. Klasa Games

Encja przechowywana w bazie danych reprezentująca gry danego gracza.

Kod źródłowy:

```

package com.shipsgame.domain.entity;

import com.shipsgame.domain.dto.ShipDto;

import javax.persistence.Entity;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class Games implements Serializable {
    private static final long serialVersionUID = 100L;
    @Id
    private String login;

```

```

private Integer steps;

private char[] board;

private ShipDto[] shipsList;

private int[][] boardNumbers;

public Games(){
}

private Games(Builder builder) {
    this.login = builder.login;
    this.steps = builder.steps;
    this.board = builder.board;
    this.shipsList = builder.shipsList;
    this.boardNumbers = builder.boardNumbers;
}

public String getLogin() {
    return login;
}

public Integer getSteps() {
    return steps;
}

public char[] getBoard() {
    return board;
}

public ShipDto[] getShipsList() {
    return shipsList;
}

public int[][] getBoardNumbers() {
    return boardNumbers;
}

public static final class Builder {
    private String login;
    private Integer steps;
    private char[] board;
    private ShipDto[] shipsList;
    private int[][] boardNumbers;

    public Builder(){
    }

    public Builder login(String login) {

```

```

        this.login = login;
        return this;
    }

    public Builder steps(Integer steps) {
        this.steps = steps;
        return this;
    }

    public Builder board(char[] board) {
        this.board = board;
        return this;
    }

    public Builder shipsList(ShipDto[] shipsList) {
        this.shipsList = shipsList;
        return this;
    }

    public Builder boardNumbers(int[][] boardNumbers) {
        this.boardNumbers = boardNumbers;
        return this;
    }

    public Games build() {
        return new Games(this);
    }
}
}

```

4.1.6. Klasa Player

Encja przechowywana w bazie danych reprezentująca danego gracza.

Kod źródłowy:

```

package com.shipsgame.domain.entity;

import javax.persistence.Entity;
import javax.persistence.Id;
import java.io.Serializable;

@Entity
public class Player implements Serializable {
    private static final long serialVersionUID = 100L;
    @Id
    private String login;
}

```



```

private String password;

private Integer bestScore;

public Player(){
}

public Player(Builder builder) {
    this.login = builder.login;
    this.password = builder.password;
    this.bestScore = builder.bestScore;
}

public String getLogin() {
    return login;
}

public String getPassword() {
    return password;
}

public Integer getBestScore() {
    return bestScore;
}

public static final class Builder {
    private String login;
    private String password;
    private Integer bestScore;

    public Builder() {
    }

    public Builder login(String login) {
        this.login = login;
        return this;
    }

    public Builder password(String password) {
        this.password = password;
        return this;
    }

    public Builder bestScore(Integer bestScore) {
        this.bestScore = bestScore;
        return this;
    }

    public Player build() {
        return new Player(this);
    }
}

```

```
    }  
}  
}
```

4.1.7. Klasa GameMapper

Mapper GameMapper jest to mapper odpowiedzialny za konwersję z Games na Game.

Kod źródłowy:

```
package com.shipsgame.domain.mapper;  
  
import com.shipsgame.domain.converter.Converter;  
import com.shipsgame.domain.dto.ShipDto;  
import com.shipsgame.domain.entity.Games;  
import com.shipsgame.domain.model.Game;  
import com.shipsgame.domain.model.Ship;  
import org.springframework.stereotype.Component;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
  
@Component  
public class GameMapper implements Converter<Games, Game> {  
  
    public Game convert(Games from) {  
        ShipDto[] fromShipsList = from.getShipsList();  
        List<Ship> ships = Arrays.asList(fromShipsList).stream()  
            .map(s -> {  
                String[] pos = s.getPositions();  
                List<String> position = new ArrayList<>();  
                position.addAll(Arrays.asList(pos));  
                return new Ship(s.getName(), s.getType(), position);  
            })  
            .collect(Collectors.toList());  
        return new Game(from.getLogin(),  
            from.getSteps(),  
            from.getBoard(),  
            ships,  
            from.getBoardNumbers());  
    }  
}
```

4.1.8. Klasa GamesMapper

Mapper GamesMapper jest to mapper odpowiedzialny za konwersję z Game na Games.

Kod źródłowy:

```
package com.shipsgame.domain.mapper;

import com.shipsgame.domain.converter.Converter;
import com.shipsgame.domain.dto.ShipDto;
import com.shipsgame.domain.entity.Games;
import com.shipsgame.domain.model.Game;
import com.shipsgame.domain.model.Ship;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class GamesMapper implements Converter<Game, Games> {
    @Override
    public Games convert(Game from) {
        List<Ship> fromShipsList = from.getShipsList();
        ShipDto[] ships = new ShipDto[fromShipsList.size()];
        int i=0;
        for(Ship s : fromShipsList) {
            List<String> pos = s.getPositions();
            String[] positions = new String[pos.size()];
            for(int j=0; j<pos.size(); j++) {
                positions[j] = pos.get(j);
            }
            ships[i++] = new ShipDto.Builder()
                .type(s.getType())
                .name(s.getName())
                .positions(positions)
                .build();
        }
        return new Games.Builder()
            .login(from.getUser())
            .board(from.getBoard())
            .boardNumbers(from.getBoardNumbers())
            .steps(from.getSteps())
            .shipsList(ships)
            .build();
    }
}
```

4.1.9. Klasa Game

Klasa przechowująca informacje dotyczące gry użytkownika, w tym liczbę wykonanych kroków (steps), nazwę użytkownika (user), aktualny stan planszy (board), listę statków (shipsList), a także tablicę (boardNumbers) przechowującą informacje dotyczące liczby statków znajdujących się w sąsiedztwie danego pola.

boardGenerating()

Metoda generująca ułożenie statków na planszy: 2 x "Transportowiec", 3 x "Samolot kosmiczny", 3 x "Wahadłowiec", 2 x "Szturmowiec". Generowanie rozmieszczenia statków odbywa się na podstawie następującego algorytmu. Dla każdego statku losowane jest pole od którego będzie rozpoczynał się dany statek, następnie losowo wybierana jest orientacja pionowa lub pozioma. Kolejnym krokiem jest sprawdzenie, czy kolejne pola na których ma zostać umieszczony statek są puste i czy sąsiednie pola również są puste (statki nie mogą się stykać). Jeżeli te warunki będą spełnione do tablicy danego statku są wpisywane pozycje pól na których zostanie umieszczony. Jeżeli natomiast którykolwiek z warunków nie będzie spełniony to algorytm ponownie losowo wybiera inne pole. Cały ten proces trwa do momentu, aż wszystkie statki zostaną właściwie ustawione na planszy. Następnie dla każdego pola na którym nie został umieszczony statek jest obliczana wartość i zapisywana do tablicy boardNumbers, która wskazuje na ilu polach sąsiadujących są umieszczone statki.

shot(String position)

Metoda sprawdzająca za pomocą zmiennej position pole na planszy i zwracająca w klasie StatusDto wynik tego sprawdzenia:

"ENDGAME" - koniec gry, gdy wszystkie statki zostaną zestrzelone

"SHOTDOWN" - zestrzelony statek, gdy wszystkie jego pola zostaną odsłonięte, a także informacje jaki to statek

"HIT" - trafienie pola należącego do danego statku oraz dane tego statku

"MISS" - brak trafienia

"CHECKED" - pole ponownie sprawdzane

Kod źródłowy:

```
package com.shipsgame.domain.model;

import com.shipsgame.domain.dto.StatusDto;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Game {

    private int steps; //number of steps
    private String user; //username
    private char[] board; //current state of board
```

```

private List<Ship> shipsList; //list of user's ships
private int[][] boardNumbers; //number of fields occupied by ships around the selected
                                position

public Game(String user, int steps, char[] board, List<Ship> shipsList, int[][]
boardNumbers) {
    this.steps = steps;
    this.user = user;
    this.board = board;
    this.shipsList = shipsList;
    this.boardNumbers = boardNumbers;
}

public Game(String user) {
    this.user = user;
    this.steps = 0;
    this.board = new char[81];
    this.shipsList = new ArrayList<>();
    this.boardNumbers = new int[9][9];

    boardGenerating();
}

public void setSteps(int steps) {
    this.steps = steps;
}

public void setUser(String user) {
    this.user = user;
}

public void setBoard(char[] board) {
    this.board = board;
}

public void setShipsList(List<Ship> shipsList) {
    this.shipsList = shipsList;
}

public void setBoardNumbers(int[][] boardNumbers) {
    this.boardNumbers = boardNumbers;
}

//method to get game status
public StatusDto getStatus(){
    return (new StatusDto("LOADGAME", "", 0, steps, new String(board)));
}

//method to get number of steps
public int getSteps() {
    return steps;
}

```

```

//method to get username
public String getUser() {
    return user;
}

//method to get current state of board
public char[] getBoard() {
    return board;
}

public int[][] getBoardNumbers() {
    return boardNumbers;
}

public List<Ship> getShipsList() {
    return shipsList;
}

//method which generates ships on board
private void boardGenerating(){
    Kod oraz opis algorytmu rozmieszczenia statków został przedstawiony w punkcie
    „Algorytm generowania rozmieszczenia statków”
    ...
}

//function which checks if spaceship was hit or not
public StatusDto shot(String position) {
    int p, m, pos;
    int x, y;
    boolean hit = false;
    StatusDto status;

    if (shipsList.isEmpty()){
        return new StatusDto("ENDGAME","", 0, steps, new String(board));
    }

    x = Integer.parseInt(position.substring(0,1));
    y = Integer.parseInt(position.substring(1,2));
    pos = 9 * y + x;

    steps++;
    m = 0;
    while((m < shipsList.size()) & !hit) {

        hit = false;
        p = 0;
        while((p < shipsList.get(m).getPositions().size()) & !hit) {
            if (shipsList.get(m).getPositions().remove(position)) {
                hit = true;
                board[pos] = (char) (64 + shipsList.get(m).getType());

                if (shipsList.get(m).getPositions().isEmpty()) {
                    status = new StatusDto("SHOTDOWN", shipsList.get(m).getName(),

```

```

        shipsList.get(m).getType(), steps, new String(board));
        shipsList.remove(m);
        if (shipsList.isEmpty()) {
            status = new StatusDto("ENDGAME", "", 0, steps, new
                String(board));
        }
        return status;
    } else {
        return (new StatusDto("HIT", shipsList.get(m).getName(),
            shipsList.get(m).getType(), steps, new String(board)));
    }
}
p++;
}
m++;
}

if (board[pos] == ' ') {
    board[pos] = String.valueOf(boardNumbers[x][y]).charAt(0);
    return (new StatusDto("MISS", "", 0, steps, new String(board)));
} else {
    return (new StatusDto("CHECKED", "", 0, steps, new String(board)));
}
}

@Override
public String toString() {
    return "Game{" +
        "steps=" + steps +
        ", user='" + user + '\'' +
        ", board=" + Arrays.toString(board) +
        ", shipsList=" + shipsList +
        ", boardNumbers=" + Arrays.toString(boardNumbers) +
        '}';
}
}

```

4.1.10. Klasa Ship

Klasa zawierająca informacje dotyczące statku kosmicznego, w tym typ (type), nazwę (name) oraz listę pozycji położenia poszczególnych elementów statku na planszy (positions).

Kod źródłowy:

```

package com.shipsgame.domain.model;

import java.util.List;
import java.util.ArrayList;

```

```

public class Ship {

    private String name; //name of the ship
    private int type; //type of spaceship
    private List<String> positions; //ship positions on the board

    //constructor
    public Ship(String name, int type) {
        this.type = type;
        this.name = name;
        this.positions = new ArrayList<>();
    }

    public Ship(String name, int type, List<String> positions) {
        this.name = name;
        this.type = type;
        this.positions = positions;
    }

    //function to get the type
    public int getType() {
        return type;
    }

    //function to set the type
    public void setType(int type) {
        this.type = type;
    }

    //function to get name of the ship
    public String getName() {
        return name;
    }

    //function to set name of the ship
    public void setName(String name) {
        this.name = name;
    }

    //function to get list of ship positions
    public List<String> getPositions() {
        return positions;
    }

    //function to set list of ship positions
    public void setPositions(List<String> positions) {
        this.positions = positions;
    }

    @Override
    public String toString() {
        return "Ship{" +

```



```

        "type=" + type +
        ", name='" + name + '\'' +
        ", positions=" + positions +
        '}';
    }
}

```

4.1.11. Interfejs GamesRepository

Interfejs GamesRepository pozwala na wykonywanie zapytań SQL z wykorzystaniem encji Games.

Kod źródłowy:

```

package com.shipsgame.domain.repository;

import com.shipsgame.domain.entity.Games;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

@Repository
public interface GamesRepository extends JpaRepository<Games, String>,
    CrudRepository<Games, String> {

    @Query(value = "SELECT g FROM Games g WHERE g.login = :login")
    Games findGamesByLogin(@Param("login") String login);

}

```

4.1.12. Interfejs PlayerRepository

Interfejs PlayerRepository pozwala na wykonywanie zapytań SQL z wykorzystaniem encji Player.

Kod źródłowy:

```

package com.shipsgame.domain.repository;

import com.shipsgame.domain.entity.Player;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

```

```

import javax.transaction.Transactional;

@Repository
public interface PlayerRepository extends JpaRepository<Player, String>,
CrudRepository<Player, String> {
    @Modifying
    @Transactional
    @Query(value = "UPDATE Player p SET p.bestScore = :bestScore WHERE p.login = :login
        AND (p.bestScore > :bestScore OR p.bestScore IS NULL)")
    void updateScore(@Param("login") String login, @Param("bestScore") Integer bestScore);
}

```

4.1.13. Klasa GameController

Klasa zawierająca metody związane z obsługą zapytań do serwera.

getUserGame(@PathVariable String user, @PathVariable String pass)

Metoda obsługująca zapytanie POST postaci "api/getgame/user" związane ze sprawdzeniem czy użytkownik o nazwie user posiada grę na serwerze, a jeżeli tak to przesłanie informacji dotyczącej tej gry, w szczególności liczby wykonanych kroków oraz aktualnej planszy.

setNewGame(@PathVariable String user, @PathVariable String pass)

Metoda obsługująca zapytanie POST postaci "api/newgame/user" związane utworzeniem nowej gry użytkownika o nazwie user oraz przesłanie informacji o utworzeniu gry.

setShot(@RequestParam String user, @PathVariable String pass, @RequestParam String shot)

Metoda obsługująca zapytanie POST postaci "api/shotgame?user=user&shot=shot" związane ze sprawdzeniem pozycji zapisanej w zmiennej "shot" dla użytkownika "user" na planszy oraz przesłaniem wyniku sprawdzenia.

getRank()

Metoda obsługująca zapytanie GET postaci "api/getrank" związana z pobieraniem rankingu graczy.

loginPlayer(@RequestParam String user, @PathVariable String pass)

Metoda obsługująca zapytanie POST postaci "api/login?user=user&pass=pass" związana z logowaniem użytkownika.

Kod źródłowy:

```
package com.shipsgame.rest;

import com.shipsgame.domain.dto.ErrorOrder;
import com.shipsgame.domain.dto.StatusDto;
import com.shipsgame.service.GameService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping(value = "/api")
public class GameController {

    private static final Logger LOGGER = LoggerFactory.getLogger(GameController.class);

    private final GameService gameService;

    @Autowired
    public GameController(GameService gameService) {
        this.gameService = gameService;
    }

    @CrossOrigin
    @PostMapping(value = "/getgame", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> getUserGame(@RequestParam String user, @RequestParam String
        pass) {
        final StatusDto statusDto = gameService.getGame(user, pass);
        if(statusDto == null) {
            return new ResponseEntity<>("Access denied." ,HttpStatus.UNAUTHORIZED);
        }
        return new ResponseEntity<>(statusDto, HttpStatus.OK);
    }

    @CrossOrigin
    @PostMapping(value = "/newgame", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<> setNewGame(@RequestParam String user, @RequestParam String
        pass) {
        final StatusDto statusDto = gameService.newGame(user, pass);
        if(statusDto == null) {
            return new ResponseEntity<>("Access denied. Invalid login details."
            ,HttpStatus.UNAUTHORIZED);
        }
        return new ResponseEntity<>(statusDto, HttpStatus.OK);
    }
}
```

```

@CrossOrigin
@PostMapping(value = "/shotgame", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<?> setShot(@RequestParam String user, @RequestParam String pass,
    @RequestParam String shot) {
    final StatusDto statusDto = gameService.shotGame(user, pass, shot);
    if(statusDto == null) {
        return new ResponseEntity<>("Access denied. Invalid login details."
,HttpStatus.UNAUTHORIZED);
    }
    if(!shot.matches("\\d\\d")) {
        throw new ErrorOrder("Invalid positions.");
    }
    return new ResponseEntity<>(statusDto, HttpStatus.OK);
}

@CrossOrigin
@GetMapping(value = "/getrank", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> getRank() {
    final String rank = gameService.getRank();
    return (rank==null || rank.isEmpty()) ? new ResponseEntity<>(rank,
    HttpStatus.NOT_FOUND) : new ResponseEntity<>(rank, HttpStatus.OK);
}

@CrossOrigin
@PostMapping(value = "/login", produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Boolean> loginPlayer(@RequestParam String user, @RequestParam
    String pass) {
    boolean status = gameService.loginPlayer(user,pass);

    return (status) ? new ResponseEntity<>(status, HttpStatus.OK) : new
    ResponseEntity<>(status, HttpStatus.UNAUTHORIZED);
}
}

```

4.1.14. Klasa GameServiceImpl

Klasa przechowująca wszystkie gry użytkowników, zawiera logikę biznesową.

getGame(String user, String pass)

Metoda wyszukująca grę użytkownika po parametrze user oraz pass. Metoda zwraca obiekt klasy StatusDto, który zawiera informacje dotyczące danej gry, w szczególności liczbę dotychczas wykonanych kroków steps oraz aktualną planszę board. W przypadku braku gry użytkownika obiekt klasy Status zwraca kod "NOGAME".

loginPlayer(String user, String pass)

Metoda odpowiedzialna za logowanie użytkownika. Przekazywane parametry to nazwa użytkownika (user) oraz hasło (pass).

newGame(String user, String pass)

Metoda tworząca nową grę użytkownika o parametrze user oraz pass. Metoda zwraca obiekt klasy StatusDto, który zawiera informacje dotyczące nowej gry oraz kod "NEWGAME".

shotGame(String user, String pass, String shot)

Metoda wyszukująca grę użytkownika po parametrze user oraz pass, sprawdzająca pole wskazane parametrem shot. Parametr user tworzony jest w oparciu o nazwę użytkownika oraz hasło, natomiast parametr shot jest typu String i zawiera sprawdzaną pozycję np. "02". Metoda zwraca obiekt klasy StatusDto, zawierający między innymi aktualny stan planszy board, liczbę wykonanych steps kroków oraz wynik sprawdzenia w postaci kodu code:

"ENDGAME" - koniec gry, gdy wszystkie statki zostaną zestrzelone

"SHOTDOWN" - zestrzelony statek, gdy wszystkie jego pola zostaną odsłonięte, a także informacje jaki to statek

"HIT" - trafienie pola należącego do danego statku oraz dane tego statku

"MISS" - brak trafienia

"CHECKED" - pole ponownie sprawdzane

getRank()

Metoda odpowiedzialna za pobranie rankingu graczy z serwera.

Kod źródłowy (interfejs):

```
package com.shipsgame.service;

import com.shipsgame.domain.dto.StatusDto;

public interface GameService {
    StatusDto newGame(String user, String pass);
    StatusDto getGame(String user, String pass);
    StatusDto shotGame(String user, String pass, String shot);
    String getRank();
    boolean loginPlayer(String login, String password);
}
```

Kod źródłowy (implementacja):

```
package com.shipsgame.service.impl;

import com.shipsgame.domain.converter.Converter;
import com.shipsgame.domain.dto.StatusDto;
import com.shipsgame.domain.entity.Games;
```

```

import com.shipsgame.domain.entity.Player;
import com.shipsgame.domain.model.Game;
import com.shipsgame.domain.repository.GamesRepository;
import com.shipsgame.domain.repository.PlayerRepository;
import com.shipsgame.service.GameService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Service
public class GameServiceImpl implements GameService {

    private final PlayerRepository playerRepository;
    private final GamesRepository gamesRepository;
    private final Converter<Games, Game> gameMapper;
    private final Converter<Game, Games> gamesMapper;

    @Autowired
    public GameServiceImpl(PlayerRepository playerRepository,
                           GamesRepository gamesRepository,
                           Converter<Games, Game> gameMapper,
                           Converter<Game, Games> gamesMapper) {
        this.playerRepository = playerRepository;
        this.gamesRepository = gamesRepository;
        this.gameMapper = gameMapper;
        this.gamesMapper = gamesMapper;
    }

    @Override
    public boolean loginPlayer(String login, String password) {
        List<Player> players = playerRepository.findAll();

        for(Player player : players) {
            if(player.getLogin().equals(login)) {
                return player.getPassword().equals(password);
            }
        }
        playerRepository.save(new Player.Builder()
                              .login(login)
                              .password(password)
                              .build());

        return true;
    }

    @Override
    public StatusDto newGame(String user, String pass) {
        if(!loginPlayer(user, pass)) {
            return null;
        }
        Game newGame = new Game(user);
    }

```

```

        gamesRepository.save(gamesMapper.convert(newGame));
        return new StatusDto.Builder()
            .code("NEWGAME")
            .shipName("")
            .type(0)
            .steps(newGame.getSteps())
            .board(new String(newGame.getBoard()))
            .build();
    }

    @Override
    public StatusDto getGame(String user, String pass) {
        if(!loginPlayer(user, pass)) {
            return null;
        }
        Optional<Games> optionalGames = gamesRepository.findById(user);
        if(optionalGames.isPresent()) {
            Games games = gamesRepository.findGamesByLogin(user);
            return gameMapper.convert(games).getStatus();
        }
        return new StatusDto.Builder()
            .code("NOGAME")
            .board("")
            .shipName("")
            .steps(0)
            .type(0)
            .build();
    }

    @Override
    public StatusDto shotGame(String user, String pass, String shot) {
        if(!loginPlayer(user, pass) || !Optional.ofNullable(loadGame(user)).isPresent()) {
            return null;
        }
        Game game = loadGame(user);
        StatusDto statusDto = game.shot(shot);
        if(statusDto.getCode().equals("ENDGAME")) {
            playerRepository.updateScore(user, statusDto.getSteps());
            gamesRepository.deleteById(user);
            return statusDto;
        }
        gamesRepository.save(gamesMapper.convert(game));
        return statusDto;
    }

    @Override
    public String getRank() {
        List<Player> players = playerRepository.findAll()
            .stream()
            .filter(player -> player.getBestScore() != null)
            .sorted(Comparator.comparing(Player::getBestScore))
            .collect(Collectors.toList());

        String rank = "";
    }

```

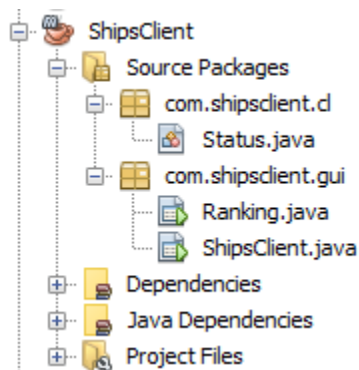
```

        for(Player player : players) {
            rank += player.getLogin() + " " + player.getBestScore() +
                System.LineSeparator();
        }
        return rank;
    }

    private Game loadGame(String user) {
        if(gamesRepository.findById(user).isPresent()) {
            return gameMapper.convert(gamesRepository.findGamesByLogin(user));
        }
        return null;
    }
}

```

4.2. Struktura projektu – klient



Rys. Struktura projektu – klient

Klient gry składa się z następujących klas:

Status – klasa odpowiedzialna za komunikację klient/serwer.

Ranking – GUI odpowiedzialne za wygląd rankingu

ShipsClient – GUI odpowiedzialne za wygląd gry

4.2.1. Klasa ShipsClient

Główna klasa klienta zawierająca metodą main().

4.2.2. Klasa BackgroundPanel

Klasa zawierająca metodę wyświetlającą tło klienta.

4.2.3. Klasa BoardPanel

Klasa zawierająca metodę paintBoard(Graphics2D g) rysującą planszę oraz obsługę myszki.

4.2.4. Metoda paintBoard(Graphics2D g)

Metoda rysująca "kratownicę" planszy oraz wyświetlająca zasady gry, a także ikony statków kosmicznych, które zostały trafione.

4.2.5. Metoda mouseClicked(MouseEvent e)

Metoda klasy BoardPanel obsługująca zdarzenie naciśnięcia lewego przyciskumyshi. Po wskazaniu miejsca na planszy oraz naciśnięciu przycisku myszy wysyłane jest zapytanie do serwera w którym podana jest pozycja na planszy. Z serwera za pośrednictwem klasy Status jest przekazywana odpowiedź w postaci kodu.

"CHECKED" - pole było już sprawdzane

"MISS" - brak trafienia

"HIT" - statek trafiony

"SHOTDOWN" statek zestrzelony

"ENDGAME" - koniec gry (wszystkie statki zestrzelone)

4.2.6. Metoda jButtonNewGameActionPerformed()

Metoda odpowiedzialna za obsługę przycisku tworzenia nowej gry użytkownika. Po naciśnięciu przycisku do serwera wysyłana jest informacja za pośrednictwem zapytania HTTP o utworzeniu nowej gry użytkownika o podanej nazwie. Serwer za pośrednictwem klasy Status zwraca kod o utworzeniu nowej gry użytkownika lub informację o istnieniu takiej gry.

4.2.7. Metoda jButtonGetGameActionPerformed()

Metoda odpowiedzialna za obsługę przycisku wczytania gry użytkownika. Po naciśnięciu przycisku do serwera wysyłana jest informacja za pośrednictwem zapytania HTTP o wczytaniu gry użytkownika dla podanej nazwy. Serwer za pośrednictwem klasy Status zwraca kod oraz informacje dotyczące zapisanej gry lub informację, że gra nie istnieje.

4.2.8. Metoda jButtonLoginActionPerformed()

Metoda odpowiedzialna za obsługę przycisku logowania użytkownika. Po naciśnięciu przycisku do serwera wysyłana jest informacja za pośrednictwem zapytania HTTP z danymi użytkownika – nazwą użytkownika oraz hasłem. Metoda sprawdza również czy użytkownik jest już zalogowany.

4.2.9. Metoda jButtonRankActionPerformed()

Metoda odpowiedzialna za obsługę przycisku wyświetlania rankingu użytkowników. Po naciśnięciu przycisku do serwera wysyłana jest informacja za pośrednictwem zapytania HTTP o ranking użytkowników. Metoda pozwala wyświetlić ranking w formie tabeli.

4.2.10. Metoda newGame(String user, String pass)

Metoda odpowiedzialna za wysłanie zapytania do serwera w celu utworzenia nowej gry użytkownika.

4.2.11. Metoda getGame(String user, String pass)

Metoda odpowiedzialna za wysłanie zapytania do serwera w celu wczytania gry wskazanego użytkownika.

4.2.12. Metoda getRank(String user, String pass)

Metoda odpowiedzialna za wysłanie zapytania do serwera w celu wczytania rankingu wskazanego użytkownika.

4.2.13. Metoda MD5(String password)

Metoda szyfrowania łańcucha znaków podanych w zmiennej password na kod MD5.

4.2.14. Metoda sound(String audioFile)

Metoda odpowiedzialna za odtwarzanie dźwięków w programie po podaniu nazwy pliku dźwiękowego z rozszerzeniem .wav w zmiennej audioFile.

4.2.15. Metoda checkName(String s)

Metoda sprawdzająca poprawność nazwy użytkownika, która może składać się z małych i dużych liter oraz cyfr.

4.2.16. Klasa Status

Klasa Status wykorzystywana jest do przesyłania informacji z serwera do klienta. Zawiera informacje dotyczące liczby wykonanych przez użytkownika kroków (steps), nazwę kodu wykonanych operacji lub stanu gry (code), nazwę statku kosmicznego (shipName) na którym była wykonana operacja oraz jego typ (type), a także przechowuje stan planszy (board).

4.3. Algorytm generowania rozmieszczenia statków

W związku z założeniami gry koniecznym było stworzenie odpowiedniego algorytmu, który w sposób losowy dokonuje rozmieszczenia wszystkich statków na planszy przy założeniach, że statki nie mogą się stykać. Realizację algorytmu oparto na wykorzystaniu funkcji losowej, a jego implementację zamieszczono poniżej:

```
//method which generates ships on board
private void boardGenerating(){
    int length, orientation, pozX, pozY, number;
    int xmin, xmax, ymin, ymax;
    boolean find;

    for (int i=0; i < 81; i++) {
        board[i] = ' ';
    }

    //creating spaceships and adding them to list
    shipslist.add(new Ship("Transportowiec", 4));
    shipslist.add(new Ship("Transportowiec", 4));
    shipslist.add(new Ship("Samolot kosmiczny", 3));
    shipslist.add(new Ship("Samolot kosmiczny", 3));
    shipslist.add(new Ship("Samolot kosmiczny", 3));
    shipslist.add(new Ship("Wahadlowiec", 2));
    shipslist.add(new Ship("Wahadlowiec", 2));
    shipslist.add(new Ship("Wahadlowiec", 2));
    shipslist.add(new Ship("Szturmowiec", 1));
```

```

shipslist.add(new Ship("Szturmowiec", 1));

for (int i = 0; i < shipsList.size(); i++) {
    find = false;
    //find = true - found positions at which the ship will not go beyond the boards
    //find = false - NOT found positions at which the ship will not go beyond the
    boards
    while (!find) {
        orientation = (int) (Math.random() * 2);
        //orientation - 0: horizontal, 1: vertical
        pozX = (int) (Math.random() * 9);
        //generates random position X for ship on board
        pozY = (int) (Math.random() * 9);
        //generates random position Y for ship on board

        if (orientation == 0) {
            if ((pozX + shipsList.get(i).getType()) < 9) {
                List<String> tmpPos = new ArrayList<>();
                find = true;
                //find = true - found positions for the ship successfully, fits on the
                board
                length = 0;
                //temporary variable for length of ships depended on ship type
                //for example: type = 4 (length: 0, 1, 2, 3), type = 2 (length: 0,1)
                while ((length < shipsList.get(i).getType()) & find) {
                    //check if positions on board are not occupied by other ships
                    number = pozX + length;
                    if (boardNumbers[number][pozY] != 0) {
                        //boardNumbers[number][pozY] == 0 - position is not occupied
                        //boardNumbers[number][pozY] != 0 - position is occupied
                        find = false;
                    }
                    if (number - 1 >= 0){
                        //ships can't touch each others, check if position is not
                        occupied and fits on the board
                        if (boardNumbers[number - 1][pozY] != 0){
                            find = false;
                        }
                    }
                    if (number + 1 < 9){
                        if (boardNumbers[number + 1][pozY] != 0){
                            find = false;
                        }
                    }
                }
                if (pozY - 1 >= 0){
                    if (boardNumbers[number][pozY - 1] != 0) {
                        find = false;
                    }
                }
                if (pozY + 1 < 9){
                    if (boardNumbers[number][pozY + 1] != 0) {
                        find = false;
                    }
                }
            }
            length++;

```

```

        //I need check the entire length of the ship depended on its type
    }
    if (find) {
        //if (find) - I checked positions, positions are not occupied,
        //ship can be placed on board
        for (int m = 0; m < shipsList.get(i).getType(); m++) {
            number = pozX + m;
            boardNumbers[number][pozY]=(shipsList.get(i).getType()*(-1));
            //multiply by (-1) because later I must know what positions
            //are occupied by ships
            //negative numbers represents the postions of ships
            //positive numbers represents the number of occupied positions
            //around
            tmpPos.add("" + number + "" + pozY);
            //every ship has own list positions which occupies on board
            //(tmpPos)
        }
        shipsList.get(i).setPositions(tmpPos);
        //add ships positions to list
    }
}
} else {
    //Same operations for vertical orientation
    if ((pozY + shipsList.get(i).getType()) < 9) {
        List<String> tmpPos = new ArrayList<>();
        find = true;
        length = 0;
        while ((length < shipsList.get(i).getType()) & find) {
            number = pozY + length;
            if (boardNumbers[pozX][number] != 0) {
                find = false;
            }
            if (number - 1 >= 0 ){
                if (boardNumbers[pozX][number - 1] != 0) {
                    find = false;
                }
            }
            if (number + 1 < 9 ){
                if (boardNumbers[pozX][number + 1] != 0) {
                    find = false;
                }
            }
            if (pozX - 1 >= 0){
                if (boardNumbers[pozX - 1][number] != 0) {
                    find = false;
                }
            }
            if (pozX + 1 < 9){
                if (boardNumbers[pozX + 1][number] != 0) {
                    find = false;
                }
            }
            length++;
        }
        if (find) {

```


5. Lista wykorzystanych technologii oraz bibliotek.

Lista technologii, które zostały wykorzystane w projekcie:



Java (klient/serwer)– współbieżny, oparty na klasach, obiektowy język programowania ogólnego zastosowania.



NetBeans IDE (klient) – środowisko programistyczne, projekt otwartego oprogramowania mający za zadanie dostarczanie efektywnych narzędzi programowania.



IntelliJ IDEA (serwer) - komercyjne zintegrowane środowisko programistyczne dla Javy firmy JetBrains.



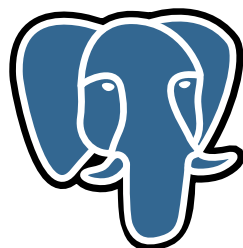
Spring framework (serwer) – szkielet tworzenia aplikacji w języku Java dla platformy Java Platform, Enterprise Edition. Spring Framework powstał na bazie kodu opublikowanego w książce Roda Johnsona Design and Development.



Swing (klient)– biblioteka graficzna używana w języku programowania Java, upubliczniona w lipcu roku 1997. Jest nowszą, ulepszoną wersją biblioteki AWT.



Java Persistence API (baza danych)– oficjalny standard mapowania obiektowo-relacyjnego firmy Sun Microsystems dla języka programowania Java.



PostgreSQL (baza danych) – relacyjna baza danych implementująca standard SQL.



Postman - narzędzie do testowania Web API, badania komunikacji klient-serwer.

6. Określenie scenariuszy użytkownika i przypadków użycia.

1. Rejestracja użytkownika.

- Przypadek rozpoczyna się, kiedy użytkownika loguje się w aplikacji.
- Podczas rejestracji system prosi o podanie nazwy użytkownika i hasła, które będą używane w celu logowania się do aplikacji.

2. Utworzenie nowej gry.

- Przypadek rozpoczyna się w momencie gdy serwer otrzyma komunikat o rozpoczęciu nowej gry przez użytkownika.
- Sprawdzany jest stan gier użytkownika, jeżeli posiada on zapisaną grę może zdecydować czy ją wczytać lub rozpocząć nową.
 - a. Jeżeli zdecyduje rozpocząć nową grę, zapisana gra zostanie usunięta.
 - b. Jeżeli zdecyduje wczytać grę, to zostanie ona wczytana z serwera.

3. Wczytanie gry.

- Przypadek rozpoczyna się, gdy użytkownika wyśle komunikat o wczytanie gry.
 - a. Jeżeli użytkownika posiada zapisaną grę zostanie ona wczytana.
 - b. Jeżeli użytkownika nie posiada zapisanej gry zostanie wyświetlony komunikat o braku zapisanej gry.

4. Sprawdzenie pola.

- Przypadek rozpoczyna się, kiedy użytkownika naciśnie na plansze gry.
- Informacja o naciśniętym polu zostaje wysłana na serwer gdzie jest sprawdzane czy na nim znajduje się statek.
 - a. Jeżeli udało się trafić w statek zostanie wysłany komunikat o trafionym statku.
 - b. Jeżeli nie udało się trafić w statek zostanie wysłany komunikat o pudle.

5. Koniec gry.

- Przypadek rozpoczyna się gdy użytkownika zestrzeli wszystkie statki na planszy.
- Pojawia się komunikat o wyniku gry, użytkownika ma możliwość zakończenia gry lub rozpoczęcia nowej.
 - a. Jeżeli użytkownik zakończy grę, zostanie zamknięte okno aplikacji.
 - b. Jeżeli użytkownik nie zakończy gry, zostanie utworzona nowa gra.

7. Identyfikacja funkcji.

1. Zarządzanie użytkownikami.

Opis modułu:

System pozwala tworzenie unikalnych kont.

Podsumowanie funkcjonalności:

1.1 Rejestracja

1.1.1 Założenie konta użytkownika.

1.2 Logowanie

1.2.1 Logowanie użytkownika

2. Zarządzanie grami użytkownika.

Opis modułu:

System pozwala na zapisywanie, wczytywanie, modyfikowanie gier użytkownika.

Podsumowanie funkcjonalności:

2.1 Zapis gry

2.1.1 Zapis obecnego stanu gry.

2.2 Wczytanie gry

2.2.1 Wczytanie stanu gry.

2.3 Usunięcie gry.

2.3.1 Usunięcie gry z bazy.

3. Zarządzanie grą użytkownika.

Opis modułu:

System pozwala na generowanie planszy ze statkami, sprawdzanie ich położenia, usuwania z listy, zliczania liczby kroków.

Podsumowanie funkcjonalności:

3.1 Generuj plansze

3.1.1 Tworzenie planszy gracza dla nowo utworzonej gry.

3.2 Sprawdź położenie

3.2.1 Sprawdzenie czy na danej pozycji znajduje się statek.

3.3 Usuń statek

3.3.1 Usuń zestrzelony przez gracza statek z listy.

3.4 Zliczanie kroków

3.4.1 Po każdym sprawdzeniu pola, liczba kroków zwiększana jest o jeden.

4. Ranking graczy

Opis modułu:

System daje możliwość pobrania rankingu najlepszych graczy, który jest aktualizowany po każdej ukończonej grze i zapisywany w bazie.

Podsumowanie funkcjonalności:

4.1 Modyfikacja rankingu

4.1.1 Dodawanie do listy wyniku nowego gracza.

4.1.2 Aktualizowanie wyniku gracza.

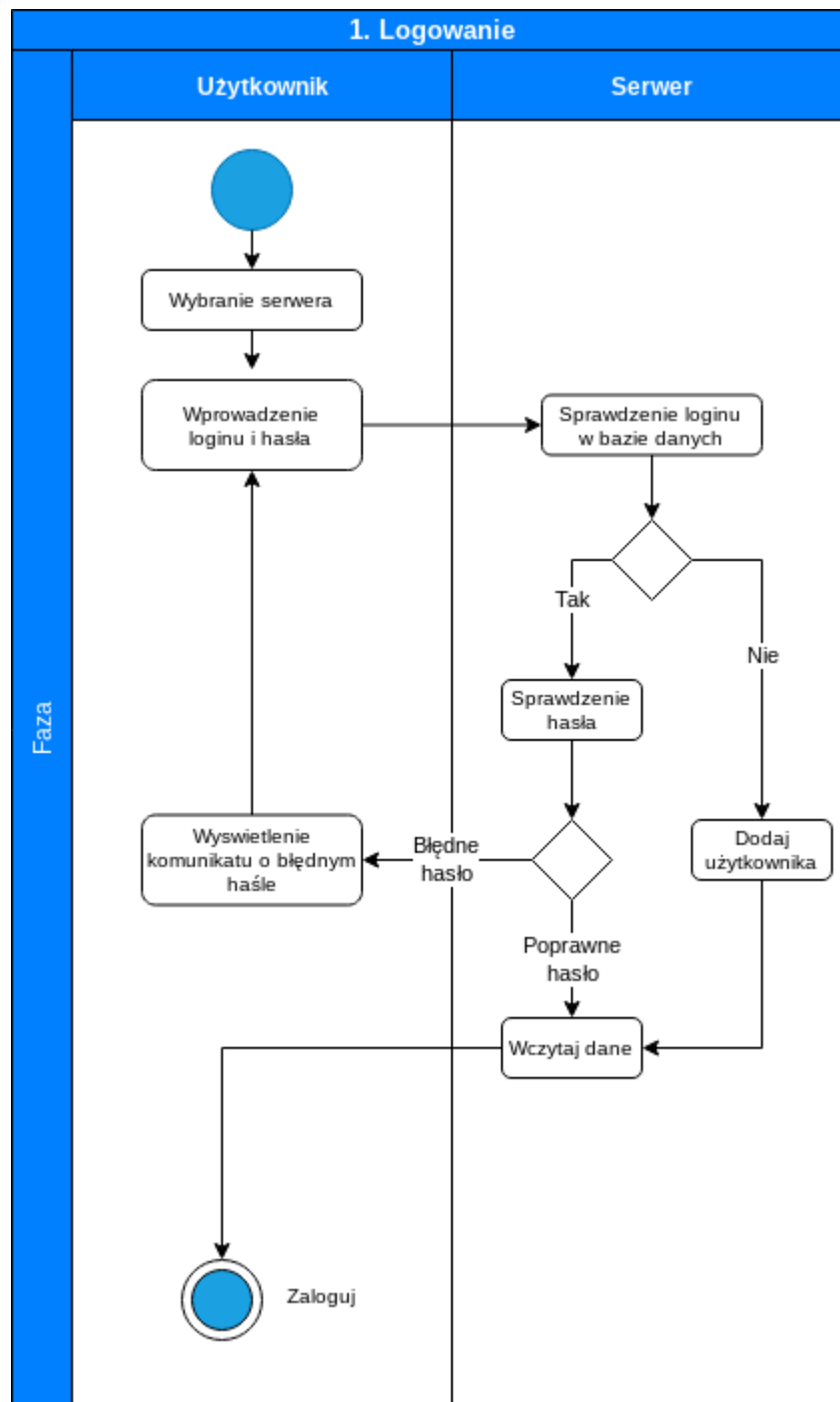
4.1.3 Sortowanie wyników.

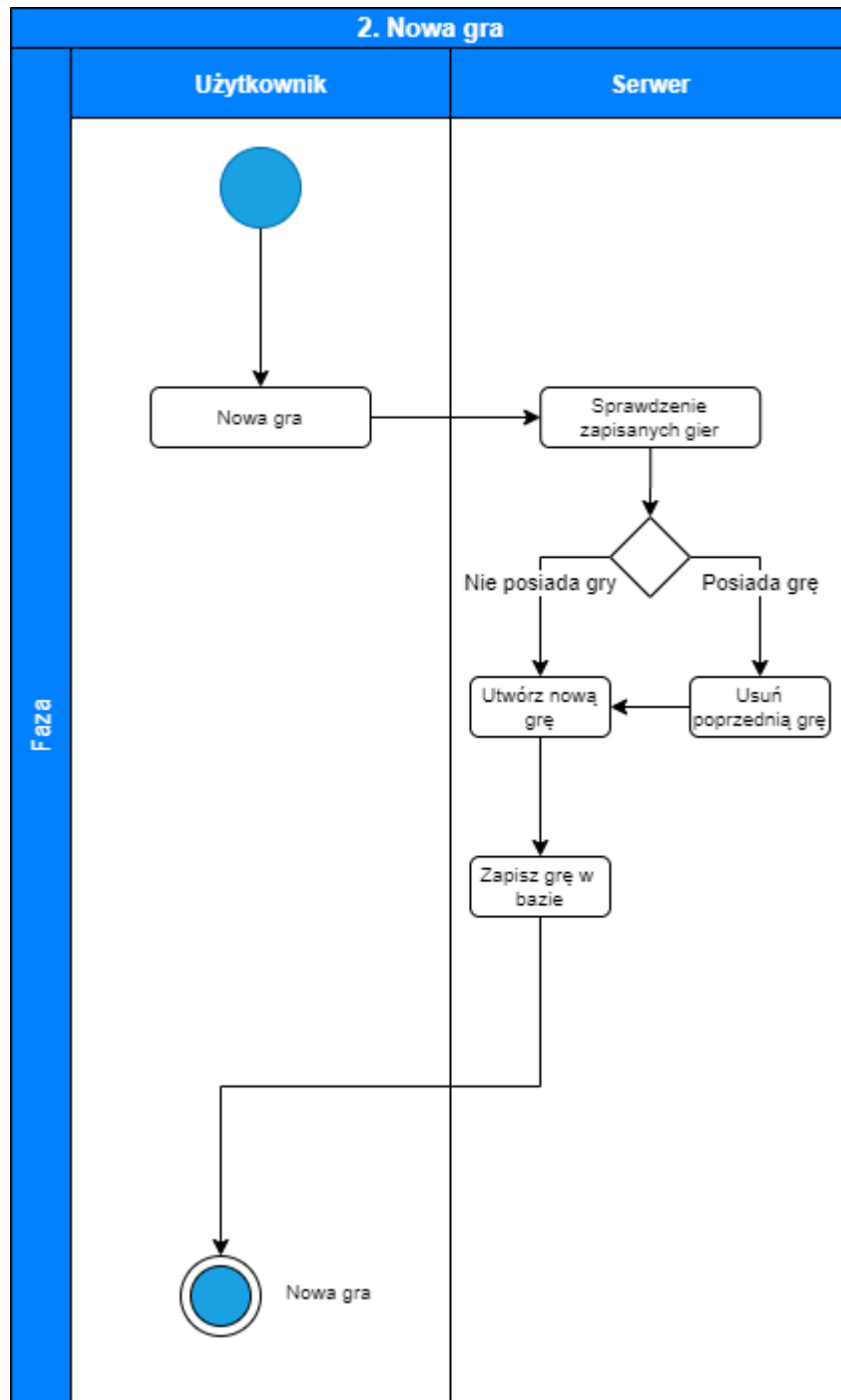
4.2 Zarządzenie rankingiem

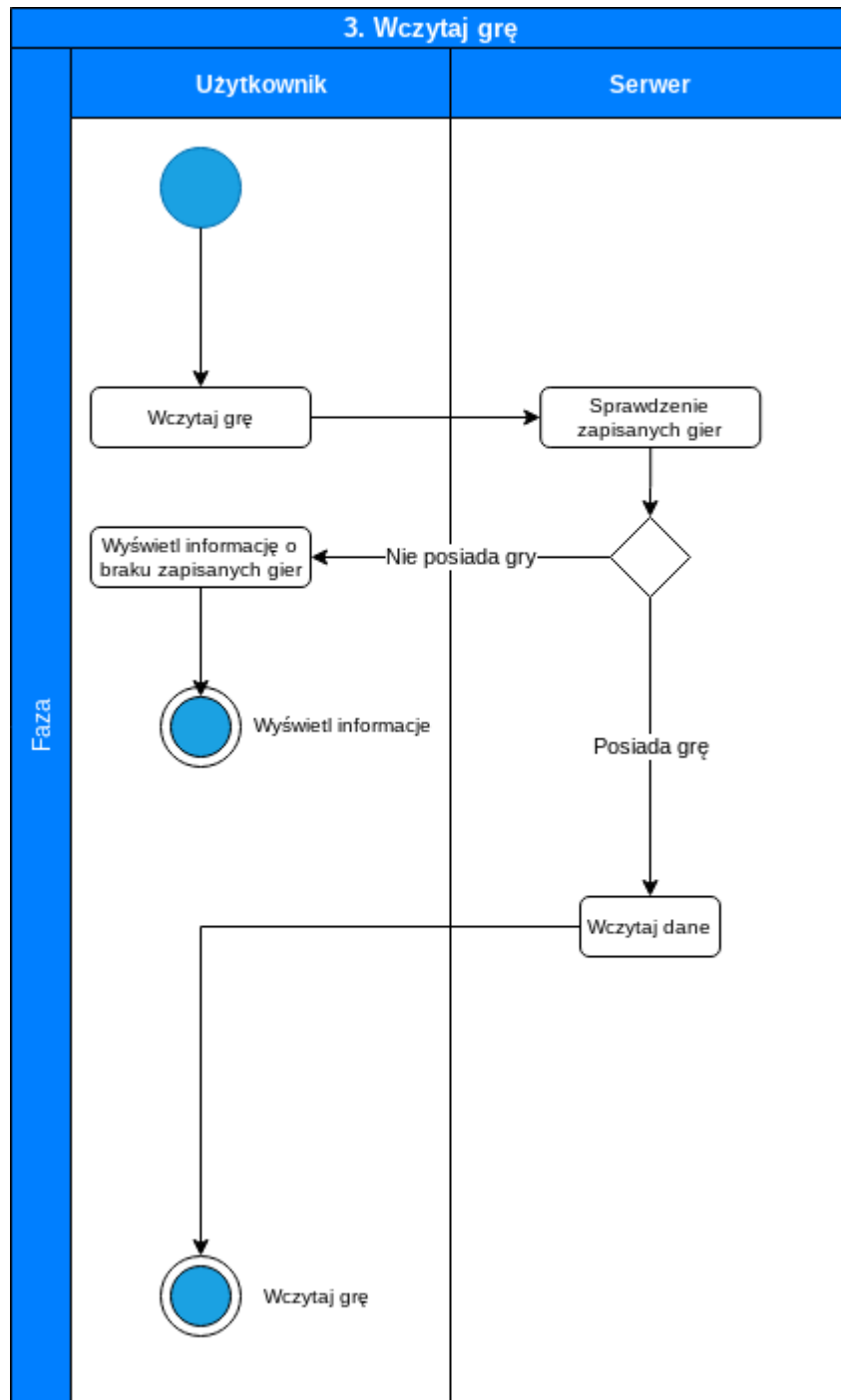
4.2.1 Pobranie obecnego rankingu

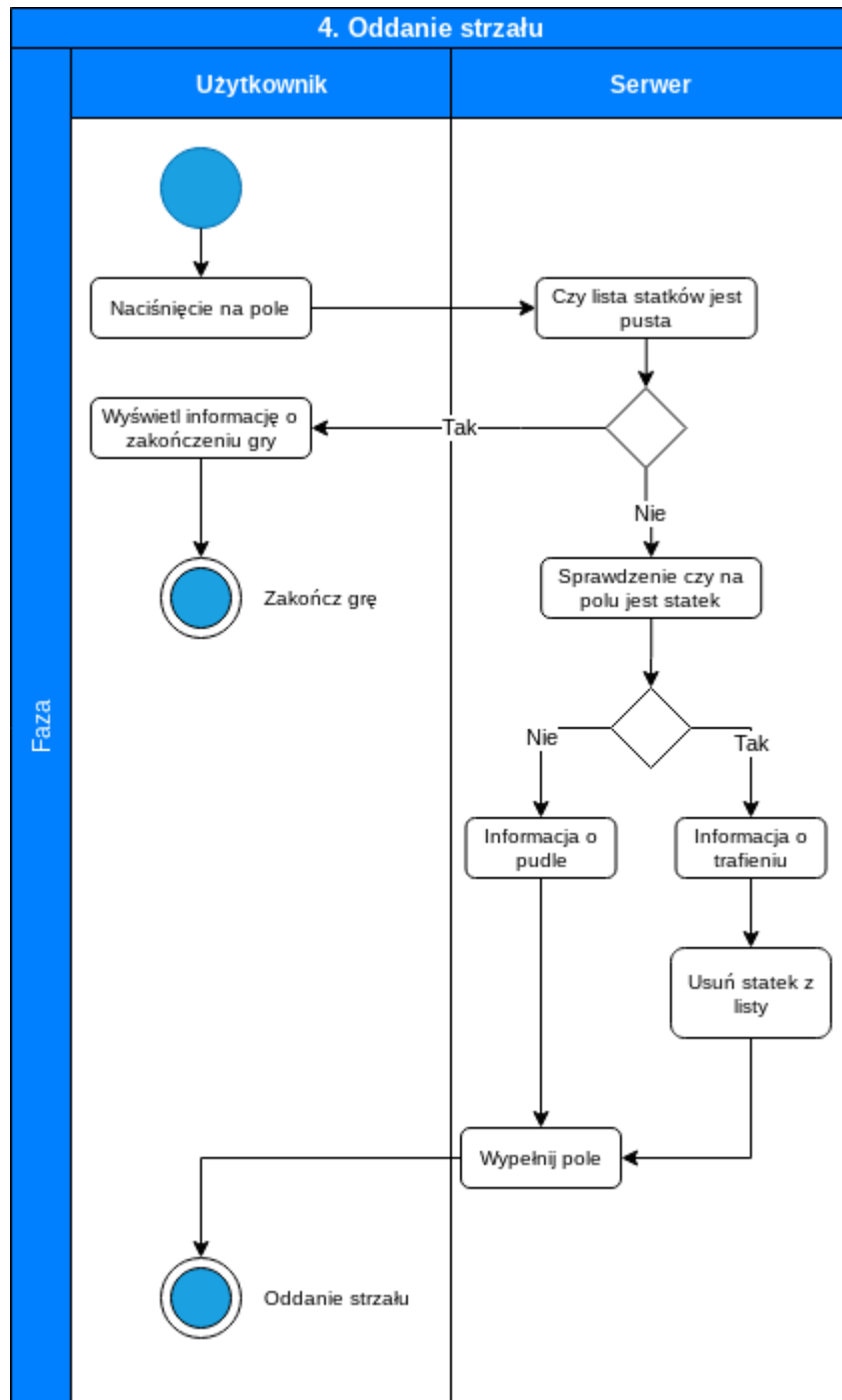
8. Diagramy UML

8.1. Diagramy aktywności

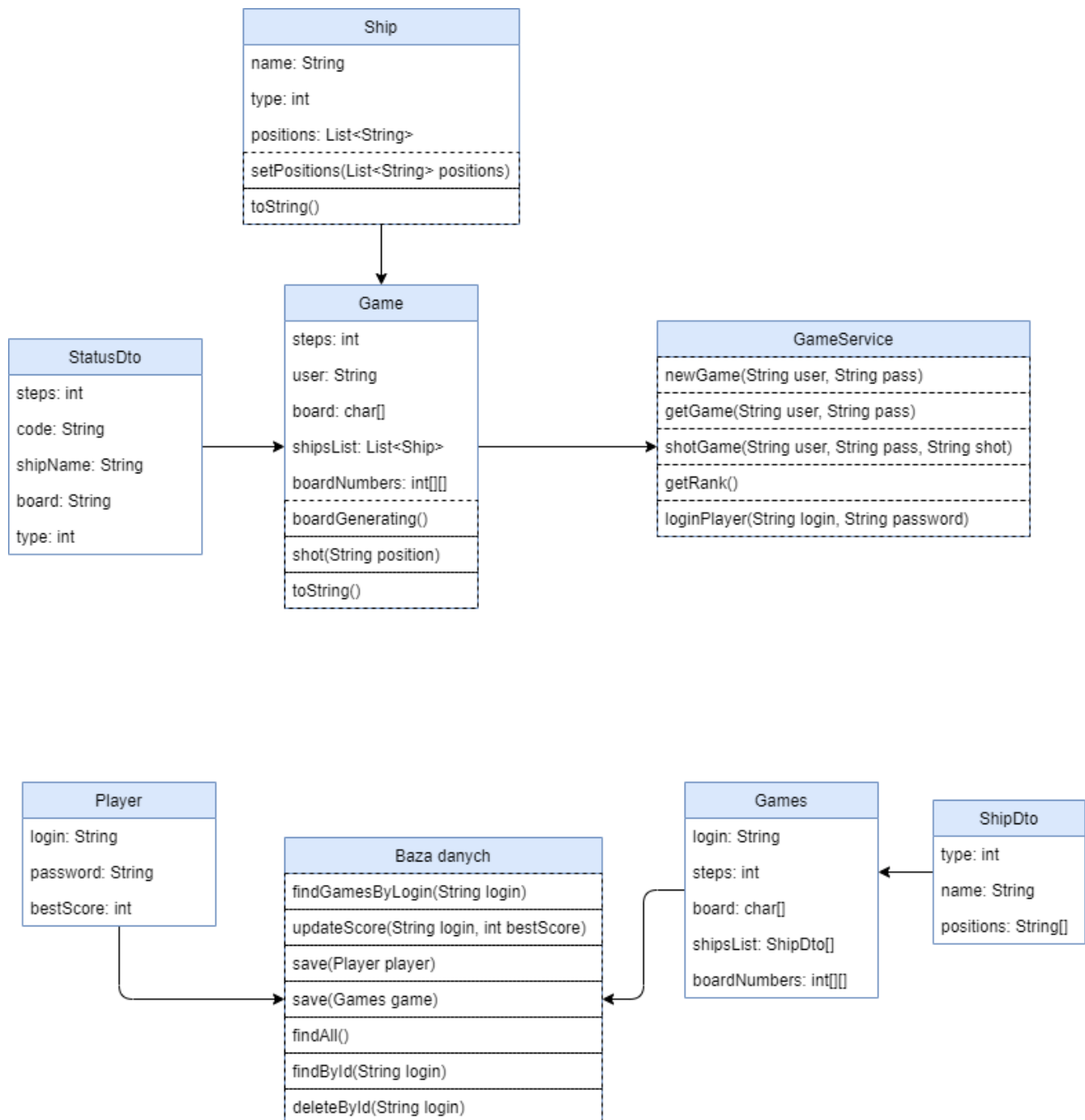




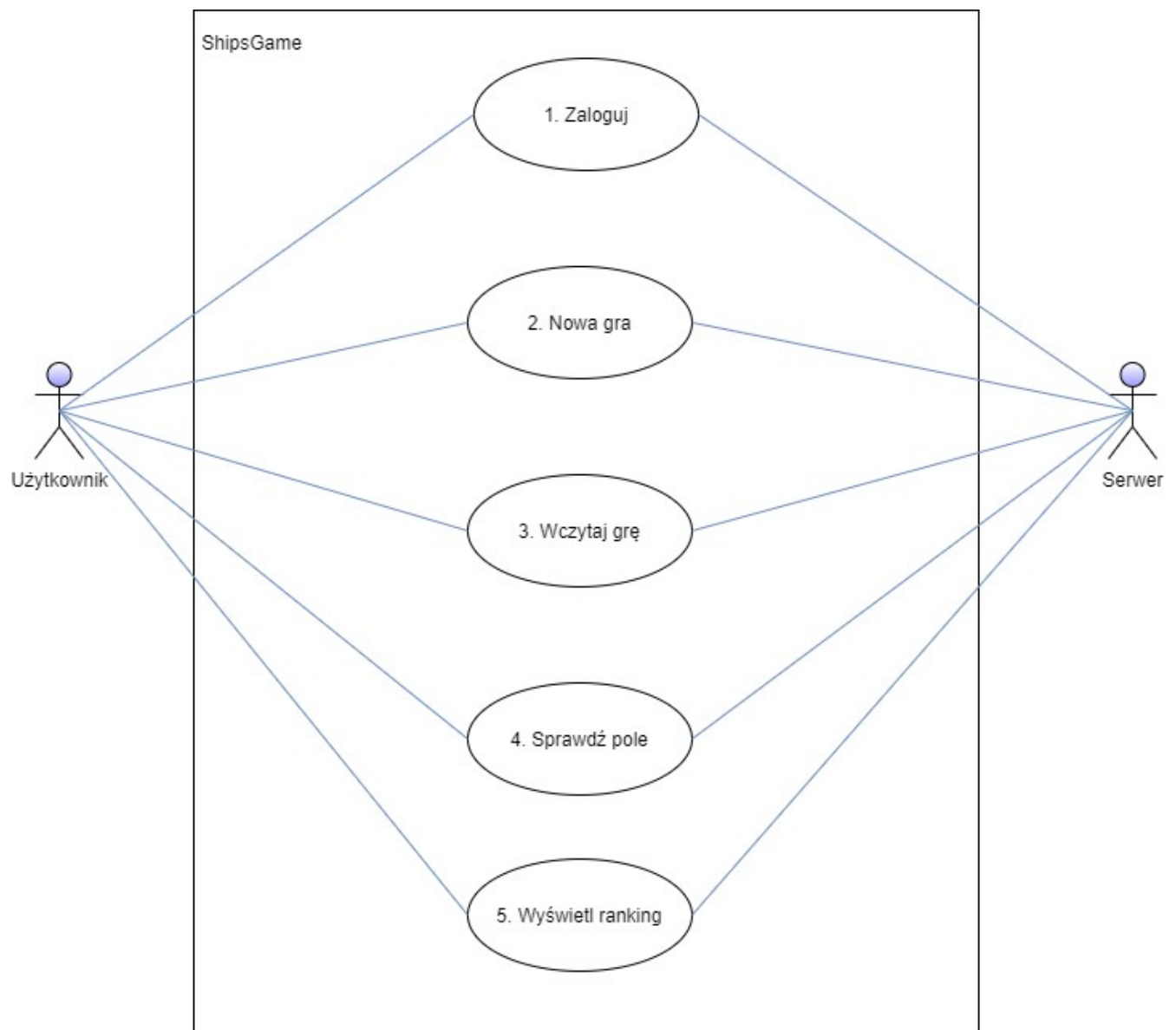




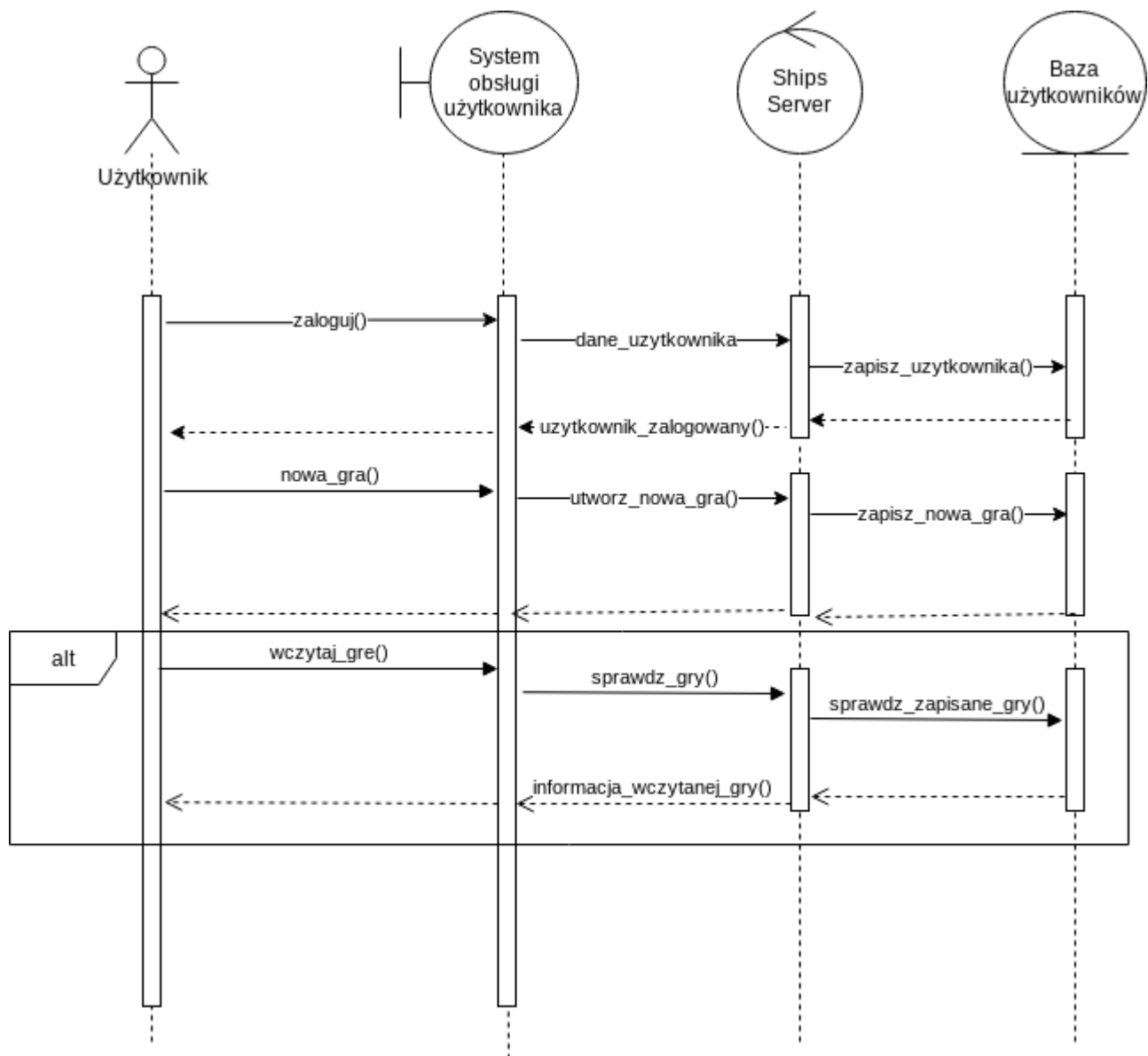
8.2. Diagramy klas

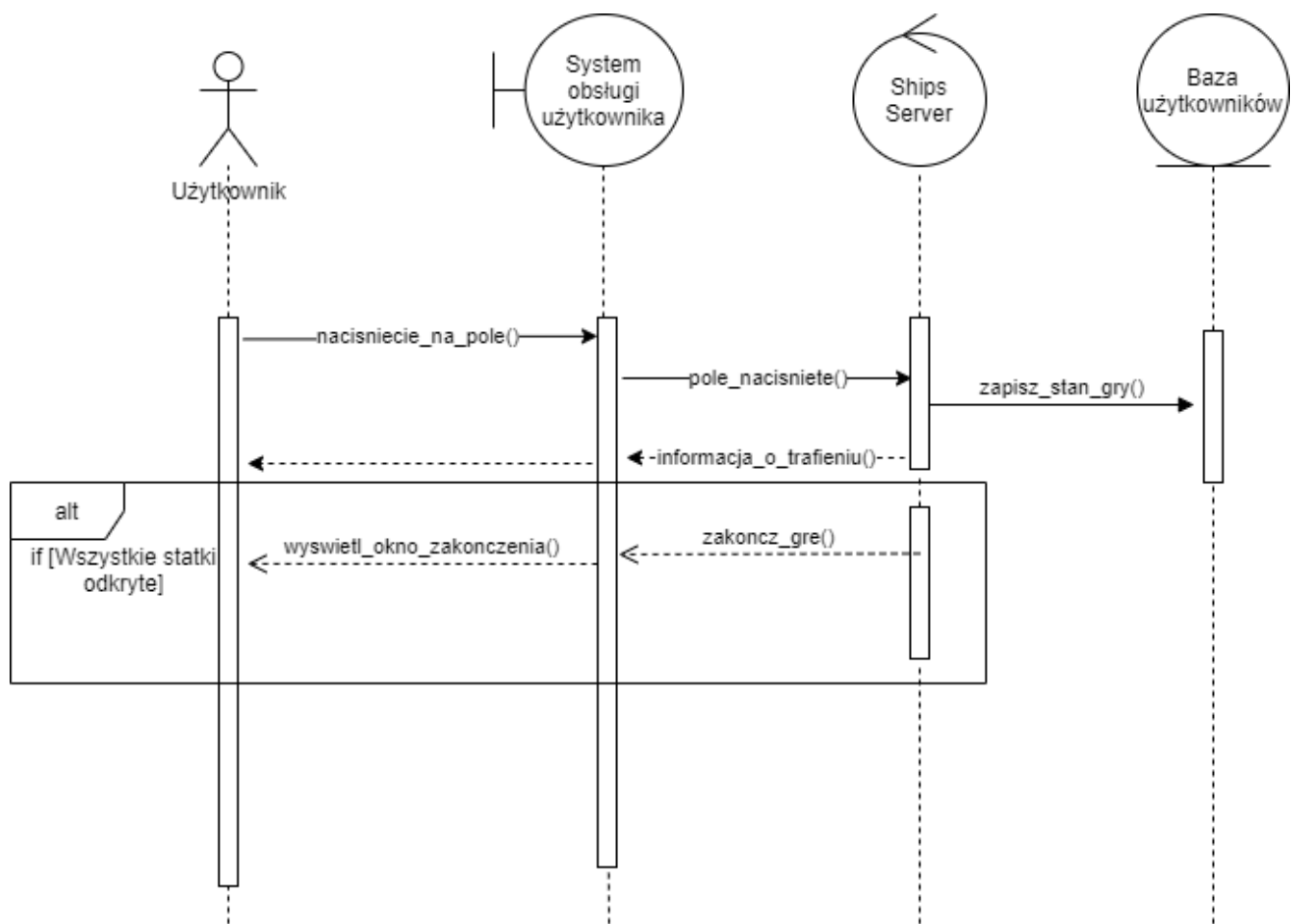


8.3. Diagramy przypadków użycia

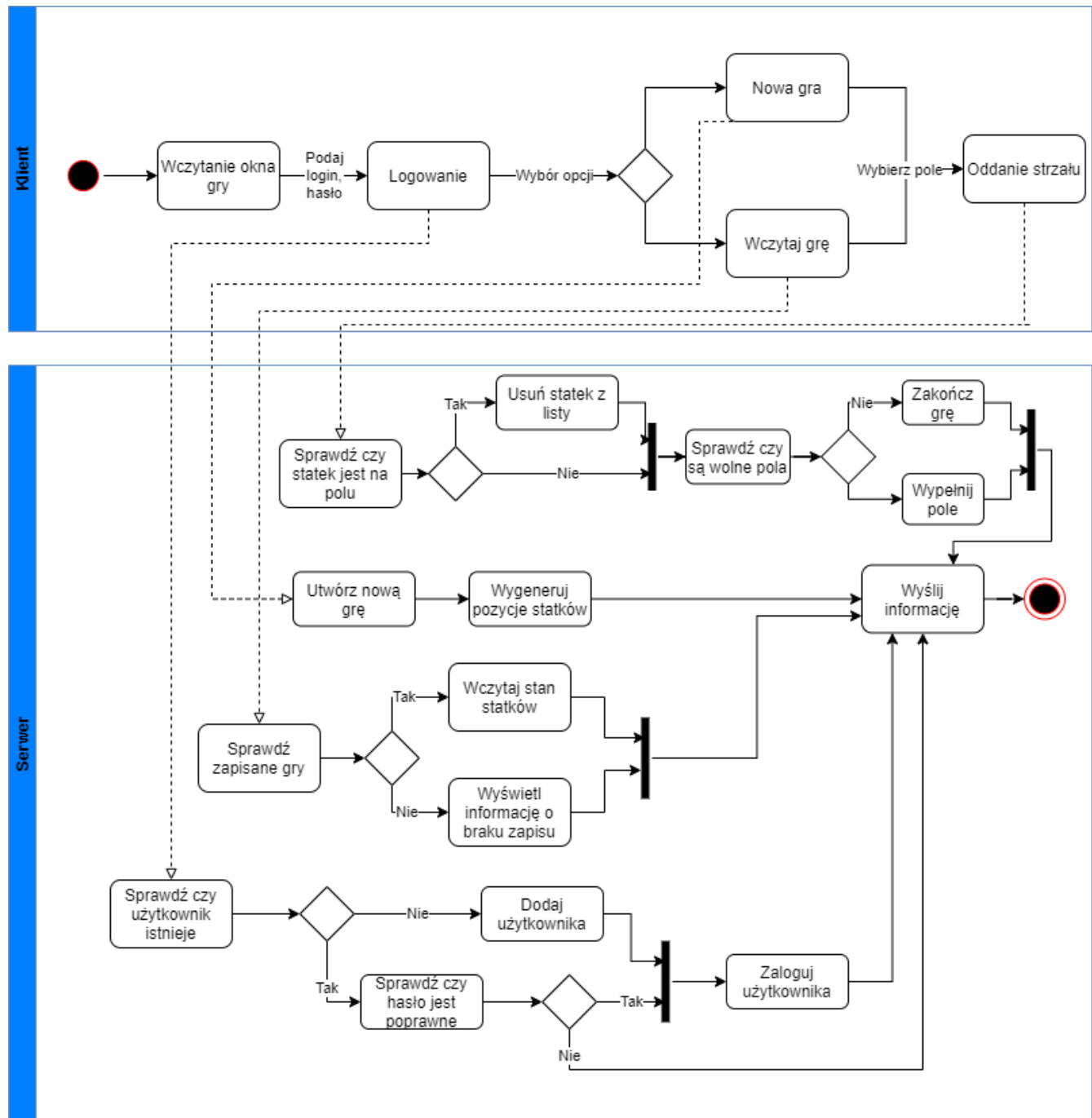


8.4. Diagramy sekwencji

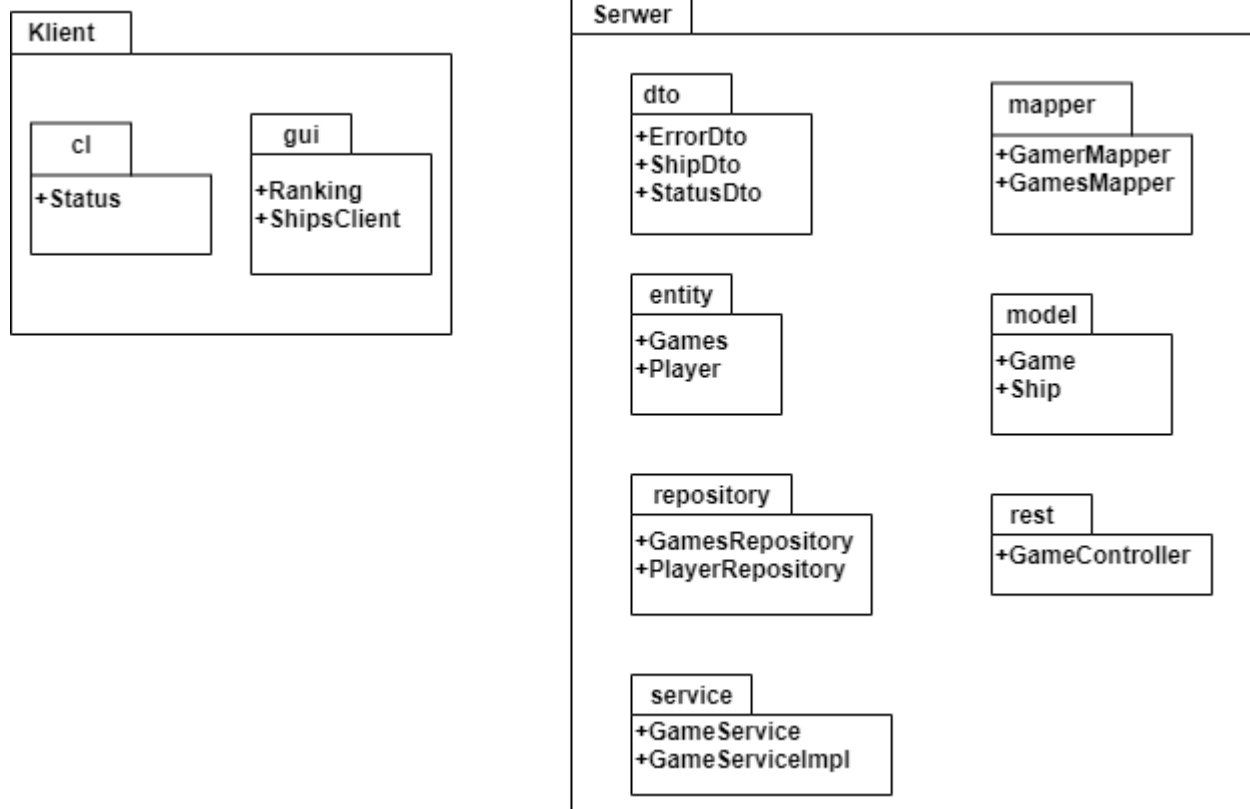




8.5. Diagramy stanów



8.6. Diagramy pakietów

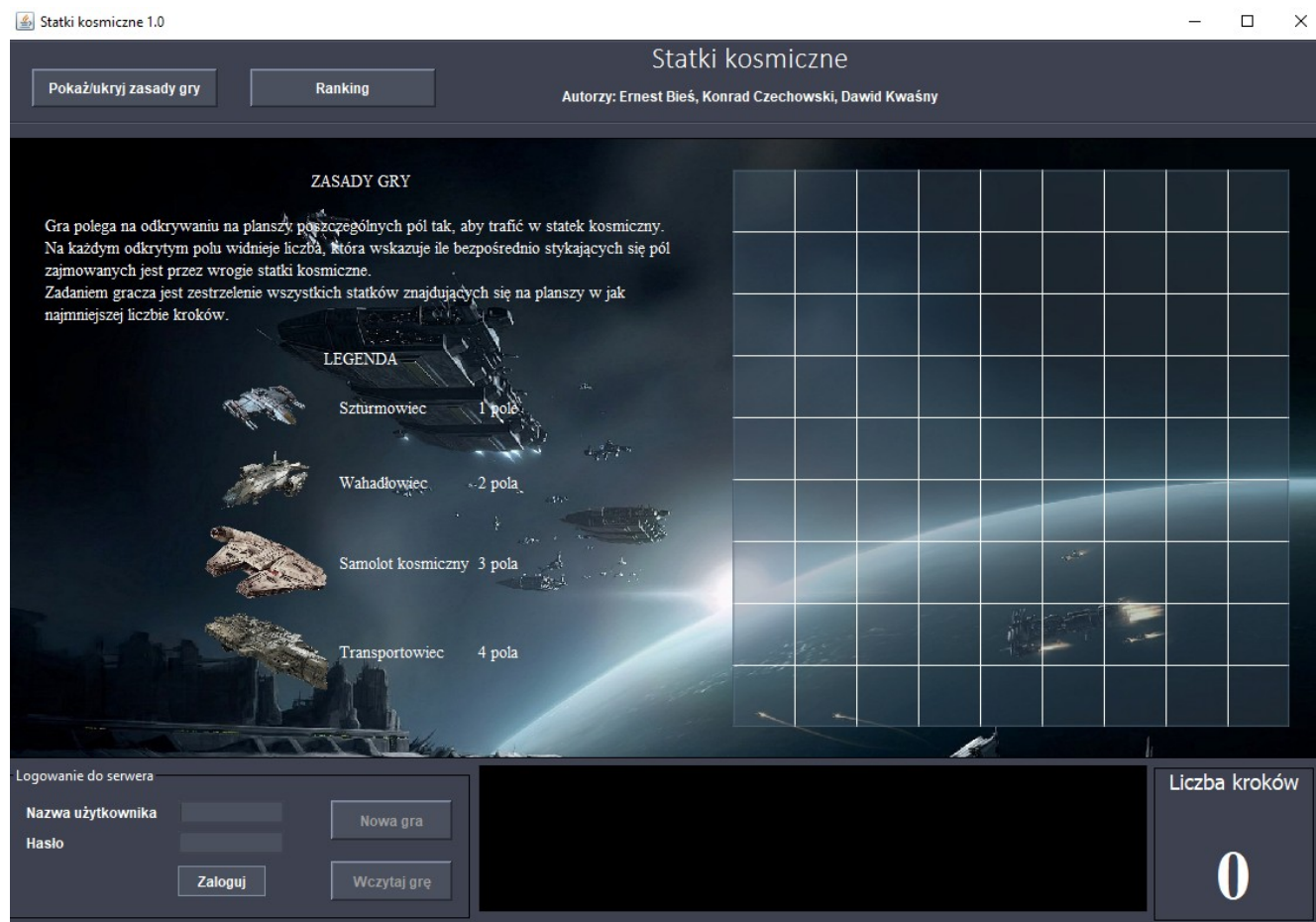


9. Opis klienta

9.1. Środowisko programistyczne

Klient gry "Statki kosmiczne" został utworzony z wykorzystaniem darmowego środowiska programistycznego **NetBeans IDE 11.2**.

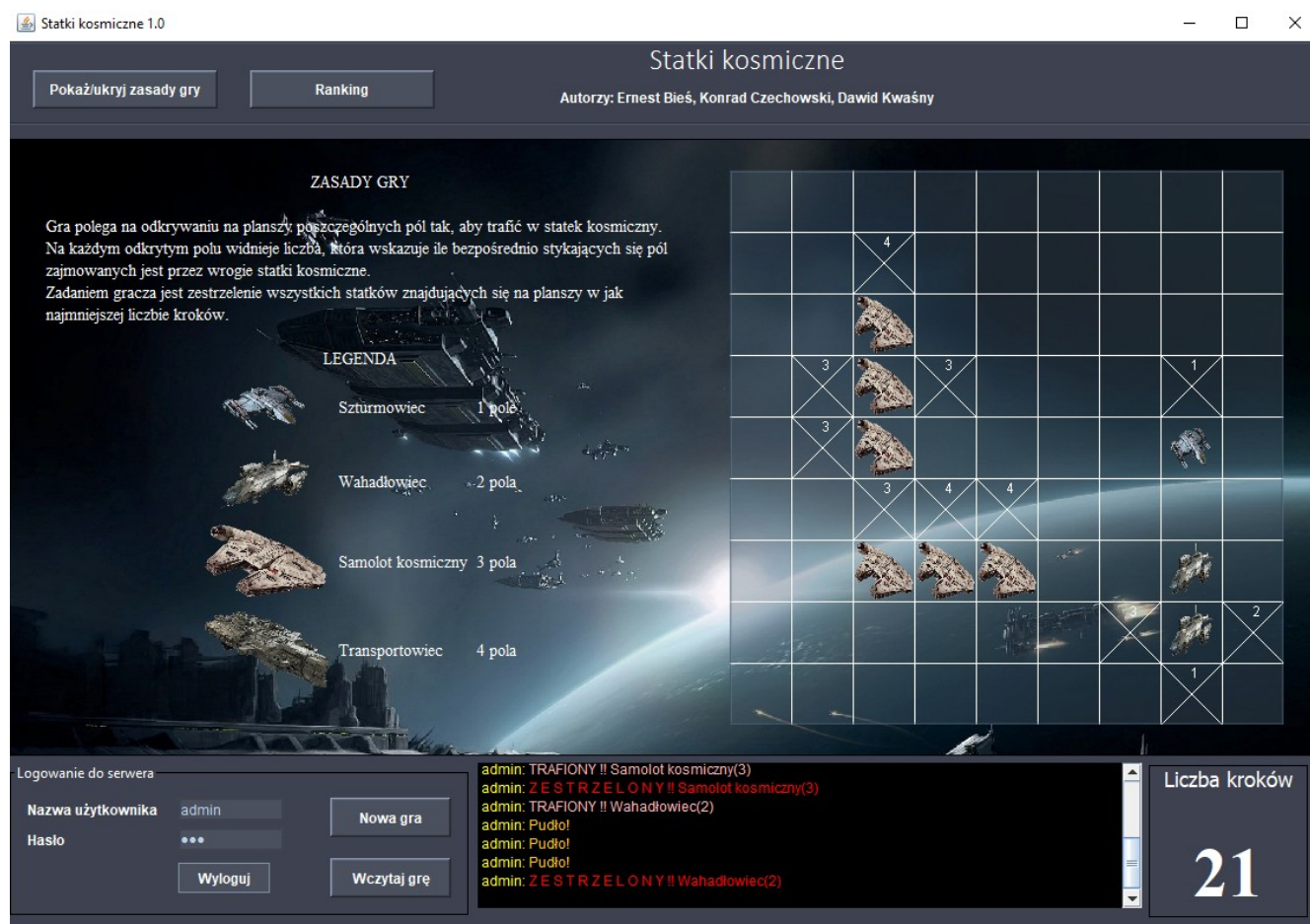
9.2. Wygląd interfejsu graficznego klienta



Rys. 9.2.1 – Okno główne programu

W głównym oknie programu po prawej stronie widoczna jest plansza o rozmiarze 9x9 kwadratów na której wyświetlane są ikony zestrzelonych statków oraz pola, które zostały

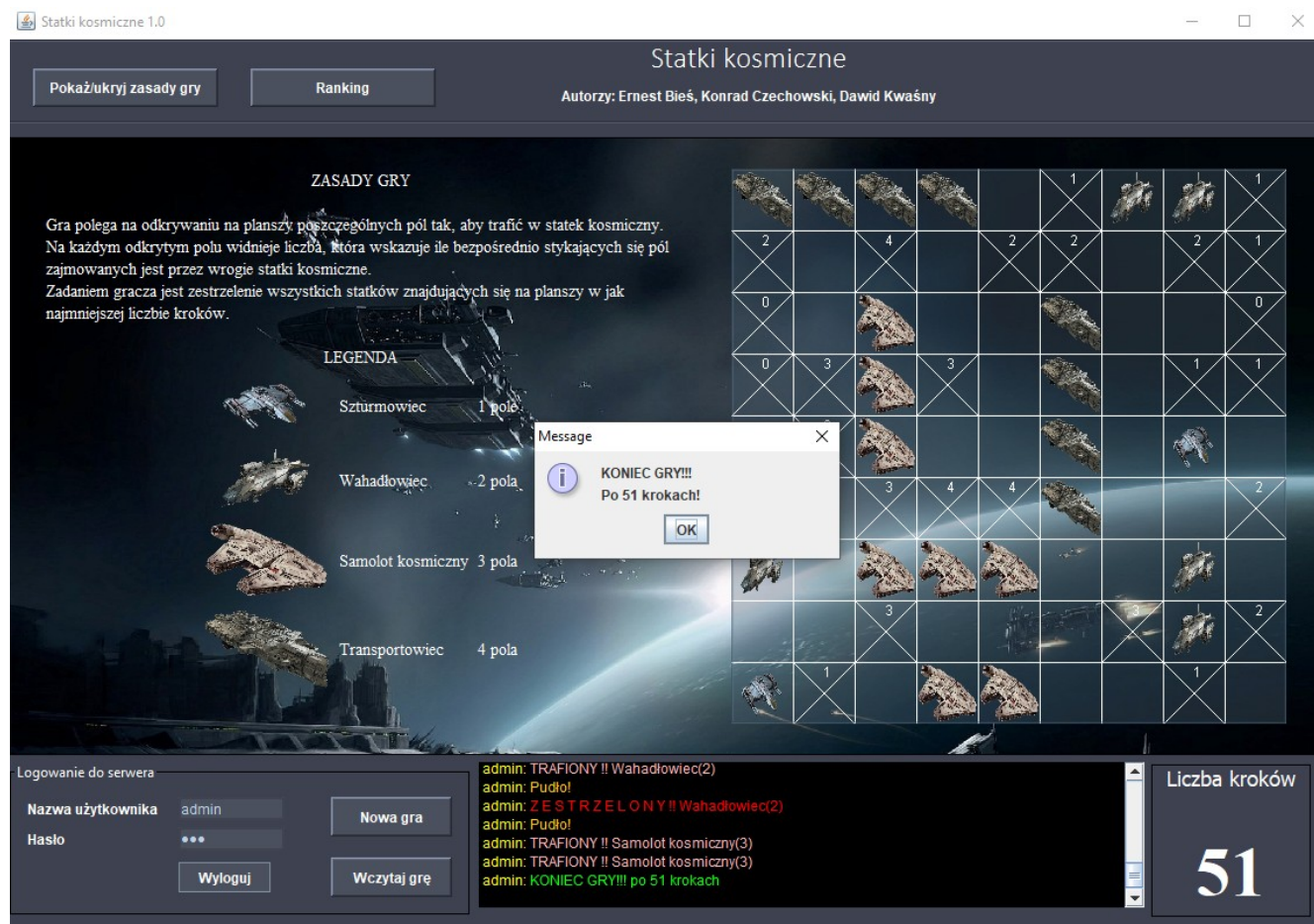
odkryte przez gracza, a nie zawierają żadnych statków. Po uruchomieniu gry po lewej stronie wyświetlone zostają podstawowe informacje dotyczące zasad gry oraz statków jakie musimy zestrzelić. W lewym górnym rogu znajduje się przycisk za pomocą którego możemy ukryć zasady gry oraz przycisk odpowiedzialny za wyświetlenie rankingu gracza. W lewym dolnym rogu okna znajdują się pola służące do wprowadzenia nazwy użytkownika i hasła oraz przyciski za pomocą których możemy zalogować się na swoje konto, utworzyć nową grę na serwerze lub wczytać już istniejącą. Na dole w środku okna znajduje się konsola na której wyświetlane są informacje dotyczące akcji podejmowanych przez gracza. Z prawej strony w dolnej części okna znajduje się licznik z aktualnie wykonanymi krokami przez gracza.



Rys. 9.2.2 – Okno w trakcie gry

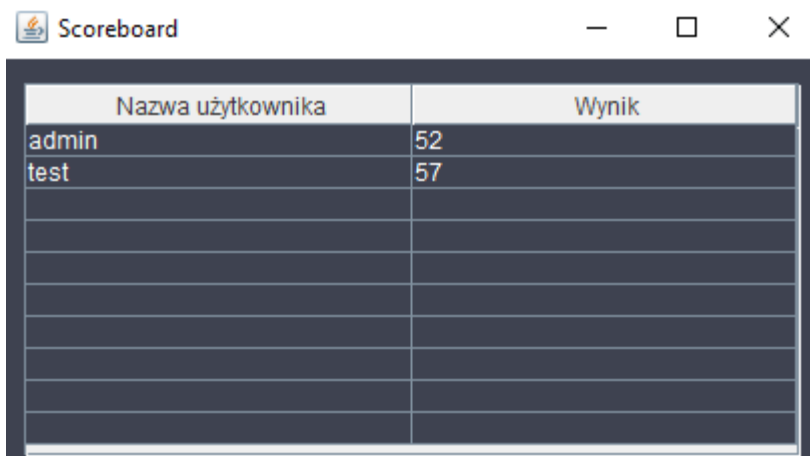
W trakcie gry użytkownik po wskazaniu myszką wybranego pola i naciśnięciu lewego przycisku myszki odsłania poszczególne pola na planszy. W przypadku trafienia statku jego ikona zostaje wyświetlona w odpowiednim miejscu na planszy, natomiast w

przypadku "pudła" we wskazanym polu pojawia się znak X oraz liczba wskazująca na ilu polach sąsiadujących ze wskazanym znajdują się statki. Równocześnie po wskazaniu pola odtwarzany jest odpowiedni dźwięk.



Rys. 9.2.3 – Zakończenie gry

Podczas zakończenia gry widoczna jest informacja o końcu gry oraz liczbie kroków w której użytkownik ukończył grę.



Nazwa użytkownika	Wynik
admin	52
test	57

Rys. 9.2.4 – Ranking graczy

Okno rankingu gracza przedstawia aktualnie najlepsze wyniki graczy oraz informacje takie jak:

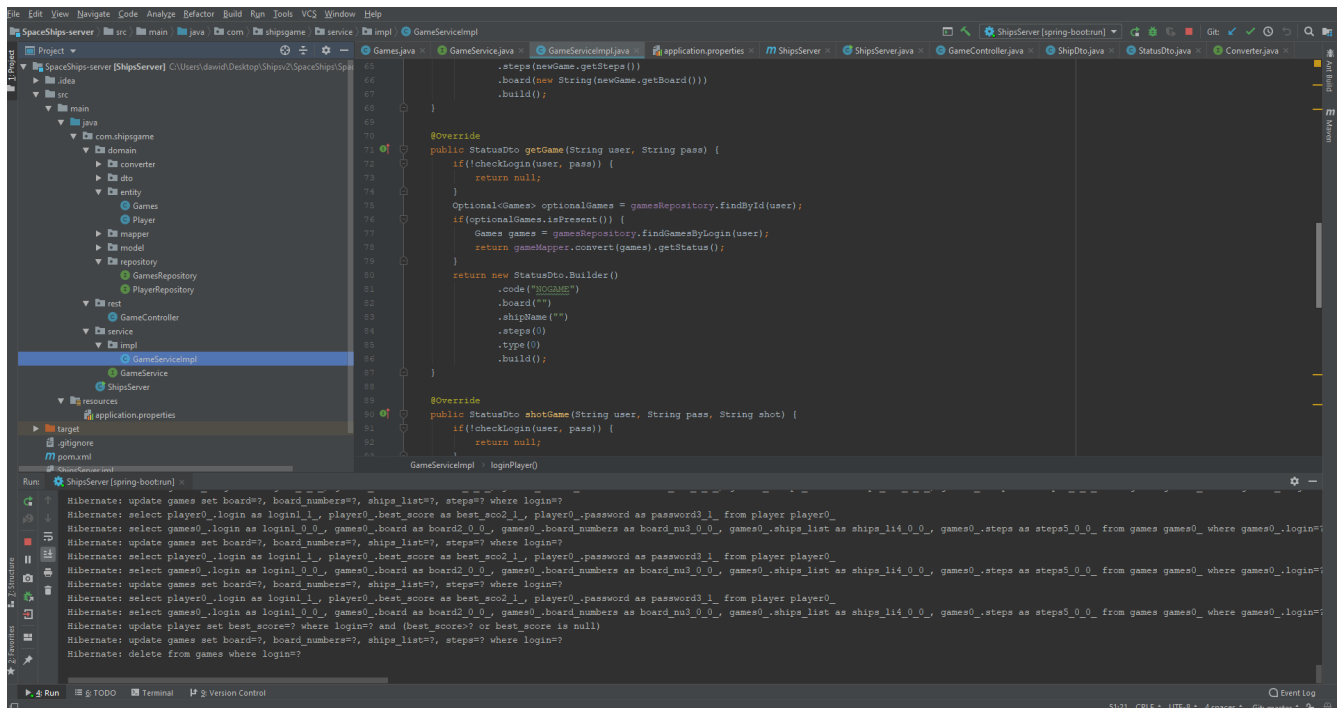
Nazwa użytkownika oraz najlepszy wynik gracza.

9.3. Instrukcja obsługi

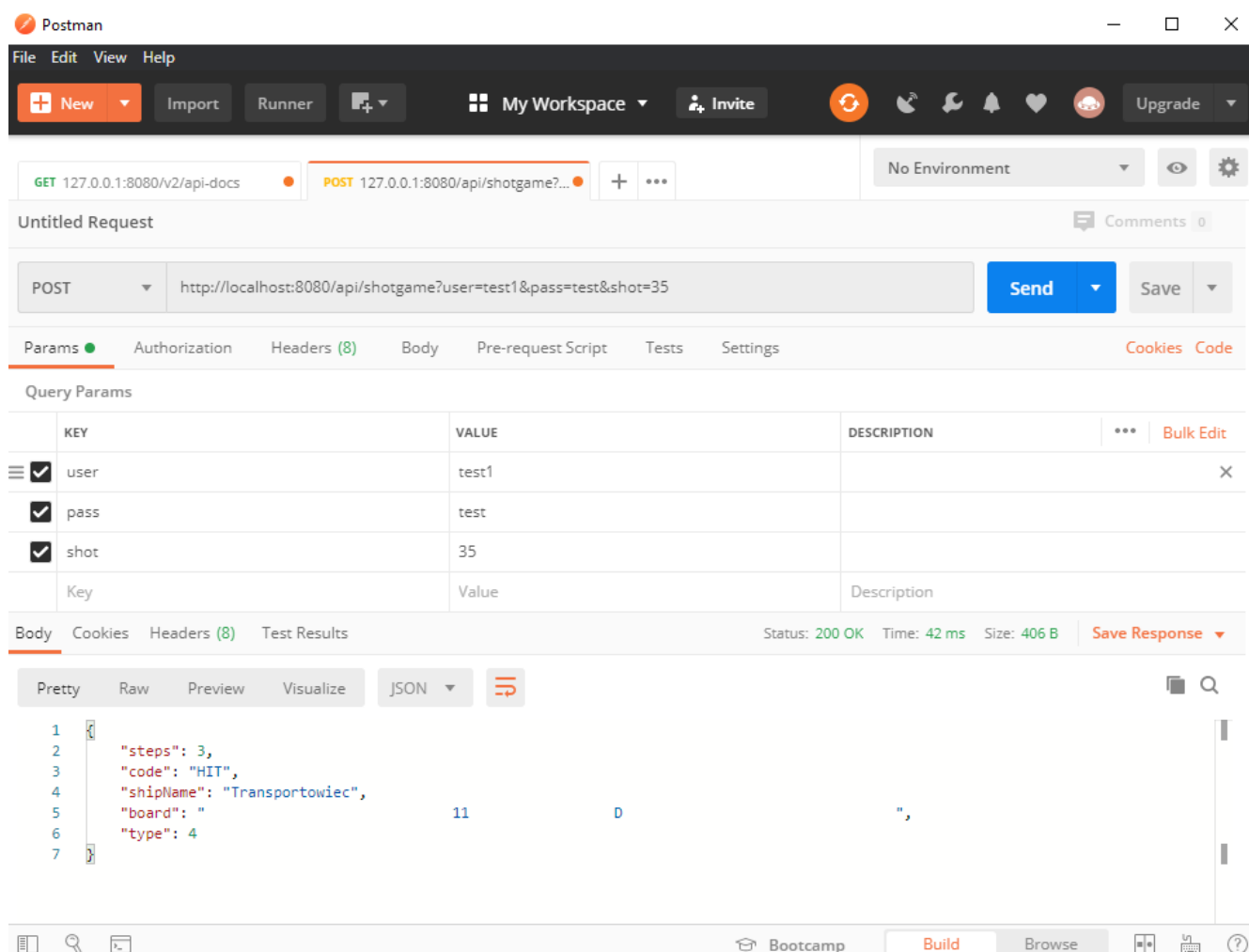
Po uruchomieniu klienta użytkownik wprowadza nazwę użytkownika oraz hasło, a następnie loguje się do serwera naciskając przycisk „Zaloguj”. W celu utworzenia nowej gry naciska przycisk „Nowa gra”. Zostaje utworzona nowa gra dla użytkownika, następnie korzystając z myszki naciska lewy przycisk na wybranym polu na planszy. Jeżeli pod odkrytym polem znajduje się statek, jego ikona zostanie wyświetlona w odpowiednim kwadracie oraz program zasygnalizuje trafienie odpowiednim dźwiękiem. Jeżeli natomiast wskazane pole nie zawiera żadnego statku to zostanie na nim wyświetlony znak X oraz liczba wskazująca na ilu polach sąsiadujących znajdują się statki. Jednocześnie za każdym wskazaniem pola zostaje zwiększony licznik znajdujący się w prawym, dolnym rogu ekranu. Jednocześnie na konsoli będą pojawiały się komunikaty dotyczące trafienia danego statku, jego nazwy lub też, że oddany strzał to niestety „pudło”. Po zestrzeleniu wszystkich statków kosmicznych wyświetlany jest komunikat z zapytaniem czy użytkownik chce zakończyć działanie programu. W przypadku udzielenia negatywnej odpowiedzi gra jest zapisywana na serwerze, natomiast klient zeruje wszelkie dane i ustawienia celem umożliwienia użytkownikowi rozpoczęcia nowej gry.

10. Proces debugowania.

W procesie debugowania wykorzystano głównie narzędzie dostępne w środowisku NetBeans IDE. W następnym punkcie przedstawiono zrzuty ekranu w czasie przeprowadzenia sprawdzenia działania klienta. Obserwacja na bieżąco zmiennej board pozwoliła śledzić zmiany na planszy i ewentualnie poprawiać błędy związane z wyświetlaniem obiektów w odpowiednim miejscu. W przypadku serwera w procesie wykorzystano narzędzie Postman. Narzędzie to bardzo ułatwiło pracę w poszukiwaniu błędów oraz testowaniu nowych funkcjonalności, w szczególności odpowiedzi serwera na przesyłane zapytania.



Rys. 10.1 – Proces debugowania



Rys. 10.2 – Testowanie zapytań z wykorzystaniem narzędzia Postman

Opis poszczególnych pozycji:

steps - liczba kroków,

code - zwracany kod gry,

shipName - nazwa statku (zwracana tylko w przypadku obsługi zapytania "shotgame"),

board - zwracana plansza jako String o długości 81, puste miejsca oznaczają, że pole nie było jeszcze sprawdzane, cyfry wskazują ile bezpośrednio stykających się pól zajmowanych jest przez wrogie statki, litery zaś oznaczają jaki statek został trafiony (A, B, C, D)

type - określa długość trafionego statku (zwracana tylko w przypadku zapytania "shotgame")