

Politechnika Świętokrzyska

Studia stacjonarne (semestr letni)

**Programowanie obiektowe(Java) -
Projekt**

2022

Skład zespołu:

Przemysław Kałuziński
Michał Kaczor
Grzegorz Kalarus

Grupa: 2ID13A

Temat projektu: Biuro podróży

Data oddania: 06.06.2022r.

Spis treści

1. Krótki opis projektu	2
2. Założenia projektowe.....	2
3. Opis interfejsu komunikacji Client-Server	3
▪ Po stronie serwera (administratora)	3
▪ Po stronie klienta (pracownika).....	8
4. Funkcjonalność aplikacji	11
5. Opis programu.....	13
6. Opis podziału pracy	21
7. Validacja danych	22
8. Testowanie aplikacji.....	23
9. Wnioski	23

1. Krótki opis projektu

Projekt miał na celu utworzenie aplikacji dla biura podróży, która będzie znacząco ułatwiać pracę jego pracownikom. Dzięki aplikacji, pracownicy mają m.in. przejrzysty podgląd klientów, wycieczek oraz mają możliwość szybkiego wystawienia faktury.

Projekt napisany został w języku programowania Java 15 z użyciem socketów i lokalnej bazy danych. Wyбралиśmy środowisko programistyczne Eclipse, co było spowodowane tym, że IntelliJ nie pozwalał nam używać Loggerów, a zmiana środowiska rozwiązała ten problem. Dodatkowo obecny kreator graficzny biblioteki Swing wydał nam się prostszy w obsłudze i zrozumieniu niż ten z konkurencyjnego środowiska. Do hostowania serwera bazy danych MySQL skorzystaliśmy z narzędzia XAMPP. Za to do utworzenia w niej tabel i encji użyliśmy framework'a Hibernate.

Poprzez implementację panelu admina mamy możliwość uruchomienia serwera, dodanie lub modyfikację pracowników, czy też samych biur podróży.

Następnym panelem jest panel pracownika, który dzięki aplikacji, może dodać ofertę biura podróży lub dodać, modyfikować klienta. Następnie może przypisać klienta do danej wycieczki, a kolejnym krokiem będzie już wystawienie umowy.

Aplikacja działa szybko, bez błędów, posiada dużą ilość opcji dla pracownika do obsługi biura, przez co mogłaby faktycznie znaleźć zastosowanie w którychś z realnych biur podróży.

2. Założenia projektowe

- rejestracja oraz logowanie pracownika
- rejestracja oraz logowanie admina
- łączenie się klienta z serwerem za pomocą socketów
- łączenie się z bazą danych
- zapisywanie danych pracownika do bazy danych
- zapisywanie danych admina do bazy danych
- zapisywanie danych wycieczek do bazy danych
- zapisywanie danych klientów do bazy danych
- dodawanie pracowników przez admina
- usuwanie pracowników przez admina
- dodawanie biura podróży przez admina
- usuwanie biura podróży przez admina
- dodawanie wycieczek przez pracowników biur podróży
- dodawanie klientów biur podróży przez pracownika
- możliwość wyboru dowolnej dostępnej wycieczki dla klienta
- wyszukiwanie klienta po peselu
- wyświetlanie wszystkich wycieczek
- wybieranie daty od kiedy do kiedy chcemy wycieczkę
- walidacja wszystkich danych

3. Opis interfejsu komunikacji Client-Server

▪ Po stronie serwera (administratora)

Aby sprawnie móc korzystać z interfejsu komunikacyjnego klient-serwer, musielibyśmy stworzyć klasę przyjmującą nowe połączenia oraz tworzącą nowe wątki. W konstruktorze tej klasy tworzymy nowe gniazdo do przyjmowania użytkowników. Gniazdo przyjmuje port przekazany mu przez parametr.

```
public class Serwer {
    private ServerSocket sock;
    private AtomicBoolean isworking = new AtomicBoolean(true);
    private AtomicBoolean issleeping = new AtomicBoolean(false);

    public Serwer(int port) throws IOException
    {
        sock = new ServerSocket(port);
    }
```

Następnie zaimplementowaliśmy wewnątrz tej klasy kilka przydatnych funkcji. Są one odpowiedzialne m.in. za:

Przyjmowanie i akceptowanie nowych połączeń

```
public void activeWaiting() throws IOException
{
    System.out.println("Aktywowano serwer, pod adresem " + sock.getInetAddress().getHostName());
    sock.setSoTimeout(50);

    while(this.isworking.get())
    {

        try
        {
            Socket socket = sock.accept();

            System.out.println("Odebrano nowe połaczenie z adresu: " + socket.getLocalAddress().getHostName());
            ClientOperator.addOperator(new Client(socket));
        }
        catch(SocketTimeoutException e)
        {
            while(this.issleeping.get());
        }
    }
}
```

Uśpienie akceptowania nowych połączeń

```
public void setSleep(boolean issleeping) {
    this.issleeping.set(issleeping);

}
```

Wyłączenie oczekiwania na połączenia

```
public void stopWaiting()
{
    this.isworking.set(false);
}
```

Zamknięcie wszystkich aktywnych połączeń

```
public void Close()
{
    try {
        ClientOperator.closeAll();
        sock.close();

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

Aby powyższe metody działały poprawnie potrzebna nam jest jeszcze jedna klasa, w której będziemy przechowywać nowo zaakceptowanego użytkownika.

```
public class ClientOperator extends Thread{
    private static List<ClientOperator> op = new ArrayList<ClientOperator>();
    private Client client;
```

Zawiera ona także metody potrzebne do zarządzania tym użytkownikiem. Między innymi metodę tworzącą nowy obiekt dla zaakceptowanego klienta oraz dodającą go do listy wszystkich aktywnych klientów.

```
public synchronized static void addOperator(@NotNull Client client)
{
    ClientOperator cop = new ClientOperator();
    cop.client = client;
    op.add(cop);
    cop.start();
}
```

Kolejną przydatną metodą z tej klasy jest metoda wymuszająca zamknięcie połączenia dla konkretnego klienta.

```
public synchronized static void removeWorker(@NotNull String nick)
{
    for(ClientOperator cl : op)
        if(cl.client.isNickCorrect(nick))
        {
            cl.client.setStop(true);
            cl.client.forceclose();
            op.remove(cl);
            return;
        }
}
```

Klasą, bez której powyższa struktura nie będzie mogła zostać poprawnie użyta jest klasa obrazująca pojedyncze połączenie klienta z serwerem. To właśnie obiekt tej klasy jest przekazywany jako parametr dla metody dodającej nowego użytkownika do listy (addOperator(Client client)). Wywoływana jest ona w metodzie ActiveWaiting() z klasy Serwer, oczekującą i akceptującą nadchodzące połączenia klientów.

```
public class Client{
    private Socket sock;
    private boolean isworking;
    private boolean stoper;
    private Worker wo = null;
    private ObjectOutputStream stream;
    private ObjectInputStream streamin;
    private static final Logger logger = Logger.getLogger(Client.class);

    public Client(@NotNull Socket sock)
    {
        this.sock = sock;
        try {
            stream = new ObjectOutputStream(sock.getOutputStream());
            streamin = new ObjectInputStream(sock.getInputStream());
        } catch (IOException e) {
            logger.error("Error!", e);
        }
    }
}
```

Klasa zawiera także szereg innych metod obsługujących wymianę danych z klientem. Przykładem może być poniższa metoda, która służy do wysyłania klientowi obiektu zawierającego umowy wycieczkowe.

```
public void AggrementOperatorGet(@NotNull com.Kaczor.Kaluzinski.Kalarus.Communication.Aggrement ag) throws IOException
{
    com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Aggrement[] aggremets = DataBaseManager.AgrrementManager.getAggements(wo.getAgreement());
    if(aggremets == null)
    {
        Aggrement buff1 = new Aggrement(true, "Error!!!");
        buff1.setWhatToDo(WhatToDo.TRIPS_ERROR);
        stream.writeObject(buff1);
        return;
    }
    for(com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Aggrement buff : aggremets)
    {
        Payment pay = DataBaseManager.PaymentManager.getPaymentByAggrement(buff.getIdAggrement());
        com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Trip tr = DataBaseManager.TripManager.getTrip(buff.getIdTrip());
        if(pay == null || tr == null)
        {
            Aggrement buff1 = new Aggrement(true, "Error!!!");
            buff1.setWhatToDo(WhatToDo.TRIPS_ERROR);
            stream.writeObject(buff1);
            return;
        }
        Aggrement buffer = new Aggrement(buff.getIdAggrement(),buff.getIdTrip(), 0, 0, 0, buff.getGuestcount(), 0, null, null,
        stream.writeObject(buffer));
    }
    stream.writeObject(new Aggrement(true, null));
}
```

Serwer tworzymy w metodzie z klasy głównej (Window.java) poprzez wywołanie metody prepareServer(). Uruchamia ona metodę nasłuchującą nadchodzących połączeń.

```
private void prepareServer()
{
    if(thread == null)
    {
        try {
            ser = new Serwer(4021);
        } catch (IOException e2) {
            e2.printStackTrace();
        }
        thread = new Thread(()->
        {
            try {
                ser.activeWaiting();
            } catch (IOException e1) {
                Logger.error("ERROR!", e1);
            }
        });
    }
}
```

Następnie potrzebujemy czegoś zapewniającego nam komunikację. W tym celu wykorzystujemy interfejs Communication, który zapewnia trzy podstawowe metody wykorzystywane przez wszystkie inne klasy komunikacyjne. Pierwsza z nich zwraca wiadomość błędu, druga zwraca true jeśli wystąpił błąd, a ostatnia zwraca rodzaj operacji wybrany z typu wyliczeniowego enum.

```
public interface Communication extends Serializable{

    public String getMessage();

    public boolean isError();

    public @NotNull WhatToDo getWhatToDo();
}
```

Wspomniany wyżej typ wyliczeniowy enum służy nam do określania jaką operację zamierzamy wykonać na obiekcie. Np. operacja GET powiadamia serwer, że klient prosi o zwrócenie konkretnych danych.

```
public enum WhatToDo {
    GET,
    ADD,
    UPDATE,
    REMOVE,
    GET_TRIPS,
    TRIPS_ERROR
}
```

Z dwoma powyższymi punktami bezpośrednio wiąże się zestaw klas komunikacyjnych. To w tym miejscu ulokowaliśmy metody służące do przesyłania konkretnych danych związanych z używaniem aplikacji przez klienta. Przykładem może być klasa Client.java, która zawiera szczegółowe dane osobowe na temat dodanego użytkownika. Klasa nazywa się identycznie jak poprzednia, ale spełnia zupełnie inne zadanie. Poprzednia odpowiadała obsłudze informacji połączeniowych klienta (np. socket), a ta służy do przechowywania danych użytkowych (np. dane osobowe). Dzięki różnym paczkom, w których owe klasy się znajdują, nie występuje konflikt nazw.

```
public class Client implements Communication {
    private static final long serialVersionUID = 272484732295243662L;
    private String name, surname, phoneNumber, pesel, country, city, street, homenumber, postcode, message;
    private WhatToDo what;
    private int idClient;
    private boolean isError = false;

    public Client(int idClient,@Nullable String name,@Nullable String surname,@Nullable String phoneNumber,@NotNull String pesel,
                 @Nullable String street,@Nullable String homenumber,@Nullable String postcode,@NotNull WhatToDo what) {
        this.name = name;
        this.surname = surname;
        this.phoneNumber = phoneNumber;
        this.pesel = pesel;
        this.country = country;
        this.city = city;
        this.street = street;
        this.homenumber = homenumber;
        this.postcode = postcode;
        this.what = what;
        this.idClient = idClient;
    }
}
```

Tego typu klasy służą nam do tworzenia obiektów do manipulacji przechowywania danych użytkowych z całej aplikacji.

■ Po stronie klienta (pracownika)

Po stronie użytkownika sytuacja jest o wiele prostsza, gdyż do komunikacji z serwerem wykorzystujemy tylko jedną klasę. W jej konstruktorze tworzone jest nowe gniazdo użytkownika do komunikacji z serwerem oraz dwa strumienie – wejściowy i wyjściowy.

```
public class SerwerConnector {
    private static final Logger Logger = Logger.getLogger(SerwerConnector.class);
    private Socket sock;
    private ObjectOutputStream stream;
    private ObjectInputStream streamin;
    public SerwerConnector() throws UnknownHostException, IOException
    {
        sock = new Socket("127.0.0.1", 4021);
        stream = new ObjectOutputStream(sock.getOutputStream());
        streamin = new ObjectInputStream(sock.getInputStream());
    }
}
```

Zawiera ona dwie bardzo ważne metody, które zamykają lub inicjalizują połączenie z serwerem. Obie zwracają false w przypadku niepowodzenia.

```
public boolean close()
{
    try {
        this.sock.close();
    } catch (IOException e) {
        Logger.error("Error!", e);
        return false;
    }
    return true;
}

public boolean open()
{
    try {
        this.sock = new Socket("127.0.0.1", 4021);
    } catch (IOException e) {
        Logger.error("Error!", e);
        return false;
    };
    return true;
}
```

Znajduje się tu także zestaw metod, które mają za zadanie wykonywanie odpowiednich operacji wymaganych do zapewnienia spójności danych podczas korzystania z aplikacji. Oto kilka przykładowych metod:

Wysyła żądanie do serwera o zalogowanie użytkownika

```
public boolean login(@NotNull String nick,@NotNull String password)
{
    try {

        stream.writeObject(new WorkerLogin(nick,password));
        Object obj = streamin.readObject();
        if(obj instanceof Communication)
        {
            if(((Communication)obj).isError())
            {
                showErrorMessage(((Communication)obj).getMessage());
                return false;
            }
            else
            {
                showInfoMessage(((Communication)obj).getMessage());
                return true;
            }
        }
        else
            return false;
    } catch (IOException e) {
        Logger.error("Error!", e);
        return false;
    } catch (ClassNotFoundException e) {
        Logger.error("Error!", e);
        return false;
    }
}
```

Aktualizuje płatność klienta naszego biura

```
public boolean updatePayment(@NotNull Aggrement ag)
{
    ag.setWhatToDo(WhatToDo.UPDATE);
    try {
        stream.writeObject(ag);
        Object obj = streamin.readObject();
        if(!(obj instanceof Communication))
        {
            showErrorMessage("Nieprawidłowy obiekt!");
            return false;
        }
        Communication com = (Communication)obj;
        if(com.isError())
        {
            showErrorMessage(com.getMessage());
            return false;
        }
        showInfoMessage(com.getMessage());
        return true;
    } catch (IOException | ClassNotFoundException e) {
        Logger.error("Error!", e);
    }

    return false;
}
```

Dodaje wycieczkę do bazy danych

```
public boolean addTrip(@NotNull Trip of)
{
    try {
        of.setWhatToDo(WhatToDo.ADD);
        stream.writeObject(of);
        Object obj = streamin.readObject();
        if(obj instanceof Communication)
        {
            Communication com = (Communication)obj;
            if(com.isError())
            {
                showErrorMessage(com.getMessage());
                return false;
            }
            else
            {
                showInfoMessage(com.getMessage());
                return true;
            }
        }
    } catch (IOException | ClassNotFoundException e) {
        Logger.error("Error!", e);
    }

    return false;
}
```

Na koniec w metodzie main() aplikacji klienta tworzony jest nowy obiekt klasy SerwerCOnnector(). Pozwala to na nawiązanie połączenia między bieżącym użytkownikiem, a serwerem.

```
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                @SuppressWarnings("unused")
                Window window = new Window();
                Window.frame.setVisible(true);
                connector = new SerwerConnector();
            } catch (Exception e) {
                Logger.error("Error!", e);
            }
        }
    });
}
```

Plikami głównymi, służącymi do uruchomienia obydwu aplikacji są pliki **Window.java** znajdujące się w poniższych paczkach:

Klient: com.Kaczor.Kaluzinski.Kalarus.Klient.*;

Serwer: com.Kaczor.Kaluzinski.Kalarus.server.*;

4. Funkcjonalność aplikacji

Panel admina (serwer):

- podstrona do logowania się jako admin
- przycisk do włączania serwera aby klient mógł się połączyć z serwerem
- podstrona obsługi użytkownika, gdzie możemy dodać lub modyfikować pracowników danego biura oraz ich usuwać
- podstrona obsługi biur podróży, gdzie możemy dodać lub modyfikować dane biuro oraz je usunąć

Panel pracownika (klient):

- podstrona do logowania pracowników
- podstrona, gdzie możemy wybrać dodanie oferty, operacje związane z klientem oraz wylogowanie się z naszego konta
- podstrona z dodawaniem oferty wycieczki
- podstrona z wyborem: dodawania, modyfikacji i usuwania użytkowników oraz z operacjami na dodanych już wycieczkach
- podstrona z dodawaniem i edycją klienta
- podstrona z wyświetlaniem wycieczek, wystawianiem umów, wybieraniem wycieczki oraz znajdowaniem danego klienta po peselu

5. Tworzenie bazy danych (Hibernate)

Baza danych tworzona jest w oparciu o framework Hibernate służący do wymiany danych pomiędzy relacyjną bazą danych, a światem obiektowym naszego projektu. Jego konfiguracja odbywa się w pliku persistence.xml, gdzie wskazujemy wszystkie klasy zawierające implementację tabel oraz encji. Dzięki odpowiednim poleceniom, baza danych jest tworzona tylko raz, a jeśli już istnieje to jest aktualizowana. Pozwala to na zachowanie zapisanych w niej wcześniejszych danych nawet po zamknięciu aplikacji.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2<persistence version="2.0"
3   xmlns="http://java.sun.com/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
7<persistence-unit name = "Server" transaction-type = "RESOURCE_LOCAL">
8   <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
9   <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Aggrement</class>
10  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Adresy</class>
11  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Worker</class>
12  <class>com.Kaczor.Kaluzinski.Kalarus.Database.Client</class>
13  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Payment</class>
14  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Trip</class>
15  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.Office</class>
16  <class>com.Kaczor.Kaluzinski.Kalarus.Serwer.Database.AdminLogin</class>
17<properties>
18   <property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
19   <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/baza" />
20   <property name="javax.persistence.jdbc.user" value="root" />
21   <property name="javax.persistence.jdbc.password" value="" />
22   <property name="hibernate.hbm2ddl.auto" value="update"/> <!-- create tworzy za kazdym -->
23 </properties>
24 </persistence-unit>
25 </persistence>
```

Implementację frameworka umożliwiły nam zależności narzędzia Maven. Wystarczyło dodać je w klauzuli <dependencies> </dependencies> pliku pom.xml.

```

<dependency>
  <groupId>org.jetbrains</groupId>
  <artifactId>annotations</artifactId>
  <version>20.1.0</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.5.Final</version>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.5.2</version>
</dependency>
```

6. Opis programu

Szczegółowe informacje dotyczące kodu programu, wszystkich klas, metod itd. znajdują się w dokumentacji wygenerowanej przez program Doxygen dołączonej do sprawozdania.

Poniżej zostaną przedstawione przykładowe screeny z działania programu.

Dodanie pierwszego administratora

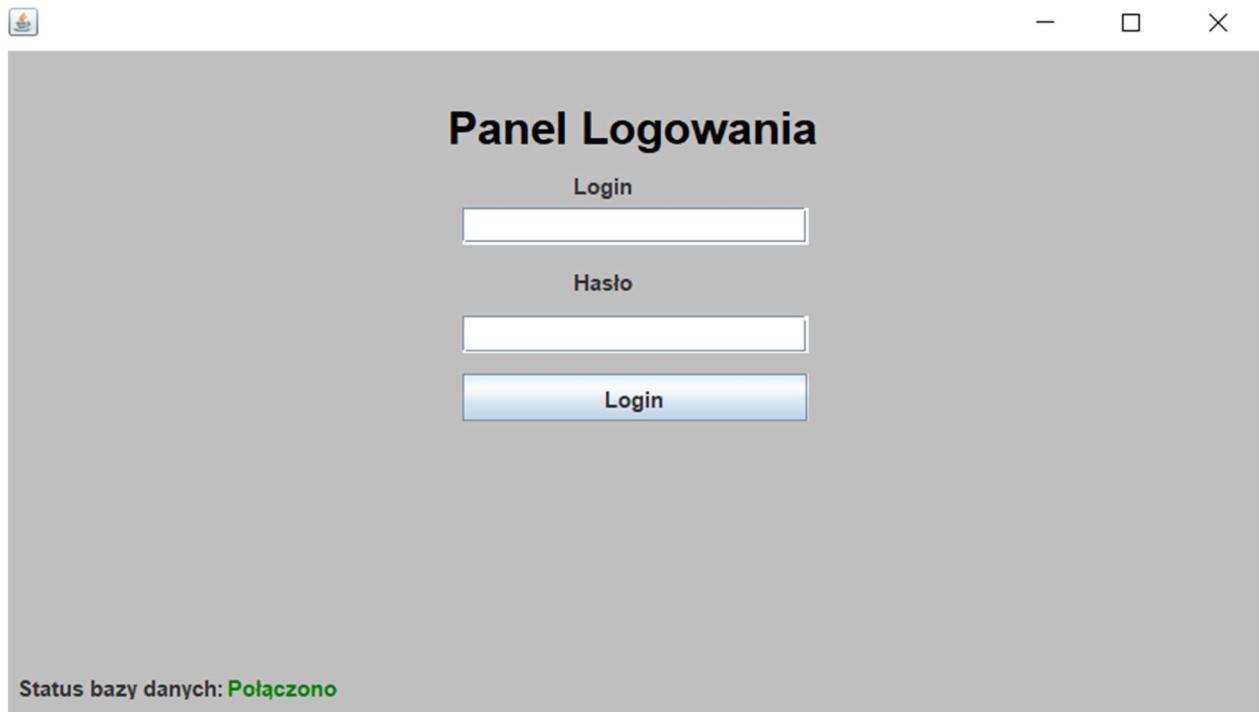
Zanim zalogujemy się do serwera jako admin musimy dodać jego konto w sposób manualny – z poziomu bazy danych. Na początek uruchamiamy aplikację XAMPP i włączamy serwer MySQL. Później należy skompilować i uruchomić aplikację serwera w celu stworzenia wszystkich encji. Następnie przy pomocy przeglądarki wchodzimy do nowej bazy danych (localhost -> phpMyAdmin). Teraz musimy dodać dane logowania do tabeli panel. Robimy to przy pomocy polecenia SQL INSERT pokazanego poniżej. Login i hasło są dowolne, lecz gdy przypiszemy im domyślne wartości (admin, admin) to aplikacja poprosi nas o ich zmianę.

The top screenshot shows the phpMyAdmin interface for the 'baza' database. The left sidebar lists various tables: Nowa, adresy, adresyseq, biura, biuraseq, klienci, klienciseq, panel, platnosci, platnosciseq, pracownicy, pracownikseq, umowy, umowysq, wycieczki, and wycieczkiseq. The right pane displays the 'panel' table structure and data. The table has columns: Tabela, Działanie, Rekordy, Typ, Metoda porównywania napisów, Rozmiar, and Nadmiar. There are 24 records, all of type InnoDB and utf8mb4_general_ci. The bottom screenshot shows a terminal window with the following SQL command:

```
1 INSERT INTO panel VALUES ('admin', 'admin1234');
```

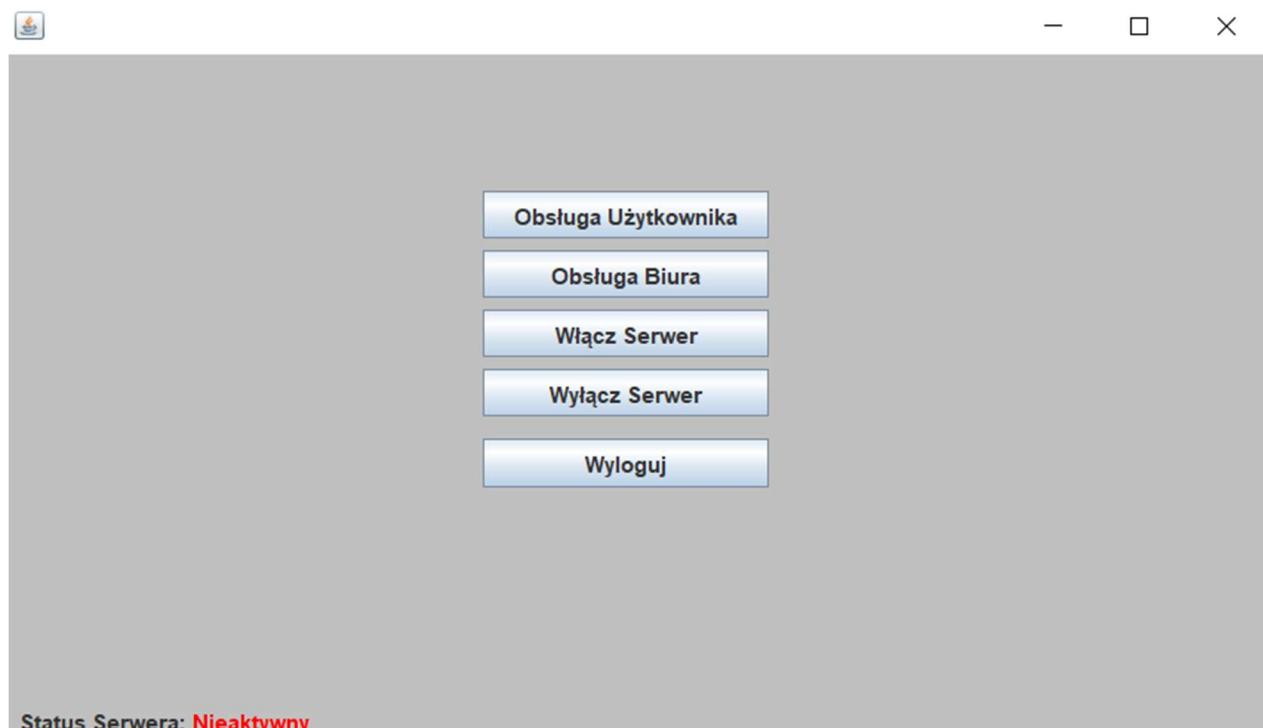
Logowanie do aplikacji admina:

Po dodaniu danych logowania administratora możemy na nowo uruchomić aplikację serwera. Ukaże nam się panel logowania. Aby przejść dalej należy podać wcześniej ustalone dane.



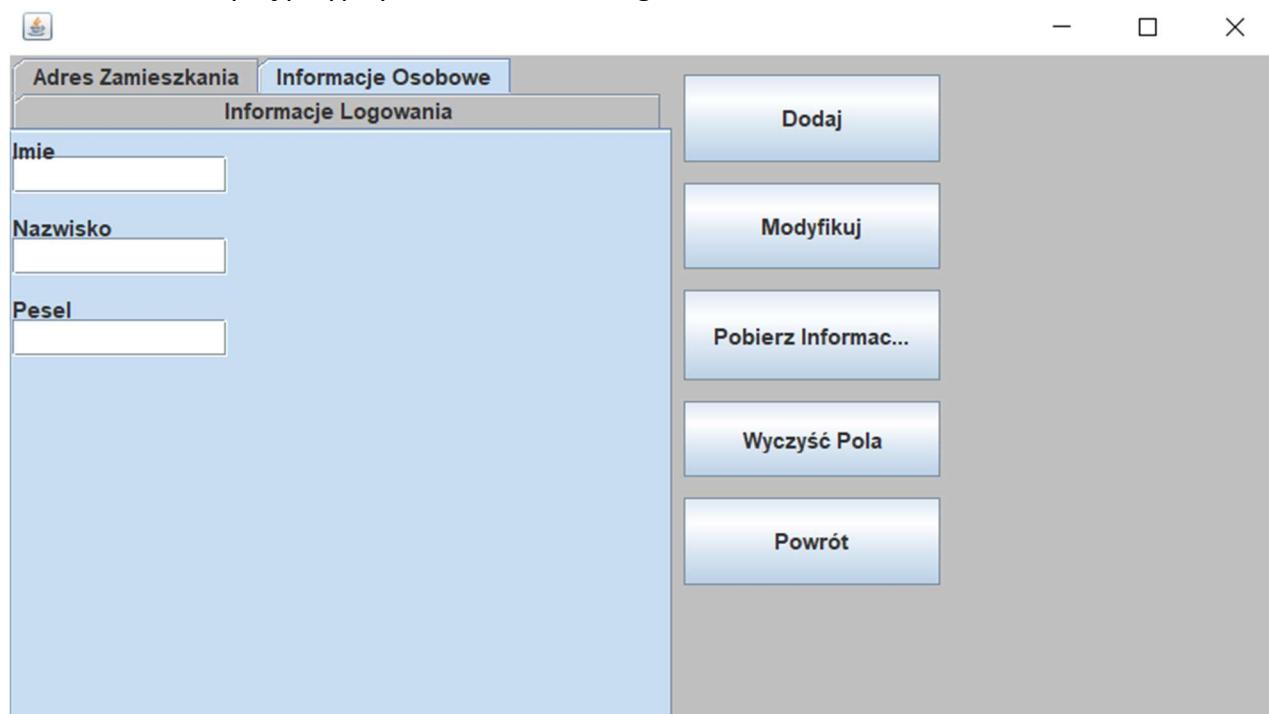
Menu opcji admina:

Jeśli podane powyżej dane logowania będą poprawne aplikacja przeniesie nas do głównego MENU administratora. Jak można zauważyć w lewym dolnym rogu - serwer jest nieaktywny. Znaczy to, że do czasu włączenia go przy pomocy przycisku „Włącz serwer” dołączanie klientów i tworzenie dla nich gniazd nie jest obsługiwane.



Podstrona, gdzie dodajemy lub modyfikujemy pracownika:

Po wybraniu opcji „Obsługa użytkownika” przeniesiemy się do panelu, w którym możemy dodać nowego pracownika do bazy danych. Za pomocą zakładek dane podzielone są na zestawy dotyczące poszczególnych rzeczy, np. dane osobowe, adres oraz dane logowania. Możemy również pobrać informacje istniejącego już użytkownika z bazy danych. W tym celu należy wpisać jego pesel i kliknąć przycisk „Pobierz informacje”. Teraz możemy zmodyfikować jego dane i zapisać je spowrotem do bazy danych. Każdy pracownik musi mieć przypisane biuro podróży, w którym pracuje, dlatego jeśli zaczynamy pracę z aplikacją powinniśmy najpierw dodać właśnie biuro. Pracownicy są przypisywani do NIPu danego biura.



Podstrona, gdzie dodajemy lub modyfikujemy biuro:

Po wybraniu opcji „Obsługa biura” przeniesiemy się do panelu, gdzie możemy dodać nowe miejsce pracy dla naszych pracowników. Podobnie jak w przypadku pracowników i tutaj możemy dodawać lub modyfikować informacje zawarte w bazie danych. Numerem identyfikującym biuro jest NIP. Jeśli wszystkie dane będą spełniać wyznaczone ograniczenia, to będziemy mogli przeprowadzić operację.

NIP
123456789012345678

Nazwa
Nazwa Biura

Ulica
Ulica Biura

Kraj
Kraj Biura

Kod pocztowy
Kod pocztowy Biura

Numer Domu
Numer Domu Biura

Miejscowość
Miejscowość Biura

Dodaj

Pobierz Informacje

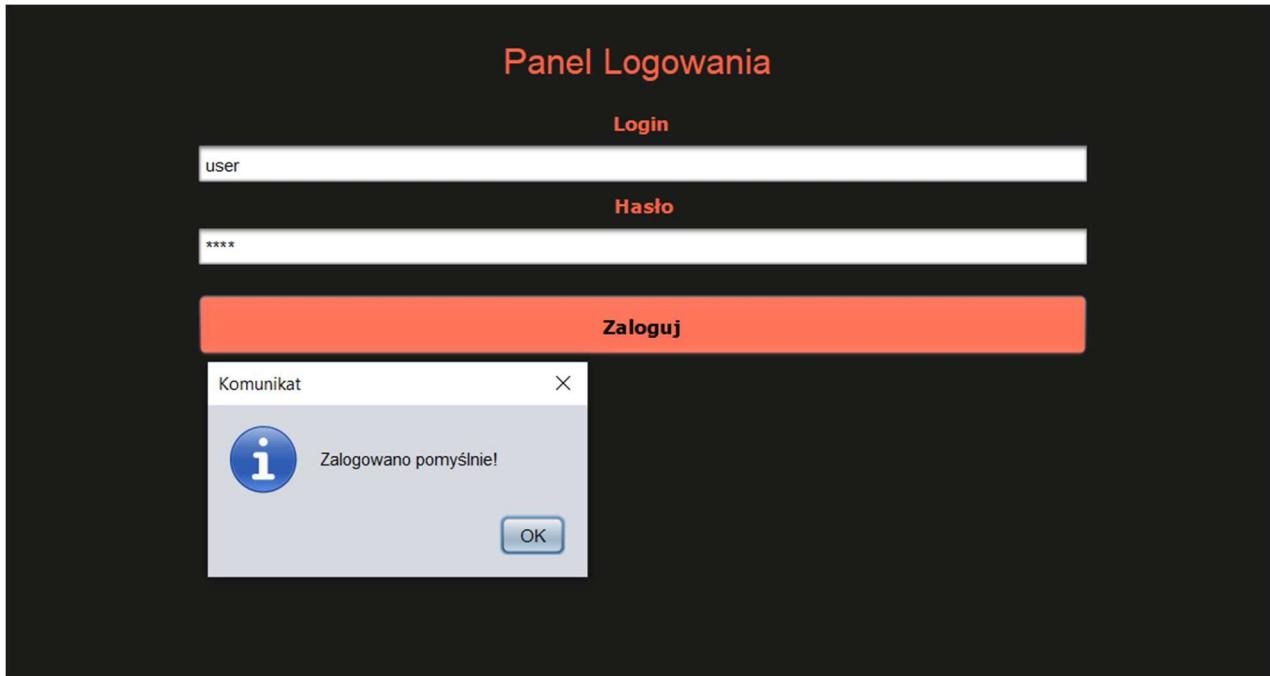
Wyczyść Pola

Modyfikuj

Powrót

Logowanie do aplikacji pracownika:

Gdy wykonaliśmy już wszystkie powyższe czynności w panelu administratora, możemy przejść do aplikacji pracownika. Tutaj również powita nas ekran logowania, gdzie możemy zalogować się przy pomocy loginu i hasła, które przypisaliśmy użytkownikowi przy jego tworzeniu w panelu admina. Gdy informacje będą poprawne, wyświetli się okno dialogowe o pomyślnym zalogowaniu.



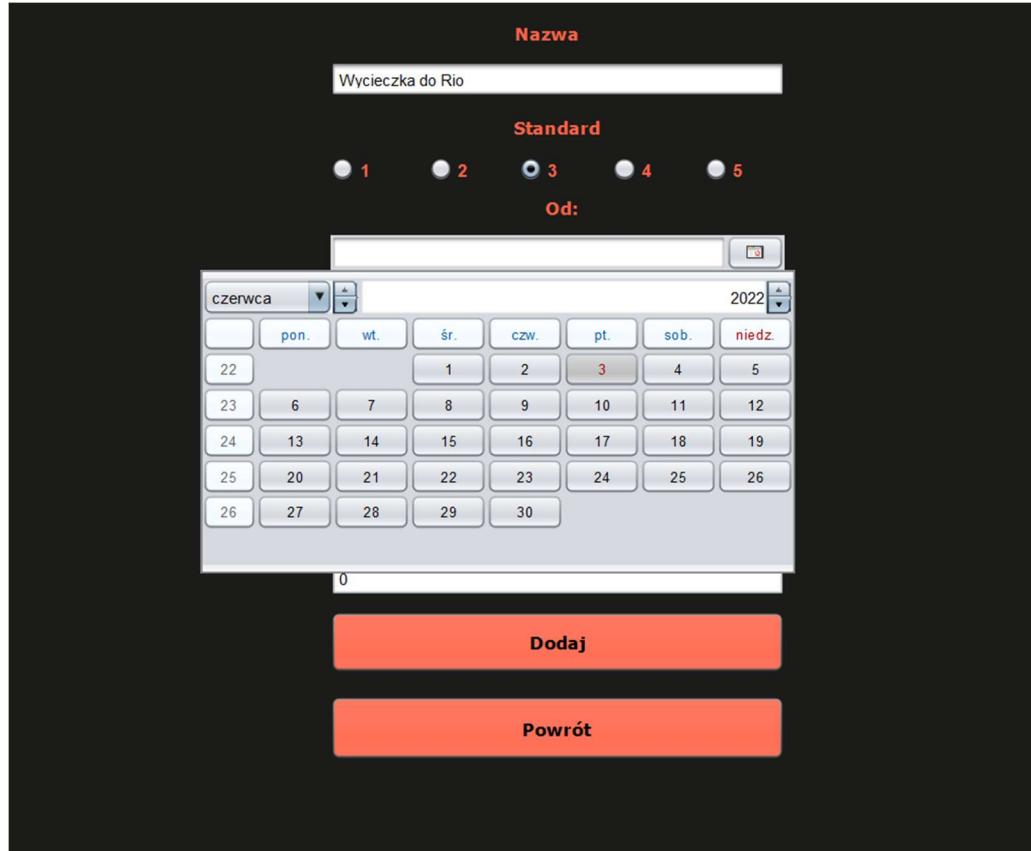
Menu główne pracownika:

Jeśli logowanie przebiegnie pomyślnie przeniesiemy się do głównego MENU użytkownika, gdzie wybieramy na jakich danych chcemy przeprowadzić operację.



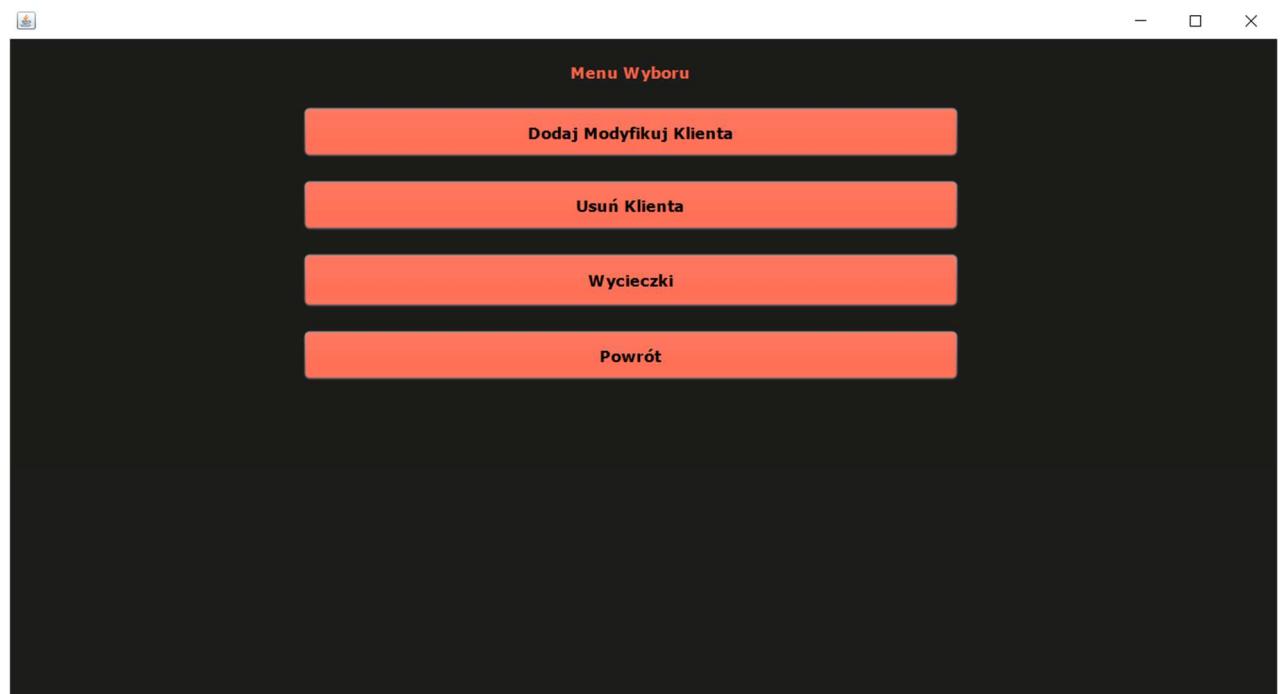
Dodawanie wycieczek:

Przycisk „Dodaj ofertę” przeniesie nas do panelu, w którym możemy stworzyć nową ofertę wycieczkową dostępną dla klientów.



Menu opcji pracownika:

Po kliknięciu przycisku „Klient” napotkamy MENU z wyborem wszystkich opcji dotyczących manipulacji danymi klientów.



Dodawanie i modyfikacja klientów:

Przechodząc dalej, mamy funkcję dodawania nowego i modyfikacji istniejącego już klienta. Przyciski obsługi tego panelu pozwalają nam na utworzenie nowego zestawu danych dla klienta biura w bazie danych lub jego modyfikację. Do identyfikacji klienta wykorzystaliśmy jego numer PESEL. Jeśli osoba o takim numerze już figuruje w bazie danych, to zostaną wyświetlane wszystkie informacje na jej temat. Następnie będziemy mogli je zmodyfikować i zapisać.

Dane Klienta

Imię
Wiesiek

Nazwisko
Bączek

PESEL
12345678910

Numer Telefonu
1111111111

Komunikat
Klient został dodany do bazy danych

Dodaj

Pobierz informacje

Modyfikuj

Wyczyszc Pola

Powrót

Wybieranie klienta:

Po przejściu do panelu „Wycieczki” musimy najpierw wybrać klienta, dla którego będziemy wybierać wycieczki. Aby to zrobić wybieramy zakładkę „Znajdź klienta”. Wpisujemy jego numer PESEL i klikamy „Szukaj”. Jeśli taki klient istnieje, to zostanie on wybrany do przyszłych operacji oraz zostaniemy o tym powiadomieni oknem dialogowym.

Pesel Klienta:

Wyświetl wycieczki **Wystaw umowę** **Wybierz wycieczkę** **Znajdz Klienta**

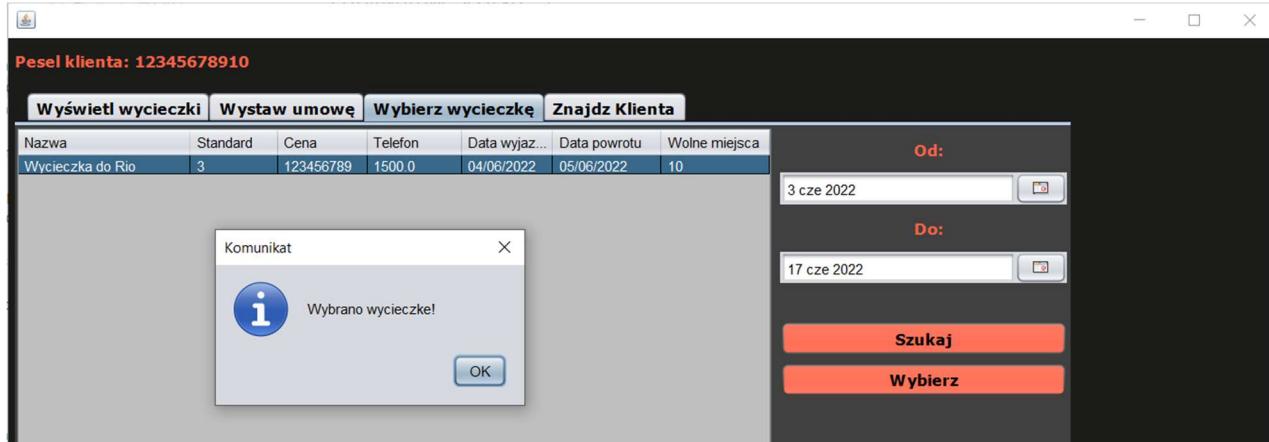
Pesel Klienta
12345678910

Szukaj Klienta

Komunikat
Znaleziono klienta o peselu: 12345678910

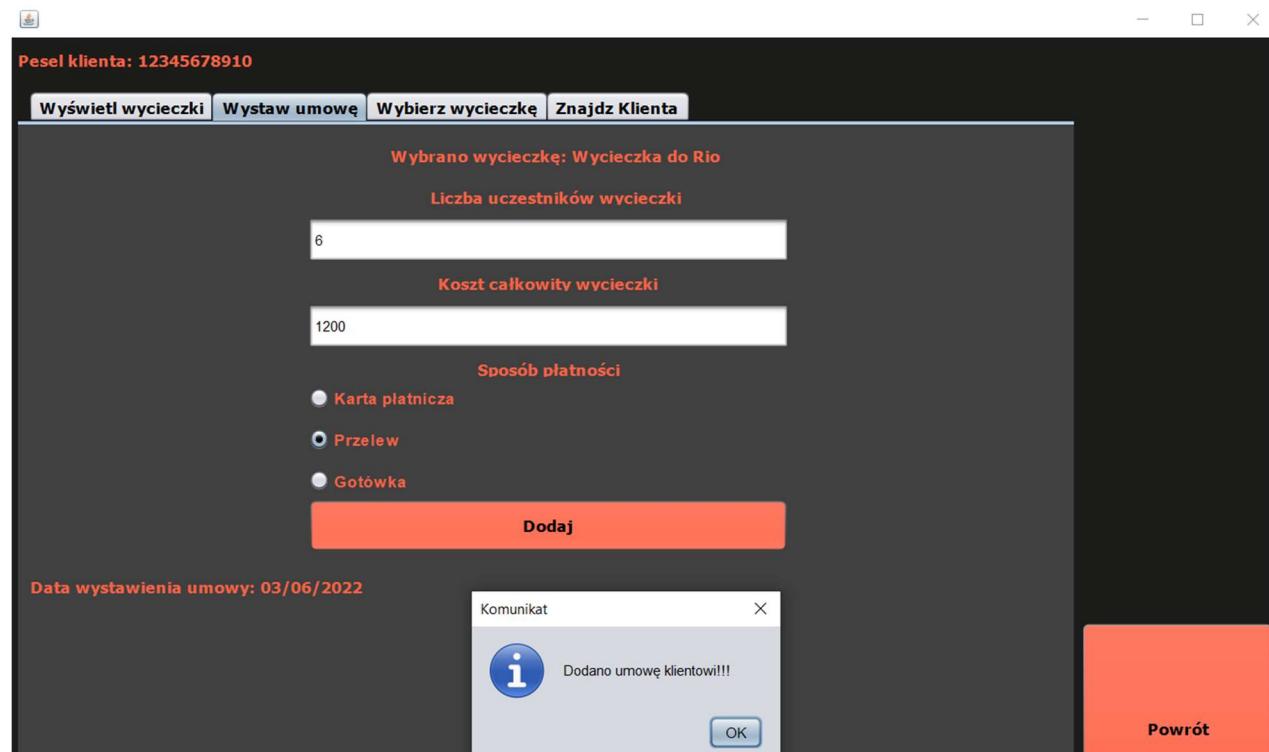
Wybieranie wycieczek:

Następnie możemy przystąpić do wybrania wycieczki dla wybranego przed chwilą klienta. W zakładce „Wybierz wycieczkę” klikamy na interesującą nas ofertę wycieczkową i klikamy przycisk „Wybierz”. Od teraz mamy zestaw danych (klient oraz wycieczka) gotowych do wystawienia umowy płatniczej. Musimy pamiętać o tym, że tabela nie wyświetli żadnych pozycji do czasu wyszukania ich według interesującej nas daty.



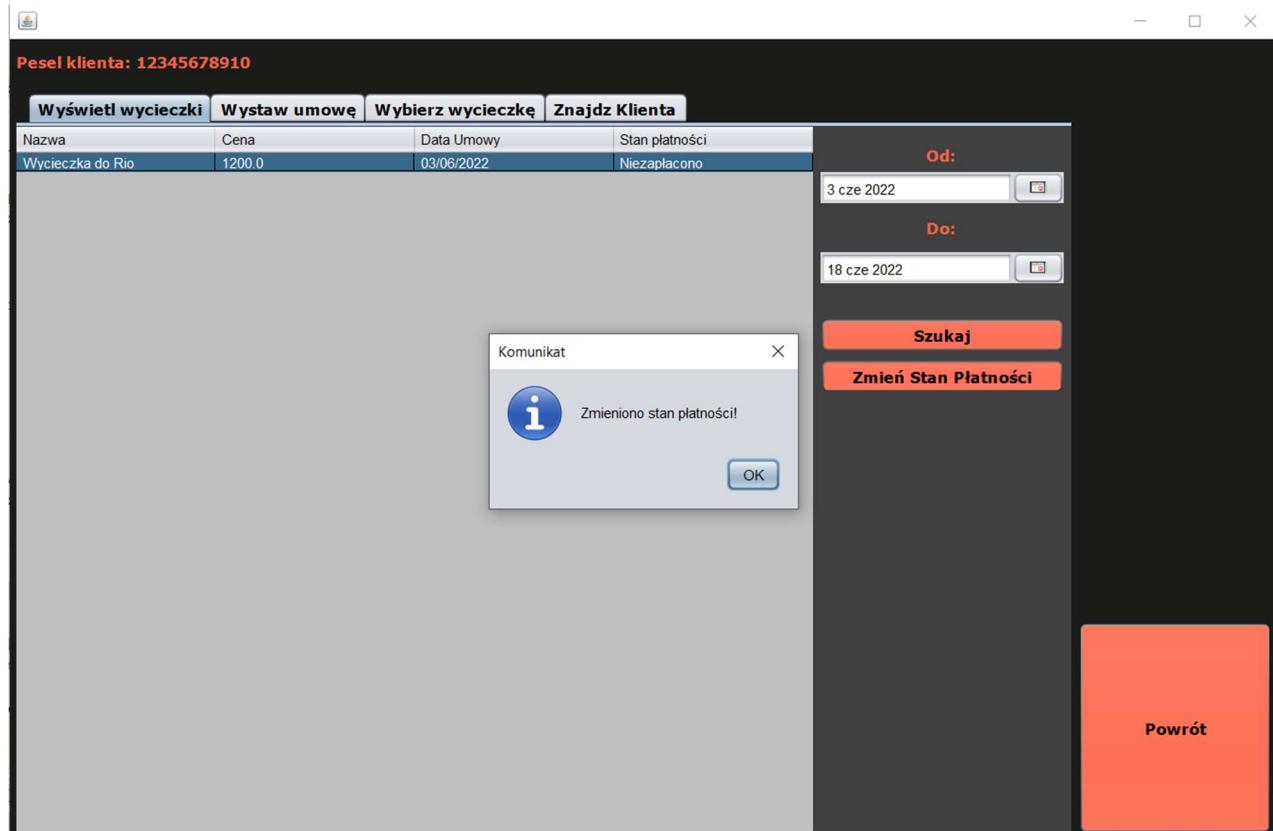
Wystawianie umów:

Kolejnym krokiem jest wybranie zakładki „Wystaw umowę”. Tutaj wypełniamy informacje na temat ilości osób, całkowitego kosztu wycieczki oraz rodzaju płatności. Liczba uczestników nie może być większa niż przewidziana w ofercie. Umowa zostanie wystawiona dla wybranego wcześniej według PESELu klienta. Po pomyślnym zakończeniu operacji, jeśli klient nie zarezerwował wszystkich dostępnych miejsc dla danej oferty, to będzie ona dalej widoczna w zakładce „Wybierz wycieczkę”. Wtedy liczba dostępnych miejsc zostanie pomniejszona o zarezerwowaną przez tego klienta.



Wyświetlanie wycieczek:

W ostatniej zakładce możemy wyświetlić wszystkie umowy zawarte przez wybranego klienta. W przypadku płatności przelewem, stan płatności będzie pokazywał wartość „Niezapłacono”. Jeśli klient ureguluje należność, możemy zmienić ten stan na „Zapłacono”. Zmiany stanu płatności nie są możliwe w przypadku płatności gotówką lub kartą. Podobnie jak wcześniej, aby widzieć jakiekolwiek wycieczki musimy je najpierw wyszukać według daty.



7. Opis podziału pracy

Michał Kaczor zajmował się głównie tworzeniem aplikacji od strony graficznej oraz tworzeniem interfejsu menu programu. Dbał o zabezpieczenie wszelkich niepożądanych działań związanych z korzystaniem z biura podróży. Zapewniał poprawianie większości kodu oraz jego opis działania. Przygotował dokumentację całego programu, która w prosty i przyjazny sposób wyjaśni jego działanie.

Przemysław Kałuziński przygotował klasy potrzebne do poruszania się po programie oraz zapewnił gromadzenie danych w bazie danych. Włożył również swój wkład w wykonanie graficzne aplikacji. Zapewnił poprawne przeprowadzanie wszelkich operacji. Wykonał sprawozdanie ze stworzonej aplikacji.

Grzegorz Kalarus zajmował się łączeniem serwera z klientem. Dbał o bezpieczeństwo wymiany danych. Zapewnił połączenie aplikacji z bazą danych co ułatwiło gromadzenie danych oraz poruszanie się po nich w bezpieczny sposób.

8. Validacja danych

W celu zapewnienia poprawności wszystkich danych zastosowaliśmy podwójną validację. Pierwszym etapem jest ograniczenie wpisywania poszczególnych znaków lub wartości w konkretne pola. Osiągnęliśmy to przy pomocy ActionListenerów oraz funkcji oferowanych przez biblioteki języka Java. Co więcej pozwoliły nam one na wyznaczenie maksymalnej ilości wpisanych przez użytkownika znaków, tak aby dane były w miarę realistyczne (tj. liczba uczestników wycieczki nie wynosiła 10 milionów). Poniżej przykład takiego zabezpieczenia.

```
/*
 * Funkcja uniemożliwiająca wpisanie liter i znaków w pole peselu wybieranego klienta.
 * Dozwolone są tylko cyfry. Jeśli wpisany znak nie jest cyfrą to "zjada go".
 */
clientpeseltrip.addKeyListener(new KeyAdapter() {
    @Override
    public void keyTyped(KeyEvent e) {
        super.keyTyped(e);
        char c = e.getKeyChar();

        if(clientpeseltrip.getText().length()+1 > 11)
            e.consume();
        else if(!Character.isDigit(c))
            e.consume();
    }
});

/*
 * Funkcja uniemożliwiająca wpisanie cyfr w pole imienia dodawanego klienta.
 * Dozwolone są tylko litery. Jeśli wpisany znak nie jest literą to "zjada go".
 */
clientname.addKeyListener(new KeyAdapter() {
    @Override
    public void keyTyped(KeyEvent e) {
        super.keyTyped(e);
        char c = e.getKeyChar();

        if(Character.isDigit(c) || !Character.isAlphabetic(c))
            e.consume();
    }
});
```

Następnym etapem validacji są bloki try – catch. Za ich pomocą wychwytyujemy wyjątki i błędy, jeśli jakieś nieodpowiednie wartości prześlizną się przez pierwszą warstwę zabezpieczeń. Przykładem może być np. blok obsługujący wyjątek próby konwersji tekstu na liczbę.

```
if(absta.isSelected())
{
    try {
        standard = Integer.parseInt(absta.getText());
    }
    catch(NumberFormatException exc)
    {
        exc.printStackTrace();
    }

    break;
}
```

W ten sposób jesteśmy pewni, że obiekty klas posiadają sprawdzone i poprawne zestawy danych, które są gotowe do przeprowadzenia na nich żądanych operacji. Serwer otrzymuje taki obiekt i nie musi już przeprowadzać jego ponownej validacji, gdyż ma pewność, że spełniają one wszystkie warunki.

9. Testowanie aplikacji

Naszym celem było zabezpieczenie aplikacji w możliwie najlepszy sposób oraz przygotowanie jej do radzenia sobie z każdą sytuacją wyjątkową jaką udało nam się wymyślić. W tym celu musieliśmy przeprowadzić niezliczone ilości testów. Po każdej zmianie dodanej w programie uruchamialiśmy go wielokrotnie, aby sprawdzić jego poprawność. Zazwyczaj pracowaliśmy razem przez co mieliśmy aktualny wgląd do wszelkich poprawek. Gdy jednak dodawaliśmy coś sami, to przy najbliższej okazji prezentowaliśmy tą część członkowi zespołu. Często, gdy już udało nam się uporać z jakimś problemem, to staraliśmy się razem przetestować odporność na błędy. Na koniec, gdy już dopięliśmy ostatni guzik, postanowiliśmy przekazać funkcjonalny projekt w ręce naszych kolegów. Ich zadaniem było starać się „zepsuć program”, czyli robić wszystko to, czego się nie powinno, np. wpisywać w pole peselu litery i znaki czy ustawiać taki sam login jak inny użytkownik. Celem tego zabiegu była zmiana podejścia osoby testującej i wykorzystanie umysłu, który nie miał wcześniej do czynienia z kodem programu. Pomogło nam to znaleźć kilka luk w kodzie i przygotować odpowiednie metody służące do ich „załatwania”. Ostatecznie uważamy, że aplikacja jest przygotowana na dość duże nadużycia i potrafi sobie z nimi poradzić. Oczywiście zdajemy sobie sprawę, że pewnie znajdą się jeszcze sytuacje, których nie rozważyliśmy, ale będzie to raczej znikomy procent wszystkich sytuacji wyjątkowych.

10. Wnioski

Na wstępie chcielibyśmy podziękować za wytrwałość w przebrnięciu przez wszystkie strony sprawozdania. Po szczegółowe informacje na temat kodu projektu zapraszamy do przejrzenia dokumentacji wygenerowanej przez program Doxywizard. Na chwilę obecną uznajemy, że projekt jest gotowy do oddania oraz zaprezentowania. Zrealizowaliśmy wszystkie wymagane założenia projektowe oraz wprowadziliśmy wiele swoich pomysłów. Wykorzystaliśmy w pełni możliwości biblioteki graficznej Swing, implementując w projekcie różnorodne elementy. Część z nich wymagała od nas ręcznej modyfikacji, gdyż GUI designer, oferowany przez środowisko Eclipse.