

# Dokumentacja

## Projekt Bazy Danych 1: „Wypożyczalnia samochodów”

Marcin Urbanowicz

### 1. Projekt koncepcji, założenia

#### 1.1. Temat projektu:

Celem projektu było stworzenie bazy danych oraz graficznego interfejsu do jej obsługi. Tematem była wypożyczalnia samochodów. Aplikacja ma za zadanie pozwolić na dostęp i dokonywanie operacji na bazie danych.

#### 1.2. Analiza wymagań użytkownika:

Aplikacja pozwala na łatwy dostęp do bazy danych. Można dzięki niej sprawdzić stan poszczególnych tabel, dodać nowe rekordy, a także zobaczyć parę przygotowanych przez autora raportów dotyczących pewnych dziedzin.

#### 1.3. Zaprojektowane funkcje

Baza danych umożliwia dodawanie danych do tabel i sprawdzanie ich zawartości za pomocą interfejsu graficznego napisanego w języku Java. Wszystkie funkcje dostępu do bazy danych obsługuje biblioteka JDBC, która wykorzystuje polecenia w języku SQL.

### 2. Projekt diagramów

#### 2.1. Zdefiniowanie encji oraz ich atrybutów

Wszystkie tabele zostały wycięte z diagramu ERD w SQL Power Architect

- **Dział**

Dział
id_działu: INTEGER NOT NULL [ PK ]
nazwa_działu: VARCHAR(60) NOT NULL
ilość_etatow: INTEGER NOT NULL

Obrazek1: Tabela Dział

- **Pracownik:**

Pracownik
id_pracownik: INTEGER NOT NULL [ PK ]
id_dzialu: INTEGER NOT NULL [ FK ]
imie: VARCHAR(60) NOT NULL
nazwisko: VARCHAR(60) NOT NULL
staz: INTEGER NOT NULL
wynagrodzenie: INTEGER NOT NULL
stanowisko: VARCHAR(40) NOT NULL

Obrazek 2: Tabela Pracownik

- **Klient**

Klient
id_klienta: INTEGER NOT NULL [ PK ]
telefon: VARCHAR(15) NOT NULL
imie: VARCHAR(60) NOT NULL
nazwisko: VARCHAR(60) NOT NULL

Obrazek 3: Tabela Klient

- **Ubezpieczyciel**

Ubezpieczyciel
id_ubezpieczyciela: INTEGER NOT NULL [ PK ]
nazwa_ubezpieczyciela: VARCHAR(40) NOT NULL
telefon: VARCHAR(15) NOT NULL
adres: VARCHAR(60) NOT NULL

Obrazek 4: Tabela Ubezpieczyciel

- **Ubezpieczenia\_samochodu**

Ubezpieczenia_samochodu
id_ubezpieczenia: INTEGER NOT NULL [ PK ]
rodzaj_ubezpieczenia: VARCHAR(60) NOT NULL
koszt: INTEGER NOT NULL

Obrazek 5: Tabela Ubezpieczenia samochodu

- Wypożyczenie

Wypożyczenie	
id_wypożyczenia:	INTEGER NOT NULL [ PK ]
id_klienta:	INTEGER NOT NULL [ FK ]
id_pracownik:	INTEGER NOT NULL [ FK ]
data_wypożyczenia:	DATE NOT NULL
data_zwrotu:	DATE

Obrazek 6: Tabela Wypożyczenie

- Samochod

Samochod	
id_samochodu:	INTEGER NOT NULL [ PK ]
id_wypożyczenia:	INTEGER NOT NULL [ FK ]
marka:	VARCHAR(30) NOT NULL
model_samochodu:	VARCHAR(60) NOT NULL
rocznik:	INTEGER NOT NULL
silnik:	VARCHAR(60) NOT NULL
wartosc:	INTEGER NOT NULL

Obrazek 7: Tabela Samochod

- Ubezpieczyciel\_Asoc

Ubezpieczyciel_Asoc	
id_ubezpieczyciela:	INTEGER NOT NULL [ FK ]
id_samochodu:	INTEGER NOT NULL [ FK ]

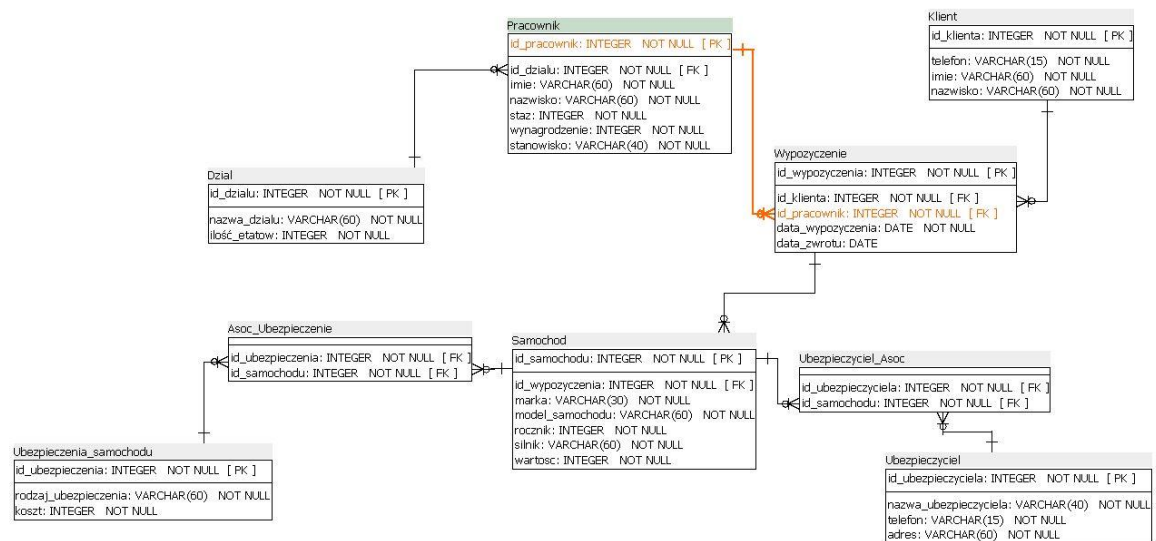
Obrazek 8: Tabela Ubezpieczyciel\_Asoc

- Asoc\_ubezpieczenie

Asoc_Ubezpieczenie	
id_ubezpieczenia:	INTEGER NOT NULL [ FK ]
id_samochodu:	INTEGER NOT NULL [ FK ]

Obrazek 9: Tabela Asoc\_Ubezpieczenie

## 2.2. Relacje między encjami



Obrazek 10: Diagram ERD bazy danych

## 3. Projekt logiczny

### 3.1. Tabele, klucze obce i indeksy

- **Dział**

Tabela zawiera informacje o działach w firmie oraz ilościach etatów jakimi dany dział dysponuje. Kluczem głównym jest `id_dzialu`

- **Pracownik:**

Tabela zawiera informacje o pracownikach firmy. Kluczem głównym jest `id_pracownika` natomiast klucz obcy `id_dzialu` łączy tą tabelę z tabelą **Dział**. Kilku pracowników może pracować w tym samym dziale. Kolumny zawierające imię i nazwisko mogą zawierać wyłącznie litery co jest walidowane na poziomie wprowadzania. Na tym poziomie także wprowadzane dane są poprawiane tak aby zaczynały się od wielkich liter.

- **Klient**

Tabela klient zawiera informację o klientach firmy. Podobnie jak przy tabeli **Pracownik** `imie` oraz `nazwisko` są walidowane przy wprowadzaniu tak aby zawierały tylko litery, a także od razu poprawiane tak, aby zaczynać się od wielkiej litery. Kolumna `telefon` może zawierać tylko numery w formacie: 123-234-567 co jest walidowane przy wprowadzaniu. Kluczem głównym jest `id_klienta`.

- **Ubezpieczyciel**

Tabela przechowuje informacje o firmach ubezpieczeniowych. Kolumna *telefon* jest identycznie jak w tabeli **Klient** walidowany i przyjmowane są tylko numery w formacie 123-456-767. Kluczem głównym jest *id\_ubezpieczyciela*.

- **Ubezpieczenia\_samochodu**

Tabela zawiera rodzaje ubezpieczenia oraz jego koszty. Kluczem głównym jest *id\_ubezpieczenia*.

- **Wypożyczenie**

Tabela przechowuje informacje o wypożyczeniach. Kolumna *data\_wypożyczenia* zawiera datę rozpoczęcia, a *data\_zwrotu* może być polem typu null ze względu na to, że samochód może być dalej wypożyczony (klient jeszcze nie oddał samochodu). Klucz obcy *id\_klienta* łączy nas z tabelą **Klient** i wskazuje, z którym klientem związane jest dane wypożyczenie. Kolumna *id\_pracownik* łączy się natomiast z tabelą **Pracownik** i jest to informacje o tym, który pracownik przeprowadził dane wypożyczenie. Kluczem głównym jest *id\_wypożyczenia*.

- **Samochod**

Tabela przechowuje informacje o samochodach w wypożyczalni. Klucz obcy *id\_wypożyczenia* łączy ją z tabelą **Wypożyczenia**. Kluczem głównym jest *id\_samochodu*.

- **Ubezpieczyciel\_Asoc**

Jest to tabela asocjacyjna, która łączy ze sobą tabelę **Samochod** i **Ubezpieczyciel** w celu stworzenia relacji wiele do wielu (jeden samochód może mieć różnych ubezpieczycieli, jak również wiele samochodów może mieć tego samego ubezpieczyciela).

- **Asoc\_ubezpieczenie**

Jest to tabela asocjacyjna, która łączy ze sobą tabelę **Samochod** i **Ubezpieczenie\_samochodu** w celu stworzenia relacji wiele do wielu. (Jeden samochód może mieć kilka rodzajów ubezpieczenia, a także wiele samochodów może mieć takie same ubezpieczenia ).

### 3.2. Operacje na danych

W aplikacji przygotowane są trzy rodzaje operacji:

- I. Sprawdzanie stanu tabel
- II. Wprowadzanie do bazy danych
- III. Raporty

Obsługiwane są one następującymi kwerendami.

## Ad I

- Tabela Klient

**SELECT** id\_klienta, imie, nazwisko, telefon **FROM** Klient;

- Tabela Samochody

**SELECT** id\_samochodu, marka, model\_samochodu, rocznik, silnik, wartość **FROM** samochod;

- Tabela Pracownicy

**SELECT** id\_pracownika, imie, nazwisko, staz, wynagrodzenie, stanowisko, nazwa\_dzialu **FROM** Dzial **JOIN** Pracownik **USING**(id\_dzialu);

- Tabela Dzialy

**SELECT** id\_dzialu, nazwa\_dzialu, ilość\_etatow **FROM** Dzial;

- Tabela Ubezpieczenia\_samochodu

**SELECT** id\_ubezpieczenia, rodzaj\_ubezpieczenia, koszt **FROM** Ubezpieczenia\_samochodu

- Tabela Ubezpieczyciel

**SELECT** id\_ubezpieczyciela, nazwa\_ubezpieczyciela, telefon, adres **FROM** Ubezpieczyciel;

- Tabela Wypozyczenia

**SELECT** id\_wypozyczenia, marka, model\_samochodu, imie, nazwa, data\_wypozyczenia, data\_zwrotu **FROM** Wypozyczenie **JOIN** Klient **USING**(id\_klienta) **JOIN** Samochod **USING**(id\_wypozyczenia);

## Ad II

- Tabela Klient

**INSERT INTO** Klient (id\_klienta, telefon, imie, nazwisko) **values** (...)

- Tabela Samochod

**INSERT INTO** Samochod (id\_samochodu, id\_wypozyczenia, marka, model\_samochodu, rocznik, silnik, wartosc) **values** (...)

- Tabela Pracownicy

**INSERT INTO** Pracownik (id\_pracownika, id\_dzialu, imie, nazwisko, staz, wynagrodzenie, stanowisko) **values** (...)

- Tabela Dzialy

**INSERT INTO** Dzial (id\_dzialu, nazwa\_dzialu, ilość\_etatow) **values** (...)

- Tabela Ubezpieczenia\_samochodu

**INSERT INTO** ubezpieczenia\_samochodu (id\_ubezpieczenia, rodzaj\_ubezpieczenia, koszt) **values** (...);

- Tabela Ubezpieczyciel

**INSERT INTO** ubezpieczyciel (id\_ubezpieczyciela, nazwa\_ubezpieczyciela, telefon, adres) **values** (...)

- Tabela Wypozyczenia

**INSERT INTO** Wypozyczenie (id\_wypozyczenia, id\_klienta, id\_pracownika, data\_wypozyczenia, data\_zwrotu ) **values** (...)

Oraz dodatkowo trzeba zmienić id\_wypozyczenia w tabeli Samochod dla wypożyczanych samochodów.

**UPDATE** samochod **SET** id\_wypozyczenia="..." **WHERE** id\_samochodu="..."

### Ad III

Przy tworzeniu raportów posłużyłem się widokami, które potem były po prostu wywoływane w aplikacji za pomocą polecenia:

**SELECT \* FROM** „nazwa\_widoku”

- **Raport: Sumaryczny koszt ubezpieczeń dla poszczególnych samochodów**

**CREATE VIEW** koszt\_ubezpieczeń **AS**

**SELECT** id\_samochodu, marka, model\_samochodu, **SUM**(koszt) **FROM** Samochod  
**JOIN** Asco\_ubezpieczemoe **USING**(id\_samochodu)

**JOIN** Ubezpieczenie\_samochodu **USING**(id\_ubezpieczenie)

**GROUP BY** id\_samochodu, marka, model\_samochodu **ORDER BY** id\_samochodu

- **Raport: Wypożyczone i jeszcze nie oddane samochody oraz kto je wypożyczył**

**CREATE VIEW** niedostepne **AS**

**SELECT** marka, model\_samochodu, imie, nazwisko, telefon **FROM** Samochod

**JOIN** Wypozyczenie **USING**(id\_wypozyczenia)

**JOIN** Klient **USING**(id\_klienta)

**WHERE** data\_zwrotu **IS NULL**

- **Raport: Dostępne samochody oraz kto je ostatnio je wypożyczał**

**CREATE VIEW** dostepne **AS**

**SELECT** marka, model\_samochodu, imie, nazwisko, telefon **FROM** Samochod

**JOIN** Wypozyczenie **USING**(id\_wypozyczenia)

**JOIN** Klient **USING**(id\_klienta)

WHERE data\_zwrotu IS NOT NULL

- **Raport: Ilość przeprowadzonych wypożyczeń przez danych pracowników posortowana malejąco**

CREATE VIEW wyniki AS

SELECT imie, nazwisko, COUNT(Wypozyczenie.id\_pracownika) AS ilość\_umow FROM Pracownik

JOIN Wypozyczenie USING(id\_pracownika) GROUP BY imie, nazwisko ORDER BY ilosc\_umow DESC;

## 4. Projekt Funkcjonalny

### 4.1. Panel Sterowania Aplikacji

Po włączeniu aplikacji otrzymujemy następujący interfejs:



Obrazek 11: Interfejs główny

Stamtąd możemy się dostać, poprzez naciśnięcie na przycisk w 3 miejsca:

#### 1) Sprawdzanie stanu bazy danych

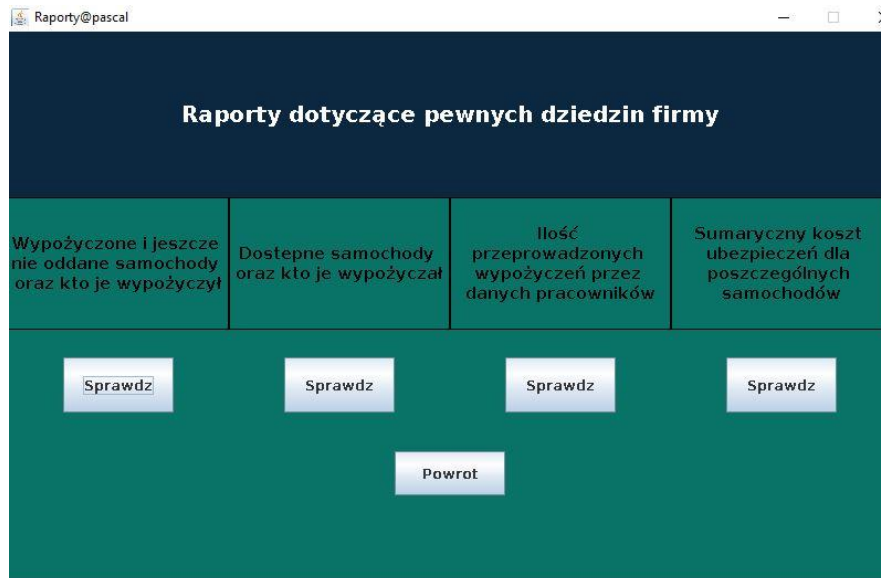




Obrazek 12: Interfejs do sprawdzania stanu bazy danych

Po naciśnięciu na dowolny przycisk **Sprawdz** uzyskamy zawartości poszczególnych tabel.

## 2) Raporty specjalne



Obrazek 13: Interfejs do sprawdzania raportów

Po naciśnięciu na przycisk **Sprawdz** uzyskamy odpowiedni raport

## 3) Dodawanie nowe dane do bazy danych



Obrazek 14: Interfejs do dodawania do tabel nowych rekordów

W niemal wszystkich przypadkach formularze wyglądają podobnie do tego poniższego:

The screenshot shows a web browser window with the title 'Formlarz@pascal'. The main heading is 'Nowy Klient'. Below the heading, there are three input fields labeled 'Telefon', 'Imie', and 'Nazwisko'. At the bottom, there are two buttons: 'Dodaj' (Add) and 'Powrot' (Return).

Obrazek 15: Formularz wprowadzania danych do tabeli Klient

Wyjątek stanowi formularz dodający nowe wypożyczenie, na którym najpierw trzeba wybrać samochody do wypożyczenia (wyświetlane są tylko te dostępne).

The screenshot shows a web browser window with the title 'Formlarz@pascal'. The main heading is 'Jakie samochody sa wypożyczane'. Below the heading, there is a table with the following columns: 'Wypożyczony', 'Marka', 'Model', 'silnik', 'rocznik', and 'wartosc'. The table contains 8 rows of data. At the bottom, there are two buttons: 'Dalej' (Next) and 'Powrot' (Return).

Wypożyczony	Marka	Model	silnik	rocznik	wartosc
<input type="checkbox"/> 3	Audi	A6	2.0	2021	300000
<input type="checkbox"/> 5	Peugeot	208	1.4	2017	73000
<input type="checkbox"/> 6	Renault	Megane	1.6	2019	75000
<input type="checkbox"/> 7	Dacia	Sandero	1.0	2014	44000
<input type="checkbox"/> 9	Citroen	Berlingo Multisp...	1.6	2015	54000
<input type="checkbox"/> 12	Hyundai	I40	1.7	2017	80000
<input type="checkbox"/> 13	Nissan	Qashqai	1.8	2016	85000
<input type="checkbox"/> 14	Mitsubishi	Lancer	2.0	2011	70000

Obrazek 16: Formularz wybierania wypożyczanych samochodów

Kiedy wybierzemy samochody przechodzimy do kolejnego okna, na którym musimy zaznaczyć, który pracownik przeprowadził transakcję oraz klienta który wypożyczył te samochody. Jeżeli klient, który wypożycza samochody jest nowy jest tam też opcja, która pozwala go dodać, a potem można go spokojnie wybrać z listy.

Pracownik			Klient		
Id	Imie	Nazwisko	Id	Imie	Nazwisko
<input type="radio"/> 1	Michal	Wojtasik	<input type="radio"/> 1	Piotr	Tusin
<input type="radio"/> 2	Mateusz	Gotowka	<input type="radio"/> 2	Marek	Urban
<input type="radio"/> 3	Anna	Zaradna	<input type="radio"/> 3	Tomasz	Problem
<input type="radio"/> 4	Adam	Obecny	<input type="radio"/> 4	Roman	Zupa
<input type="radio"/> 5	Patrycja	Hurkacz	<input type="radio"/> 5	Kordian	Gwiazdowski
<input type="radio"/> 6	Grzegorz	Brzezyszczykie...	<input type="radio"/> 6	Zbigniew	Bialek
			<input type="radio"/> 7	Andrzej	Biedny
			<input type="radio"/> 8	Aleksander	Pila
			<input type="radio"/> 9	Tymoteusz	Wrzutka
			<input type="radio"/> 10	Michal	Imigrant
			<input type="radio"/> 11	Jeremiasz	Smietana
			<input type="radio"/> 12	Antek	Szwagier
			<input type="radio"/> 13	Marzanna	Topielec
			<input type="radio"/> 14	Michalina	Mickiewicz-slowa...
			<input type="radio"/> 15	Karol	Majster

Gotowe   Nowy Klient   Powrot

Obrazek 17: Formularz wybierania związanych z transakcją pracownika i klienta.

Po każdej operacji dodawania do bazy danych wyskakuje nam okienko z komunikatem o powodzeniu lub niepowodzeniu operacji.

Operacja sie powiodla

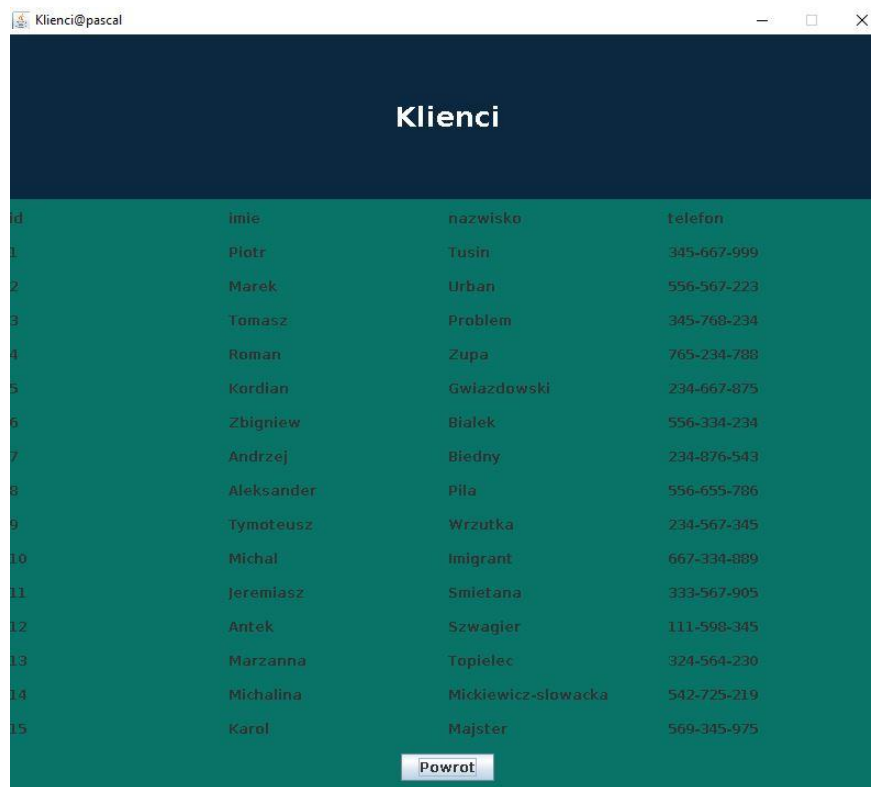
Powrot

Obrazek 18: Komunikat o powodzeniu operacji

Wszystkie przyciski **Powrot** pozwalają na powrót – zazwyczaj do poprzedniego miejsca, w którym się znajdowaliśmy.

## 4.2. Wizualizacja danych

Wybierając dowolną tabelę lub raport z interfejsów opisanych powyżej zawsze uzyskamy rezultat wyglądający podobnie jak ten, który uzyskałem sprawdzając tabelę **Klient**:



id	imie	nazwisko	telefon
1	Piotr	Tusin	345-667-999
2	Marek	Urban	556-567-223
3	Tomasz	Problem	345-768-234
4	Roman	Zupa	765-234-788
5	Kordian	Gwiazdowski	234-667-875
6	Zbigniew	Bialek	556-334-234
7	Andrzej	Biedny	234-876-543
8	Aleksander	Piła	556-655-786
9	Tymoteusz	Wrzutka	234-567-345
10	Michał	Imigrant	667-334-889
11	Jeremiasz	Smietana	333-567-905
12	Antek	Szwagier	111-598-345
13	Marzanna	Topielec	324-564-230
14	Michalina	Mickiewicz-słowacka	542-725-219
15	Karol	Majster	569-345-975

Obrazek 19: Wynik sprawdzenia za wartości tabeli Klient

## 5. Dokumentacja

### 5.1. Wprowadzanie danych

Tworzenie bazy danych jest ręczne i w tym celu dostarczam 4 pliki .sql

- 1) Create\_Database.sql
- 2) Triggery.sql
- 3) Insert\_Data.sql
- 4) Widoki.sql

Tworząc bazę danych najlepiej wywoływać je w kolejności jakiej zostały podane powyżej. Oprócz tego dane do bazy danych wprowadzane są przez formularze, które opisywałem w rozdziale 4.1

### 5.2. Instrukcja obsługi

Aplikacja była w całości postawiona i testowana na serwerze wydziałowym pascal.fis.agh.edu.pl i tam też trzeba ją uruchamiać. Ze względu też na serwer Pascal wszystkie nazwy są pozbawione polskich znaków i w celu poprawnego przebiegu programu zalecane jest ich unikać.

Aplikacja jest w całości napisana w języku Java z pomocą bibliotek Swing i JDBC, więc w celu uruchomienia programu trzeba go najpierw skompilować. W toku testów aplikacji wyszło, że aby mieć absolutną pewność, że wszystko działa trzeba było podać wszystkie pliki, więc poniżej zostawiam komendę do kompilacji (wszystkie pliki są w jednym katalogu)

```
javac Main.java GUI.java ReportFrame.java ReportSQL.java CheckFrame.java  
CheckSQL.java AddFrame.java EnsuranceForm.java EnsurerForm.java CarForm.java  
ClientForm.java DepartmentForm.java WorkerForm.java RentForm.java  
RentClientWorker.java RentFinalise.java AddResultFrame.java
```

Następnie trzeba uruchomić skompilowany program wraz ze sterownikiem dla postgresQL, który dostarczam wraz z plikami źródłowymi. Odbywa się to poleceniem

```
java -cp ../postgresql-42.3.1.jar Main
```

Przy czym nie musi to być koniecznie ta wersja sterownika.

Wszystkie połączenia do baz danych odbywają się do mojej prywatnej udostępnionej mi przez wydział bazy danych **u9urbanowicz** na stworzonej SCHEMA'ie **projekt** i aby nie komplikować więcej niż to konieczne wszelkie testy mogą być wykonywane w oparciu o tamtą bazę danych, która jest przygotowana w oparciu o dostarczone przeze mnie pliki \*.sql.

Zaznaczam też tutaj drobną sprawę która wymaga komentarza, chodzi o dodawanie przez formularz do tabeli **Pracownik**. Jest tam dostępne pole tekstowe, w którym napisać trzeba dział pracownika i aby operacja się udała, to musi to być prawdziwa nazwa, która istnieje w bazie danych. W celu umożliwienia sprawdzania działania tego formularza dostarczam tutaj poprawne nazwy działów:

- Obsługi klienta
- Sprzatajacy
- Marketing
- Analiza danych
- Kontrola jakosci

Oczywiście nowe działy, które można dodawać innym formularzem będą tu obsługiwane, jednak trzeba pamiętać, że istnieje Trigger, który poprawia nazwę działu tak aby zaczynała się od wielkiej litery.

### 5.3. Dokumentacja techniczna

W kodzie źródłowym znajduje się 16 klas, z których jednak spora część jest bardzo podobna (np. formularze wprowadzania), więc opiszę przede wszystkim te wzorcowe, na podstawie których możemy dowiedzieć się najwięcej.

- **Klasa ReportFrame**

Odpowiada za interfejs raportów (obrazek 13)

#### 1) Biblioteki

```
import java.awt.Color;  
import java.awt.Font;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.awt.BorderLayout;  
  
import javax.swing.*;
```

## 2) Deklaracja klasy

```
public class ReportFrame extends JFrame implements ActionListener
```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności.

## 3) Pola

```
// Panel
JPanel header;
JPanel content;
// Przyciski
private JButton button1;
private JButton button2;
private JButton button3;
private JButton button4;
private JButton buttonReturn;
// Etykiety z tekstem
private JLabel headerLabel;
private JLabel label1;
private JLabel label2;
private JLabel label3;
private JLabel label4;
```

Są to przede wszystkim elementy pochodzące z biblioteki Swing, które muszą być zdefiniowane jako pola klasy, gdyż są używane w innych metodach poza konstruktorem. Przyciski z numerami odpowiadają ramkom z numerami

## 4) Konstruktor

```
public ReportFrame()
```

W konstruktorze ustawiane są funkcjonalności przycisków i etykiet oraz paneli

## 5) Metody

a. **public void actionPerformed(ActionEvent e)**

```

@Override
public void actionPerformed(ActionEvent e) {
    // Wypozyczone
    if (e.getSource() == button1) {
        this.dispose();
        new ReportSQL("Wypozyczone");
    } // Dostepne
    else if (e.getSource() == button2) {
        this.dispose();
        new ReportSQL("Dostepne");
    } // Wyniki pracownikow
    else if (e.getSource() == button3) {
        this.dispose();
        new ReportSQL("Wyniki Pracownikow");
    } else if (e.getSource() == button4) {
        this.dispose();
        new ReportSQL("Koszty ubezpieczen");
    } else if (e.getSource() == buttonReturn) {
        this.dispose();
        new GUI();
    }
}

```

Metoda pochodzi z interfejsu **ActionListener** i zajmuje się obsługą naciskania przycisków. Przycisk `buttonReturn` pozwala na powrót do interfejsu głównego, natomiast pozostałe wywołują klasę **ReportSQL** z odpowiednimi parametrami.

Klasa **ReportFrame** działa podobnie do klas takich jak `CheckFrame.java`, `AddFrame.java`, `GUI.java`. Różnice polegają zazwyczaj na nazwach i ilości pól.

- **Klasa ReportSQL.java**

Klasa ta przetwarza dane dostarczone do niej przez klasę **ReportFrame**

## 1) Biblioteki

```

import java.sql.*;

import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.FlowLayout;

import javax.swing.*;

```

## 2) Deklaracja klasy

```

public class ReportSQL extends JFrame implements ActionListener

```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności



### 3) Pola

```
// Panele
private JPanel header;
private JPanel content;
private JPanel footer;
// Etykiety z tekstem
private JLabel headerLabel;

// Przyciski
private JButton returnButton;
```

Są to przede wszystkim elementy pochodzące z biblioteki Swing, które muszą być zdefiniowane jako pola klasy, gdyż są używane w innych metodach poza konstruktorem. Przyciski z numerami odpowiadają ramkom z numerami

### 4) Konstruktor

```
public ReportSQL(String data)
```

W konstruktorze ustawiane są funkcjonalności przycisków i etykiet oraz paneli, a także jest nawiązywane połączenie z bazą danych, następnie wywoływana jest odpowiednia metoda w zależności od wartości zmiennej *data*

```
if (c != null) {
    System.out.println("Połączenie z baza danych OK ! ");
    try {
        // Wypozyczone
        if (data == "Wypozyczone") {
            PreparedStatement pst1 = c.prepareStatement("SELECT * FROM niedostepne");
            PreparedStatement pst2 = c.prepareStatement("SELECT COUNT(*) AS ilosc FROM niedostepne");
            selectNiedostepne(pst1, pst2);
        }
        if (data == "Dostepne") {
            PreparedStatement pst1 = c.prepareStatement("SELECT * FROM dostepne");
            PreparedStatement pst2 = c.prepareStatement("SELECT COUNT(*) AS ilosc FROM dostepne");
            selectDostepne(pst1, pst2);
        }
        if (data == "Wyniki Pracownikow") {
            PreparedStatement pst1 = c.prepareStatement("SELECT * FROM wyniki_pracownikow");
            PreparedStatement pst2 = c.prepareStatement("SELECT COUNT(*) AS ilosc FROM wyniki_pracownikow");
            selectWyniki(pst1, pst2);
        }
        if (data == "Koszty ubezpieczen") {
            PreparedStatement pst1 = c.prepareStatement("SELECT * FROM koszt_ubezpieczen");
            PreparedStatement pst2 = c.prepareStatement("SELECT COUNT(*) AS ilosc FROM koszt_ubezpieczen");
            selectKoszt(pst1, pst2);
        }
    } catch (SQLException e) {
        System.out.println("Bład podczas przetwarzania danych:" + e);
    }
} else {
    System.out.println("Brak polaczenia z baza, dalsza czesc aplikacji nie jest wykonywana.");
}
```



## 5) Metody

### a. `public void actionPerformed(ActionEvent e)`

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == returnButton) {
        this.dispose();
        new ReportFrame();
    }
}
```

Metoda pochodzi z interfejsu **ActionListener** i zajmuje się obsługą naciskania przycisków. Przycisk `buttonReturn` pozwala na powrót do okna **ReportFrame**

### b. `private void selectNiedostępne(PreparedStatement pst1, PreparedStatement pst2)`

```
private void selectNiedostępne(PreparedStatement pst1, PreparedStatement pst2) {
    try {
        ResultSet rs1 = pst1.executeQuery();
        ResultSet rs2 = pst2.executeQuery();

        // Ilosc Wierszy
        rs2.next();
        int rows = rs2.getInt("ilosc");
        rows++;

        content.setLayout(new GridLayout(rows, 5));
        content.add(new JLabel("Marka"));
        content.add(new JLabel("Model"));
        content.add(new JLabel("Imie"));
        content.add(new JLabel("Nazwisko"));
        content.add(new JLabel("Telefon"));

        while (rs1.next()) {
            String marka = rs1.getString("marka");
            String model = rs1.getString("model_samochodu");
            String imie = rs1.getString("imie");
            String nazwisko = rs1.getString("nazwisko");
            String telefon = rs1.getString("telefon");
            content.add(new JLabel(marka));
            content.add(new JLabel(model));
            content.add(new JLabel(imie));
            content.add(new JLabel(nazwisko));
            content.add(new JLabel(telefon));
        }
        rs1.close();
        rs2.close();
        pst1.close();
        pst2.close();
    } catch (SQLException e) {
        System.out.println("Błąd podczas przetwarzania danych:" + e);
    }
}
```

Metoda wykonuje dwa przesłane do niej obiekty klasy **PreparedStatement** (Pierwszy to tabela z danymi, drugi to ilość wierszy). Następnie pobiera dane i umieszcza je na

oknie (Przykład: obrazek 19) i na koniec zamyka połączenie. Może wywołać wyjątek **SQLException**. Metoda ta jest związana z widokiem **niedostępne**.

c. **private void selectDostępne(PreparedStatement pst1, PreparedStatement pst2)**

```
private void selectDostępne(PreparedStatement pst1, PreparedStatement pst2) {
    try {
        ResultSet rs1 = pst1.executeQuery();
        ResultSet rs2 = pst2.executeQuery();

        // Ilość wierszy
        rs2.next();
        int rows = rs2.getInt("ilosc");
        rows++;

        content.setLayout(new GridLayout(rows, 5));
        content.add(new JLabel("Marka"));
        content.add(new JLabel("Model"));
        content.add(new JLabel("Imię"));
        content.add(new JLabel("Nazwisko"));
        content.add(new JLabel("Telefon"));

        while (rs1.next()) {
            String marka = rs1.getString("marka");
            String model = rs1.getString("model_samochodu");
            String imie = rs1.getString("imie");
            String nazwisko = rs1.getString("nazwisko");
            String telefon = rs1.getString("telefon");
            content.add(new JLabel(marka));
            content.add(new JLabel(model));
            content.add(new JLabel(imie));
            content.add(new JLabel(nazwisko));
            content.add(new JLabel(telefon));
        }
        rs1.close();
        rs2.close();
        pst1.close();
        pst2.close();
    } catch (SQLException e) {
        System.out.println("Błąd podczas przetwarzania danych:" + e);
    }
}
```

Metoda wykonuje dwa przesłane do niej obiekty klasy **PreparedStatement** (Pierwszy to tabela z danymi, drugi to ilość wierszy). Następnie pobiera dane i umieszcza je na oknie (Przykład: obrazek 19) i na koniec zamyka połączenie. Może wywołać wyjątek **SQLException**. Metoda ta jest związana z widokiem **dostępne**.

d. `private void selectWyniki(PreparedStatement pst1, PreparedStatement pst2)`

```
private void selectWyniki(PreparedStatement pst1, PreparedStatement pst2) {
    try {
        ResultSet rs1 = pst1.executeQuery();
        ResultSet rs2 = pst2.executeQuery();

        // Ilosc Wierszy
        rs2.next();
        int rows = rs2.getInt("ilosc");
        rows++;

        content.setLayout(new GridLayout(rows, 3));
        content.add(new JLabel("imie"));
        content.add(new JLabel("Nazwisko"));
        content.add(new JLabel("Ilosc zawartych umow"));

        while (rs1.next()) {
            String imie = rs1.getString("imie");
            String nazwisko = rs1.getString("nazwisko");
            String umowy = rs1.getString("ilosc_umow");
            content.add(new JLabel(imie));
            content.add(new JLabel(nazwisko));
            content.add(new JLabel(umowy));
        }
        rs1.close();
        rs2.close();
        pst1.close();
        pst2.close();
    } catch (SQLException e) {
        System.out.println("Bład podczas przetwarzania danych:" + e);
    }
}
```

Metoda wykonuje dwa przesłane do niej obiekty klasy **PreparedStatement** (Pierwszy to tabela z danymi, drugi to ilość wierszy). Następnie pobiera dane i umieszcza je na oknie (Przykład: obrazek 19) i na koniec zamyka połączenie. Może wywołać wyjątek **SQLException**. Metoda ta jest związana z widokiem **wyniki\_pracownikow**.

e. **private void selectWyniki(PreparedStatement pst1,  
PreparedStatement pst2)**

```
// Koszty ubezpieczen
private void selectKoszt(PreparedStatement pst1, PreparedStatement pst2) {
    try {

        ResultSet rs1 = pst1.executeQuery();
        ResultSet rs2 = pst2.executeQuery();

        // Ilosc Wierszy
        rs2.next();
        int rows = rs2.getInt("ilosc");
        rows++;

        content.setLayout(new GridLayout(rows, 4));
        content.add(new JLabel("Id"));
        content.add(new JLabel("Marka"));
        content.add(new JLabel("Model"));
        content.add(new JLabel("Koszt ubezpieczen"));
        while (rs1.next()) {
            int id = rs1.getInt("id_samochodu");
            String nazwa = rs1.getString("marka");
            String model = rs1.getString("model_samochodu");
            int suma = rs1.getInt("sum");
            content.add(new JLabel(Integer.valueOf(id).toString()));
            content.add(new JLabel(nazwa));
            content.add(new JLabel(model));
            content.add(new JLabel(Integer.valueOf(suma).toString()));
        }
        rs1.close();
        rs2.close();
        pst1.close();
        pst2.close();
    } catch (SQLException e) {
        System.out.println("Bład podczas przetwarzania danych:" + e);
    }
}
```

Metoda wykonuje dwa przesłane do niej obiekty klasy **PreparedStatement** (Pierwszy to tabela z danymi, drugi to ilość wierszy). Następnie pobiera dane i umieszcza je na oknie (Przykład: obrazek 19) i na koniec zamyka połączenie. Może wywołać wyjątek **SQLException**. Metoda ta jest związana z widokiem **koszt\_ubezpieczen**.

Klasą analogiczną do tej klasy jest klasa **CheckSQL**, która przetwarza dane, które dostała od klasy **CheckFrame**.

- **Klasa ClientForm**

Jest to formularz, służący do wprowadzania danych do tabeli Klient (obrazek 15)

### 1) Biblioteki

```
import java.sql.*;

import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.FlowLayout;

import javax.swing.*;
```

### 2) Deklaracja

```
public class ClientForm extends JFrame implements ActionListener
```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności.

### 3) Pola

```
// Panele
private JPanel header;
private JPanel content;
private JPanel footer;
// Etykiety z tekstem
private JLabel headerLabel;
private JLabel telefonLabel;
private JLabel imieLabel;
private JLabel nazwiskoLabel;
// Pola do wpisywania
private JTextField telefonField;
private JTextField imieField;
private JTextField nazwiskoField;
// Przyciski
private JButton returnButton;
private JButton submitButton;
```

Są to przede wszystkim elementy pochodzące z biblioteki Swing, które muszą być zdefiniowane jako pola klasy, gdyż są używane w innych metodach poza konstruktorem. Etykiety odpowiadają nazwowo polom tekstowym.



#### 4) Konstruktor

```
public ClientForm()
```

W konstruktorze ustawiane są funkcjonalności przycisków i etykiet oraz paneli.

#### 5) Metody

##### a. `public void actionPerformed(ActionEvent e)`

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == returnButton) {
        this.dispose();
        new AddFrame();
    }
    if (e.getSource() == submitButton) {
        String imie = imieField.getText();
        String nazwisko = nazwiskoField.getText();
        String telefon = telefonField.getText();
        if (telefon != "" && imie != "" && nazwisko != "") {
            insertToDataBase(telefon, imie, nazwisko);
        }
    }
}
```

Metoda pochodzi z interfejsu **ActionListener** i zajmuje się obsługą naciskania przycisków. Przycisk *returnButton* pozwala na powrót do okna **AddFrame**. Natomiast przycisk *submitButton* pobiera dane z pól tekstowych, potem następuje sprawdzenie czy nie są to puste wartości i wywołuje metodę **insertToDataBase(String telefon, String imie, String nazwisko)**

##### b. `private void insertToDataBase(String telefon, String imie, String nazwisko)`

Metoda obsługuje operację na bazie danych. Najpierw nawiązuje połączenie, a następnie gdy to połączenie już nawiąże dokonuje operacji wstawienia do bazy danych

```
// Udane połączenie
if (conn != null) {
    System.out.println("Połączenie z baza danych OK ! ");
    try {
        PreparedStatement pst = conn.prepareStatement("SELECT COUNT(*) AS ilosc FROM klient");
        ResultSet rs = pst.executeQuery();
        rs.next();
        int rows = rs.getInt("ilosc");
        rows++;
        System.out.println(rows);
        String sql = "INSERT INTO klient (id_klienta,telefon,imie,nazwisko) values (" + rows + ",'" + telefon
            + "','" + imie + "','" + nazwisko + "')";
        Statement myStmt = conn.createStatement();
        myStmt.executeUpdate(sql);
        pst.close();
        rs.close();
        this.dispose();
        new AddResultFrame("Operacja sie powiodla");
    } catch (SQLException e) {
        System.out.println("Bład podczas przetwarzania danych:" + e);
        this.dispose();
        new AddResultFrame("Operacja nie powiodla sie");
    }
}
```

Po dokonaniu wszystkich operacji wyłącza okno i tworzy nowe okno **AddResultFrame** z odpowiednim komunikatem.

Wszystkie klasy obsługujące formularze poza tym obsługującym wypożyczenia, czyli **EnsuranceForm EnsurerForm CarForm DepartmentForm WorkerForm**. są zbudowane i działają podobnie jak klasa **ClientForm**.

- **Klasa AddResultFrame**

Klasa, która pokazuje rezultat operacji dodawania do baz danych (obrazek 18).

### 1) Biblioteki

```
import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;
```

### 2) Deklaracja

```
public class AddResultFrame extends JFrame implements ActionListener
```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności.

### 3) Pola

```
// Przycisk
JButton returnButton;
// Label
JLabel message;
```

Są to przede wszystkim elementy pochodzące z biblioteki Swing, które muszą być zdefiniowane jako pola klasy, gdyż są używane w innych metodach poza konstruktorem.

### 4) Konstruktor

```
public AddResultFrame(String text)
```

W konstruktorze ustawiane są wartości pól elementów java Swing. Wiadomość końcowa jest ustawiana jako wartość zmiennej *text*.

## 5) Metody

### a. `public void actionPerformed(ActionEvent e)`

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == returnButton) {
        this.dispose();
        new AddFrame();
    }
}
```

Obsługuje przycisk *returnButton*, który wyłącza okno i wraca do interfejsu *AddFrame()*.

- **Klasa RentForm**

Jest to formularz do wprowadzania nowych wypożyczeń (Obrazek 16). Jest on inny od pozostałych formularzy

### 1) Biblioteki

```
import java.sql.*;

import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.FlowLayout;

import javax.swing.*;
import java.util.Vector;
```

### 2) Deklaracja

```
public class RentForm extends JFrame implements ActionListener
```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności.

### 3) Pola

```
// Panele
private JPanel header;
private JPanel content;
private JPanel footer;
// Etykiety z tekstem
private JLabel headerLabel;
// Przyciski
private JButton returnButton;
private JButton submitButton;

// CheckBox
private JCheckBox[] checkBoxTable;
```



Są to przede wszystkim elementy pochodzące z biblioteki Swing, które muszą być zdefiniowane jako pola klasy, gdyż są używane w innych metodach poza konstruktorem. Tablica typu `JCheckBox[]` była konieczna aby móc swobodnie pobierać, który samochód jest zaznaczony.

#### 4) Konstruktor

```
public RentForm()
```

W konstruktorze tworzone jest połączenie do bazy danych i pobierane są dane konieczne do stworzenia formularza. Formularz składa się z elementów z biblioteki Swing i w konstruktorze ustawiane są ich parametry.

#### 5) Metody

##### a. Public void actionPerformed(ActionEvent e)

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == returnButton) {
        this.dispose();
        new AddFrame();
    }
    if (e.getSource() == submitButton) {
        this.dispose();
        new RentClientWorker(areSelected(checkBoxTable, checkBoxTable.length));
    }
}
```

Metoda pochodzi z interfejsu **ActionListener** i zajmuje się obsługą naciskania przycisków. Przycisk *returnButton* pozwala na powrót do okna **AddFrame**. Natomiast *submitButton* zamyka okno i tworzy obiekt nowej klasy **RentClientWorker**.

##### b. Public void actionPerformed(ActionEvent e)

```
private Vector<Integer> areSelected(JCheckBox[] checkBoxTable, int n) {
    Vector<Integer> vect = new Vector<Integer>(0);
    for (int i = 0; i < n; i++) {
        if (checkBoxTable[i].isSelected()) {
            vect.addElement(Integer.valueOf(i + 1));
        }
    }
    return vect;
}
```

Metoda ta bierze jako argumenty tablicę obiektów `JCheckBox` i jej długość i sprawdza które z tych obiektów są zaznaczone. Odpowiednie indeksy są następnie zapisywane w wektorze typu `Integer`, który jest zwracany. Jest to informacja, które samochody zostały wypożyczone.

- **Klasa RentClientWorker**

Po wybraniu samochodów w klasie **Konstruktor** teraz wybieramy, który klient i który pracownik byli związani z daną transakcją (Obrazek 17).

### 1) Biblioteki

```
import java.sql.*;

import java.awt.Color;
import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.FlowLayout;

import javax.swing.*;
import java.util.Vector;
```

### 2) Deklaracja

```
public class RentClientWorker extends JFrame implements ActionListener
```

Klasa dziedziczy po klasie **JFrame** w celu ułatwienia obsługi okna, które teraz może być obsługiwane za pomocą parametru **this**.

Klasa implementuje interfejs **ActionListener**, aby móc nadać przyciskom funkcjonalności.

### 3) Pola

```
// Vector wypożyczon
private Vector<Integer> vector;
// Panele
private JPanel header;
private JPanel contentWorker;
private JPanel contentWorkerHeader;
private JPanel contentClient;
private JPanel contentClientHeader;
private JPanel footer;
// Label
private JLabel headerLabel;
private JLabel workerLabel;
private JLabel clientLabel;

// Przyciski
private JButton submitButton;
private JButton newClientButton;
private JButton returnButton;

// Tablice ButtonRadio
JRadioButton[] workerButtons;
ButtonGroup workerGroup;
JRadioButton[] clientButtons;
ButtonGroup clientGroup;
```

Poza polami, które są typowymi elementami java Swing, znajduje się tam `Vector<Integer>`, który przechowuje które samochodu zostały wypożyczone, a także dwie tablice `JRadioButton`, które są potrzebne stworzenia formularza i pomagają potem w znalezieniu zaznaczonej wartości.

#### 4) Konstruktor

```
public RentClientWorker(Vector<Integer> vect)
```

Konstruktor tworzy wygląd formularza (Obrazek ...), a także ustawia wartość pola, które jest wektorem Integerów, na wektor *vect*. Aby uzyskać wygląd formularza tworzone jest połączenie z bazą danych, w celu pobrania potrzebnych wartości.

#### 5) Metody

##### a. Public void actionPerformed(ActionEvent e)

```
@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == returnButton) {
        this.dispose();
        new RentForm();
    }
    if (e.getSource() == submitButton) {
        this.dispose();
        int idClient = clientChosen(workerButtons, workerButtons.length);
        int idWorker = workerChosen(clientButtons, clientButtons.length);
        System.out.println(idClient + " " + idWorker);
        new RentFinalise(idClient, idWorker, vector);
    }
    if (e.getSource() == newClientButton) {
        this.dispose();
        new ClientForm();
    }
}
```

Obsługuje przyciski. Przycisk *returnButton* wyłącza okno i przywraca nas do poprzedniego interfejsu na którym wybieramy samochody. Przycisk *newClientButton* włącza okno do dodawania klientów – klasę **ClientForm**. Natomiast przycisk **submitButton** przesyła dane tworząc nową klasę **RentFinalise**.

##### b. Private int clientChosen(JRadioButton[] clientButtons, int n)

```
private int clientChosen(JRadioButton[] clientButtons, int n) {
    for (int i = 0; i < n; i++) {
        if (clientButtons[i].isSelected()) {
            return i + 1;
        }
    }
    return 0;
}
```

Sprawdza, który z klientów został zaznaczony w formularzu i zwraca jego indeks. Jako argumenty przyjmuje tablicę przycisków *JRadioButtons* i długość tej tablicy.

c. `Private int workerChosen(JRadioButton[] workerButtons, int n)`

```
private int workerChosen(JRadioButton[] workerButtons, int n) {  
    for (int i = 0; i < n; i++) {  
        if (workerButtons[i].isSelected()) {  
            return i + 1;  
        }  
    }  
    return 0;  
}
```

Sprawdza, który z pracowników został zaznaczony w formularzu i zwraca jego indeks. Jako argumenty przyjmuje tablicę przycisków `JRadioButtons` i długość tej tablicy.

- **Klasa `RentFinalise`**

Klasa ta zbiera wszystkie dotychczas zebrane informacje o wypożyczeniu i dokonuje operacji zapisania do bazy danych

**1) Biblioteki**

```
import java.sql.*;  
  
import java.util.Vector;
```

**2) Deklaracja**

```
public class RentFinalise
```

**3) Pola**

```
private int idClient;  
private int idWorker;  
private Vector<Integer> cars;
```

Pola przechowują dotychczasowe informacje zebrane w poprzednich formularzach

**4) Konstruktor**

```
public RentFinalise(int idC,int idP,Vector<Integer> vect)
```

Konstruktor ustawia odpowiednie pola wartościami, otwiera połączenie z bazą danych oraz wprowadza je do bazy danych

```

if(conn!=null){
    System.out.println("Połączenie z bazą danych OK ! ");
    try{
        PreparedStatement pst=conn.prepareStatement("SELECT COUNT(*) AS ilosc FROM wypozyczenie");
        ResultSet rs=pst.executeQuery();

        Date dataWypozyczenia=new Date(System.currentTimeMillis());

        rs.next();
        int rows = rs.getInt("ilosc");
        rows++;
        String sql="INSERT INTO Wypozyczenie (id_wypozyczenia, id_klienta,id_pracownika,data_wypozyczenia,data_zwrotu) values ('"+rows+"','"+idKli
        System.out.println(sql);
        Statement myStat1 = conn.createStatement();
        myStat1.executeUpdate(sql);

        //Update samochodow
        for(Integer i:cars){
            String command="UPDATE samochod SET id_wypozyczenia='"+rows+"' WHERE id_samochodu='"+i";
            System.out.println(command);
            Statement myStat2 = conn.createStatement();
            myStat2.executeUpdate(command);
        }
        new AddResultFrame("Operacja sie powiodla");
    }
    catch(SQLException e){
        System.out.println("Blad podczas przetwarzania danych:"+e);
        new AddResultFrame("Operacja sie nie powiodla");
    }
}
}

```