Politechnika Łódzka

Kolegium Logistyki

Zaawansowane Techniki Programowania

DOKUMENTACJA PROJEKTOWA

APLIKACJA DO OBLICZANIA CEN PRZESYŁEK

Autorzy:

Piotr Stępniowski 195383

Piotr Kostyk 195347

Spis treści:

[Interfejs użytkownika 3](#_Toc485577464)

[Ekran tytułowy 3](#_Toc485577465)

[„NADAJ PRZESYŁKĘ” 4](#_Toc485577466)

[„SPRAWDŹ PRZESYŁKĘ” 6](#_Toc485577468)

[„ZOBACZ BAZĘ DANYCH” 8](#_Toc485577469)

[Działanie programu 8](#_Toc485577470)

[Bazy danych: 9](#_Toc485577471)

[Klasy: 10](#_Toc485577472)

[Klasa baza: 10](#_Toc485577473)

[Klasy z pakietu model: 15](#_Toc485577474)

[Klasa Obliczenia: 16](#_Toc485577475)

[Klasa Model: 18](#_Toc485577476)

[Klasa Warunki: 19](#_Toc485577477)

[Klasa Test: 20](#_Toc485577478)

[Klasa Okno: 20](#_Toc485577479)

[Różnice w projektach: 24](#_Toc485577480)

[Spis ilustracji: 26](#_Toc485577481)

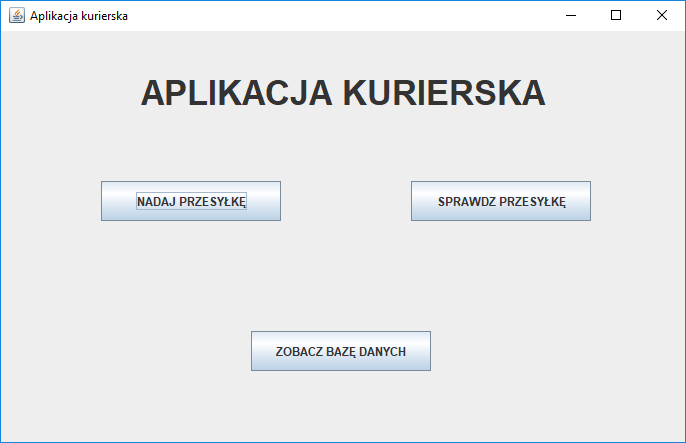
# Interfejs użytkownika

## Ekran tytułowy

Cała aplikacja działa w jednym okienku, zmienia się tylko widoczność obiektów po odpowiednim działaniu użytkownika (naciśnięcie odpowiedniego przycisku, wybranie pozycji w rozwijanej liście).

Po włączeniu aplikacji wita nas ekran tytułowy (rys. 1), na którym na samej górze znajduje się nazwa aplikacji „**APLIKACJA KURIERSKA**” a poniżej widoczne są trzy przyciski, po kliknięciu których przechodzimy do dalszych części programu:

1. NADAJ PRZESYŁKĘ
2. SPRAWDŹ PRZESYŁKĘ
3. ZOBACZ BAZĘ DANYCH



Rysunek : Ekran tytułowy

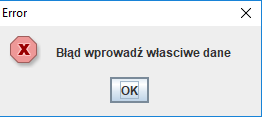
## „NADAJ PRZESYŁKĘ”

Po kliknięciu na ten przycisk zostaje wywołane okno formularza, w którym następuje dodanie zamówienia do bazy danych (rys. 2). Użytkownik wypełnia odpowiednimi danymi kolejne miejsca a formularzu (ilość przesyłek, waga paczki, wymiary paczki), po czym z rozwijanych list wybiera miejsca nadania i odbioru paczki. Warto zwrócić uwagę, że do każdego województwa jest przypisanych pięć miast, które się w nim znajdują. Na sam koniec użytkownik uzupełnia pole rabatu, który maksymalnie może wynosić 20%, wpisanie liczy większej niż 20 daje nadal 20% rabatu, dodatkowo pozostawienie pola pustym jest równoznaczne z 0% rabatem.

## C:\Users\irons\AppData\Local\Microsoft\Windows\INetCache\Content.Word\formularz nadania.png

Rysunek : Formularz nadania przesyłki

Jeśli dane zostaną uzupełnione źle, wyskoczy komunikat informujący o błędnych danych (rys. 3).



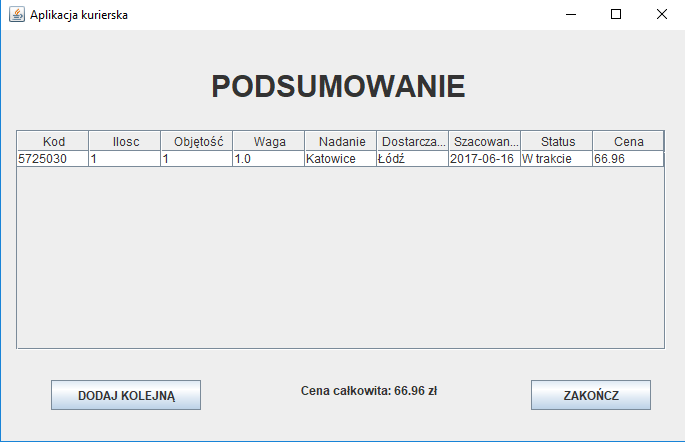
Rysunek : Okienko błędu

Na samym dole znajdują się dwa przyciski:

* **COFNIJ** – dzięki niemu wracamy do ekranu tytułowego
* **DALEJ** – klikając go przechodzimy do okna podsumowania

Jeśli dane zostały poprawnie wprowadzone pokazuje się okno podsumowania (rys. 4), w którym są widoczne wszystkie zamówienia złożone podczas tej sesji zamówienia. Podane są nie tylko wprowadzone dane ale również szacowana data dostarczenia, status przesyłki oraz cena jednostkowa zamówienia. Każda przesyłka dostaje również swój unikatowy kod.

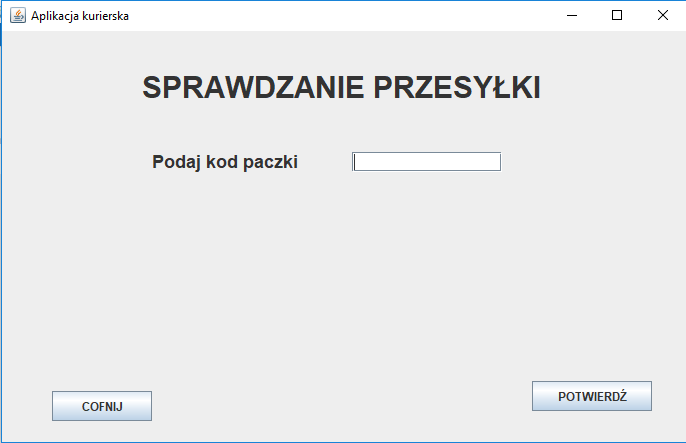
Istnieje możliwość dodania kolejnego zamówienia poprzez kliknięcie przycisku „**DODAJ KOLEJNĄ**”, który prowadzi do poprzedniego okna z formularzem nadania przesyłki. Po prawej stronie znajduje się przycisk „**ZAKOŃCZ”** prowadzący do ekranu tytułowego. Między przyciskami na samym dole znajduje się również informacja o całkowitej cenie zamówień w bieżącej sesji. Po wyjściu przez przycisk „**ZAKOŃCZ”** sesja zostaje wyczyszczona.



Rysunek :Okno podsumowania zamówień

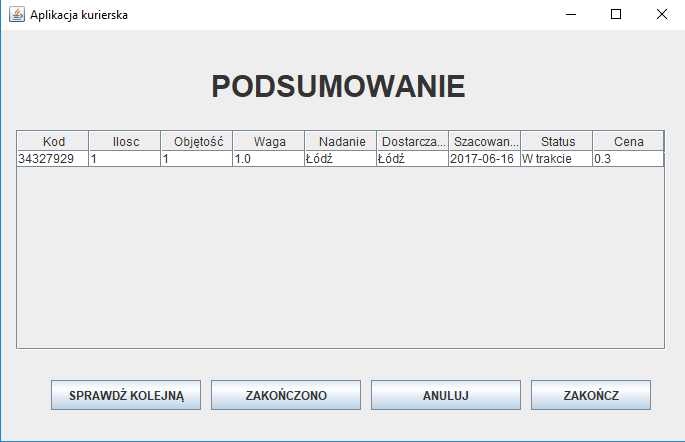
## „SPRAWDŹ PRZESYŁKĘ”

Kolejną funkcją programu jest sprawdzanie wprowadzonych wcześniej zamówień na podstawie kodu paczki. Aby wejść w tą część programu na ekranie tytułowym klikamy przycisk „**SPRAWDŹ PRZESYŁKĘ**”. Następnie pojawia nam się okno (rys. 5) z polem, w którym użytkownik podaje kod paczki. Aby przejść dalej należy wcisnąć przycisk „**POTWIERDŹ**” lub można wyjść do menu głównego przyciskiem „**COFNIJ**”.



Rysunek :Okno sprawdzania przesyłki

Jeśli użytkownik źle wpisze kod lub nie wpisze go w ogóle wyskakuje okno błędu (rys. 3), jeśli wpisze zły numer program nie pokaże żadnego zamówienia a jeśli wszystko będzie poprawne wyskoczy okno z pojedynczym, szukanym przez użytkownika wpisem (rys 6.).

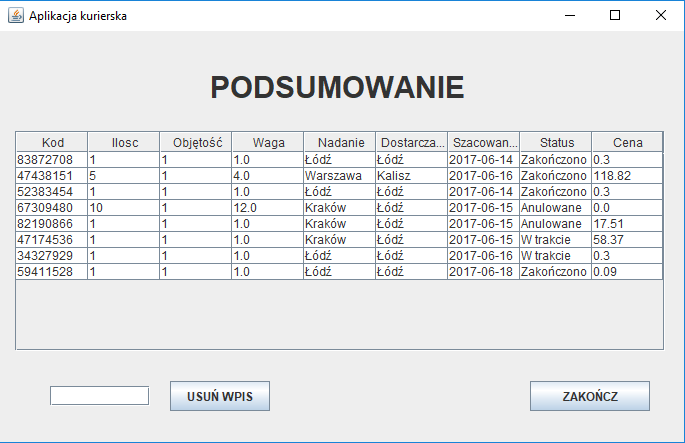


Rysunek : Wyszukany wpis

Istnieje możliwość ponownego wyszukania innego zamówienia poprzez kliknięcie przycisku „**SPRAWDŹ KOLEJNĄ**”, który prowadzi do okna z polem do wpisania kodu zamówienia, oraz zmiana statusu przesyłki na „Zakończono” lub „Anulowane” (rys. 5). Jeśli użytkownik chce wyjść do ekranu głównego klika przycisk „**ZAKOŃCZ**”.

## „ZOBACZ BAZĘ DANYCH”

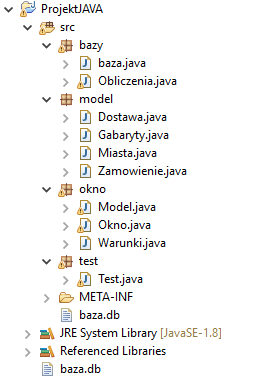
Ostatnią oferowaną opcją programu jest wyświetlenie całej bazy zamówień oraz usuwanie wpisów na podstawie kodu zamówienia. Użytkownik wchodzi w nią poprzez kliknięcie na ekranie tytułowym przycisku „**ZOBACZ BAZĘ DANYCH**”. Ukazuje się wtedy okienko zawierające listę wszystkich zamówień w bazie (rys. 7). Na samym dole po lewej stronie znajduje się pole do wpisania kodu paczki do usunięcia, który użytkownik potwierdza przyciskiem „**USUŃ WPIS**”. Jeśli chce wyjść do ekranu tytułowego klika „**ZAKOŃCZ**”.



Rysunek :Okno z całą bazą zamówień

# Działanie programu

Program działa w jednym oknie, zmieniana jest tylko widoczność poszczególnych elementów na podstawie działania użytkownika. Zawiera 4 pakiety, w których znajduje się 10 klas. Oraz wykorzystuje interfejs JDBC, dzięki któremu łączy się z bazą danych oraz możliwe jest przeprowadzanie na niej określonych działań. Konstrukcja programu wygląda następująco:



Rysunek : Drzewo konstrukcji programu

## Bazy danych:

W celu działania programu zostały stworzone 4 bazy danych:

* Baza miasta zawierająca kolumny:

- id\_miasta – int, autoincrement

- Nazwa\_miasta - varchar(255)

- Województwo - varchar(255)

- Szerokosc - double

- Dlugosc – double

Zawiera wszystkie dane dotyczące miast ( nazwę, województwo do którego należy oraz współrzędne geograficzne).

* Baza gabaryty zawierająca kolumny:

- id\_gabarytu – int, autoincrement

- Objetosc\_paczki – int

- Max\_cena – double

Zawiera dane dotyczące cen gabarytów w poszczególnych przedziałach objętościowych.

* Baza dostawa zawierająca kolumny:

- id\_dostawy – int, autoincrement

- Województwo\_dostawcy – varchar(255)

- Województwo\_odbiorcy – varchar(255)

- Cena\_transportu – double

- Czas\_dostawy – int

Zawiera dane dotyczące ceny transportu oraz czasu dostawy (w dniach) miedzy danymi województwami.

* Baza zamówienie zawierająca kolumny:

- id\_zamowienia – int, autoincrement

- Ilość – int

- Objetosc – int

- Waga – double

- Nadanie – varchar(255)

- Dostarczenie – varchar(255)

- Szacowana\_data\_odbioru – varchar(255)

- status – varchar(255)

- kod – int

- cena – double

Zawiera wszystkie dane dotyczące zamówień.

## Klasy:

### Klasa baza:

Należy do pakietu bazy i znajdują się w niej wszystkie działania wykonywane na bazie podczas działania programu. Wszystkie zawarte tutaj metody są typu booleanoraz zawierają raportowanie błędów metodą try catch. Wykorzystuje następujące biblioteki:

**-** java.sql.Connection – Wykorzystywana do łączenia z bazą danych

- java.sql.DriverManager – Wykorzystywana do zarządzania sterownikami bazy JDBC

- java.sql.PreparedStatement – Wykorzystywana do poleceń SQL

- java.sql.ResultSet – Wykorzystywana do zapisywania pobranych z bazy danych obiektów

- java.sql.SQLException – Wykorzystywana do raportowania błędów SQL

- java.sql.Statement – Wykorzystywana do wykonywania poleceń SQL

- java.util.LinkedList – Wykorzystywana do tworzenia list typu LinkedList

- java.util.List – Wykorzystywana do tworzenia kolekcji

Zostały również zaimportowane modele z pakietu model które są używane do tworzenia list elementów czyli w naszym przypadku baz danych:

- model.Miasta;

- model.Gabaryty;

- model.Dostawa;

- model.Zamowienie;

Na początku klasy zostaje załadowany sterownik bazy danych oraz zostaje podana do niej ścieżka. Zadeklarowana zostaje również zmienna klasy Connection która będzie wykorzystywana do ustanawiania połączenia z bazą danych oraz zmienna klasy Statement używana do przechowywania poleceń SQL.

**public** **static** **final** String ***DRIVER*** = "org.sqlite.JDBC";

**public** **static** **final** String ***DB\_URL*** = "jdbc:sqlite:baza.db";

**private** Connection conn;

**private** Statement stat;

**Metody:**

1. **Łączenie z baza:**

**public** baza(){

**try** {

Class.*forName*(baza.***DRIVER***);

} **catch** (ClassNotFoundException e) {

System.***err***.println("Brak sterownika JDBC");

e.printStackTrace();

}

**try** {

conn = DriverManager.*getConnection*(***DB\_URL***);

stat = conn.createStatement();

} **catch** (SQLException e) {

System.***err***.println("Problem z otwarciem polaczenia");

e.printStackTrace();

}

createTables();

}

Metoda ta próbuje w blokach try połączyć się z bazą danych podając zadeklarowany wcześniej adres bazy oraz ładując jej sterownik. Jeśli coś pójdzie nie tak blok catch zwraca bląd. Na koniec jest włączana metoda createTables().

1. **Tworzenie baz danych:**

**public** **boolean** createTables() {

String createMiasta = "CREATE TABLE IF NOT EXISTS miasta (id\_miasta INTEGER PRIMARY KEY AUTOINCREMENT, Nazwa\_miasta varchar(255), Województwo varchar(255), Szerokosc double, Dlugosc double)";

String createGabaryty = "CREATE TABLE IF NOT EXISTS gabaryty (id\_gabarytu INTEGER PRIMARY KEY AUTOINCREMENT, Objetosc\_paczki int, Max\_cena double)";

String createDostawa = "CREATE TABLE IF NOT EXISTS dostawa (id\_dostawy INTEGER PRIMARY KEY AUTOINCREMENT, Województwo\_dostawcy varchar(255), Województwo\_odbiorcy varchar(255), Cena\_trasportu double, Czas\_dostawy int)";

String createZamowienie = "CREATE TABLE IF NOT EXISTS zamowienie (id\_zamowienia INTEGER PRIMARY KEY AUTOINCREMENT, Ilość int,Objetosc int, Waga double, Nadanie varchar(255), Dostarczenie varchar(255), Szacowana\_data\_odbioru varchar(255), status varchar(255), kod int, cena double)";

**try** {

stat.execute(createMiasta);

stat.execute(createGabaryty);

stat.execute(createDostawa);

stat.execute(createZamowienie);

} **catch** (SQLException e) {

System.***err***.println("Blad przy tworzeniu tabeli");

e.printStackTrace();

**return** **false**;

}

**return** **true**;

}

Metoda ta tworzy bazy danych (jeśli nie istnieją) poprzez użycie zadeklarowanych w ciągach znakowych typu String poleceń SQL. Jeśli połączenie zostało pomyślnie ustanowione oraz polecenie SQL napisane poprawnie bazy zostają stworzone. Jeśli coś pójdzie nie tak jak powinno blok try catch zwraca określony błąd.

1. **Wprowadzanie danych do bazy:**

**public** **boolean** insertMiasta(String nazwaMiasta, String wojewodztwo, **double** szer, **double** dlug) {

**try** {

PreparedStatement prepStmt = conn.prepareStatement(

"insert into miasta values (NULL, ?, ?,?,?);");

prepStmt.setString(1, nazwaMiasta);

prepStmt.setString(2, wojewodztwo);

prepStmt.setDouble(3, dlug);

prepStmt.setDouble(4, szer);

prepStmt.execute();

} **catch** (SQLException e) {

System.***err***.println("Blad przy wstawianiu miasta");

e.printStackTrace();

**return** **false**;

}

**return** **true**;

}

Metoda ta wprowadza dane do bazy danych. W przypadku błędu jest wyświetlany błąd z bloku try catch. W klasie znajdują się również funkcje o takiej samej budowie do uzupełnienia pozostałych baz:

**- public** **boolean** insertGabaryty(**int** objetoscPaczki, **double** maxCena)

**- public** **boolean** insertDostawa(String wojNad, String wojOdb, **double** cena, **int** czasDost)

**- public** **boolean** insertZamowienie(**int** ilosc, **int** objetosc, **double** waga, String nadanie, String odbior, String dataOdb, String status, **int** kod, **double** cena)

1. **Tworzenie listy na podstawie bazy danych**

**public** List<Miasta> selectMiasta() {

List<Miasta> miasta = **new** LinkedList<Miasta>();

**try** {

ResultSet result = stat.executeQuery("SELECT \* FROM miasta");

**int** id;

**double** szer, dlug;

String nazwaMiasta, wojewodztwo;

**while**(result.next()) {

id = result.getInt("id\_miasta");

nazwaMiasta = result.getString("Nazwa\_miasta");

wojewodztwo = result.getString("Województwo");

szer = result.getDouble("Dlugosc");

dlug = result.getDouble("Szerokosc");

miasta.add(**new** Miasta(id, nazwaMiasta, wojewodztwo, dlug, szer));

}

} **catch** (SQLException e) {

e.printStackTrace();

**return** **null**;

}

**return** miasta;

}

Metoda ta tworzy listę na podstawie bazy danych. W bloku try zostaje wybrane wszystko z bazy a później odpowiednio przypisane do konkretnych miejsc w liście. Na koniec zwraca listę, którą później można wyświetlić i która jest interpretacją bazy danych. Jeśli coś pójdzie nie tak zostaje zwrócony błąd. Metoda ta jest zastosowana później analogicznie dla pozostałych baz:

- **public** List<Dostawa> selectDostawa()

-**public** List<Gabaryty> selectGabaryty()

-**public** List<Zamowienie> selectZamowienie()

Istnieje również piąta analogiczna metoda:

**public** List<Zamowienie> selectZamowienie(**int** kod2)

Jest ona używana do wyciągnięcia z bazy jednego wiersza na podstawie kodu zamówienia poprzez zmodyfikowaną wersję polecenia SQL:

"SELECT \* FROM zamowienie WHERE kod =" + kod2

1. **Wyciąganie z bazy szerokości i długości geograficznej do obliczeń**

**public** **double** Oblicz(**double** szerokosc2, String indeks2 ){

String szerokosc = "SELECT Szerokosc FROM miasta WHERE Nazwa\_miasta = '" + indeks2+"'";

ResultSet resultset = **null**;

**try**{

resultset = stat.executeQuery(szerokosc);

szerokosc2 = resultset.getDouble("Szerokosc");

}**catch** (SQLException e) {

System.***err***.println("Blad przy wyszukiwaniu wiersza");

e.printStackTrace();

}

**return** szerokosc2;}

Metoda ta wykorzystywana jest do wyciągnięcia z bazy danych szerokości geograficznej miasta w celu dalszego użycia jej do obliczeń. Analogicznie później zastosowana jest metoda wyciągająca długość geograficzną, cenę za km między województwami, cenę paczki do operacji anulowania, cenę za gabaryt oraz ilość dni potrzebnych do transportu zamówienia :

**public** **double** Oblicz2(**double** dlugosc2, String indeks)

**public** **double** cena(**double** dlugosc2, String indeks, String indeks2)

**public** **int** dni(**int** dlugosc2, String indeks, String indeks2)

**public** **double** CenazaPaczke(**double** cenapaczki, **int** indeks)

**public** **double** CenaAnuluj(**double** cenapaczki, **int** kodPaczki2)

1. **Zmiana statusu przesyłki na “Zakończono”**

**public** **void** zakoncz(**int** kodPaczki2){

String zakoncz = "UPDATE zamowienie SET status = 'Zakończono' WHERE kod = " + kodPaczki2;

**try**{

stat.executeUpdate(zakoncz);

}

**catch**(SQLException e){

System.***err***.println("Blad przy zmianie statusu");

e.printStackTrace();

}

}

Metoda typu void, czyli nic nie zwraca. Służy do zmiany statusu przesyłki na podstawie kodu paczki. Jeśli coś pójdzie nie tak blok try catch zwraca określony błąd.

1. **Zmiana statusu przesyłki na „Anulowane”**

**public** **void** anuluj(**int** kodPaczki2, **double** cena){

String zakoncz = "UPDATE zamowienie SET status = 'Anulowane', cena = " + cena + " WHERE kod = " + kodPaczki2;

**try**{

stat.executeUpdate(zakoncz);

}

**catch**(SQLException e){

System.***err***.println("Blad przy zmianie statusu");

e.printStackTrace();

}

}

Metoda typu void, czyli nic nie zwraca. Służy do zmiany statusu przesyłki na podstawie kodu paczki. W porównaniu do metody zakoncz, jest tutaj ustawiana również nowa cena przesyłki. Jeśli coś pójdzie nie tak blok try catch zwraca określony błąd.

1. **Zamykanie połączenia z bazą**

**public** **void** closeConnection() {

**try** {

conn.close();

} **catch** (SQLException e) {

System.***err***.println("Problem z zamknieciem polaczenia");

e.printStackTrace();

}

}

Jest to metoda typu void czyli nic nie zwraca. Służy do zamknięcia połączenia z bazą. Jeśli coś pójdzie nie tak blok try catch zwraca określony błąd.

### Klasy z pakietu model:

W pakiecie model są cztery klasy (Gabaryty, Dostawa, Miasta, Zamowienie), których implementacja wygląda tak samo. Zawiera ona deklaracje zmiennych, gettery, setery, metody toString oraz konstruktory. Poniżej przykładowa klasa Gabaryty:

**public** **class** Gabaryty {

**private** **int** id;

**private** **int** objetoscPaczki;

**private** **double** maxCena;

**public** Gabaryty(**int** id, **int** objetoscPaczki, **double** maxCena) {

**this**.id = id;

**this**.objetoscPaczki = objetoscPaczki;

**this**.maxCena = maxCena;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **int** getObjetoscPaczki() {

**return** objetoscPaczki;

}

**public** **void** setObjetoscPaczki(**int** objetoscPaczki) {

**this**.objetoscPaczki = objetoscPaczki;

}

**public** **double** getMaxCena() {

**return** maxCena;

}

**public** **void** setMaxCena(**double** maxCena) {

**this**.maxCena = maxCena;

}

**public** String toString(){

**return** "["+ id +"] - "+ objetoscPaczki +" - "+ maxCena;

}

}

### Klasa Obliczenia:

Klasa zawierająca metody niezbędne do obliczeń używanych w klasie Okno. Zawiera następujące biblioteki:

- java.lang.Math – Zawiera podstawowe operacje matematyczne

- java.text.SimpleDateFormat – Zawiera prosty układ daty

- java.util.Calendar – Zawiera operacje związane z datami

**-** java.util.Date – Zawiera obsługę dat

**Metody:**

1. **Generowanie kodu przesyłki**

**public** **int** kod(**int** g){

**int** e = (**int**) (Math.*random*()\*g);

**return** e;

}

Funkcja używana do stworzenia randomowego kodu przydzielanego do przesyłki.

1. **Funkcja haversine**

**final** **static** **double** ***R*** = 6372.8; // W kilometrach

**public** **static** **double** haversine(**double** dl1, **double** szer1, **double** dl2, **double** szer2) {

**double** dLat = Math.*toRadians*(dl2 - dl1);

**double** dLon = Math.*toRadians*(szer2 - szer1);

dl1 = Math.*toRadians*(dl1);

dl2 = Math.*toRadians*(dl2);

**double** a = Math.*pow*(Math.*sin*(dLat / 2),2) + Math.*pow*(Math.*sin*(dLon / 2),2) \* Math.*cos*(dl1) \* Math.*cos*(dl2);

**double** c = 2 \* Math.*asin*(Math.*sqrt*(a));

**return** ***R*** \* c;

}

Funkcja używana do obliczania odległości po linii prostej między dwoma miastami na podstawie długości i szerokości geograficznych danych miast.

1. **Obliczanie daty dostarczenia zamówienia**

**public** String data(**int** iloscdni){

Calendar cal = Calendar.*getInstance*();

cal.add(Calendar.***DATE***, iloscdni);

SimpleDateFormat format1 = **new** SimpleDateFormat("yyyy-MM-dd");

String formatted = format1.format(cal.getTime());

**return** formatted;

}

Program na podstawie ilości dni wyciągniętych z bazy danych dodaje do aktualnie pobranej daty ta ilość i wylicza przybliżony czas dostarczenia przesyłki i zamienia go w ciąg znakowy.

1. **Liczenie objętości paczki**

**public** **int** objetosc(**int** x, **int** y, **int** z, **int** objetosc){

objetosc = x\*y\*z;

**return** objetosc;

}

Na zmiennych pobranych z pól w formularzu program wylicza objętość paczki.

1. **Obliczanie ceny za gabaryt**

**public** **int** wartosc(**int** objetosc, **int** i ){

**if**(objetosc > 0 && objetosc <200){

i = 1;

}

**else** **if**(objetosc >=200 && objetosc <500 ){

i = 2;

}

**else** **if**(objetosc >=500 && objetosc <1000){

i = 3;

}

**else** **if**(objetosc >=1000){

i = 4;

}

**return** i;

}

Metoda wylicza odpowiednia cenę za gabaryt.

1. **Wyliczanie ceny ostatecznej**

**public** **double** CenaOstateczna(**double** wagaPaczki, **int** objetoscPaczki, **double** cenaZaGabaryt, **double** iloscKm, **double** cenaZaKm, **int** iloscPaczek, **double** rabat){

**double** cena,cenarab,rabb;

rabb=rabat;

Math.*round*(wagaPaczki);

**if** (rabb >20)

{

rabb=20;

}

cenarab = ((iloscPaczek\*wagaPaczki \* 0.2 + objetoscPaczki\*cenaZaGabaryt + iloscKm\*cenaZaKm))\*(rabb/100);

cena=(iloscPaczek\*wagaPaczki \* 0.2 + objetoscPaczki\*cenaZaGabaryt + iloscKm\*cenaZaKm);

cena=cena-cenarab;

**return** cena;

}

Metoda wykorzystuje wszystkie wyliczone i pobrane z bazy zmienne i wylicza cenę ostateczną zamówienia. Dodatkowo dołożony jest warunek ustawiający rabat na maksimum 20%.

### Klasa Model:

Jest to klasa zawierająca układ wyświetlanej tabeli z zamówieniami na podstawie modelu z klasy Zamowienie z pakietu model. Zawiera biblioteki do obsługi abstrakcyjnych modeli tabelowych oraz list.

**import** java.util.List;

**import** javax.swing.table.AbstractTableModel;

**import** model.Zamowienie;

**public** **class** Model **extends** AbstractTableModel{

**private** List<Zamowienie> listazamowienie ;

**private** **final** **static** Object[] ***nazwyKolumn*** = { "Kod", "Ilosc", "Objętość", "Waga", "Nadanie","Dostarczanie", "Szacowana data odbioru", "Status", "Cena"};

**public** Model(List<Zamowienie> listazamowienie ) {

**this**.listazamowienie = listazamowienie;

}

@Override

**public** **int** getColumnCount() {

**return** ***nazwyKolumn***.length;

}

@Override

**public** **int** getRowCount() {

**if**(listazamowienie.size() == 0) {

**return** 0;

}

**return** listazamowienie.size();

}

@Override

**public** Object getValueAt(**int** row, **int** column) {

**if**(listazamowienie == **null**) {

**return** **null**;

}

Zamowienie oneZamowienie = listazamowienie.get(row);

**switch** (column) {

**case** 0:

**return** oneZamowienie.getKod();

**case** 1:

**return** oneZamowienie.getIlosc();

**case** 2:

**return** oneZamowienie.getObjetosc();

**case** 3:

**return** oneZamowienie.getWaga();

**case** 4:

**return** oneZamowienie.getNadanie();

**case** 5:

**return** oneZamowienie.getOdbior();

**case** 6:

**return** oneZamowienie.getDataOdb();

**case** 7:

**return** oneZamowienie.getStatus();

**case** 8:

**return** oneZamowienie.getCena();

**default**:

**return** oneZamowienie;

}

}

@Override

**public** String getColumnName(**int** column) {

**return** ***nazwyKolumn***[column].toString();

}

}

### Klasa Warunki:

Zawiera warunki potrzebne do prawidłowego wprowadzania danych do pól tekstowych.

**Metody:**

**public** **static** **boolean** isParsableAsLong(**final** String s) {

**try** {

Long.*valueOf*(s);

**return** **true**;

} **catch** (NumberFormatException numberFormatException) {

**return** **false**;

}

}

Metoda pobiera ciąg znakowy pobrany z pola JTextField i sprawdza czy można go sparsować na odpowiedni typ danych. W tym przypadku na long. Analogicznie wyglądają pozostałe metody.

- **public** **static** **boolean** isParsableAsDouble(**final** String s)

-**public** **static** **boolean** isParsableAsInteger(**final** String s)

Ostatnia metoda **public** **static** **boolean** isParsableAsDoubleRab(**final** String s) została napisana specjalnie dla rabatu ponieważ gdy użytkownik nie wpisze żadnej wartości w pole Rabatu ma być on ustawiany na 0 a nie jak do tej pory wyrzucać błędy.

### Klasa Test:

Zawiera wszystkie konstruktory i wywołania okna. Do poprawnego działania bazy trzeba dodać 80 wpisów do bazy miasta które mają konstrukcję:

a.insertMiasta("Tychy","Śląskie",18.59,50.08); gdzie podawane wartości to: nazwa miasta, województwo oraz współrzędne geograficzne

Następnie należy dodać 256 wpisów do bazy Dostawa które mają kostrukcję:

a.insertDostawa( "Łódzkie", "Łódzkie", 0.20, 1); gdzie podawane wartości to: województwo nadania, województwo odbioru, cena za km oraz ilość potrzebnych dni na transport.

Następnie należy podać gabaryty. Konstrukcja wygląda następująco:

a.insertGabaryty(100, 0.1); gdziepodawane wartości to gabaryt oraz cena za gabaryt.

Na koniec następuje wywołanie okna oraz ustawienie domyślnej operacji przy wyłączeniu programu:

Okno okno = **new** Okno(zamowienie);

okno.setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

okno.setVisible(**true**);

Oraz zamknięcie połączenia z bazą:

a.closeConnection();

### Klasa Okno:

Zawiera wszystkie metody, deklaracje zmiennych oraz operacje związane z wyświetlaniem i obsługa okna programu. Korzysta z bibliotek:

javax.swing.\* - zawiera operacje związane z graficznym interfejsem

java.awt.\* - Obsługuje grafikę javie

java.awt.event.ActionEvent – Obsługuje zdarzenia

java.awt.event.ActionListener – Obsługuje nasłuchiwanie działania użytkownika

java.sql.Date – Obsługuje operacje na datach

java.util.ArrayList – Obsługuje tworzenie i działanie na listach typu Array

java.util.List - Obsługuje tworzenie i działanie na listach

Na samym początku zadeklarowane zostają zmienne od przycisków, pól tekstowych, list rozwijanych, tabel, oraz list i dodatkowych zmiennych:

**private** JButton nadajPrzesylke, sprawdzPrzesylke, dalej, potwierdz, zakoncz, zakoncz2, zakoncz3, dodaKol, sprKol, zobBaze, cofnij, cofnij2, usunWpis;

**private** JLabel tytul, formularz, wybPrzes, wymPaczki, wagaPaczki, wybWoj, miejNad, miejDost, sprawdz, podajKod, pods, rabatto, cenacalk;

**private** JTextField wymX, wymY, wymZ, wagaPole, iloscPole, kodP, rabattokek, usunWpisy;

**private** JComboBox<String> mNad, mNad2, mDostWoje, mDostMiast;

**private** JTable table;

**private** JScrollPane scrollPane;

**private** Model tabModel;

List<Zamowienie> zamowienia = **new** ArrayList<>();

List<Zamowienie> zamowienia2 = **new** ArrayList<>();

List<Zamowienie> zamowienia3 = **new** ArrayList<>();

Date currentDate;

**double** cenacalko;

**int** itemcount;

Ustawiony zostaje rozmiar okna i nazwa programu:

setSize(700,450);

setTitle("Aplikacja kurierska");

setLayout(**null**);

Następnie utworzone zostają wszystkie składowe (przykładowo dla każdego, reszta analogicznie):

**JButton:**

nadajPrzesylke = **new** JButton("NADAJ PRZESYŁKĘ");

nadajPrzesylke.setBounds(100, 150, 180, 40);

add(nadajPrzesylke);

nadajPrzesylke.addActionListener(**this**);

**JLabel:**

tytul = **new** JLabel("APLIKACJA KURIERSKA");

tytul.setBounds(140, 10, 600, 100);

tytul.setFont(**new** Font("SansSerif", Font.***BOLD***, 35));

add(tytul);

**JTextField:**

iloscPole = **new** JTextField("");

iloscPole.setBounds(250, 120, 160, 20);

add(iloscPole);

iloscPole.setVisible(**false**);

**JComboBox:**

Zadeklarowana zostaje tablica Wojewodztwa:

String [] Wojewodztwa = {

"Łódzkie","Mazowieckie","Śląskie","Wielkopolskie","Małopolskie","Dolnośląskie","Pomorskie","Lubelskie","Podkarpackie",

"Kujawsko-pomorskie","Zachodniopomorskie",

"Warmińsko-mazurskie","Świętokrzyskie","Podlaskie","Lubuskie","Opolskie"

};

mNad = **new** JComboBox(Wojewodztwa);

mNad.setBounds(300, 260, 160, 20);

add(mNad);

mNad.setVisible(**false**);

mNad.addActionListener(**this**);

**JScrollPane:**

scrollPane = **new** JScrollPane();

scrollPane.setBounds(15, 100, 650, 220);

scrollPane.setVisible(**false**);

add(scrollPane);

**Model tablicowy:**

table = **new** JTable(tabModel);

table.setBounds(15, 100, 650, 220);

add(table);

table.setVisible(**false**);

scrollPane.setViewportView(table);

Następnie następuje nasłuchiwanie zdarzeń a metodzie **public** **void** actionPerformed(ActionEvent e)

Głównie reszta część kodu opiera się na zmienianiu widoczności odpowiednich elementów za pomocą (obiekt).setVisible(true/false).

Listy rozwijane są zaprojektowane aby w zależności od wybranego województwa w drugiej rozwijanej liście obok pokazywał się inny zestaw miast. Jest to objęte następującymi warunkami:

**if** (mNad2 == **null**) {

mNad2 = **new** JComboBox(Lodz);

mNad2.setBounds(490, 260, 160, 20);

add(mNad2);

mNad2.setVisible(**true**);

} **else** {

**for** (String a : Lodz) {

mNad2.addItem(a);

}}

Dla reszty (32 ogólnie) wygląda to tak samo.

Każde pole JTextField jest również opisane warunkiem sprawdzającym poprawność wprowadzonych danych. Jest wywoływany z klasy Warunki.

Dodatkowo w tej klasie są wywołane wszystkie metody liczące z klasy Obliczenia ( po kliknięciu przycisku „**DALEJ”**).

baza c = **new** baza();

**double** cena;

cena = c.cena(0, miasto3, miasto4);

c.closeConnection();

**ilość dni dostawy:**

baza d = **new** baza();

**int** dni;

dni = d.dni(0, miasto3, miasto4);

d.closeConnection();

**Wywołanie daty:**

String data = o.data(dni);

**Liczenie odległości między miastami:**

baza b = **new** baza();

**double** zmienna = 0;

zmienna = b.Oblicz(0, miasto);

**double** zmienna2 = 0;

zmienna2 = b.Oblicz2(0, miasto);

**double** zmienna3 = 0;

zmienna3 = b.Oblicz(0, miasto2);

**double** zmienna4 = 0;

zmienna4 = b.Oblicz2(0, miasto2);

**double** odleglosc = o.*haversine*(zmienna, zmienna2, zmienna3, zmienna4);

b.closeConnection();

**Liczenie objętości paczki:**

**int** objetoscp;

objetoscp = o.objetosc(wymiarX2, wymiarY2, wymiarZ2, 0);

**Cena za gabaryt:**

**int** index;

index = o.wartosc(objetoscp, 0);

baza f = **new** baza();

**double** cenaPaczki;

cenaPaczki = f.CenazaPaczke(0, index);

f.closeConnection();

**Cena pojedynczego zamówienia:**

**double** cenaZam;

CenaZam = o.CenaOstateczna(waga2, objetoscp, cenaPaczki, odleglosc, cena, ilosc2,rab2);

**double** finalValue = Math.*round*( cenaZam \* 100.0 ) / 100.0;

**Wyświetlanie i liczenie ceny całkowitej:**

cenacalko = Math.*round*( (cenacalko + finalValue) \* 100.0 ) / 100.0;

**double** cenarab = cenacalko;

cenacalk = **new** JLabel("Cena całkowita: " + cenacalko + " zł" );

cenacalk.setBounds(300, 350, 300, 20);

cenacalk.setFont(**new** Font("SansSerif", Font.***BOLD***, 12));

add(cenacalk);

cenacalk.setVisible(**true**);

**Wstawianie do listy i do bazy Zamowienia:**

Zamowienie zamowienie = **new** Zamowienie(kod, ilosc2, objetoscp, waga2, miasto, miasto2, data ,"W trakcie",kod,finalValue);

zamowienia.add(zamowienie);

baza a = **new** baza();

a.insertZamowienie(ilosc2, objetoscp, waga2, miasto, miasto2,data,"W trakcie",kod,finalValue);

a.closeConnection();

**Wyświetlanie okienka błędu:**

JOptionPane.*showMessageDialog*(**new** JFrame(), "Błąd wprowadź własciwe dane", "Error" ,JOptionPane.***ERROR\_MESSAGE***);

**Obliczanie nowej ceny zamówienia po kliknięciu przycisku „ANULUJ”**

baza rr = **new** baza();

**double** cenaAn = Math.*round*( (rr.CenaAnuluj(0, kodPaczki2)\*0.3) \* 100.0 ) / 100.0;

rr.closeConnection();

Cena jest pobierana na podstawie wpisanego kodu zamówienia. Nowa cena jest równa 30% ceny pobranej, procent ten jest wstawiony na stałe a całość jest zaokrąglona do drugiego miejsca po przecinku.

Dla każdego wyświetlenia wpisów jest deklarowana nowa lista, która jest wyświetlana i do której jest kopiowane pożądane przez użytkownika dane.

Ze względu na liczne błędy występujące podczas pisania kodu musieliśmy każdy przycisk zrobić osobno, tak samo jak listy(AWT-0 Event Queue) . Za każdym razem trzeba ustanawiać nowe połączenie z bazą danych żeby uniknąć błędu SQL Busy.

# Różnice w projektach:

W toku pisania programu doszliśmy do wniosku, że przydałoby się trochę ulepszyć program kolegów ze względu na niedociągnięcia oraz błędy. Wprowadziliśmy następujące zmiany:

1. Dodaliśmy przyciski „**COFNIJ”** ponieważ gdy użytkownik nie zna kodu paczki nie jest w stanie dalej korzystać z programu bez jego wyłączenia. W tej chwili można wchodzić w każdą funkcję programu bez obawy, że się gdzieś utknie
2. Dodaliśmy wyświetlanie całej bazy Zamówienia, w pierwotnym projekcie ten pomysł nie występował.
3. Dodaliśmy opcję usuwania pojedynczych wpisów na podstawie kodu paczki. Opcja ta znajduje się od razu po wyświetleniu całej bazy Zamówienia.
4. Baza Gabaryty została zmieniona. Wcześniej posiadała kolumny od każdego wymiaru paczki. Uznaliśmy, że lepszym rozwiązanie jest stworzenie jednej kolumny zawierającej objętość paczki którą i tak trzeba by było policzyć.
5. Dodaliśmy warunki na poprawne wpisywanie danych aby uniknąć błędów programu. Każde pole jest odpowiednio zabezpieczone.
6. Pole z rabatem dodaliśmy do formularza aby użytkownik sam mógł je uzupełnić. Maksymalną wartość ustawiliśmy warunkiem na 20%
7. Do okna „SPRAWDŹ PRZESYŁKĘ” dodaliśmy dwa przyciski zmieniające status przesyłki na „Zakończono” lub „Anulowane”. W przypadku tego drugiego, przesyłce jest nadawana nowa cena w wysokości 30% starej ceny, ponieważ może zdarzyć się sytuacja, w której paczka znajduje się już w drodze. Jest to zatem forma zwrotu za ewentualny transport lub kara za anulowanie zamówienia.

# Spis ilustracji:

[Rysunek 1: Ekran tytułowy 3](#_Toc485070987)

[Rysunek 2: Formularz nadania przesyłki 4](#_Toc485070988)

[Rysunek 3: Okienko błędu 4](#_Toc485070989)

[Rysunek 4:Okno podsumowania zamówień 5](#_Toc485070990)

[Rysunek 5:Okno sprawdzania przesyłki 6](#_Toc485070991)

[Rysunek 6: Wyszukany wpis 6](#_Toc485070992)

[Rysunek 7:Okno z całą bazą zamówień 7](#_Toc485070993)

[Rysunek 8: Drzewo konstrukcji programu 8](#_Toc485070994)