JAVA Project

Wykonawca: Roman Uliashyn

Temat: „Admin System”

**Spis Treści**

[Cel: 3](#_Toc30542714)

[Funkcjonalności: 3](#_Toc30542715)

[Śriodowisko programowe oraz ustawienia: 5](#_Toc30542716)

[Dodaktowe szczegóły: 6](#_Toc30542717)

[Klasy a ich wykorzystanie 19](#_Toc30542718)

[Przykład Działania 26](#_Toc30542719)

[Widok Katalogu 38](#_Toc30542720)

# **Cel:**

Utworzyć aplikacje, która pozwoli administratoram różnego rodzaju zarządzać dannymi studentów uczelni.

# **Funkcjonalności:**

*“Zarządzanie dannymi ”*

Logowanie do systemu   
3 typa userów - (Główny Admin, Zwykły a Księgowy )  
Każdy ma swoje prawa na wykorzystanie aplikacji.  
  
*Administrator Główny*

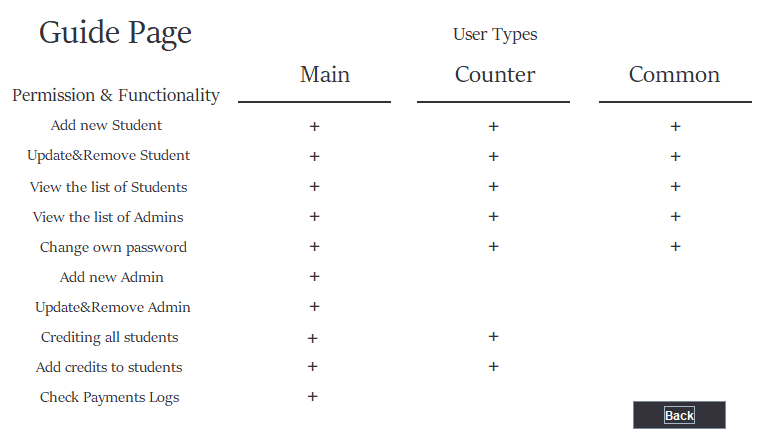
Ma uprawnienia na wszystko:  
1. Dodawać Studenta lub administratora do systemu   
2. Robić aktualizacyjne danych studenta lub innych administratorów  
3. Obejrzeć listę studentów i administratorów   
4. Ma uprawnienia nad operacjiamii płatności (Odejmować ratę u wszystkich raz w miesiącu) i dodawać konkretnemu studentowi pieniedzie opłacone offline

*Administrator Zwykły*   
Ma uprawnienia:  
1. Dodawać tylko studenta   
2. Aktualizować dane studenta lub usuwać jego   
3. Oglądać listę studentów a administratorów

*Administrator Księgowy*

Ma te same uprawnienia co i zwykły ale + do tego, może działać nad operacjami płatności   
W okienku z lista studentów, na panele są dodatkowę inputy i przyciski

**Możliwości wykorzystania**



# **Śriodowisko programowe oraz ustawienia:**

1. Wykorzystane było środiowisko NetBeans 8.0.2
2. Project SDK 1.8 , java\_version 1.8.0\_231
3. Plik koćnow w formacie .jar „Admin\_System”
4. Program uruchamiamy przez XXAMP Server Apache

**Wykorzystanie baz danych:**

* SQL zapytania
* Zarządzanie dannymi na localhostcie phpmyadmin
* UWAGA. Żeby system zadziawał, trzeba podłączyć Bazę student.sql do *localhost/do phpMyAdmin/dbImport*. Aby dostać się do bazy, trzeba w katalogu głównym Admin\_System przejść do ./database/student.sql
* W kierunku ./database/student.sql znajduje się DOCUMENT student.pdf, można obejrzyć widok testowych danych.

**Biblioteki wykorzystane:**

- Dla połączenia

* import java.sql.Connection;
* import java.sql.DriverManager;
* import javax.swing.JOptionPane;

- Wypisywanie danych do Tabeli

* import net.proteanit.sql.DbUtils;

-Dla formatu daty

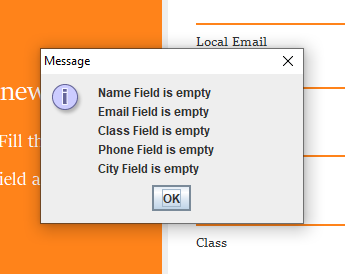
* import java.text.DateFormat;
* import java.text.SimpleDateFormat;
* import java.util.Calendar;
* import java.util.Date;

**Dodaktowe szczegóły:**

1. Opracowanie inputow

Pomysłem jest, wywołanie funkcji Catch\_Err, która tworzy StringBuilder a nadpisuje go za każdym razem, złapionym wyjątkiem, od tworzonego fabryką validatora. Typ validatora jest tworzony odpowiednio do przyszłego typu objekta.

* 1. Czy field pusty?



*Wywołując kod w addStudent:*

try{

Catch\_Err();

....

*Funkcja w swojej kolejności sprawdza:*

StringBuilder errorMessage = new StringBuilder();

//Name

try {

IValidation nameValidation = ValidatorsFactory.make(ValidationType.NAME\_VALIDATOR);

nameValidation.Validate(txtName.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

}

//Email

try {

IValidation emailValidation = ValidatorsFactory.make(ValidationType.EMAIL\_VALIDATOR);

emailValidation.Validate(txtLocalEmail.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

}

......

if (errorMessage.length() != 0) {

throw new IllegalArgumentException(errorMessage.toString());

}

}

*nameValidation.Validate(txtName.getText());*

Zmienna ta jest typu IValidation, a jej zawartośc jest twrorzony odpowiednego typu Validator, przez fabrykę.

Po wywołaniu .Validate(zmienna\_do\_tesowania) – validator sprawdza go w taki sposób

public void Validate(Object data) {

try{

String name = (String) data;//Check on string

if(**name.equals("")**){

throw new IllegalArgumentException("Name Field is empty");

}

...

}catch(ClassCastException e){

throw new IllegalArgumentException("Name is not of type string");

}

}

Potem zwraca wyjątek w razie błedu, a Catch\_Err(){

..

...

///Jeśli zmienna errorMessage była zmodyfikowana, to ->>

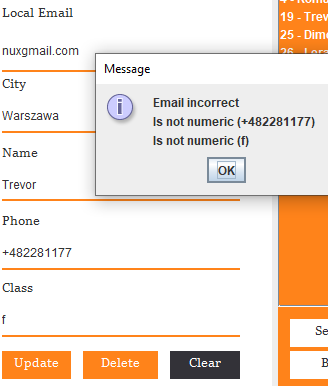
if (errorMessage.length() != 0) {

throw new IllegalArgumentException(errorMessage.toString());

}

}

* 1. Typ wprowadzonych danych odpowiedni?



Tak samo, jeśli nasz input nie spełnia reguł, wyjątek będzię złapiony:

*Wywołując kod w Student\_Operations:*

try{

**Catch\_Err();**

....

*Funkcja w swoją kolejności sprawdza:*

StringBuilder errorMessage = new StringBuilder();

//Phone

try {

IValidation phoneValidation = ValidatorsFactory.make(ValidationType.PHONE\_VALIDATOR);

phoneValidation.Validate(txtPhone.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

} //Email

try {

IValidation emailValidation = ValidatorsFactory.make(ValidationType.EMAIL\_VALIDATOR);

emailValidation.Validate(txtLocalEmail.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

}

......

if (errorMessage.length() != 0) {

throw new IllegalArgumentException(errorMessage.toString());

}

}

*zmienna.Validate(txtName.getText());*

Zmienna ta jest typu IValidation, a jej zawartośc jest twrorzony odpowiednego typu Validator, przez fabrykę.

Po wywołaniu .Validate(zmienna\_do\_tesowania) – validator sprawdza go w podobny sposób, ale:

Zmienna typu **EMAIL** ma dodatkową regułę:

private static final String **EMAIL\_PATTERN** =

"^[\_A-Za-z0-9-\\+]+(\\.[\_A-Za-z0-9-]+)\*@"

+ "[A-Za-z0-9-]+([\\.[A-Za-z0-9]+)\*(\\.[A-Za-z]{2,})$](file:///\\.[A-Za-z0-9]+)*(\\.%5bA-Za-z%5d%7b2,%7d)$)";

public void Validate(Object data) {

try {

String email = (String) data;

.....

if(!email.matches(**EMAIL\_PATTERN**)){

throw new IllegalArgumentException("Email incorrect");

}

} catch (ClassCastException e) {

throw new IllegalArgumentException("Email is not of type string");

}

}

Zmienne typu **PHONE,CLAS,ID** ma dodatkową regułę:

private Pattern **pattern** = Pattern.compile("-?\\d+(\\.\\d+)?");

public boolean **isNumeric**(String strNum) {

if (strNum == null) {

return false;

}

return **pattern**.matcher(strNum).matches();

}

public void Validate(Object data) {

try{

String phone = (String) data;//check on string

....

if(!**isNumeric**(phone)){

throw new IllegalArgumentException("Phone is not numeric");

}

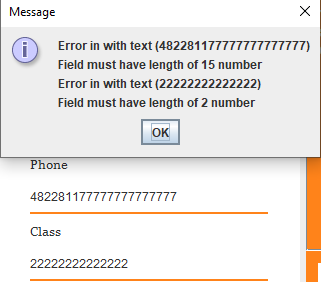
.....

}catch(ClassCastException e){

throw new IllegalArgumentException("Phone is not of type string");

}

}

* 1. długość wprowadzonego odpowiednia regułam?

*Wywołując kod w Student\_Operations:*

try{

**Catch\_Err();**

....

*Funkcja w swoją kolejności sprawdza:*

StringBuilder errorMessage = new StringBuilder();

//Phone

try {

IValidation phoneValidation = ValidatorsFactory.make(ValidationType.PHONE\_VALIDATOR);

phoneValidation.Validate(txtPhone.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

} //Email

try {

IValidation emailValidation = ValidatorsFactory.make(ValidationType.EMAIL\_VALIDATOR);

emailValidation.Validate(txtLocalEmail.getText());

}catch(IllegalArgumentException e) {

errorMessage.append(e.getMessage() + '\n');

}

......

if (errorMessage.length() != 0) {

throw new IllegalArgumentException(errorMessage.toString());

}

}

*zmienna.Validate(txtName.getText());*

Zmienna ta jest typu IValidation, a jej zawartośc jest twrorzony odpowiednego typu Validator, przez fabrykę.

Po wywołaniu .Validate(zmienna\_do\_tesowania) – validator sprawdza go w podobny sposób, a konkretnie długość:

@Override

public void Validate(Object data) {

try{

String phone = (String) data;//check on string

....

if(**phone.length**() > 15){

throw new IllegalArgumentException("Phone is too long");

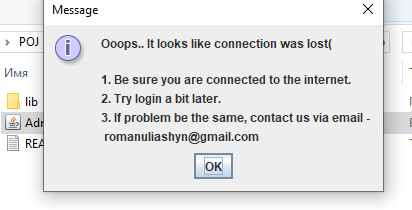
}

}catch(ClassCastException e){

throw new IllegalArgumentException("Phone is not of type string");

}

}

1. Aplikacja nie puszcza jeśli nie ma połączenia

Po uruchameniu, aplikacja sprawdza,czy jest podłączenie do MySQL.

public static Connection connection(){

try{

Class.forName(JDBC\_DRIVER);

Connection conn;

conn = DriverManager.getConnection(DB\_URL,USER,PASS);

return conn;

}catch(Exception ex){

JOptionPane.showMessageDialog(null,"Ooops.. It looks like connection was lost(\n\n"

+ "1. Be sure you are connected to the internet.\n2. Try login a bit later.\n"

+ "3. If problem be the same, contact us via email - \n romanuliashyn@gmail.com");

System.exit(0);

return null;

}

}

3. Aplikacja zakazuje dostęp i usuwa przycisku jeśli Admin nie ma uprawnienia

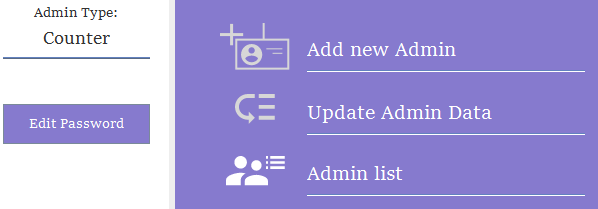
*Wywołując kod w HOME:*

if(!err.**isMainHere**()){

System.out.println("Need to hide");

btnAddAdmin.setEnabled(false);

btnAdminOperation.setEnabled(false);

}

Dynamicznie zakazuje korzystanie lub odczytanie informacji według reguł opisanych w możliwościach wykorzystania.

*Tą funkcje obsługuje kod w Error\_Hadling:*

public boolean isMainHere(){

if(current\_user.getStatus().toLowerCase().equals("main")){

return true;

}else{

return false;

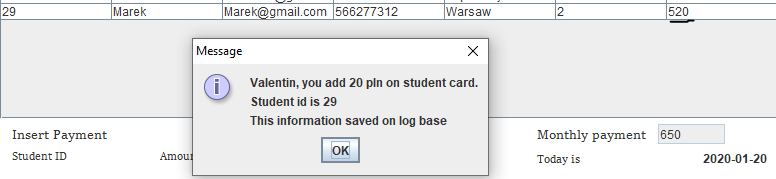
}

}

4. Dodawać pieniedzę do karty studenta można nieskończono liczbę razy:

*Wywołując kod:*

qr.**Update\_Credit**();

qr.update\_query(qr.GetSQL\_Statement());

*Tą funkcje obsługuje kod:*

public void **Update\_Credit(){**

System.out.println(getDb\_name());

Payment payment = Payment.getInstance();

this.SQL\_Statement = "UPDATE "+getDb\_name()+" SET credit = credit + "+Integer.toString(payment.getAmount())+" WHERE id = '"+user.getId()+"'";

}

Odejmować można tylko raz w miesięcu. Sprawdza się w bazie danych. Wypisuje bląd jeśli Główny admin lub Księgowy admin probóje wziąć kolejną ratę:

*Wywołując kod:*

if(qr.**Insert\_Month\_Log**()){

*W swojej koleji, wywołuje funkcje Exist\_Log:*

public boolean Insert\_Month\_Log(){

if(!**Exist\_Log**()){

this.SQL\_Statement = "INSERT INTO month\_payment (date,counter\_name) VALUES ('"+Payment.getInstance().getLogDate()+"','"+CurrentUser.getInstance().getName()+"')";

System.out.println("Set insert");

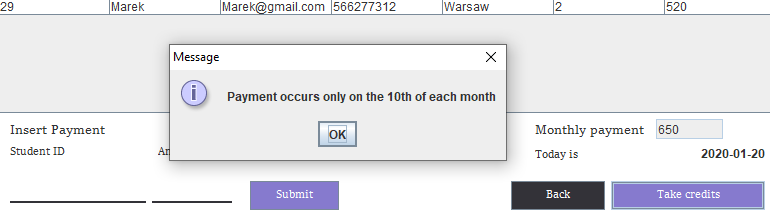
return true;

}else{

return false;

}

}



*Tą funkcje obsługuje kod:*

*public boolean* ***Exist\_Log****(){*

*......*

*try{*

*withdraw\_day = new SimpleDateFormat("yyyy-MM-dd").parse(Payment.getInstance().getLogDate());*

*if(dayFormat.format(withdraw\_day).equals("10")){*

*....*

*stmt = conn.createStatement();*

*this.SQL\_Statement = "SELECT \* FROM month\_payment WHERE date='"+Payment.getInstance().getLogDate()+"'";*

*rs = stmt.executeQuery(SQL\_Statement);*

*if(rs.next()){*

*System.out.println(rs.getString("date"));*

*return true;*

*}else{*

*return false;*

*}*

*}else{*

*JOptionPane.showMessageDialog(null, "Payment occurs only on the 10th of each month\n");*

*return true;*

*}*

*......*

5. Operacji nad płatnościami są zapisywane wa bazie „Logi” działań)

*Wywołując kod:*

if(qr\_logs**.Insert\_Logs**(Integer.toString(qr.user.getId()))){

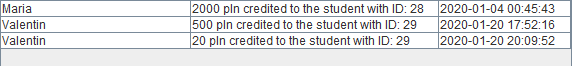
qr\_logs.update\_query(qr\_logs.get\_SQL\_Statement());

JOptionPane.showMessageDialog(null, CurrentUser.getInstance().getName() + ", you add "

+ ""+payment.getAmount()+" pln on student card.\nStudent id is "+qr.user.getId()+"\n"

+ "This information saved on log base");

}



*Tą funkcje obsługuje kod:*

public boolean **Insert\_Logs**(String ID){

boolean isError=true;

if(CurrentUser.getInstance().getName().equals("")||Payment.getInstance().getDate().equals("")){

isError = false;

}

else{

this.SQL\_Statement = "INSERT INTO "+getDb\_name()+" (admin\_name,transfer\_info,date) "

+ "VALUES('"+CurrentUser.getInstance().getName()+"','"

+Integer.toString(Payment.getInstance().getAmount())+" pln credited to the student with ID: "+ID+"','"+Payment.getInstance().getDate()+"')";

}

return isError;

}

6. Jest mozliwosc przeglądu historji opłat studentów

Dzięki eventListener`u , za każdym razem jak jest zmieniona wartość w ComboBoksie, wyłowana jest funkcja **SQL\_STUDENT\_LOGS**

*Wywołując kod:*

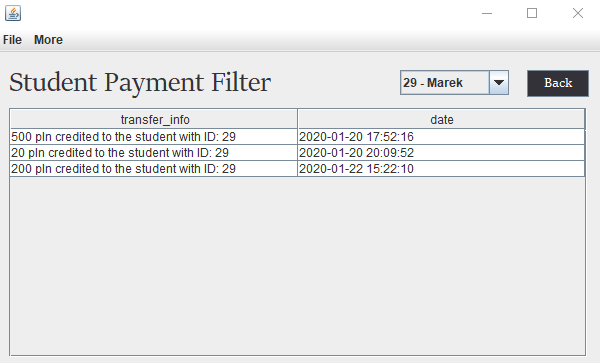
if(!filterBox.getSelectedItem().equals("\*")){

user = new User.UserBuilder().id(item.substring(0,2)).build();

qr.**SQL\_STUDENT\_LOGS**(user);

}

....



*Tą funkcje obsługuje kod:*

public void **SQL\_STUDENT\_LOGS**(User user){

try{

stmt = conn.createStatement();

System.out.println(user.getId());

//expretion,poczatek , dlugosc pobrania

this.SQL\_Statement = "SELECT transfer\_info,date FROM "+getDb\_name()+" WHERE SUBSTRING(transfer\_info, LENGTH(transfer\_info)-1,2) = '"+user.getId()+"'";

rs = stmt.executeQuery(SQL\_Statement);

student\_logs.tbPayment.setModel(DbUtils.resultSetToTableModel(rs));

.....

7. Nowy administrator może zmienić swoje hasło po pierwszym logowaniu

Program też sprawdza, czy aktualne hasło było wprowadzone, aby zmienić na nowe. *Wywołując kod:*

if(err**.EqualPasswords**(CurrentUser.getInstance().getPassword(), txtCurrentPassword)){

CurrentUser.getInstance().setPassword(txtNewPassword.getText());

qr = new Query();

qr.setDb\_name("admin");

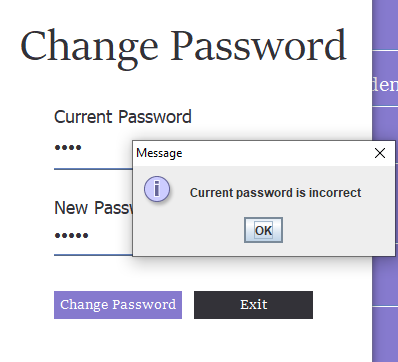
qr.Update\_Password();

qr.update\_query(qr.GetSQL\_Statement());

JOptionPane.showMessageDialog(null, "Password was successfully updated");

Clear\_Fields();

}

*Tą funkcje obsługuje kod:*

public boolean **EqualPasswords**(String prev,JTextField next){

boolean stm;

if(prev.equals(next.getText())){

stm = true;

}else{

err\_list.add("Current password is incorrect\n");

stm = false;

}

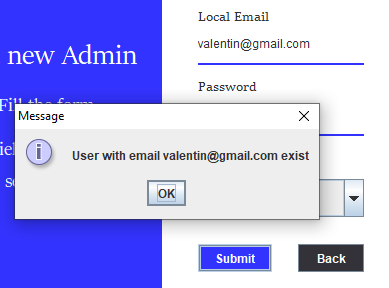
return stm;

}

8. Nie można zrobic użytkownika z podobnym emailem

if(!err.**Exist\_Email**(txtLocalEmail,qry.getDb\_name())){

err.Print\_Errors();

}

Tą funkcjonalnośc obsługuje kod:

boolean **Exist\_Email**(JTextField s,String db) {

qr.setDb\_name(db);

if(!qr.emailExist(s.getText())){

err\_list.add("User with email "+s.getText()+" exist");

return false;

}else{

return true;

}

}

# **Klasy a ich wykorzystanie**

Opracowanie Błędów

* Error\_Handling.java

Interface

* Interface Ivalidation.java

Validators

* IdValidator.java
* NameValidator.java
* ... *Cała lista validatorów jest wypisana na ostaniej kartce.*

Enumerator

* ValidationType.java

Fabric Pattern

* ValidatorFactory.java

Singleton Patterny

* CurrentUser.java
* Payment.java

Builder Pattern

* User

Set & Get Class

* Next\_Statement

SQL Zapytania

* Query.java

DB Połączenie

* databaseConnection.java

Opracowanie Błędów

* Error\_Handling.java

*Funkcje w klasie:*

* isMainHere – boolean funkcja, sprawdza z Singelton klasy status usera.(Admin główny?)
* isCounter - boolean funkcja, sprawdza z Singelton klasy status usera. (Admin księgowy?)
* EqualPasswords – boolean funkcja, porówna hasła. (Prev z Next)
* Check\_Permission – bollean funkcja, sprawdza w Admin\_Operation Klasie, czy mbiezący użytkownik ma poszwolenie na zmiane danych innych adminów
* Print\_Errors – void funkcja, która wypisuje listę dodanych błędów.
* Exist\_Email – sprawdza czy istneje użytkownik o takim samym Emailu, w konkretnej bazie.

Opracowanie Błędów *Validators package*

Klasy w Validators:

* Ivalidation – Klasa szłyży jako interface dla dalszego wykorzystania w odzielnych validatorów.
* IdValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* NameValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar)*
* EmailValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,czy jest poprawnie napisany email,rozmiar)*
* PwdValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* PhoneValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* CityValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar)*
* ClassValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* NewPasswordValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* AmtValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* DebtValidator – klasa która dziedziczy Interface *Ivalidation* oraz sprawdza uzyskany objekt na prawidwołość. *(Typ,czy jest pusty,rozmiar,czy jest numeric)*
* ValidationType – Enumerator. Stworzony aby dynamicznie oznaczać typ, w momencie tworzenia nowego obiektu w fabryce.
* ValidatorsFactory – tworzy odzielnie, w zależności od podanego typu validator i zwraca go.
* Logika łapenia błędów – wykorzystanie mecahnizmu try,catch, tworzenie nowego stringu z opisem błedu i tworzenie wyjątków typu *ClassCastException* a *IllegalArgumentException* .
  + *ClassCastException* wykorzystany, kiedy typ otrzymanego objektu jest nie odpowiedni do podanego.
  + *IllegalArgumentException* wykorzystany, dla tworzeniu błędu jak objekt ma złe argumenty.
* Każdy validator sprawdza różnych rodzajów objektów.
* Dla sprawdzenia czy jest objekt jest liczbą wykorzystujemy wnętrzny *Pattern.compile("-?\\d+(*[*\\.\\d+)*](file:///\\.\\d+))*?")*
* Dla sprawdzenia prawidlowości napisania emailu, wykorzystywany jest StringPattern *EMAIL\_PATTERN = "^[\_A-Za-z0-9-\\+]+(*[*\\.[\_A-Za-z0-9-]+)\*@*](file:///\\.[_A-Za-z0-9-]+)*@)*" + "[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)\*(\\.[A-Za-z]{2,})$";*

Singleton Patterny

* CurrentUser.java – pozwala ustawić dane(Imię,Email,Status,Hasło) a późnej wykorzystać ich gzie kolwiek w systemie. Korzystujemy w momencie *logowania*, aby ustawić dane wprowadzone, a późnej np. pobieramy w celach (*zapisania Logu lub ustawiania nowego hasła*)
* Payment.java - pozwala ustawić dane(Kwota), inicjalizować dane(Data Płatności) a późnej wykorzystać ich gzie kolwiek w systemie. Korzystujemy w momencie, gdy dodajemy wpłacone pieniądze do uczelni. Napoczątek robimy setAmount, a potem pobieramy dane w celach (*Zapisania Logów,Sprawdzania Logów,Dodania nowej kwoty do konta użytkownika lub Obniżania wszystkich użytkowników raz w miesiącu*)

Builder Patterny

* User – klasa która pozwala dynamicnie tworzyć użytkownika w różnych miejscach oraz nie obowiązkowo korzystać ustawiania wszystkich danych konstruktora.

Np. Kiedy poszukujemy użytkownika, to tworzymy jego w Query.java według bazy danych.

Jeśli baza danych „student”, to utworzony jest użytkownik o takich dannych:

if(rs.next()){

user = new User.UserBuilder()

.id(id)

.name(rs.getString("name"))

.email(rs.getString("email"))

.city(rs.getString("city"))

.clas(rs.getInt("class"))

.credit(rs.getInt("credit"))

.phone(rs.getString("phone"))

.build();

Lub kiedy baza danych „admin”, tworzony jest użytkownik o danych:

if(rs.next()){

user = new User.UserBuilder()

.id(id)

.name(rs.getString("name"))

.email(rs.getString("email"))

.status(rs.getString("admin\_status"))

.build();

Set & Get Class

* Next\_Statement – klasa pomocnicza, korzystowana jest w celu porównania danych w momencie aktualizacji ich do bazy danych. Stworzona była klasa metodą kapsulacji konstruktora.

Np. w po nacisku Update danych studenta:

if(Catch\_Err()){

err.Print\_Errors();

}else{

//Setter

**nxt**.setId(txtID.getText());

**nxt**.setName(txtName.getText());

**nxt**.setPhone(txtPhone.getText());

**nxt**.setClas(txtClass.getText());

**nxt**.setEmail(txtLocalEmail.getText());

**nxt**.setCity(txtCity.getText());

//execute\_insert

if (qry.SetSQL\_Update(**nxt**)) {

qry.update\_query(qry.GetSQL\_Statement());

Clear\_Fields();

JOptionPane.showMessageDialog(null, "Data sucessfully updated");

} else {

txtID.setText(Integer.toString(qry.user.getId()));

JOptionPane.showMessageDialog(null, "You can`t change the ID");

}

}

I późnej, w funkcji SetSQL\_Update „Klasa Query” – porównujemy czy zostali dane zmienone. Jeśli tak, to budujemy dynamiczne zapytanie aby zmienić odrazu kilka danych w tabeli.

...

if(GetById(Integer.toString(nxt.getId()))){

this.SQL\_Statement = "UPDATE "+getDb\_name()+" SET id = '"+user.getId()+"'";

if(!Prev\_Next(user.getId(),nxt.getId())){

return false;

}

if(!Prev\_Next(user.getName(),nxt.getName())){

this.SQL\_Statement += ", name = '"+nxt.getName()+"'";

}

.....

this.SQL\_Statement +=" WHERE id ='"+user.getId()+"'";

SQL Zapytania

* Query.java

-W klasie są trzy konstruktory które używają inne klasy aby dostać się do tabeli a wypisać wszystko do niej

-Jest także konstruktor który przyjmuje objekt typu „User”, dla korzystania zewnętrznych danych

*Funkcje typu SET:* - ustawiają zmienną SQL\_Statement

* setDb\_name – ustawianie nazwy bazy dannych
* Prev\_Next – porównowanie zmiennych
* SQL\_Login – ustawia zapytanie oraz zapisuje do Singleton klasy CurrentUser dane.
* Select\_All\_FromDB – w zależności jaka baza danych jest ustawiona, pobiera dane i wypisuje odpowiednio do konkretnej tabeli.
* GetById – poszukuje użytkownika o wpowadzonym indeksie oraz tworzy go w Systemie.
* emailExist – Sprawdza czy Email wpisany już istnieje lub nie.
* Insert\_Student – dodaje do tabeli stworzonego użytkownika tupy student
* Insert\_Admin - dodaje do tabeli stworzonego użytkownika tupy admin
* Insert\_Logs – zapisuje akcje administratora, który dodał kwotę na konto
* Insert\_Month\_Log – zapisuje akjce administratora który pobrał miesięczną kwote.
* Exist\_Log – sprawdza czy można pobrać miesięczną kwote i czy jest ona wykonana wcześnej.
* Set\_Select\_All – zwykłe zapytanie (wybrać wszystkich)
* SQL\_Delete\_Student – usuwa użytkownika w wpowadzonej tabeli
* SetSQL\_Update – tworzy dynamiczne zapytanie według typu użytkownika a ilości zmienionych danych
* Update\_Credit - tworzy zapytanie które dodaje wprowadzoną kwote do istniejącej
* Update\_Credit\_Month\_Debt - tworzy zapytanie które odejmuje wprowadzoną kwotę u wszystkich użytkowników
* Update\_Password – tworzy zapytanie które pozwoli zaktualizować hasło

*Funkcje tupu GET:* - Pobieranie zmiennej SQL\_Statement

* get\_SQL\_Statement – zwraca Zapytanie w formacie String
* getDb\_name – zwraca nazwę Bazy Danych w formacie String

*Funkcje typu EXECUTE:* - Wykonuje zmienną SQL\_Statement

* executeQuery - wykona zapytanie oraz zapisuje wszystkie wierszy do wprowadząnej listy. Wykorzystane dlatego, aby wypisać istniejących użytkowników w tabeli (USER/ADMIN Operations)
* update\_Query – wykona aktualizacje zapytania

DB Połączenie

* databaseConnection.java

Ta klasa pozwala połączyć się z local-sewerem w tym projekcie. Jeśli niestety nie udało się połączyć się z bazą danych a serwerem, wypisany jest MessageBox z szczegółami.

*Zmienne startowe :*

final static String JDBC\_DRIVER = "com.mysql.cj.jdbc.Driver";

final static String DB\_URL = "jdbc:mysql://localhost:3306/student";

final static String USER = "root";

final static String PASS = "";

*Funkcja która włączy:*

public static Connection **connection**(){

try{

Class.forName(**JDBC\_DRIVER**);

Connection conn;

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

return conn;

}catch(Exception ex){

JOptionPane.showMessageDialog(null,"*Ooops.. It looks like connection was lost(\n\n"*

*+ "1. Be sure you are connected to the internet.\n2. Try login a bit later.\n"*

*+ "3. If problem be the same, contact us via email - \n romanuliashyn@gmail.com*");

System.exit(0);

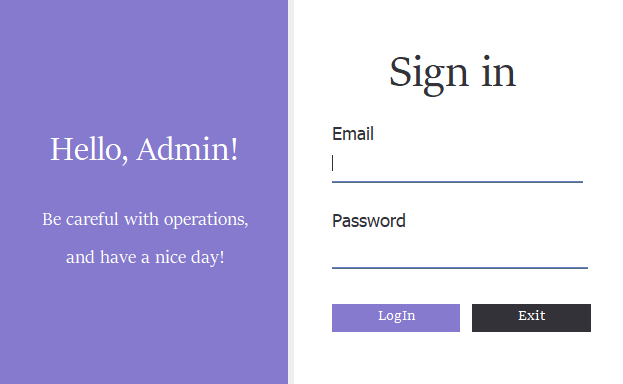
return null;

}

}

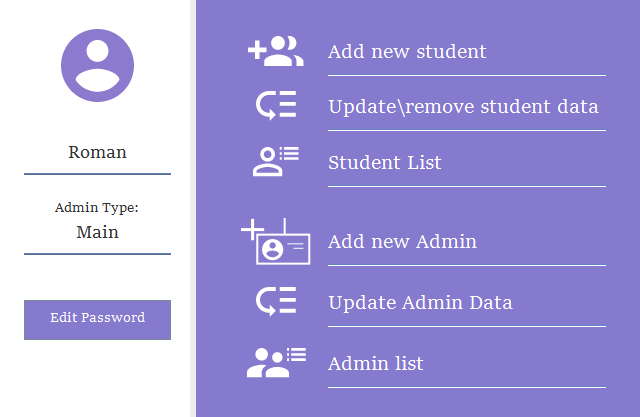
# **Przykład Działania**

Po uruchamieniu Admin\_System.jar, jeśli jest dostęp do Serwera i MySQL – uruchomi się aplikacja:



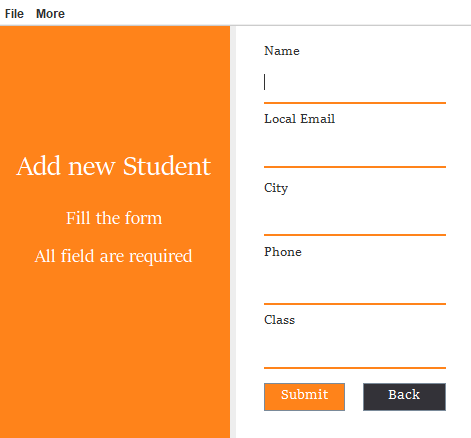
Oczywiste jest logowanie do Systemu, więc w proponuje wejść na głównego administratora login:[roman@gmail.com](mailto:roman@gmail.com) , hasło:Admin1

Po logowaniu widzimy okienko z głównym menu:



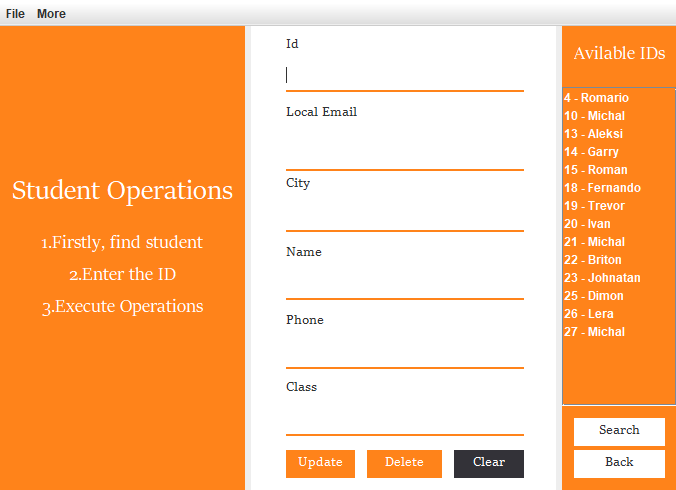
UWAGA: Typ Administratora jest Main i dlatego ma on prawa na wszystkie funkcjonalności.

Klikając na dodać Studenta, otwiera się okienko:



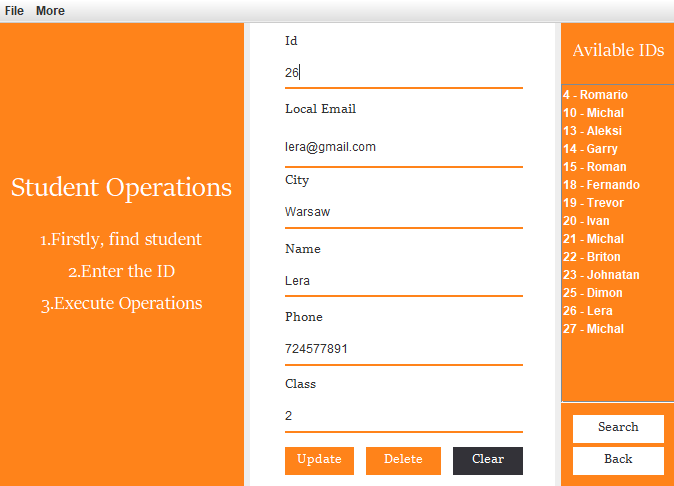
Tutaj możemy dodać nowego użytkownika typu student, będzię on odrazu dodany do tabeli.

Klikając na operacje z dannymi Studenta, otwiera się okienko:



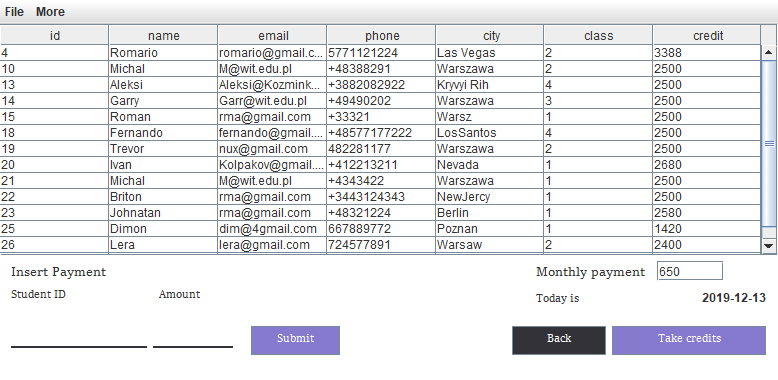
Tutaj najpierw aby zrobić coś, trzeba wybrać istiejącego użytkownika.

Wpisujemy ID poznane zewnątz lub wybierając z listy z prawa:



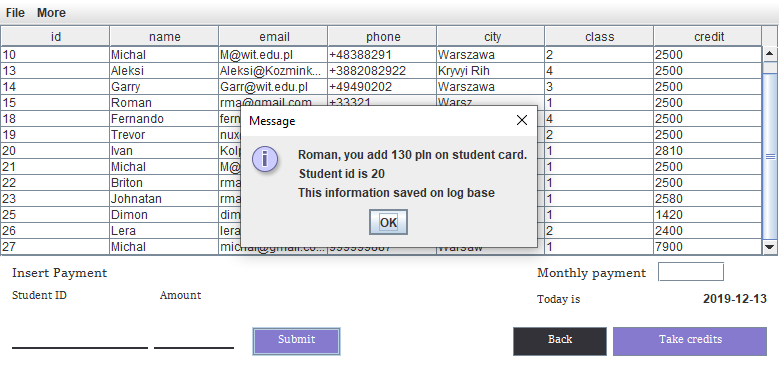
Jeśli pola odpowiadają regułam, można usunąć użytkownika lub zaktualizować danne

Klikając na listę Studentów, otwiera się okienko:

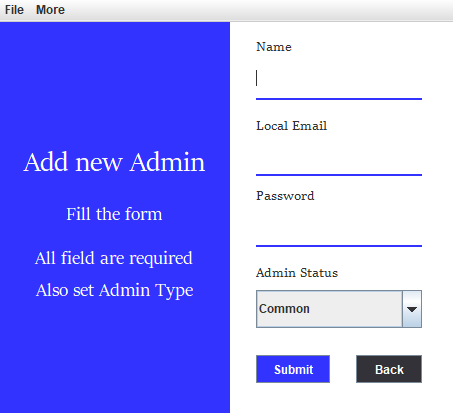


Tutaj administrator typu „Main”(Główny) i „Counter”(księgowy) mają możliwość działać z opłatami. *Reguły i funkcjonalności opisane wcześnie.*

Jeśli chcemy dodać pieniądze do konkretnego użytkownika, to trzeba wpisać jego ID oraz kwote do dodania. Widzimy więc to:

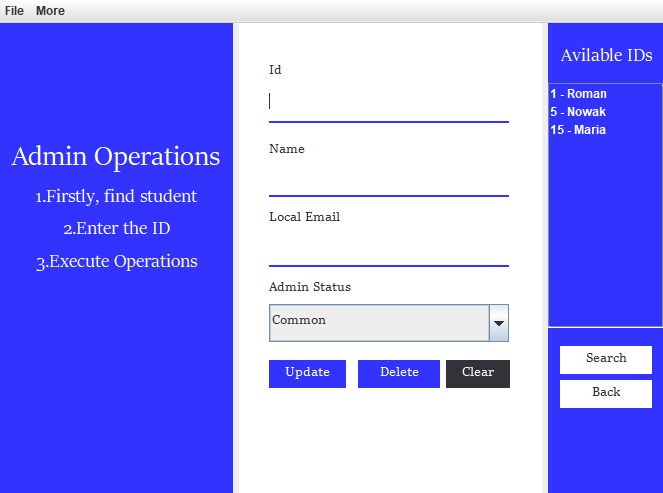


Klikając na dodać Administratora, otwiera się okienko:



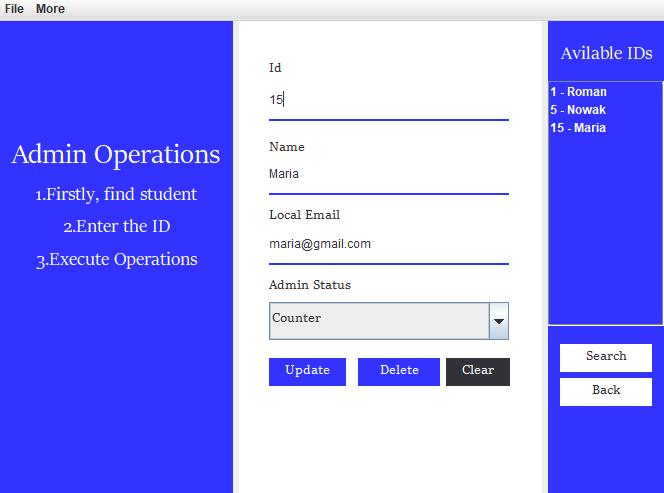
UWAGA: *Otwiera się to okienko według reguł opisanych wyżej.*

Klikając na operacje z dannymi Administratora, otwiera:



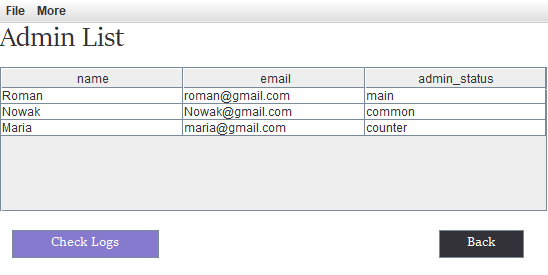
Tutaj najpierw aby zrobić coś, trzeba wybrać istiejącego użytkownika.

Wpisujemy ID poznane zewnątz lub wybierając z listy z prawa:

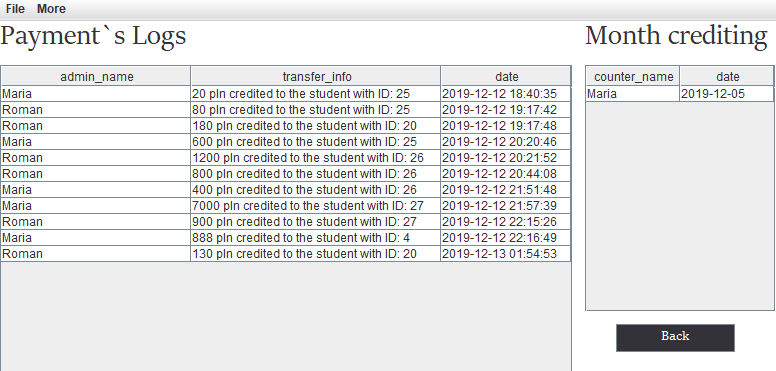


Jeśli pola odpowiadają regułam, można usunąć użytkownika lub zaktualizować danne.

Klikając na listę Administratorów, otwiera się:

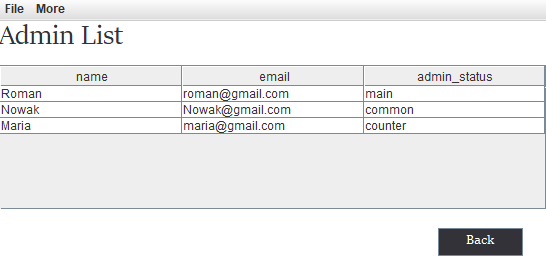


Klikając poniżej na Check Logs (Sprawdż aktywność), otwiera się:

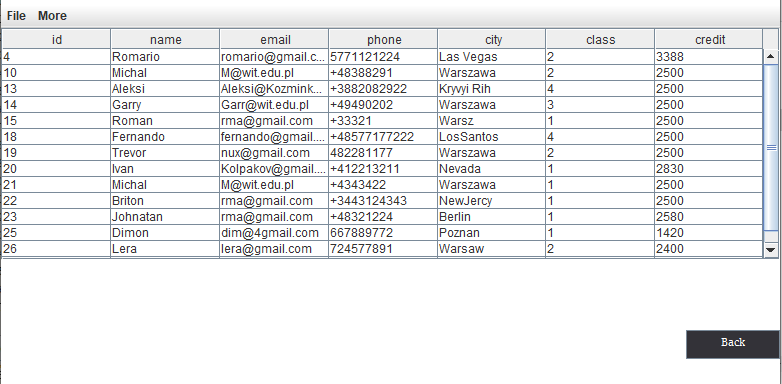


Tutaj głowny Administrator monitoruje aktywność opłat.

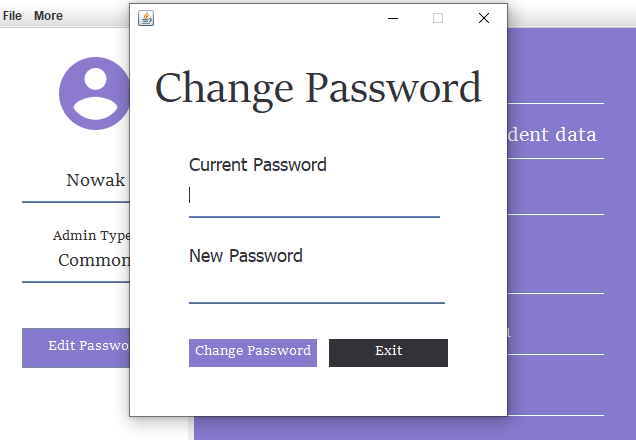
Tak wygląda okienko jesli typ administratora nie Main (Główny)

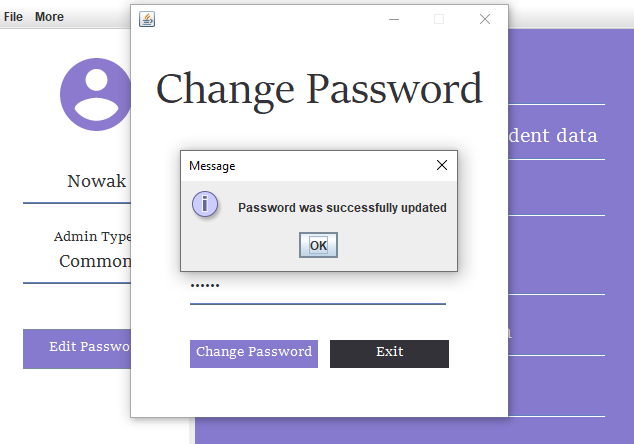


Tak wygląda okienko Listy Studentów, jeśli typ zapytającego administratora Common (Zwykły):



Klikając na edytowanie Hasła otwiera się okienko:



Jeśli wprowadzone dane opdowiadają regułam wpisanych danych, dostejemy takie okienko:

# **Widok Katalogu**

./

**forms:**

* Login.java
* Home.java
* addAdmin.java
* addStudent.java
* Student\_Operations.java
* Admin\_Operations.java
* showStudent.java
* showAdmins.java
* Logs.java
* Edit\_Password.java
* About.java
* GuidePage.java
* Error\_Handling.java

**utils:**

* СurrentUser.java
* Payment.java
* User.java
* Query.java
* Next\_Statement.java
* databaseConnection.java

**validators:**

* IValidation.java
* IdValidator.java
* NameValidator.java
* PhoneValidator.java
* CityValidator.java
* ClassValidator.java
* PwdValidator.java
* NewPasswordValidator.java
* AmtValidator.java
* DebtValidator.java
* EmailValidator.java
* ValidationType.java
* ValidatorsFactory.java