Help & Manual

Manual - eLibrary

Struktura projektu

eLibrary - struktura projektu

- elibrary-core

  Obsługa warstwy prezentacyjnej (view) oraz kontrolera (controller).

- elibrary-hibernate

  Obsługa warstwy danych oraz dostępu do nich (model).

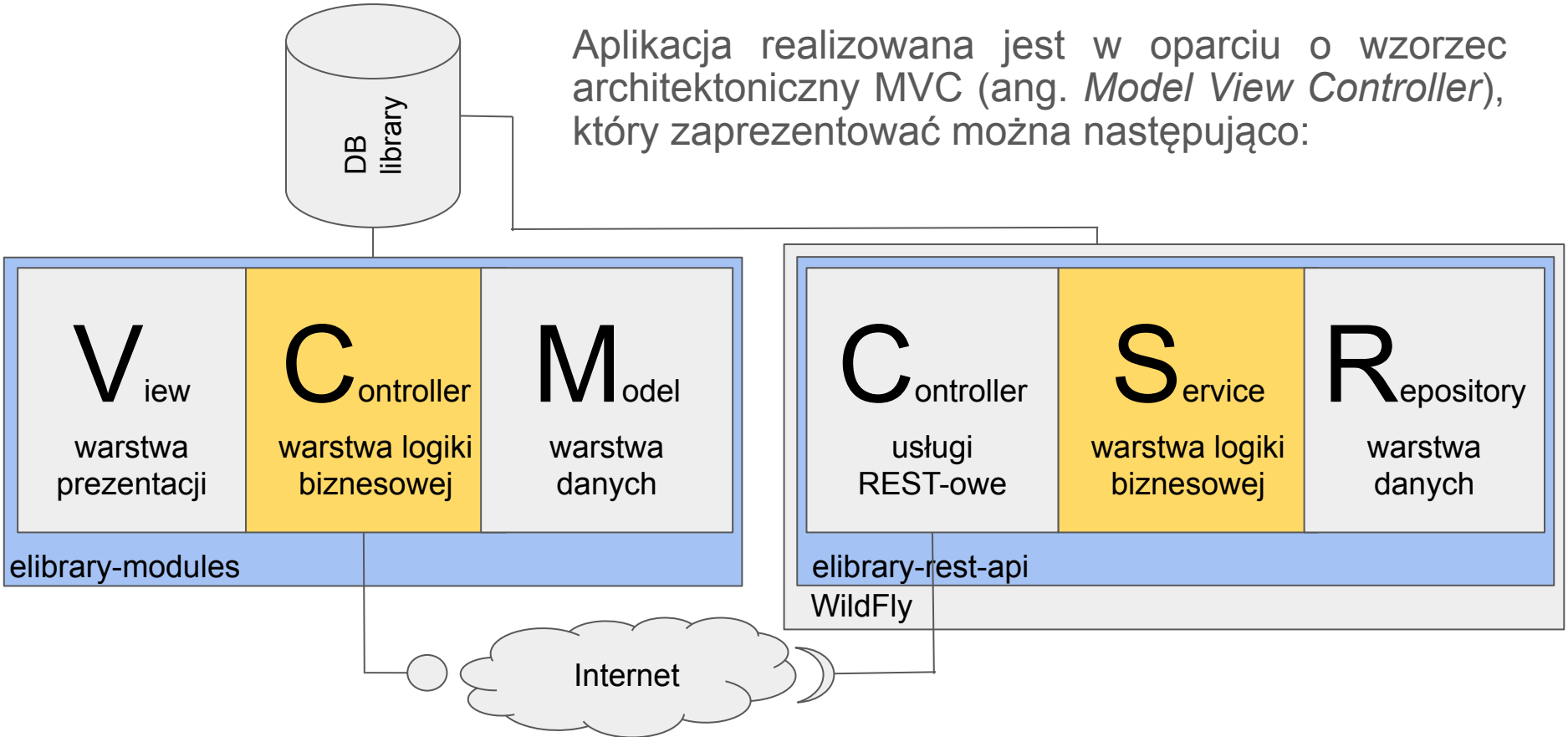
- elibrary-rest-api

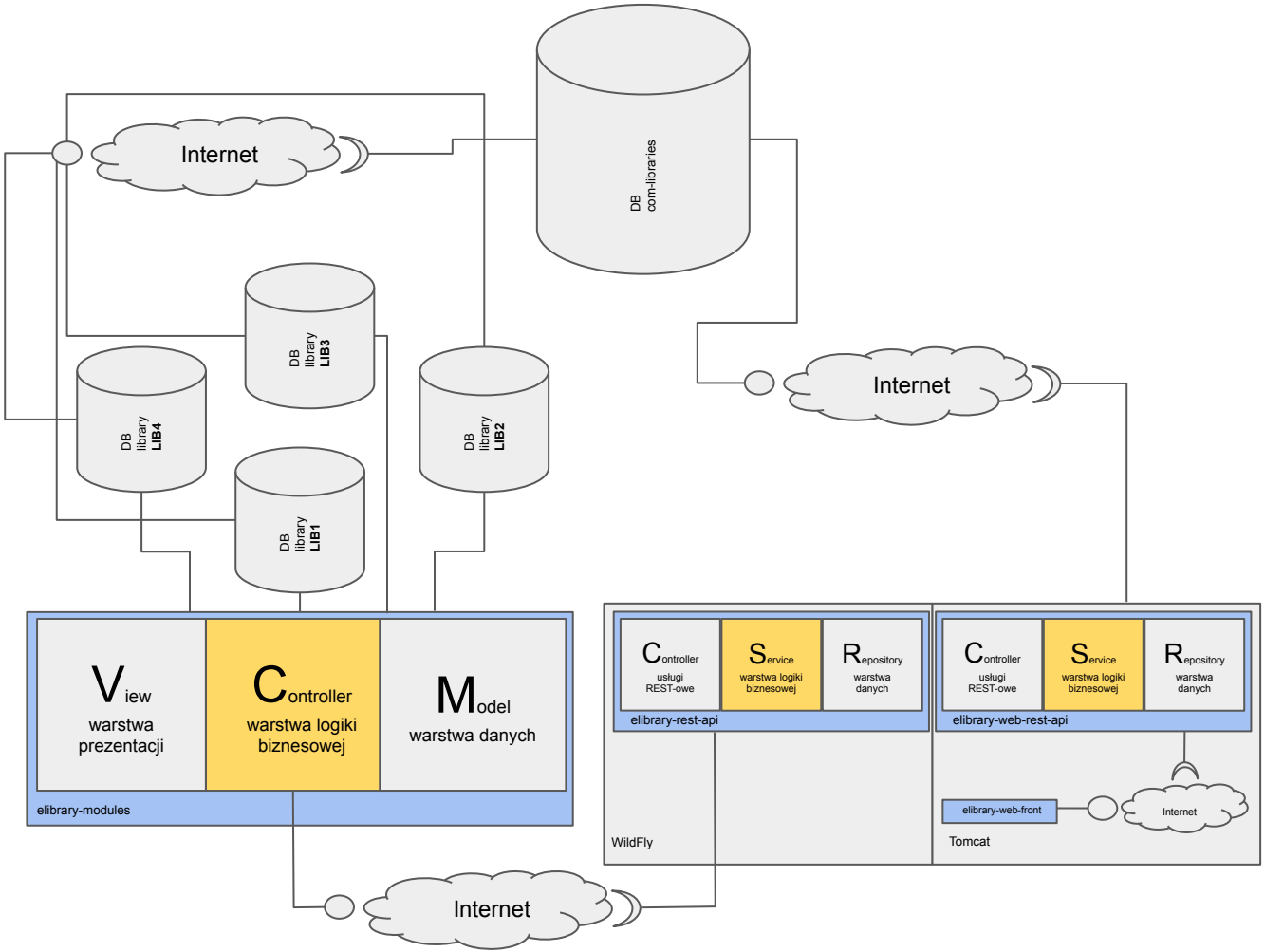
  Moduł wystawiający usługi dostępne poprzez sieć Internet (API).

Architektura systemu

Architektura systemu – główny koncept

Aplikacja realizowana jest w oparciu o wzorzec architektoniczny MVC (ang. *Model View Controller*), który zaprezentować można następująco:




Internet
DB
com-libraries
DB
library
LIB3
DB
library
LIB4
DB
library
LIB2
DB
library
LIB1
Internet
View
warstwa
prezentacji
Controller
warstwa logiki
biznesowej
Model
warstwa danych
elibrary-modules
Controller
usługi
REST-owe
Service
warstwa logiki
biznesowej
Repository
warstwa
danych
elibrary-rest-api
WildFly
Controller
usługi
REST-owe
Service
warstwa logiki
biznesowej
Repository
warstwa
danych
elibrary-web-rest-api
elibrary-web-front
Internet
Tomcat
Internet

Model

Model - warstwa danych

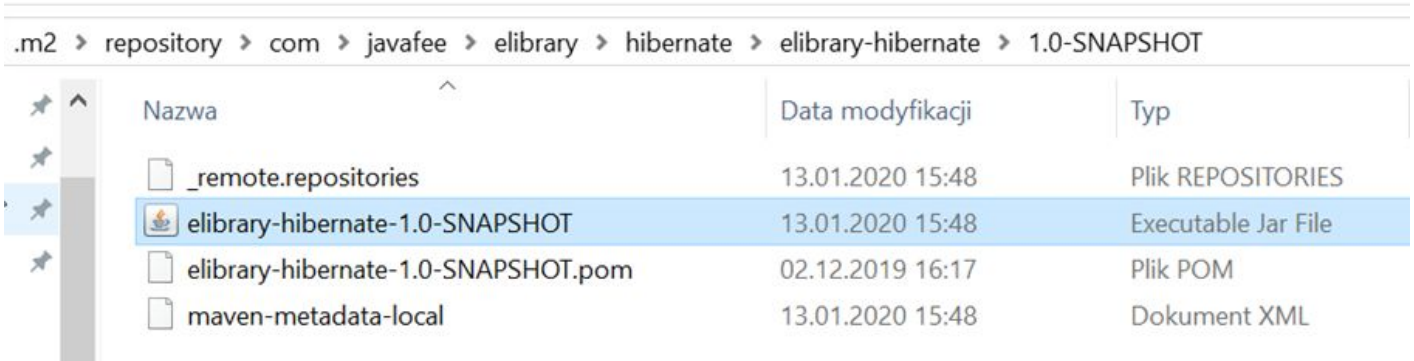
Warstwa danych obejmuje obiekty **DTO** (ang. *Data Transfer Object*) zwanych inaczej obiektami domenowymi (ang. *Domain Object*) lub z j. ang. Entities mapujące tabele w bazie danych na obiekty aplikacji.

Ponadto warstwa danych obsługuje metody dostępowe do bazy danych (CRUD), które reprezentowane są przez klasy **DAO** (ang. *Data Access Object*).

Warstwa modelu implementowana jest w ramach osobnego modułu (aplikacji). Moduł ten nazwany został elibrary-hibernate i stanowi aplikację, która z użyciem Mavena powinna znajdować się w lokalnym repozytorium Mavena (m2) i stanowi zależność dla projektu elibrary-core. Przy realizacji funkcjonalności skorzystano z frameworku Hibernate.

Model - warstwa danych - elibrary-hibernate jako zależność do elibrary-core

```
<dependency>
    <groupId>com.javafee.elibrary.hibernate</groupId>
    <artifactId>elibrary-hibernate</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
<dependency>
```



Model - Obiekty domenowe - struktura

Klasy domenowe mapowania obiektowo - relacyjnego implementowane są z wykorzystaniem standardu JPA - framework'u Hibernate. Struktura klas domenowych jest następująca i została przedstawiona na następnym slajdzie:

- Nad nazwą klasy znajdują się adnotacje framework'u lombok oraz standardu JPA. Do pierwszych należą @Data i @EqualsAndHashCode. Do drugich natomiast **@Entity**, @NamedQueries/@NamedQuery, @Table, @SequenceGenrator. W ramach danej klasy znajdują się jeszcze inne adnotacje standardu JPA i są nimi m.in.: @Id, @GeneratedValue, **@Column** oraz te związane ze związkami pomiędzy encjami np. @ManyToOne, @ManyToMany, @JoinColumn etc. Znaczenie poszczególnych adnotacji zostanie omówione w ramach następnych slajdów. Aby utworzyć najprostszą klasę domenową należałoby użyć wytłuszczonych adnotacji;
- Klasy domenowe to w dużym uproszczeniu proste obiekty POJO, które nie zawierają logiki biznesowej. Ich struktura ogranicza się jedynie do odpowiednich adnotacji i właściwości oraz w niektórych przypadkach przeładowań metod klasy Object/String tj. equals(), hashCode(), toString().

Model - Obiekty domenowe - struktura

Volume.java

```java
@Data
@EqualsAndHashCode(exclude = "lend")
@Entity
@NamedQueries({
        @NamedQuery(name = "Volume.checkIfInventoryNumberExist", query = "from Volume where inventoryNumber = :inventoryNumber")})
@Table(name = "lib_volume", uniqueConstraints = {@UniqueConstraint(columnNames = {"inventory_number"})})
@SequenceGenerator(name = "seq_lib_volume", sequenceName = "seq_lib_volume", allocationSize = 1)
public class Volume implements Cloneable {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_lib_volume")
    @Column(name = "id_volume", unique = false, nullable = false, insertable = true, updatable = true)
    private Integer idVolume;

    @Column(name = "inventory_number", unique = false, nullable = true, insertable = true, updatable = true, length = 13)
    private String inventoryNumber;

    @Column(name = "state", unique = false, nullable = true, insertable = true, updatable = true, length = 1)
    private Character state;

    @Column(name = "is_reading_room", unique = false, nullable = true, insertable = true, updatable = true)
    private Boolean isReadingRoom = false;

    @Column(name = "is_reserve", unique = false, nullable = true, insertable = true, updatable = true)
    private Boolean isReserve = false;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_book")
    private Book book;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "volume",
```

Model - Obiekty domenowe - adnotacje Lombok

Jak zostało wspomniane na slajdzie “Model - Obiekty domenowe - struktura” adnotacje @Data oraz @EqualsAndHashCode pochodzą z framework’u Lombok.

- @Data - dzięki tej adnotacji, nie ma potrzeby implementacji metod dostępowych do właściwości danej klasy domenowej - metody setter’y i getter’y są generowane w sposób automatyczny. Ponadto dana entity’ka ma wygenerowane odpowiednio metody toString(), equals() i hashCode();
- @EqualsAndHashCode - jednak, czasem automatyczne generowanie metod klasy Object/String nie jest wskazane. Wówczas, gdy chce się wykluczyć pewne właściwości dla celów generowania ww. metod należy, przekazać je jako parametr ‘exclude’ w adnotacji np.:

```
@EqualsAndHashCode(exclude = "lend")
```

  Dla powyższego przykładu podczas generowania metody klas Object i String, właściwość ‘lend’ zostanie pominięta.

Więcej informacji o adnotacjach z Lombok można znaleźć pod adresem https://projectlombok.org/features/all

Model - Obiekty domenowe - adnotacje Hibernate/JPA

W klasach domenowych, oprócz adnotacji Lombok, znajdują się adnotacje JPA. Są one kluczowe dla realizacji mapowania relacyjno-obiektowego. Najważniejszymi są:

- @Entity - przy pomocy adnotacji @Entity określa się docelową tabelę mapowania tzn. domyślnie, gdy nie zdefiniuje się w inny sposób nazwy tabeli na jaką ma być zmapowany dany obiekt, nazwa tabeli będzie równa nazwie danego obiektu. Nazwa tabeli może być jednak inna i aby ten cel osiągnąć należy użyć adnotacji @Table(name = “table_name”) z określonym parametrem ‘name’ jako nazwa docelowej tabeli;
- @Column - podobna sytuacja występuje w przypadku adnotacji @Column, która podczas mapowania dostarcza informację nt. tego jakiej kolumnie po stronie bazy danych odpowiada dana właściwość w klasie. Domyślnie, nazwa kolumny jest identyczna z nazwą właściwości w klasie, jednak poprzez użycie parametru ‘name’ adnotacji, można wymusić specyficzną nazwę dla kolumny np.:

```
@Column(name = "document_number", unique = false, nullable = true, insertable = true, updatable = true, length = 20)
private String documentNumber;
```

W powyższym przykładzie właściwości ‘documentNumber’ odpowiadać będzie kolumna o nazwie ‘document_number’. W dalszej części zostaną opisane inne parametry dla adnotacji @Entity i @Column.

Model - Komunikacja z BD - DAO

Warstwa DAO (ang. *Data Access Objects*) służy do komunikacji z bazą danych. Do podstawowych operacji należą (opisane wcześniej i określane akronimem CRUD): *create*, *read*, *update*, *delete*. Operacje te wykonywać można na wiele sposobów:

1. Poprzez obiekty Dao - wykorzystujące klasę HibernateUtil;
2. Poprzez klasę HibernateUtil;
3. Poprzez natywne zapytania - createNativeQuery;
4. Poprzez predefiniowane zapytania @NamedQuery.

Model - Komunikacja z BD - DAO - obiekty Dao

Struktura obiektów DAO została przedstawiona na następnym slajdzie. Poprzez obiekty DAO można realizować różne operacje na bazie danych. W klasie tej bowiem znajdują się różne metody. Należą do nich m.in.: save(), saveOrUpdate(), findAll(), update(), delete(). Dzięki ww. metodom możemy zrealizować każdą operację opisywaną przez skrót CRUD. Obiekt HibernateDao jest obiektem generycznym i związany jest z każdą klasą domenową. To znaczy - dla każdej Entity'ki można utworzyć powiązany z nią obiekt DAO. Realizowane jest to poprzez mechanizm generyczności i zaobserwować go można analizując nagłówek klasy:

```
public class HibernateDao<T, Id extends Serializable> implements GenericDao<T, Id>
```

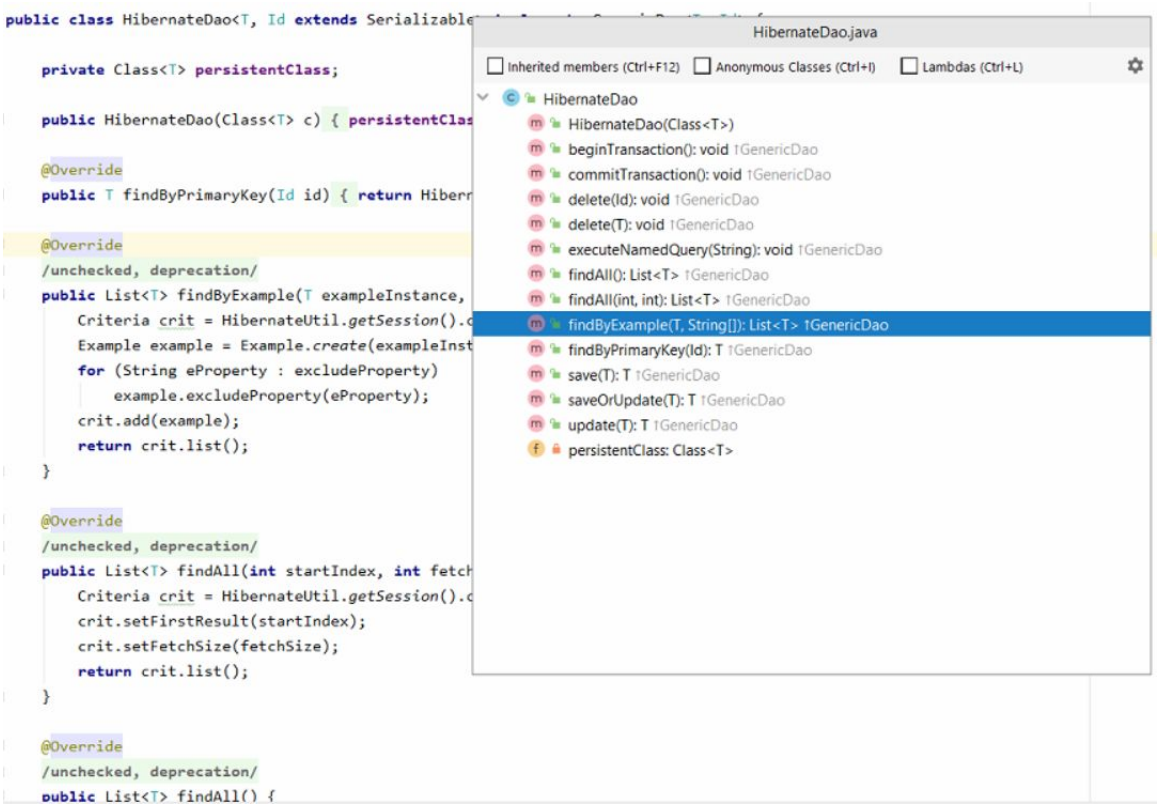
Obiektem typu T, może być każdy obiekt DTO. Poniżej zamieszczono przykład użycia obiektu DAO.

```
this.clientDao = new HibernateDao<Client, Integer>(Client.class);
this.clients = clientDao.findAll();
```

I podobnie:

```
List<MessageType> messageTypeList = new HibernateDao<>(MessageType.class).findAll();
```

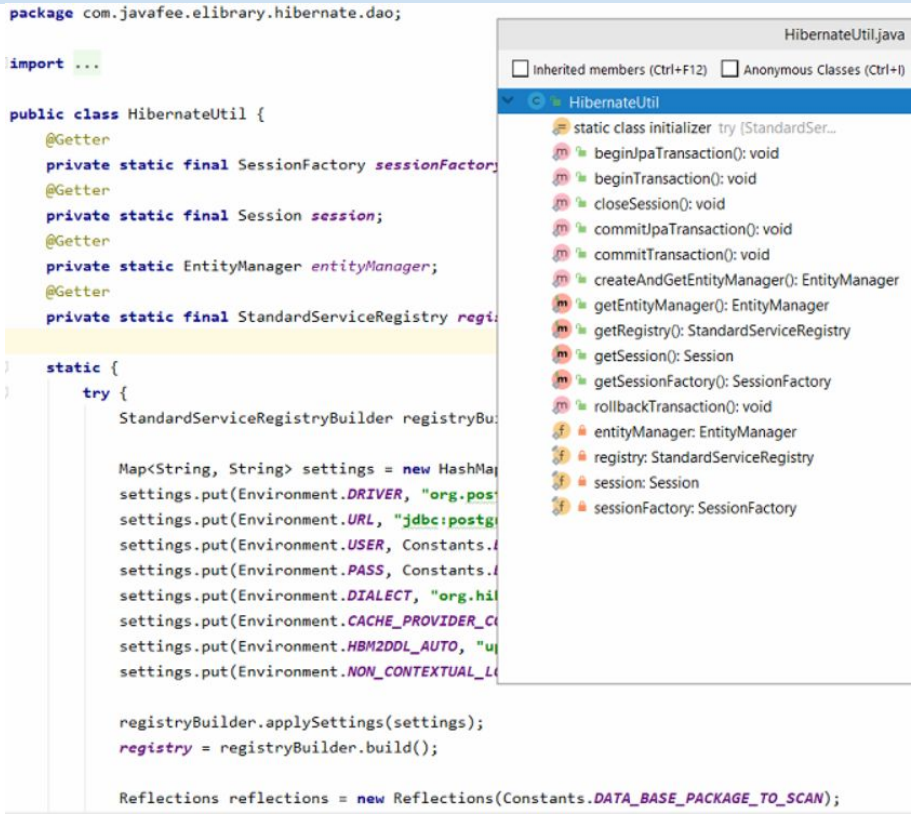
Model - Komunikacja z BD - DAO - obiekty DAO



Model - Komunikacja z BD - DAO - HibernateUtil

W projekcie eLibrary zaimplementowana została również klasa HibernateUtil. Stanowi ona klasę użyteczności, która daje szereg możliwości związanych z obsługą bazy danych oraz zawiera metody służące do inicjalizacji połączenia z nią. Klasa ta wykorzystywana jest do wykonywania operacji w klasie Dao. Operacje CRUD i inne, dostępne w opisywanej klasie realizowane są za pośrednictwem obiektów sesyjnych (ang. *Session*) tj. aby je wykonać należy najpierw wywołać metodę getSession() - obiekty sesyjne i transakcji zostaną opisane na następnych slajdach.

Metody opisywanej klasy przedstawiono na zrzucie obok.



Model - Komunikacja z BD - DAO - HibernateUtil

W celu wykonania operacji zapisu (save) na bazie danych należy posłużyć się następującym kodem:

```
HibernateUtil.beginTransaction();
Volume volume = new Volume();
volume.setBook(selectedBook);
volume.setInventoryNumber(inventoryNumber);
if (loanOrReadingRoom == Context.READING_ROOM)
    volume.setIsReadingRoom(true);

HibernateUtil.getSession().save(volume);
HibernateUtil.commitTransaction();
```

Na zrzucie ekranu obok, zaprezentowano sposób zapisu danych - obiektu domenowego Volume - do bazy danych. Operacja na bazie danych zakończona może być zasadniczo na dwa sposoby: zapis - *commit*, lub odrzucenie zmian - *rollback*. Metody te - commit i rollback są metodami obiektu transakcji (jest on dostępny w klasie HibernateUtil - beginTransaction()). Wewnątrz transakcji można realizować zmiany na obiektach domenowych, co jeszcze nie ma swojego bezpośredniego przełożenia na bazę danych. Zmiana w bazie danych następuje dopiero po ‘zatwierdzeniu’ zrealizowanych zmian - *commitTransaction()*, lub ich odrzuceniu - *rollbackTransaction()*. Wywołanie tych dwóch metod powoduje realizację właściwej zmiany w BD.

Natomiast, operacje które nie są jeszcze zatwierdzone, wykonywane są w obrębie obiektu sesyjnego (session) - *getSeession()*. Operacją taką może być na przykład save, delete etc. Jednak, samo wywołanie HibernateUtil.getSession().seve(volume) nie powoduje realnego zapisu obiektu DTO w bazie danych. Dzieje się to dopiero po wykonaniu metody commitTransaction(). Podsumowując zatem każda operacja na bazie danych powinna być objęta transakcję i wykonywana przez sesję, transakcja powinna być otwarta przez jak najkrótszy czas.

```
HibernateUtil.beginTransaction();
HibernateUtil.getSession().update(UserData.class.getName(), LogInEvent.getUserData());
HibernateUtil.commitTransaction();
```

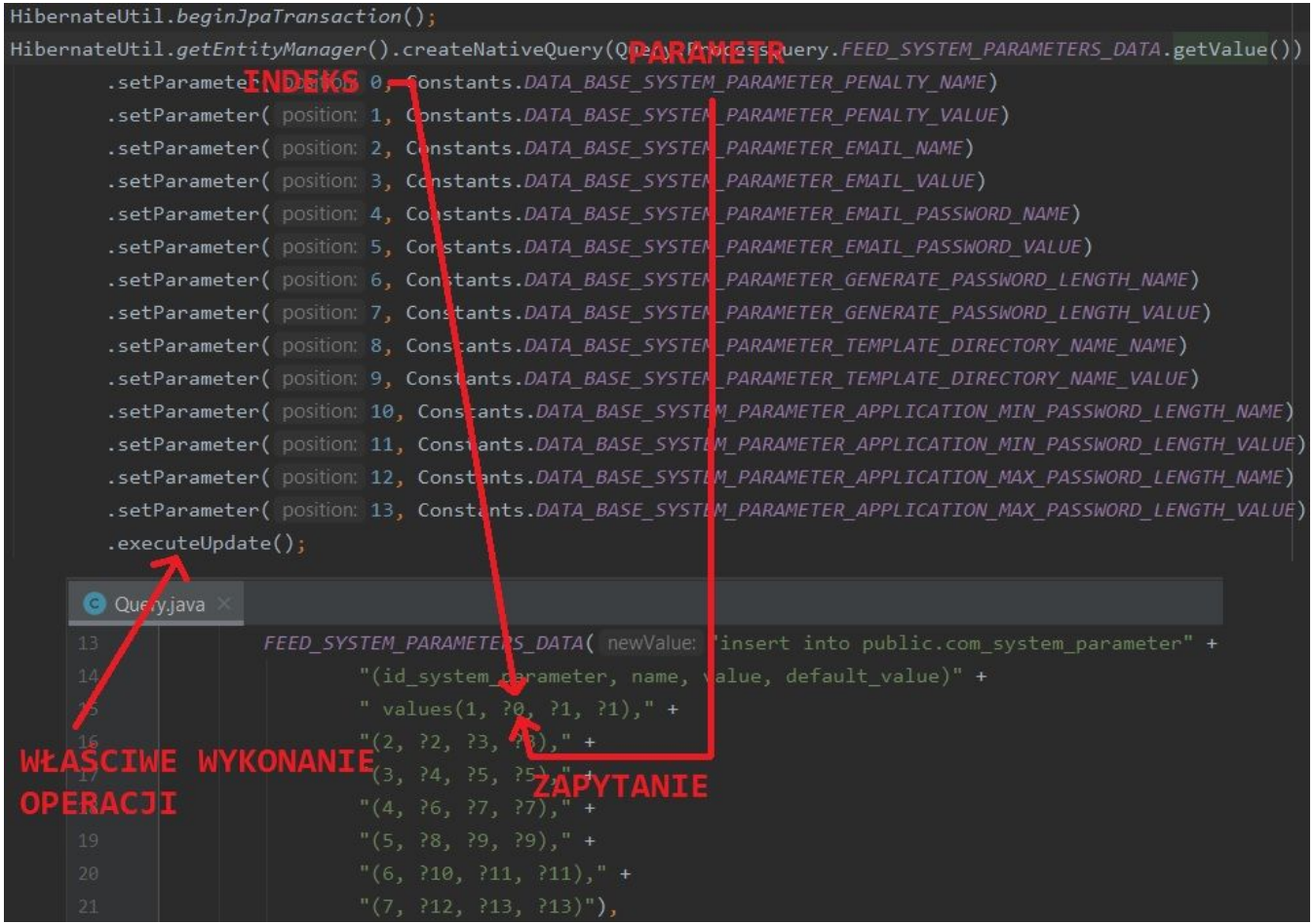
```
HibernateUtil.beginTransaction();
HibernateUtil.getSession().delete(selectedMessage);
HibernateUtil.commitTransaction();
```

Model - Komunikacja z BD - DAO - natywne zapytania

Kolejnym sposobem komunikacji z bazą danych jest wykonywanie natywnych (bezpośrednich) zapytań. Odbywa się to m.in. przy pomocy metody ‘createNativeQuery’ klasy ‘EntityMenager’. Wywołać ją można z poziomu klasy ‘HibernateUtil’.

```
HibernateUtil.getEntityManager().createNativeQuery(Query.ProcessQuery.FEED_SYSTEM_PARAMETERS_DATA.getValue())
```

Parametrem metody ‘createNativeQuery’ jest natywne zapytanie w języku SQL przekazane w postaci obiektu ‘String’. W projekcie eLibrary dostępna jest klasa ‘Query’ wewnątrz której definiowane są zapytania w ramach enum’ów np. ‘ProcessQuery’. Zapytania mogą być parametryzowane. Wówczas parametry zapisywane są w postaci ‘?0’, gdzie wartość 0 to indeks parametru. Następnie, dzięki użyciu metody ‘setParameter’, parametry na podstawie wartości indeksów wstawiane są w odpowiednie miejsca w zapytaniu przekazanym w metodzie ‘createNativeQuery’. Istotnym faktem jest to, że użycie opisywanej metody wymaga użycia transakcji. Jest ona dostępna po wywołaniu metody ‘beginJpaTransaction’. Przykład użycia metody ‘createNativeQuery’ wraz z komentarzem dotyczącym klasy ‘Query’ znajduje się na następnym slajdzie.



Użycie metody ‘createNativeQuery’

Model - Komunikacja z BD - DAO - @NamedQuery

Następną metodą dostępu do danych są zapytania @NamedQuery. Zapytania @NamedQuery są wczytywane, walidowane w trakcie kompilacji programu dlatego też, zgodnie z dobrymi praktykami programowania powinno się je stosować w sytuacjach gdy dane zapytanie wykorzystywane jest w wielu miejscach w systemie (względu wydajnościowe). W systemie eLibrary definiowane są one nad klasami domenowymi. Zapytanie @NamedQuery definiowane jest przy pomocy adnotacji @NamedQuery lub, jeśli jest ich więcej niż jedno - otoczone dodatkowo adnotacją @NamedQueries. W celu zdefiniowania zapytania należy określić następujące właściwości: **‘name’** (unikalna nazwa zapytania, która będzie używana w celu wykonania go w kodzie, zgodnie z konwencją określa się ją w następujący sposób *NazwaEntity.nazwaZapytania*) oraz ‘query’ (treść zapytania w języku HQL - w przeciwieństwie do SQL’a zamiast używać nazw tabel, używane są nazwy klas domenowych i/lub nazwy kolumn zdefiniowane w klasie Entity (niekoniecznie nazwa ta jest tożsama z nazwami kolumn w bazie danej)). Aby wykonać zapytanie @NamedQuery należy użyć metody obiektu sesji - ‘getNamedQuery’, do której jako parametr, przekazuje się nazwę zapytania (tożsama z parametrem **‘name’**). Jeżeli zapytanie ma zdefiniowane parametry, wówczas ustawia się je przy pomocy metody ‘setParameter’, do której przekazuje się jako pierwszy argument nazwę parametru (zgodną z tą zdefiniowaną w klasie domenowej, wg. schematu *:parameter*) oraz wartość parametru jako następny argument. Aby pobrać wyniki zwrócone przez wywoływane zapytanie należy wywołać jedną z następujących metod:

- ‘uniqueResult’ - jeśli zapytanie zwraca tylko jedną, unikalną wartość - metoda zwraca Object (typ generyczny <R>);
- ‘getResultList’ - jeśli zapytanie może zwrócić więcej niż jedną wartość - metoda zwraca obiekt typu List.

```java
@Data
@Entity
@NamedQueries({
        @NamedQuery(name = "Reservation.checkIfVolumeActiveReservationExists", query = "from Reservation where id_volume = :idVolume " +
                "and is_active is not null and is_active = true and (is_cancelled is null or (is_cancelled is not null and is_cancelled = false))"),
        @NamedQuery(name = "Reservation.countActiveClientReservations", query = "select count(*) from Reservation where id_client = :idClient " +
                "and is_active is not null and is_active = true and (is_cancelled is null or (is_cancelled is not null and is_cancelled = false))")})
@Table(name = "lib_reservation")
@SequenceGenerator(name = "seq_lib_reservation", sequenceName = "seq_lib_reservation", allocationSize = 1)
public class Reservation {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_lib_reservation")
    @Column(name = "id_reservation", unique = false, nullable = false, insertable = true, updatable = true)
    private Integer idReservation;
```

```java
HibernateUtil.getSession() Session
 .getNamedQuery( queryName: "Reservation.checkIfVolumeActiveReservationExists") Query
 .setParameter( name: "idVolume", idVolume)
 .uniqueResult() != null;
```

Definicja zapytań @NamedQuery oraz sposób wywołania w systemie eLibrary

Liquibase - informacje ogólne

Liquibase to narzędzie, które zostało wdrożone w systemie eLibrary (w module elibrary-hibernate) i służy do zarządzania zmianami w bazie danych. Zarządzanie zmianami dotyczy dwóch obszarów:

- Zmiany DML (ang. *data manipulation language*) - zmiany dotyczące danych w bazie danych;
- Zmiany DDL (ang. *data definition language*) - zmiany dotyczące struktury bazy danych.

Narzędzie Liquibase (alternatywnie stosuje się również narzędzie ‘FlyWay’) daje możliwość łatwego zarządzania, wdrażania i śledzenia wykonywanych zmian. W eLibrary zastosowano plugin ‘Maven’owy’, dzięki czemu polecenia Liquibase stosowane są przy użyciu ‘Maven’a’ oraz zaimplementowana jest integracja z serwerem ciągłej integracji (ang. *Continuous Integration*) (‘Heroku’), dzięki czemu skrypty ‘Liquibase’owe’ są wykorzystywane w procesie budowy aplikacji i generowania wzorcowej bazy danych.

Więcej informacji: https://www.liquibase.org/.

Liquibase - konfiguracja

Plugin ‘Maven’owy’ związany z ‘Liquibase’em’ został umieszczony w pliku ‘pom.xml’ w projekcie elibrary-hibernate. Dodatkowo w celu osiągnięcia integracji z serwerem CI dodano plugin związany z ‘Heroku’. Dodatkowo w katalogu ‘resources’ dodano plik ‘db-changelog-master.xml’, w którym umieszcza się generowane pliki zmian - tzw. ‘changelog’i’ - znajdujące się w katalogu ‘resources/db/changelog’ (dokładna specyfikacja realizacji zmian zostanie opisana w następnych slajdach).

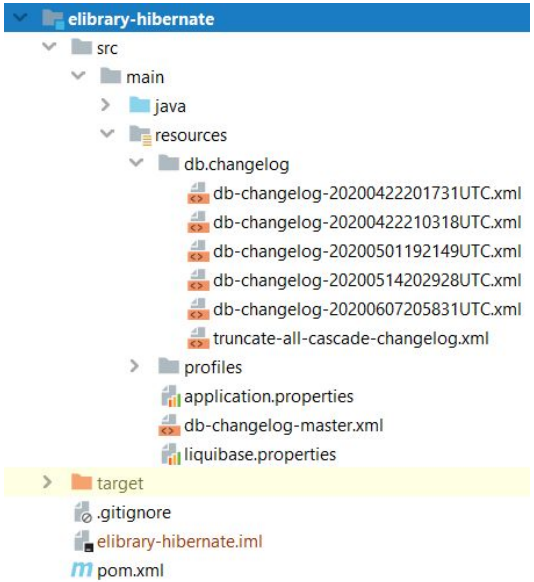
Konfiguracja plugin’u ‘Maven’owego’ została wykonana w ten sposób, że serwer CI podczas budowy wersji wykonuje zmiany w bazie danej, które zostały wprowadzone w plikach ‘changelog’ów’. Zatem, konfiguracja plugin’u dotyczy zdalnej bazy danych, znajdującej się na serwerze Heroku.

Mechanizm działania ‘Liquibase’a’ powoduje wygenerowanie dodatkowych struktur w bazie danych, które służą do śledzenia zmian wykonywanych w bazie danych i należy do nich tabela ‘databasechangelog’.

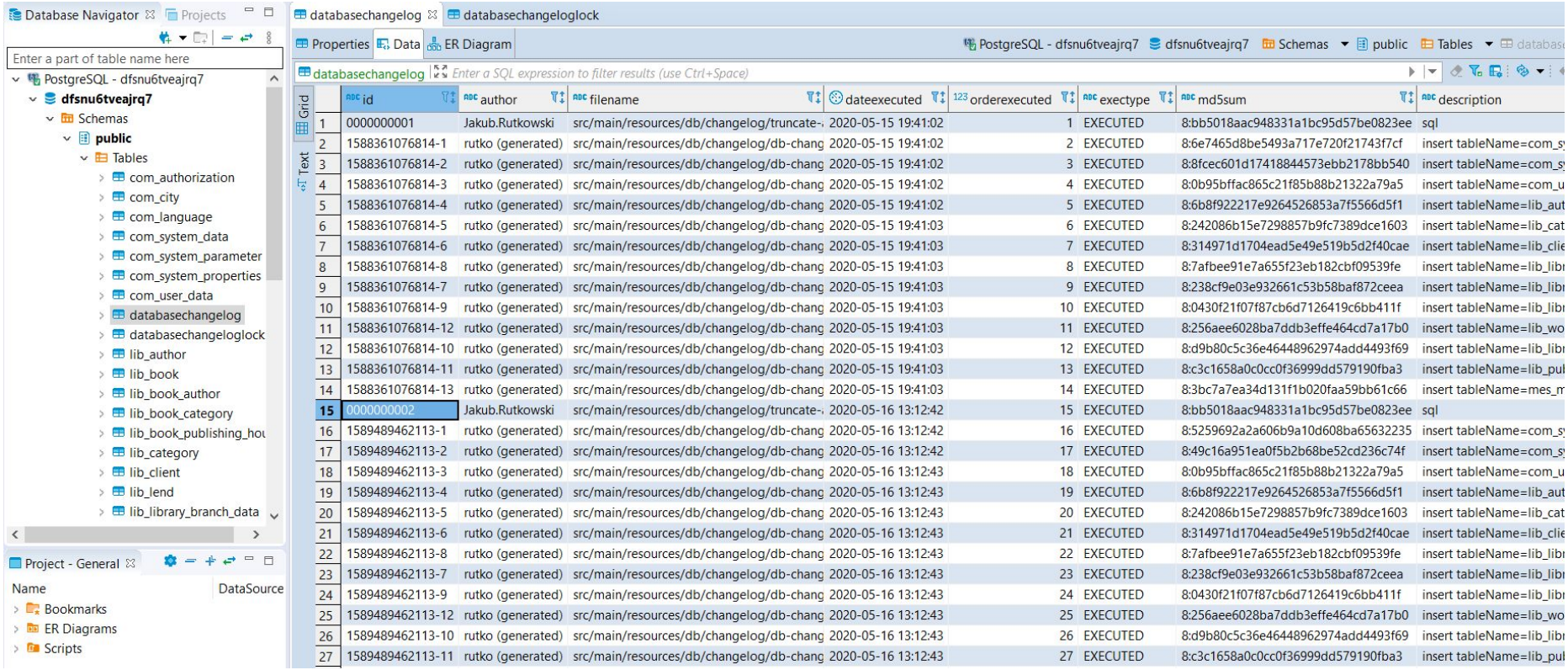
pom.xml (elibrary-hibernate)

```xml
    <build>
        <plugins>
            <plugin>
                <groupId>org.liquibase</groupId>
                <artifactId>liquibase-maven-plugin</artifactId>
                <version>${org.liquibase.version}</version>
                <configuration>
                    <propertyFile>liquibase.properties</propertyFile>
                    <changeLogFile>src/main/resources/db-changelog-master.xml</changeLogFile>
                    <!-- comment if running locally -->
                    <url>
                        jdbc:postgresql://
                    </url>
                    <username>              </username>
                    <password>                                                  </password>
                    <promptOnNonLocalDatabase>false</promptOnNonLocalDatabase>
                </configuration>
            </plugin>

            <plugin>
                <groupId>com.heroku.sdk</groupId>
                <artifactId>heroku-maven-plugin</artifactId>
                <version>3.0.2</version>
                <configuration>
                    <jdkVersion>${maven.compiler.target}</jdkVersion>
                    <configVars>
                        <MAVEN_JAVA_OPTS>liquibase:update -P prod</MAVEN_JAVA_OPTS>
                    </configVars>
                </configuration>
            </plugin>
        </plugins>
        <filters>
            <filter>src/main/resources/profiles/${build.profile.id}/application.properties</filter>
        </filters>
        <resources>
            <resource>
                <filtering>true</filtering>
                <directory>src/main/resources</directory>
                <includes>
                    <include>*.properties</include>
                </includes>
            </resource>
        </resources>
    </build>
```

Plugin’y ‘Maven’owe’ związane z konfiguracją ‘Liquibase’a’ oraz integracją go z ‘Heroku’



Struktura katalogów projektu przedstawiająca lokalizacje plików ‘master’ oraz ‘changelog’ów’



Wygenerowane przez ‘Liquibase’ struktury tabel ‘databasechangelog’ oraz ‘databasechangeloglock’

Liquibase - Realizacja zmian DML

Algorytm realizacji zmian DML przy użyciu ‘Liquibase’a’ jest następujący:

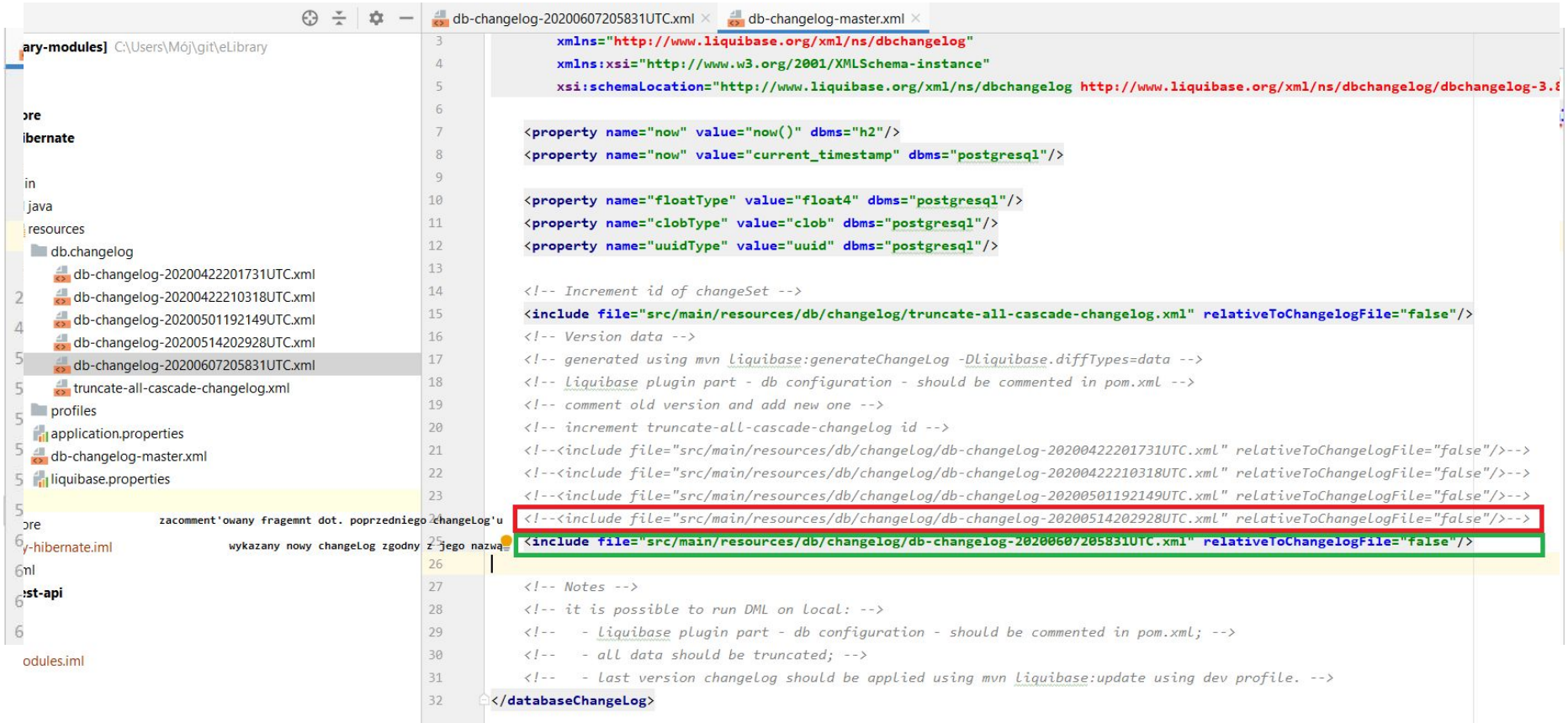
1. Zmianę DML zaleca się wykonać na bazie danych zgodnej z bazą produkcyjną (wzorcową). Jeżeli nie chce się tracić danych z lokalnej bazy testowej można zmienić tymczasowo jej nazwę a następnie utworzyć kolejną, zgodną z nazwą bazy danych obsługiwaną przez system - ‘library’. Następnie w celu zaimportowania danych wzorcowych zgodnych z bazą produkcyjną należy postępować zgodnie z algorytmem przedstawionym na slajdzie ‘Liquibase - Import danych wzorcowych’;
2. Kolejny krok polega na zacomment’owaniu fragmentu kodu w pliku pom.xml modułu elibrary-hibernate pod komentarzem *<!-- comment if running locally -->* (należy zacomment’ować znaczniki: url, username, password, promptOnNonLocalDatabase); UWAGA: nie należy zacomment’owanego kodu commitować!
3. Następnie należy upewnić się, że w bazie danych znajdują się odpowiednie zmiany w danych, które mają być wdrożone w bazie produkcyjnej;
4. W pliku truncate-all-cascade-changelog.xml należy zinkrementować wartość parametru ‘id’ w znaczniku ‘changeSet’ np. z **<changeSet author="Jakub.Rutkowski" id="0000000003">** na **<changeSet author="Jakub.Rutkowski" id="0000000004">**;
5. Kolejny krok polega na wygenerowaniu pliku changelog’u (znajdować się on będzie w lokalizacji resources/db/changelog, a jego struktura będzie następująca: db-changelog-datetimezone.xml np. db-changelog-20200607205831UTC.xml) zawierającego dane które mają być wdrożone. W celu wygenerowania takiego pliku należy wykonać polecenie *mvn liquibase:generateChangeLog -Dliquibase.diffTypes=data*;
6. Następnie należy w wygenerowanym pliku w ramach punktu (5) poprawić kolejność ‘changeSet’ów’ tak aby uniknąć konfliktów i błędów związanych z nieprawidłowościami w zakresie więzów integralności (klucze główne i klucze obce) (patrz następny slajd);
7. W ramach kroku (7) należy wykazać wygenerowany plik ‘changeLog’u’ w pliku ‘master’ - ‘db-changelog-master.xml’ tzn. należy zacomment’ować poprzednio dodany/e ‘changeLog/i’ i dodać następny wpis pod poprzednim (patrz slajd po następnym);
8. Ostatni krok polega na odcomment’owaniu zmian wykonanych w punkcie (2) oraz zacommit’owaniu plików: wygenerowanego changeLog’u oraz pliku ‘master’;
9. Ostatnim etapem jest sprawdzenie czy zmiana została rzeczywiście wprowadzona w bazie wzorcowej. Popełnienie błędu w ww. krokach może skutkować błędem podczas budowy nowej wersji aplikacji (po ‘zmerge’owaniu gałęzi do gałęzi ‘develop’) co spowoduje wysłanie odpowiedniego powiadomienia e-mail na które należy niezwłocznie zareagować.

db-changelog-20200607205831UTC.xml

```
1   <?xml version="1.1" encoding="UTF-8" standalone="no"?>
2   <databaseChangeLog xmlns="http://www.liquibase.org/xml/ns/dbchangelog" xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext" xm
3       <changeSet author="Mój (generated)" id="1591996331364-0"...>
12      <changeSet author="Mój (generated)" id="1591995646370-1"...>
19      <changeSet author="Mój (generated)" id="1591995646370-2"...>
32      <changeSet author="Mój (generated)" id="1591995646370-3"...>
82      <changeSet author="Mój (generated)" id="1591995646370-4"...>
276     <changeSet author="Mój (generated)" id="1591995646370-5"...>
404     <changeSet author="Mój (generated)" id="1591995646370-6"...>
534     <changeSet author="Mój (generated)" id="1591995646370-7"...>
557     <changeSet author="Mój (generated)" id="1591995646370-9"...>
563     <changeSet author="Mój (generated)" id="1591995646370-8"...>
572     <changeSet author="Mój (generated)" id="1591995646370-10"...>
578     <changeSet author="Mój (generated)" id="1591995646370-13"...>
596     <changeSet author="Mój (generated)" id="1591995646370-11"...>
618     <changeSet author="Mój (generated)" id="1591995646370-12"...>
668     <changeSet author="Mój (generated)" id="1591995646370-14"...>
685 </databaseChangeLog>
686
```

Domyślnie kolejność changeSet'ów jest zgodna z wartościami parametrów id, należy tą kolejność zmodyfikować i zachować kolejność changeSet'ów związanych z tabelami taka jaka jest w poprzednim pliku changeLog'u.

Zmiana w kolejności ‘changeSet’ów’ w pliku ‘changeLog’u’ w celu uniknięcia błędów związanych z kluczami



Wykazywanie ‘changeLog’u’ w pliku ‘master’

Liquibase - Realizacja zmian DDL

Algorytm realizacji zmian DDL przy użyciu ‘Liquibase’a’ jest następujący:

1. Zmianę DDL zaleca się wykonać na bazie danych zgodnej z bazą produkcyjną (wzorcową). Jeżeli nie chce się tracić danych z lokalnej bazy testowej można zmienić tymczasowo jej nazwę a następnie utworzyć kolejną, zgodną z nazwą bazy danych obsługiwaną przez system - ‘library’. Następnie w celu zaimportowania danych wzorcowych zgodnych z bazą produkcyjną należy postępować zgodnie z algorytmem przedstawionym na slajdzie ‘Liquibase - Import danych wzorcowych’;
2. Kolejny krok polega na zacomment’owaniu fragmentu kodu w pliku pom.xml modułu elibrary-hibernate pod komentarzem *<!-- comment if running locally -->* (należy zacomment’ować znaczniki: url, username, password, promptOnNonLocalDatabase); UWAGA: nie należy zacomment’owanego kodu commitować!
3. Następnie należy upewnić się, że w bazie danych znajdują się odpowiednie zmiany w strukturze, które mają być wdrożone w bazie produkcyjnej;
4. Kolejny krok polega na wygenerowaniu pliku changelog’u (znajdować się on będzie w lokalizacji resources/db/changelog, a jego struktura będzie następująca: db-changelog-datetimezone.xml np. db-changelog-20200607205831UTC.xml) zawierającego dane które mają być wdrożone. W celu wygenerowania takiego pliku należy wykonać polecenie *mvn liquibase:generateChangeLog*;
5. Następnie należy w wygenerowanym pliku w ramach punktu (4) wyszukać odpowiedniej zmiany DDL (odpowiedniego ‘changeSet’a) i usunąć pozostałe ‘changeSet’y’. Dodatkowo zgodnie z przyjętą konwencją, należy zmodyfikować nazwę ‘changeLog’a’ dodając w odpowiednie miejsce frazę ‘ddl’ - db-changelog-ddl-datetimezone.xml np. db-changelog-ddl-20200607205831UTC.xml;
6. W ramach kroku (6) należy wykazać wygenerowany plik ‘changeLog’u’ w pliku ‘master’ - ‘db-changelog-master.xml’ tzn. należy zacomment’ować poprzednio dodany/e ‘changeLog/i’ i dodać następny wpis pod poprzednim (patrz slajd ‘Wykazywanie ‘changeLog’u’ w pliku ‘master’’);
7. Ostatni krok polega na odcomment’owaniu zmian wykonanych w punkcie (2) oraz zacommit’owaniu plików: wygenerowanego changeLog’u oraz pliku ‘master’;
8. Ostatnim etapem jest sprawdzenie czy zmiana została rzeczywiście wprowadzona w bazie wzorcowej. Popełnienie błędu w ww. krokach może skutkować błędem podczas budowy nowej wersji aplikacji (po ‘zmerge’owaniu gałęzi do gałęzi ‘develop’) co spowoduje wysłanie odpowiedniego powiadomienia e-mail na które należy niezwłocznie zareagować.

Liquibase - Realizacja zmian DML/DDL

Algorytm realizacji zmian hybrydowych DML/DDL przy użyciu ‘Liquibase’a’ jest następujący:

1. Realizując zmianę DML/DDL należy postępować zgodnie z procedurą dla obu typu zmian, modyfikacji ulegają natomiast punkty (7) w procedurze dla zmian DML oraz punkt (6) dla zmian DDL. Punkty odpowiednio (8) oraz (9) i (7) oraz (8) zostają zastąpione wspólną procedurą jak niżej:
    a. Efektem wykonania zmian specyficznych dla DML oraz DDL są dwa pliki ‘changeLog’ów’ - jeden ze zmianami DML, drugi natomiast ze zmianami DDL. W takim przypadku należy oba pliki wykazać w pliku ‘master’, z pewnym bardzo istotnym ograniczeniem. Plik DDL musi być wykazany jako pierwszy. Oczywiście należy zacomment’ować poprzednio dodany/e ‘changeLog/i’;
    b. Następnie należy wykonać dwa ostatnie opisane kroki, które są wspólne dla obu procedur - DML i DDL czyli:
        i. Ostatni krok polega na odcomment’owaniu zmian wykonanych w punkcie (2) oraz zacommit’owaniu plików: wygenerowanego changeLog’u oraz pliku ‘master’;
        ii. Ostatnim etapem jest sprawdzenie czy zmiana została rzeczywiście wprowadzona w bazie wzorcowej. Popełnienie błędu w ww. krokach może skutkować błędem podczas budowy nowej wersji aplikacji (po ‘zmerge’owaniu gałęzi do gałęzi ‘develop’) co spowoduje wysłanie odpowiedniego powiadomienia e-mail na które należy niezwłocznie zareagować.

Liquibase - Import danych wzorcowych

Algorytm realizacji zmian importu danych wzorcowych przy użyciu ‘Liquibase’a’ jest następujący:

1. Przed wykonaniem importu należy usunąć lokalną bazę danych (bądź zmienić jej nazwę np. library-old-1) oraz utworzyć nową. Następnie, należy uruchomić system ‘eLibrary’ tak aby wykonały się zmiany DDL (struktura bazy danych w systemie generowana jest przy pierwszym uruchomieniu aplikacji). Nie należy się logować do systemu. Po wygenerowaniu struktury i pojawieniu się ekranu logowania należy zamknąć system.
2. Kolejny krok polega na zacomment’owaniu fragmentu kodu w pliku pom.xml modułu elibrary-hibernate pod komentarzem *<!-- comment if running locally -->* (należy zacomment’ować znaczniki: url, username, password, promptOnNonLocalDatabase); UWAGA: nie należy zacomment’owanego kodu commitować!
3. Następnie należy zmienić profil, lub upewnić się, że używany jest profil develop’erski tzn. należy wykonać polecenie *mvn clean install -Pdev*;
4. W następnym kroku należy wykonać polecenie *mvn liquibase:update*;
5. Ostatni krok polega na odcomment’owaniu zmian wykonanych w punkcie (2);
6. Po wykonaniu kroku (5) zostaje tylko sprawdzenie czy wszystkie dane zostały poprawnie zaimportowane.

Liquibase - dodatkowe informacje

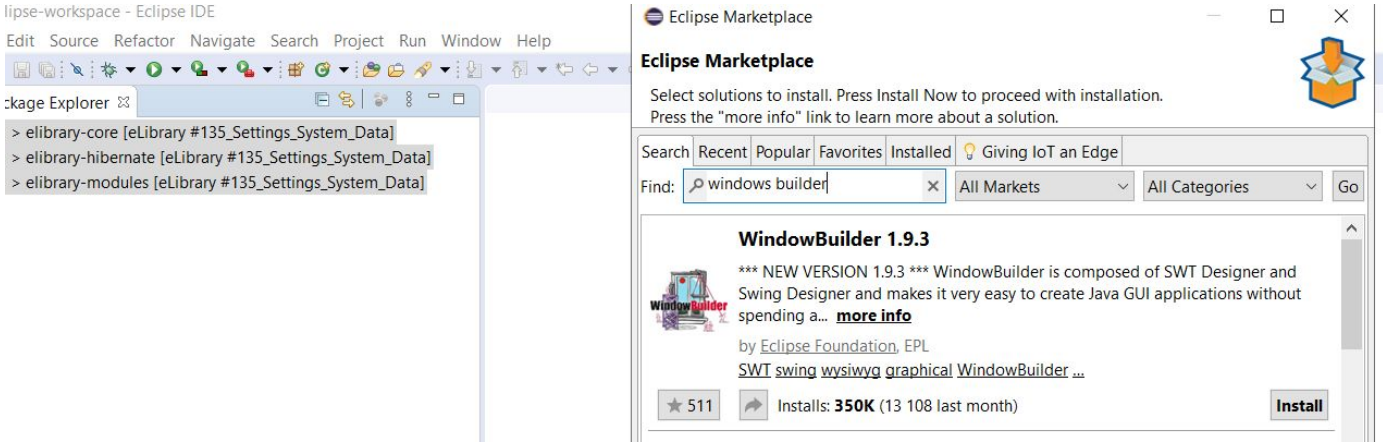
Liquibase jest narzędziem niezwykle użytecznym i daje wiele możliwości. Część z nich dotyczy na przykład możliwości wykonywanie złożonych zapytań oraz sposobności porządkowania zmian realizowanych w bazie i ich śledzeniu/analizie.

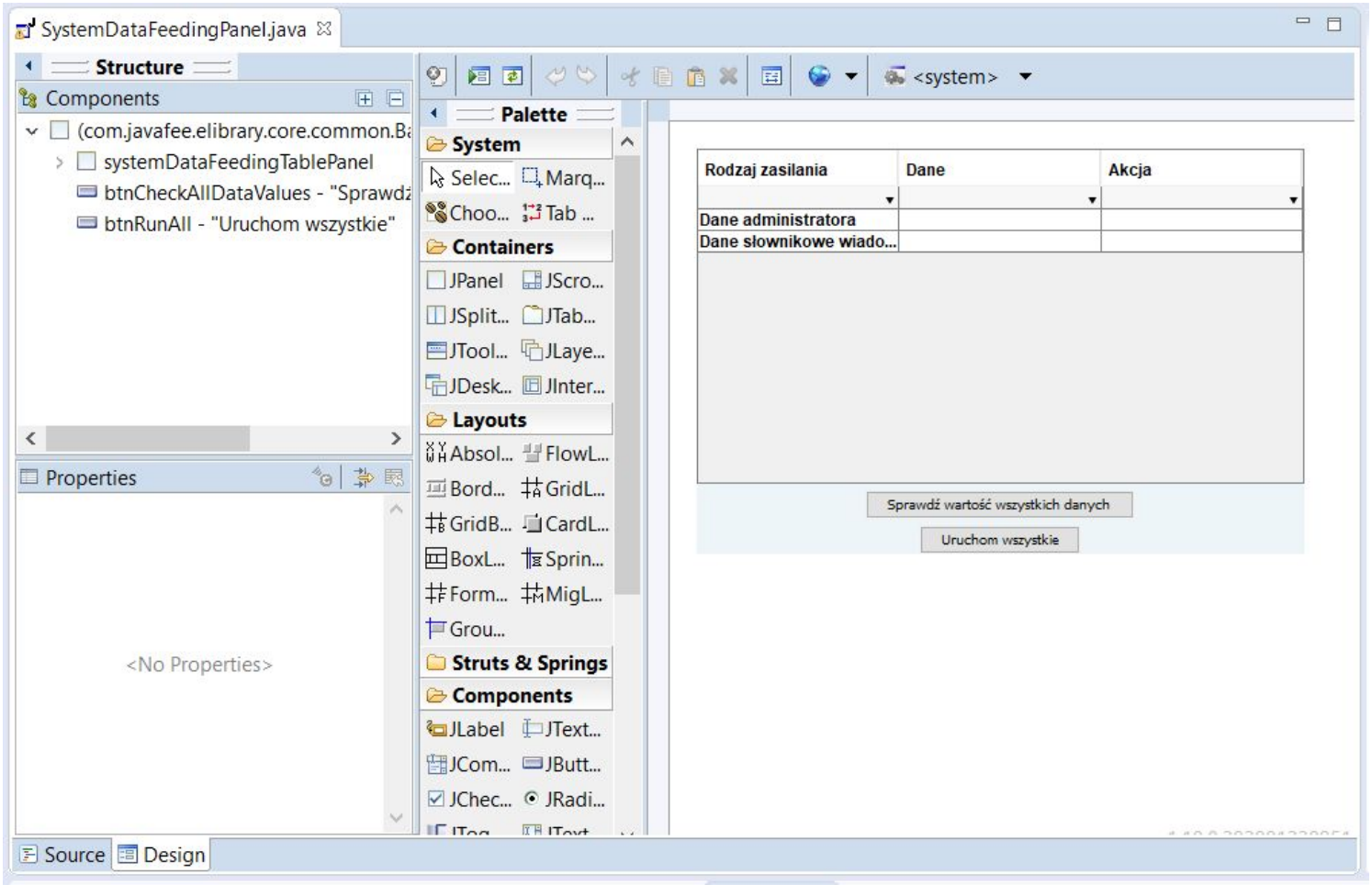
Warto zwrócić uwagę, że w pliku ‘db-changelog-master.xml’ umieszczono podstawowe, skrótowe informacje w komentarzach, które mogą być cennymi wskazówkami przydatnymi w procesie realizacji zmian DML, DDL, DML/DDL.

View

View - realizacja zmian

W systemie elibrary, realizacja zmian po stronie front-end'u odbywa się przy użycia środowiska Eclipse IDE, plugin'u Windows Builder. Plugin ten udostępnia edytor 'drag&drop' wspierający framework Swing, dzięki któremu realizacja zmian po stronie widoku jest łatwiejsza i szybsza. Planowana jest realizacja migracji z edytora Eclipse na edytor dostępny w ramach środowiska IntelliJ IDEA.

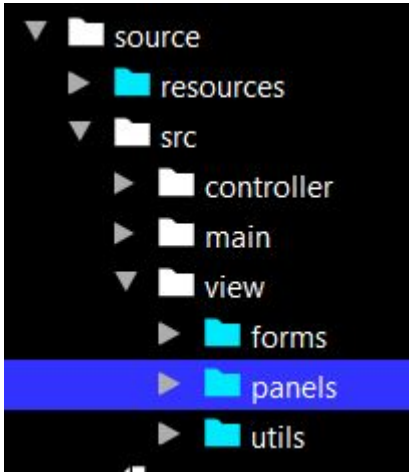




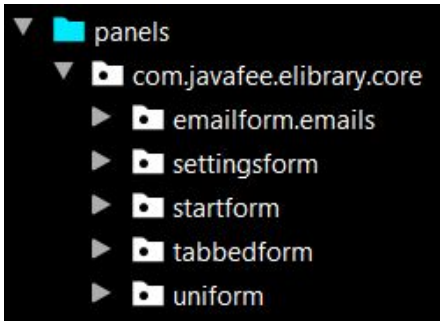
Windows Builder plugin w środowisku Eclipse IDE

View

Klasy obsługi interfejsu graficznego użytkownika (GUI) znajdują się w ramach projektu elibrary-core w folderze źródłowym ‘view’. W ramach tego folderu, znajdują się trzy podkatalogi: ‘forms’, ‘panels’, ‘utils’.

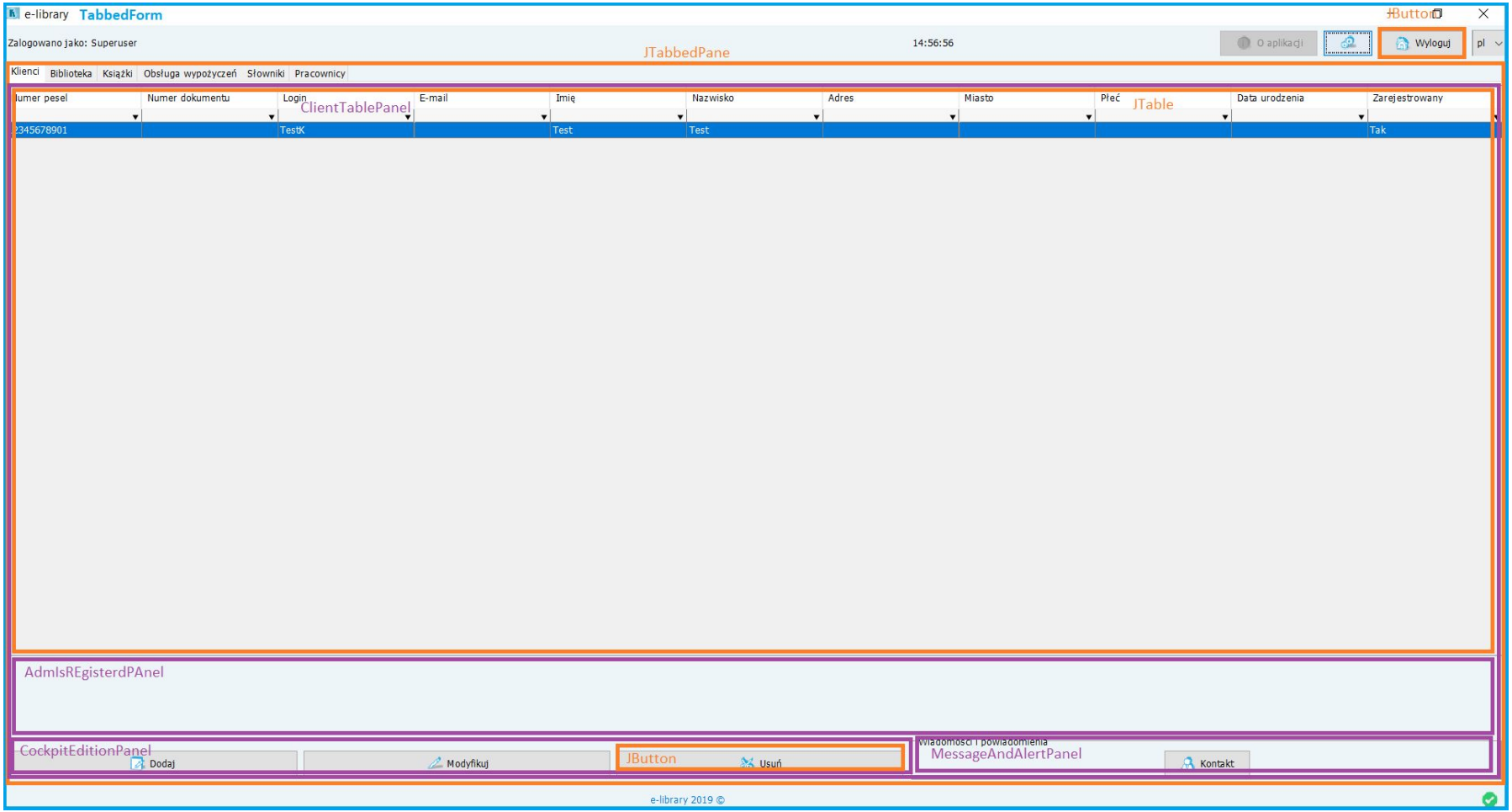


Dodatkowo w ramach folderu źródłowego panels, oprócz paczek związanych z poszczególnymi oknami, wyodrębniono również paczkę ‘uniform’ w której umieszcza się uniwersalne panele mogące występować na różnych oknach (z tego właśnie powodu nie są umieszczone w konkretnej paczce, konkretnego form).



Podstawowe elementy GUI w elibrary

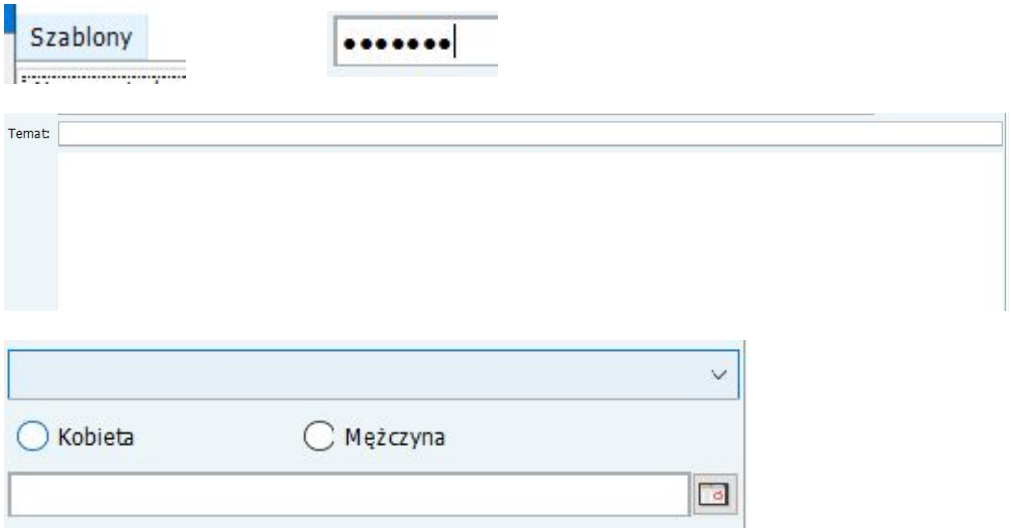
Głównym elementem GUI są okna aplikacji (forms). W nich mogą znajdować się panele (panels), które z kolei grupują logicznie pewne komponenty graficzne (components). Na następnym slajdzie zaprezentowano zrzut ekranu prezentujący opisany podział i opisujący elementy okna.



Podział okna ze względu na klasy GUI - form i panels

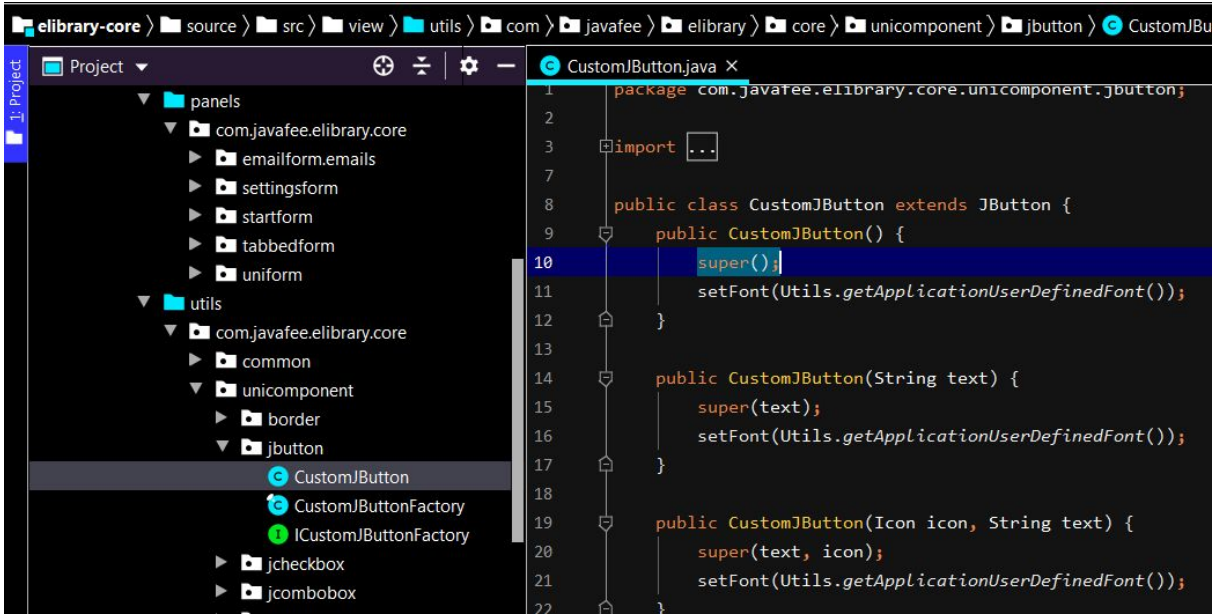
Podstawowe elementy GUI w elibrary - komponenty

Na ekranach (inaczej: oknach - frame) oraz w panelach znajdować się mogą różne elementy interfejsu graficznego. Komponenty zaimplementowane są w ramach framework'u Swing i należą do nich m.in.:

JLabel JTextField JTextArea JButton JCheckBox JRadioButton JComboBox JDatePicker	JPasswordField JTable JTabbedPane JMenuBar JMenu JMenuItem	

Podstawowe elementy GUI w elibrary - komponenty projektowe

Wbrew temu co zostało napisane w ramach poprzedniego slajdu, w systemie e-library używane są zawsze (lub prawie zawsze :)) komponenty projektowe. Pod pojęciem komponentu projektowego rozumiemy komponenty które zostały zdefiniowane w ramach projektu i które rozszerzają komponenty z framework'u Swing. Dzięki temu komponenty projektowe mogą posiadać 'predefiniowane' cechy dodatkowe np. takie jak ustawienie czcionki, lub koloru. Dzięki temu nie ma potrzeby ustawiania pewnych cech przy każdym, nowym użyciu danego komponentu. Komponenty projektowe znajdują się w paczce 'unicomponent' w folderze źródłowym 'utils'



Dzięki ustawieniu czcionki w CustomJButton, nie ma potrzeby ustawiania jej za każdym razem, dla każdego, nowego przycisku. Należy jednak pamiętać uby używać komponentu projektowego CustomJButton, zamiast tego ze Swing'a - JButton. Zmianę tą należy wykonać ręcznie - CustomJButton nie jest dostępny w palecie.

Użycie plugin’u Windows Builder

Plugin Windows Builder służy do tworzenia GUI z użyciem edytora ‘drag&drop’. Windows Builder posiada ‘paletę’ która zawiera komponenty. Aby dodać komponent do okna należy go przeciągnąć na panel/frame. Przeciągnięcie elementu skutkuje wygenerowaniem odpowiedniego fragmentu kodu w klasie obsługującej dany element - panel/frame. Domyślnie, Windows Builder generuje elementy wewnątrz konstruktora. Jednak, na potrzeby projektu, sposób generowania kodu został zmodyfikowany tak, aby większość komponentów (zazwyczaj za wyjątkiem elementów typu JLabel) dodawana była jako właściwości danej klasy. Dzięki temu, po dodaniu adnotacji @Getter, programista jest w stanie odwołać się do komponentu z poziomu warstwy Controller’a.

Controller

Profile oraz klasa SystemProperties

W systemie eLibrary wyróżnia się klasę ‘SystemProperties’ do odpowiedzialności której należą:

- inicjalizacja połączeń tj. z bazą danych oraz np. z API;
- załadowanie konfiguracji dot. m.in. ww. elementów;
- dostęp do mapy parametrów systemowych;
- dostęp do pliku odpowiedzialnego za obsługę wielojęzycznych komunikatów.

Z punktu widzenia technicznego w projekcie eLibrary zaimplementowano profile Maven’owe, przy pomocy których źródła można inicjalizować różnymi konfiguracjami (dotyczy to ustawienia połączenia z bazą danych oraz API). Wyróżniamy następujące profile:

- dev (deweloperski);
- prod (produkcyjny).

Aby zmienić profil aplikacji w celu np. uruchomienia aplikacji z użyciem bazy wzorcowej tj. produkcyjnej należy przebudować źródła będąc ustawionym w projekcie ‘elibrary-hibernate’ przy pomocy polecenia ‘*mvn clean install -Pprod*’.

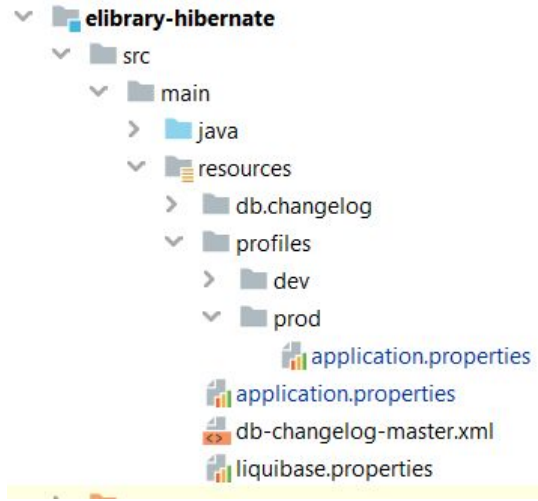
Opisane funkcje klasy ‘SystemProperties’ oraz wspomniane mechanizmy będą rozwinięte w ramach następnych slajdów.

Profile oraz klasa SystemProperties - profile

Profile pozwalają uruchomić źródła ze zróżnicowaną konfiguracją. W przypadku eLibrary zdefiniowano profile deweloperski (dev) oraz produkcyjny (prod). Konfiguracja odbywa się w pliku ‘pom.xml’ projektu ‘elibrary-hibernate’ i w zależności od wybranego profilu podczas wykonywania polecenia ‘mvn clean install -P<profil>’ (gdzie <profil> to ‘dev’ lub ‘prod’) źródła uruchamiane są w danym trybie, który sprowadza się do załadowania określonego pliku ‘application.properties’, zawierającego konfigurację. W ten sposób, uruchamiając aplikację w trybie produkcyjnym system będzie np. łączył się z bazą produkcyjną, natomiast uruchamiając w trybie deweloperskim, system będzie łączył się z bazą lokalną - deweloperską.

pom.xml (elibrary-hibernate)

```xml
        <filters>
            <filter>src/main/resources/profiles/${build.profile.id}/application.properties</filter>
        </filters>
        <resources>
            <resource>
                <filtering>true</filtering>
                <directory>src/main/resources</directory>
                <includes>
                    <include>*.properties</include>
                </includes>
            </resource>
        </resources>
    </build>

    <profiles>
        <profile>
            <id>prod</id>
            <properties>
                <build.profile.id>prod</build.profile.id>
                <flag>prod</flag>
            </properties>
        </profile>
        <profile>
            <id>dev</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <properties>
                <build.profile.id>dev</build.profile.id>
                <flag>dev</flag>
            </properties>
        </profile>
```



Definicja profili w systemie eLibrary

Profile oraz klasa SystemProperties - application.properties

Pliki konfiguracyjne znajdują się w lokalizacji ‘resources’ (elibrary-hibernate/src/main/resources /profiles/). Stanowią je pliki o nazwie ‘application.properties’. W zależności od katalogu w którym się znajdują (‘dev’ lub ‘prod’) definiują właściwości charakterystyczne dla danego profilu.

Wewnątrz pliku znajdują się pary - klucze i wartości. Należą do nich dane dotyczące konfiguracji bazy danych oraz REST’owego API.

Z punktu widzenia technicznego, plik wczytywany jest do strumienia ‘InputStream’ a następnie obiekt ze standardowej biblioteki Java’y (java.util) - ‘Properties’ zostaje załadowany zawartością tego pliku. Obiekt ‘Properties’ dostępny jest w klasie ‘SystemProperties’ za pośrednictwem ‘getter’a’ - ‘getConfigProperties()’. Metoda ta, zwraca wspomniany obiekt ‘Proprties’ i udostępnia metodę ‘getProperty’ przy pomocy której można pobierać wartości na podstawie przekazanego klucza w parametrze metody.

W zależności od aktywnego profilu, plik ‘properties’ załadowany jest wartościami parametrów z pliku dla danego profilu (w pliku w głównym katalogu nie ma wartości, znajdują się natomiast odnośniki do wartości z danego właściwego pliku konfiguracyjnego

resources\application.properties

```
# Database Config
db.url      = ${db.url}
db.username = ${db.username}
db.password = ${db.password}
db.name     = ${db.name}
```

prod\application.properties

```
# Database Config
db.url      = jdbc:p
db.ip       = ec2-46
db.username = ppqrms
db.password = 39f03b
db.name     = dfsnu6
```

SystemProperties.java

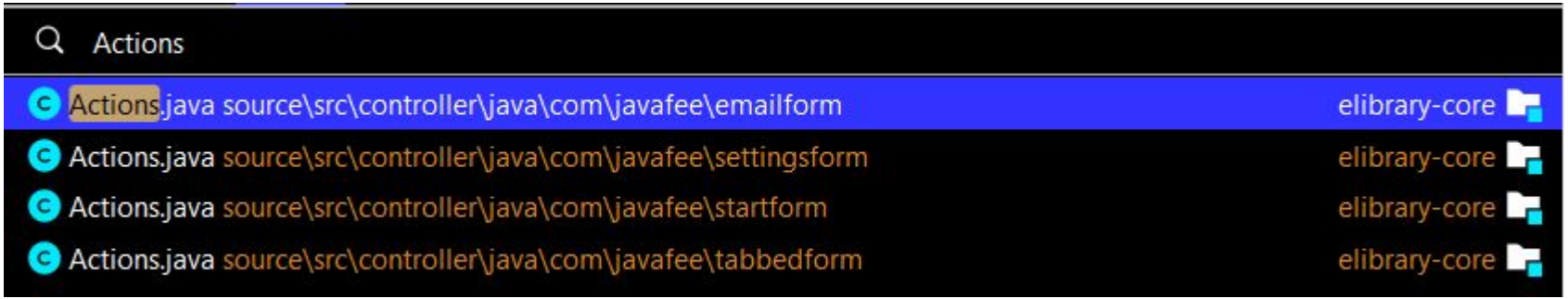
```
@Getter
private static Properties configProperties;

private static void initializeCoreProperties() {
    configProperties = new Properties();
    try (InputStream resourceAsStream = HibernateUtil.class.getClassLoader().getResourceAsStream(Constants.APPLICATION_PROPERTIES)) {
        configProperties.load(resourceAsStream);
    } catch (IOException e) {
        log.severe(e.getMessage());
    }
}
```

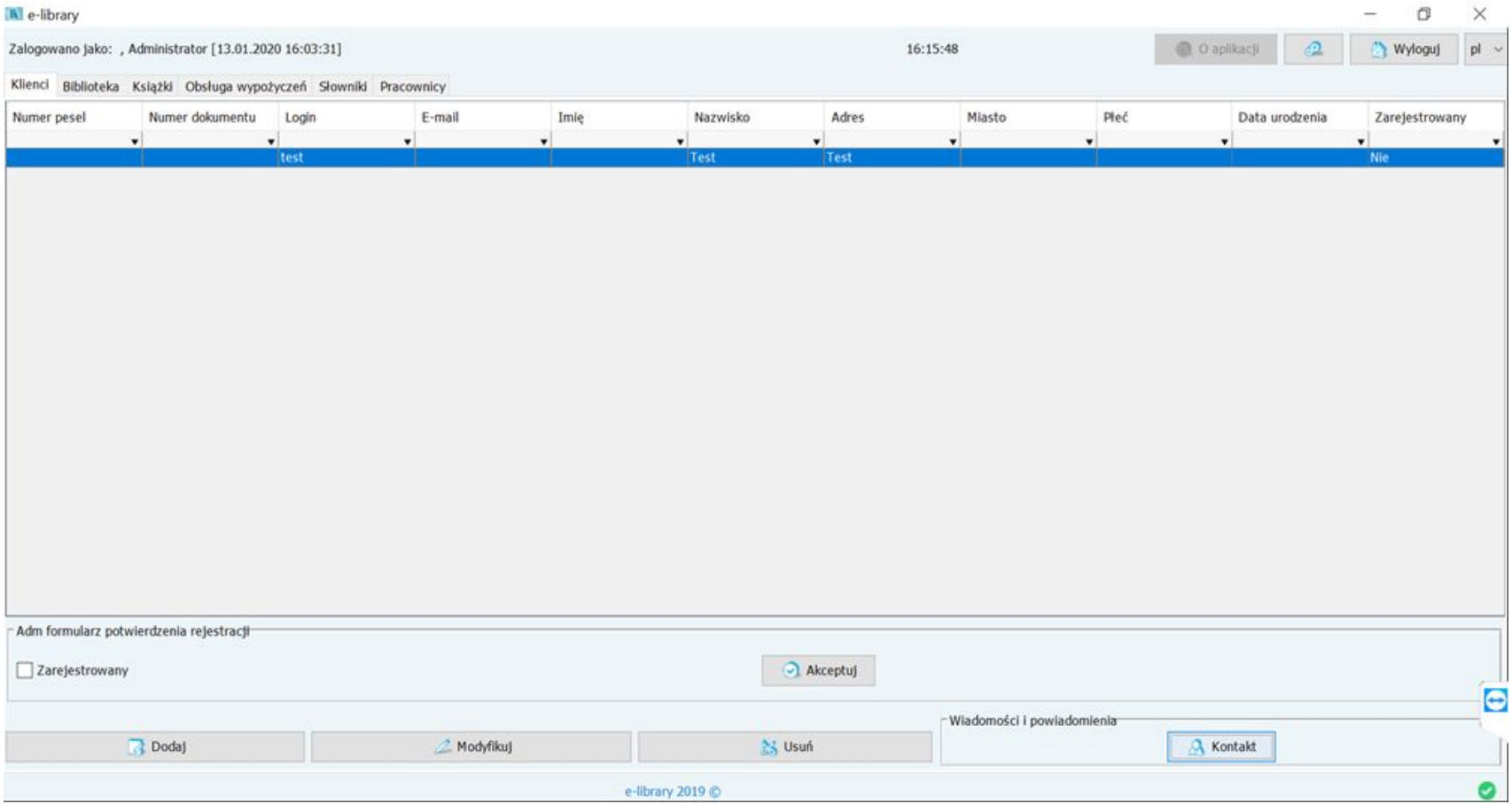
```
.getProperty("db.url")
```

Podstawianie wartości do pliku application.properties w zależności od profilu oraz implementacja inicjalizacji obiektu ‘Properties’ wraz z prezentacją metody służącej do pobierania wartości z obiektu ‘configProperties’

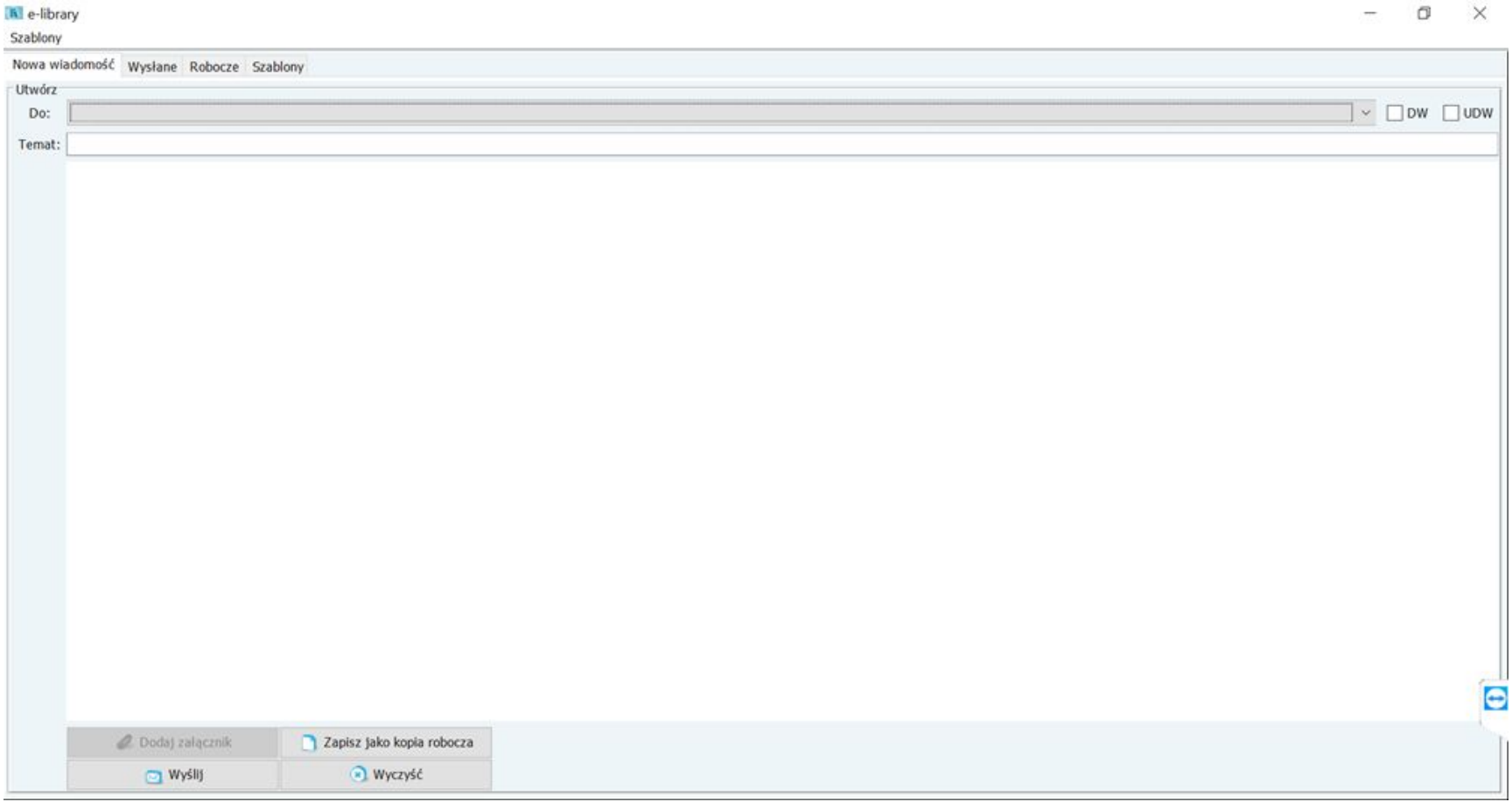
Klasa Actions

Klasa Actions służy do obsługi zdarzeń na oknach głównych w aplikacji (Form). Każde okno główne aplikacji e-library posiada swoją klasę Actions w odpowiedniej paczce odpowiadającej obsługiwanemu obszarowi merytorycznego.

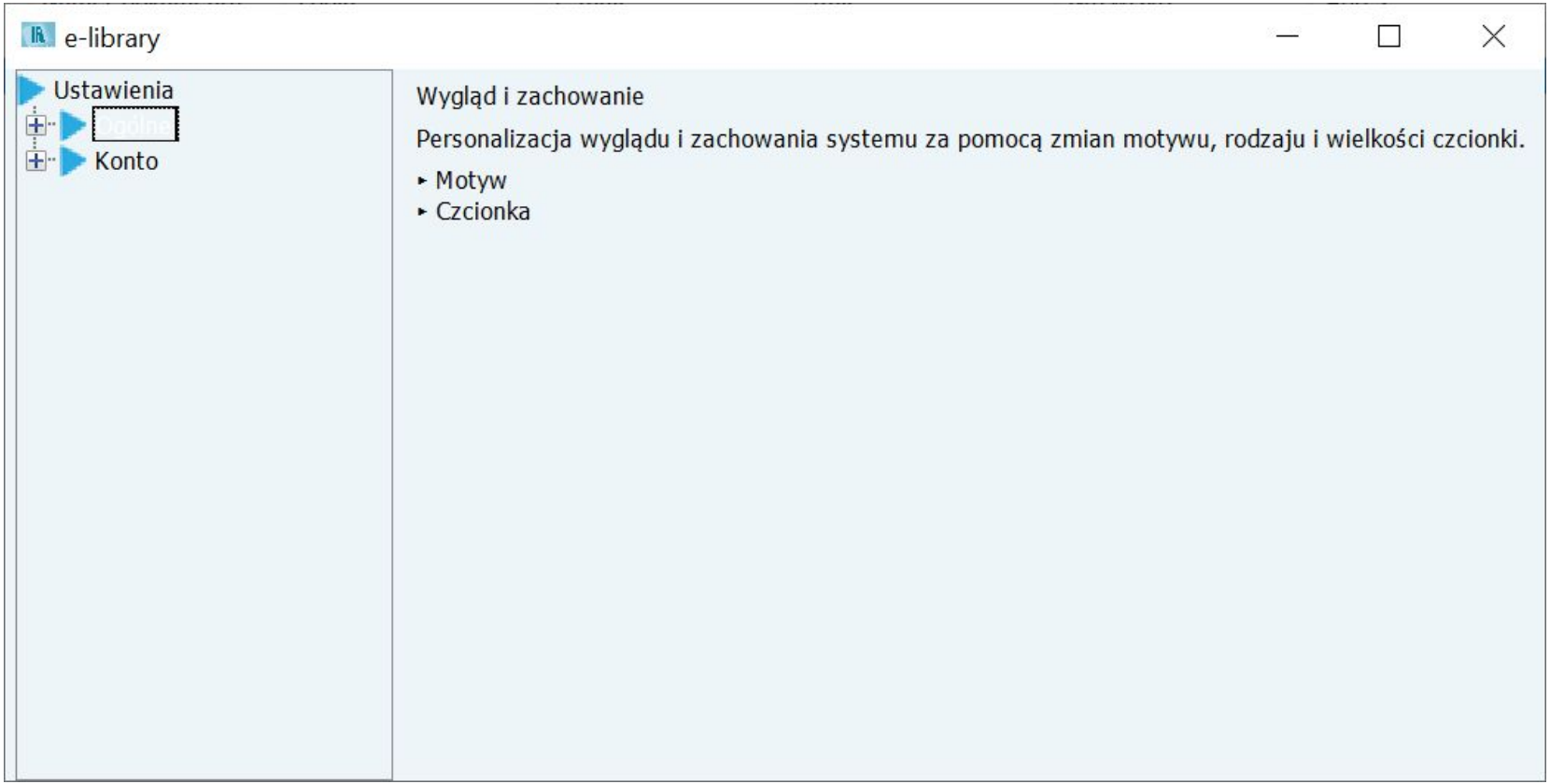




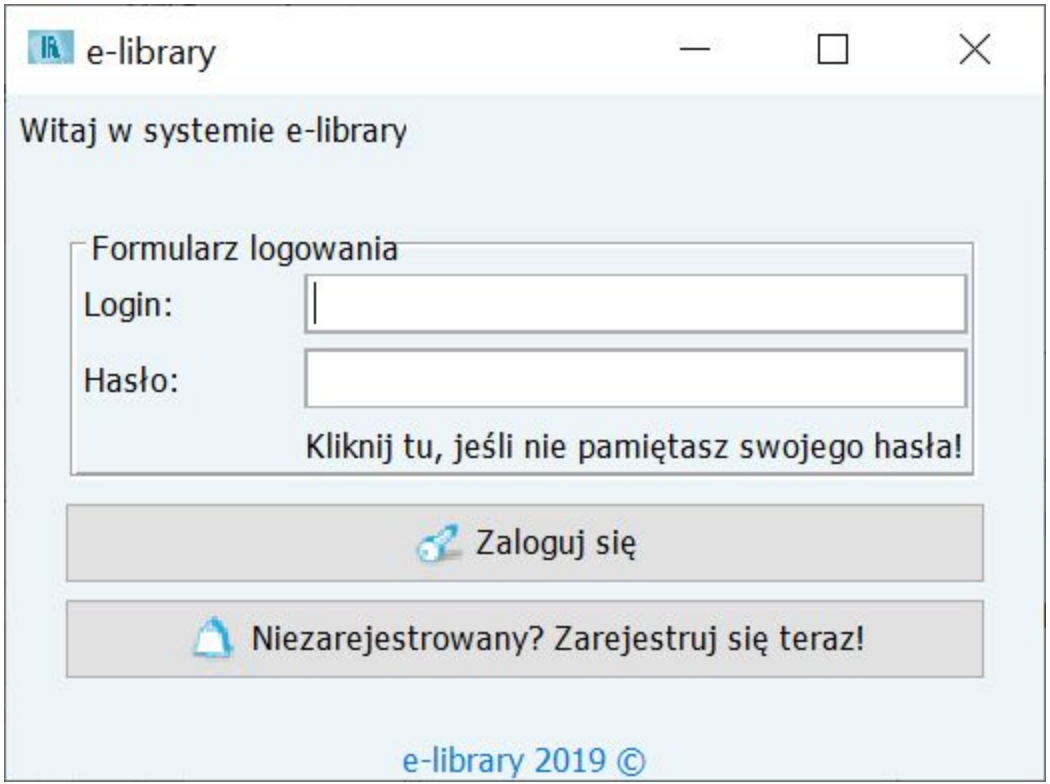
source/src/controller/java/com/javafee/tabbedform/Actions.java



source/src/controller/java/com/javafee/emailform/Actions.java



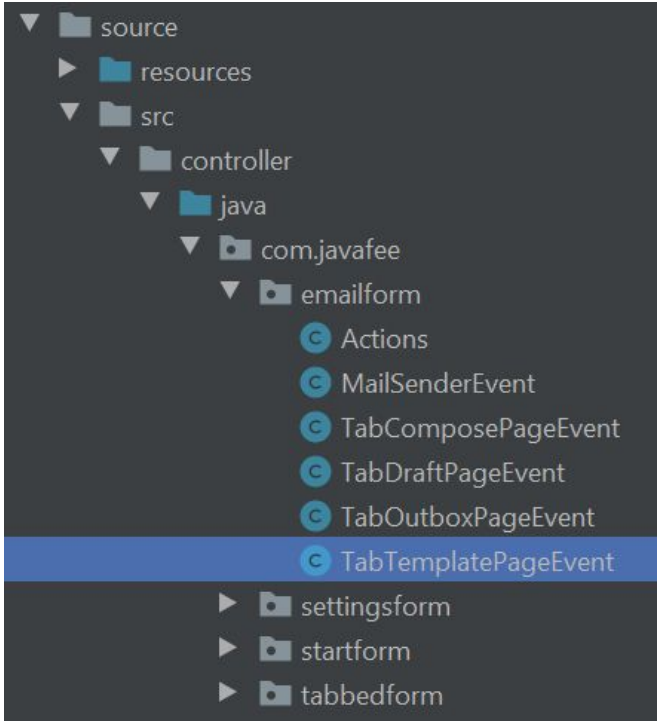
source/src/controller/java/com/javafee/settingsform/Actions.java



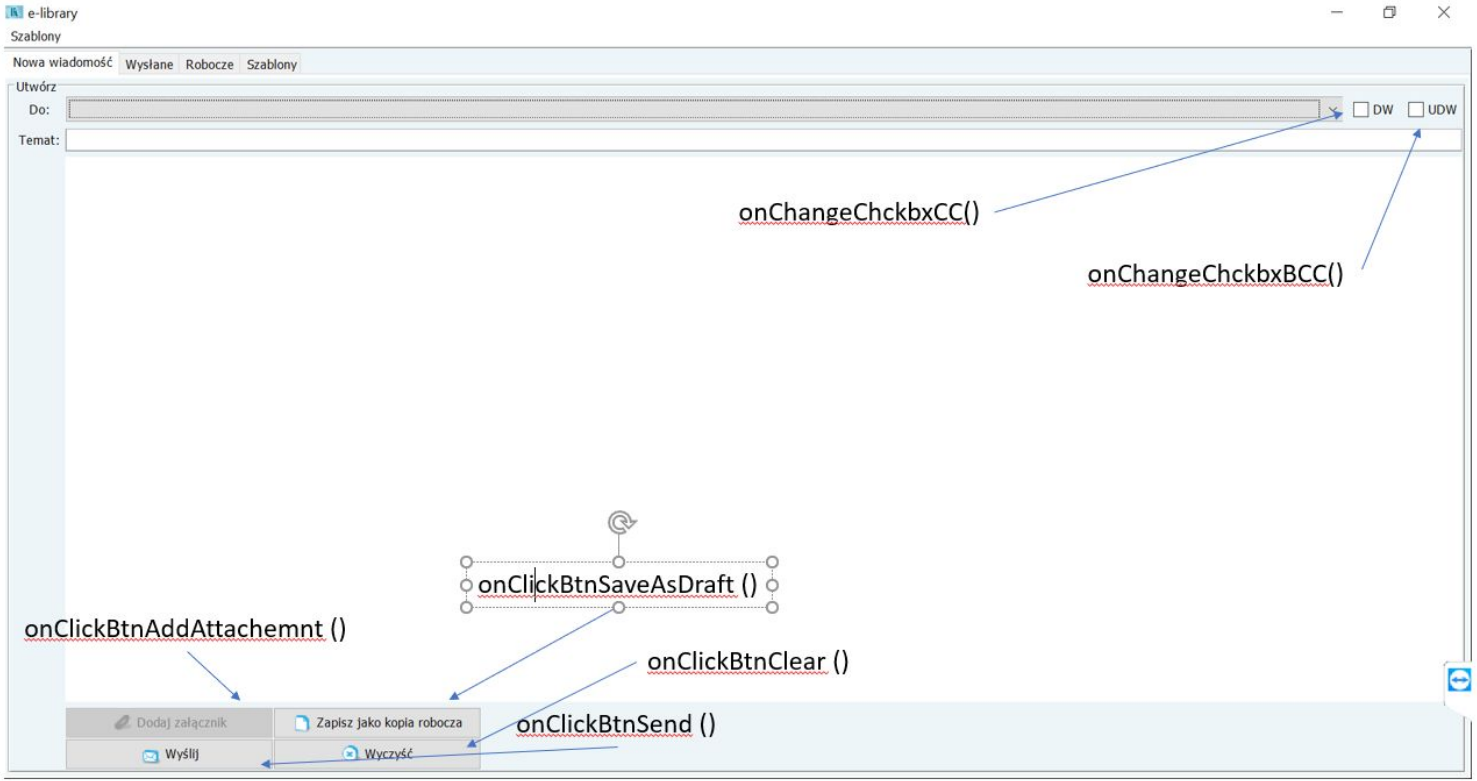
source/src/controller/java/com/javafee/startform/Actions.java

Struktura klas controller’a obsługujących okno “e-mail”

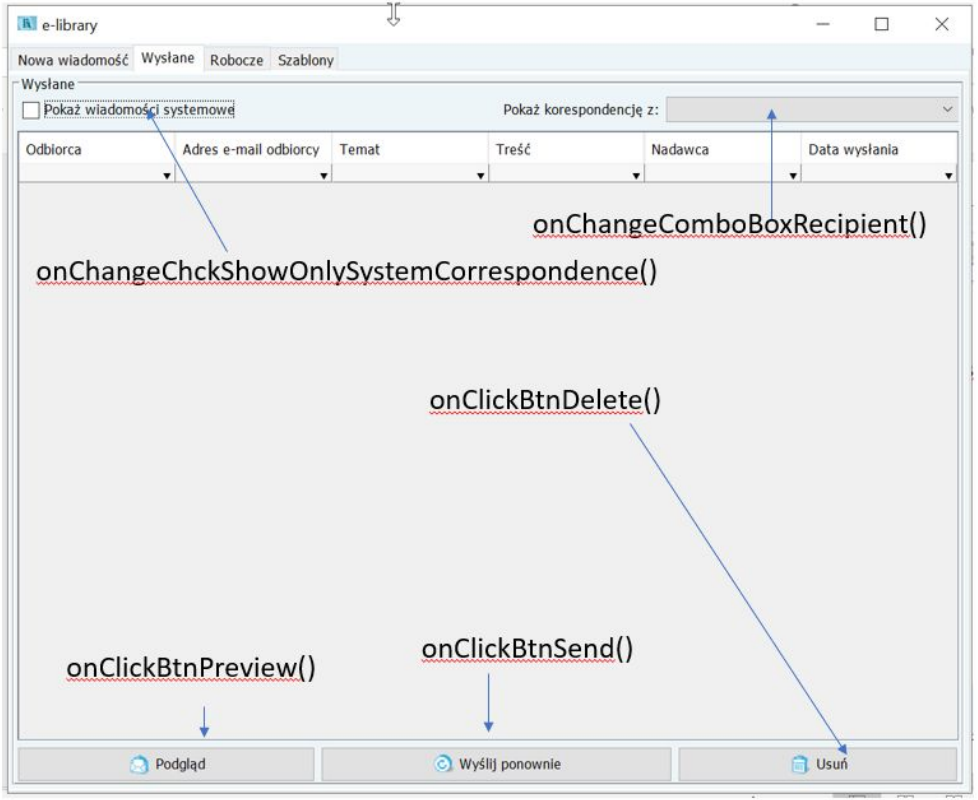
Ze względów wydajnościowych oraz z uwagi na poprawność stylu kodowania obsługa zdarzeń okien głównych nie jest realizowana jedynie w ramach klasy Actions. Funkcjonalności zostają obsłużone w zdekomponowanych klasach tzw. “zdarzeń” co jest inspirowane zasadom programowania zdarzeniowego. Powyższe klasy łączy wspólny interfejs IActionForm. Dekompozycja odbywa się według analizy merytorycznej.



Struktura klas controller’a obsługujących okno “e-mail” - obsługa zdarzeń w klasie TabComposePageEvent



Struktura klas controller’a obsługujących okno “e-mail” - obsługa zdarzeń w klasie TabOutboxPageEvent

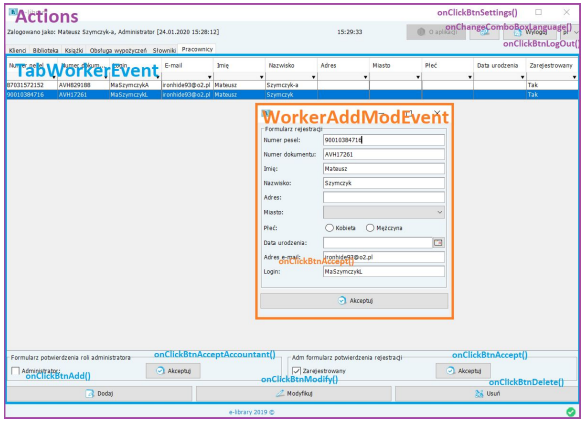


Struktura klas controller’a - klasa Event - informacje ogólne

Jak opisano na poprzednich slajdach w celu dekompozycji obsługi zdarzeń i rozłożeniu ich na poszczególne klasy podrzędne wyodrębnia się w systemie klasy ‘Event’ów’ np. te związane z obsługą zdarzeń na poszczególnych zakładkach na oknie głównym ‘TabbedForm’ - **‘TabClientEvent’**, **‘TabLibraryEvent’** - czy na przykład te, związane z obsługą zdarzeń na oknie wiadomości e-mail ‘EmailForm’ - **‘TabComposePageEvent’**, **‘TabTemplatePageEvent’** lub te związane z obsługą zdarzeń na oknie ustawień ‘SettingsForm’ - **‘ThemePanelEvent’**, **‘PersonalDataChangeEvent’** itd. Wyróżniamy dwie podgrupy klasy ‘Event’ - klasy ‘Event’ związane z ‘form’ami’ oraz z ‘fram’ami’.

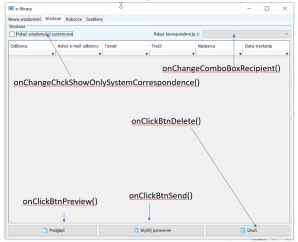
Actions
TabbedForm

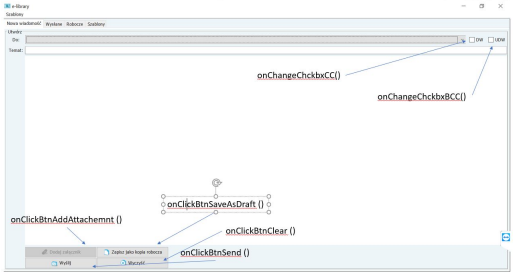
TabClientEvent
ClientTablePanel
TabLibraryEvent
LibraryTablePanel
TabBookEvent
BookTablePanel
TabLoanServiceEvent
LoanServicePanel
TabAdmDictionaryEvent
AdmDictionaryPanel
TabWorkerEvent
WorkerTablePanel



Actions
EmailForm

TabComposePageEvent
TabOutboxPageEvent
TabDraftPageEvent
TabTemplatetPageEvent





Actions
SettingsForm

PersonalDataChangePanelEvent
ThemePanelEvent
FontPanelEvent
SystemParametersPanelEvent
SystemDataFeedingPanelEvent
SystemProcessesPanelEvent

Przykłady organizacji klas ‘Event’ów według odpowiadającym im klasom ‘Actions’

Struktura klas controller’a - klasa Event - związana z form’em

Implementacja klasy ‘Event’ związanej z ‘form’em’ wymaga trzymania się następujących założeń:

- Klasa Event’u powinna implementować interfejs **‘IActionForm’** oraz jej metodę ‘initializeForm’;
- **W klasie powinno się stosować wzorzec architektoniczny *Singleton*, co implikuje implementację metody ‘getInstance()’ poprzez którą tworzy się nowe obiekty klasy (w zależności od przeznaczenia, metoda przyjmuje odpowiednie parametry), we właściwości klasy przechowywana jest instancja obiektu danej klasy ‘Event’;**
- W klasie ‘Event’u’ wyróżnia się metodę ‘control()’, **wywoływaną w konstruktorze** i **odpowiedzialną za właściwe ‘podpięcie’ obsługi zdarzeń do komponentów**;
- W klasie tej powinno się w miarę możliwości trzymać konwencji związanej z umiejscowieniem odpowiednich metod we właściwych miejscach oraz ich nazewnictwem wzorując się na istniejących już klasach - i tak, należy umiejscawiać w klasie najpierw metody związane z przeładowaniem (np. ‘reload’, ‘refresh’), następnie te z obsługą zdarzeń (np. ‘onClickBtn…’) oraz związane z walidacjami (np. ‘validate…’);
- W klasie Event’u powinna znajdować się zależność do klasy widoku (np. odpowiedniego panelu, form’a), która przekazywana jest jako parametr metody **‘getInstance()’**.

TestEvent.java

```java
package com.javafee.elibrary.core.startform;

import com.javafee.elibrary.core.common.action.IActionForm;
import lombok.Setter;

public class TestEvent implements IActionForm {
    @Setter
    private TestForm testForm;

    private static TestEvent testEvent;

    private TestEvent(TestForm testForm) { this.control(testForm); }

    public static TestEvent getInstance(TestForm testForm) {
        if (testEvent == null)
            testEvent = new TestEvent(testForm);
        return testEvent;
    }

    public void control(TestForm testForm) {
        setTestForm(testForm);
        initializeForm();

        testForm.getTestPanel().getBtnTest().addActionListener(e -> onClickBtnTest());
    }

    @Override
    public void initializeForm() {
    }

    private void onClickBtnTest() {
    }
}
```

Schemat klasy ‘Event’ związanej z ‘form’em’

Struktura klas controller’a - klasa Event - związana z frame’em

Implementacja klasy ‘Event’ związanej z ‘frame’em’ wymaga trzymania się następujących założeń:

- Klasa Event’u powinna implementować interfejs **‘IEvent’** oraz jej metodę **‘initializeEventHandlers’ odpowiedzialną za obsługę zdarzeń na ‘frame’ie’**;
- W klasie ‘Event’u’ wyróżnia się metodę ‘control()’, **wywoływaną w miejscu tworzenia obiektu i odpowiedzialną za wywołanie metody ‘open...Frame()’**;
- **W odróżnieniu do klasy ‘Event’u’ związanej z ‘form’em’, w klasie ‘Event’u’ związanej z w ‘frame’em’ umieszcza się implementację metody ‘open...Frame()’, która odpowiedzialna jest za tworzenie i otwieranie okna (jeśli zostało już otwarte, przenoszone jest na wierzch);**
- W klasie tej powinno się w miarę możliwości trzymać konwencji związanej z umiejscowieniem odpowiednich metod we właściwych miejscach oraz ich nazewnictwem wzorując się na istniejących już klasach - i tak, należy umiejscawiać w klasie najpierw metody związane z przeładowaniem (np. ‘reload’, ‘refresh’), następnie te z obsługą zdarzeń (np. ‘onClickBtn…’) oraz związane z walidacjami (np. ‘validate…’).

TestEvent.java

```java
package com.javafee.elibrary.core.startform;

import com.javafee.elibrary.core.common.action.IEvent;

public class TestEvent implements IEvent {
    private TestFrame testFrame;

    public void control(TestFrame testFrame) {
        openTestFrame();
    }

    @Override
    public void initializeEventHandlers() {
        testFrame.getTestPanel().getBtnTest().addActionListener(e -> onClickBtnTest());
    }

    private void openTestFrame() {
        if (testFrame == null || (testFrame != null && !testFrame.isDisplayable())) {
            testFrame = new TestFrame();
            initializeEventHandlers();
            testFrame.setVisible(true);
        } else
            testFrame.toFront();
    }

    private void onClickBtnTest() {
    }
}
```

Schemat klasy ‘Event’ związanej z ‘frame’em’

TestEvent.java

```java
package com.javafee.elibrary.core.startform;

import com.javafee.elibrary.core.common.action.IActionForm;
import lombok.Setter;

public class TestEvent implements IActionForm {
    @Setter
    private TestForm testForm;

    private static TestEvent testEvent;

    private TestEvent(TestForm testForm) { this.control(testForm); }

    public static TestEvent getInstance(TestForm testForm) {
        if (testEvent == null)
            testEvent = new TestEvent(testForm);
        return testEvent;
    }

    public void control(TestForm testForm) {
        setTestForm(testForm);
        initializeForm();

        testForm.getTestPanel().getBtnTest().addActionListener(e -> onClickBtnTest());
    }

    @Override
    public void initializeForm() {
    }

    private void onClickBtnTest() {
    }
}
```

Schemat klasy ‘Event’ związanej z **‘form’em’**

TestEvent.java

```java
package com.javafee.elibrary.core.startform;

import com.javafee.elibrary.core.common.action.IEvent;

public class TestEvent implements IEvent {
    private TestFrame testFrame;

    public void control(TestFrame testFrame) {
        openTestFrame();
    }

    @Override
    public void initializeEventHandlers() {
        testFrame.getTestPanel().getBtnTest().addActionListener(e -> onClickBtnTest());
    }

    private void openTestFrame() {
        if (testFrame == null || (testFrame != null && !testFrame.isDisplayable())) {
            testFrame = new TestFrame();
            initializeEventHandlers();
            testFrame.setVisible(true);
        } else
            testFrame.toFront();
    }

    private void onClickBtnTest() {
    }
}
```

Schemat klasy ‘Event’ związanej z **‘frame’em’**

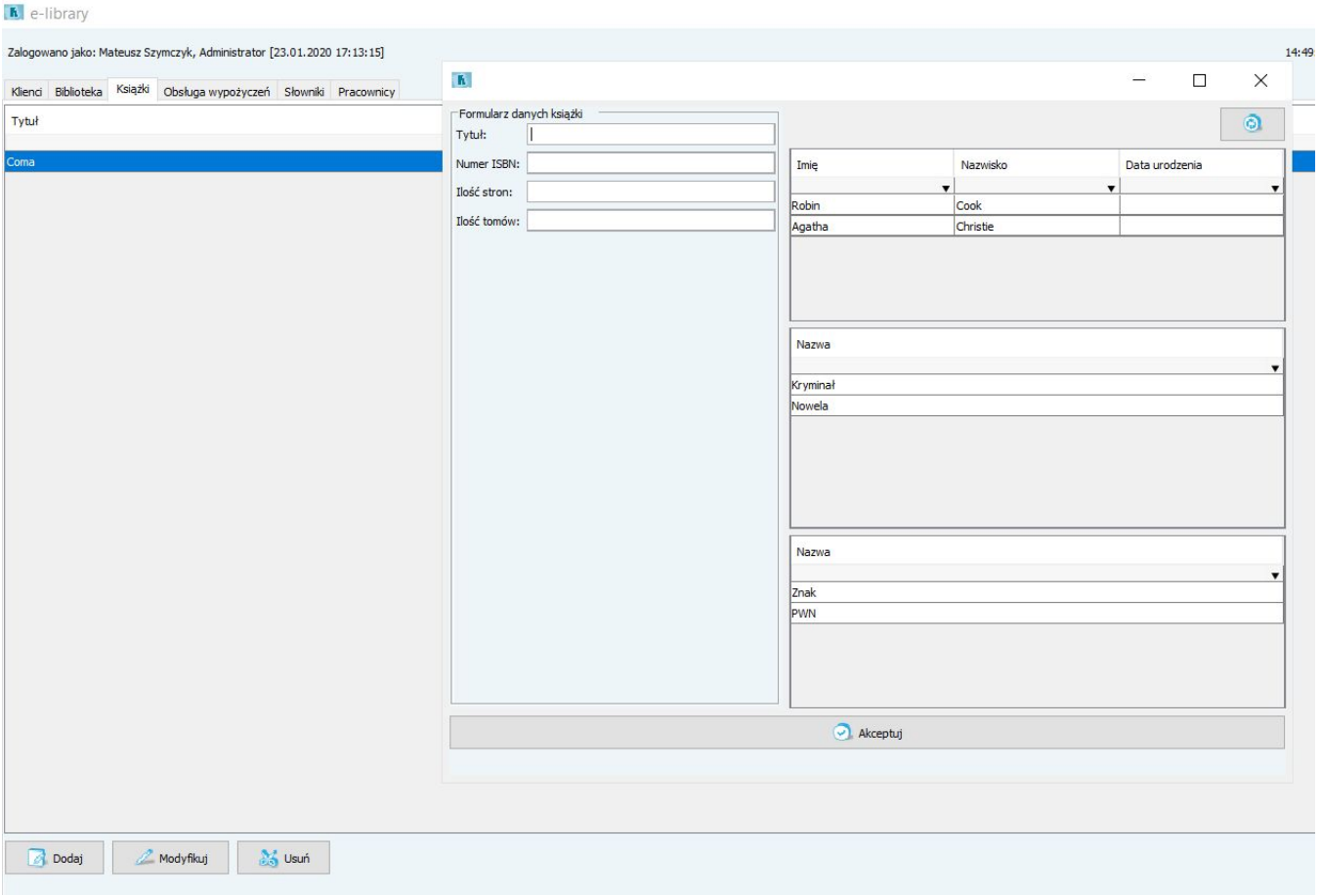
Porównanie implementacji klasy ‘Event’ dla ‘form’a’ i ‘frame’a’

Form’y a Fram’y - okna główne, a okna podrzędne

Tak jak zostało napisane na slajdzie ‘Klasa Actions’, klasa ‘Actions’ służy do obsługi zdarzeń na oknach głównych w aplikacji (**Form**). Zdarza się jednak, że część funkcjonalności związanych z oknem głównym zostaje przeniesiona do okna podrzędnego (**Frame**). Najbardziej popularnym przykładem takiego rozwiązania w aplikacji eLibrary są okna formularzy np. rejestracji klienta/pracownika na zakładkach ‘Klienci’/’Pracownicy’, czy dodawania nowego rodzaju książek (zakładka ‘Biblioteka’) lub nowych ich egzemplarzy (zakładka ‘Książki’).

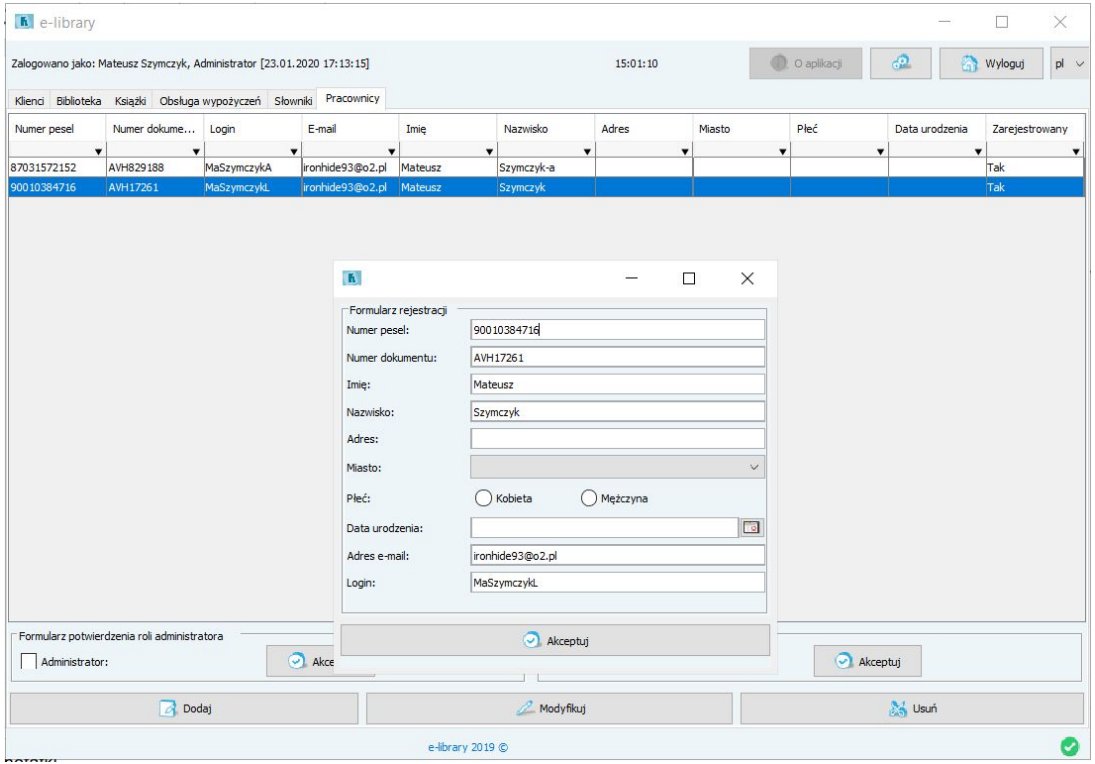
Akcje komponentów na oknach podrzędnych obsługiwane są w osobnych klasach, ewentualna komunikacja pomiędzy Form’em a Fram’em odbywa się przy pomocy klasy Params. Zwyczajowo Fram’y nie posiadają na pasku nazwy okna (w przeciwieństwie do Form’ów).

Przykład okna głównego i okna podrzędnego



Form’y a Fram’y - analiza techniczna

Na poniższych slajdach przedstawiona zostanie ogólna analiza techniczna implementacji Form’y i związanego z nią Frame’a. Przykład zostanie oparty o okno na zakładce ‘Pracownicy’ i związanym z nim oknem podrzędnym, które otwierane jest na akcję przycisku ‘Modyfikuj’ i składa się z formularza rejestracji.



Form’y a Fram’y - analiza techniczna

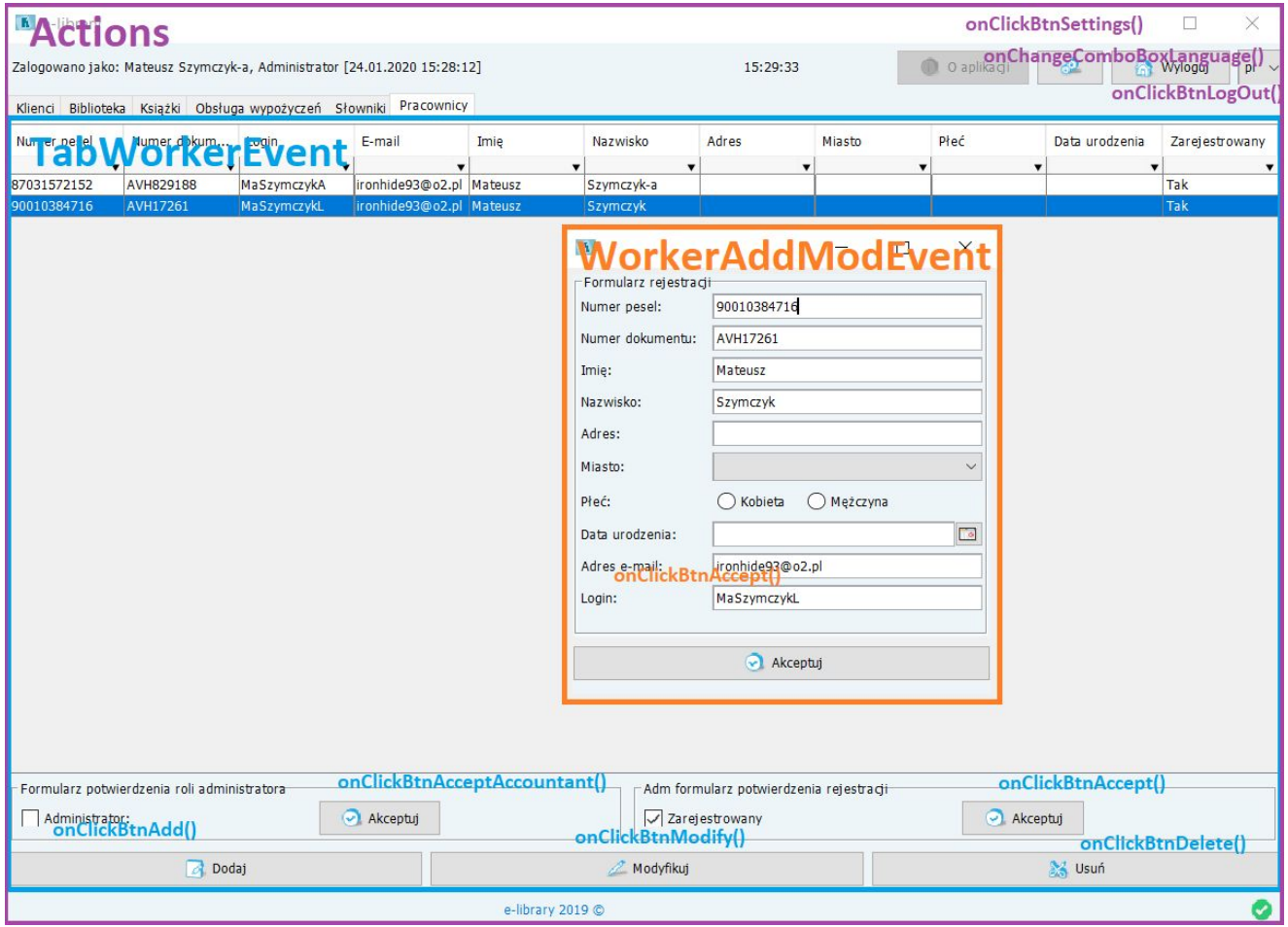
Zakładka ‘Pracownicy’ na oknie głównym (Form) obsługiwana jest przez klasę **TabWorkerEvent**. Wyświetlenie Frame’a odbywa się na akcję przycisku ‘Modyfikuj’. Akcja ta obsługiwana jest przez metodę **onClickBtnModify()**. Akcje komponentów są ‘podpinane’ w metodzie control:

```
tabbedForm.getPanelWorker().getCockpitEditionPanel().getBtnModify().addActionListener(e -> onClickBtnModify());
```

W metodzie onClickBtnModify, otwarcie okna podrzędnego następuje poprzez wywołanie metody **control** klasy, która obsługuje akcje na otwieranym Frame’ie - **WorkerAddModEvent**.

```
if (workerAddModEvent == null)
    workerAddModEvent = new WorkerAddModEvent();
workerAddModEvent.control(Constants.Context.MODIFICATION,
        (WorkerTableModel) tabbedForm.getPanelWorker().getWorkerTable().getModel());
```

Wywołanie tej metody powoduje ‘oddanie kontroli klasie WorkerAddModEvent.



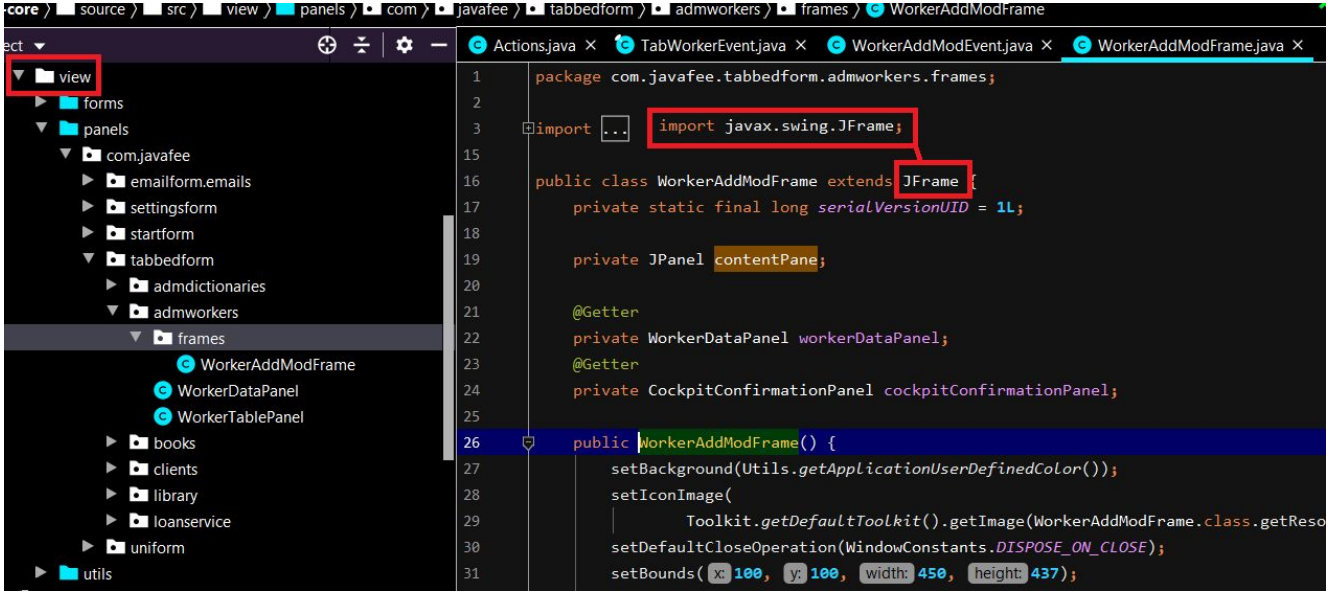
Obsługa zdarzeń - klasy i metody

Form’y a Fram’y - analiza techniczna

Po wywołaniu metody control klasy WorkerAddModEvent otwierane jest okno podrzędne, ‘podpinane’ są akcje komponentów. Dzieje się tak dlatego, że w metodzie control znajdują się następujące elementy:

- Wywołanie metody **openWorkerAddModFrame()**, która tworzy nowy obiekt typu Frame (WorkerAddModFrame z ‘modułu’ view - klasa obiektu graficznego rozszerzająca obiekt JFrame (klasa framework’u Swing)) i ustawia jego widoczność (poprzez wywołanie metody setVisible(true) (lub jeśli okno jest już otwarte toFront(true)) - okno Frame pokazuje się użytkownikowi);
- Podpięcie zdarzeń na oknie (dodanie ‘WindowListener’a’ dla Frame’u (obsługa akcji na zamknięcie okna), dodanie ‘ActionListener’a’ dla przycisku ‘Accept’).

```java
private void openWorkerAddModFrame(Context context) {
    if (workerAddModFrame == null || (workerAddModFrame != null && !workerAddModFrame.isDisplayable())) {
        workerAddModFrame = new WorkerAddModFrame();
        if (context == Context.MODIFICATION) {
            workerAddModFrame.getWorkerDataPanel().getLblPassword().setVisible(false);
            workerAddModFrame.getWorkerDataPanel().getPasswordField().setVisible(false);
            reloadRegistrationPanel();
        }
        reloadComboBoxCity();
        workerAddModFrame.setVisible(true);
    } else {
        workerAddModFrame.toFront();
    }
}
```



Form’y a Fram’y - analiza techniczna - przekazywanie parametrów między oknami

W omawianym przypadku otwierane okno podrzędne (Frame) wypełniane jest danymi, które pochodzą z zaznaczonego rekordu (wiersza) w tabeli na oknie nadrzędnym (Form). Problem sprowadza się zatem do przekazania danych (rekordu - obiektu Worker) z jednego okna do drugiego.

Rozwiązanie problemu polega na przekazaniu danych z użyciem klasy **Params**. Params stanowi klasę parametrów które reprezentowane są w postaci obiektu typu Map’y - klucz, wartość. Klasa Params pełni rolę ‘worka’ do którego ‘wrzucany’ jest obiekt na oknie nadrzędnym (Form) z przypisaniem pewnego klucza. Następnie, na podstawie tegoż klucza, obiekt pobierany jest z ‘worka’ na oknie nadrzędnym.

```
Worker selectedWorker = ((WorkerTableModel) tabbedForm.getPanelWorker().getWorkerTabl
        .getWorker(selectedRowIndex);

Params.getInstance().add("selectedWorker", selectedWorker);
Params.getInstance().add("selectedRowIndex", selectedRowIndex);

if (workerAddModEvent == null)
    workerAddModEvent = new WorkerAddModEvent();
workerAddModEvent.control(Constants.Context.MODIFICATION,
        (WorkerTableModel) tabbedForm.getPanelWorker().getWorkerTable().getModel());
```

Dodanie parametru na oknie głównym Form (‘Workers’)

```
private void fillRegistrationPanel() {
    Common.fillUserDataPanel(workerAddModFrame.getWorkerDataPanel(), (Worker) Params.getInstance().get("selectedWorker"));
}
```

Użycie dodanego parametru (obiektu typu Worker) na oknie podrzędnym Frame w celu wypełniania formularza danych rejestracji pracownika

```
workerAddModFrame.addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosed(WindowEvent e) {
        Params.getInstance().remove( key: "selectedRowIndex");
        Params.getInstance().remove( key: "selectedWorker");
    }
});
```

Usunięcie dodanego parametru po zamknięciu okna Frame

Obsługa parametrów aplikacji (SystemParameter)

W aplikacji eLibrary istnieją parametry systemowe, służące do konfiguracji systemu. Parametry systemowe przechowywane są w tabeli ‘com_system_parameter’ i obsługiwane są w aplikacji poprzez obiekt ‘SystemParameter’. Obecnie wyróżnić można m.in. następujące parametry systemowe:

- ‘APPLICATION_MAX_PASSWORD_LENGTH’ służy do określenia maksymalnej długości hasła w systemie;
- ‘APPLICATION_PENALTY_VALUE’ natomiast pozwala na zmianę wartości kary.

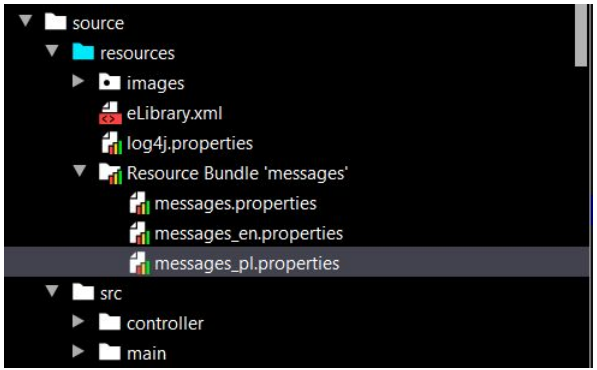
Parametry systemowe są przechowywane w obiekcie mapy w klasie ‘com.javafee.elibrary.core.common.SystemProperties’ i inicjalizowane są w metodzie ‘initializeSystemParameters()’ w momencie startu aplikacji. Poniżej podano przykład pobierania wartość parametru systemowego:

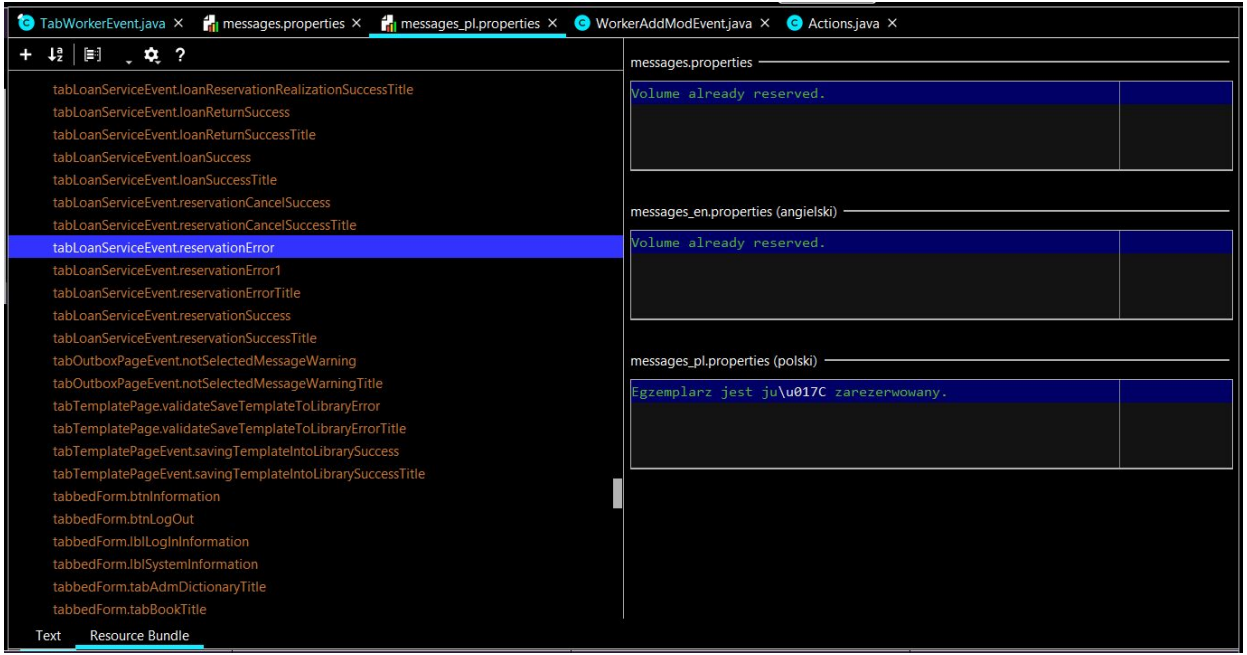
```
SystemProperties.getInstance().getSystemParameters().get(Constants.APPLICATION_MAX_PASSWORD_LENGTH).getValue()
```

Obsługa wielojęzyczności - informacje ogólne

Obsługa wielojęzyczności w aplikacji odbywa się przy użyciu plików “.properties” typu “Resource Bundle”. Wszystkie teksty występujące w aplikacji przechowywane są w wyżej wymienionych plikach. Każda wersja językowa posiada dedykowany plik którego nazwa zawiera przyrostek poprzedzony znakiem podkreślenia, będący skrótem języka. System eLibrary obsługuje polską oraz angielską wersję językową, przy czym angielska jest domyślna w plikach “.properties”. Na pliki te składają się następujące pliki:

- messages.properties - przechowuje zasoby tekstów w wersji domyślnej;
- messages_pl.properties - przechowuje zasoby tekstów w wersji polskiej;
- messages_en.properties - przechowuje zasoby tekstów w wersji angielskiej.





Lokalizacja plików “.properties” obsługi wielojęzyczności

Plik “messages.properties” otwarty przy użyciu narzędzia “Resource Bundle”

Obsługa wielojęzyczności - wprowadzanie nowych zasobów

Pliki “.properties” obsługujące wielojęzyczność aplikacji składają się z par kluczy oraz wartości. Klucze stanowią identyfikatory wprowadzonego tekstu za pośrednictwem którego wydobywa się wartość w aplikacji. Klucze zazwyczaj nawiązują do klasy w której wykorzystywany jest dany zasób tekstowy. Zazwyczaj również treść klucza skorelowana jest z komponentem którego dotyczy np. panelem i/lub przyciskiem, etykietą etc. W ten sposób na przykład treść występująca na przycisku służącym do wylogowania w pliku “.properties” związana jest z następującym kluczem:

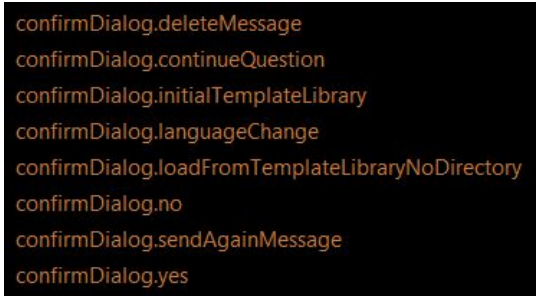
```
390    tabbedForm.btnLogOut                                    = Wyloguj
```

“tabbedForm” odnosi się do panelu na którym znajduje się przycisk stanowiąc jego nazwę, “btnLogOut” stanowi nazwę komponentu który go bezpośrednio dotyczy. Wyżej pokazanemu kluczowi przypisana jest wartość tekstu. Poniżej zaprezentowano inne przykłady:

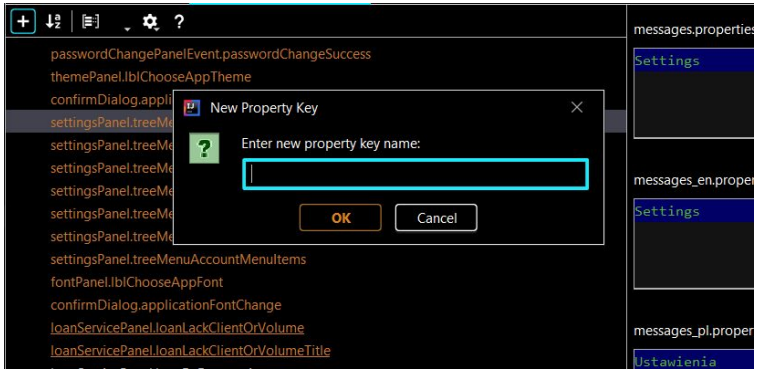
```
tabWorkerEvent.deleteWorkerSuccess                  = Pomy\u015Blnie usuni\u0119to pracownika.
tabWorkerEvent.deleteWorkerSuccessTitle             = Sukces
tabWorkerEvent.notSelectedWorkerWarning             = Pracownik nie zosta\u0142 zaznaczony. Akcja nie mo\u017Ce zosta\u0107 wykonana.
tabWorkerEvent.notSelectedWorkerWarningTitle        = Nie zaznaczono pracownika
settingsPanel.treeMenuGeneral                       = Og\u00F3lne
settingsPanel.treeMenuAccount                       = Konto
```

Obsługa wielojęzyczności - wprowadzanie nowych zasobów

W niektórych przypadkach dany klucz stosowany jest dla więcej niż jednego przypadku użycia, może to dotyczyć okien dialogowych zawierających wiadomości które wyświetlane są w więcej niż jednym miejscu w systemie, w niektórych przypadkach klucze nie nawiązują w ogóle do komponentu na którym są wykorzystywane a oddają grupę z którą są związane. Dotyczy to na przykład okna dialogowego potwierdzenia - zasoby ukazano obok na rysunku. Bywa również że zasoby są wprowadzane dla tekstów występujących w kodzie programu, lecz w jakiś sposób wpływających na interakcję z użytkownikiem końcowym.



W celu wprowadzenia nowego zasobu należy wprowadzić odpowiedni klucz oraz wartość dla wszystkich możliwych wersji językowych (najlepiej w tym celu użyć narzędzia “Resource Bundle”



Obsługa wielojęzyczności - pobieranie wartości tekstu w aplikacji

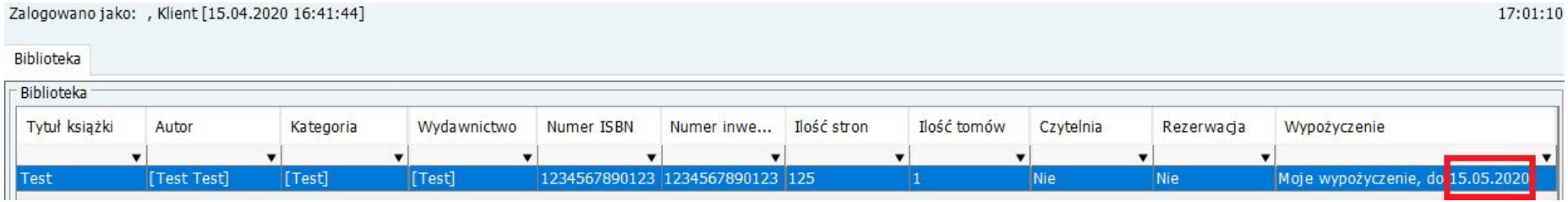
Wartości przechowywane w plikach “.properties” mogą być pobierane po kluczu z poziomu aplikacji. W klasie SystemProperties znajduje się statyczna metoda która zwraca obiekt ResourceBundle który z kolei posiada metodę getString przyjmującą parametr typu string będący kluczem odpowiedniej wartości w pliku.

```
Utils.displayOptionPane(
        SystemProperties.getInstance().getResourceBundle()
                .getString( key: "tabClientEvent.validateClientTableSelectionWarning1"),
        SystemProperties.getInstance().getResourceBundle().getString(
                key: "tabClientEvent.validateClientTableSelectionWarning1Title"),
        JOptionPane.WARNING_MESSAGE);
```

Ustawienie języka aplikacji nie jest zapisywane w baze danych aczkolwiek istnieje możliwość zmiany wersji językowej - wymaga ona przelogowania użytkownika. Możliwość personalizacji języka aplikacji może stać się elementem jej rozwoju.

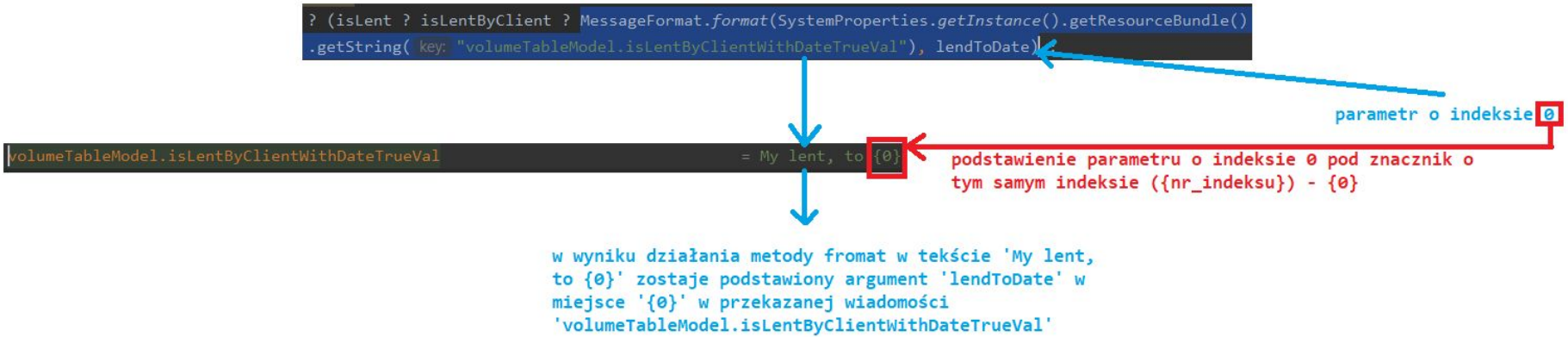
Obsługa wielojęzyczności - parametryzowane teksty

W niektórych przypadkach tekst przechowywany w opisywanych plikach “.properties” wymaga parametryzacji. To znaczy - nie składa się jedynie z predefiniowanych, stałych tekstów, lecz z pewnych zmiennych elementów, które mogą być na przykład wyliczane w trakcie działania aplikacji. Np. w kolumnie “Wypożyczenie” w tabeli wypożyczeń znajduje się informacja o tym, że dana pozycja książkowa została wypożyczona przez zalogowanego Klienta i powinna być zwrócona do danego dnia. Informacja ta jest parametryzowana - przekazywana dynamicznie do tekstu przechowywanego w “.properties”.



Obsługa wielojęzyczności - parametryzowane teksty

Przypadek opisany w ramach poprzedniego slajdu obsługiwany jest przy pomocy klasy MessageFormat i jej metody format, której przekazuje się w pierwszym argumencie tekst, który ma zostać sparametryzowany a na kolejnych miejscach argumenty które stanowią właśnie te parametry i są podstawiane pod znaczniki w przekazanym w pierwszym argumencie tekście - zgodnie z indeksami - kolejnością przekazania.



Realizacja zmian - eLibrary

Proces tworzenia zmian w systemie

Identyfikacja zadania

Wprowadzenie zadania

Realizacja

Weryfikacja

Wdrożenie

Identyfikacja zadania - informacje ogólne

Zadanie (ang. *Issue*) może zostać zidentyfikowane w wyniku np.:

- Znalezienia błędu w systemie po przeprowadzonych np. testach manualnych lub w wyniku użytkowania aplikacji;
- Identyfikacji potrzeb realizacji dodatkowych funkcjonalności w systemie;
- Realizacji potrzeb integracyjnych.

Zidentyfikowane zadania należy poddać analizie biznesowej i wstępnej analizie technicznej. Po wykonaniu ww. czynności oraz potwierdzeniu zasadności wprowadzenia zadania w systemie zarządzania zadaniami (github.com) należy zarejestrować zgłoszenie.

Wprowadzenie zadania - informacje ogólne

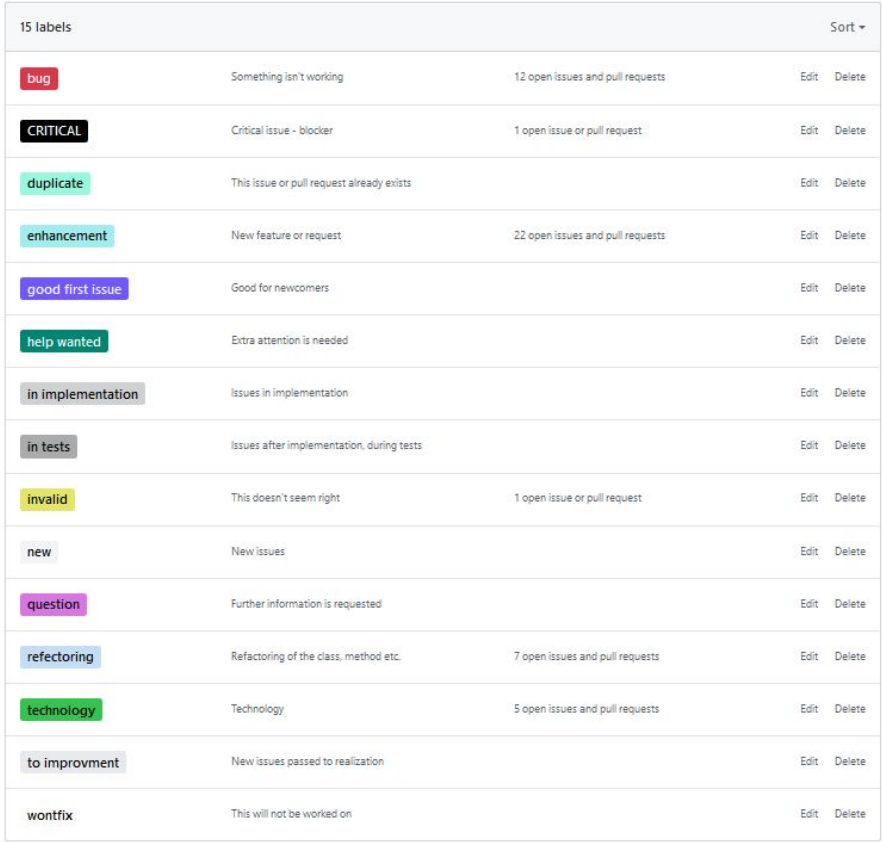
Rejestracja zgłoszeń (zadań) odbywa się za pośrednictwem platformy github.com (https://github.com/JaRutkowski/eLibrary/issues), modułu “Issues” i wymaga zachowania następujących zasad:

- Zadanie musi posiadać przyporządkowany priorytet (lider) oraz mieć zwięzły lecz oddający sens zgłoszenia tytuł;
- Zadanie musi posiadać obszerny opis, w razie potrzeby powinien zawierać również zrzuty ekrany, treści występujących wyjątków etc.;
- Zadanie musi posiadać odpowiednie etykiety;
- Zadanie musi posiadać odpowiednie przyporządkowania do projektów (lider) oraz milestone’ów (kamieni milowych);

Wprowadzenie zadania - etykiety

W systemie wyróżniono następujące grupy etykiet:

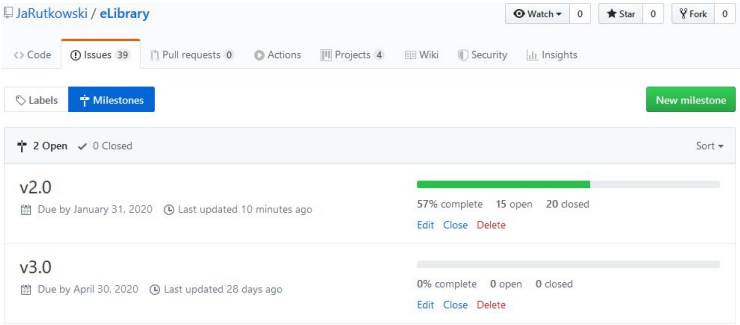
- typ zadania (bug, enhancement, refactoring, technology, good first issue, duplicate, invalid, new, CRITICAL);
- status realizacji (to improvement, in implementation, in tests);
- inne (help wanted, question, wontfix).



Wprowadzenia zadania - projekty

W systemie wyróżniono następujące grupy projektów:

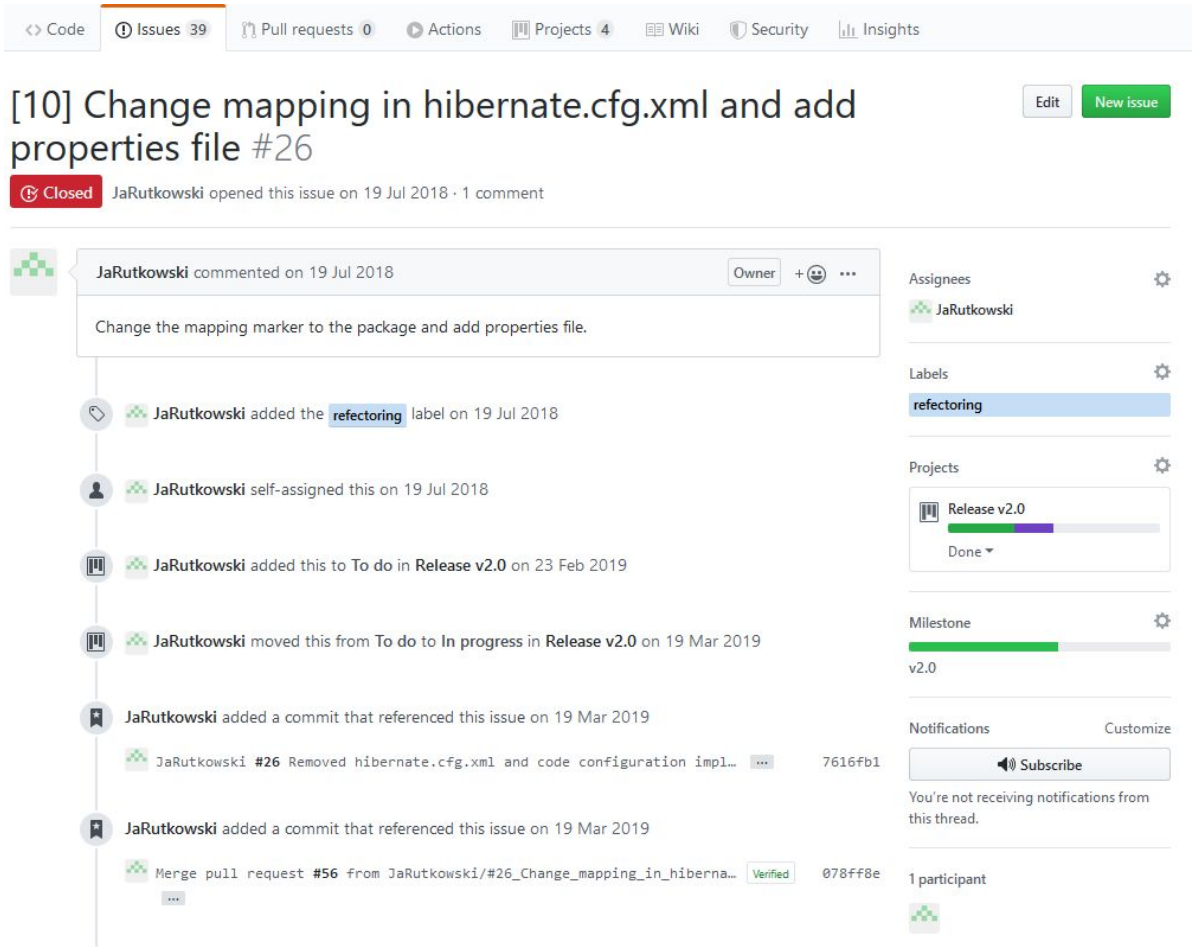
- produkcyjne;
- błędy.



Projekty produkcyjne związane są z pulą zadań do realizacji w kontekście produkcji następnych wersji oprogramowania. Ograniczone są limitem czasowym.

Projekty błędów związane są z pulą zadań typu błąd i nie mają ograniczeń czasowych. Zadania mogą należeć do wielu grup projektów i tym samym również do wielu projektów.

Projekty znajdują się z platformie github.com w sekcji “Projects”

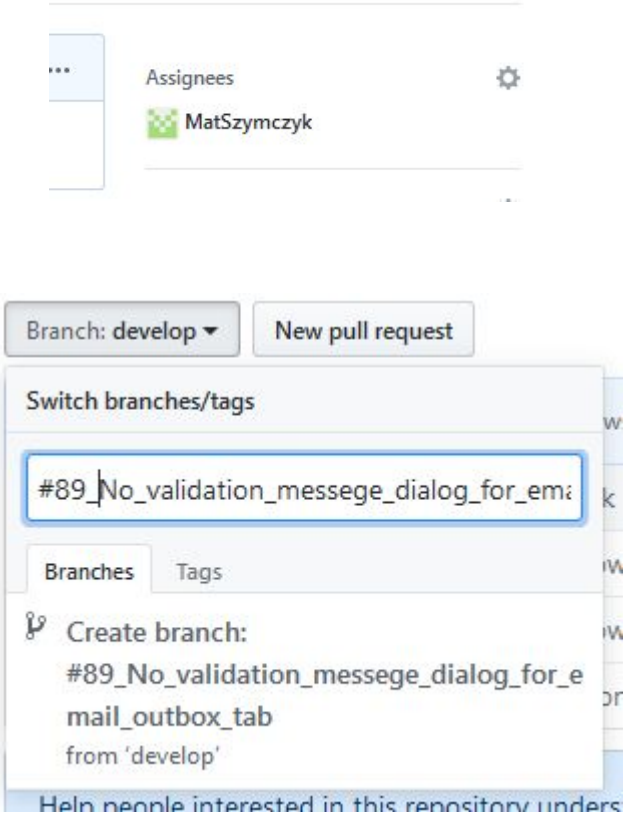


Przykład wprowadzonego zadania w systemie

Realizacja - rozpoczęcie

Realizacja zadania zaczyna się w momencie **przypisania osoby realizującej** je do zadania (Assign/Assign to me). Zadanie może zostać po konsultacji przydzielone samodzielnie, lub może zostać przydzielone przez lidera.

Praca nad zadaniem **rozpoczyna się od utworzenia nowej gałęzi** na zadanie, której tytuł stanowi numer zadania oraz jego tytuł oddzielone znakiem podkreślenia np. dla zadania o numerze #1 i tytule “[100] Do something”, nazwa branch’a będzie następująca: #1_Do_something.



Realizacja - wykonanie

Po odpowiednim rozpoznaniu zadania według opisu na platformie github.com i ew. uściśleniu zadania należy przejść do implementacji. Nie należy rozpoczynać zadania bez uprzedniego przemyślenia realizacji i zrozumienia funkcjonalności (większe zadania (najczęściej typu enhencement) wymagają wykonania odpowiednich projektów zmian, obejmujących analizę biznesową (funkcjonalną) wymagań, mock'i np. nowych okien aplikacji, projektu struktury bazy danych). Wykonując zadanie należy pamiętać o tym, że w przypadku standardowych funkcjonalności wiele przykładowych rozwiązań istnieje już w systemie, więc należy się nimi wzorować (sprzyja to utrzymaniu spójności tworzonej aplikacji - wykorzystuje się już gotowe, najczęściej sprawdzone rozwiązania). Należy jednak unikać bezmyślnego kopiowania kodu, czasem zmiany wymagają uwspólnienia części kodu.

Kod należy pisać w sposób optymalny oraz czytelny. Zmienne, nazwy metod, klas muszą być czytelne i jednoznacznie wskazywać na ich przeznaczenie. Kod musi być sformatowany, a import'y uporządkowane. Realizując zmianę należy szczególnie uważać, aby nie wpłynąć negatywnie na już istniejące funkcjonalności.

Realizacja - zakończenie

Po wykonaniu zmian należy wykonać następujące czynności:

- Skompilować kod (*mvn clean install*) oraz wykonać testy manualne realizowanej funkcjonalności w celu upewnienia się, że działa ona zgodnie z wymaganiami funkcjonalnymi opisanymi w zadaniu (kod zawsze musi się kompilować (!));
- Weryfikacja kodu i ewentualny refactoring (w tym celu można użyć polecenia konsoli *git diff* (*git diff --cached* (po dodaniu plików (*git add* *)), lub narzędzia TortoiseGIT/IntelliJ).

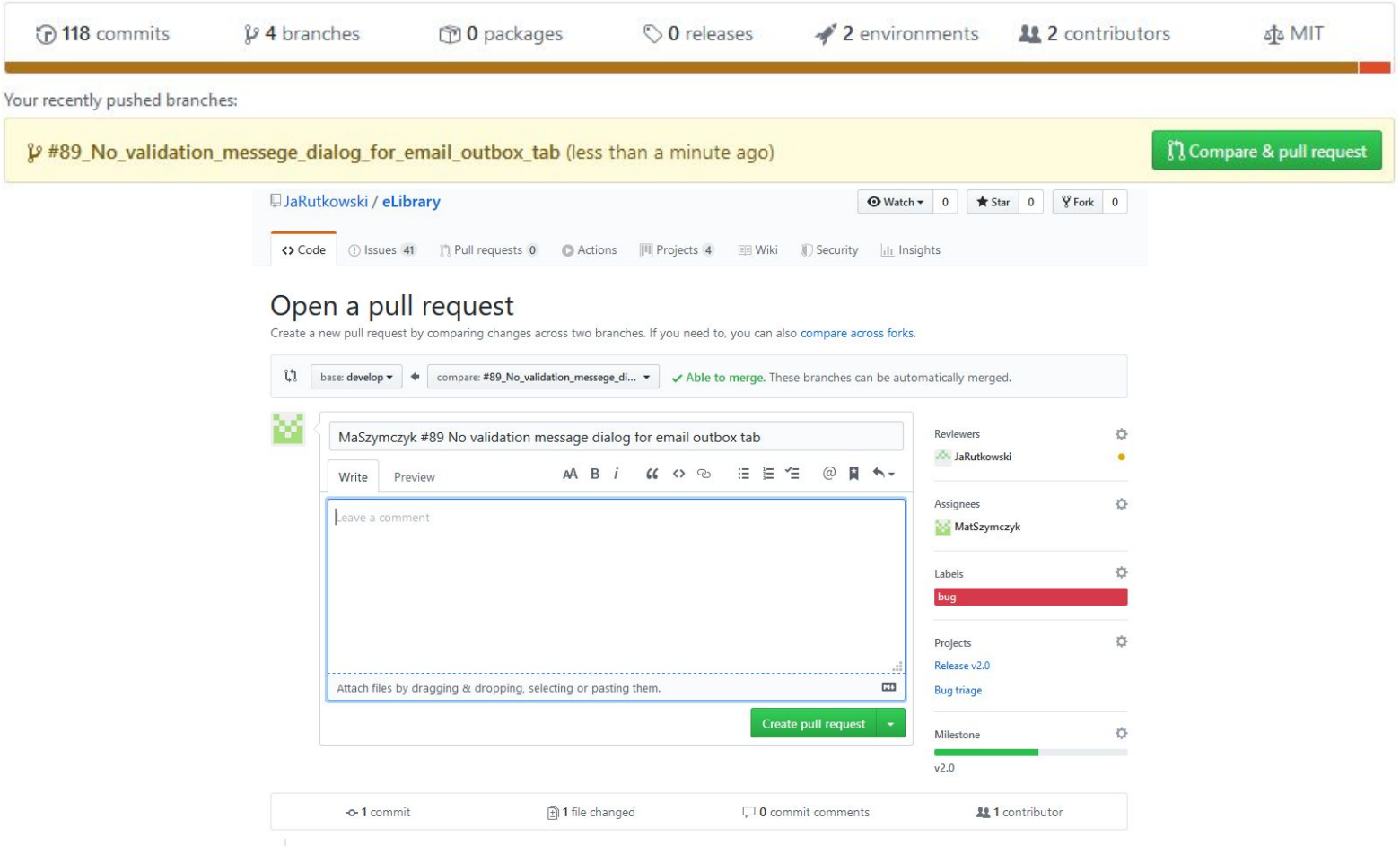
Po ich wykonaniu przechodzi się do fazy przekazywania zmiany do weryfikacji. Odbywa się to w sposób następujący.

- Należy zmianę zacommit'ować (git commit -m "JaRutkowski #1 Some bug fix." oraz git push). Commitując zmianę trzeba podać podpowiedni tytuł składający się z elementów:

  ImNazwisko #NrZad Tytuł zadania z zakładki Issue

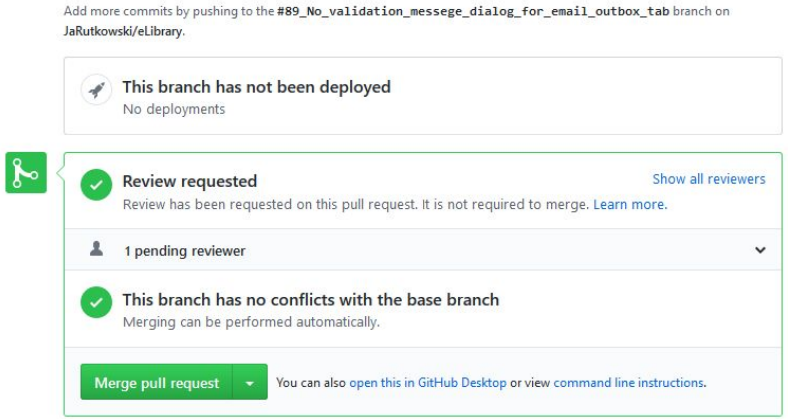
Realizacja - zakończenie

- Należy utworzyć pull request’a w celu przekazania zadania do weryfikacji. Tworząc pull requesta trzeba uzupełnić odpowiednie pola - reviewer, assignees, label, project, milestone według danych z zadania (Issue). W razie potrzeby można przekazać osobie wykonującej weryfikację komentarz.



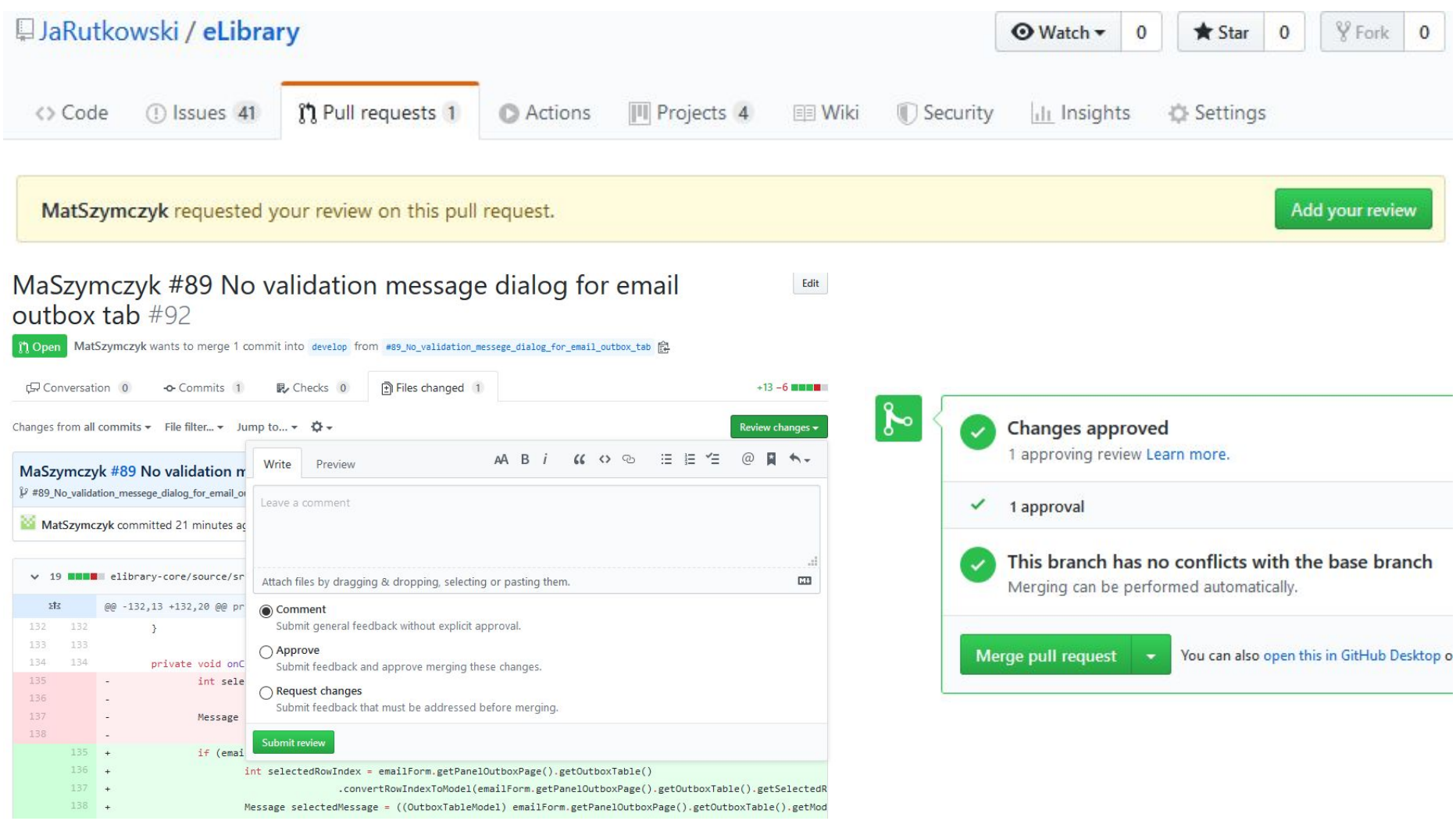
Przykład tworzenia pull request'a systemie

Weryfikacja



Weryfikacji muszą podlegać wszystkie realizowane zadania. Weryfikacja rozpoczyna się po przekazaniu pull requesta do ‘reviewer’a’. Osoba taka otrzymuje stosowną wiadomość e-mail. Weryfikacja składa się z etapów:

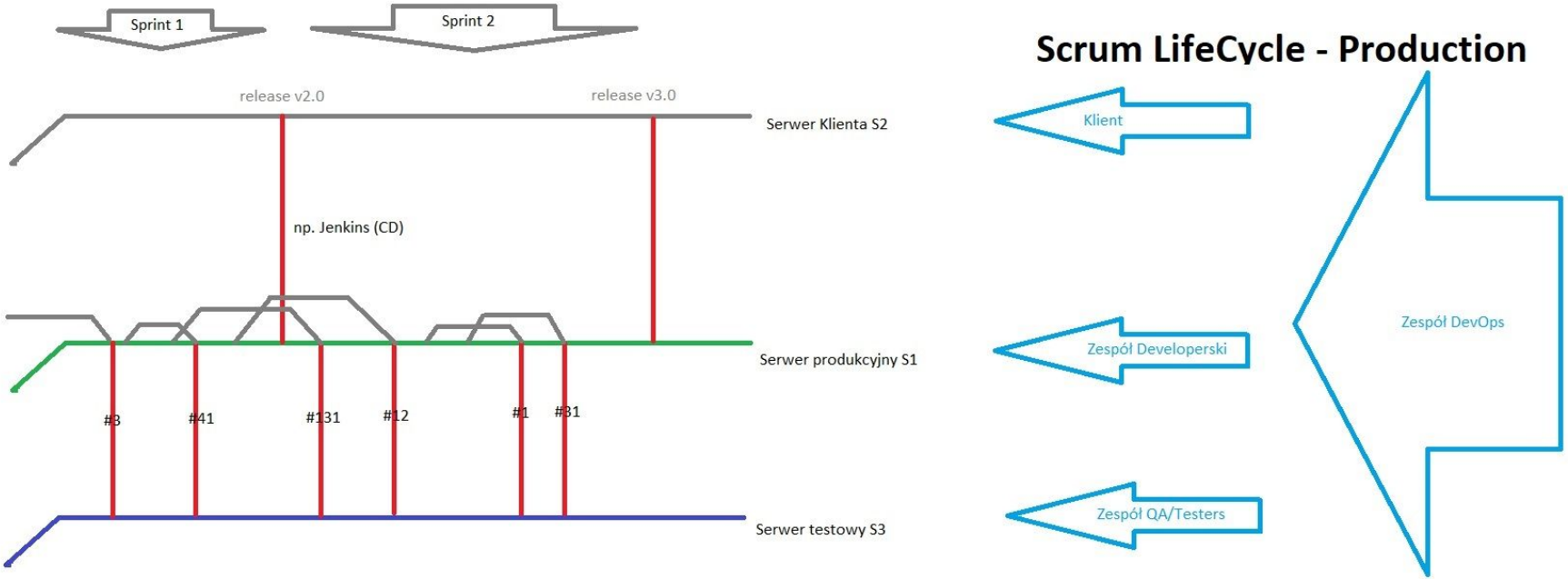
- Weryfikacja techniczna kodu (na poziomie platformy github.com należy przeglądnąć wszystkie pliki zaznaczając przy tym pole ‘view’, w przypadku uwag, należy je przekazać w komentarzach).
- Weryfikacja merytoryczna (w ramach tego etapu osoba weryfikująca zobowiązana jest do uruchomienia zmiany na swoim lokalnym środowisku i przeprowadzenia testu manualnego).
- Weryfikacja zakończyć się może akceptacją, co rozpoczyna proces wdrażania zmiany, lub zwróceniem zmiany do dalszej realizacji np. z powodu uwag dotyczących kodu, sposobu działania - niezgodnego ze specyfikacją zadania.


JaRutkowski / eLibrary
Watch 0
Star 0
Fork 0
Code
Issues 41
Pull requests 1
Actions
Projects 4
Wiki
Security
Insights
Settings
MatSzymczyk requested your review on this pull request.
Add your review
MaSzymczyk #89 No validation message dialog for email outbox tab #92
Edit
Open
MatSzymczyk wants to merge 1 commit into develop from #89_No_validation_messege_dialog_for_email_outbox_tab
Conversation 0
Commits 1
Checks 0
Files changed 1
+13 −6
Changes from all commits
File filter...
Jump to...
Review changes
Write
Preview
Leave a comment
Attach files by dragging & dropping, selecting or pasting them.
Comment
Submit general feedback without explicit approval.
Approve
Submit feedback and approve merging these changes.
Request changes
Submit feedback that must be addressed before merging.
Submit review
MaSzymczyk committed 21 minutes ag
19
elibrary-core/source/sr
@@ -132,13 +132,20 @@ pr
}
private void onC
int sele
Message
if (emai
int selectedRowIndex = emailForm.getPanelOutboxPage().getOutboxTable()
.convertRowIndexToModel(emailForm.getPanelOutboxPage().getOutboxTable().getSelectedR
Message selectedMessage = ((OutboxTableModel) emailForm.getPanelOutboxPage().getOutboxTable().getMod
Changes approved
1 approving review Learn more.
1 approval
This branch has no conflicts with the base branch
Merging can be performed automatically.
Merge pull request
You can also open this in GitHub Desktop o

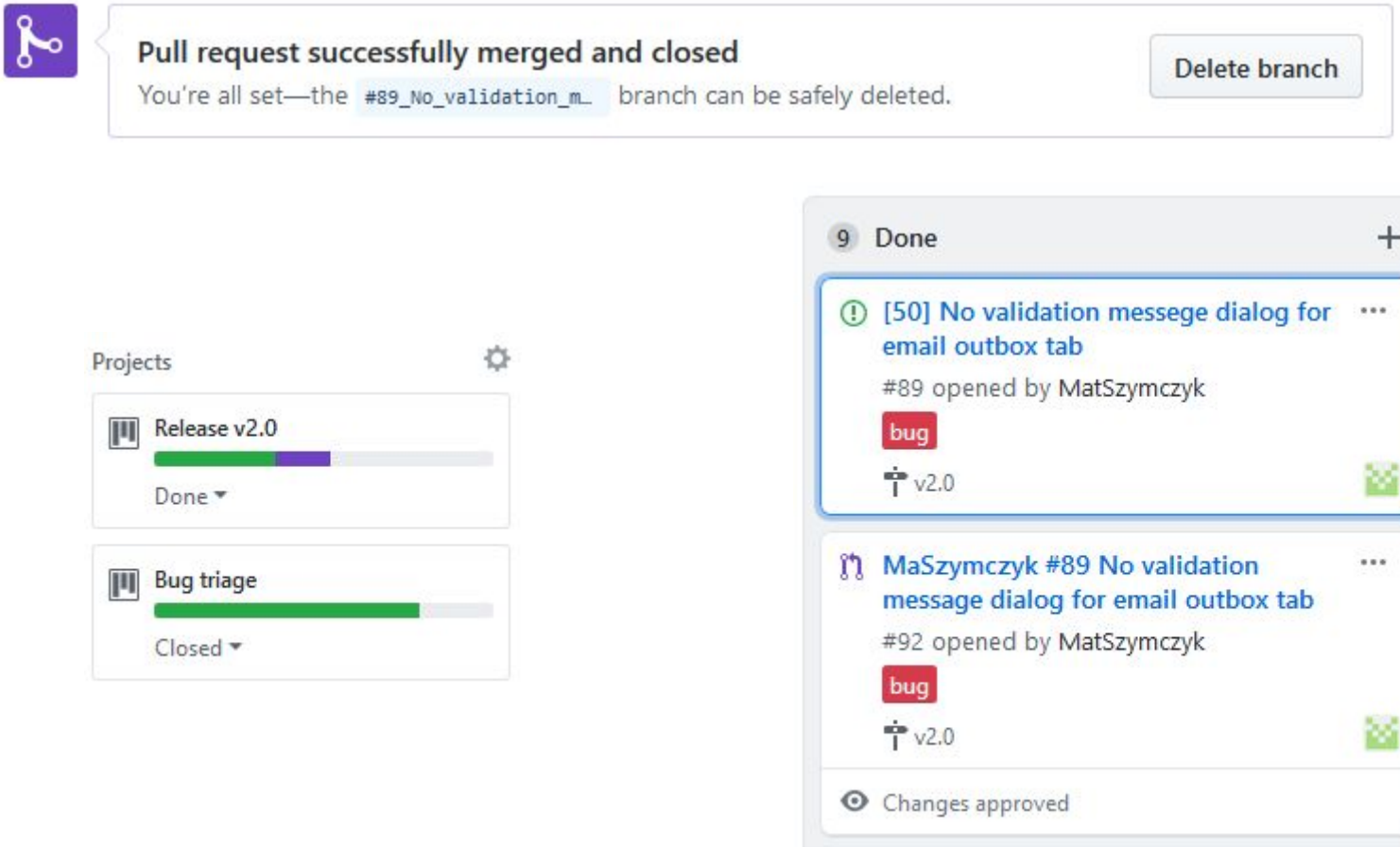
Wdrażanie

Wdrażanie zmiany polega na wykonaniu merge'u (wykonywany jest on zawsze przez weryfikatora (!)). W jego wyniku gałąź na której wykonywana była zmiana zostaje scalona z gałęzią produkcyjną (gałąź produkcyjna dla środowiska developerskiego to gałąź 'develop', mogą jednak istnieć inne gałęzie - na tym samym bądź innych serwerach - przeznaczone np. dla dostarczania wersji Klientowi). Czynność ta rozpoczyna etap wdrażania zmiany w zależności od skomplikowania procesu ciągłej integracji (ang. *continuous integration - CI*) i ciągłego dostarczania (ang. *continuous delivery - CD*) uruchamiając kaskadę czynności. W przypadku systemu eLibrary używane jest narzędzie heroku budujące wersję po każdym merge'u do gałęzi develop. Zagadnienie to zostanie opisane w ramach następnych slajdów (Temat: XXX).

Wdrożenie zmiany powinno być uzupełnione o wykonanie testów na zbudowanej wersji na gałęzi develop (testy integracyjne). Po wykonaniu wdrożenia należy pamiętać o zmianach statusów wykonywanych zmian oraz o zmian labal'i (np. z 'in progress' na 'in tests'). Zmerge'owane gałęzie są usuwane przez osobę wykonującą review - merge'ującego (istnieje możliwość przywrócenia usuniętych gałęzi).



Przykładowy schemat poglądowy cyklu produkcyjnego oprogramowania wytwarzanego w metodyce Scrum


Pull request successfully merged and closed
You're all set—the #89_No_validation_m… branch can be safely deleted.
Delete branch
Projects
Release v2.0
Done
Bug triage
Closed
9 Done
[50] No validation messege dialog for email outbox tab
#89 opened by MatSzymczyk
bug
v2.0
MaSzymczyk #89 No validation message dialog for email outbox tab
#92 opened by MatSzymczyk
bug
v2.0
Changes approved