PAŃSTWOWA WYŻSZA SZKOŁA ZAWODOWA
W NOWYM SĄCZU

INSTYTUT TECHNICZNY

PRACA DYPLOMOWA

SYSTEM GENEROWANIA RAPORTÓW W PROCESIE REKRUTACJI KANDYDATÓW NA STUDIA PROWADZONE W PWSZ W NOWYM SĄCZU.

Autor: Paweł Mysiński

Kierunek: Informatyka

Nr albumu: 20747

Promotor: dr inż. Antoni Ligęza

NOWY SĄCZ 2016

Spis treści

1. Wprowadzenie

Dokumenty, listy, protokoły to nierozłączna część każdego procesu rekrutacji na uczelni wyższej. Ręczne tworzenie takich dokumentów może nastręczać wielu problemów. Przede wszystkim głównym problemem jest możliwość wystąpienia błędów człowieka. W nawet najlepiej zorganizowanej placówce szkolnej może wkraść się błąd, który spowoduje iż na przykład kandydat zostanie przypadkowo odrzucony lub przyjęty na studia. Usuwanie takich błędów może być czasami nie możliwe, dlatego ręczne tworzenie takich dokumentów zajmuje wiele czasu, aby mieć pewność, by nie popełnić błędu. Przy czym istotnym jest, że rekrutacja powinna przebiegać szybko ze względu na fakt, iż kandydaci potrzebują znać decyzję o przyjęciu w miarę szybko aby móc w razie czego zgłosić się do innych szkół w tym samym okresie. Dodać jeszcze należy, że podczas nieefektywnie zorganizowanej rekrutacji, w procesie tym udział muszą wziąć pracownicy dydaktyczni, którzy muszą poświęcać swój czas na okres rekrutacji.

1.1. Zagadnienie generowania raportów

Z rozwiązaniem tego problemu przychodzi informatyzacja procesu rekrutacji. Pozwala ona na eliminacje praktycznie wszystkich błędów człowieka poprzez automatyczne korekty danych oraz auto uzupełnianie dokumentów danymi. Przyspiesza proces rekrutacji poprzez ułatwienie go dla osób prowadzących go oraz fakt iż dokumenty generowane są w bardzo krótkim czasie.

W procesie generowania raportów zawsze bierze udział pewien rodzaj bazy danych, z której pobierane są informacje. Dane są poddawane selekcji a następnie wklejane odpowiednie miejsce we wcześniej przygotowany szablon danego raportu. W ten sposób powstaje dokument wygenerowany z dynamicznych danych, zapisany w różnych formach. Najpopularniejszą formą zapisu jest *Portable Document Format* w skrócie *PDF*. Dokument wygenerowany do tego formatu pozwala na natychmiastowe przejrzenie zawartości oraz możliwość wydruku.

1.2. Dotychczasowy proces generowania raportów

Po przeanalizowaniu obecnego systemu można stwierdzić, iż jest on mało wydajny i wymaga poprawy. Do generowanie dokumentów używana jest funkcja programu *IBExpert*, *Report Manager*. Stworzenie szablonu raportu w tym podprogramie jest procesem żmudnym oraz skomplikowanym dla osoby zajmującej się administracją bez wiedzy informatycznej. Dodatkowo należy dodać, że nie posiada on wystarczającej funkcjonalności, przez co proces generowania raportów nie przebiega w pełni automatycznie i wymaga pełnego nadzoru osoby wdrożonej w ten system. Dobrym przykładem na pokazanie niedoskonałości jest fakt, iż każdy raport musi być generowany oddzielnie dla każdego kierunku studiów, stopnia czy też formy. Wytwarza to problemy związane z powstawaniem błędów czy też segregacją raportów.

1.3. Cel i zakres pracy

Celem pracy, czyli informatyzacji procesu rekrutacji, jest zapewnienie pracownikom administracji jak i również osobom upoważnionym, dostępu do narzędzia użytecznego, bezpiecznego oraz prostego w stosowaniu, które w znacznym stopniu przyspieszy ten proces oraz zapewni minimalizację błędów, które mogę powstać podczas tworzenia dokumentów potrzebnych przy rekrutacji.

Osiągnięcie postawionego celu, wymaga wykonania następujących zadań:

1. Wybranie systemu generowania dokumentów z wcześniej uzupełnionych szablonów.

2. Wybranie języka programowania oraz środowiska tego języka do zaimplementowania programu, który będzie stanowił interfejs dla użytkownika oraz wykonywał algorytm uzupełniania szablonów raportów.

3. Utworzenie szablonów dokumentów w procesie rekrutacji odpowiadających tym, które są obecnie używane.

4. Zaimplementowanie programu zdolnego uzupełnić wykonane wcześniej szablony oraz zapewnić łatwe użytkowanie.

5. Przeprowadzenie testu działania systemu na wyczerpującej potrzeby liczbie kandydatów.

6. Przygotowanie dokumentacji oraz instrukcji obsługi systemu.

2. Szablony raportów w systemie LaTeX

W tym rozdziale przedstawiony zostanie proces przygotowania szablonów dokumentów potrzebnych przy rekrutacji na uczelni PWSZ Nowy Sącz. Opisane zostaną tylko problemy wynikające z tworzenia automatycznie uzupełnianych szablonów oraz ich rozwiązania.

2.1. Idea tworzenia szablonów

Do wszystkich tych dokumentów potrzebny jest szablon w języku oprogramowania do zautomatyzowanego składu tekstu. W tej pracy został wybrany program LaTeX ze względu na jego możliwości automatyzacji procesu parsowania danych i uzupełniania nimi danych miejsc w tekście.

Stworzenie szablonów polega, więc na wcześniejszym przygotowaniu plików tex, zawierających wcześniej strukturę danego dokumentu z "pustymi"miejscami do uzupełnienia przez program. Do uzupełnienia tych miejsc można wykorzystać funkcję LaTeXu jaką jest tworzenie nowych środowisk z parametrami, gdzie odpowiednio parametry te będą wartościami, które zostaną wpisane w dane miejsce w danym dokumencie. Następnie wystarczy wywołać dane środowisko z odpowiednimi wartościami aby otrzymać uzupełniony dokument. Dane środowisko możemy wywoływać wielokrotnie od różnych wartości tworząc w ten sposób wiele dokumentów tego samego typu o różnych zmiennych wartościach takich jak np imię i nazwisko.

Do wytworzenia wywołań tych środowisk posłuży właśnie program stworzony w javie. Poprzez dodanie zapytania SQL w odpowiedniej formule do plików tex. Program *DBLatexRaport* wyszuka takie zapytanie i uzupełni szablon wywołaniami środowisk z wartościami parametrów, jakimi będą wartości pola z rekordów zapytania SQL.

2.2. Środowisko kompilacji raportów

Środowisko do kompilacji dokumentów zostało specjalnie przygotowane poprzez usuniecie nadmiarowych (nieużywanych) bibliotek. Dzięki temu cały system będzie zajmował mniej pamięci na dysku twardym i będzie łatwiejsze do przenoszenia. Dodatkowo kompilatora nie trzeba instalować, dzięki czemu cały system będzie szybki w użyciu. Środowisko kompilacji wymaga systemu operacyjnego Windows. Środowisko to znajduje się w załączniku razem z programem.

2.3. Tworzenie szablonów raportów do systemu rekrutacji

W rekrutacji na uczelnie wykorzystuje się dokumenty, które należało dokładnie odwzorować w nowym systemie. Są to następujące dokumenty:

1. protokół przekazania

2. listy potwierdzenia podjęcia studiów
3. listy rankingowe
4. listy przyjętych
5. listy nieprzyjętych
6. decyzja o przyjęciu danego kandydata
7. decyzja o nieprzyjęciu danego kandydata

Przy tworzeniu szablonów wystąpiły problemy, które należało rozwiązać. W dużej mierze problemy te powtarzają się, opisane więc zostały tylko rozwiązania tych problemów a nie każdy szablon raportu. W poniższych podsekcjach znajdują się przedstawione problemy oraz ich rozwiązania.

2.3.1. Definiowanie zmiennych

Pewne dynamicznie pobrane z bazy danych informacje powtarzają się w szablonach w wielu miejscach, zaistniała więc potrzeba zapisania tych informacji jednorazowo do zmiennych, które można wywołać w każdym miejscu w dokumencie.

Do uzyskania takiego efektu wykorzystana została poniższa procedura:

```
\long\def\parametrRekrutacyjny#1#2{\expandafter\def\csname#1\endcsname{#2}}
```

Każde wywołanie polecenia `parametrRekrutacyjny` utworzy zmienną, której nazwą będzie parametr pierwszy, natomiast wartością drugi.

Po wywołaniu przykładowych danych pokazanych poniżej:

```
\parametrRekrutacyjny{dataWydaniaDecyzji}{2015-03-06}
\parametrRekrutacyjny{miejsceWydaniaDecyzji}{Nowy Sącz}
\parametrRekrutacyjny{przewodniczacyIKR}{dr inż. Tomasz Kądziołka}
```

będzie można pobrać te zmienne za pomocą wywołania komendy o nazwie 1 parametru. Wywołanie więc `\dataWydaniaDecyzji` zwróci wartość w danym miejscu `2015-03-06`.

2.3.2. Wyświetlanie listy

Problem występujący w protokole przekazania oraz we wszystkich listach. Przykładowo potrzebujemy wyświetlić poniższą listę, gdzie oczywiście wartości pochodzą z bazy danych:

```
Lp. Tok studiów Liczba kopert
1 Informatyka -- niestacjonarne STUDIA pierwszego stopnia 1455
2 Informatyka -- stacjonarne STUDIA pierwszego stopnia 729
3 Mechatronika -- niestacjonarne STUDIA pierwszego stopnia 1447
...
```

2.3.3. Grupowanie

2.3.4. Przekazanie informacji do bazy danych

3. Projekt programu i implementacja

Ten rozdział przedstawia dokładny proces implementacji programu *DBLatexRaport,* który ma na celu spełnienia wymagań oraz celów zawartych w rozdziale 1. Osoba po przeanalizowaniu poniższego materiału, będzie w stanie w przyszłości ulepszyć o nowe funkcję istniejący już program lub stworzyć nowy, podobny program w innym języku programowania, który będzie w stanie obsłużyć istniejące już szablony.

Wyjaśnione będą decyzje oraz postępowania przy pisaniu danego kodu, aby w pełni oddać idee tworzenia tego programu. Sekcję tę, można więc potraktować jako pewien samouczek, który jednak wymaga minimalnej znajomości języka programowania *JAVA*. Pominięte też zostaną kwestie związane z importem podstawowych bibliotek, czy też wszelkie obsługi wyjątków, które mogą zaistnieć przy złej obsłudze programu. Cała uwaga zostanie skupiona tylko i wyłącznie na kodzie, przy poprawnym wykonaniu. W razie potrzeby dokładnej analizy cały kod programu znajduje się w załączniku.

Wszelki kod przedstawiony tutaj zawierający obok linie, znajduje się w programie. Numery linii odzwierciedlają dokładne położenie danego kodu w programie.

3.1. Algorytm działania systemu raportowania

W poprzednim rozdziale przedstawiony został proces tworzenia samych szablonów, natomiast w tej sekcji skupiona uwaga zostanie na tym jak zaprojektować system uzupełniania tych szablonów. Między innymi właśnie o tym jak umieszczać informacje w szablonie na temat selekcjonowania danych czy też jak wygenerować odpowiednią strukturę danych.

3.1.1. Algorytm parsowania

Aby szablon raportu został uzupełniony o potrzebne dane, musi on posiadać pewną informację o tym, co i w jakiej formie należy w nim zapisać. Program musi przeszukać szablon w celu znalezienia tej informacji i przeanalizowanie jej, aby wywołać odpowiednie procedury na rzecz danego szablonu. Jako, że informacja ta przeznaczona jest tylko dla programu przeszukującego, idealnym było by, gdyby zapis tej informacji byłby ignorowany przez środowisko kompilacji raportów. W wybranym wcześniej środowisku *Latex* znajduje się komenda `iffalse` oraz jej zamknięcie `fi` dzięki której wszystko pomiędzy zostanie zignorowane w czasie kompilacji dokumentów. Daje nam to taki zapis gdzie nasza informacja może zajmować wiele linii:

```
\iffalse
...
...
\fi
```

Jeśli polecenie jest w stanie zmieścić się w jednej linii, można użyć % aby za komentować tę linię, co przyniesie dokładnie taki sam efekt.

Kolejnym krokiem jest ustalenie struktury informacji, która powinna zawierać instrukcje do stworzenia wywołań środowisk z danymi przesłanymi jako parametry. Instrukcjami tymi

są po kolei: nazwa środowiska, grupowanie pól oraz selekcja danych. Utworzony został więc na potrzeby tego programu standardowy zapis w różnych wariantach z wykorzystaniem znaków `@` jako separatorów oraz informacji o zakończeniu instrukcji `@END@`:

1. Pusty - Polecenie do bazy danych, które nie zwraca żadnych danych.

```
\iffalse
@@@@Polecenie do bazy danych@END@
\fi
```

2. Prosty - Polecenie do bazy danych, zwracające wyselekcjonowane dane pod daną nazwą środowiska

```
\iffalse
@@Nazwa środowiska@@Selekcja danych@END@
\fi
```

3. Z grupowaniem - Polecenie do bazy danych, zwracające wyselekcjonowane dane pod daną nazwą środowiska dodatkowo z informacją o grupowaniu, która składa się z cyfr oraz przecinków.

```
\iffalse
@@Nazwa środowiska@Grupowanie@@Selekcja danych@END@
\fi
```

3.1.2. Selekcjonowanie danych

Wybieranie danych z bazy odbywać się będzie na poziomie połączenia z wybranym silnikiem bazodanowym. Oznacza to ze zapytania o dane muszą zostać napisane tak, by interpreter poleceń SQL danego silnika był w stanie je przetworzyć i wykonać, zwracając przy tym potrzebne dane. Oznacza to, że zapytanie zapisane w danej instrukcji wywoływane jest bez żadnych zmian na bazie danych z którą połączona jest aplikacja.

3.1.3. Struktura uzupełnianych danych

Środowiska utworzone w szablonach są w stanie same, za pomocą argumentów, uzupełnić dane miejsca, o daną wartość. Zadaniem programu jest utworzyć z wyselekcjonowanych danych, wywołania tego środowiska jednorazowo dla każdego rekordu pobranego z bazy danych. Wywołanie środowiska odbywa się poprzez polecenie:

```
\nazwasrodowiska{parametr1}{parametr2}{parametr3} ...
```

Taka struktura może być wynikiem prostego wariantu polecenia:

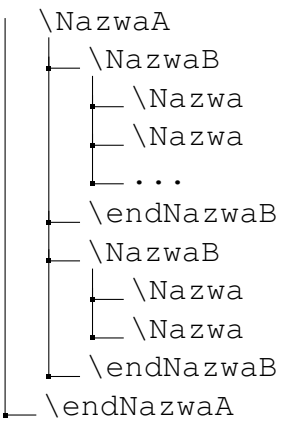
```
\iffalse@@parametrRekrutacyjny@@
SELECT klucz,wartosc FROM setup_aligeza
@END@\fi
```

Gdzie wynikiem takiego polecenia będzie właśnie:

```
\parametrRekrutacyjny{rokAkademicki}{2014/2015}
\parametrRekrutacyjny{czyUwzglednicDateWydaniaDecyzji}{N}
\parametrRekrutacyjny{instytutNazwa}{Instytut Techniczny}
```

Na potrzeby systemu rekrutacji, musiała zostać stworzona dodatkowa struktura, pełniąca funkcję grupowania. Wykorzystana może być także w przypadku gdy rekordy zwrócone z bazy danych zawierają więcej niż 9 pól, ze względu na to, że środowiska mogą być wywoływane maksymalnie od 9 argumentów. Poniżej przedstawiona zostanie tylko struktura z krótkim wprowadzeniem. Dokładny opis jej tworzenia znajduje się w implementacji.

Strukturę grupowania odzwierciedla struktura drzewa. Wywołanie środowiska z dodaną dużą literą alfabetu łacińskiego na końcu nazwy rozpoczyna grupę, natomiast wywołanie środowiska z dodaną frazą ęndńa początku nazwy, kończy daną grupę. Uwagę należy zwrócić na fakt, iż w alfabecie łacińskim jest 26 znaków, co ogranicza ilość grup do 26. Na przykładzie drzewa może wyglądać to następująco:



Od każde grupujące środowisko wymaga co najmniej 1 parametru, który zabierany jest z pól rekordów pobranych z bazy danych. Pole dla wszystkich rekordów w danej grupie jest takie samo dlatego jest ono właśnie przerzucane do wywołania środowiska grupy. Poniżej prosty przykład ukazujący dane zachowanie:

```
\nazwasrodowiskaA{pole1}
\nazwasrodowiska{pole2}{pole3}{pole4}...
\nazwasrodowiska{pole2}{pole3}{pole4}...
...
\endnazwasrodowiskaA
```

3.1.4. Wywołanie kompilacji szablonu

Ostatecznie aby wytworzyć dokument w *PDF* należy go skompilować wybranym kompilatorem. Do tego posłuży polecenie powłoki systemu Windows. Poniżej przykładowe wywołanie kompilatora LaTeX:

```
cmd /c start texlive\2010min\bin\win32\pdflatex.exe
--output-directory=output/ output/main.tex
```

3.2. Wybór języka oraz środowiska programistycznego

Pierwszym podstawowym kryterium wyboru języka programowania, w tym projekcie, jest fakt posiadania przez język gotowych bibliotek obsługujących połączenie z serwerem bazodanowym. Cała reszta wymagań takich jak obsługa operacji na plikach, operacje na łańcuchach tekstu, obiektowość języka czy też multiplatformowość schodzą na drugi plan, ze względu na to, że każdy współczesny język posiada większość podstawowych funkcjonalności.

Na uczelni wykorzystywany jest serwer bazodanowy o silniku *Firebird 2.5*. Najpopularniejsze języki programowania, które obsługują połączenie z tym serwerem to:

— JAVA
— C++
— C#
— Delphi

— Perl
— Python
— wszystkie języki obsługujące połączenie z *ODBC (Open DataBase Connectivity)*

We wszystkich powyższych językach jest wstanie powstać potrzebny program, jednak najlepszym wyborem okazał się język ***JAVA***, ze względu na wiele zalet:
— prostota importu bibliotek połączenia z serwerem bazodanowym. Dodatkowo na jednym interfejsie można obsłużyć połączenia z innymi silnikami bazodanowymi.
— multiplatformowość
— obiektowość
— wiele zaimplementowanych już funkcji, które zostaną wykorzystane w programie.
— łatwość pisania kodu
— wystarczająca wydajność na potrzeby projektu

Natomiast na środowisko w jakim powstanie projekt wybrany został program ***NetBeans 8.0.2*** ze względu wiele przydanych funkcjonalności oraz prostotę obsługi. Dodatkowo posiada on pełną dokumentację jak i wiele samouczków w Internecie.

3.3. Utworzenie projektu i jego struktury

Środowisko *NetBeans 8.0.2* posiada funkcję utworzenia projektu typu *JAVA Application*. Automatycznie stworzony zostanie package o nazwie projektu *DBLatexRaport* oraz klasa, o tej samej nazwie, zawierającą metodę `main(String[] args)`. Od tej metody program zacznie swoje wykonanie. Tak więc, w metodzie tej, także będą umieszczane deklaracje wszystkich obiektów klas głównych oraz ich inicjalizacja.

Aplikacja, oprócz swojej głównej klasy zawierającej metodę textttmain(), będzie wymagała podziału na moduły, które będą odzwierciedlane poprzez odpowiednie klasy. Wymagane do działania będą:
— klasa Config - moduł obsługi pliku konfiguracyjnego
— klasa DBHandle - moduł obsługi połączenia z bazą danych.
— klasa Templates - moduł obsługi szablonów.
— klasa LatexCompiler - moduł obsługi wywoływania kompilatora szablonów.

Podczas tworzenia programu może okazać się, że przydadzą się jeszcze dodatkowe klasy pomocnicze, przechowujące pewne dane w spójnej strukturze. Zastosowanie takich klas, znacznie ułatwi i przyspieszy pracę z danymi.Klasy te dokładnie opisane zostaną w jednej z następnych podsekcji:
— klasa ParsedSQLInfo - do przechowywania informacji o selekcji danych
— klasa RecordSet - do przechowywania pobranych danych z bazy.

3.4. Zarządzanie konfiguracją - klasa Config

Większość programów przed uruchomieniem wymaga konfiguracji. Konfiguracje można przeprowadzić na wiele sposobów jednak najodpowiedniejszym sposobem do tej aplikacji

będzie plik konfiguracyjny, wczytywany przed uruchomieniem programu. Taki plik powinien posiadać informacje, z których skorzysta program, przypisane do zmiennych nazwanych w stały sposób. Dobrym zwyczajem jest, aby plik konfiguracyjny posiadał też możliwość pisania komentarzy. Dzięki komentarzom, łatwiej jest skonfigurować dany program.

Autorskim pomysłem jest dodatkowa funkcja pliku konfiguracyjnego, jaką jest aby plik ten był jednocześnie plikiem wsadowym. Polecenia powłoki systemu rozpoczynają plik, i kończą się na poleceniu `EXIT`. Następnie od lini *#dbLatexRaportConfig* rozpoczyna się zapis zmiennych konfiguracyjnych.

Klasa ta została tak zaprojektowana by mogła przyjąć wszelkie nazwy zmiennych oraz ich wartości w postaci łańcuchów znakowych, a następnie za pomocą odpowiedniej metody można było w środku programu odwołać się do wartości danej zmiennej po jej nazwie. Dla przykładu zdeklarowanie zmiennej za pomocą jej nazwy oraz zaraz po znaku "="jej wartości:

```
user=SYSDBA
```

Następnie po utworzeniu się obiektu klasy Config i załadowaniu poprawnie pliku konfiguracyjnego powinniśmy móc pobrać wartość zmiennej `user` poprzez wywołanie odpowiedniej metody. Oczywiście dla braku zmiennej, powinna zwracać pustą wartość i informować o tym użytkownika.

```
Config cfg = new Config("dblatexraportconfig.bat");
System.out.println(cfg.getString("user"));
```

Podejściem do zaprojektowania tej klasy można uznać za metodę od ogółu do szczegółu. Na początek potrzeba zaimportować odpowiednie pakiety do obsługi plików, a następnie pozostaje już tylko ustalić składowe, zaimplementować sparametryzowany konstruktor klasy oraz metodę zwracającą daną zmienną. Składowe klasy muszą przede wszystkim przechowywać załadowane zmienne z pliku w pamięci. Łatwym sposobem realizacji tego jest użycie zmiennych tablicowych o typie *String* czyli łańcuchów znaków. Dla każdego indeksu przechowywana będzie informacja o nazwie oraz wartości zmiennej.

```
23 public  class Config {
24
25       String[] strindex;
26       String[] value;
```

Następnie konstruktor klasy z jednym parametrem posłuży jako metoda otwarcia pliku i załadowania wszystkich zmiennych. Parametrem będzie nazwa pliku konfiguracyjnego. Konstruktor klasy w *JAVA*'e musi nazywać się tak jak klasa. Po uruchomieniu konstruktora otwieramy plik wykorzystując dostępne klasy obsługi plików.

```
39 public Config(String path){
40  File cfg = new File(path);
41  BufferedReader in = new BufferedReader(
42  new InputStreamReader(
43  new FileInputStream(cfg), "UTF-8"));
```

Do odczytu przygotować należy pewien bufor oraz zmienne pomocnicze:

```
55  String tmp;
56  String[] strindextmp = new String[50];
57  String[] valuetmp = new String[50];
58  int count=0;
59  int flag = 0;
```

Następnie potrzebna jest pętla odczytująca plik linia po linii. Pętla zakończy się po odczytaniu całego pliku, przekazując odczytaną linie do jednej iteracji w pętli. Wewnątrz pętli pierwszym krokiem jest sprawdzenie czy linia zawiera flagę `#dbLatexRaportConfig` by móc zacząć odczytywać zmienne poprzez ustawienie zmiennej flag na wartość 1. Pozostaje już tylko sprawdzić czy linia nie jest pusta oraz czy nie jest komentarzem, aby móc

zapisać zmienne poprzez wyłączenie z linii nazwy (cała linia do znaku "=") oraz wartości zmiennej (cała linia od znaku "=") do aktualnego indeksu bufora. Ostatecznie potrzeba zwiększyć indeks bufora o 1 i zakończyć iteracje.

```
60 while ((tmp = in.readLine()) != null) {
61  if(flag == 0 && tmp.equals("#dbLatexRaportConfig"))
62   flag = 1;
63  if(flag == 1 && tmp.indexOf('#') != 0
64  && tmp.length() != 0 && tmp.indexOf('=') != -1){
65   strindextmp[count] = tmp.substring(0,tmp.indexOf('='));
66   valuetmp[count] = tmp.substring(tmp.indexOf('=')+ 1,tmp.length());
67   count++;
68  }
69 }
```

Na koniec zamykany jest plik i alokowana jest pamięć o dokładnie *count* elementów dla składowych klasy. Ostatecznie przepisywany jest cały bufor do tych składowych.

```
70 in.close();
71
72 strindex = new String[count];
73 value = new String[count];
74
75 for(int i=0;i < count; i++){
76  strindex[i] = strindextmp[i];
77  value[i] = valuetmp[i];
78 }
```

Pozostaje zaimplementowanie metody zawracającej wartość zapamiętanej zmiennej. Do tego posłuży metoda o nazwie `getString` i parametrze `String name`. Parametr jest kluczem do wartości zmiennej, która zostanie zwrócona. Aby znaleźć tą wartość należy przejrzeć tablicę nazw zmiennych i porównać każdą wartość z nazwą szukanej zmiennej. Jeśli znajdziemy odpowiednik oznaczać to będzie, że pod tym samym indeksem w tablicy wartości znajduje się szukana zmienna, którą zwróci metoda. W przypadku braku zmiennej metoda zwróci pustą wartość i wyświetli informacje o jej braku.

```
169 public  String getString(String name){
170  for(int i=0;i < value.length; i++){
171    if(strindex[i].indexOf(name) == 0)
172     return(value[i]);
173   }
174  System.out.print("CONFIG VARIABLE ERROR: " + name + "\n");
175  return("");
176 }
```

3.5. Klasy typu kontener

Przed przystąpieniem do tworzenia reszty głównych klas programu, potrzebne będzie stworzenie dwóch klas pomocniczych zawierających pewną strukturę danych oraz interfejs. Posłużą one do zapewnienia łatwej wymiany danych pomiędzy modułami.

3.5.1. Klasa ParsedSQLInfo

Pierwszą taką klasą będzie struktura reprezentująca informację o selekcji danych, wczytaną z szablonu. Klasa zostanie nazwana `ParsedSQLInfo` i zawierać będzie następujące składowe wyczerpujące wszelkie informacje na temat selekcji:

```
11 public class ParsedSQLInfo {
12  String query;
13  String name;
14  String group;
15  int index;
16  String data;
```

Odpowiednio:

— `query` - polecenie SQL
— `name` - nazwa środowiska.
— `group` - grupowanie
— `index` - miejsce zapisu do pliku
— `data` - wygenerowane dane dla tego zapytania.

Do klasy został stworzony konstruktor sparametryzowany, który przyda się przy tworzeniu obiektu:

```
21 public ParsedSQLInfo(String query, String name,String group, int index) {
22  this.query = query;
23  this.name = name;
24  this.group = group;
25  this.index = index;
26  this.data = "";
27 }
```

Oraz podstawowy interfejs do pobrania lub zmiany danych:

```
30 public int getIndex() {return index;}
31 public String getQuery() {return query;}
32 public String getName() {return name;}
33 public String getGroup() {return group;}
34 public String getData() {return data;}
35 public void setIndex(int index) {this.index = index;}
36 public void setQuery(String query) {this.query = query;}
37 public void setName(String name) {this.name = name;}
38 public void setGroup(String group) {this.group = group;}
39 public void setData(String data) {this.data = data;}
```

3.5.2. Klasa RecordSet

Druga klasa typu kontener (`RecordSet`) przechowywać będzie wyniki zapytań SQL w postaci listy rekordów. Każdy rekord natomiast będzie reprezentowany jako tablica zmiennych typu String o danej długości. Składowe wyczerpujące wszelkie informacje wyglądać będą następująco:

```
13 public class RecordSet {
14
15 ArrayList<String[]> val;
16 int m;
```

Jak w poprzedniej klasie, zaimplementowany został interfejs oraz konstruktor służący do zainicjowania obiektu ilością kolumn w rekordach:

```
18 RecordSet(int columncount){
19 val = new ArrayList<String[]>();
20 m = columncount;
21 }
```

Interfejs wymagał by łatwo można było nadpisać daną wartość oraz pobrać w zależności od numeru rekordu oraz numeru pola. Zwrócić uwagę należy na to iż przy zapisie nowych rekordów, lista automatycznie się poszerzy. Dodatkowo dodana jest możliwość pobrania ilości rekordów czy też pól w 1 rekordzie.

```
23 void setVal(String str,int i, int j){
24  while(val.size() <= i)
25   val.add(new String[m]);
26  val.get(i)[j]=str;
27 }
28 String getVal(int i, int j){return(val.get(i)[j]);}
29
30 int get_rows(){return(val.size());}
31 int get_cols(){return(m);}
```

3.6. Obsługa połączenia z bazą danych - klasa DBHandle

Stworzenie klasy obsługującej połączenie z bazą danych oraz wywołania poleceń na tej bazie. wymagało będzie zaimportowania dodatkowych bibliotek. W języku *Java* znajduje się moduł *JDBC(TM) Database Access*, który zostanie wykorzystany w implementacji tej klasy. Dokładna dokumentacja tego modułu znajduje się na stronie *http://www.cs.mun.ca/ michael/java/jdk1.1.5-docs/guide/jdbc/index.html*. Poniżej przedstawiona zostanie jedynie idea zastosowanie tego rozwiązania oraz implementacja wykorzystująca ten moduł.

3.6.1. Wykorzystanie JDBC(TM) Database Access

Dzięki wykorzystaniu *JDBC(TM) Database Access* obsługa połączeń z wieloma rodzajami baz danych staje się prosta. Idea tego rozwiązania jest taka aby dla każdego rodzaju bazy danych interfejs połączenia był dokładnie taki sam. Interfejs ten dostępny jest przez klasę `java.sql.DriverManager`, pozwalającą załadować odpowiednią bibliotekę dla wybranego silnika bazodanowego, zwracając obiekt z takim samym interfejsem dla każdej możliwej bazy danych.

W przypadku pracy z serwerem *Firebird 2.5* potrzebne będzie jeszcze zaimportowanie do projektu odpowiedniego pliku *jaybird-full-2.2.9.jar*, który został specjalnie przygotowany dla połączeń z tym serwerem. Jest on do pobrania z oficjalnej strony "*http://www.firebirdsql.org/*". W razie potrzeby połączenie z innym rodzajem bazy danych, wystarczy w analogiczny sposób zaimportować biblioteki przygotowane przez twórców danego serwera a następnie tylko wywołać je poprzez klasę `java.sql.DriverManager` bez większej ingerencji w kod źródłowy programu.

3.6.2. Implementacja klasy

Posiadając zaimportowaną odpowiednią bibliotekę można rozpocząć implementację. Klasa obsługująca połączenie powinna posiadać metody do połączenia się z serwerem oraz metodę do wysłania zapytania i pobrania danych, które zwróci dane polecenie. Do przechowywania zwróconych danych użyta zostanie wcześniej stworzona klasa kontener `RecordSet`. Próba nawiązania połączenia będzie odbywała się w czasie tworzenia obiektu w konstruktorze sparametryzowanym, zawierające parametry dotyczące połączenia, takie jak użytkownik, hasło, adres czy port.

Składowe klasy powinny przechowywać obiekt obsługujący aktualne połączenie oraz obiekt obsługujący wysyłanie zapytań do serwera:

```
19 public class DBHandle {
20  Connection con;
21  Statement stmt;
```

Zgodnie ze wcześniejszymi ustaleniami, nawiązanie połączenia odbywać się będzie w konstruktorze klasy. Do konstruktora przekazaną zostaną wszystkie informacje możliwe informacje na temat połączenia w postaci parametrów:

```
44    public DBHandle(String engine,String hostname,
45    String port, String dbpath,String encoding,
46     String user,  String password){
```

Odpowiednio:

— `engine` - Nazwa silnika bazodanowego
— `hostname` - adres serwera
— `port` - port serwera

— `dbpath` - ścieżka do bazy danych lub po prostu nazwa
— `encoding` - Kodowanie po stronie serwera łańcuchów znakowych.
— `user` - Nazwa użytkownika
— `password` - hasło

Następnie w ciele konstruktora nawiążemy połączenie poprzez stworzenie tak zwanego *Connection String*, czyli łańcucha znaków w odpowiednim formacie zawierającego informacje na temat połączenia, które ma zinterpretować klasa `java.sql.DriverManager` i spróbować nawiązać połączenie. *Connection String* przyjmuje wiele zmiennych także w zależności od rodzaju połączenia. Poniżej pokazany zostanie przypadek nawiązania połączenia z serwerem *Firebird 2.5* po adresie IP lub nazwie hosta. Najpierw ładowny jest sterownik `org.firebirdsql.jdbc.FBDriver`, następnie tworzony jest `Connection String` z parametrów przekazanych do konstruktora i ostatecznie wywoływana jest próba połączenia przez metodę `getConnection()`:

```
61 if(engine.equals("firebirdsql")){
62  try {
63   Class.forName("org.firebirdsql.jdbc.FBDriver");
64  } catch (ClassNotFoundException ex) {
65   System.out.println( "Data Base connection driver error for
66   Firebird \nEXCEPTION: " + ex.getMessage( ) );
67  }
68   constr = "jdbc:" + engine + ":" + hostname + ":" + port + "/"
69    + dbpath + "?encoding=" + encoding;
70   con = DriverManager.getConnection(constr,user,password);
71 }
```

Na koniec jeśli połączenie się powiedzie tworzymy obiekt odpowiedzialny za wywoływanie zapytań na danym połączeniu i referencję do niego zapisujemy pod składową `stmt`.

```
94 stmt = con.createStatement(
95 ResultSet.TYPE_FORWARD_ONLY,
96 ResultSet.CONCUR_READ_ONLY);
97 System.out.println( "Connection to database: OK" );
98 }
```

Pozostaje zaimplementować metodę odpowiedzialną za wysyłanie zapytania i pobieranie danych na obecnym połączeniu. Metoda będzie przyjmować parametr w postaci zapytania SQL i zwracać obiekt klasy `RecordSet` uzupełniony o wszystkie rekordy zwrócone przez zapytanie.

```
122 public  RecordSet executeSQL2(String SQL){
```

Następnie deklaracje potrzebnych zmiennych oraz wywołanie polecenia na bazie danych. Wyniki zapytania zwracane są do klasy `ResultSet`. Dodatkowo uzyskane zostaną jeszcze meta dane na temat ilości kolumn w pobranych rekordach.

```
123 int n=0,m=0,i=0,j=0;
124 String tmp;
125 RecordSet ret = null;
126 ResultSet rs = stmt.executeQuery( SQL );
127 rsmd = rs.getMetaData();
128 m=rsmd.getColumnCount();
```

Ze względu na brak odpowiedniego interfejsu ze strony obiektu klasy `ResultSet` na potrzeby dalszych manipulacji danymi, wszystkie rekordy zostaną przepisane do obiektu klasy `RecordSet`. W razie wartości `null` jakiegoś pola w rekordzie, zastąpiona ona będzie łańcuchem znaków:

```
135 while(rs.next()) {
136  for(j=0;j < m; j++){
137   tmp = rs.getString(j+1);
138   if(tmp != null){
139    ret.setVal(tmp, i, j);
```

```
140   }
141   else
142    ret.setVal("null", i, j);
143   }
144  i++;
145 }
146 rs.close();
147 return(ret);
148 }
```

3.7. Zarządzanie szablonami - klasa Templates

Wielkość i złożoność tej klasy wymaga podzielenia jej opisu na kilka podsekcji. Należy także dodać, że jest to jedyna klasa w której dochodzi do jakichkolwiek manipulacji danymi. Proces działania tej klasy można pokazać na krótkiej liście:

1. Przeszukanie wprowadzonych ścieżek pod kątem istnienia szablonów, czyli plików z rozszerzeniem *.tex*
2. Załadowanie do pamięci wszystkich znalezionych szablonów pod podaną ścieżką.
3. Przeszukanie załadowanych szablonów pod kątem informacji o selekcji danych, które mają zastać dodane do danego szablonu.
4. Przetworzenie odpowiednio rekordów z bazy danych zgodnie z informacją zawartą w szablonach.
5. Zapisanie wygenerowanych danych do szablonów w odpowiednich miejscach.
6. Utworzenie i zapisanie uzupełnionych szablonów do plików o tej samej nazwie w innej podanej lokalizacji.

Cała implementacja tego procesu znajduje się w podsekcjach poniżej.

3.7.1. Składowe klasy

Klasa powinna przechowywać podstawowe informacje takie jak:

1. `path` - Ścieżka do szablonów
2. `pathoutput` - Ścieżka do zapisania nowych uzupełnionych szablonów o dane.
3. `encoding` - Kodowanie zapisu szablonów.

```
30 public class Templates {
31
32  String path;
33  String pathoutput;
34  String encoding;
```

Deklaracje składowych tablicowych, przechowujących o plikach:

1. `listOfFiles` - Lista plików w podanej lokalizacji z rozszerzeniem *.tex*
2. `data` - Sczytana zawartość pliku, gdzie indeks w tablicy odpowiada indeksowi w składowej `listOfFiles`
3. `compileable` - Zmienna informująca czy plik szablonu o danym indeksie ma możliwość kompilacji.

```
36  File[] listOfFiles;
37  String[] data;
38  int[] compileable;
```

3.7.2. Konstruktor klasy

Konstruktor klasy będzie miał ze zadanie zainicjowanie obiektu. Powinien przyjmować więc parametry o ścieżkach odczytu i zapisu szablonów jak i ich kodowania:

```
51  public Templates(String pathin, String pathout, String enc){
```

Następnie zapisanie parametrów pod składowe obiektu i sprawdzenie ich poprawności. Tworzony jest obiekt `File` ze ścieżki wynikowej, który pozwala na sprawdzenie, czy istnieje podany katalog i w razie braku tworzony jest nowy o tej nazwie.

```
54  path = pathin;
55  pathoutput = pathout;
56  encoding = enc;
57  int n;
58  if(path.indexOf("\\",path.length()-1) < 0 &&
59   path.indexOf("/",path.length()-1) < 0){
60    path = path + "\\";
61   }
62  if(pathoutput.indexOf("\\",pathoutput.length()-1) < 0 &&
63  pathoutput.indexOf("/",pathoutput.length()-1) < 0){
64   pathoutput = pathoutput + "\\";
65  }
66  File tmp = new File(pathoutput);
67  if (!tmp.exists()) {
68   tmp.mkdirs();
69   }
```

Kolejnym krokiem będzie sprawdzenie i sporządzanie listy plików zawierających szablony znajdujących się pod ścieżką `path`. Listę tą zapiszemy pod tymczasową tablicą `files`.

```
74  File folder = new File(path);
75  File[] listOfFilestmp = folder.listFiles();
76  n = listOfFilestmp.length;
77  String[] files = new String[n];
78  for (int i = 0; i < n; i++)
79  {
80   if (listOfFilestmp[i].isFile())
81   {
82    test = listOfFilestmp[i].getName().indexOf(".tex");
83     if(test != -1 && listOfFilestmp[i].getName().length()
84     -".tex".length()-test == 0){
85      files[k] = listOfFilestmp[i].getName();
86      k++;
87      }
88     }
89    }
```

Na koniec konstruktora pod składową `listOfFiles` przypisywana jest tablica o dokładnej ilości elementów, którą jest liczba szablonów. Tworzona jest także dynamiczna macierz obiektów typu `ParsedSQLInfo` do przechowywania sparsowanych informacji z szablonów. Ostatecznie tworzone są obiekty typu `File` ze ścieżki przechowującej szablony oraz całej tablicy `files` posiadającej nazwę pliku ze szablonem.

```
90  listOfFiles = new File[k];
91  sqlinfo = new ArrayList<ArrayList<ParsedSQLInfo>>();
92  for(int i=0;i < listOfFiles.length; i++)
93   sqlinfo.add(new ArrayList<ParsedSQLInfo>());
94  compileable = new int[k];
95  data = new String[k];
96  for (int i = 0; i < listOfFiles.length; i++){
97   listOfFiles[i] = new File(path + files[i]);
98  }
```

W ten sposób dostajemy zainicjalizowany obiekt, w którym należy wywołać metody załadowujące szablony do pamięci.

3.7.3. Metody zarządzające operacjami na plikach

Do zaimplementowania potrzeba metody odczytującej oraz metody zapisującej szablon. Dodatkowo przydatnym byłoby napisanie metod wywołujących te metody dla wszystkich szablonów. Informacje o istniejących szablonach znajdują się w tablicy `listOfFiles`.

Ładowanie szablonu

Potrzebne więc będzie odczytać pojedynczy szablon o indeksie k i zapisać go w pamięci pod składową `data` z tym samym indeksem. Dodatkowo po załadowaniu szablonu wywołamy metodę `isCompilable(k)`, która sprawdzi czy jest możliwość kompilacji danego szablonu. Jest to jedna z dodatkowych funkcji programu. Metoda ta opisana została w podsekcji metod parsujących.

```
121 public void loadTemplate(int k){
122
123 try {
124  BufferedReader in = new BufferedReader(
125   new InputStreamReader(
126    new FileInputStream(listOfFiles[k]), encoding));
127
128   StringBuilder str = new StringBuilder();;
129   String tmp;
130   while ((tmp = in.readLine()) != null) {
131    str.append(tmp+"\n");
132    }
133   in.close();
134   in = null;
135   data[k]=str.toString();
136   System.out.println( "Loaded file: " +  listOfFiles[k].getPath());
137
138  isCompilable(k);
```

Zapisywanie uzupełnionego szablonu

Szablon zapisany pod indeksem `k` powinien zostać zapisany w miejscu zapisanym w składowej `pathoutput` pod nazwą taką samą jak nazwa szablonu czyli `listOfFiles[k].getName()`. Dodatkowym parametrem klasy `PrintWriter` przy otwarciu pliku do zapisu jest kodowanie, które trzymane jest w składowej `encoding`. Po otwarciu pliku potrzeba zapisać zawartość składowej `data` o indeksie `k` do pliku.

```
171 public void saveTemplate(int k){
172
173  try {
174   PrintWriter output = new PrintWriter(pathoutput + listOfFiles[k].getName(),encoding);
175   output.print(data[k]);
176   output.close();
177   output = null;
178   System.out.println( "Saved to file: "  + pathoutput +  listOfFiles[k].getName() );
179 }
```

Wczytanie wszystkich szablonów

Wywołanie metody `loadTemplate(i)` dla wszystkich znalezionych szablonów.

```
188 public void loadAllTemplates(){
189
190  for (int i = 0; i < listOfFiles.length; i++){
191   this.loadTemplate(i);
192  }
193 }
```

Zapisanie wszystkich szablonów

Wywołanie metody `saveTemplate(i)` dla wszystkich znalezionych szablonów.

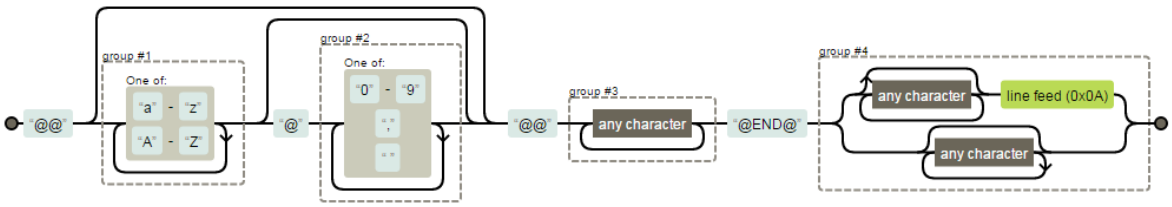
```
200 public void saveAllTemplates(){
201
202  for (int i = 0; i < listOfFiles.length; i++){
203   this.saveTemplate(i);
204  }
205
206 }
```

3.7.4. Metody parsujące zawartość szablonów

Parsowanie czyli inaczej analiza zawartości jest podstawową funkcją tego systemu. W szablonach zgodnie z wcześniej ustalonym algorytmem, znajdować się będą informacje o selekcji danych, które należy znaleźć oraz zapisać w odpowiedniej strukturze. Dodatkowo zaimplementowana zostanie także funkcja, sprawdzająca czy dany szablon może zostać skompilowany samodzielnie.

Metoda wyszukująca informacje o selekcji we wszystkich szablonach za pomocą wzorca

Celem metody jest przeszukanie wszystkich szablonów, używając do tego dostarczonego wzorca, pod kątem znalezienia informacji o selekcji. Następnie informacje te powinny zostać zapisane do przygotowanej składowej klasy, która może przyjąć te dane. Metoda przyjmować będzie za parametr wzorzec wyrażeń regularnych w celu możliwości łatwej zmiany w przyszłości w pliku konfiguracyjnym. Najważniejszym elementem tej metody jest właśnie zapisany w niej wzorzec (linia 218), za pomocą którego można znaleźć selekcje danych. Podzielony on został grupy, dzięki czemu będzie można oddzielić od siebie informacje i zapisać je w przygotowanej strukturze. Na rysunku 3.7.4 przedstawiony został wzorzec. Jeżeli nie podany zostanie żaden wzorzec, użyty zostanie standardowy.



Rysunek 3.1. Graf przedstawiający wzorzec przechwytujący informacje o selekcji w szablonach

Grupy przechwytujące:
— Grupa 1: Nazwa środowiska
— Grupa 2: Grupowanie
— Grupa 3: Zapytanie SQL
— Grupa 4: Przechwycenie pozostałości (wykorzystane tylko w celu znalezienia końca informacji)

W języku JAVA zaimplementowana została już obsługa wyrażeń regularnych. Wystarczy więc wykorzystać dostępne klasy `Pattern` oraz `Matcher`. Przez wywołanie statycznej metody `compile` z flagami `Pattern.DOTALL | Pattern.MULTILINE` uzyskamy odpowiednio skompilowane wyrażenie regularne. Zewnętrzna pętla wykorzystana będzie

do iterowania szablonów, natomiast wewnętrzna pętla do zapisu znalezionych selekcji dla danego szablonu. Składowa `sqlinfo` po jednorazowym wywołaniu tej metody przechowywać będzie dla k-tego szablonu wszystkie znalezione selekcje.

```
214 public void parseSqlStatements(String pat)
215 {
216
217  if(pat.equals(""))
218   pat = "@@(?:([a-zA-Z]+)(?:@([0-9, ]+))?)?@@(.+?)@END@(.*?\\n|.*)";
219
220  Pattern localPattern = Pattern.compile(pat, Pattern.DOTALL | Pattern.MULTILINE);
221  Matcher localMatcher;
222  for(int k=0;k < this.getLenght();k++){
223
224   localMatcher = localPattern.matcher(this.data[k]);
225   while (localMatcher.find())
226   {
227     sqlinfo.get(k).add(new ParsedSQLInfo(localMatcher.group(3),
228     localMatcher.group(1), localMatcher.group(2), localMatcher.end()));
229    }
230   }
231 }
```

Metoda sprawdzająca zdolność szablonu do kompilacji

W czasie tworzenia projektu wynikła potrzeba wprowadzenia dodatkowej funkcji, która jest dodatkową funkcjonalnością wykorzystywaną w klasie odpowiadającej za kompilacje szablonów. Poniższa metoda sprawdza każdy szablon czy jest możliwe skompilowanie go, poprzez sprawdzenie czy wewnątrz szablonu znajduje się początek i koniec środowiska `document`. Informacje te zapisywane są odpowiednie dla k-tego szablonu w składowej klasy, która jest tablicą. Dla kompilowanych szablonów zapisywana jest odpowiednio jedynka, natomiast dla braku możliwości kompilacji jest to zero. Wywoływana jest ona w metodzie ładowania szablonu(138 linia).

```
155  private void isCompilable(int k) {
156   String start = "\\begin{document}";
157   String end = "\\end{document}";
158   int from = data[k].indexOf(start);
159   int to = data[k].indexOf(end);
160
161   if(from == -1 || to == -1)
162    compileable[k] = 0;
163   else
164    compileable[k] = 1;
165  }
```

3.7.5. Metody prostego przetwarzania danych z bazy

Podsekcja opisuje proces przetwarzania danych pobranych z bazy danych do formatu, który jest w stanie odczytać kompilator LaTeX.

Metoda przygotowująca dane i wybierająca rodzaj przetwarzania

Do metody przekazywane są następujące parametry:

— `int k` - Indeks pliku zapisanego w składowej
— `int c` - Indeks selekcji ze znalezionych w danym pliku
— `String env` - Znaleziona nazwa środowiska dla danej selekcji
— `String grouping` - Znaleziona informacja o grupowaniu danych
— `RecordSet rs` - Dane pobrane z bazy danych

Jeśli parametr `env` wygenerowana zostanie nazwa środowiska za pomocą nazwy pliku oraz dodany do tego postfiks, który będzie dużą literą alfabetu łacińskiego w zależności od numeru selekcji.

Jeśli parametr `grouping` jest pusty to znaczy ze wykorzystane zostanie proste przetwarzanie poprzez wywołanie metody `prepareValuesSimple(k,c, prefix, rs);`.

W przeciwnym wypadku przygotowana zostanie tablica `int group`, która zawierająca informację o ilości i wielkości wszystkich grup. Tablica ta zostanie przekazana przy wywołaniu metody przetwarzającą grupowanie danych `prepareValuesGrouping(group, 0,0,rs.get_rows(), rs, prefix)`. Metoda ta potrzebuje pewnych inicjacji w postaci parametrów 0, dlatego że jest to metoda rekurencyjna. Dokładny opis znajduje się w późniejszej sekcji.

```
246 public int prepareValues(int k,int c,String env,String grouping,RecordSet rs){
247  if(rs == null)
248    return(0);
249
250   char l = (char) ('A' + (char)c);
251   String prefix;
252   if(env == null)
253    prefix =  listOfFiles[k].getName().substring(0,
254            listOfFiles[k].getName().indexOf('.')) + l;
255   else
256    prefix = env;
257
258   if(grouping == null){
259    prepareValuesSimple(k,c, prefix, rs);
260    System.out.println(  rs.get_rows() + " records SQL" + (c+1));
261   }
262   else{
263    String[] groupstmp = grouping.split(",");
264    int[] group = new int[groupstmp.length];
265    for(int i=0;i<groupstmp.length;i++)
266     group[i] = Integer.parseInt(groupstmp[i]);
267
268    sqlinfo.get(k).get(c).setData(prepareValuesGrouping(
269            group, 0,0,rs.get_rows(),  rs, prefix));
270
271    groupstmp = null;
272    group = null;
273    System.out.println(  rs.get_rows() + " records SQL" + (c+1));
274    rs = null;
275    }
276   return(0);
277  }
```

Metoda prostego przetworzenia danych

Metoda ma na celu utworzenie dla każdego przekazanego rekordu w parametrze metody `RecordSet rs` utworzenie linii tekstu zawierającego wywołanie środowiska o nazwie przekazanej w parametrze metody `String prefix` i od parametrów, którymi będą kolejne pola w rekordach.

Obiekt typu `RecordSet` jest macierzą typu `String`, którą należy połączyć w odpowiedni sposób. Do tego celu wykorzystana została klasa `StringBuilder`, która potrafi połączyć znaczą ilość obiektów typu `String` w optymalny sposób poprzez metodę `append` i utworzyć z nich jeden obiekt.

W odpowiedni sposób przygotowane dane zapisane zostają pod `sqlinfo.get(k).get(c).setData(temp2);` czyli k-tym plikiem oraz c-tym zapytaniem w składowej do tego przeznaczonej.

```
289  public int prepareValuesSimple(int k,int c, String prefix, RecordSet rs){
290   String temp1 = "";
291   String temp2 = "";
```

```
292
293   StringBuilder result = new StringBuilder();
294
295   for(int i=0; i < rs.get_rows(); i++){
296    temp1 = "\\" + prefix;
297    for(int j=0; j < rs.get_cols(); j++)
298     temp1 += "{" + rs.getVal(i, j) + "}";
299
300    result.append(temp1 + "\n");
301   }
302
303   temp2 = result.toString();
304   sqlinfo.get(k).get(c).setData(temp2);
305   return(0);
306 }
```

3.7.6. Metody grupujące dane z bazy

Do stworzenia drzewa grup, opisanego w sekcji działania algorytmu zaimplementowana została specjalnie do tego celu rekurencyjna metoda, która wywołuje samą siebie. Ze względu na czytelność, podzielona została ona na 2 metody opisane poniżej.

Metoda rekurencyjnie generująca grupy

Metoda przyjmuje parametry:

— `int[] group` - tablica grup
— `int gid` - Indeks aktualnej grupy
— `int s` - Numer rekordu od którego rozpoczyna pracę w tym wywołaniu w `rs record`
— `int e` - Numer rekordu na którym kończy pracę w tym wywołaniu w `rs record`
— `RecordSet rs` - Dane pobrane z bazy danych
— `String prefix` - Nazwa środowiska

Dla aktualnej grupy i zakresu działania metoda uruchamia proste przetwarzanie danych lub w razie istnienia niższych grup, wykonuje kolejny podział na kolejną grupę poprzez analizę zawartości pól po których jest ta grupa. Dla każdej unikatowej kombinacji tworzone jest wywołanie grupy z tymi wartościami, a następnie metoda wywołuje samą siebie od zakresu w którym występują dane unikatowe wartości. Wywołanie w ten sposób tej metody zwróci zawartość tej grupy która zostanie umiejscowiona pomiędzy wywołaniem grupy oraz wywołaniem zamknięcia grupy.

Do nazwy grupy zgodnie z algorytmem dodawane są litery alfabetu łacińskiego w zależności od numeru grupy.

Jeśli skończą się grupy do tworzenia, wywoływana jest metoda `prepareInjectValues(prefix, rs,s,e,sc)` która zwróci dla danego zakresu rekordów i wyciętych odpowiednich pól, proste wywołania środowiska o przekazanej nazwie. Wywołanie tej metody, zwróci przetworzone odpowiednio wartości, które przekazane zostaną dalej do grupy z której została wywołana metoda grupująca.

```
322  public String prepareValuesGrouping(
323          int[] group, int gid,int s, int e, RecordSet rs, String prefix){
324
325   if(gid == group.length){
326    int sc=0;
327    for(int i=0;i<gid;i++)
328     sc += group[i];
329    return(prepareInjectValues(prefix, rs,s,e,sc));
330    }
331   if(gid < group.length){
332    StringBuilder result = new StringBuilder();
333    char x;
```

```
334    String strreturn="";
335    String strtmp="";
336    String endin;
337    int index1=s;
338    int index2=s;
339    int check=0;
340
341    int col =0;
342    for(int i=0;i<gid;i++)
343     col += group[i];
344
345    index2++;
346    while(index2 <= rs.get_rows() && index2 <= e){
347     if(index2 == rs.get_rows() || index2 == e)
348      check = 2;
349     else{
350      for(int i=0;i<group[gid];i++)
351       if(!rs.getVal(index1,i+col).equals(rs.getVal(index2,i+col)))check = 1;
352     }
353
354     if(check >= 1){
355
356      x = (char) ('A' + (char)gid);
357      strtmp= "\\" + prefix+x;
358
359
360      for(int i=0;i<group[gid];i++)
361       strtmp += "{" + rs.getVal(index1,i+col)+ "}";
362
363      strtmp += "\n";
364      strtmp += prepareValuesGrouping(group, gid+1,index1, index2, rs, prefix);
365      endin =  "\\end" + prefix + x + "\n";
366
367      result.append(strtmp.concat(endin));
368
369      index1 = index2;
370      check = 0;
371     }
372     index2++;
373    }
374   return(result.toString());
375   }
376 return("");
377 }
```

Metoda przetwarzająca dane po zgrupowaniu

Metoda prostego już przetwarzania danych rozszerzona o dodatkowe rozróżnianie rekordów oraz pól, które ma zapisywać.

W parametrach przekazywane są informacje:

— `int s` - rekord od którego rozpoczyna pracę w `rs record`
— `int c` - rekord na którym kończy pracę w `rs record`
— `int sc` - numer pole w rekordzie od którego ma dopiero pobierać dane, a wszystkie wcześniejsze pominąć
— `String prefix` - nazwa środowiska dla danej selekcji
— `RecordSet rs` - Dane pobrane z bazy danych

Metoda ostatecznie zwraca przetworzone dane. Łączenie tych danych w całość odbędzie się w metodzie `prepareValuesGrouping` opisanej wcześniej.

```
380  private String prepareInjectValues(String prefix,RecordSet rs,int s, int e,int sc){
381   String temp1 = "";
382   String temp2 = "";
383
384   StringBuilder result = new StringBuilder();
385
386   for(int i=s; i < e; i++){
```

```
387    temp1 = "\\" + prefix;
388    for(int j=sc; j < rs.get_cols(); j++)
389     temp1 += "{" + rs.getVal(i, j) + "}";
390
391    temp1 += "\n";
392    result.append(temp1);
393
394       }
395
396   return(result.toString());
397 }
```

3.7.7. Metoda uzupełnienia szablonu o przetworzone dane

Metoda dodająca przetworzone wartości do zmiennej przechowującej zawartość szablonu. Dodawanie odbywa się od ostatniego zapytania do pierwszego, aby nie zmieniać dodatkowo zapamiętanych indeksów miejsc, w których znajdują się selekcje.

```
402  public void InjectValues(int k){
403   int to;
404   for(int i=sqlinfo.get(k).size()-1; i >= 0; i--){
405     to = sqlinfo.get(k).get(i).getIndex();
406     data[k] = data[k].substring(0,to) + sqlinfo.get(k).get(i).getData()
407     + data[k].substring(to,data[k].length());
408    }
409
410  }
```

3.7.8. Akcesory

Akcesory (gettery i settery) to określenie metod jakie wykorzystuje obiekt do pobrania wartości i modyfikacji swoich atrybutów. Są one bardzo często dołączane do klas przez programistów Javy i nawet niektóre edytory (np. Eclipse) posiadają funkcjonalność automatycznego wygenerowania ich.

Pobranie ilości szablonów:

```
414  public int getLenght(){return data.length;}
```

Pobranie ścieżki do pliku uzupełnionego szablonu w tex:

```
419  public String getSavedPath(int k){return (pathoutput + listOfFiles[k].getName());}
```

Pobranie ścieżki wynikowej:

```
423  public String getOutputPath(){return (pathoutput);}
```

Pobranie informacji o tym czy k-ty szablon może zostać skompilowany:

```
429  public int compileCheck(int k){return(compileable[k]);}
```

Pobranie wszystkich informacji o selekcji dla k-tego szablonu:

```
436  public ArrayList<ParsedSQLInfo> getStmts(int k){return sqlinfo.get(k);}
```

Pobranie nazwy pliku o indeksie k:

```
443  public String getFileName(int k ){ return(listOfFiles[k].getName());}
```

3.8. Zarządzanie kompilatorem LaTeX - klasa LatexCompiler

Uzupełnione szablony przez klasę Templates, należy następnie skompilować przy pomocy wywołania kompilatora który został podany w konfiguracji. Do tego posłuży właśnie klasa LatexCompiler, która wywoła polecenie w konsoli systemowej, uruchamiające kompilator od odpowiednich parametrów. W składowych zapamiętane zostaną informacje o ścieżce do kompilatora oraz ścieżka plików wynikowych.

```
18 public class LatexCompiler {
19
20  String latexcompilerpath;
21  String pdfoutput;
```

Konstruktor

Konstruktor z 2 parametrami przekazującymi ścieżkę do kompilatora oraz katalog wynikowy, będzie miał za zadanie zapisać te parametry do składowych oraz sprawdzić czy katalog wynikowy istnieje. W razie jego braku, utworzy go.

```
31 public LatexCompiler(String latexcompilerpathh,String pdfoutputt){
32  latexcompilerpath = latexcompilerpathh;
33  pdfoutput = pdfoutputt;
34
35  File tmp = new File(pdfoutput);
36  if (!tmp.exists()) {
37   tmp.mkdirs();
38  }
39
40  }
```

Metoda wywołująca kompilacje pliku LaTeX

Parametrem metody jest nazwa pliku, który zostanie poddany kompilacji. Następnie aby skompilować dany plik należy uruchomić kompilator, do którego ścieżka zapisana jest w składowej `latexcompilerpath` z parametrem `-output-directory` w którym przekażemy katalog wynikowy oraz z parametrem którym jest ścieżka pliku do kompilacji. Wygenerowane polecenie zostanie wywołane w powłoce systemowej za pomocą metody `lexecuteCommand` opisanej w podsekcji niżej.

```
46 public int compileTemplate(String filetexpath){
47  System.out.print( executeCommand(latexcompilerpath
48   + " --output-directory=" + pdfoutput + " " + filetexpath));
49  return(0);
50  }
```

Metoda wywołująca polecenie powłoki systemu

Metoda ta ma na celu uruchomić nowy proces, którym będzie wywołanie komendy powłoki przesłanej przez parametr `String commmand`. `Runtime.getRuntime().exec(ćmd /c start "+ command);` Wywoła polecenie w nowym oknie i przypnie proces do zmiennej `Process p`. Następnie aby synchronicznie wywoływać polecenie jedno po drugim, program czeka na zakończenie procesu.

```
56 private String executeCommand(String command) {
57
58  StringBuffer output = new StringBuffer();
59  System.out.print("Executing shell command: \n" + "cmd /c start " + command);
60  Process p;
61  try {
62   p = Runtime.getRuntime().exec("cmd /c start " + command);
63   p.waitFor();
```

3.9. Metoda main() - klasa DBRaportLatex

Metoda `main()` jest metodą od której program rozpoczyna swoją pracę. Poniżej przedstawione zostaną jej części:

— Przyjmuje ona argumenty, które można przekazać podczas uruchomienia programu w postaci tablicy Stringów.

```
38 public static void main(String[] args) {
```

— Pierwszą rzeczą którą należy zrobić po uruchomieniu programu jest załadowanie pliku konfiguracyjnego. Dodatkową opcją jest przekazanie nazwy pliku konfiguracyjnego jako pierwszy argument. Ładowanie konfiguracja odbywa się poprzez stworzenie nowego obiektu klasy `Config` przez wywołanie konstruktora z 1 parametrem, a dokładniej nazwą pliku konfiguracyjnego:

```
44  Config cfg;
45
46  if(args.length == 0)
47   cfg = new Config("dblatexraportconfig.bat");
48  else{
49   System.out.print(args[0] + "\n");
50   cfg = new Config(args[0]);
51  }
```

— Kolejną rzeczą do zrobienia jest uzyskanie połączenia z bazą danych. Wykorzystane zostaną przy tym zmienne załadowane z pliku konfiguracyjnego poprzez wywołanie metody `getString` na obiekcie `cfg`. Do stworzenia połączenia wykorzystany zostanie obiekt klasy `DBHandle`. Po utworzeniu obiektu przez konstruktor sparametryzowany do przekazane zostaną odpowiednie zmienne konfiguracyjne, połączenie powinno zostać utworzone:

```
61  DBHandle FBH = new DBHandle(cfg.getString("dbengine"),cfg.getString("hostname"),
62   cfg.getString("port"), cfg.getString("dbpath"),
63   cfg.getString("dbencoding"), cfg.getString("user"),
64   cfg.getString("password"));
```

— Inicjalizacja obiektu zarządzającego kompilacją:

```
67  LatexCompiler comp = new LatexCompiler(cfg.getString("pdflatexpath"),
68  cfg.getString("output"));
```

— Inicjalizacja obiektu zarządzającego szablonami:

```
72  Templates temps = new Templates(cfg.getString("templatepath"),
73   cfg.getString("output"), cfg.getString("encodingtex"));
```

— Załadowanie wszystkich szablonów do pamięci za pomocą metody `loadAllTemplates();` na zainicjalizowanym odpowiednimi zmiennymi, obiekcie zarządzającym szablonami:

```
85  temps.loadAllTemplates();
```

— Przygotowanie zmiennych potrzebnych do parsowania szablonów oraz wywołanie metody parsującej wszystkie szablony od wyrażenia regularnego zapisanego w pliku konfiguracyjnym (jeśli nie zostało wpisane wyrażenie regularne, użyte zostanie standardowe).

```
String pattern;
pattern = cfg.getString("pattern");
ArrayList<ParsedSQLInfo>  SQLSt;

temps.parseSqlStatements(pattern);
```

— Główna pętla przechodząca przez wszystkie szablony, pobierająca wszystkie polecenia selekcji z każdego szablonu. Następnie wewnętrzna pętla dla każdego polecenia wywołuje zapytanie w bazie danych i od zwróconych wyników wywołuje funkcje generującą dane do uzupełnienia w szablonie. Na koniec zewnętrznej pętli wywołuje metodę uzupełniającą szablon o wygenerowane dane:

```
for(int i=0; i < temps.getLenght(); i++){
 System.out.print("\n");
 SQLSt = temps.getStmts(i);

 if(SQLSt != null){
  System.out.println(temps.getFileName(i) + " - processing SQL statements");
  for(int j=0; j < SQLSt.size(); j++){


   if(SQLSt.get(j).getName() == null){
    System.out.println("no returns SQL" + (j+1));

    FBH.executeSQL2(SQLSt.get(j).getQuery());

    }
    else

     temps.prepareValues(i,j,SQLSt.get(j).getName(),
     SQLSt.get(j).getGroup(),FBH.executeSQL2(SQLSt.get(j).getQuery()));
   }
  }
temps.InjectValues(i);
SQLSt = null;
}
```

— Zapisanie wszystkich uzupełnionych szablonów do plików:

```
temps.saveAllTemplates();
```

— Na koniec przeprowadzenie kompilacji uzupełnionych szablonów w zależności od zmiennej `pdfcompilemainfile` używając przygotowanego obiektu `comp` klasy `LatexCompiler`.

```

if(cfg.getString("pdfcompilemainfile").equals("ALL"))
 for(int i=0; i < temps.getLenght(); i++){
  comp.compileTemplate(temps.getSavedPath(i));
  }
if(cfg.getString("pdfcompilemainfile").equals("ONLYBEGDOC"))
 for(int i=0; i < temps.getLenght(); i++){
  if(temps.compileCheck(i)==1)
   comp.compileTemplate(temps.getSavedPath(i));
   }

else if(cfg.getString("pdfcompilemainfile").equals("NONE"));
else{

 String[] tmpfiles = cfg.getString("pdfcompilemainfile").split(",");

 for(int i=0;i < tmpfiles.length; i++)
   comp.compileTemplate(temps.getOutputPath() + tmpfiles[i].trim());

}
```

3.10. Kompilacja programu

Do kompilacji programu dołączony jest plik *build.xml*. Opisuje on przebieg pakowania pliku jar. Ze względu na wykorzystanie paru zewnętrznych bibliotek, tworzony był katalog *library* zawierający te biblioteki i aby tego uniknąć zastosowana została funkcja pakująca wszystkie biblioteki do jednego pliku jar. W listingu poniżej pokazana jest konfiguracja pliku *build.xml*:

```
 <?xml version="1.0" encoding="UTF-8"?>

<project name="DBRaportLatex" default="default" basedir=".">
    <description>Builds, tests, and runs the project DBRaportLatex.</description>
    <import file="nbproject/build-impl.xml"/>
<target name="-post-jar">

    <property name="store.jar.name" value="DBRaportLatex"/>

    <property name="store.dir" value="dist"/>
    <property name="store.jar" value="${store.dir}/${store.jar.name}.jar"/>

    <echo message="Packaging ${application.title} into a single JAR at ${store.jar}"/>

    <jar destfile="${store.dir}/temp_final.jar" filesetmanifest="skip">
        <zipgroupfileset dir="dist" includes="*.jar"/>
        <zipgroupfileset dir="dist/lib" includes="*.jar"/>

        <manifest>
            <attribute name="Main-Class" value="${main.class}"/>
        </manifest>
    </jar>

    <zip destfile="${store.jar}">
        <zipfileset src="${store.dir}/temp_final.jar"
        excludes="META-INF/*.SF, META-INF/*.DSA, META-INF/*.RSA"/>
    </zip>

    <delete file="${store.dir}/temp_final.jar"/>
    <delete dir="${store.dir}/lib"/>
    <delete file="${store.dir}/README.TXT"/>
</target>

</project>
```

4. Uruchomienie oraz testowanie systemu

Przed wdrożeniem programu do realnego systemu, program należy przetestować. Testy powinny być prowadzone na tymczasowej bazie danych, ponieważ idea testów jest taka, że po podmianie bazy danych na realną wszystko ma działać bez zmian. Zmienić się może tylko zapis połączenia z bazą danych w pliku konfiguracyjnym. Dzięki takiemu zabiegowi, będzie można być pewnym tego, że wszystko będzie działać na prawdziwej bazie danych. Do przeprowadzenia testów potrzebne będzie odtworzyć przyszłe środowisko, w którym będzie działać program, przygotować szablony dokumentów, które są wytwarzane w procesie rekrutacji oraz wygenerować dokumenty. Ostatnim już krokiem będzie sprawdzenie czy podczas tego procesu nie ma żadnych komplikacji oraz czy wygenerowane dokumenty nie zawierają błędów.

4.1. Wstępne ustalenia

Przed rozpoczęciem jakichkolwiek działań należy przygotować podstawowe rzeczy do rozpoczęcia testów.

4.1.1. Specyfikacja maszyny do testów

Testy przeprowadzone będą na komputerze o następującej specyfikacji:

```
OS Name Microsoft Windows 7 Ultimate
Version 6.1.7601 Service Pack 1 Build 7601
System Model    Z97-lstlisting
System Type     x64-based PC
Processor       Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz,
3301 Mhz, 4 Core(s), 4 Logical Processor(s)
Installed Physical Memory (RAM) 8,00 GB
Total Physical Memory   7,86 GB
Available Physical Memory       3,99 GB
Total Virtual Memory    9,86 GB
Available Virtual Memory        3,75 GB
Page File Space 2,00 GB
```

Do działania programu *DBRaportlatex* zainstalowana została JAVA w wersji:

```
C:\Users\Pawlos>java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

4.1.2. Struktura katalogowa

Do przejrzystego przechowywania wszystkich rzeczy potrzebnych w systemie rekrutacji, potrzebna będzie odpowiednia struktura katalogów. Do przechowywania potrzebne będą:

— program: `DBLatexRaport.jar`
— plik konfiguracyjny: `dblatexraportconfig.bat`
— środowisko kompilacji latex

— szablony dokumentów
— plik bazy danych(tylko w testach)
— pliki wynikowe
— biblioteki potrzebne dla silnika bazy danych firebird w wersji embedded

Proponowana struktura więc może wyglądać następująco:

```
/
├── Firebird
├── output
├── template
├── texlive
├── DBLatexRaport.jar
├── dblatexraportconfig.bat
└── rekrutacja.fdb
```

Do katalogu *texlive* przegrane zostało środowisko do kompilacji plików latex. W katalogu firebird są biblioteki tylko na czas testów do wersji embedded bazy danych. W katalogu *template* dograne zostaną szablony dokumentów.

4.2. Generowanie przykładowych danych

W celu przetestowania systemu generowania raportów w procesie rekrutacji kandydatów na studia potrzebne będą testowe dane w dokładnie tej samej strukturze co w systemie rekrutacji, ponieważ w szablonach latexu znajdują się zapytania SQL do danych tabel w bazie danych. W celu otrzymania tych danych potrzebne będzie:
1. Utworzenie nowej bazy danych na silniku Firebird'a
2. Stworzenie wystarczającej struktury tabel odzwierciedlającą strukturę w systemie uczelnianym.
3. Wygenerowanie dużej ilości testowych danych osobowych.
4. Uzupełnienie tabel danymi, które zostały wygenerowane wcześniej oraz dodanie do nich dodatkowych, jednocześnie losowych, informacji na temat procesu rekrutacji.

Po wykonaniu tych kroków, powinna powstać baza danych do której bez problem program połączy się i wyciągnie z niej potrzebne dane dokładnie jak z realnej bazy danych.

4.2.1. Tworzenie bazy danych

Do stworzenia pliku bazy danych na silniku firebird'a posłużyć się można narzędziem dostępnym w katalogu bin zainstalowanego serwera firebird. Narzędzie to pozwala z linii komend tworzyć i łączyć się z bazami danych. W tym przypadku użyta zostanie komenda `CREATE DATABASE`

```
C:\Program Files\Firebird\bin>isql
SQL>CREATE DATABASE 'D:\test systemu\rekrutacja.fdb'
CON>user 'SYSDBA' password 'masterkey';
```

Po wykonaniu tego polecenia zostanie utworzona baza danych. Takie same dane należy teraz wpisać do pliku konfiguracyjnego programu czyli `DBLatexRaport.bat` aby program mógł się połączyć z tą bazą:

```
dbengine=firebirdsql
```

```
hostname=//localhost
port=3050
dbpath=D:\test systemu\rekrutacja.fdb
user=SYSDBA
password= masterkey
```

4.2.2. Stworzenie struktury bazy

Dotychczasowy system wykorzystywał tabelę (widok) która była generowana dynamicznie i która zawiera wszystkich studentów w rekrutacji. Zawiera ona wszystkie dane potrzebne do wytworzenia dokumentów. Jeden rekord to jeden student ze wszystkimi informacjami na jego temat. Dodatkowo jeszcze potrzebna jest tabela z informacjami na temat rekrutacji, takimi jak na przykład nazwisko i imię przewodniczącego komisji, czy data wydania decyzji przyjęcia studenta. Z tych tabel będą pobierane informacje, natomiast do wygenerowania danych potrzebne będą dwie dodatkowe tymczasowe tabele. Tabela główna z kandydatami(zapis SQL):

```
CREATE TABLE KANDYDAT_ALIGEZA (
    STUD_ID                         INTEGER NOT NULL,
    OSOBA_ID                        INTEGER NOT NULL,
    STUD_NRTECZKI                   INTEGER,
    NAZWISKO                        VARCHAR(100),
    IMIE                            VARCHAR(100),
    NAZWISKOIMIONA                  VARCHAR(200),
    ADR_ULICA_MIEJSCOWOSC_NR_DOMU   VARCHAR(200),
    ADR_KOD_POCZTOWY_POCZTA         VARCHAR(100),
    STUDIA_NAZWA                    VARCHAR(100),
    STUD_ILPUNKTOW                  INTEGER,
    STUD_ILPUNKTOWKREM              INTEGER,
    TOKNAUKI_NAZWATOKU              VARCHAR(200),
    KIERUNEK                        VARCHAR(100),
    SPEC_ID                         INTEGER,
    DATAPRZYJECIAPODANIA            DATE,
    TOKNAUKI_ID                     INTEGER,
    OSOBA_PESEL                     VARCHAR(50),
    KIERUNEK_MY                     VARCHAR(200),
    FORMA_STUDIOW_MY                VARCHAR(200),
    STOPIEN_STUDIOW_MY              VARCHAR(100),
    KIERUNEK_FORMA_SKROT_MY         VARCHAR(100),
    NR_DECYZJI                      VARCHAR(100),
    CZY_PRZYJETY                    INTEGER,
    DATA_DECYZJI                    DATE,
    ILE_PUNKTOW                     INTEGER,
    PANPANI                         CHAR(1)
);
```

Tabela z dodatkowymi informacjami(wartości przypisane są do kluczy tekstowych, jest to tablica asocjacyjna):

```
CREATE TABLE SETUP_ALIGEZA (
    KLUCZ    VARCHAR(50) NOT NULL,
    WARTOSC  VARCHAR(100)
);
```

Tymczasowa tabela do zaimportowania listy imion i nazwisk oraz losowych peseli.

```
CREATE TABLE DANE (
    IMIE_NAZ  VARCHAR(200),
    ADRES     VARCHAR(200),
    PESEL     VARCHAR(50),
    NAZWISKO  VARCHAR(100),
    IMIE      VARCHAR(100)
);
```

Tymczasowa tabela do procedury losowego uzupełniania informacji o rekrucie o kierunku jaki wybrał.

```
CREATE TABLE TOKNAUKI_ALIGEZA (
    TOKNAUKI_ID              INTEGER NOT NULL,
    KIERUNEK_MY              VARCHAR(50),
    FORMA_STUDIOW_MY         VARCHAR(50),
    STOPIEN_STUDIOW_MY       VARCHAR(50),
    KIERUNEK_FORMA_SKROT_MY  VARCHAR(10),
    LICZBA_MIEJSC            SMALLINT,
    DATA_DECYZJI_OD          TIMESTAMP,
    DATA_DECYZJI_DO          TIMESTAMP,
    KOD_IKR                  VARCHAR(3)
);
```

4.2.3. Generowanie testowych danych osobowych

Do wygenerowania kandydatów potrzebne jest imię, nazwisko oraz adres. Takie dane dostępne są w książkach telefonicznych. Posługując się jedną z nich stworzony został plik *CSV* o separatorze „;” zawierający po kolei imię z nazwiskiem, adres, pesel, nazwisko oraz imię. Pesel został dodany do każdej osoby jako losowy ciąg cyfr spełniający walidację peselu. Ze względu na fakt iż pesel został wygenerowany losowo, w wygenerowanych dokumentach nie będzie możliwości ustawienia płci osoby, dlatego też w bazie danych do testów 5 pierwszych kandydatów dostanie płeć żeńską, a reszta męską.
Struktura pliku:

```
 Abram Andrzej; Lwowska 116;88071640299;Abram;Andrzej
 Abram Halina; Ludwika Zamenhofa 2;86111210691;Abram;Halina
...
```

Tak sformatowany plik *CSV*, łatwo zaimportować do bazy danych do tabeli `dane` ze względu na identyczną kolejność danych w kolumnach. Do importu wykorzystana została funkcja programu *IBExpert*, `import data`. Jedna linijka w pliku zostaje zaimportowana jako jeden rekord, w którym każde pole po kolei odpowiada wartościom między średnikami. Zaimportowanych w ten sposób zostało 10001 rekordów (osób) do tabeli `dane` do dalszych manipulacji, co całkowicie wyczerpuje ilość kandydatów potrzebnych do testów. Ze statystyk wynika, że na uczelni średnio na rekrutacje przypada około trzystu kandydatów.

4.2.4. Generowanie kandydatów na studentów

Kolejnym krokiem jest uzupełnienie tabeli z tokami studiów. W testach dodanych zostało 8 przykładowych toków nauki.

```
1 Informatyka   niestacjonarne  pierwszego stopnia      INF-n
2 Mechatronika  niestacjonarne  pierwszego stopnia      MT-n
3 Mechatronika  stacjonarne     pierwszego stopnia      MT-s
...
```

Uzupełnienia wymaga także tabela z dodatkowymi informacjami `setup_aligeza` przykładowymi danymi:

```
dataWydaniaDecyzji      09.10.2015
miejsceWydaniaDecyzji   Nowy Sącz
przewodniczacyIKR       mgr inż. Sławomir Jurkowski
rokAkademicki   2015/2016
czyUwzglednicDateWydaniaDecyzji N
...
```

Mając już to wszystko potrzebna jest procedura, która utworzy listę kandydatów z tych wszystkich danych.

```
create procedure GENERUJ
returns (
    TESTCHAR varchar(50),
    TEST integer)
```

```
as
declare variable IMIE varchar(100);
declare variable NAZ varchar(100);
declare variable IMIENAZ varchar(200);
declare variable ADRES varchar(200);
declare variable PESEL varchar(50);
declare variable LICZNIK integer;
declare variable STOPIEN varchar(50);
declare variable KIERUNEK varchar(50);
declare variable FORMA varchar(50);
declare variable SKROT varchar(10);
declare variable RANDINT integer;
declare variable PUNKTY integer;
declare variable CZY_PRZYJETY integer;
declare variable DATA_DEC varchar(100);
begin
licznik = 1;
for select * from dane into
:imienaz,:adres,:pesel,:naz,:imie
do
begin
randint = CAST(round(rand()*7+1) as INTEGER);
punkty = CAST(round(rand()*500) as INTEGER);
if(punkty > 250) then czy_przyjety = 1;
if(punkty <= 250) then czy_przyjety = 2;

select kierunek_my,forma_studiow_my,stopien_studiow_my,
kierunek_forma_skrot_my
FROM toknauki_aligeza where toknauki_id = :randint
INTO :kierunek,:forma,:stopien,:skrot;

select wartosc FROM setup_aligeza
WHERE klucz='dataWydaniaDecyzji'
INTO :data_dec;

INSERT INTO kandydat_aligeza (stud_id,osoba_id,stud_nrteczki,nazwisko,imie,
        nazwiskoimiona,adr_ulica_miejscowosc_nr_domu,adr_kod_pocztowy_poczta,
        osoba_pesel,panpani, studia_nazwa,toknauki_nazwatoku,kierunek,
        kierunek_my,forma_studiow_my,stopien_studiow_my,
        kierunek_forma_skrot_my,    stud_ilpunktow,stud_ilpunktowkrem,
        ile_punktow,czy_przyjety,data_decyzji,nr_decyzji,dataprzyjeciapodania)
VALUES (:licznik,:licznik,cast(round(rand()*200+1) as integer),
        :naz,:imie,:imienaz,:adres, cast( 'Nowy Sacz 33-300' as varchar(100)),
        :pesel,cast( 'M' as char(1)),    :forma,:kierunek ||
        ' N inz. 3.50 2015/2016 zimowy',:kierunek,:kierunek,:forma,
        :stopien,:skrot,    :punkty,:punkty,:punkty,:czy_przyjety,
        cast(:data_dec as DATE),'328/2015','2015-08-14');

licznik = :licznik + 1;
end
test = :licznik;
suspend;
end
```

Powyższa procedura z jednej osoby z tabeli `dane` tworzy jednego kandydata, losując mu tok nauki, ilość punktów oraz czy zostanie przyjęty lub nie. Dodawane są także pewne stałe wartości, podobne do tych w oryginalnej bazie danych, które nie wymagają uzmiennienia. Procedura ta, po jednorazowym wywołaniu, wygenerowała 10001 rekordów w tabeli `kandydat_aligeza`. Daje to wystarczającą ilość testowych kandydatów do przeprowadzenia testów. Baza danych z przeprowadzonych testów znajduje się w załączniku.

4.3. Dodanie zapytań SQL do szablonów

Stworzone szablony w poprzednim rozdziale nie zawierają zapytań, które odzwierciedlały by wyselekcjonowanie danych z istniejącej struktury bazy. Zawierają tylko sztywne

dane testujące środowiska LaTeX'owe. Aby szablony podczas procesu tworzenia dokumentów zostały uzupełnione poprawnymi danymi potrzeba je uzupełnić o odpowiednia zapytania SQL w odpowiedniej formie tak aby parser zawarty w programie był wstanie je znaleźć i wykonań.

Potrzebnych jest 9 zapytań, za pomocą których wyciągnięte zostaną dane do szablonów. Poniżej opisane zostały każde z zapytań oraz przykład jakie dane zostaną wyselekcjonowane z bazy przez to zapytanie. Ta sekcja skupia się tylko i wyłącznie na uzyskaniu potrzebnych danych do środowisk LaTeX'a, a nie jak one zostaną wykorzystane. Dokładny opis jak wygląda wykorzystanie takiego zapisu znajduje się w rozdziale 2. Dodać jeszcze należy ze poniższe zapytania znajdują się pod koniec każdego szablonu, zaraz po zdefiniowaniu środowisk. Takie umiejscowienie tych zapytań sprawi, że dane zostaną uzupełnione zaraz pod tymi zapytaniami.

Szablon z ustawieniami - aaasetup.tex

Pierwsze zapytanie które zostanie omówione jest to zapytanie o podstawowe parametry, informacje na temat rekrutacji. Znajduje się ono w pliku *aaasetup.tex*. Plik posiada 3 litery a w nazwie ponieważ program parsujący, otwiera pliki alfabetycznie a zapytanie to musi zostać wykonane jako pierwsze. Poniżej mamy przykład danych do zapisania:

```
\parametrRekrutacyjny{dataWydaniaDecyzji}{09.10.2015}
\parametrRekrutacyjny{miejsceWydaniaDecyzji}{Nowy Sącz}
\parametrRekrutacyjny{rokAkademicki}{2015/2016}
...
```

Aby uzyskać takie dane, potrzeba będzie pobrać wszystkie rekordy z tabeli `setup_aligeza` (struktura tabeli znajduje się w poprzedniej sekcji) klucz oraz wartość. Zapytanie nie będzie więc zbyt skomplikowane:

```
SELECT klucz,wartosc FROM setup_aligeza
```

Po wywołaniu go faktycznie otrzymamy dane potrzebne nam dane. Jednak aby program parsujący był wstanie znaleźć takie zapytanie w szablonie potrzeba je obudować w odpowiednią konstrukcję, która będzie odpowiadać wyrażeniu regularnemu zapisanemu w parserze. Należy także wpisać nazwę środowiska w odpowiednie miejsce między znakami @@. Dodatkowo musimy jeszcze postarać się aby kompilator LaTeX'a zignorował nasz zapis i go nie wyświetlał. Do tego celu zastosowana została komenda `iffalse`.

```
\iffalse@@parametrRekrutacyjny@@
SELECT klucz,wartosc FROM setup_aligeza
@END@\fi
```

Szablon protokołu przekazania - protokolprzekazania.tex

Następnie zapytanie zawarte w pliku *protokolprzekazania.tex*. Występuje tu nowy problem otworzenia środowiska na początku oraz zamknięcia na końcu. Pokazane jest to na przykładowych danych poniżej:

```
\protokolprzekazaniaA{1}
\protokolprzekazania{Informatyka ---
 niestacjonarne STUDIA pierwszego stopnia}{1455}
\protokolprzekazania{Informatyka ---
 stacjonarne STUDIA pierwszego stopnia}{729}
\protokolprzekazania{Mechatronika ---
 niestacjonarne STUDIA pierwszego stopnia}{1447}
...
\endprotokolprzekazaniaA
```

Pomijając na razie otwarcie środowiska `protokolprzekazaniaA{1}` oraz zamkniecie `endprotokolprzekazaniaA`, w zapytaniu potrzebujemy wyświetlić każdy tok nauczania oraz z agregowaną liczbę kandydatów na tym toku. Do agregacji użyjemy polecenia `group` na polu tok by móc otrzymać liczbę kandydatów używając funkcji count(*). Dodatkowo potrzeba także wybrać kandydatów tylko posiadającą daną datę decyzji. Do tego posłuży podzapytanie, które pobierze datę decyzji z aktualnej rekrutacji i porówna ją z datą decyzji dla danego kandydata.

```
SELECT  k.kierunek_my||' --- '||k.forma_studiow_my||
' STUDIA '||k.stopien_studiow_my AS tok, count(*) AS ile
FROM kandydat_aligeza k
WHERE  k.data_decyzji=(SELECT wartosc FROM
 setup_aligeza WHERE klucz='dataWydaniaDecyzji')
GROUP BY tok
```

Takie zapytanie wyświetli nam dane pomiędzy tymi pominiętymi dotychczas komendami. Rozwiązaniem wyświetlenia tych komend jest funkcja programu uzupełniającego szablony, a dokładnie funkcja grupowania. Dokładny opis tej funkcji znajduje się w rozdziale ref. Do zapytania dodamy jako pierwsze pole wartość 1, a następnie użyta zostanie funkcja grupowania po 1 polu wewnątrz szablonu:

```
\iffalse @@protokolprzekazania@1@@
SELECT 1 AS n,k.kierunek_my||' --- '||
k.forma_studiow_my||' STUDIA '||
k.stopien_studiow_my AS tok, count(*) AS ile
FROM kandydat_aligeza k
WHERE  k.data_decyzji=(SELECT wartosc FROM
 setup_aligeza WHERE klucz='dataWydaniaDecyzji')
GROUP BY tok
@END@\fi
```

Szablon z nadrukami na koperty- pocztex.tex

Dla szablonu 'pocztex.tex' potrzebujemy prostą listę z adresami każdego kandydata z danej rekrutacji:

```
\iffalse@@pocztex@@
SELECT ka.nazwiskoimiona,
       ka.adr_ulica_miejscowosc_nr_domu,
           ka.adr_kod_pocztowy_poczta,
           ka.stud_nrteczki||' '||ka.kierunek_forma_skrot_my||' '||ka.stud_id
FROM kandydat_aligeza ka
WHERE  ka.data_decyzji=
(select wartosc from setup_aligeza
where klucz='dataWydaniaDecyzji')
ORDER BY ka.kierunek_forma_skrot_my, ka.nazwiskoimiona
@END@\fi
```

Szablon z listami kandydatów do sekretariatu - listasekretariat.tex

W kolejnym szablonie *listasekretariat.tex* potrzebne jest utworzenie list kandydatów dla każdego toku nauki. Przez tok nauki rozumie się każdą kombinację kierunku, formy oraz stopnia studiów. W szablonie zostało to rozwiązane tak, że każda lista otwiera się poprzez wywołanie komendy `listarsekretariatA` z trzema parametrami, którymi są właśnie kierunek, forma oraz stopień studiów. Na przykładowych danych wygląda to następująco:

```
\listarsekretariatA{drugiego stopnia}{Informatyka}{stacjonarne}
\listarsekretariat{ Ablewicz Monika}{39}
\listarsekretariat{ Abram Halina}{69}
...
\endlistarsekretariatA
\listarsekretariatA{drugiego stopnia}{Mechatronika}{stacjonarne}
\listarsekretariat{ Adamek Danuta}{141}
\listarsekretariat{ Adamek Urszula}{35}
```

```
\listarsekretariat{ Adamik Zbigniew}{95}
...
\endlistarsekretariatA
...
```

W zapytaniu więc potrzebne będzie wybrać wszystkich kandydatów z datą decyzji aktualnej rekrutacji, a następnie wyświetlić kierunek, formę, stopień studiów, imię, nazwisko oraz numer teczki. W zapytaniu także potrzebne będzie sortowanie po polach, które będą grupowane w programie uzupełniającym szablon. Każde na tych polach w zapytaniu SQL użyjemy polecenia `ORDER BY`. Ostatecznie poinformujemy program, aby otrzymane dane grupował po 3 pierwszych polach:

```
\iffalse@@listarsekretariat@3@@
SELECT stopien_studiow_my,kierunek_my,forma_studiow_my,
       nazwiskoimiona,stud_nrteczki
FROM kandydat_aligeza
WHERE data_decyzji=
(select wartosc from setup_aligeza
 where klucz='dataWydaniaDecyzji')
ORDER BY kierunek_forma_skrot_my, nazwiskoimiona
@END@\fi
```

Szablony z listami listaprzyjetych.tex, listanieprzyjetych.tex, listawysylkowa.tex, listarankingowa.tex

W szablonach znajdują się jeszcze 4 kolejne podobne listy: przyjętych, nieprzyjętych,wysyłkowa oraz rankingowa. Zasada utworzenia zapytań do tych list jest identyczna jak w liście do sekretariatu. Różnią się one będą jedynie pewnymi warunkami wewnątrz zapytań SQL czy też polami po których będą sortowane dane.

Lista przyjętych:

```
\iffalse@@listaprzyjetych@3@@
SELECT stopien_studiow_my,kierunek_my,forma_studiow_my,
       nazwiskoimiona
FROM kandydat_aligeza
WHERE czy_przyjety=1 and
     data_decyzji=
     (SELECT wartosc FROM setup_aligeza
     WHERE klucz='dataWydaniaDecyzji')
ORDER BY kierunek_forma_skrot_my,
         nazwiskoimiona
@END@\fi
```

Lista nieprzyjętych:

```
\iffalse@@listaprzyjetych@3@@
SELECT stopien_studiow_my,kierunek_my,forma_studiow_my,
       nazwiskoimiona
FROM kandydat_aligeza
WHERE czy_przyjety=2 and
     data_decyzji=(SELECT wartosc FROM setup_aligeza
     WHERE klucz='dataWydaniaDecyzji')
ORDER BY kierunek_forma_skrot_my,
         nazwiskoimiona
@END@\fi
```

Lista wysyłkowa:

```
\iffalse @@listawysylkowa@3@@
SELECT stopien_studiow_my,kierunek_my,forma_studiow_my,
      nazwiskoimiona, adr_ulica_miejscowosc_nr_domu||'; '||
       ADR_KOD_POCZTOWY_POCZTA
FROM kandydat_aligeza
WHERE data_decyzji=(select wartosc from setup_aligeza
where klucz='dataWydaniaDecyzji')
ORDER BY kierunek_forma_skrot_my, nazwiskoimiona
@END@\fi
```

Lista rankingowa:

```
\iffalse@@listarankingowa@3@@
SELECT stopien_studiow_my,kierunek_my,forma_studiow_my,
       nazwiskoimiona,STUD_ILPUNKTOW
FROM kandydat_aligeza
WHERE  data_decyzji=(select wartosc from setup_aligeza
where klucz='dataWydaniaDecyzji')
ORDER BY kierunek_forma_skrot_my, STUD_ILPUNKTOW desc
@END@\fi
```

Szablon z decyzją odnośnie przyjęcia kandydata - decyzja.tex

W ostatnim szablonie wystąpił problem ilości parametrów jakie trzeba przekazać do poprawnego działania szablonu. W LaTeX'ie można stworzyć środowisko o maksymalnie 9 parametrach, natomiast w szablonie *decyzja.tex* dla każdego kandydata potrzeba 12 różnych informacji. Rozwiązaniem tego problemu także było grupowanie przez program uzupełniania raportów. Grupowanie wytnie 3 pierwsze pola i wstawi je do nowego środowiska. W zapytaniu SQL wyświetlanie daty musi posiadać dodatkowe 0 przy datach typu 01.02.2015 gdzie właśnie liczby dni lub miesięcy są mniejsze od 10:

```
\iffalse@@decyzja@3@@
SELECT ka.stopien_studiow_my,ka.kierunek_my,ka.forma_studiow_my,
        ka.stud_ilpunktow,ka.nr_decyzji,ka.nazwiskoimiona,ka.adr_ulica_miejscowosc_nr_domu,
        ka.adr_kod_pocztowy_poczta,ka.stud_nrteczki||' '||ka.kierunek_forma_skrot_my||
        ' '||ka.stud_id,czy_przyjety,extract(day from ka.dataprzyjeciapodania)||'.'||
        substring(100+extract(month from ka.dataprzyjeciapodania) from 2 for 2)||'.'||
        extract(year from ka.dataprzyjeciapodania), panpani
FROM kandydat_aligeza ka
WHERE  ka.data_decyzji=(select wartosc from setup_aligeza
where klucz='dataWydaniaDecyzji')
ORDER BY ka.kierunek_forma_skrot_my, ka.nazwiskoimiona
@END@\fi
```

4.4. Konfiguracja programu

W głównym katalogu znajduje się plik konfiguracyjny programu o nazwie *dblatexraportconfig.bat*. Plik ten jest jednocześnie plikiem wsadowym oraz zawiera konfiguracje programu *DBRaportlatex.jar*. Część kodu batch uruchamia program DBraportlatex razem z odpowiednimi parametrami do uruchomienia *embedded firebird*:

```
@echo off
java -Djava.library.path=.\firebird -jar dbraportlatex.jar
pause
exit
```

Następnie od linijki zawierającej `#dbLatexRaportConfig` zaczynają się zmienne konfiguracyjne:

— Ścieżka do szablonów LaTeX

```
templatepath=template/
```

— Ścieżka do uzupełnionych szablonów LaTeX oraz skompilowanego pliku PDF

```
output=output/
```

— Ścieżka do uzupełnionych szablonów LaTeX oraz skompilowanego pliku PDF

```
output=output/
```

— Kodowanie szablonów LaTeX

```
encodingtex=UTF-8
```

— Konfiguracja połączenia z bazą danych

```
dbengine=firebirdsql
hostname=embedded
dbpath=rekrutacja.fdb
user=SYSDBA
password=
dbencoding=WIN1250
```

— Ścieżka do komilatora LaTeX

```
pdflatexpath=texlive\2010min\bin\win32\pdflatex.exe
```

— Lista plików do kompilacji

```
pdfcompilemainfile=main.tex,main.tex
```

4.5. Przebieg testu

W bazie danych znajduje się 10001 kandydatów. Należy spodziewać się tyle samo wygenerowanych na listach. Program został uruchomiony z pliku wsadowego *dblatexraportconfig.bat*. Poniżej znajduje się wydruk z konsoli po uruchomieniu programu z krótkimi opisami poszczególnych procesów:

1. Ładowanie pliku konfiguracyjnego przez program:

```
Config loaded: dblatexraportconfig.bat
pattern=
templatepath=template/
output=output/
encodingtex=UTF-8
dbengine=firebirdsql
hostname=embedded
dbpath=rekrutacja.fdb
user=SYSDBA
password=*********
dbencoding=WIN1250
pdflatexpath=texlive\2010min\bin\win32\pdflatex.exe
pdfcompilemainfile=main.tex,main.tex
```

2. Połączenie z bazą danych:

```
Connection to database: OK
```

3. Ładowanie szablonów do pamięci:

```
Loaded file: template\aaasetup.tex
Loaded file: template\decyzja.tex
Loaded file: template\listanieprzyjetych.tex
Loaded file: template\listaprzyjetych.tex
Loaded file: template\listarankingowa.tex
Loaded file: template\listarsekretariat.tex
Loaded file: template\listawysylkowa.tex
Loaded file: template\main.tex
Loaded file: template\pocztex.tex
Loaded file: template\protokolprzekazania.tex
```

4. Wywoływanie zapytań SQL oraz uzupełnianie szablonów:

```
aaasetup.tex - processing SQL statements
no returns SQL1
9 records SQL2

decyzja.tex - processing SQL statements
10001 records SQL1

listanieprzyjetych.tex - processing SQL statements
5051 records SQL1
```

```
listaprzyjetych.tex - processing SQL statements
4950 records SQL1

listarankingowa.tex - processing SQL statements
10001 records SQL1

listarsekretariat.tex - processing SQL statements
10001 records SQL1

listawysylkowa.tex - processing SQL statements
10001 records SQL1

main.tex - processing SQL statements

pocztex.tex - processing SQL statements
10001 records SQL1

protokolprzekazania.tex - processing SQL statements
8 records SQL1
```

5. Zapisywanie uzupełnionych szablonów do katalogu wynikowego:

```
Saved to file: output/aaasetup.tex
Saved to file: output/decyzja.tex
Saved to file: output/listanieprzyjetych.tex
Saved to file: output/listaprzyjetych.tex
Saved to file: output/listarankingowa.tex
Saved to file: output/listarsekretariat.tex
Saved to file: output/listawysylkowa.tex
Saved to file: output/main.tex
Saved to file: output/pocztex.tex
Saved to file: output/protokolprzekazania.tex
```

6. Wywoływanie kompilatora LaTeX dla głównego pliku *main.tex*, co spowoduje skompilowanie wszystkich szablonów:

```
Executing shell command:
cmd /c start texlive\2010min\bin\win32\pdflatex.exe
 --output-directory=output/ output/main.tex
Executing shell command:
cmd /c start texlive\2010min\bin\win32\pdflatex.exe
 --output-directory=output/ output/main.tex

WORK DONE!!!

Directory pdf results output: output/
```

Program wykonał się bez żadnych błędów i wykonał swoje zadanie. Uzupełnienie szablonów trwało poniżej 1 sekundy, natomiast podwójna kompilacja wszystkich szablonów około jednej minuty, co spełnia założenia projektu.

4.6. Wyniki testu

Wszystkie szablony zostały uzupełnione poprawnie danymi z bazy danych oraz kompilacja systemu LaTeX przebiegła bezproblemowo. W pliku wynikowym PDF ze wszystkimi dokumentami gotowymi do druku wszystko jest zgodnie z oczekiwaniami. Wygenerowany plik PDF z testów dołączony jest w załączniku.

5. Instrukcja obsługi

DBRaportLatex jest programem napisanym w JAVA'ie służącym do tworzenia raportów w środowisku latex z informacji przechowywanych w bazie danych.Program ten został przygotowany specjalnie w celu obsługi tworzenia dokumentów potrzebnych do rekrutacji na uczelni PWSZ Nowy Sącz. Program oparty jest głównie o wcześniej przygotowane szablony w środowisku latex, które w czasie tworzenia raportów są uzupełniane informacjami pobranymi przez program z bazy danych. Środowisko kompilacji szablonów zostało także przygotowane razem z programem. Kompilacja szablonów jest uruchamiana zaraz po uzupełnieniu szablonów i wynikiem jej jest plik pdf gotowy do wydruku.

5.1. Wymagania systemowe

Program do uruchomienia wymaga systemu operacyjnego z zainstalowanym pakietem *Java SE Runtime Environment 8*.

5.2. Przygotowanie pliku konfiguracyjnego

Plik konfiguracyjny powinien mieć nazwę `dblatexraportconfig.bat`. Plik konfiguracyjny jest jednocześnie plikiem wsadowym który uruchamia program. Polecania wsadowe rozpoczynają plik i kończą się przy fladze `#dbLatexRaportConfig`. Przykładowy początek pliku konfiguracyjnego:

```
@echo off
java dblatexraport.jar
exit
#dbLatexRaportConfig
```

Po fladze `#dbLatexRaportConfig` wprowadzane są zmienne. Wprowadza się za pomocą nazwy zmiennej oraz jej wartości po znaku "=". Linie zaczynające się od # zostaną zignorowane przez parser i mogą służyć jako komentarze. Na przykład:

```
#Ścieżka do katalogu z szablonami
templatepath=template/

#Ścieżka do katalogu wynikowego: szablonami uzupełnionymi
# o dane z bazy danych oraz plików pdf
output=output/
```

Poniżej opisane są wszystkie zmienne:

— templatepath - ścieżka do szablonów.
— output - ścieżka do zapisu wyników pracy programu
— encodingtex - kodowanie zapisu szablonów tex
— pattern - opcjonalnie, wyrażenie regularne przechwytujące polecania w szablonach.
— dbengine - nazwa sterowników do obsługi danego silnika bazy danych (obsługiwane firebirdsql, mysql, sqlite)

— hostname - IP lub nazwa hosta serwera bazy danych (`//192.168.1.2` lub `//Komputer-PC`) lub dla lokalnego pliku firebird'a `embedded`.
— port - port serwera bazy danych
— dbpath - ścieżka do bazy danych w przypadku firebirda lub nazwa bazy danych w przypadku MySQL
— user - nazwa użytkownika
— password - hasło
— dbencoding - kodowanie znaków w bazie danych (np. `dbencoding=WIN1250` . Jeśli puste, nastąpi automatyczne wykrycie kodowania)
— pdflatexpath - ścieżka do kompilatora latex
— pdfcompilemainfile - zmienna określająca jakie pliki kompilować za pomocą kompilatora latex. Można tutaj użyć wartości `ALL` (wszystkie), `ONLYBEGDOC` (skompilowane zostaną dokumenty tylko z begindoc), `NONE` (żadne) lub wymienić nazwy plików po przecinku (`main.tex,setup.tex ...`)

5.3. Polecenia w szablonach

Informacje zawarte w poleceniach do programu mają na celu wyselekcjonowanie danych z bazy danych i uzupełnienie szablonu zaraz po poleceniu o te pobrane dane. Proste polecenie zawiera w sobie:
— nazwę generowanego wywołania środowiska
— zapytanie SQL
Specjalnie dla tego programu został opracowany zapis polecenia tak aby kompilator dokumentów LaTeX ignorował to polecenie ale program mógł go przechwycić (w zapytaniu jest możliwość łamania linii):

```
\iffalse
@@Nazwa środowiska@@Zapytanie
SQL
do bazy danych@END@
\fi
```

Wynikiem takiego polecenia będzie następująca struktura dla każdego rekordu pobranego przez zapytanie SQL:

```
\nazwasrodowiska{pole1}{pole2}{pole3}...
```

Ograniczenie parametrów przyjmowanych przez środowisko niestety powoduje iż w zapytaniu może być maksymalnie 9 zwracanych wartości.

5.3.1. Polecenie puste

Polecenie puste jest jedynie wywoływane na bazie danych i nie zwraca ono żadnych danych. Nawet jeżeli umieścimy w nim jakąś selekcje zwracającą dane to i tak nie zostaną one wpisane do szablonu. Polecenia tego można użyć do uaktualnienia np. daty w której przeprowadzamy generowanie dokumentów.

Zapis polecenia pustego:

```
\iffalse
@@@@UPDATE .....@END@
\fi
```

5.3.2. Grupowanie

W programie istnieje także możliwość grupowania po polach, które zwraca zapytanie. Zapis grupowania jest niezależny od zapisu grupowania, natomiast działanie wymaga pewnego dodatku do zapytania jakim jest dodanie sortowania po tych polach(w SQL polecenie ORDER BY).
Zapis grupowania:

```
\iffalse
@@Nazwa środowiska@Grupowanie@@Selekcja danych@END@
\fi
```

W miejscu `Grupowanie` należy wpisać liczby oddzielone przecinkami, gdzie każda liczba będzie oznaczać grupę i ilość pobranych pól do tej grupy w zależności od wpisanej liczby. Poniżej przykład zapisu grupowania:

```
\iffalse
@@lista@1,2@@SELECT pole1, pole2, pole3, pole4@END@
\fi
```

Grupowanie działa według zasad:

— Z pobranych rekordów z bazy danych od lewej strony zabierane sa te pola, które sa grupowane i stwarzane są z nich grupy ze wszystkich ich możliwych kombinacji ich wartości.
— Ostatecznie wywołania środowisk będą miały tylko te wartości pól które nie były grupowane.
— Każda grupa będzie otwarta i zamknięta poprzez dodanie do nazwy środowiska end na początku.
— Grupy nie przeplatają się.
— Wszystkie rekordy objęte są wszystkimi grupami.
— Nazwa grupy tworzona jest poprzez dodanie na końcu nazwy środowiska dużej litery łacińskiego alfabetu, w zależności która jest to grupa(Grupa pierwsza A, druga B ...).
— Ograniczenie ilości grup to 26.
— Dla grupowanych pól wymagane jest dodanie sortowania (w SQL ORDER BY).

Struktura przykładowego grupowania:

```
\listaA{pole1}{pole2}
\listaB{pole3}
\lista{pole4}
\lista{pole4}
...
\endlistaB
\listaB{pole3}
\lista{pole4}
\lista{pole4}
...
\endlistaB
\endlistaA
...
```

5.4. Przygotowanie prostego szablonu

Uwaga: Do tworzenia szablonów wymagana jest minimalna znajomość pracy w środowisku Latex, dlatego też przedstawiony zostanie proces tworzenia prostego szablonu bez wyjaśniania znaczenia i działania poszczególnych komend.

Program generuje z otrzymanych danych, wywołania komend w Latex'ie. Dlatego też, aby wykorzystać te dane należy stworzyć nową komendę lub środowisko, które przetworzy

dane parametry.
Na przykład polecenie przyjmujące dwa argumenty, wyświetlające jeden wiersz w tablicy, dodatkowo posiadające licznik i wyświetlające licznik w pierwszej kolumnie a następnie oba argumenty w dwóch kolejnych kolumnach:

```
\newcommand{\wiersz}[2]{\stepcounter{lp}\thelp&{\tiny #1}&#2\\}
```

Przykładowe dane wywołujące to polecenie będą wyglądać następująco:

```
\lista{Przykładowy tekst}{Przykładowy  tekst w kolumnie 3}
\lista{Przykładowy tekst pojawi się w 2 wierszu}{Przykładowy tekst w kolumnie 3 wiersz 2}
```

Posiadając skonfigurowane polecenie należy utworzyć prosty szablon rozpoczynający tablicę:

```
\begin{center}
\textbf{Przykładowa lista:}
\end{center}

Poniżej powinna być tabela z wartościami numerowana od 1:
\begin{center}
\begin{tabular}{clc}
\hline
Lp.&\multicolumn{1}{c}{Nagłówek 2 kolumny}&Nagłówek 3 kolumny\\
\hline
```

Następnie w tym miejscu należy umieścić polecenie uzupełniające dane. Jak napisać takie polecenie opisane jest w następnej sekcji. Teraz użyte zostanie proste polecenie pobierające 2 pola z bazy danych, zapisujące pod nazwą wywołania komendy lista:

```
\iffalse @@lista@@
SELECT pole1, pole2
FROM tabela
@END@\fi
```

Polecenie to podczas kompilacji dokumentu zostanie zignorowane, wywołane zostaną tylko komendy które ów polecenie zwróciło podczas uzupełniania szablonów przez program.

Ostatecznie należy zamknąć tabelę:

```
\hline\end{tabular}\end{center}
```

W ten sposób otrzymany został prosty szablon tablicy uzupełnianej poprzez zapytanie do bazy danych i uzupełniane rekordami posiadającymi 2 pola z danej tabeli. Wynik uruchomienia programu i przeprowadzenia uzupełnienia szablonu wygląda następująco:

```
\newcommand{\lista}[2]{\stepcounter{lp}\thelp&{\tiny #1}&#2\\}

\begin{center}
\textbf{Przykładowa lista:}
\end{center}

Poniżej powinna być tabela z wartościami numerowana od 1:
\begin{center}
\begin{tabular}{clc}
\hline
Lp.&\multicolumn{1}{c}{Nagłówek 2 kolumny}&Nagłówek 3 kolumny\\
\hline
\iffalse @@lista@@
SELECT pole1, pole2
FROM tabela
@END@\fi
\lista{testtest}{123}
\lista{tekest teskt}{5555}
\lista{przykladowy tekst}{1111}

\hline\end{tabular}\end{center}
```

Natomiast wynik kompilacji powyższego szablonu przedstawiony został na rysunku poniżej.

Przykładowa lista:

Poniżej powinna być tabela z wartościami numerowana od 1:

Lp.	Nagłówek 2 kolumny	Nagłówek 3 kolumny
1	testtest	123
2	tekest teskt	5555
3	przykładowy tekst	1111

Rysunek 5.1. Wynik kompilacji przykładowego szablonu przedstawiającego tabelę

5.5. Uruchomienie programu

Program można uruchomić za pomocą polecenia, które jednak najlepiej jest umieścić w pliku konfiguracyjnym (dblatexraport.bat):

```
java -jar dblatexraport.jar
```

5.5.1. Uruchomienie programu dla wersji embedded Firebird

Wersja embedded dla silnika Firebird wymaga dodatkowego polecenia przy starcie programu:

```
java -Djava.library.path=.\firebird -jar dblatexraport.jar
```

gdzie jako parametr `Djava.library.path` podajemy ścieżkę do rozpakowanej paczki zawierającej wszelkie potrzebne biblioteki pobranej ze strony *http://www.firebirdsql.org/* dla wersji embedded.

6. Podsumowanie

Opracowany system generowania raportów usprawni i przyspieszy wszelkie procesy, w których wymagana jest duża ilość dokumentów, miedzy innymi właśnie proces rekrutacji na uczeni PWSZ w Nowym Sączu.

Dzięki programowi DBLatexRaport będzie możliwe:

— pełna automatyzacja procesu tworzenia dokumentów oraz raportów przy pomocy wcześniej stworzonych szablonów.

— usprawnienie pracy pracownikom przy segregowaniu raportów

— znaczny wzrost prędkości przebiegu całego procesu.

— eliminacja błędów człowieka, które mogły powstać przy ręcznym tworzeniu raportów.

— ze względu na kompatybilność z wieloma silnikami bazodanowymi, uruchomienie programu z różnych źródeł danych.

— ze względu na fakt iż program jest napisany w języku JAVA 1.8, uruchomienie go na wielu systemach operacyjnych, które mają możliwość zainstalowania pakietu JAVA.

Do programu dołączona jest krótka instrukcja obsługi, dzięki której każdy będzie w stanie skonfigurować i uruchomić program. Jedynym wymaganiem jest umiejętność pisania dokumentów w systemie LaTeX aby móc stworzyć szablony raportów.

Biorąc pod uwagę powyższe stwierdzenia, można uznać, że osiągnięto postawiony cel w pracy.

Bibliografia

[1] Kurs języka JAVA *http://javastart.pl/static/category/podstawy-jezyka/* [dostęp: 07.11.2015]

[2] Kurs języka JAVA *http://javastart.pl/static/category/podstawy-jezyka/* [dostęp: 07.11.2015]

[3] Marcin Lis: *Java. Leksykon kieszonkowy*, Helion 2005

[4] Dokumentacja NetBeans: *https://netbeans.org/kb/*, [dostęp: 07.11.2015]

[5] Dokumentacja Jaybird: *http://www.firebirdsql.org/en/drivers-documentation/* [dostęp: 05.12.2015]

[6] Forum poświęcone systemowi LaTeX: *http://www.latex-community.org/forum/* [dostęp: 12.12.2015]

[7] Dokumentacja silnika bazodanowego Firebird: *http://www.firebirdsql.org/en/documentation/* [dostęp: 28.11.2015]

[8] Poradnik korzystania z JDBC(TM) Database Access: *http://www.tutorialspoint.com/jdbc/* [dostęp: 16.11.2015]

[9] Wyrażenia regularne w języku JAVA: *http://edu.pjwstk.edu.pl/wyklady/mpr/scb/W2/W2.htm* [dostęp: 23.11.2015]

[10] Akcesory w programowaniu *http://blog.zabiello.com/2006/02/07/akcesory-w-javie-pythonie-ruby-i-php5* [dostęp: 27.11.2015]

Spis rysunków

Załączniki

Wszystkie załączniki umieszczono na dołączonej do pracy płycie DVD. Symbol „Ξ” jest początkiem drzewa katalogów, umieszczonego na płycie DVD. Ścieżki zostały oznaczone przy pomocy znaku slasha(/).

1. Tekst pracy

Tytułowy załącznik umieszczono w pliku: `Ξ/20747.pdf`

2. Kod programu w postaci projektu programu NetBeans

Katalog `Ξ/DBLatexRaport/` zawiera kod programu DBLatexRaport, w formacie projektu JAVA w NetBeans.

3. Środowisko stworzone do testów projektu

Folder `Ξ/RekrutacjaTest/` zawiera wszystkie szablony, bazę danych potrzebną do przeprowadzenia testów oraz skonfigurowany program DBRaportLatex gotowy do uruchomienia. Dodatkowo znajdują się wygenerowane dokumenty podczas testów.