

**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 Ligeza

NOWY SĄCZ 2015

Spis treści

1. Wprowadzenie	5
1.1. Zagadnienie generowania raportów	5
1.2. Dotychczasowy proces generowania raportów	5
1.3. Cel i zakres pracy	6
2. Szablony raportów w systemie Latex	7
2.1. Środowisko kompilacji raportów	7
2.2. Idea działania szablonów	7
2.3. Tworzenie szablonów raportów do systemu rekrutacji	7
2.3.1. Wyświetlanie listy	8
2.3.2. Grupowanie	8
2.3.3. Aktualizacja daty	8
3. Projekt programu i implementacja	9
3.1. Algorytm działania systemu raportowania	9
3.1.1. Algorytm parsowania	9
3.1.2. Selekcjonowanie danych	10
3.1.3. Struktura uzupełnianych danych	10
3.1.4. Wywołanie kompilacji szablonu	11
3.2. Wybór języka oraz środowiska programistycznego	11
3.3. Utworzenie projektu i jego struktury	12
3.4. Zarządzanie konfiguracją - klasa Config	12
3.5. Klasy typu kontener	14
3.5.1. Klasa ParsedSQLInfo	14
3.5.2. Klasa RecordSet	15
3.6. Obsługa połączenia z bazą danych - klasa DBHandle	16
3.6.1. Wykorzystanie JDBC(TM) Database Access	16
3.6.2. Implementacja klasy	16
3.7. Zarządzanie szablonami - klasa Templates	18
3.7.1. Składowe klasy	18
3.7.2. Konstruktor klasy	19
3.7.3. Metody zarządzające operacjami na plikach	20
3.7.4. Metody parsujące zawartość szablonów	21
3.7.5. Metody prostego przetwarzania danych z bazy	22
3.7.6. Metody grupujące dane z bazy	23
3.7.7. Metoda uzupełnienia szablonu o przetworzone dane	25
3.7.8. Metody get	25
3.8. Zarządzanie kompilatorem Latex - klasa LatexCompiler	25
3.9. Metoda main() - klasa DBRaportLatex	25
3.10. Kompilacja programu	25
4. Uruchomienie oraz testowanie systemu	26
4.1. Wstępne ustalenia	26
4.1.1. Specyfikacja maszyny do testów	26
4.1.2. Struktura katalogowa	26

4.2.	Generowanie przykładowych danych	27
4.2.1.	Tworzenie bazy danych	27
4.2.2.	Stworzenie struktury bazy	28
4.2.3.	Generowanie testowych danych osobowych	29
4.2.4.	Generowanie kandydatów na studentów	29
4.3.	Dodanie zapytań SQL do szablonów	31
	Szablon z ustawieniami - aaasetup.tex	31
	Szablon protokołu przekazania - protokolprzekazania.tex	31
	Szablon z nadrukami na koperty- pocztex.tex	32
	Szablon z listami kandydatów do sekretariatu - listasekretariat.tex	32
	Szablony z listami listaprzyjetych.tex, listanieprzyjetych.tex, listawysylkowa.tex, listarankingowa.tex	33
	Szablon z decyzją odnośnie przyjęcia kandydata - decyzja.tex	34
4.4.	Konfiguracja programu	34
4.5.	Przebieg testu	35
4.6.	Wyniki testu	37
5.	Instrukcja obsługi	38
5.1.	Wymagania systemowe	38
5.2.	Przygotowanie pliku konfiguracyjnego	38
5.3.	Przygotowanie prostego szablonu	38
5.4.	Dodatkowe polecenia w szablonach	38
5.4.1.	Polecenie puste	38
5.4.2.	Grupowanie	38
5.5.	Uruchomienie programu	38
5.6.	Rozwiązywanie problemów	38
6.	Podsumowanie	39
	Bibliografia	40

1. Wprowadzenie

Dokumenty, listy, protokoły to nierozłączna część każdego procesu rekrutacji na uczelni wyższej. Ręczne tworzenie takich dokumentów może narażać 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 eliminację 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 przejrzanie 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 generowania 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 mogą 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

2.1. Środowisko kompilacji raportów

blablabla nie mam pojęcia

2.2. Idea działania 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 *DBLate-xRaport* wyszuka takie zapytanie i uzupełni szablon wywołaniami środowisk z wartościami parametrów, jakimi będą wartości pola z rekordów zapytania SQL.

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 powtarzające się problemy, które należało rozwiązać. Jako że rozwiązania tych problemów powtarzają się, to zamiast opisywania każdego szablonu po kolei, przedstawione poniżej zostały najważniejsze problemy, wynikające z tworzenia tych szablonów.

2.3.1. 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
...
```

2.3.2. Grupowanie

2.3.3. Aktualizacja daty

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*. Pominęte 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 że 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{czyUwzględnicDateWydaniaDecyzji}{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 `\end{}` 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:

```

\NazwaA
|
|_ \NazwaB
|   |
|   |_ \Nazwa
|   |_ \Nazwa
|   |_ ...
|   \end{NazwaB}
|_ \NazwaB
|   |
|   |_ \Nazwa
|   |_ \Nazwa
|   \end{NazwaB}
\end{NazwaA}

```

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 pisanie 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, zawierająca 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 *main()*, 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. Konfigurację 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 linii *#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 trzeba 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 linię 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 iterację.

```
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 zwracającej 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 pobierania zmiany lub pobrania 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 stworzymy obiekt odpowiedzialny za wywołanie 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 za 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 szablony. 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.

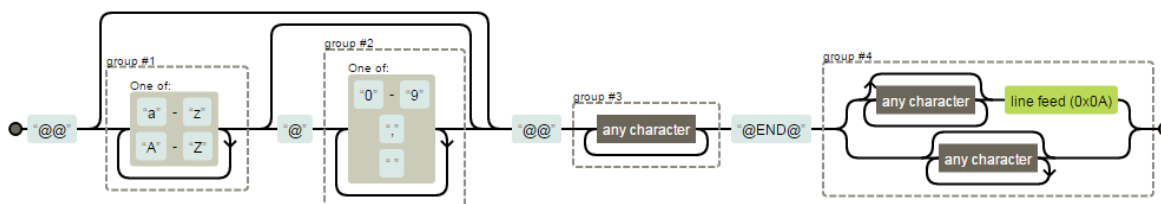
```
200 public void saveAllTemplates() {
201
202     for (int i = 0; i < listOfFiles.length; i++) {
203         this.saveTemplate(i);
204     }
205
206 }
```

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 wyrażen 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 3.1 przedstawiony został wzorzec. Jeżeli nie podany zostanie żaden wzorzec, użyty zostanie standardowy.



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

Grupy przechwytyjące:

- 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 wynikała potrzeba wprowadzenia dodatkowej funkcji, która jest dodatkową funkcjonalnością wykorzystywaną w klasie odpowiadającej za kompilację 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

opis podsekcji

Metoda przygotowująca i wybierająca rodzaj przetwarzania danych

opis

```

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,listOfFiles[k].getName().indexOf('.') + 1);
254     else
255         prefix = env;
256
257     if(grouping == null){
258         prepareValuesSimple(k,c, prefix, rs);
259         System.out.println( rs.get_rows() + " records SQL" + (c+1));
260     }
261     else{
262         String[] groupstmp = grouping.split(",");
263         int[] group = new int[groupstmp.length];
264         for(int i=0;i<groupstmp.length;i++)
265             group[i] = Integer.parseInt(groupstmp[i]);
266
267         sqinfo.get(k).get(c).setData(prepareValuesGrouping(group, 0,0,rs.get_rows(), rs, prefix));
268
269         groupstmp = null;
270         group = null;
271         System.out.println( rs.get_rows() + " records SQL" + (c+1));
272         rs = null;
273     }
274     return(0);
275 }

```

Przetworzenie danych do szablonu

opis

```

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     sqinfo.get(k).get(c).setData(temp2);
305     return(0);
306 }

```

3.7.6. Metody grupujące dane z bazy

opis subsekcji

Metoda rekurencyjna generująca grupy

opis metody

```

322 public String prepareValuesGrouping(int[] group, int gid,int s, int e, RecordSet rs, String prefix){
323
324     if(gid == group.length){
325         int sc=0;
326         for(int i=0;i<gid;i++)
327             sc += group[i];
328         return(prepareInjectValues(prefix, rs,s,e,sc));
329     }

```

```

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

```

Metoda przetwarzająca dane po zgrupowaniu

opis metody

```

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     return(result.toString());
396 }
397 }

```


3.7.7. Metoda uzupełnienia szablonu o przetworzone dane

opis metody

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

3.7.8. Metody get

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

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

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

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

402 public ArrayList<ParsedSQLInfo> getStmts(int k){return sqllinfo.get(k);}

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

3.8. Zarządzanie kompilatorem Latex - klasa LatexCompiler

test

3.9. Metoda main() - klasa DBRaportLatex

test

3.10. Kompilacja programu

test

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 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      Z97-1stlisting
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: dbletexraportconfig.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_PESSEL     VARCHAR(50),
    KIERUNEK_MY      VARCHAR(200),
    FORMA_STUDIOW_MY VARCHAR(200),
    STOPNEN_STUDIOW_MY VARCHAR(100),
    KIERUNEK_FORMA_SKROT_MY VARCHAR(100),
    NR_DECYZJI       VARCHAR(100),
    CZY_PRZYJETEY    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),
    STOPNIEN_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 rekrutację 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
przewodniczącyIKR     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_nrteccki,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 odzwierciedlałyby wyselekcjonowanie danych z istniejącej struktury bazy. Zawierają tylko sztywne dane testujące środowiska latexowe. 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 w stanie je znaleźć i wykonać.

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 latexu, a nie jak one zostaną wykorzystane. Dokładny opis jak wygląda wykorzystanie takiego zapisu znajduje się w rozdziale 2. Dodać jeszcze należy że 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 w stanie znaleźć takie zapytanie w szablonie potrzeba je obudować w odpowiednią konstrukcję, która będzie odpowiadać wyrażeniu regularnemu zapisanemu w parze. 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 zamknięcie *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 szablonu, 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 *listasekretariatA* 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 teczek. 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 miesiące 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_nrtecki||' '||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 konfigurację 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 liniiki 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 kompilatora 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 *dblalexraport-config.bat*. Poniżej znajduje się wydruk z konsoli po uruchomieniu programu z krótkimi opisami poszczególnych procesów:

1. Ładowanie pliku konfiguracyjnego przez program:

```
Config loaded: dblalexraportconfig.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 niżej 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

a

5.1. Wymagania systemowe

5.2. Przygotowanie pliku konfiguracyjnego

5.3. Przygotowanie prostego szablonu

5.4. Dodatkowe polecenia w szablonach

5.4.1. Polecenie puste

5.4.2. Grupowanie

5.5. Uruchomienie programu

5.6. Rozwiązywanie problemów

6. Podsumowanie

Bibliografia