POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynieria Oprogramowania i Baz Danych

Inżynieria oprogramowania, procesów biznesowych i baz danych

Jakub Kisiała
Nr albumu s20108

Mocno typowany język zapytań dla rozwiązań bazodanowych korzystający z płynnego API

Praca magisterska

dr inż. Mariusz Trzaska

Warszawa, 06.2021

Słowa kluczowe: Hibernate, JPA, bazy danych, Java

Streszczenie

Obecnie mappery obiektowo-relacyjne są głównym wyborem, każdego programisty, który poszukuje rozwiązania dla połączenia swojej aplikacji z bazą danych. Korzyści płynącego z tego wyboru są znaczne. Zaczynając od zamaskowania czasami dotkliwych różnic pomiędzy modelami obiektowymi i relacyjnymi, a kończąc na cache'owaniu wyników zapytań. W języku Java najpopularniejszy mapperem jest biblioteka Hibernate. W procesie odpytywania bazy danych wykorzystuje ona język JPQL, który to jest głównym tematem niniejszej pracy. W zależności od posiadanego środowiska programistycznego programista posiada zróżnicowaną pomoc oferowaną przy tworzeniu zapytań w tym języku. Tworzone są Stringi, których poprawność weryfikowana jest na etapie wykonania programu. Powoduje to często znaczne trudności z rozwojem aplikacji, gdyż potencjalne literówki wymagają ciągłej poprawy.

Inspiracją oraz przykładem jak powinna wyglądać praca z mapperem, w zakresie tworzenia zapytań, może być EntityFramework z języka C#. Ta biblioteka posiada bardzo mocną kontrolę topologiczną pisanego zapytania. Osiągnięte to przez mechanizm wywoływania kolejnych metod oraz przekazywania im wartości biznesowych, których zapytanie dotyczy. W połączeniu ze środowiskiem programistycznym daje to możliwość solidnego wsparcia programisty już na etapie tworzonego kodu.

W pracy podjęto próbę stworzenia bliźniaczego rozwiązania dla języka Java, które ułatwiałoby prace z Hibernate. Wzorowano się na języku LINQ wykorzystywanym w EntityFramework. Głównym celem stawianym przed prototypem oprogramowania jest maksymalne uproszczenie procesu tworzenia zapytań, z jednoczesnym minimalnym ograniczeniem możliwości samego mapppera.

Abstract

Currently, object-relational mappers are the main choice of any developer who is looking for a solution to connect their application to the database. The benefits of this choice are considerable. Starting with masking the sometimes acute differences between object-oriented and relational models, and ending with caching the query results. In Java, the most popular mapper is the Hibernate library. In the process of querying the database, it uses the JPQL language, which is the main topic of this work. Depending on the programming environment, the programmer has a variety of help offered in creating queries in this language. Thongs are created, the correctness of which is verified at the stage of program execution. This often causes significant difficulties with the development of the application, as potential typos require constant improvement.

The EntityFramework from the C # language can be an inspiration and an example of how working with a mapper should look like in terms of creating queries. This library has very strong topological control of the written query. This is achieved by the mechanism of calling subsequent methods and passing them the business values that the inquiry concerns. In combination with the programming environment, it gives the possibility of a solid developer support already at the stage of creating the code.

The work attempts to create a twin solution for Java, which would facilitate work with Hibernate. The LINQ language used in the EntityFramework was modeled on. The main goal of the software prototype is the maximum simplification of the query creation process, while minimizing the capabilities of the mappper itself.

Spis treści

Streszczenie ... 3
Abstract ... 4
Spis treści ... 5
1. Wprowadzenie ... 7
1.1. Cel pracy ... 7
1.2. Rozwiazania przyjete w pracy ... 7
1.3. Rezultaty ... 7
1.4. Organizacja pracy ... 7
2. Języki zapytań ... 8
2.1. SQL ... 8
2.2. HQL i JPQL ... 9
2.3. LINQ ... 11
2.4. Fluent Query projekty Github ... 12
3. Rozwiązania programistyczne ... 15
3.1. Klasy anonimowe ... 15
3.2. Wyrażenia lambda ... 17
3.3. Java Generics ... 18
3.4. Interfejsy funkcyjne ... 19
3.5. Java STREAM API ... 20
3.6. Refleksja ... 21
4. Założenia ... 22
5. Implementacja ... 23
5.1. Przygotowanie klasy ... 23
5.2. Użycie ... 24
5.3. Pobieranie nazwy atrybutu ... 26
5.4. Kontrola przepływu ... 29
5.5. Proces generowania zapytań ... 30
5.6. Obsługa błędów ... 31
5.7. Dodatkowe funkcjonalności ... 32
6. Funkcjonalność ... 33
7. Podsumowanie ... 37
7.1. Założenia ... 37
7.2. Ograniczenia ... 37
7.3. Wnioski ... 39
Bibliografia ... 40
Dodatki ... 42

Dodatek A: Spis rysunków 42
Dodatek B: Spis tabel 42
Dodatek C: Spis listingów 42

1. Wprowadzenie

Praca porusza problem pisania zapytań w języku JPQL, który jest podzbiorem języka HQL. Oba te języki są wykorzystywana we współpracy z Hibernate – obecnie jednym z najpopularniejszych ORM-ów w języku Java.

1.1. Cel pracy

Głównym celem pracy jest utworzenie prototypu języka, który mógłby skutecznie wspierać programistę w procesie tworzenia zapytań w języku JPQL. Aby upewnić się że dostarczone rozwiązanie jest optymalne, wymagane jest również dokonania studium już istniejących rozwiązań.

1.2. Rozwiazania przyjete w pracy

Prototyp wykonano w języku Java w wersji 8 oraz testowano na popularnych bazach relacyjnych (Postgres, Oracle). Środowisko programistyczne użyte w procesie tworzenia oprogramowania to IntelliJ Ultimate.

1.3. Rezultaty

Prototypem utworzonym w ramach niniejszej pracy jest język Flux. Został on w całości wykonany w języku Java i jest on przeznaczony do użycia również tylko w Javie. Wprowadza on możliwość kontrolowania procesu tworzenia zapytania przez środowisko programistyczne. Uzyskano to przez zastąpienie pisania ręcznego poleceń, znanego z Hibernate, na rzecz łańcucha wywołań metod.

Dokonano również analizy możliwości biblioteki Flux. Nie ograniczono się jedynie do stworzonego prototypu, a także opisano czy koncepcja proponowanego rozwiązania jest możliwa do spełnienia w języku Java. W rozważaniach wzięto pod uwagę ograniczenia samego języka Java, jak również zakres lub też skale pracy programisty, jaka jest wymagana by je zniwelować podczas implementacji podobnych rozwiązań w projektach biznesowych.

1.4. Organizacja pracy

Praca rozpoczyna się przeglądem języków skorelowanych z tematyką pracy oraz istniejących rozwiązań w tym zakresie. Zawiera ona opis języka SQL oraz pochodnych JPQL i HQl, używanych w Hibernate. Następnie umieszczono opis języka LINQ, z którego zaczerpnięto ogólny pogląd na możliwy sposób użycia takiej biblioteki. Rozwiązania te w znacznym stopniu zaimplementowano w prototypie. Kolejny podrozdział zawiera opis obecnych rozwiązań, których działanie pokrywa się z celami stawianymi przed prototypem niniejszej pracy.

W kolejnym rozdziale dokonano przeglądu rozwiązań programistycznych, z których specyficznych właściwości korzysta prototyp. Duży nacisk położono na pewne aspekty, na których bazuje poprawność działania kluczowych algorytmów.

Następne dwa rozdziały dzielą się odpowiednio na założenia implementacyjne i opis samej implementacji. Sformułowano listę wymagań jakie musi spełniać finalny prototyp. W dalszej części zamieszczono szczegółowy opis proponowanych rozwiązań, których działanie zaprezentowano w kolejnym rozdziale.

Pracę zakończono szczegółowym podsumowaniem ze wskazaniem wad oraz zalet utworzonego języka.

2. Języki zapytań

Mianem języka zapytań określa się języki służące do odpytywania baz danych lub też systemów informacyjnych. Posiadają one zazwyczaj ściśle określoną strukturę. Najpopularniejszym z nich jest język SQL. Stał się on wspólnym językiem dla relacyjnych baz danych i jest częściowo obecny we wszystkich pozostałych językach zapytań. Podobieństwa można dostrzec chociażby w procesie formułowania zapytań i są one oczywistym uproszczeniem dla programistów, gdyż nie wymaga od nich zbyt dużego przygotowania do użycia takich języków jak JPQL czy też LINQ.

2.1. SQL

Rozwój relacyjnych baz danych skutecznie wyparł na długo inne koncepcje przechowywania danych. Jednocześnie wprowadził znaczny postęp w zagadnieniu odpytywania danych. Obecnie wszystkie silniki bazodanowe implementują język SQL. Oczywiście posiadają one swoje własne dialekty, ale na poziomie typowych zapytań są one w większości zgodne ze standardem z roku 1984 [1]. W opinii autora język ten zdobył swoją niezwykłą popularność dzięki przejrzystości. Jest on niezwykle zwięzły i intuicyjny.

SQL służy do odpytywania, manipulacji danymi oraz manipulacji strukturą danych. Składa się z wyraźnie wycielonych deklaracji zamkniętych w klauzulach. Składa się on z czterech głównych pod-języków:

- DQL – służący do odpytywania danych,
- DDL – służący do manipulacji strukturą danych,
- DCL – służący do manipulacji dostępem do danych,
- DML – służący do manipulacją danymi.

Głównymi zaletami języka SQL jest deklaratywna natura samego języka. Oznacza to, że użytkowani nie musi specyfikować w instrukcjach sterujących, w jaki sposób ma być wykonana dana operacja. W przypadku przeciwnym, programowaniu imperatywnym, istnieje potrzeba dokładnego określenia w jaki sposób wykonać dany krok programu. Aby korzystać z tego typu rozwiązań wymagana jest znajomość procesów zachodzących wewnątrz danego programu, aby świadomie sterować wykonaniem operacji. Deklaratywny kod pozwala na dowolną zmianę struktury programu bez konieczności przepisania danego rozwiązania. W takich przypadkach nie istnieje również potrzeba aby użytkownik koniecznie znała strukturę programu.

Tego typu rozdzielenie implementacji programu od języka API pozwala na szerokie zastosowanie różnych baz danych. Obecnie w przypadku użytkownik, który swobodnie posługuję się językiem SQL, może obsługiwać większość popularnych silników bazodanowych. Kolejnym niezwykle ważnym aspektem, bardziej biznesowym, jest możliwość stosunkowo łatwej wymiany bazy danych. Często w kodzie umieszczane są różne zapytania SQL, które w przypadku kontaktu imperatywnego zapewne byłaby potrzeba poprawić. Dzięki temu że w zakresie języka DQL nie zachodzą znaczne różnice pomiędzy najpopularniejszymi serwerami, zapytanie napisane dla jednego silnika bazodanowego, najpewniej będzie działać bez żadnych zmian na innych serwerach lub będzie wymagać tylko drobnych modyfikacji składniowych. Język DQL jest najbardziej znaną podgrupą języka SQL. Składnie języka przedstawiono w Listing 1:

- SELECT – określa kolumny, które mają zostać zawarte w odpowiedzi. Może zawierać również różne operacje wykonywane na kolumnach, takie jak wyliczenie lub też konkatenacja.
- FROM – opisuję z jakiego źródła mają zostać pobrane rekordy. Najczęściej są to tabele, ale mogą to być również widoki czy też tabele tymczasowe.

- JOIN – zawiera się w poprzedniej klauzuli. Doprecyzowuję w jaki sposób należy połączyć tabele oraz precyzuje rodzaj złączenia.
- WHERE – służy do filtrowania zwracanych wartości. Może zawierać warunki, które muszą zostać spełnione przez rekordy, które są zwracane przez zapytanie. W przypadku nie spełnienia warunków, rekord nie jest zawierany w odpowiedzi.
- GROUP BY – udostępnia możliwość grupowania wyników. W klauzuli zawarte są kolumny, po których zostaną pogrupowane rekordy wynikowe. Najczęściej występuje z funkcjami agregującymi w klauzuli `Select`.
- HAVING – pozwala na filtrowanie danych po grupowaniu, w przeciwieństwie do klauzuli `Where`, której filtrowanie poprzedza operacje grupujące.
- ORDER BY – pozwala ustalić sposób sortowania wyniku danego zapytania.

Listing 1. Schemat składni języka DQL

```
1. SELECT ...
2. FROM ...
3. [JOIN ...]
4. [WHERE ...]
5. [GROUP BY ...]
6. [HAVING ...]
7. [ORDER BY ...]
```

W Listing 2 przedstawiono przykładowe zapytanie. Większość zapytań kierowanych do silników bazodanowych posiada właśnie taką strukturę [2]. W pierwszej kolejności wykonywane jest złączenie tabel według klauzuli `Join`. Następnie wykonywane jest filtrowanie predykatami zawartymi w klauzuli `Where`. Wówczas następuje sortowanie oraz odczytanie wybranych kolumn.

Listing 2. Przykładowe zapytanie w języku SQL

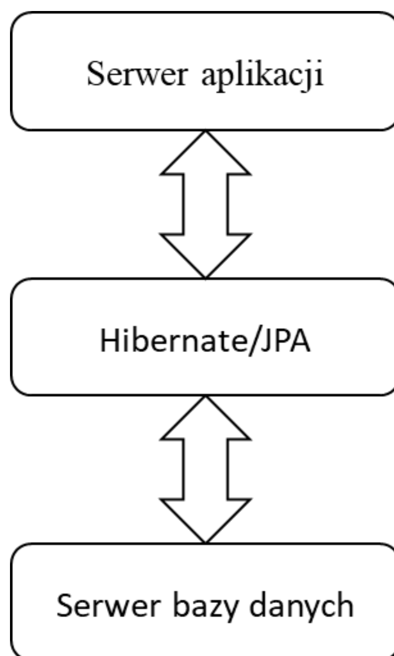
```
1. SELECT  m.id,  m.Title
2. FROM  Movie m
3. JOIN  m.genre g ON m.genreId = g.genreID
4. WHERE  g.name IN ( 'Horror', 'Thriller')  OR  m.Title != 'Matrix'
5. ORDER BY  g.id ASC,  m.id DESC
```

Obecnie język SQL stał się uniwersalnym, nie związanym z żadną konkretną platformą, językiem zapytań. Pojawiające się nowe koncepty baz danych, posiadające odmienny od relacyjnego model adanych, również zawierają API umożliwiające odpytywanie danych w języku SQL. Ogranicza to do pewnego stopnia unikalne możliwości danego rozwiązania, ale świadczy o poziomie popularyzacji tego języka zapytań.

2.2. HQL i JPQL

Transfer danych miedzy systemami relacyjnymi i obiektowymi stanowił od zawsze poważny problem [3]. Możliwym rozwiązaniem, lecz pracochłonnym, jest zastosowanie podstawowych sterowników JDBC i ręczne mapowanie każdego z zapytań. W takim przypadku ilość powtarzającego się kodu może być olbrzymia. Problem ten częściowo rozwiązały biblioteki mapujące model relacyjny na model obiektowy i odwrotnie. Dla języka Java powstał standard, opisujący jakie właściwości musi

spełniać taka biblioteka. Osiągnięto to przez stworzenie specyfikacji JPA (and. *Java Persistence Api*). Jest to zestaw adnotacji, które są wykorzystywane do opisu klasy. Najpopularniejszą biblioteką implementującą ten standard jest Hibernate [4]. W praktyce jest on warstwą pośrednią pomiędzy aplikacją i serwerem baz danych, zdejmujący z użytkownika problem mapowanie dwóch niekompatybilnych światów (Rys. 1).



Rys. 1. Schemat połączenia aplikacji i bazy danych

Język JPQL został wprowadzony przez standard JPA. Nie odbiega on znacznie od SQL, którym był mocno inspirowany. Składnia obu języków nie posiada znaczących różnic. Służy on do pisania zapytań do warstwy implementującej JPA z poziomu aplikacji. Natomiast Hibernate zaimplementował w swoim rozwianiu język JPQL pod postacią języka HQL (ang. *Hibernate Query Language*). Ponadto zawiera pewne uproszczenia składniowe, jaki i wprowadza kilka dodatkowych możliwości. Umożliwia między innymi obsługę leniwego ładowania (ang. *Lazy loadingi*). Każde zapytanie napisane w języku JPQL jest poprawnym zapytaniem HQL, natomiast odwrotnie nie jest to zawsze prawdą. Z tego też powodu oraz faktu, że nie rzadko ciężko jest odróżnić te języki od siebie, w dalszej części pracy autor ograniczył się wyłącznie do języka HQL.

Hibernate obsługuje zapytania w obu z tych języków. Składniowo język jest niezwykle podobny do SQL (Listing 3). Natomiast fundamentalną różnicą pomiędzy nim i SQL jest fakt, że HQL-em odpytywana jest warstwa pośrednia pomiędzy aplikacją, a serwerem baz danych, czyli Hibernate. Obsługuje on również zapytania w natywnym SQL, natomiast w przypadku użycia dedykowanego języka użytkownik otrzymuje wiele funkcjonalności w zamian. Jedną z nich i chyba najważniejszą jest tworzenie, często trudnych, deklaracji złączeń danych tabel. Hibernate do poprawnego działania wymaga odpowiednich adnotacji naniesionych na atrybuty danej klasy. To właśnie na podstawie tych danych potrafi on samodzielnie określić warunki złączenia danych encji. Po przeprocesowaniu zapytania Hibernate generuje natywne zapytanie SQL, które kieruję do bazy danych.

Listing 3. Przykładowe zapytanie w HQL, semantycznie identyczne z Listing 2

```
1. SELECT  m.id,  m.Title
2. FROM  Movie m
3. JOIN  m.genre g
4. WHERE  g.name IN ( 'Horror', 'Thriller')  OR  m.Title != 'Matrix'
5. ORDER BY  g.id ASC,  m.id DESC
```

Kolejną wyraźnym atutem tej biblioteki jest możliwość mapowania odpowiedzi z serwera baz danych. W przypadku klasycznego rozwiązania wymagane byłoby ręczne przepisanie wyniku na obiekt, co może być stosunkowo uciążliwe w przypadku większej ilości takich operacji. Hibernate udostępnia możliwość automatycznego mapowania wyników z bazy danych. Również jest to możliwe dzięki naniesienie odpowiednich adnotacji określających relacje pomiędzy obiektem i rekordem w tabeli.

2.3. LINQ

Poprzednie języki cechowała możliwość użycia z poziomu języka Java. Natomiast język LINQ jest dostępny tylko w języku C#. Natomiast był on główną inspiracją i wzorem funkcjonalności dla rozwiązań, które niniejsza praca prezentuje.

Język ten jest natywnym językiem oferowanym przez firmę Microsoft w rodzinie bibliotek .NET. Służy on do manipulacji kolekcjami. Natomiast klasyczną biblioteką mapującą dla języka C# jest EF (Entity Framework). Udostępnia on dane pochodzące z bazy danych w postaci kolekcji obiektów reprezentujących rekordy bazy danych. Dokumentacji języka sugeruje, że język LINQ używany do współpracy z EF jest podtypem głównego języka i nosi nazwę LINQ-to-Entities. Współpraca z kolekcjami „zwykłymi" oraz kolekcjami obiektów z bazy danych wykazuje pewne różnice semantyczne. Z tego też powodu szczególnie należy zaznaczyć, że w dalszej części niniejszej pracy nazwą LINQ określny jest szczególny podtyp języka służący do współpracy z EF.

Język LINQ pełni dokładnie tą samą funkcję we współpracy z EF jak HQL z Hibernate. Oba rozwiązania wykazują pewne różnice. Zapytania w HQL nie posiadają kontroli topologicznej. Na etapie deweloperskim są one umieszczane w cudzysłowach, gdzie często występują literówki, które są znajdowane na etapie wykonania programu lub też co gorsze nie są znajdowane w ogóle. LINQ oferuje mocno kontrole topologiczną oraz podpowiedzi na etapie tworzenia zapytania. Język ten jest znacznie bardziej przyjazny w procesie pisania zapytań [5].

LINQ wykorzystuje koncepcje *Fluent Api*. Oznacza to w praktyce pewną konwencję pisania zapytań. Na obiekcie, który dopuszcza użycie języka, wykonujemy szereg metod, których wywołania układają się w charakterystyczny łańcuch (Listing 4). Zwiększa to przejrzystość kodu oraz zmniejsza koszty czasowy jego utrzymania. Ponadto jest niezwykle intuicyjny w użyciu.

Listing 4. Przykładowe zapytanie w języku LINQ

```
1. entries.Where(e => e.Approved)
2.     .OrderBy(e => e.Rating).Select(e => e.Title)
3.     .FirstOrDefault();
```

Obiektem na rzecz, którego wykonywane są metody jest zbiór obiektów, które reprezentują dane. Następnie można wykonywać poszczególne metody, charakteryzujące odpowiednie operacje języka SQL. Zależnie od sposobu złączenia wywołani można uzyskiwać oczekiwane efekty. Brak potrzeby umieszczania nazw kolumn z bazy danych w cudzysłowie pozwala na lepszą kontrolę błędów na etapie produkcyjnym.

LINQ udostępnia również możliwość pisania zapytań w formie zbliżonej do języka SQL (Listing 5). Przepływ sterowania oraz możliwości takiej składni są bardzo podobne do standardowej składnie tego języka.

Listing 5. Zapytanie w języku LINQ w składni podobnej do języka SQL

```
1. var res = from l in my_list
2.               where l.Contains("my")
3.               select l;
```

2.4. Fluent Query projekty Github

Jedynym w pełni funkcjonalnym rozwiązaniem zbliżonym do poruszanego problemu w niniejszej pracy jest biblioteka JOOQ, a dokładniej jej niewielki fragment. Oprogramowanie to wspomaga proces współpracy z bazami danych i może działać samodzielnie bez standardu JPA. JOOQ do pewnego stopnia jest mapperem obiektowo-relacyjnym natomiast zdecydowanie mniej wszechstronnym niż Hiberante. Umożliwia:

- pobieranie danych – wspiera bezpośrednią komunikację z bazą danych,
- mapowanie struktur relacyjnych na obiektowe,
- tworzenie zapytań.

Dokumentacja biblioteki zawiera również wskazówki użycia JOOQ w połączeniu z JPA [6]. Wskazuje na współobecność obu rozwiązań, której to biblioteka ma dopełniać słabsze strony popularnego standardu Javy. Mianowicie wymieniono jako szczególną zaletę łatwość użycia do realizowania prostych zapytań, których to najczęściej w aplikacji jest przytłaczająca większość. Osiągnięto to przez udostępnienie API, które przyjmowało elementy generowanego kodu. Z drugiej strony wskazano JPA jako lepsze narzędzie do odtwarzania oraz aktualizacje skomplikowanych grafów zależności pomiędzy obiektami. Użycie JOOQ sprowadza się do prostych kroków:

- dodanie odpowiednich zależności do projektu,
- wygenerowanie encji, na podstawie bazy danych, z użycie generatora dostarczonego przez twórców biblioteki,
- utworzenie i zrealizowanie zapytania do bazy danych.

Przykładowe zapytania cechują się mocną kontrolą typów w trakcie tworzenia zapytania (Listing 6), która to jest ściśle związana z generowanym kodem. Na Listing 7 zamieszczono przykładową encję z wybranymi polami

Listing 6. Użycie JOOQ

```
1. DSLContext ctx = DSL.using(conn, SQLDialect.MYSQL);
2.
3. ctx.select(
4.              ACTOR.FIRSTNAME,
5.              ACTOR.LASTNAME,
6.              count().as("Total"),
7.              count().filterWhere(LANGUAGE.NAME
8.                           .eq("English")).as("English"),
9.              count().filterWhere(LANGUAGE.NAME
10.                          .eq("German")).as("German"),
11.             min(FILM.RELEASE_YEAR),
12..             max(FILM.RELEASE_YEAR))
13..        .from(ACTOR)
14..        .join(FILM_ACTOR).on(ACTOR.ACTORID
15.                                .eq(FILM_ACTOR.ACTORS_ACTORID))
16..        .join(FILM).on(FILM.FILMID.eq(FILM_ACTOR.FILMS_FILMID))
17..        .join(LANGUAGE).on(FILM.LANGUAGE_LANGUAGEID
18.                                .eq(LANGUAGE.LANGUAGEID))
19..        .groupBy(
20..             ACTOR.ACTORID,
21..             ACTOR.FIRSTNAME,
22..             ACTOR.LASTNAME)
23..        .orderBy(ACTOR.FIRSTNAME, ACTOR.LASTNAME, ACTOR.ACTORID)
24..        .fetch()
25..);
```

Główna wadą takiego rozwiązania jest potrzeba generowania kodu i umieszczanie go w plikach źródłowych jedynie dla JOOQ, gdyż encje JPA nie są w żaden sposób powiązane z wygenerowanymi encjami. Struktura klasy odbiega znacznie od standardu JPA i sama w sobie nie jest obiektem biznesowym, a jedynie metadanymi o nim. W przypadku użycia obu narzędzi oznacza to potrzebę utrzymywania dwóch zestawów reprezentujących kształt bazy danych, a co za tym idzie spójności pomiędzy nimi.

Listing 7.Kod wygenerowany do poprawnego działania biblioteki JOOQ

```
1. /**
2.  * This class is generated by jOOQ.
3.  */
4. @SuppressWarnings({ "all", "unchecked", "rawtypes" })
5. public class Actor extends TableImpl<ActorRecord> {
6.
7.     /**
8.      * The reference instance of <code>ACTOR</code>
9.      */
10.    public static final Actor ACTOR = new Actor();
11.
12.    /**
13.     * The class holding records for this type
14.     */
15.    @Override
16.    public Class<ActorRecord> getRecordType() {
17.        return ActorRecord.class;
18.    }
19.
20.    /**
21.     * The column <code>ACTOR.FIRSTNAME</code>.
22.     */
23.    public final TableField<ActorRecord, String> FIRSTNAME =
24.            createField(DSL.name("FIRSTNAME"),
25.                    SQLDataType.VARCHAR(255),
26.                    this,
27.                    "");
28.
29.
30.    /**
31.     * Create a <code>ACTOR</code> table reference
32.     */
33.    public Actor() {
34.        this(DSL.name("ACTOR"), null);
35.    }
36.
37.    //...
38.
39.}
```

3. Rozwiązania programistyczne

W niniejszej pracy zostały zastosowane pewne specyficzne właściwości języka Java. Wpłynęły one zarówno na możliwości jak na ograniczenia jakimi dysponuje prototyp języka Flux. Wykorzystanymi elementami są między innymi wyrażenia lambda czy też refleksja. Ich szczegółowe właściwości zostały opisane w niniejszym rozdziale.

3.1. Klasy anonimowe

Klasy anonimowe pozwalają na tworzenie zwięzłego kodu. Umożliwiają jednoczesną deklarację klasy oraz stworzenie jej instancji. Mogą być stosowane do tworzenia instancji klas implementujących interfejs lub też dziedziczących z klas. Przeważnie klasami nadrzędnymi są klasy abstrakcyjne, jednak można stworzyć również klasę anonimową dziedziczącą ze zwykłej klasy. Na Listing 8 użycia klas anonimowych w języku Java.

Listing 8. Przykład klasy abstrakcyjnej, klasy oraz interfejsu

```
1. abstract class Calculator {
2.     abstract int multiply(int a);
3. }
4.
5. class Computer {
6.     public int multiply(int a) {
7.         return a * 3;
8.     }
9. }
10.
11.interface Multipliable {
12.    int multiply(int a);
13.}
```

W Listing 8 umieszczono definicję trzech rodzajów elementów, które mogą być podstawą do utworzenia klasy anonimowej. Pierwszym z nich jest klasa abstrakcyjna z jedną abstrakcyjną metodą `multiply`. Kolejnym elementem jest interfejs z metodą o tej samej nazwie. Ostatnim z nich jest klasa `Computer` posiadająca już implementację klasy `multiply`.

Listing 9. Program tworzący klasy anonimowe

```
1. public class Main {
2.     public static void main(String[] args) {
3.
4.         Calculator calculator = new Calculator() {
5.             @Override
6.             public int multiply(int a) {
7.                 return a;
8.             }
9.         };
10.
11.        Multipliable multipliable = new Multipliable() {
12.            @Override
13.            public int multiply(int a) {
14.                return a * 2;
15.            }
16.        };
17.
18.        Computer computer = new Computer() {
19.            @Override
20.            public int multiply(int a) {
21.                return a * 4;
22.            }
23.        };
24.    }
25.}
```

Program umieszczony w Listing 9 pokazuje w jaki sposób tworzone są klasy anonimowe oraz, jednocześnie, ich instancję. W Listing 10 zaprezentowano wydruk działania programu, w którym wywołano metodę multiply(1) z argumentem o wartości 1. W wydruku zamieszczona jest również nazwa klasy obiektu, z którego została wykonana. Każda z klas anonimowych jest nazywana pochodnie od klasy nadrzędnej np.NazwaKlasyNadrzędnej$1.

Listing 10. Wydruk działanie programu prezentującego różnice pomiędzy klasami i interfejsami.

```
1. Calculator-abstract class:
2.  Class name:com.company.Main$1
3.  Value: 1
4. Multipliable-interface:
5.  Class name:com.company.Main$2
6.  Value: 2
7. Multipliable-class:
8.  Class name:com.company.Main$3
9.  Value: 4
```

Kod cechuje się uniwersalnością gdyż na podstawię klasy nadrzędnej, możliwe jest stworzenie dedykowanego obiektu do konkretnego zastosowania. Idealnym przykładem tego typu „kontenera” jest klasa `Thread`. Przesłonięcie metody `run()` pozwala na korzystanie z zasobu metod klasy nadrzędnej, jednocześnie ograniczając ilość pisanego kodu.

Istotnym aspektem również jest proces kompilacji klas anonimowych (Listing 11). Każda klasa anonimowa jest kompilowana do odrębnej nowej klasy. Czyli zastosowanie klasy anonimowej zamiast zwykłej klasy, nie ma wpływu na wydajność programu, a jedynie poprawia proces tworzenia oprogramowania

Listing 11. Klasa anonimowa po kompilacji [7]

```
1. //Przed kompilacją
2. public class AnonymousClassExample {
3.     Function<String, String> format =
4.  new Function<String, String>() {
5.         public String apply(String input) {
6.         }
7.     };
8. }
9.
10.//Po kompilacji
11.public class AnonymousClassExample$1 {
12.    public String apply(String input) {
13.         return Character
14.             .toUpperCase(input.charAt(0)) + input.substring(1);
15.    }
16.}
```

3.2. Wyrażenia lambda

Klasy anonimowe pozwalają uniknąć potrzeby tworzenia licznych klasy, które są stosowane w pojedynczych miejscach. Natomiast wyrażenia lambda, są naturalnym rozwojem tej koncepcji. Składnie wyrażenia lambda zaprezentowano w Listing 12. Ich zastosowanie wiąże się nierozłącznie z pojęciem interfejsów funkcyjnych (rozdział 3.4). Poprawiają one znacznie przejrzystość kodu oraz pozwalają one tworzyć bardzo elastyczne fragmenty kodu, które mogą być stosowane po dostarczeniu ciała poszczególnych funkcji. Gdyż można uprościć koncepcje wyrażeń lambda do tworzenia instancji danych funkcji, analogicznie do klas anonimowych.

Listing 12. Składnia wyrażenia lambda

```
1. Runnable anonymousClass = new Runnable() {
2.     @Override
3.     public void run() {
4.         //...
5.     }
6. };
7.
8. Runnable lambdaExpression = () -> {
9.     //...
10.};
```

Lambdy zostały wprowadzone do Javy w wersji 8. W kolejnych wersjach nie wystąpiły znaczne zmiany w ich działaniu. Wersja 11 wprowadziła możliwość używania konstrukcji `var` wewnątrz ciała lambdy, która to została wprowadzona w JDK 10. Natomiast w wersji 13 poprawiono błędy w kompilacji wyrażeń lambda dotyczące metody `getEnclosingMethod()` [8].

W języku Java wyrażenie lambda jest popularnie uważane za lukier składniowy (ang. *syntactic sugar*). Istnieją jednak badania, które wskazują na to że mogą one skrócić czas wykonania, w niektórych przypadkach. Np. w przypadku iterowania po dużych kolekcjach możliwy jest zysk na poziomie do 12% [9]. Wyniki jednak pokazują też że dla niektórych mniej licznych kolekcji zaobserwowano spadek wydajności. Badania zaprezentowane przez jednego z inżynierów wydajności firmy Oracle, wskazują precyzyjniej operacje i sytuacje, w których pojawiają się różnice [10]. Na korzyść lambd w stosunku do klas anonimowych szczególnie wpływają operacje czasochłonne i wielowątkowe. Klasy anonimowe mogą wypaść lepiej w przypadku operacji jednostkowych na jednym wątku. Różnice te wynikają z faktu że wyrażenie lambda nie są traktowane przez kompilator jako jedynie lukier syntaktyczny, a są tworem samodzielnym od klas anonimowych.

Każda z klas anonimowych pojawiające się w klasie jest kompilowana do odrębnego pliku i nazywana zgodnie z zasadami opisanymi w podrozdziale o klasach anonimowych niniejszej pracy (Listing 13). Wraz ze wzrostem liczby takich klas zwiększa się liczba odczytów dyskowych oraz zapełnia się pamięć maszyn wirtualnej (JVM) [7]. Wyrażenia lambda w języku Java są kompilowane jako statyczne metody klasy, w której występują (Listing 13).

Listing 13. Wyrażenia lambda po kompilacji [7]

```
1. //Przed kompilacją
2. Function<String, Integer> f = s -> Integer.parseInt(s);
3.
4. //Po kompilacji
5. static Integer lambda$1(String s) {
6.     return Integer.parseInt(s);
7. }
```

Biorąc pod uwagę zebrane informację, wady i zalety, zdecydowanie optymalniejszym rozwiązaniem jest korzystanie z wyrażeń lambda zamiast klas anonimowych. Pomimo faktu mniejszej wydajności w niektórych nieobciążonych rozwiązaniach, korzyści wypływające z przejrzystszego kodu nadal są znaczące.

W języku C#, w odróżnieniu od Java'y, wyrażenia lambda stanowią obiekty klasy `Expression`. Dzięki dostępowi do właściwości *Body* istnieje możliwość dostępu do ciała funkcji [5]. Pozwala to na analizę, do jakich atrybutów czy metod odwołuje się wyrażenie. Takie podejście pozwala znacznie zwiększyć możliwości lambd w kontekście dalszego upraszczania kodu. Ta z pozoru drobna różnica, która zostaje omówiona w rozdziale Implementacja, miała kluczowy wpływ na opracowanie prototypu języka zapytań w niniejszej pracy.

3.3. Java Generics

Języka Java posiada statyczne typowanie. Oznacza to że typy referencji muszą ściśle odpowiadać typowi obiektu, do którego referencja ta wskazuje. W przypadku chęci posiadania uniwersalnej referencji, która ma możliwość utrzymania pożądanych cech obiektu, można zastosować interfejs jako typ zmiennej. Natomiast w przypadku zagnieżdżenia atrybutów złożonych, zastosowanie interfejsu nie przyniesie pożądanych rezultatów. W takim przypadku można stworzyć jedynie szablon danej klasy i jej funkcjonalności.

Java Generics określa możliwość parametryzowania poszczególnych elementów w języku Java. Określenie parametru może dotyczyć zarówno klasy jaki i metody. W przypadku klasy deklaracja zaprezentowano w Listing 14.

Listing 14. Java Generics deklaracja klasy

```
1. class Car<T> {
2.     private T parameter;
3.
4.     public static <U> void drive(U u) {
5.     }
6. }
```

Określenie parametru następuje podczas tworzenia referencji do obiektu (Listing 15). Na tym przykładzie również widać cały sens stosowania sparametryzowanych klas. Możliwe jest stworzenie instancji klasy `Car` i sparametryzowania jej zawartości zgodnie z deklaracją. Atrybut `parametr` w każdym z przypadków jest innego typu. Pozwala to na korzystanie z właściwości klasy parametru po utworzeniu obiektu.

Listing 15. Java Generics zastosowanie

```
1. Car<String> car1 = new Car<>();
2. Car<Long> car2 = new Car<>();
3.
4. String car1Parameter = car1.getParameter();
5. Long parameter = car2.getParameter();
```

Metody natomiast można parametryzować na podstawie klasy atrybutu przekazanego do danej metody (Listing 14). Zastosowanie tego typu narzędzia programistycznego zwiększa możliwość pisania elastycznego kodu, który można ponownie użyć w innym miejscu.

3.4. Interfejsy funkcyjne

Programowanie obiektowe posiada pewne wady. Jedną z nich może być brak możliwości tworzenia instancji funkcji [11]. W takim przypadku wymagane jest przypisanie danej funkcji do danej klasy. Mogą tu wystąpić problemy semantyczne lub też błędy logiczne. Ponadto wiele z tych funkcji stosowana jest w pojedynczych miejscach, więc utrzymywanie odseparowanych dużych fragmentów kodu staje się uciążliwe. Język C++, który jest hybrydą języka obiektowego i funkcyjnego, pozwalał na stosowanie referencji do funkcji [12].

Język Java natomiast rozwiązał ten problem w 8 wersji przez zastosowanie interfejsów. W połączeniu z wyrażeniami lambda możliwe jest wygodne tworzenie obiektów implementujący dany interfejs. Stosując takie podejście funkcja zostaje zamknięta w obiekcie, do którego to bez problemu możemy utworzyć referencje. Taki typ interfejsu nazywany jest funkcyjnym. Deklaracje takiego interfejsu przedstawiono w Listing 16.

Listing 16. Przykład interfejsu funkcyjnego

```
1. interface Worker<T>{
2.  public void work(T t);
3. }
```

Dzięki zastosowaniu parametrów, każdy z interfejsów może być przystosowany do przyjmowania dowolnych klas. Takie możliwości same w sobie nie są przełomowe, natomiast możliwości, które płyną z osadzenia zestawu tego typu interfejsu w większym obiekcie, są znaczne. Łatwo jest wyobrazić sobie klasę realizującą proces, który potrzebuje typowych podprocesów i całego zestawu operacji

wykonywanych zawsze. Przez sparametryzowanie obiektu wyrażeniem lambda, możliwe jest pisanie kodu niezwykle generycznego.

Interfejsy funkcyjne można tworzyć samemu, jak również można skorzystać z zestawu typowych interfejsów dostarczonych przez producentów (Tabela 1). Są one również szeroko stosowane w przetwarzaniu strumieniowym, jakie oferuje Java.

Tabela 1. Niektóre interfejsy funkcyjne

Interfejs	Metoda abstrakcyjna
Predicate<T>	boolean test(T t)
Function<S, T>	T apply(S s)
Supplier<T>	T get()

3.5. Java STREAM API

W Javie występują liczne kolekcje, które posiadają jeden wspólny mianownik. Jest nim implementowanie interfejsu `Collection`. Umożliwia to do pewnego stopnia translację różnych kolekcji na inne. Częste manipulacje ich zawartością może być uciążliwe. Proste operacje jak filtrowanie, które jest niezwykle częstą operacją, wymaga napisania kilku linii powtarzalnego kodu. Bardzo przydatne operacje mapowania czy redukcji wymagają już całych bloków kodu oraz kilku referencji tymczasowych.

Z myślą o tych problemach powstały strumienie. Pozwalają one dokonywać przeróżnych operacji na kolekcjach. Składnie tego rozwiązania zaprezentowano w Listing 17. Dla stworzonej listy samochodów wykonano operacje filtrowania, mapowania oraz redukcji. Widoczna jest przejrzystość kodu. Pojawiają się również korzyści wydajnościowe [10]. Oczywiście efekt uzyskiwany jest przez zastosowanie sparametryzowanych interfejsów funkcyjnych oraz wyrażeń lambda w celu dostarczenia implementacji dla danej operacji [13].

Listing 17. Przykład strumieni funkcyjnych w Javie

```
1. List<Car<Long>> carList = new ArrayList<>(
2.         Arrays.asList(
3.                 new Car<>(),
4.                 new Car<>()
5.                 //  ...
6.         ));
7.
8. Long result = carList
9.         .stream()
10.         .filter(
11.                 c -> c.getParameter() < 10
12.         )
13.         .map(Car::getParameter)
14.         .reduce((aLong, aLong2) -> aLong + aLong2)
15.         .get();
```

Strumienie w języku Java łamią pewną konwencje i zacierają jasny podział pomiędzy cechami języków obiektowych oraz języków funkcyjnych. Pod spodem nadal tworzone są obiekty, natomiast podczas pisania kodu można mieć pewne wątpliwości czy jest to nadal programowanie obiektowe.

Pomijając problematykę natury teoretycznej, zmiany wprowadzone w Javie 8 dotyczące kolekcji były rewolucyjne. Żadna z następnych wersji nie wprowadziła tak znaczącej poprawy jakości tworzenia

kodu. Pomijając poprawę wydajności w pewnych aspektach [10], składania tego rozwiązania znacząco ułatwia późniejszą obsługę tego kodu. Na świecie strumienie zyskały dużą popularność [14].

Innym wartym uwagi aspektem jest sam styl kodu, który tworzy łańcuchy wywołań. Styl ten popularnie nazywany płynnym (ang. *Fluent*) zyskał również szerokie zastosowanie w języku konkurencyjnym – C#.

3.6. Refleksja

Java 8 również wprowadziła refleksje. Jest to zestaw narzędzi umożliwiających dynamicznej inspekcji lub też ingerencji w program z poziomu kodu. Można w ten sposób uzyskać listę atrybutów lub też wywołać metodę danego obiektu. Narzędzia te pozwalają na dostęp do każdego zakątka programu. Mogą wykonywać takie operację jak: dostęp do prywatnych atrybutów/metod tworzenie obiektów przez wywołania prywatnego konstruktora (Listing 18). Jest to zaskakująca funkcjonalność biorąc pod uwagę łamanie konwencji dotyczącej specyfikatorów dostępu. Natomiast korzyści płynące z dobrze stosowanej refleksji pozwalają tworzyć bardzo generyczny kod.

Listing 18. Metody pakietu java.reflection

```
1. Class<? extends Actor> actorClass = actor.getClass();
2.
3. Field[] declaredFields = actorClass.getDeclaredFields();
4. Constructor<? extends Actor> declaredConstructor =
5.         actorClass.getDeclaredConstructor();
6. Method[] declaredMethods = actorClass.getDeclaredMethods();
```

4. Założenia

Biblioteka Hibernate posiada znaczne niedogodności. Najpopularniejszym z nich, zaraz po wyczerpującej konfiguracji, jest odpytywanie danych językami JPQL oraz HQL. Problem ten do pewnego stopnia pozwoli rozwiązać wysublimowane środowisko programistyczne takie jak IntelliJ. Podpowiada ono w stopniu znacznym kolejne składowe zapytania, które pisane są w postaci String-ów w cudzysłowach. Jest to jedyne IDE znane autorowi, które wspiera aktywnie programistę w tym procesie. Niestety należy ono również do jednego z droższych narzędzi programistycznych. Jednym z najpopularniejszych środowisk open-source-owych jest program Eclipse. Nie posiada on praktycznie żadnego wsparcia w zakresie pisania zapytań do Hibernate.

Posiadając doświadczenie z pracy w różnych środowiskach oraz przestudiowując możliwe połączenia z Hibernate sformułowano najważniejsze z założeń funkcjonalnych oraz technicznych:

- **Wsparcie dla Hibernate** – głównym przeznaczeniem biblioteki będzie użycie wraz z popularnym mapperem. Za najważniejszy cel uznano maksymalne wykluczenie potrzeby używania języka JPQL w bezpośredniej formie.
- **Wsparcie dla innych dialektów SQL** – biblioteka powinna wspierać również użycie jej ze sterownikami JDBC co pozwalało by na szerokie zastosowanie we wszystkich systemach spadkowych. Oznaczałoby to również potrzebę uwzględnia możliwie różnych silników bazodanowych.
- **Mocna kontrola typologiczna** – elementy zapytania dotyczące struktury danych jak również samej struktury zapytania, nie mogą zawierać fragmentów w postaci String-ów umieszczonych na stałe w kodzie. Jest to najważniejsze z założeń, wyeliminowanie potrzeby pisania ręcznego zapytań, którego błędy wykrywane są na etapie wykonania programu. Ponadto wszystkie popularne IDE bardzo dobrze wspierają typowanie obiektów. Tak więc postawienie na mechanizm dostarczany powszechnie pozwoli niższym kosztem uprościć pracę z Hibernate-em.
- **Wszechstronność użycia** – biblioteka musi być przystosowana do najpopularniejszych asocjacji pomiędzy klasami, tak aby jej użycie było możliwe w większości przypadków. Nie może ona również na żadnym etapie ograniczać możliwość jakie oferuje Hibernate.
- **Łatwość użycia –** biblioteka musi cechować się przystępnością i łatwością konfiguracji.
- **Język Java** – z uwagi na fakt że problemy opisane w rozdziale HQL i JPQL (2.2) występują w Hibernate oferowanym na tej platformie.
- **JDK 8 –** z powodu wstecznej kompatybilności, tzn. kod w Javie 8 działa w środowisku Javy 11, postanowiono nie używać nowszych wersji języka. Pozwala to na ewentualne szerokie użycie gdyż, obecnie jest to najpopularniejsza wersja tego języka, działająca również we wszystkich nowszych wersjach. Poprzednie wersje, ewentualne niekompatybilne wykluczono z racji znacznego zubożenia walorów programistycznych języka. Podjęto taką decyzje mając na uwadze, że w systemach spadkowych (legacy) nie występują trudności w podniesieniu wersji środowiska do JDK 8.

Analizując podjęte założenia zdecydowanie można stwierdzić że wykluczenie konieczności pisania ciągów znaków pozbawionych sprawdzenia ze strony IDE jest kluczową miarą przydatności biblioteki do użycia. Na podstawie analizy obecnych zasobów dostępnych w dziedzinie mapowania obiektowo-relacyjnego można stwierdzić że przykładem dla składni oraz sposobu użycia biblioteki obrano języku LINQ. Odrzucono również zdecydowania projekty dostępne na platformie Github, które zawierały znaczną ilość nadmiarowego kodu służącego do uproszczenia pisania zapytań.

5. Implementacja

Prototyp języka Flux spełnia szereg założeń projektowych. Najważniejszym z nich jest kompatybilność z Java w wersji 8 oraz wyższej. Oznacza to również że biblioteka została wykonana w/w wersji tego języka. W tym rozdziale zostanie opisana pełna implementacja prototypu, a w szczególności kluczowych jego modułów. Użycie biblioteki można podzielić na kilka kluczowych etapów następujących w danej kolejności:

- **Przygotowanie klasy** – przegląd encji pod kątem możliwości użycia języka Flux, etap ten jest wymagany do wykonania prze programistę. Istnieje potrzeba odpowiedniego dostosowania metod oraz struktury klas, które mają zostać użyte we współpracy z biblioteką.
- **Stworzenie klasy bazowej** – utworzenie głównego obiektu języka Flux oraz odpowiednia parametryzacja.
- **Uzupełnienie informacji o strukturze zapytania** – programista przekazuje atrybuty do odpowiednich klauzul zapytani.
- **Wywołanie metody kończącej zapytanie** – dokładną strukturę tych kroków przedstawiono w podrozdziałach

Te kroki wymagane są do wykonanie przez programistę. Natomiast kilka ważnych procesów wykonywanych jest przez bibliotekę. Biblioteka wykonuje je w zadanej kolejności:

- **Zapisanie atrybutów klauzul** – do momentu wywołania akcji końca zapytania, dostarczane są parametry zapytania, które trafiają do głównego obiektu klasy *Flux*.
- **Ogólna walidacja zapytania** – wykonywane są podstawowe sprawdzenia zgodności struktury zapytania. Jeśli zachodzi taka potrzeba to w przypadku jednoznacznych braków, które można wywnioskować z kontekstu, uzupełnia się odpowiednie klauzule.
- **Translacja** – na podstawie zebranych parametrów tworzone jest zapytanie. Przebieg tego procesu zależy od konkretnego dialektu języka SQL. Możliwe jest również uzupełnienie o specyficzne zależności składniowe oraz walidacja indywidualna zapytania.
- **Zwrócenie wyniku** – wynikiem działania biblioteki może być zapytania w postaci *String'a* lub też przez integrację z mapperami np. Hibernate, możliwe jest zwrócenie kolekcji encji spełniających zapytanie.

5.1. Przygotowanie klasy

Biblioteka nie wymaga od użytkownika żadnych dodatkowych działań w kontekście encji, która jest przeznaczona do użycia w zapytaniu bazodanowym. Ograniczenia jakie narzuca jej użycie są nie zbyt liczne i w całości pokrywają się z ograniczeniami jakie posiada Hibernate. Jednakże główne dostrzeżone ograniczenia to (Listing 19):

- **Typy proste** – klasa musi składać się tylko i wyłącznie z typów referencyjnych. Ograniczenie to pokrywa się z zaleceniami jakie posiada wyżej wymieniony mapper. Związane jest to z używany mechanizmem refleksji. Występują znaczne trudności przy tworzeniu obiektów typów prostych z jej użyciem.

- **Konstruktor bezparametrowy** – klasa musi posiadać konstruktor bez parametrowy. Algorytm wyznaczający atrybut, którego dotyczy zapytanie, w trakcie wykonania, może tworzyć obiekty danej klasy jaki i klas atrybutów danej klasy.
- **Getter-y** – klasa musi posiadać getter dla każdego z atrybutów które użytkownik chce użyć w zapytaniu. Są one tworzone w oparciu o przekazanie metodzie danego pola przez getter. Można również korzystać z atrybutów publicznych jednak jest to nierekomendowane podejście, zgodne z pryncypiami programowania obiektowego.

Listing 19. Przykładowa klasa bazowa

```
1. @Entity
2. public class Genre {
3.
4.     @Id
5.     @GeneratedValue(strategy = GenerationType.IDENTITY)
6.     private Long id;
7.
8.     @Column
9.     private String name;
10.    @OneToMany(mappedBy = "genre")
11.    private Set<Movie> movies;
12.
13.    public Genre() {
14.        this.movies = new HashSet<>();
15.    }
16.
17.    public String getName() {
18.        return name;
19.    }
20.
21.    public void setName(String name) {
22.        this.name = name;
23.    }
24.
25.    //…
26.}
```

5.2. Użycie

Bazową klasą biblioteki jest klasa `Flux<>`. Każde przypadek użycia rozpoczyna się od utworzenia obiektu tej klasy. Klasa ta przypomina do złudzenia `DbSet<>` znany z języka C#. Klasa parametryzowana jest klasą encji, której to pobierania będzie dokonywać. Użycie rozpoczyna się od utworzenia obiektu klasy Flux, parametryzacji oraz przekazania dwóch parametrów (Listing 20):

- **Dialektu** – język w jakim biblioteka ma generować zapytania.
- **Fabryki** – implementacji interfejsu `Supplier`, który posłuży to tworzenia „czystych" obiektów wykorzystywanych w ewaluacji obiektów zapytania. Obiekt posiada jedną metodę `get()` zwracającą nowy obiekt.

Listing 20. Utworzenie głównego obiektu biblioteki

```
1. Flux<Movie> flux = new Flux<Movie>(Dialect.HIBERNATE, Movie::new);
```

Tak przygotowany obiekt jest gotowy do wielokrotnego użycia w biznesowym kontekście aplikacji. Przez wywołanie metod o analogicznych nazwach do klauzuli z języka SQL, przekazywana jest informacja o oczekiwanej zawartości odpowiedzi. Na tym etapie każdy użytkownik zaznajomiony w stopniu podstawowym z koncepcją języka SQL intuicyjnie będzie potrafił utworzyć swoje zapytanie.

Listing 21. Użycie języka Flux

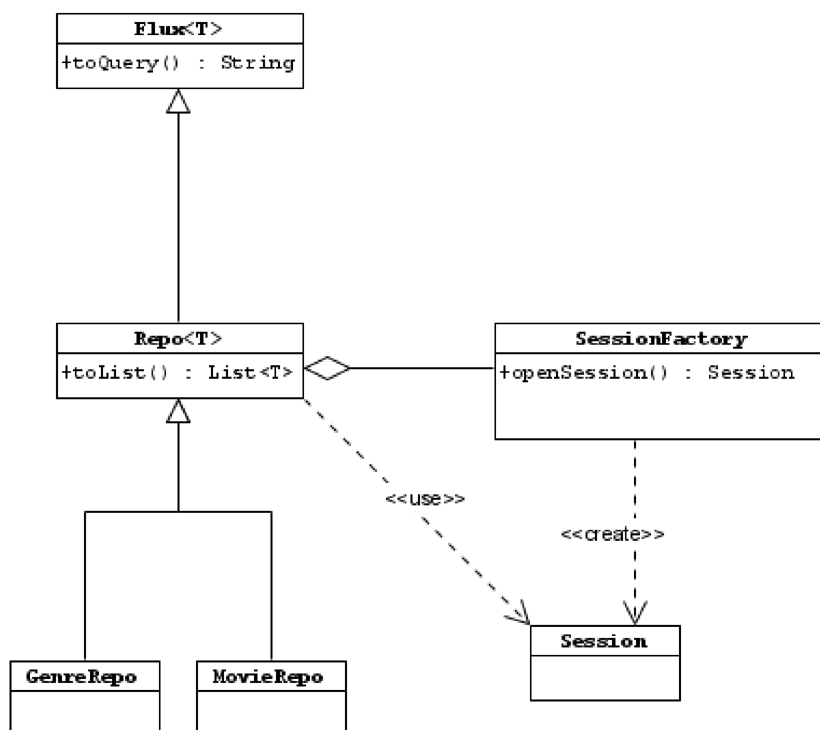
```
1. String query = flux
2.         .select(Movie::getId, Movie::getTitle)
3.         .where(e -> e.getGenre().getName())
4.         .in("Matrix", "Batman")
5.         .orderBy(e -> e.getGenre().getId())
6.         .asc()
7.         .orderBy(Movie::getId)
8.         .desc()
9.         .toQuery();
```

Wynikiem działania Listing 21 jest zapytanie w dialekcie wybranym przez programistę podczas tworzenia tego obiektu (Listing 20). Takie użycia zapewnia łatwość użycia oraz odseparowanie od technologii występujących w danym projekcie. Jest to duży atut biblioteki w przypadku wykorzystania jej w systemach spadkowych. Flux może zwracać natywne zapytania w języku SQL, które mogą być przekazywane do sterowników JDBC, które są podstawą do połączenia, aplikacji implementujących jeżyk Java z bazą danych. Obecnie jest to dosyć mało popularna architektura. Natomiast warto wspomnieć że wszystkie rozwiązania, służące do komunikacji z bazą danych umożliwiają wykorzystanie natywnego SQL.

Listing 22. Użycie architektury powiązanej z Hibernate

```
1. List<Movie> movies = movieRepo
2.                 .select(Movie::getId, Movie::getTitle)
3.                 .where(e -> e.getGenre().getName())
4.                 .in("Matrix", "Batman")
5.                 .orderBy(e -> e.getGenre().getId())
6.                 .asc()
7.                 .orderBy(Movie::getId)
8.                 .desc()
9.                 .toList();
```

Bardziej popularnym rozwiązaniem jest użycie mapperów obiektowo-relacyjnych. Dlatego też drugim podejściem do pracy z biblioteką Flux jest utworzenie fasady w postaci repozytoriów. Udostępniono proponowaną klasę bazową `Repo<>`. Może ona stanowić bazę dla repozytoriów. Jednakże mnogość możliwych koncepcji użycia i sposobów współpracy z bazą danych powoduje że jest to jedynie wskazówka, której implementacji trzeba dokonać indywidualnie. Na Listing 22 przedstawiono zapytanie wykonane z użyciem koncepcji repozytoriów. Główną różnicą względem Listing 21 jest fakt że programista posiada w tym miejscu już wynik zapytania. Natomiast w rozwiązaniu generującym zapytanie istnieje potrzeba wywołania w następnym kroku tegoż zapytania na wybranym obiekcie.



Rys. 2. Schemat połączeń pomiędzy głównymi klasami

Klasa `Repo<>` opakowuje sesje i udostępnia zestaw metod, które zwracają wynik zapytania (Rys. 2). W zależności od preferowanego sposobu działania na transakcjach można tworzyć oddzielne metody. W szablonowym repozytorium udostępniono jedną metodę `toList()` (Listing 23). Jej zadaniem jest opakowanie procesu otwarcia sesji, rozpoczęcia transakcji, faktycznego wywołania metody `getResultList()` na obiekcie reprezentującym sesje oraz zamknięcie transakcji.

Listing 23. Domyślna implementacja klasy toList()

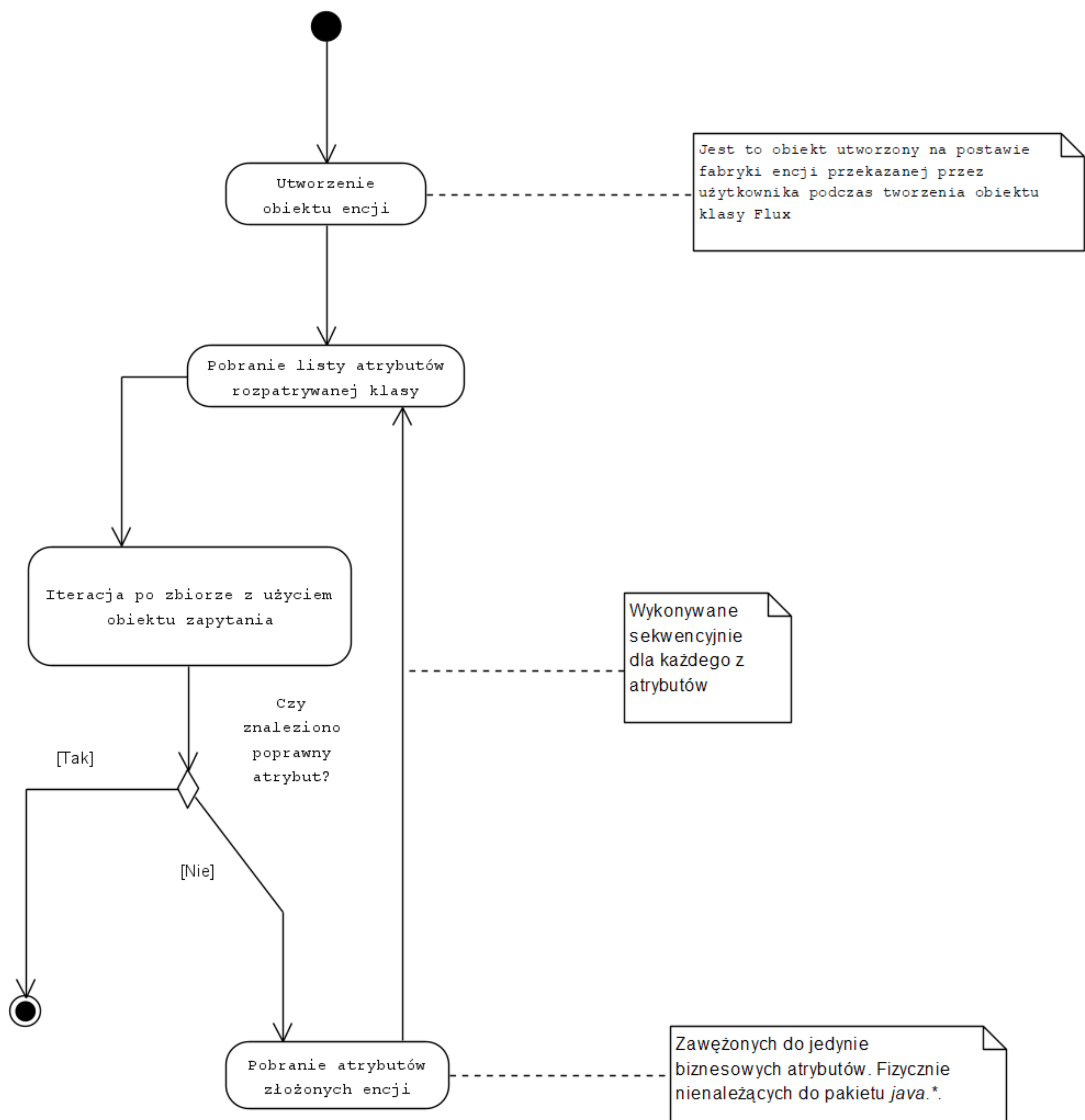
```
1. @Override
2. public List<T> toList() throws Exception {
3.     try (Session session = getSession()) {
4.         session.beginTransaction();
5.         List resultList = session
6.                                  .createQuery(toQuery()).getResultList();
7.         session.getTransaction().commit();
8.         return resultList;
9.     }
10.}
```

5.3. Pobieranie nazwy atrybutu

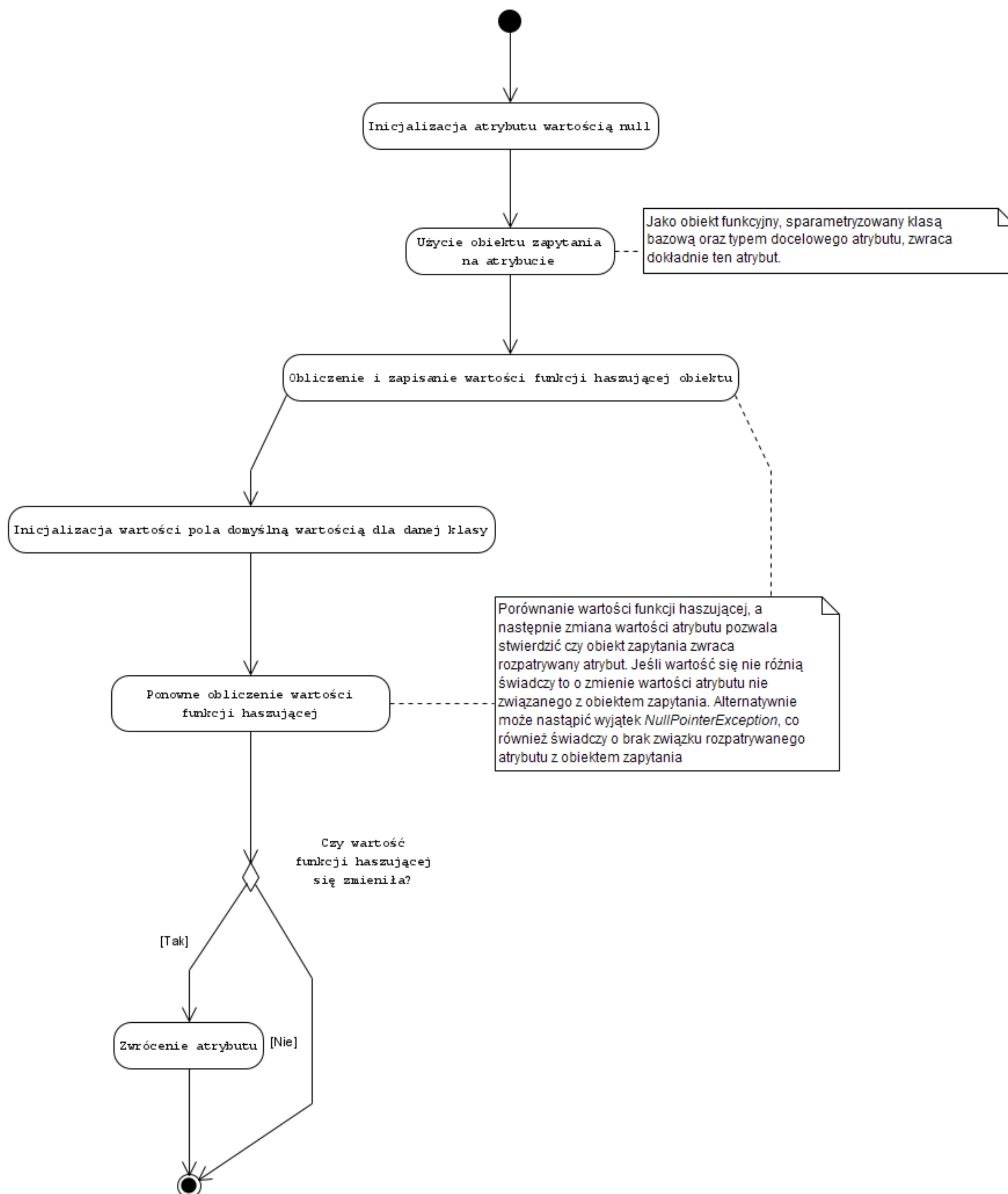
Centrum biblioteki jest moduł służący do wyznaczania składnika obiektu lub relacji jakiego dotyczy zapytanie. Opiera się ono o nowatorski algorytm pobierania atrybutu danej klasy, w całości oparty o mechanizm refleksji. Warto w tym miejscu wspomnieć o potrzebie użycia minimalnie 8 wersji języka Java, gdyż to właśnie w tej wersji przedstawiono programistom możliwość korzystania z całkiem nowego pakietu *java.reflect*. Koncepcja na której bazuje algorytm jest trywialna. Użytkownika przez

lambdę lub też referencję metody przekazuje obiekt reprezentujący implementacje interfejsu funkcyjnego. Tzw. obiekt referencyjny

Następnie tworzony jest obiekt danej klasy. Wstępnie pola inicjalizowane są wartością `null`. Zamyka to proces przygotowania obiektu do przeprowadzenia analizy, w której wyniku jest zwracany atrybut klasy, którego dotyczy dana implantacja interfejsu. W następnym kroku wykonuję się iterację po atrybutach obiektu (Rys. 3). Każdą iterację poprzedza użycie obiektu referencyjnego, w którego wyniku przekazana zostaje referencja do danego atrybutu klasy, której ów referencja dotyczy. Następnie wyliczana jest wartość funkcji `hashCode()` dla danego pola. Korzystając z faktu że jeśli obiekt referencyjny, pokryje się z aktualnym atrybutem klasy, w wyniku iteracji po tychże atrybutach, wynik obliczonej funkcji `hashcode()` będzie się różnił od wyniku tej samej funkcji po zainicjalizowaniu wartości pola z `null` na obraną stałą wartość w zależności od typu danej referencji (Rys. 4). Dla przykładu dla typów numerycznych jest to wartość „1" a dla typów napisowych jest to popularna „Ala ma kota".



Rys. 3. Pętla sprawdzenia atrybutów



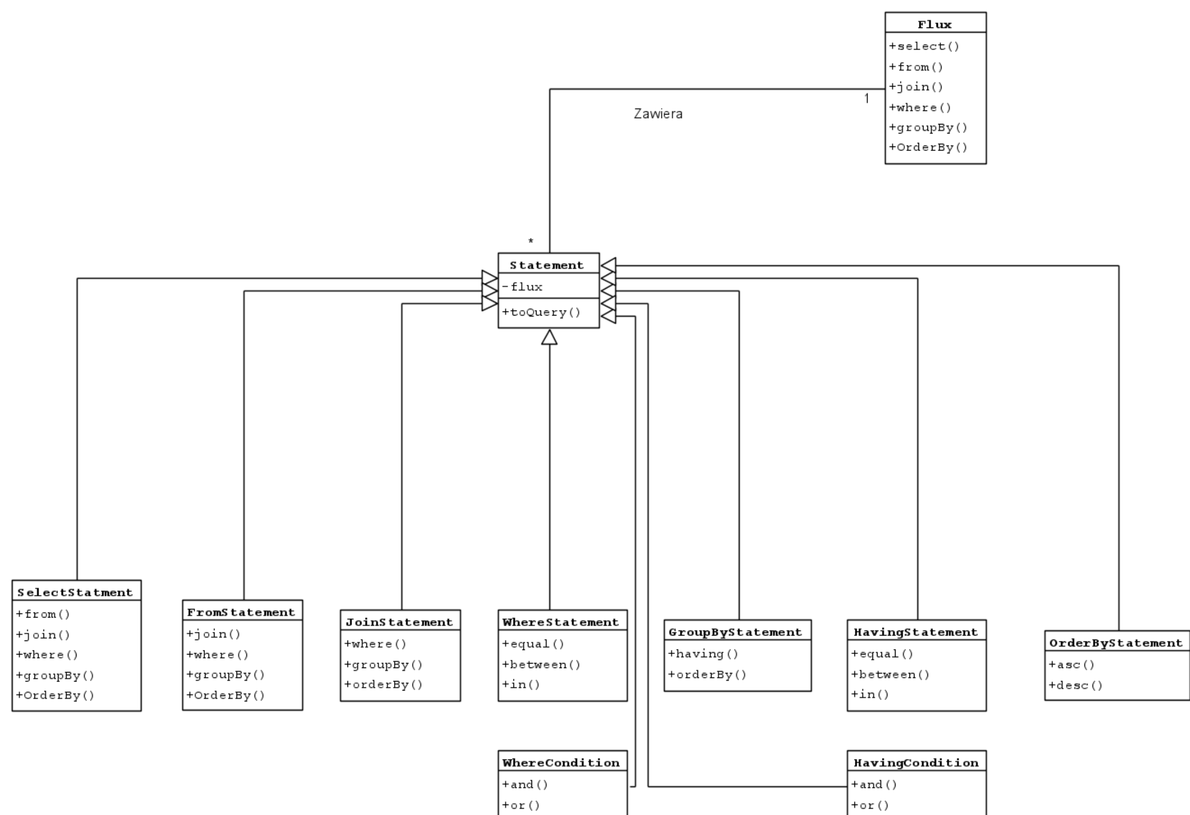
Rys. 4. Weryfikacja pobranego atrybutu

5.4. Kontrola przepływu

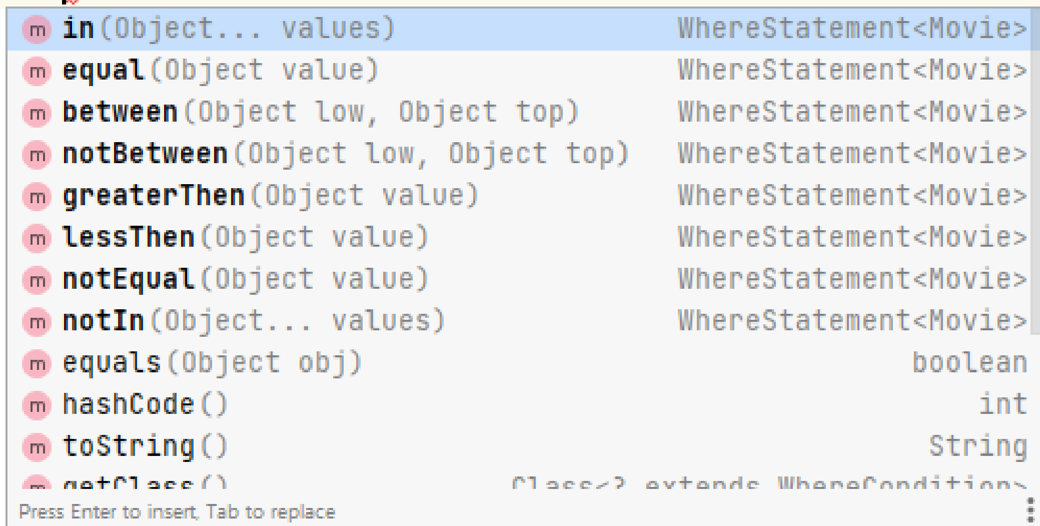
Kolejnym aspektem wartym uwagi jest odpowiedni interfejs biblioteki. Zapewniający odpowiednie wsparcie użytkownika, pokazujący w sposób przejrzysty możliwości swojego działania. Zapewnienie takiej funkcjonalności wyklucza również w stopniu znacznym możliwość złego użycia biblioteki. Programiści spędzają znaczną część pracy na czytaniu dokumentacji różnych, coraz to nowszych narzędzi. Odpowiedni interfejs biblioteki może pozwolić na przyspieszenie procesu produkcji oprogramowania.

W języku Flux, z racji jego płynnego charakteru, postanowiono zapewnić dostępność odpowiednich metod w danym kontekście zapytania. Efektem takiego zabiegu jest wykluczenie tworzenia potencjalnie błędnych zapytani oraz też wskazywanie użytkownikowi jakiej metody może lub powinien użyć. Zadbano również aby zapytania posiadały identyczną kolejność klauzul jak język SQL. Pozwala to na intuicyjne czytanie zapytań.

Efekt ten osiągnięto przez zastosowanie szeregu klas. Wywołanie każdej z metod odpowiedzialnych za konkretny element zapytania zwraca obiekt innej klasy, posiadającej odpowiednie metody, jakich można użyć w danym kontekście (Rys. 5). Oznacza to że użycie metody `where()` zwróci obiekt klasy `WhereStetament`, posiadający szereg możliwych metod służących do poprawnego zamknięcia warunku np. `between()` (Rys. 6).



Rys. 5. Klasy służące do kontroli przepływu podpowiedzi podczas pisania zapytania



Rys. 6. Podpowiedzi środowiska programistycznego w czasie pisania zapytania

5.5. Proces generowania zapytań

Wywołanie metody `toQuery()` klasy `Flux` rozpoczyna proces tworzenia zapytania, na podstawie zgromadzonych informacji w poprzednich krokach. W tym miejscu wprowadzono możliwość łatwego rozszerzenia funkcjonalności biblioteki. Osiągnięto to dzięki udostępnieniu szeregu interfejsów z których bazowym jest Translator z metodą `translate()`. Argumentem wywołania tej funkcji jest obiekt klasy `Flux`. Programista może dostarczyć własne implementacje tego interfejsu aby móc personalizować użycie biblioteki. Takie rozwiązanie zapewnia kompatybilność z potencjalnie różnymi mapperami czy też silnikami baz danych, których dialekty języka SQL różnią się między sobą.

Interfejsy są podzielone logicznie zgodnie z typami klauzul oraz dodatkowo występuje w/w interfejs `Translator` (Listing 24). Aby utworzyć własny moduł tłumaczący zapytania wystarczy dostarczyć jedynie implementacji ostatniego interfejsu. Natomiast wyszczególniono ten interfejs w celu uporządkowania danych w zapytaniu. Mogą to być np. brakujące klauzule `JOIN`, których użytkownik nie wprowadził, a odniósł się do pola z innej klasy. Natomiast interfejsy typu `WhereTranslator`, po dokonaniu niezbędnych poprawek zależnie od dialektu, otrzymują obiekty klasy opakowującej atrybuty klas. Są one przekazywane do metody `translate()`, której argumentami są podklasy `FieldHolder`. Wynikiem wywołania funkcji jest klauzula, która następnie w głównym obiekcie klasy `Translator` łączy wyniki działania każdego z interfejsów w gotowe zapytanie.

Listing 24. Interfejs modułu tłumaczącego

```
1. public interface Translator {
2.
3.     public String translate(Flux<?> flux) throws QueryException;
4.     public String translateSelect(List<SelectFieldHolder> fields);
5.     public String translateFrom(List<FromFieldHolder> fromFields);
6.     public String translateJoin(List<JoinFieldHolder> statementList);
7.     public String translateWhere(List<WhereFieldHolder> whereState-
   ment);
8.     public String translateGroupBy(List<GroupByFieldHolder> groupBy-
   Fields);
9.     public String translateHaving(List<HavingFieldHolder> having-
   Fields);
```

Prototyp języka Flux posiada ukończony moduł tłumaczący do jednego dialektu – JPQL. Po dogłębnej analizie zauważono, że aby utworzyć zestaw interfejsów dla języka SQL, wymagane jest również opracowanie analizatora struktury klas. Wymaga tego różnorodne sposoby odzwierciedlenia struktur relacyjnych w strukturach obiektowych. Natomiast język JPQL natywnie jest wspierany przez bibliotekę Hibernate, która to zajmuje się poprawnym przetłumaczeniem zapytania do natywnego SQL w zależności od wybranego sterownika.

5.6. Obsługa błędów

Bibliotek Flux posiada również moduł sprawdzający poprawność zapytań. Podstawowe sprawdzenia zostały zawarte w enumie `QueryChecklist`. Zawarto w nim kilka kluczowych sprawdzeń, między innymi:

- Zgodność typów klauzuli `Where` – obiekt przekazany jako argument do porównania w tej klauzurze musi być zgodny z typem pola, do którego nawiązuje.
- Zgodność klauzuli `Select` z `Group By` – używając funkcji grupującej sprawdzana jest czy wybrano wyłącznie odpowiednie kolumny do wyświetlenia.

Zgrupowanie sprawdzeń w jednej klasie pozwala na łatwą kontrolę nad pracą biblioteki. Zapewnia to przejrzyste zasady walidacji oraz w razie potrzeb umożliwia użycie tych samych warunków w wielu miejscach kodu. Natomiast dialekty języka SQL posiadają również swoje własne specyficzne ograniczenia. Z myślą o tym fakcie powstał interfejs `Checkable`. Jest to przepis na stworzenie własnych spersonalizowanych sprawdzeń. Użycie ich zazwyczaj jest wymuszone w trakcie tworzenia zapytania, dlatego też umiejscowienie ich wywołań naturalnie musi znaleźć się w translatorze dla danego dialektu. Wymusza to indywidualną implementacje oraz obsługę przez programistę, który chciałbym napisać własny translator. Oznacza to brak ograniczeń wyżej wymienionym interfejsem, a jedynie wskazanie na spójną architekturę.

W razie wystąpienia błędu składni ogólnej następuje zgłoszenie wyjątku klasy `QueryException`. Dziedziczy ona z klasy wyjątków wykonania programu. Oznacza to brak konieczności deklaracji zgłaszania wyjątku na etapie tworzenia kodu, co pozwala na pominięcie bloków obsługujących wszystkie błędy związane z użyciem języka. Dzięki takiemu rozwiązaniu to programista może sam zdecydować czy chce obsłużyć dany wyjątek czy też doprowadzić do przerwania działania programu, w razie jego zgłoszenia.

5.7. Dodatkowe funkcjonalności

W prototypie zaimplementowano również kilka dodatkowych funkcjonalności, przygotowujące je do prostszego użycia oraz integracji z popularnymi frameworkami. Są to głównie:

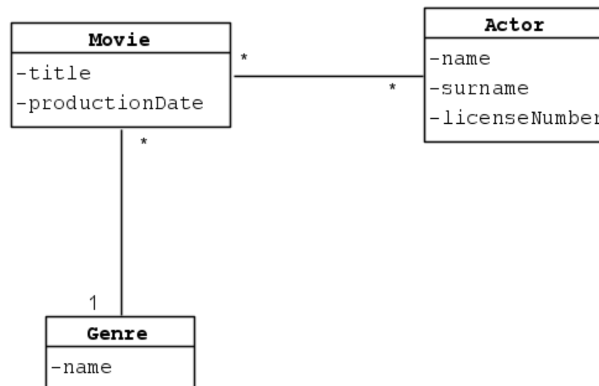
- **Mechanizm właściwości** – wprowadzenie pliku konfiguracyjnego, który umożliwi prostą konfiguracje, głównych parametrów systemu.
- **Mechanizm logowania** – ułatwiającego wykrywanie błędów oraz kontroli poprawności działania programu.

System właściwości pozwala na indywidualną konfigurację, zależnie od sposobu użycia. Mechanizm ten jest bliźniaczy do znanego ze Spring'a *application.properties*. Programista przez wpisy w postaci klucza i wartości przekazuje zadane parametry. W przypadku braku wpisu dla danych właściwości przyjmowane są domyślne właściwości.

Logowanie zaimplementowano z użyciem popularnego obecnie biblioteki-interfejsu *slf4j*. Rozwiązanie to daje szeroki wybór dostosowania do różnych rodzajów logowania, gdyż biblioteka ta sama w sobie nie jest loggerem a jedynie jest interfejsem do wielu popularnych loggerów lub też umożliwia podpięcie własnej biblioteki logującej. W przypadku braku wskazania biblioteki z implementacją wykorzystywana jest domyślna biblioteka *log4j*. Natomiast w przypadku pracy z np. Springiem ustawienia logowania zostaną pobrane z globalnych ustawień dla tworzonego projektu.

6. Funkcjonalność

W niniejszym rozdziale zaprezentowano biznesowe użycie języka Flux. Na Rys. 7 przedstawiono klasy wchodzące w skład przykładowej bazy filmów. Składają się na nią trzy tabele: gatunek, film oraz aktor. Podany przykład pozwoli w pełni zaprezentować możliwości prototypu.



Rys. 7. Schemat testowego programu

Dla tak skonstruowanej dziedziny biznesowej wymagane jest odpowiednie przygotowanie konfiguracji projektu. Należy rozpocząć od umieszczenia na ścieżce plików źródłowych lub też skorzystania z repozytorium Maven'a. W tym przypadku skorzystano z drugiego rozwizania (Listing 25). Następnym krokiem jest utworzenie zbioru danych testowych, które umożliwią przeprowadzanie testów (Listing 26).

Listing 25. Wpis w pom.xml

```
1. <dependency>
2.     <groupId>org.flux</groupId>
3.     <artifactId>FluentQueryHibernate</artifactId>
4.     <version>1.0</version>
5. </dependency>
```

Listing 26. Populacja danych

```
1. Genre romance = new Genre("Romance"),
2.         horror = new Genre("Horror");
3.
4. Movie titanic = new Movie("Titanic", romance),
5.         dirtyDancing = new Movie("Dirty dancing", romance),
6.         theRing = new Movie("The Ring", horror);
7.
8. Actor diCaprio = new Actor("Leonardo", "DiCaprio", titanic),
9.         winslet = new Actor("Kate", "Winslet", titanic),
10.        breslin = new Actor("Abigail", "Breslin", dirtyDancing);
```

Po stworzeniu populacji danych, środowisko programistyczne podpowiada sposób utworzenia obiektów głównych języka Flux. W testowym przypadku postanowiono skorzystać z domyślnej

implementacji repozytorium dostarczonej wraz z biblioteką. Inicjalizację tych obiektów przedstawiono na Listing 27.

Listing 27. Utworzenie repozytoriów dostępowych dla każdej z klas

```
1. Repo<Movie> movieRepo = new Repo<>(Dialect.HIBERNATE, Movie::new);
2. Repo<Genre> genreRepo = new Repo<>(Dialect.HIBERNATE, Genre::new);
3. Repo<Actor> actorRepo = new Repo<>(Dialect.HIBERNATE, Actor::new);
```

Poniżej zamieszczono test poprawność konfiguracji projektu (Listing 28). Pobrano ekstensje klasy *Movie*, a następnie dokonano wydruku wyników działania programu. Zamieszczono również zapytanie w języku HQL, jakie biblioteka Flux wygenerowała na podstawie przekazanych danych. Zgodnie z brakiem narzuconych warunków zapytania wyświetlił się cały utworzony zbiór filmów. Na tym etapie można z całą pewnością stwierdzić że projekt testowy został skonfigurowany poprawnie.

Listing 28. Pobranie ekstensji filmów

```
1. List<Movie> movies = movieRepo.toList();
```

Listing 29. Wygenerowane zapytanie w języku HQL

```
1. FROM
2.   Movie m
```

Listing 30. Wynik działania zapytania

```
1. Movie:
2.  id: 1
3.  Title: Titanic
4.  Genre: Romance
5. Movie:
6.  id: 2
7.  Title: Dirty dancing
8.  Genre: Romance
9. Movie:
10. id: 3
11. Title: The Ring
12. Genre: Horror
```

Następnie wykonano serie testów, w których podano analizie najczęstsze typy zapytań. Obok zamieszczono ich wyniki. W Listing 31 zamieszczono przykładowe zapytanie z warunkiem filtrującym dane. Wygenerowane zapytanie zamieszczono w Listing 32. W tym przypadku warunek `Where` jest rozbudowany, natomiast biblioteka obsługuje również takie przypadki. Listing 33 przedstawia interesujący przypadek zapytania z warunkiem filtrującym. Zazwyczaj aby odnieść się do atrybutów klasy, z którą obiekt jest związany, należy umieści w języku HQL ręcznie złączenie. Biblioteka wykonuje tą operacje samodzielnie i nie ma potrzeby ręcznego dopisania takiej klauzuli (Listing 34).

Listing 31. Prezentacja kompleksowego warunku zapytania

```
1.  List<Actor> movies = actorRepo
2.                  .where(Actor::getId)
3.                  .notEqual(1L)
4.                  .and(Actor::getLicenseNumber)
5.                  .lessThen(500)
6.                  .or(Actor::getSurname)
7.                  .notIn("DiCaprio","Winslet")
8.                  .toList();
```

Listing 32. Wydruk działania kodu z Listing 31

```
1.  FROM
2.   Actor a
3.  WHERE
4.   a.id != 1
5.   AND
6.   a.licenseNumber < 500
7.   OR
8.   a.surname NOT IN ( 'DiCaprio', 'Winslet')
```

Listing 33. Zapytanie z automatycznie generowanym warunkiem JOIN w języku Flux

```
1.  List<Movie> movies = movieRepo
2.                  .where(m->m.getGenre().getName())
3.                  .equal("Horror")
4.                  .toList();
```

Listing 34. Wydruk działania kodu w postaci zapytania JPQL, z Listing 33

```
1.  FROM
2.   Movie m
3.  JOIN
4.   m.genre g
5.  WHERE
6.   g.name = 'Horror'
```

Listing 35. Klauzula GROUP BY

```
1.  movieRepo
2.      .select(Movie::getId, Movie::getTitle)
3.      .groupBy(Movie::getGenre)
4.      .toQuery()
```

Listing 36. Wydruk działania kodu w postaci zapytania JPQL, z Listing 35

```
1. SELECT
2.  m.id,
3.  m.Title
4. FROM
5.  Movie m
6. GROUP BY
7.  m.genre
```

Listing 35 i Listing 36 przedstawiają użycie funkcji grupujących. Rezultatem wykonania takiego zapytania HQL jest zbiór wierszy. Następnie programista musi ręcznie zmapować wyniki działania takiego zapytania. Mnogość form możliwego wyniku znacznie utrudnia generyczne rozwiązanie takiego problemu. Możliwych rozwiązań jest kilka, np. posłużyć się klasami typu DTO lub też wykonanie ręcznego mapowania. W obu przypadkach istnieje potrzeba wykonania pełnego odczytu wyniku i wykonanie pewnych dodatkowych operacji związanych z poprawnym przetworzeniem danych.

7. Podsumowanie

W niniejszym rozdziale umieszczono ogólne wnioski dotyczące możliwości prototypu. Zweryfikowano również możliwość spełnienia założeń bazując na wynikach testów działania języka Flux.

7.1. Założenia

Kluczowym aspektem każdego oprogramowania wyznaczającym stopień jego użyteczności jest zgodność z określonymi wymaganiami oraz założeniami. Prototyp języka Flux stworzony w ramach niniejszej pracy spełnił większość założeń. Prostota użycia oraz wszechstronność z natury jest oceną subiektywną, natomiast w opinii autora biblioteka spełnia te wymagania również.

Po dogłębnej analizie zrezygnowano również ze spełnienia warunku pełnego wsparcia dla innych dialektów SQL. Głównym argumentem, była potrzeba stworzenia solidnego modułu analizującego i mapującego strukturę obiektową klasy na jej reprezentacje relacyjną. W przypadku użycia języka Flux z podstawowymi sterownikami JDBC, bez bibliotek JPA, wymagana byłaby ręczna adnotacja encji. Jest to krok wymagany w celu jednoznacznej translacji danych pomiędzy modelami. Natomiast Hibernate wymaga wykonania tego kroku z założenia. W dodatku korzysta on z wszechstronnych adnotacji w/w JPA. Zapewne w przypadku powstania analizatora dla systemu spadkowych, język Flux również zostałby oparty o te adnotacje. Pozostaje kwestią subiektywnej oceny czy w takim przypadku, poniekąd sporej pracy włożonej, nie warto jest również skorzystać z tych adnotacji włączając do aplikacji wsparcie Hibernate. W opinii autora taki moduł nie jest jednoznacznie wszechstronny i łatwy w użyciu, a więc nie został zawarty w finalnym prototypie. Jednakże system interfejsów pozwala na dowolną rozbudowę biblioteki w razie przyszłych potrzeb.

7.2. Ograniczenia

W trakcie testowania języka Flux zaobserwowano szereg miejsc, które mogą być źródłem potencjalnych błędów. Dodatkowo stwierdzono również, że błędy te mogą z łatwością pozostać nie zauważone przez programistę na etapie wstępnych testów.

Głównym źródłem niejednoznaczności jest moduł pobierania nazwy atrybutu. Mechanizm ten wykazuje się pewną błędogennością, związaną z faktem, że nie sprawdza poprawności przekazanych danych. Posłużenie się mechanizmem `hashCode` w zasadzie nie stwierdza jednoznacznie faktu, że zapytanie dotyczy danego atrybutu, a jedynie że jest on związany z obiektem zapytania. Przykład zamieszczono w Listing 37. Wynik działania takich zapytań jest identyczny (Listing 38), co może być mylące.

Listing 37. Test poprawności zapytań

```
1. String query1 = flux
2.         .select(m -> m.getTitle())
3.         .toQuery();
4. String query2 = flux
5.         .select(m -> m.getTitle().equals("Matrix"))
6.         .toQuery();
```

Listing 38. Wynik testu poprawności zapytań

```
1. query1: SELECT
2.      m.Title
3. FROM
4.      Movie m
5. ------------------
6. query2: SELECT
7.      m.Title
8. FROM
9.      Movie m
```

Pomijając sensowność poprzedniego przykładu użycia i umieszczenia wywołania `equals` w warunku, identyczny problem dotyczy atrybutów pochodnych. W Listing 39 zamieszczono przykład filtrowania po nazwisku oraz atrybucie pochodnym, imieniu i nazwisku. Moduł zwracający atrybut zapytania napotkał różnice wartości funkcji haszujących po zainicjalizowaniu nazwiska w obu przypadkach, natomiast w drugim przypadku była to jedynie składowa obiektu zapytania. Oba zapytania skutkują jednakowym zapytaniem HQL, które w drugim przypadku jest błędne.

Listing 39. Przypadek błędnego pobrania atrybutu

```
1. String query3 = actorRepo
2.         .where(Actor::getSurname)
3.         .equal("DiCaprio")
4.         .toQuery();
5.
6. String query4 = actorRepo
7.         .where(Actor::getFullName)
8.         .equal("Leonardo DiCaprio")
9.         .toQuery();
10.-------------------------------
11.query3:
12. FROM
13.  Actor a
14. WHERE
15.  a.surname = 'DiCaprio'
16.query4:
17. FROM
18.  Actor a
19. WHERE
20.  a.surname = 'Leonardo DiCaprio'
```

Trudności również sprawia odniesienie się np. do ilości asocjacji, gdzie do klauzuli `Where` powinna zostać przekazana wartość rozmiaru kolekcji. Natomiast funkcja pobierania atrybutów nie jest w stanie sprawdzić czy programista odnosi się do samej kolekcji czy do jej rozmiaru.

Natomiast warto wskazać na potencjalne źródło istnienia tej funkcjonalności. W opinii autora duże znaczenie powodzenia, bądź też nie, działania algorytmu ma swoje początki w samym języku Java. W odróżnieniu od C#, lambdy nie stanowią obiektów jako takich i nie zawierają metadanych o samej lambdzie. W trakcie kompilacji są one zastępowane różnymi konstrukcjami, które omówiono w rozdziale Wyrażenia lambda (str. 19). W wspomnianym języku Microsoftu, lambdy po kompilacji są faktycznymi obiektami klasy `LambdaExpression`, która to udostępnia użyteczne informacje

o samych obiektach wejściowych. Takie rozwiązanie w języku Java umożliwiłoby poprawną identyfikacje intencji programisty.

7.3. Wnioski

Analizując możliwości języka Java oraz właściwości prototypu wykonanego w ramach niniejszej pracy można sformułować pewne wnioski dotyczące języków zapytani typu Fluent dla Hibernate, czy też ogólnie rozwiązań tego typu występujących w tym języku.

- Język Flux umożliwia uproszczenie procesu produkcyjnego aplikacji w zakresie prostych zapytań, które to w większości projektów stanowią znaczną część komunikacji pomiędzy aplikacją i bazą danych. Biblioteka sprawdza się świetnie w zakresie warunków `Where` oraz `Join`. Czyli dwóch głównych klauzul obecnych w niemal każdym zapytań bazodanowym.
- Implementowanie natywnego języka SQL w języku Flux, w przypadku użycia z Hibernate, nie jest niezbędną funkcjonalnością. Jeśli istnieje potrzeba pisania zapytań, w takim przypadku zapewne najlepszym pomysłem jest bezpośrednie utworzenie tego zapytania w SQL, gdyż złożoność takich pytań i ich jednostkowość jest trudna do przewidzenia. Również tworzenie mnogich rozwiązań, z którego programista korzysta rzadko może nastąpić na bardziej zawansowanym etapie rozwoju biblioteki. Rozwój takich modułów tłumaczących wymagał by również opracowania analizatora asocjacji pomiędzy klasami, bazującego na adnotacjach JPA. Dopiero na ich podstawie da się jednoznacznie stwierdzić w jaki sposób odwzorowano relację w bazie danych.
- Prototyp nie radzi sobie ze skomplikowanymi rekurencyjnymi asocjacjami. W takich przypadkach może dojść do zapętlenia działania programu.
- Moduł pobierający nazwę atrybutu tworzy znaczną ilość obiektów, które są wykorzystywane jednokrotnie. W tym zakresie należałoby wprowadzić mechanizm fabryki, która umożliwiłaby recykling obiektów w ramach zapytań dotyczących tej samej klasy. Takie rozwiązanie mogłoby wspomóc wydajność w systemach spadkowych, które mają w zwyczaju operować mniejszą mocą obliczeniową. W nowoczesnych systemach problem ten nie powinien, w początkowych fazach, powodować spadków wydajności.
- W oparciu o podobne projekty można stwierdzić, że biblioteka przy obecnych założeniach, i obecnym stanie, raczej nie nadaje się do użytku biznesowego. Jedynym rozwiązaniem w opinii autora jest złagodzenie wymagań w zakresie ingerencji w strukturę klasy, która to musi zawierać jakieś metadane, do których to można wskazywać w czasie tworzenia zapytań. Proces ten można wspomóc ewentualnie autorskim generatorem na wzór JOOQ. W takim przypadku warto byłoby założyć maksymalną integracje wygenerowanego kodu z encjami używanymi przez JPA. Końcowym efektem byłaby poprawna interpretacja intencji programisty.

Bibliografia

[1] E. F. Codd, The Relational Model for Database Management, London: Addison-Wesley, 1990.

[2] M. Chatham, Structured Query Language By Example - Volume I: Data Query Language, 2012.

[3] T. S. D. Barry, Solving the Java object storage problem, IEEE, 1998 .

[4] G. K. Christian Bauer, Hibernate in Action, Greenwich: Manning Publications Co., 2005.

[5] Microsoft, „Dokumentacja C#," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.linq.expressions.lambdaexpression.body?view=netcore-3.1. [Data uzyskania dostępu: 23 Sierpień 2020].

[6] „Dokumentacja biblioteki JOOQ," [Online]. Available: https://www.jooq.org/doc/3.14/manual-single-page/#tutorials. [Data uzyskania dostępu: 2 6 2021].

[7] R. U. Richard Warburton, „Java 8 Lambdas - A Peek Under the Hood," 7 Październik 2014. [Online]. Available: https://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood/. [Data uzyskania dostępu: 23 Sierpień 2020].

[8] Oracle, „Dokumentacja dystrybucji," Oracle, [Online]. Available: https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html. [Data uzyskania dostępu: 23 Sierpień 2020].

[9] J. Jurinová, „ResearchGate," 2018. [Online]. Available: https://www.researchgate.net/publication/325900494_Performance_improvement_of_using_lambda_expressions_with_new_features_of_Java_8_vs_other_possible_variants_of_iterating_over_ArrayList_in_Java. [Data uzyskania dostępu: 23 Sierpień 2020].

[10] S. Kuksenko, „JDK8:Lambda Performance Study," Oracle, 2013. [Online]. Available: https://www.oracle.com/technetwork/java/jvmls2013kuksen-2014088.pdf. [Data uzyskania dostępu: 23 Sierpień 2020].

[11] G. a. M. M. Cousineau, The Functional Approach to Programming by Guy Cousineau and Michel Mauny, Cambridge : Cambridge University Press, 1998.

[12] G. J. S. J. S. Harold Abelson, Structure and Interpretation of Computer Programs, Cambridge: The MIT Press, 1999.

[13] T. Neward, „Java 8: Lambdas, Part 2," Oracle, Wrzesień/Październik 2013. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/architect-lambdas-part2.html. [Data uzyskania dostępu: 23 Sierpień 2020].

[14] I. G. A. W. HaiTao Mei, Real-Time Stream Processing in Java, Ada-Europe: Reliable Software Technologies, 2016.

[15] M. Fernandez, Models of Computation: An Introduction to Computability Theory, Springer Science & Business Media, 2009, 2009.

[16] P. Bucek, „Mail OpenJDK," 13 Marzec 2014. [Online]. Available: http://mail.openjdk.java.net/pipermail/jdk8-dev/2014-March/004050.html. [Data uzyskania dostępu: 23 Sierpień 2020].

[17] T. Neward, „Java 8: Lambdas, Part 1," Oracle, Czerwiec/Lipiec 2013. [Online]. Available: https://www.oracle.com/technical-resources/articles/java/architect-lambdas-part1.html. [Data uzyskania dostępu: 23 Sierpień 2020].

Dodatki

Dodatek A: Spis rysunków

Rys. 1. Schemat połączenia aplikacji i bazy danych 10
Rys. 2. Schemat połączeń pomiędzy głównymi klasami 26
Rys. 3. Pętla sprawdzenia atrybutów 27
Rys. 4. Weryfikacja pobranego atrybutu 28
Rys. 5. Klasy służące do kontroli przepływu podpowiedzi podczas pisania zapytania 29
Rys. 6. Podpowiedzi środowiska programistycznego w czasie pisania zapytania 30
Rys. 7. Schemat testowego programu 33

Dodatek B: Spis tabel

Tabela 1. Niektóre interfejsy funkcyjne 20

Dodatek C: Spis listingów

Listing 1. Schemat składni języka DQL 9
Listing 2. Przykładowe zapytanie w języku SQL 9
Listing 3. Przykładowe zapytanie w HQL, semantycznie identyczne z Listing 2 11
Listing 4. Przykładowe zapytanie w języku LINQ 11
Listing 5. Zapytanie w języku LINQ w składni podobnej do języka SQL 12
Listing 6. Użycie JOOQ 13
Listing 7. Kod wygenerowany do poprawnego działania biblioteki JOOQ 14
Listing 8. Przykład klasy abstrakcyjnej, klasy oraz interfejsu 15
Listing 9. Program tworzący klasy anonimowe 16
Listing 10. Wydruk działanie programu prezentującego różnice pomiędzy klasami i interfejsami. 16
Listing 11. Klasa anonimowa po kompilacji [7] 17
Listing 12. Składnia wyrażenia lambda 17
Listing 13. Wyrażenia lambda po kompilacji [7] 18
Listing 14. Java Generics deklaracja klasy 19
Listing 15. Java Generics zastosowanie 19
Listing 16. Przykład interfejsu funkcyjnego 19
Listing 17. Przykład strumieni funkcyjnych w Javie 20
Listing 18. Metody pakietu java.reflection 21
Listing 19. Przykładowa klasa bazowa 24
Listing 20. Utworzenie głównego obiektu biblioteki 25
Listing 21. Użycie języka Flux 25
Listing 22. Użycie architektury powiązanej z Hibernate 25
Listing 23. Domyślna implementacja klasy toList() 26
Listing 24. Interfejs modułu tłumaczącego 31
Listing 25. Wpis w pom.xml 33
Listing 26. Populacja danych 33
Listing 27. Utworzenie repozytoriów dostępowych dla każdej z klas 34
Listing 28. Pobranie ekstensji filmów 34
Listing 29. Wygenerowane zapytanie w języku HQL 34
Listing 30. Wynik działania zapytania 34
Listing 31. Prezentacja kompleksowego warunku zapytania 35

Listing 32. Wydruk działania kodu z Listing 31 35
Listing 33. Zapytanie z automatycznie generowanym warunkiem JOIN w języku Flux....35
Listing 34. Wydruk działania kodu w postaci zapytania JPQL, z Listing 33 35
Listing 35. Klauzula GROUP BY 35
Listing 36. Wydruk działania kodu w postaci zapytania JPQL, z Listing 35 36
Listing 37. Test poprawności zapytań....37
Listing 38. Wynik testu poprawności zapytań 38
Listing 39. Przypadek błędnego pobrania atrybutu....38