Native Queries

Tomasz Wiśniewski

26 sierpnia 2010

Spis treści

Rozdział 1

Wstęp

Wstęp TODO.

Rozdział 2

Trwałość obiektów - Stan sztuki

Aby dobrze zrozumieć treść niniejszej rozprawy, konieczna jest znajomość kilku podstawowych pojęć związanych z obiektowością, bazami danych oraz trwałością. Rozdział ten ma na celu zapoznanie czytelnika z tymi zagadnieniami.

2.1 Obiektowość

2.1.1 Obiekt

Obiektem nazywamy abstrakcyjny byt opisujący pewną rzecz lub pojęcie w świecie rzeczywistym. Obiekt jest odróżnialny od innych obiektów, ma on nazwę i dobrze określone granice. W informatyce obiektem jest pewna struktura danych przechowywana w przestrzeni pamięciowej komputera. Nie jest wymagane, aby ta struktura danych miała swój odpowiednik w świeci rzeczywistym. Obiektem może być dowolna abstrakcja programistyczna, np. moduł, procedura, zmienna itd. Istotą obiektu jest to, że programista może nim manipulować jak pojedynczą zwartą bryłą, np. tworzyć, kopiować, usuwać, przenosić. [?]

2.1.2 Programowanie obiektowe

Programowaniem obiektowe (ang. object oriented programming - OOP) nazywamy paradygmat programowania, w którym programy definiuje się przy pomocy obiektów.

Podstawowe pojęcia używanie przez języki obiektowe.

- Klasa

  Klasą w programowaniu obiektowym nazywamy definicję zbioru obiektów posiadających wspólne własności.

- Instancja

  Instancja to konkretny obiekt w pamięci operacyjnej będący członkiem pewnej, określonej klasy.

- Metoda

  Metodami są nazwane podprogramy przypisane do klasy obiektów (mówimy wtedy o metodach statycznych) lub do samego obiektu. Metody, tak jak w procedury w programowaniu proceduralnym, składają się z szeregu wykonywalnych instrukcji, przekazywanych parametrów oraz zwracanej wartości. Jednak w przeciwieństwie do nich, metody działają tylko w środowisku obiektu.

- Przesyłanie komunikatów

  Przesyłanie (ang. message passing) komunikatów jest sposobem komunikacji między obiektami. Aby przesłać komunikat (czyli wykonać metodę) należy najpierw podać identyfikator obiektu będącego odbiorcą, a następnie podać komunikat, czyli nazwę metody, która ma być wykonana. Składnia przesyłania komunikatów różni się w zależności od używanego języka programowania, ale najczęściej posiada postać $<obiekt>.<komunikat(\ parametry)>$.

- Dziedziczenie

  Dziedziczenie to sposób na tworzenie nowej klasy na podstawie klasy już istniejącej, zwanej nadklasą. Nowo powstała klasa, zwana podklasą, dziedziczy definicje atrybutów oraz metod z klasy nadrzędnej. Dziedziczenie jest jednym z podstawowych mechanizmów ponownego użycia stworzonego kodu.

- Abstrakcja

  Abstrakcją nazywamy upraszczanie rozpatrywanego problemu. Osiąga się to poprzez ukrywanie cech manipulowanych obiektów niezwiązanych bezpośrednio z algorytmem.

- Hermetyzacja

  Hermetyzacja (ang. encapsulation) polega na ograniczaniu dostępu do pewnych atrybutów i metod obiektu danej klasy, tak aby były dostępne tylko dla innych metod tej klasy.

- Polimorfizm

  W programowaniu obiektowym polimorfizmem nazywamy możliwość posiadania przez klasę wielu metod o tej samej nazwie, z możliwością dynamicznego wyboru konkretnej metody podczas czasu wykonania.

2.1.3 Java

Java jest obiektowym językiem programowania stworzonym przez firmę Sun Microsystems, czerpiącym inspirację z C++. Zrezygnowano w niej jednak z niskopoziomowego, ręcznego zarządzania pamięcią i wprowadzono znany z języka Smalltalk mechanizm automatycznego zbierania nieużytków (ang. garbage collection). Programy pisane w języku Java nie są kompilowane do kodu maszynowego, lecz do pośredniego bajtkodu, uruchamianego na wirtualnej maszynie javy (JVM).

2.2 Bazy danych

Bazą danych nazywamy zbiór danych zapisanych w strukturach odpowiadających założonemu modelowi danych.

2.2.1 Relacyjne bazy danych

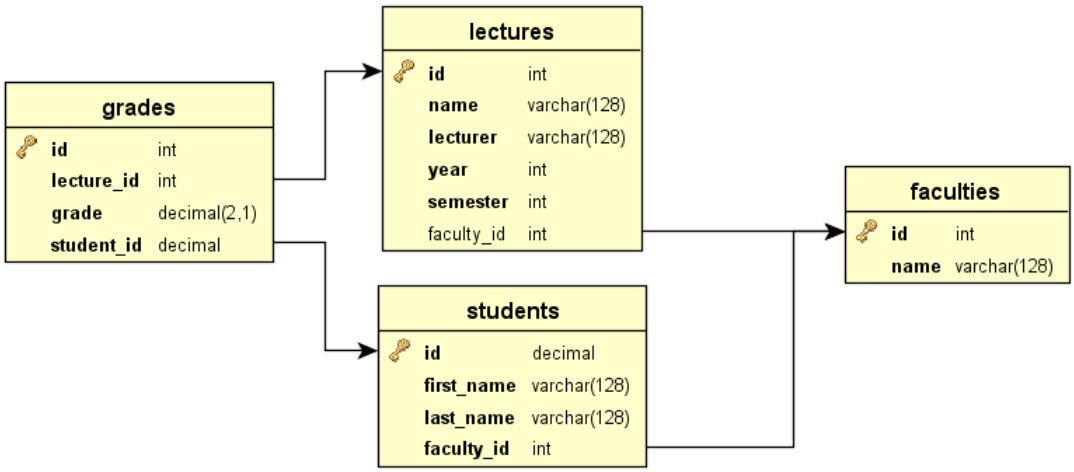
Relacyjnymi bazami danych nazywamy bazy danych zorganizowane według relacyjnego modelu danych stworzonego przez E. Codd'a w 1970. Model ten zakłada przechowywania danych w dwu wymiarowych tabelach (postrzeganych jako "relacje" z matematycznego punktu widzenia). Dostęp do danych definiowany jest przy pomocy algebry relacji, realizowanej zwykle przez język zapytań SQL, lub jego odmiany.

Poniższy uproszczony schemat przedstawia relacyjną bazę danych zawierającą informacje na temat studentów i ich ocen.

2.2.2 SQL

SQL czyli Structured Query Language to język zapytań, stworzony przez firmę IBM, na potrzeby komunikacji z relacyjnymi bazami danych. Umożliwia on pobieranie, tworzenie oraz modyfikowanie danych znajdujących się w bazach.

[Schemat]



Rysunek 2.1: Schemat uproszczonej bazy studentów

Przykładowe zapytanie SQL operujące na bazie przedstawionej na rysunku 2.1, zwracające wszystkich studentów Wydziału Matematyki i Informatyki, których średnia ocen jest większa od 4.0 wygląd następująco:

Listing 2.1: Zapytanie SQL

```
mysql> SELECT s.id AS indeks,
    ->        s.first_name AS imie,
    ->        s.last_name AS nazwisko,
    ->        AVG(g.grade) AS srednia
    -> FROM students s, faculties f, grades g
    -> WHERE f.name = 'WMiI'
    ->       AND f.id = s.faculty_id
    ->       AND s.id = g.student_id
    -> GROUP BY s.id
    -> HAVING srednia > 4.0;
+--------+---------+---------+------------+
| indeks | srednia | imie    | nazwisko   |
+--------+---------+---------+------------+
| 123456 | 4.50000 | Tomasz  | Wisniewski |
| 123457 | 4.75000 | Natalia | Wozniak    |
+--------+---------+---------+------------+
2 rows in set (0.00 sec)
```

2.2.3 Systemy zarządzania bazami danych

Systemem zarządzania bazami danych (SZBD) to systemy informatyczne umożliwiające pracę z bazami danych, czyli zapisywanie, modyfikowanie i odczytywania informacji zawartych w bazie. Istnieje wiele odmian SZBD, od małych systemów wbudowanych, do systemów działających w architekturze klient-serwer. SZBD posiadają zazwyczaj takie mechanizmy jak:

- Środki do gromadzenia, utrzymywania i administrowania trwałymi i masowymi zbiorami danych
- Środki zapewniające spójność i bezpieczeństwo danych
- Język zapytań umożliwiający sprawny dostęp do danych
- Środki programistyczne do przetwarzania danych
- Mechanizm pozwalający na równoczesny dostęp do bazy wielu użytkownikom
- Środki autoryzacji regulujące dostęp do bazy
- Środki tworzenia kopii zapasowych, umożliwiających odtworzenie zawartości bazy po awarii.
- Środki do zarządzania katalogami, schematami i innymi metadanymi
- Środki optymalizujące zajętość pamięci i czas dostępu (np. indeksy)

Przykładami SZBD są MySQL, PostgreSQL, SQLite, DB4o, Oracle.

2.3 Trwałość

Według definicji podanej w [?] byt programistyczny można nazwać trwałym(ang. persistent), jeżeli czas istnienia obiektu jest dłuższy, niż czas działania programu, który utworzył ten obiekt. Oznacza to, że trwałe byty są zapisywane i odczytywane z pamięci permanentnej. Popularne języki programowania(C++, Java, C#,...) nie posiadają obiektów trwałych, a jedynie ulotne, znajdujące się w pamięci RAM. Wszystko co zawierają bazy danych jest trwałe. Trwały obiekt cechuje się tym, że posiada własności zwykłego obiektu, ale jego wartość jest taka sama jak przy zakończeniu poprzednio uruchomionego programu.

Rodzaje trwałości:

- Trwałość poprzez dziedziczenie

  Trwałość poprzez dziedziczenie (ang. transparence through inheritance) zakłada, że trwałość jest cechą definiowanej klasy oraz wszystkich jej podklas. Oznacza to, że żadna klasa nie może posiadać zarówno obiektów trwałych i nie trwałych. Zaletą tej koncepcji jest wyraźny podział funkcjonalności systemu na część zajmującą się bazą danych i część zajmującą się nietrwałymi obiektami.

- Trwałość poprzez osiągalność

  Trwałość poprzez osiągalność (ang. transparence through reachability) zakłada, że obiekt trwały nie może mieć nietrwałych podobiektów lub atrybutów, ani też nie może zawierać wskaźnika do obiektu nietrwałego. Wszystkie obiekty trwałe są osiągalne z jednego "trwałego korzenia. Koncept ten pozwala na zachowanie spójności danych w przypadku awarii programu.

- Trwałość ortogonalna

  Ortogonalna trwałość (orthogonal persistence) to koncepcja zgodnie, z którą cecha trwałości obiektu jest całkowicie ortogonalna w stosunku do jego typu. Pozwala to na ujednolicenie środków dostępu i manipulacji trwałymi i ulotnymi obiektami. Przykładem języka oferującego ortogonalną trwałość jest dialekt języka Java - Pjama [1].

2.3.1 Trwałość w Javie

Pierwsza wersja Javy nie zezwalała na utrwalanie obiektów. Inne dane (np. tekstowe, graficzne, itp.) można było zapisywać/odczytywać z dysku przy pomocy strumieni wejścia/wyjścia. Dopiero Java Development Kit w wersji 1.1 poszło krok naprzód i wprowadziło mechanizmy zezwalające na utrwalanie obiektów: serializację oraz JDBC.

Serializacja

Serializacją obiektów nazywamy proces polegający na konwersji aktualnego stanu obiektu do strumienia bajtów. Dzięki temu obiekt może zostać zachowany w pamięci trwałej lub np. przesłany przez sieć. Procesem odwrotnym, tj. odtworzenie obiektu ze strumieni bajtów jest deserializacja.

[1] http://labs.oracle.com/forest/opj.main.html

JDBC

JDBC, powszechnie rozwijane jako Java Database Connectivity, to API języka Javy umożliwiające porozumiewanie się z bazami danych przy użyciu języka zapytań SQL.

Schemat komunikacji wygląda w następujący sposób:

- Ustanowienie połączenia z bazą danych.
- Wysłanie zapytanie SQL.
- Obsłużenie otrzymanych wyników.

Oto przykładowy kod obrazujący komunikację programu napisanego w Javie z bazą danych przedstawionej na rysunku 2.1:

Listing 2.2: Przykładowy kod JDBC

```
public List<Student> getStudents(Faculty faculty, int year)
        throws SQLException, ClassNotFoundException {

            Class.forName("com.mysql.jdbc.Driver");
            // Connection estabilishment
            Connection con = DriverManager.getConnection (
                        "jdbc:mysql://localhost:3306/sandbox",
                        "login", "password");
            // Query preparation
            PreparedStatement stmt = con.prepareStatement
                ("SELECT * FROM students s" +
                 "WHERE f.name = ? AND f.id = s.faculty_id" +
                 "AND s.year = ?");
            stmt.setString(1, faculty.getName());
            stmt.setInt(2, year);
            // Query execution
            ResultSet rs = stmt.executeQuery();
            // Results handling
            List<Student> result = new ArrayList<Student>();
            while (rs.next()) {
                Student student = new Student();
                student.setIndex(rs.getInt("id"));
                student.setFaculty(faculty);
                student.setFirstName(rs.getString("first_name"));
                student.setLastName(rs.getString("last_name"));
```

```
            result.add(student);
        }
        return result;
    }
```

Używanie JDBC, mimo że skuteczne, pociągało za sobą wiele wad wynikających z używania zapytań SQL z poziomu Javy.

2.4 Niezgodność impedancji

Termin "niezgodność impedancji" wywodzi się z elektroniki i oznacza niedopasowanie między impedancją źródła sygnału a odbiornikiem, co skutkuje stratą mocy. W informatyce nazywa się tak zbiór niezgodności powstałych w wyniku formalnego połączenia języka zapytań (w szczególności SQL) z uniwersalnym językiem programowania, takim jak Java. Niezgodności jakie mogą wystąpić w zakresie:

- Składni. Programista musi używać dwóch stylów językowych i przestrzegać reguł dwóch różnych gramatyk.
- Systemu typów. Język zapytań operuje na typach zdefiniowanych w schemacie bazy danych m.in relacjach, natomiast język programowania posiada zwykle odmienny system typów.
- Semantyki i paradygmatów języków. Język zapytań bazuje na style na stylu deklaratywnym podczas gdy języki programowania bazują na stylu imperatywnym.
- Pragmatyki użycia. Języki zapytań uwalnia programistę od wielu szczegółów organizacji i implementacji danych (np. organizacji zbiorów, obecności lub nieobecności indeksów), podczas gdy w języku programowania te szczegóły muszą być oprogramowane.
- Faz i mechanizmów wiązania. Języki zapytań oparte są o późne wiązanie (są interpretowane), podczas gdy języki programowania zakładają wczesne wiązanie (podczas kompilacji i linkowania).
- Przestrzeni nazw i reguł zakresu. Języki zapytań i język programowania posiadają własne przestrzenie nazw, które mogą zawierać identyczne nazwy o różnych znaczeniach. Odwzorowanie pomiędzy przestrzeniami nazw wymaga dodatkowych środków syntaktycznych i semantycznych.

- Traktowanie wartości zerowych. Bazy danych i języki zapytań posiadają wyspecjalizowane środki do przechowywania wartości zerowych. Środki te nie występują w językach programowania.

- Schematów iteracyjnych. W językach zapytań iteracje są wtopione w semantykę operatorów takich jak selekcja, projekcja i złączenie. W językach programowania iteracje muszą być organizowane explicite przy pomocy pętli "for", "while"i innych. Przetwarzanie wyników zapytań przy pomocy języka programowania wymaga specjalnych udogodnień, takich jak kursory i iteratory.

- Traktowanie cechy trwałości danych. Języki zapytań przetwarzają wyłącznie trwałe dane (znajdujące się na twardym dysku), podczas gdy języki programowania przetwarzają wyłącznie dane nietrwałe znajdujące się w pamięci operacyjnej. Połączenie obu języków wymaga od programisty użycia specjalnych środków językowych do parametryzacji zapytań oraz środków do transmisji danych z dysku do pamięci operacyjnej i na odwrót.

Skutkiem niezgodności impedancji nie są jedynie wady estetyczne. Niezgodności te powodują konieczność tworzenia dodatkowej warstwy oprogramowania pośredniczącego pomiędzy językiem zapytań i językiem programowania. Warstwa ta obciąża dodatkowo kod aplikacji, może być źródłem błędów oraz zwiększa czas wykonania.

Jednym ze sposobów na złagodzenie niezgodności impedancji jest stosowanie odwzorowania obiektowo relacyjnego.

2.5 Odwzorowanie Obiektowo Relacyjne

Odwzorowanie Obiektowo Relacyjne (ang. Object-Relational Mapping, ORM) to technika automatycznej konwersji danych pomiędzy obiektowymi językami programowania, a relacyjnymi bazami danych. Głównym celem narzędzi ORM jest uwolnienie programisty od bezpośredniej komunikacji z bazą danych i pisania nisko poziomowych zapytań SQL. Inne udogodnienia jakie oferują narzędzia ORM to:

- Deklaratywne odwzorowanie pomiędzy obiektowym a relacyjnym modelem danych. Opisuje w jaki sposób klasy oraz ich atrybuty mają być odwzorowane do tabel baz danych.

- Języki zapytań, służące do przeszukiwania utrwalonych obiektów. Konwertowane są do zapytań SQL wywoływanych na bazie danych.

- API do tworzenia, wczytywania, uaktualniania, oraz usuwania utrwalanych obiektów.
- Wsparcie dla transakcji.
- Wsparcie dla leniwego oraz zachłannego ładowania - możliwość wyboru przez użytkownika sposobu w jaki ORM będzie obsługiwał zależności między utrwalonymi obiektami.
- Cache'owanie wyników zapytań do bazy w celu zwiększenia wydajności.

Narzędzia ORM w Javie tworzone są na bazie JDBC, ale cały kod jest generowany automatycznie, dzięki czemu programista może skupić się na innych aspektach aplikacji.

Lista ORMów na platformę Java jest dość długa, jednak najważniejszymi z nich są bez wątpienia Hibernate firmy JBoss, oraz Java Persistence API - stworzony w oparciu o najlepsze cechy istniejących rozwiązań.

2.6 Hibernate

Hibernate jest jednym z najpopularniejszych narzędzi ORM typu open-source. Prace nad nim rozpoczęto w 2001 roku. Zespół programistów pod kierownictwem Gavin'a King'a wyznaczył sobie za cel stworzenie alternatywy dla komponentów encyjnych[2] wchodzących w skład Enterprise Java Beans.

Swą popularność zdobył głównie dzięki prostej konfiguracji odwzorowania przy użyciu plików xml oraz możliwości wykonywania zapytań do bazy przy użycia języka zapytań HQL. Dużym atutem jest również jego wysoka skalowalność, a co za tym idzie, możliwość prostego użycia w małych projektach jak i w tych bardzo skomplikowanych.

Hibernate od czasu stworzenia dorobił się wielu dodatkowych modułów rozszerzających jego bazową funkcjonalność odwzorowania obiektowo relacyjnego (przeniesioną do modułu Hibernate Core).

2.6.1 Konfiguracja

Konfiguracja odwzorowania obiektowo-relacyjnego w frameworke'u Hibernate składa się z dwóch etapów:

[2]Entity Beans

- 1. Zdefiniowanie głównego pliku konfiguracyjnego. W pliku tym konieczne jest zamieszczenie informacji o połączeniu z bazą danych oraz lokalizacje do plików odwzorowań.

  Oto zawartość przykładowego pliku konfiguracyjnego hibernate.cfg.xml.

Listing 2.3: hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="connection.driver_class">
      com.mysql.jdbc.Driver
    </property>
    <property name="connection.url">
      jdbc:mysql://localhost:3306/sandbox
    </property>
    <property name="connection.username">login</property>
    <property name="connection.password">password</property>
    <!-- Mapping files -->
    <mapping resource="Student.hbm.xml"/>
    <mapping resource="Grade.hbm.xml"/>
    <mapping resource="Faculty.hbm.xml"/>
    <mapping resource="Lecture.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

- 2. Zdefiniowanie plików odwzorowań. Pliki odwzorowań to dokumenty XML o standardowym rozszerzeniu hbm.xml. Opisują odwzorowania pomiędzy klasami a tabelami, relacje pomiędzy tabelami czy zachowania kaskadowe.

  Przykładowy plik odwzorowania opisujący odwzorowanie pomiędzy klasą Student a tabelą Students:

Listing 2.4: Przykładowy plik hbm.xml

```
<hibernate-mapping>
  <class name="sandbox.h.entities.Student" table="students">
   <id name="index" type="int" column="id" >
  </id>
  <property name="firstName">
```

```
        <column name="first_name" />
      </property>
      <property name="lastName">
        <column name="last_name"/>
      </property>
      <set name="grades" fetch="join">
            <key column="student_id"></key>
            <one-to-many class="sandbox.h.entities.Grade"/>
      </set>
      <many-to-one name="faculty" class="sandbox.h.entities.Faculty"
    column="faculty_id" fetch="join"></many-to-one>
     </class>
    </hibernate-mapping>
```

2.6.2 Hibernate Query Language

Hibernate Query Language (HQL) jest językiem zapytań dostarczanym z modułem Hibernate Core. Jest on wzorowany na języku SQL, jednak w przeciwieństwie do niego operuje nie na danych przechowywanych w tabelach, lecz na obiektach. Oto przykładowe zastosowanie HQL:

Listing 2.5: Przykładowe zastosowanie HQL

```
public static List<Student> getStudents(Faculty faculty, int year) {
  Configuration cfg = new Configuration();
  SessionFactory factory = cfg.configure().buildSessionFactory();
  Session session = factory.openSession();
  Query q  = session.createQuery(
      "FROM Student s " +
      "WHERE s.faculty.name=:name" +
        "AND s.year=:year");
  q.setString("name", faculty.getName());
  List<Student> result = q.list();
  session.close();
  return result;
}
```

2.7 Java Persistence API

Java Persistence API (JPA) to oficjalny standard odwzorowania obiektowo relacyjnego firmy Sun Microsystems. Pierwsza jego wersja została wydana jako część specyfikacji Enterprise Java Beans 3.0 [3], druga wersja standardu miała już osobne wydanie [4]. Duży wpływ na finalny kształt specyfikacji mieli, między innymi, twórcy Hibernate'a.

Należy pamiętać, że Sun udostępnił tylko specyfikację, a nie produkt umożliwiający utrwalanie obiektów. Aby to osiągnąć musi zostać użyta gotowa implementacja JPA. Do najpopularniejszych implementacji należą: EclipseLink, Hibernate (moduły EntityManager i Annotations) oraz OpenJPA.

Specyfikacja JPA składa się z 3 głównych komponentów

- Adnotacji[5] JPA - metadanych opisujących odwzorowanie obiektowo relacyjne.
- Standardowe API do zarządzania encjami.
- Języka zapytań Java Persistence Query Language (JPQL)

2.7.1 Adnotacje JPA

Specyfikacja JPA wprowadziła nowatorski sposób deklarowania odwzorowania obiektowo relacyjnego za pomocą adnotacji - metadanych zagnieżdżanych w kodzie Java. Dzięki temu programista może opisać odwzorowanie obiektowo relacyjne bezpośrednio w klasie reprezentującej tabelę z bazy danych. Opisane tak klasy nazywamy Encjami (ang. Entities).

Przykładowa Encja reprezentuje tabelę Faculties z rysunku 2.1

Listing 2.6: Przykładowy kod Entity

```
@Entity
@Table(name="faculties")
public class Faculty implements Serializable {
        private static final long serialVersionUID = 1L;
        private String facultyName;
        private int id;
        private List<Student> students;
```

[3] JSR 220 - http://jcp.org/aboutJava/communityprocess/edr/jsr220/index.html
[4] JSR 317 - http://jcp.org/aboutJava/communityprocess/edr/js317/index.html
[5] Odnośnik do rozdziału na temat implementacji Native Queries

```
        public Faculty() {}

        @Column(name="faculty_name")
        public String getFacultyName() {
                return this.facultyName;
        }

        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        public int getId() {
                return this.id;
        }

        //bi-directional many-to-one association to Student
        @OneToMany(mappedBy="faculty")
        public List<Student> getStudents() {
                return this.students;
        }

        public void setFacultyName(String facultyName) {
                this.facultyName = facultyName;
        }
        public void setId(int id) {
                this.id = id;
        }
        public void setStudents(List<Student> students) {
                this.students = students;
        }
}
```

2.7.2 Standardowe API do zarządzania trwałymi klasami

JPA dostarcza standardowego, wygodnego API do wykonywania operacji utworzenia, odczytu, uaktualnienia oraz usuwania[6] danych z pamięci trwałej.

Listing 2.7: JPA - przykład operacji CRUD

[6] CRUD - Create, Read, Update, Delete

```
public Student getStudent(Long id) {
        EntityManagerFactory emFactory =
          Persistence.createEntityManagerFactory("jpaSandbox");
        EntityManager em = emFactory.createEntityManager();
        return em.find(Student.class, id);
}
```

2.7.3 Java Persistence Query Language

Do tworzenia zapytań bardziej skomplikowanych niż proste operacje CRUD, JPA zostało wyposażone w Java Persistence Query Language (JPQL). Jest on wzorowany ja HQL i również operuje na obiektach. Zapytania tworzone w tym języku są uniwersalne i działają niezależnie od używanej relacyjnej bazy danych.

Oto przykład JPQL odpowiadający użyciu JDBC z ...

Listing 2.8: JPA - przykład JPQL

```
public List<Student> getStudentsByGrades(String facultyName,
                double lower, double upper) {
        EntityManagerFactory emFactory = Persistence
                        .createEntityManagerFactory("jpaSandbox");
        EntityManager em = emFactory.createEntityManager();
        em.getTransaction().begin();
        Query q = em.createQuery("Select s from Student s  "
                + "JOIN s.grades g  "
                + "WHERE s.faculty.facultyName = :facultyName  "
                + "GROUP BY s  "
                + "HAVING AVG(g.grade) BETWEEN :lower AND :upper");
        q.setParameter("facultyName", facultyName);
        q.setParameter("lower", lower);
        q.setParameter("upper", upper);
        return q.getResultList();
```

2.7.4 Konfiguracja

Do poprawnego działania JPA potrzebny jest główny plik konfiguracyjny persistence.xml, umieszczony w folderze META-INF . Programiści umieszczają tam w jednostkach trwałości (ang. persistence unit) dane o użytej implementacji JPA, połączeniu z bazą danych, czy listę odwzorowywanych encji. Dużą

zaletą JPA jest możliwość zmiany używanej implementacji bez konieczności modyfikacji wcześniej napisanego kodu.

Oto przykładowa zawartość pliku persistence.xml:

Listing 2.9: JPA - przykład persistence.xml

```
<persistence-unit name="jpaSandbox">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <class>sandbox.jpa.entities.Student</class>
  <class>sandbox.jpa.entities.Faculty</class>
  <class>sandbox.jpa.entities.Grade</class>
  <class>sandbox.jpa.entities.Lecture</class>

  <properties>
    <property name="hibernate.connection.url"
      value="jdbc:mysql://localhost:3306/sandbox" />
    <property name="hibernate.connection.username" value="root" />
    <property name="hibernate.connection.password" value="root" />
    <property name="hibernate.connection.driver_class"
      value="com.mysql.jdbc.Driver" />
    </properties>
</persistence-unit>
```

Rozdział 3

Native Queries - specyfikacja

Dotychczasowe narzędzia, zapewniające trwałość obiektów w Javie, mimo swej skuteczności wciąż posiadają pewne wady. Jedną z nich jest dysonans między obiektowym językiem programowania, jakim jest Java, a zapytaniami do bazy danych wyrażanymi przy pomocy łańcuchów znaków. Weźmy za przykład zapytanie stworzone przy użyciu Hibernate Query Language zwracające studentów 4 roku wydziału Matematyki i Informatyki z bazy przedstawionej na rysunku 2.1.

```
Configuration cfg = new Configuration();
SessionFactory factory = cfg.configure().buildSessionFactory();
Session session = factory.openSession();
Query q  = session.createQuery(
  "FROM Student s
   WHERE s.faculty.name='WMiI' AND s.year = 4");
List<Student> result = q.list();
```

Przedstawione podejście, podobnie jak każde inne oparte na łańcuchach znaków, posiada pewien zestaw wad:

- Zintegrowane środowiska programistyczne nie sprawdzają zapytań wyrażonych przy pomocy łańcuchów znaków zarówno pod względem poprawności semantycznej jak i syntaktycznej. Jeśli programista, na przykład, poprzestawia litery w zapytaniu, lub dla pola year poda nienumeryczną wartość, wykonanie zapytania zaowocuje błędem dopiero w czasie działania programu a nie w trakcie kompilacji.

- Środowiska programistyczne nie refaktorują nazw pól w zapytaniach. Jeśli nazwa takiego pola w klasie ulegnie zmianie, programista musiałbędzie poprawić wszystkie zapytania ręcznie. Jest to bardzo uciążliwe, zwłaszcza przy użyciu zwinnej metodyki wytwarzania oprogramowania [1], gdzie wymagania i rozwiązania stale ewoluują.

- Zapytania działają niezgodnie z regułą hermetyzacji. Zamiast używać publicznych akcesorów, operują bezpośrednio na prywatnych polach klasy.

- Programiści muszą używać dwóch różnych języków: języka programowania oraz języka zapytań.

- Dynamiczne zapytania tworzone przez konkatenację łańcuchów są narażone na ataki typu injection.

3.1 Native Queries - projekt

W 2005 roku William R. Cook oraz Carl Rosberg opublikowali pracę [?], w której zaprezentowali ideę Native Queries, alternatywę dla zapytań wyrażanych za pomocą łańcuchów znaków. Zakładała ona konstruowanie zapytań wyłącznie poprzez użycie obiektowego języka programowania. Zapytania stworzone w ten sposób uwalniają programistę od używania osobnego języka zapytań oraz w pełni wykorzystują statyczną kontrolę typów. Użycie Native Queries najprościej zademonstrować na przykładzie. Zapytanie zwracające studentów z 4 roku Wydziału Matematyki i Informatyki wyglądałby w następujący sposób:

```
(student.getYear() == 4) &&
(student.getFaculty().getName().equals("WMiIs"))
```

Mogłoby się wydawać, że takie podejście nie nadaje się na mechanizm wykonywania zapytań gdyż wiąże się z olbrzymim wzrostem obciążenia, wszystkie rekordy w tabeli musiałyby zostać odwzorowane do obiektów i dopiero

[1] Agile software development - http://en.wikipedia.org/wiki/Agile_software_development

na nich wykonane zostałoby filtrowanie. Dlatego zapytania tworzone w ten sposób powinny być optymalizowane w czasie kompilacji, bądź wykonania programu. Optymalizacja polegałaby na analizie kodu Native Query i przetłumaczeniu go na zapytania używające jednego z dotychczasowych rozwiązań (np. SQL, JPQL, itd.).

Reasumując , Native Queries muszą spełniać następujące założenia:

- Natywność

  Zapytania muszą być tworzone wyłącznie przy użyciu natywnego języka programowania.

- Obiektowość

  Programista powinien mieć możliwość wykonania niezoptymalizowanego zapytania na kolekcji obiektów.

- Bezpieczeństwo typologicznie

  Zapytania powinny wykorzystywać takie cechy zintegrowanych środowisk programistycznych jak sprawdzanie pisowni, kontrola typów, refaktorowanie.

- Optymalizowalność

  W celu zwiększenia wydajności powinno być możliwe przetłumaczenie zapytań Native Query na język zapytań oparty na łańcuchach znaków.

3.2 Native Queries - interfejs programowania

Aby korzystać z Native Queries konieczne jest zaprojektowanie interfejsu programowania aplikacji (ang. application programming interface). Aby ułatwić jego zrozumienie zostanie to przedstawione krok po kroku.

1. Po pierwsze musimy mieć zapytanie stworzone w języku Java:

```
(student.getYear() == 4) &&
(student.getFaculty().getName().equals("WMiIs"))
```

2. Następnie musimy w jakiś sposób przekazać do zapytania obiekt klasy Student oraz uzyskać rezultat zapytania. Osiągamy to poprzez metodę z parametrem Student, zwracającą typ bool.

```
public boolean match(Student student) {
  return (student.getYear() == 4) &&
    (student.getFaculty().getName().equals("WMiIs"))
}
```

3. Niestety w Javie nie jest możliwe przekazywanie metod jako argumenty, dlatego stworzona metoda musi być elementem jakiejś klasy.

```
new Predicate() {
  public boolean match(Student student) {
    return (student.getYear() == 4) &&
      (student.getFaculty().getName().equals("WMiIs"))
  }
}
```

4. Otrzymaliśmy gotowy przykład Native Query dla naszego zapytania. Teraz trzeba go uogólnić dla dowolnych zapytań.

```
public abstract class Predicate <E> {
  public <E> Predicate() {}
  public abstract boolean match (E candidate);
}
```

Poszerzając o typy generyczne nasz przykład finalnie wygląda w następujący sposób:

```
new Predicate<Student>() {
  public boolean match(Student student) {
    return (student.getYear() == 4) &&
      (student.getFaculty().getName().equals("WMiIs"))
  }
}
```

Mając tak zaprojektowane zapytanie Native Query, przedstawimy pełny przykład jego użycia w praktyce.

```
List<Students> results = db.query <Student> (
  new Predicate<Student>() {
    public boolean match(Student student) {
      return (student.getYear() == 4) &&
        (student.getFaculty().getName().equals("WMiIs"))
    }
});
```

Użycie zapytania Native Query może się różnić w zależności od implementacji. Zapytania parametryzowane tworzyłoby się analogicznie, poprzez dodawanie argumentów do metody Predicate#match.

3.3 Native Queries - optymalizacja

Optymalizacja zapytań jest kluczowym konceptem idei Native Queries, bez niej zapytania musiałyby filtrować po wszystkich instancjach klasy mapującej. Dzięki optymalizacji użytkownik jest w stanie pisać zapytania w natywnym języku programowania, a baza wykonywać je z wydajnością odpowiadającej zapytaniom opartym na łańcuchach znaków.

Implementacja takiej optymalizacji nie jest sprawą trywialną. Stworzony kod Native Query musi być poddany analizie i zostać przetłumaczony na odpowiednik w wybranym mechanizmie utrwalania korzystającym z zapytań opartych na łańcuchach znaków. Nie jest konieczne, żeby całe zapytanie Native Query zostało przetłumaczone. Jeśli optymalizator nie jest w stanie przetłumaczyć całego wyrażenie, programista może zażyczyć sobie wykonanie nie do końca przetłumaczonego zapytania, a następnie dokonać filtracji zwróconych wyników przy użyciu natywnego zapytania Native Query.

Sama optymalizacja może odbywać się na wiele sposobów i zależy od implementacji Native Query. W przypadku Javy może się to odbywać m.in w czasie kompilacji, postkompilacji lub w trakcie działania programu.

Optymalizacja zapytań zostanie omówiona dokładnie w rozdziale 4.

3.4 Native Queries - ograniczenia

Teoretycznie zapytanie Native Query może zawierać dowolny kod źródłowy, jednak w praktyce mogłoby to spowodować niestabilne działanie bądź szybkie zużycie zasobów. Dlatego twórcy specyfikacji zalecają przestrzeganie pewnych reguł przy tworzeniu wyrażeń Native Query.

- Dozwolone powinno być tworzenie nowych zmiennych, tymczasowych obiektów czy statyczne odwołania.
- Zapytania powinny wykonywać się jak najszybciej, dlatego nie powinny wchodzić w interakcję z interfejsem użytkownika.
- Zapytania mogą być wykonywane w dużych ilościach, dlatego nie powinny mieć możliwości tworzenia wątków.
- Zapytania nie mogą modyfikować zawartości utrwalonych obiektów.
- Maksymalna długość wykonywania zapytania oraz zużycie pamięci powinno być objęte limitem.
- Ze względu na bezpieczeństwo, wyrażenia Native Query nie powinny mieć dostępu do pewnych interfejsów programowania (np. dostępu do plików na dysku).

Stosowanie się do powyższych reguł ogranicza możliwości programisty, ale jednocześnie chroni przed możliwością wystąpienia potencjalnych, bardzo poważnych błędów, dlatego też przestrzeganie tych zasad jest niezmiernie ważne.

Rozdział 4

Native Queries - implementacja prototypu

Mimo wielu zalet używania Native Queries, nie powstała jak dotąd żadna implementacja tej idei w komunikacji między językiem programowania Java a relacyjnymi bazami danych. Jedyna istniejąca realiazacja Native Queries, została opracowana przez samego jej twórcę, jednakże realizuje ona komunkację kodu Javy jedynie z obiektową bazą danych db4o. Ta baza nie cieszy się jednak największą popularnością wśród twórców apliakcji komputerowych. Dlatego też celem autora niniejszej pracy stało się zbadanie możliwości realizacji idei Native Queries dla relacyjnych baz danych oraz próba implemtacji prototypu.

4.1 Problem wyboru narzędzia do rozszerzenia - zmienic lepiej

Realizacja zadania wymagała od autora dokonania kilku ważnych wyborów w sposobie impelemtacji. Pierwszy z nich, niezwykle istotny, to wybór narzędzia utrwalania danych, dla którego idea Native Queries miała zostać wdrożona. Z uwagi na to, że ważnym celem autora było stworzenie rozwiązania dla możliwie licznej grupy programistów, pod uwagę były brane tylko najpopularniejsze rozwiązania, a więc przede wszystkim JDBC, Hibernate oraz JPA. Kluczową sprawą była też łatwość realizacji tłumaczenia Native

Queries na odpowiedni kod. W tej kwestii, z tych trzech narzędzi, JPA wydało się nejlpeszym rozwiązaniem, przede wszystkim z uwagi na prostotę odwzorowań obiektowo-relacyjnych, realizowanych przez adnotacje, których odczytanie jest znacznie prostrze niż czytanie danych na przykład z plików XML, jak w Hibernate. JDBC nie ma mechanizmu odwzorowań wcale, co stanowiłoby dodatkowy problem do rozwiązania. Ponadto standard JPA jest uniwersalny i posiada wiele implemetacji, w tym również implemetację dla Hibernate.

4.2 Problem wyboru sposobu implmetacji

Mając wybrane narzędzie do utrwalania danych, konieczne było znalezienie najlepszego sposobu na implemetację całego zadania, a zatem przede wszytskim na optymalizację zapytania Native Queries na kod w JPA. Badanie możliwości realizacji tego zadania okazało się kluczowym i najtrudniejszym problemem w etapie tworzenia niniejszej pracy. Pierwszą badaną metodą była edycja bajtkodu.

4.2.1 Edycja bytecodu

Bajtkod, czyli kod do którego kompilowany jest program Javy, ca fe ba be 00 00 00 32 00 1d 0a 00 06 00 0f 09 00 10 00 11 08 00 12 0a 00 13 00 14 07 00 15 07 00 16 01 00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62 65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 72 63 65 46 69 6c 65 01 00 0a 48 65 6c 6c 6f 2e 6a 61 76 61 0c 00 07 00 08 07 00 17 0c 00 18 00 19 01 00 0d 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21 07 00 1a 0c 00 1b 00 1c 01 00 05 48 65 6c 6c 6f 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00 03 6f 75 74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01 00 13 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01 00 07 70 72 69 6e 74 6c 6e 01 00 15 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 00 21 00 05 00 06 00 00 00 00 00 02 00 01 00 07 00 08 00 01 00 09 00 00 00 1d 00 01 00 01 00 00 00 05 2a b7 00 01 b1 00 00 00 01 00 0a 00 00 00 06 00 01 00 00 00 01 00 09 00 0b 00 0c 00 01 00 09 00 00 00 25 00 02 00 01 00 00 00 09 b2 00 02 12 03 b6 00 04 b1 00 00 00 01 00 0a 00 00 00 0a 00 02 00 00 00 04 00 08 00 05 00 01 00 0d 00 00 00 02 00 0e

```
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:Ljava/io/Pri
   3:   ldc     #3; //String Hello, World!
   5:   invokevirtual   #4; //Method java/io/PrintStream.println:(Lja
   8:   return

}
```

Listings

Bibliografia

Książki

[1] *Słownik terminów z zakresu obiektowości*, K. Subieta, Akademicka Oficyna Wydawnicza PLJ, Warszawa 1999

[2] *Teoria i konstrukcja obiektowych języków zapytań*, K. Subieta, Wydawnictwo PJWSTK, Warszawa 2004

[3] *Thinking in Java - 4th ed.*, B. Eckel, 2006

[4] *JDBC^{TM} Tutorial and Reference*, M. Fisher, J. Ellis, J. Bruce, Addison Wesley 2003

[5] *Native Queries for Persistent Objects - A Design White Paper* William R. Cook, Carl Rosenberger