Projekt BD

ORM

Bean Car

```
public class Car {
   int idc;
   String make;
   String model;
   String regNum;
   Double price;
   public int getIdc() {
       return idc;
   }
   //dalsze gettery i settery
}
```

Interfejs DAO

- Interfejs CarDAO
- Metody (CRUD):
  - void add(Car car)
  - void delete(int numer)
  - Car get(int numer)
  - void update(Car car);
  - List<Car> getCars()
- Implementacja: CarDAOImpl

Metoda add

```
public void add(Car car) {
   try{
       con.createStatement().executeUpdate(
              "insert into car values("+
              car.getIdc()+","+
              "'"+car.getMake()+"',"+
              "'"+car.getModel()+"',"+
              "'"+car.getRegNum()+"',"+
              car.getPrice()+")"
       );
   }catch(SQLException ex) {ex.printStackTrace();}
}
```

Metoda getCars

```
public List<Car> getCars() {
List<Car> carList = new ArrayList<Car>();
try{
ResultSet rs = con.createStatement().executeQuery(
   "select * from car");
while(rs.next()) {
   Car car = new Car();
   car.setIdc(rs.getInt("idc"));
   car.setMake(rs.getString("make"));
   car.setModel(rs.getString("model"));
   car.setRegNum(rs.getString("regnum"));
   car.setPrice(rs.getDouble("price"));
   carList.add(car);
}
}catch(SQLException ex) {ex.printStackTrace();}
return carList;
}
```

Problemy

- Konieczność implementacji każdej metody
- Gdy dużo kolumn w tabeli – długie implementacje
- Łatwo zapomnieć o szczegółach – przecinki, nawiasy w zapytaniach SQL
- Problemy z dodawaniem, usuwaniem pól – zmianą schematu bazy
- Główny problem: konieczność ręcznego mapowania obiektów na wiersze w tabeli!

Mapowanie obiektowe

- ORM: Object-to-Relational Mapping
- Idea: jednolitość interfejsu obiektowego
- Użycie obiektów zamiast ResultSet'ów
- Jak najmniej SQLa jak najwięcej programowania obiektowego
- Istnieją biblioteki/frameworki upraszczające tworzenie ORM:
  - Hibernate
  - TopLink
  - iBatis
  - Java Persistence (JPA)

Biblioteka Hibernate

- Biblioteka dla Javy zapewniająca ORM
- Dostęp do danych **tylko** przez specjalne klasy
- Nie wymaga znajomości SQL – w ogóle go nie używamy (choć możemy użyć JPQL)
- Konfiguracja w pliku XML
  - hibernate.cfg.xml
- Każde mapowanie opisane jako adnotacje w plikach transferowych
  - Car.java

Konfiguracja projektu

- Dodanie bibliotek (katalog lib)
  - antlr-2.7.6.jar
  - asm-attrs.jar
  - asm.jar
  - c3p0-0.9.1.jar
  - cglib-2.1.3.jar
  - commons-collections-2.1.1.jar
  - commons-logging-1.0.4.jar
  - dom4j-1.6.1.jar
  - hibernate3.jar
  - jta.jar

Plik konfiguracyjny

- Domyślnie: hibernate.cfg.xml
- Definicja źródła danych
- Sterownik
- Adres serwera
- Użytkownik i hasło
- Dialekt

Konfiguracja hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
   <property name="hibernate.connection.driver_class">
        org.gjt.mm.mysql.Driver
   </property>
   <property name="hibernate.connection.url">
        jdbc:mysql://localhost/nowa
   </property>
   <property name="hibernate.connection.username">root</property>
   <property name="hibernate.connection.password"></property>
   <property name="hibernate.dialect">
        org.hibernate.dialect.MySQLInnoDBDialect
   </property>
   <!-- Klasy do mapowania -->
   <mapping class="pl.swsim.projbd.Car"/>
</session-factory>
</hibernate-configuration>
```

Adnotacje w obiektach transferowych

- Adnotacja:
  - Meta dana, informacja dla innych elementów aplikacji jak należy ten kod potraktować
- W przypadku obiektu transferowego/encji:
  - Dodanie adnotacji @Entity przed class
  - Dodanie adnotacji @Id przed gettem dla klucza głównego (getIdc())
  - Pozostałe pola domyślnie są traktowane jak kolumny w bazie
- Hibernate ładując klasę analizuje adnotacje i orientuje się co jest kluczem

Encja Car

```
@Entity
public class Car {

   int idc;

   @Id
   public int getIdc() {
       return idc;
   }
…
```

Inicjalizacja Hibernate

- hibernate.cfg.xml
  - definicja źródła danych
  - definicja mapowań
- Inicjalizacja Hibernate:

```
Configuration cfg = new
    Configuration().configure([PLIK]);
SessionFactory factory = cfg.buildSessionFactory();
```

- Obiekt *factory* posłuży do otwierania połączeń z bazą

Tworzenie fabryki sesji

- Nowy singleton przechowujący fabrykę sesji

```
public class HibFactory {
   private static SessionFactory factory;
   public static SessionFactory getInstance() {
     if(factory==null) {
       Configuration cfg = new
                AnnotationConfiguration().configure();
       factory = cfg.buildSessionFactory();
     }
     return factory;
   }
}
```

Zmiany w metodzie add

- Nowa metoda add:

```
public void add(Car car) {
   Session session =
       HibFactory.getInstance().openSession();
   Transaction tx = session.beginTransaction();
   session.saveOrUpdate(car);
   tx.commit();
   session.close();
}
```

- Za mapowania odpowiada już Hibernate!

Metoda getCars()

- Zamiast instrukcji select - zapytanie HQL zwracające gotowe obiekty

```
public List<Car> getCars() {
   Session session =
       HibFactory.getInstance().openSession();
   Transaction tx = session.beginTransaction();
   List<Car> carList =
       session.createQuery("from Car").list();
   tx.commit();
   session.close();
   return carList;
}
```

Najważniejsze metody sesji

- save(obj), persist(obj)
- saveOrUpdate(obj)
- delete(obj)
- get(Klasa.class,id)
- createQuery("from NazwaKlasy")
  - query.list()
- createCriteria("NazwaKlasy")
  - criteria.add(Example.create(obj))
  - criteria.list()

Zadanie

- Zaimplementować DAO dla dowolnej własnej tablicy (CRUD)

Entity Bean

- Adnotacja @Entity z pakietu javax.persistence
- Automatyczna synchronizacja ze źródłem danych (bazą danych)
- Zawartość: właściwości, getter i setter dla każdej
- Każda właściwość może miać adnotację
- Jedna z nich MUSI mieć annotation @Id

Przykładowe adnotacje

- @Entity
  - obowiązkowo na początku
- @Table(name="MyTableName")
  - ustawienie innej nazwy dla tabeli
- @Column(name="ColumnName", nullable=false)
  - niekonieczne
  - parametry: nullable, unique, updatable, length, presision...
- @Id
  - definicja klucza głównego (primary key)
- @GeneratedValue
  - dla PK - automatyczna generacja nowych wartości

Typy kolumn

- @Basic
  - domyślny
- @Transient
  - nie zapisywane w bazie
- @Temporal(TemporalType.TIME|DATE|TIMESTAMP)
  - date/time ze sposobem mapowania
- @Lob(fetch=FetchType.EAGER|LAZY)
  - EAGER – zawsze odczytywane wraz z encją
  - LAZY – odczytywane dopiero przy użyciu
- @Enumerated(EnumType.STRING|ORDINAL)

Użycie encji

- Encje wydobywane są z obiektu Session
- Niektóre metody Session
  - persist(entity)
  - get(Class, id)
  - merge(entity)
  - delete(entity)
  - lock(entity, type)
  - refresh(entity)
  - boolean contains(entity)
  - flush()

Tworzenie asocjacji

- Typy asocjacji:
  - OneToOne
  - OneToMany
  - ManyToOne
  - ManyToMany
- Kierunki
  - unidirectional
  - bidirectional

Asocjacja ManyToOne

- Nowa encja Owner
- W encji Car:

```
Owner owner;
@ManyToOne
public Owner getOwner() {
    return owner;
}
public void setOwner(Owner owner) {
    this.owner = owner;
}
```

Tworzenie obiektów

- Dodanie ownera i powiązania z nim

```
Car car = new Car();
car.setMake("Fiat");
car.setModel("Punto");
Owner owner = new Owner();
owner.setName("Kowalski");
session.persist(owner); // KONIECZNE!
car.setOwner(owner);
session.persist(car);
```

Asocjacja OneToMany

- Obiekt zawiera listę innych obiektów
- Przykład: samochód zawiera elementy wyposażenia (equipment)

```
class Car{
...
Set<Equipment> eqs = new HashSet();
...
@OneToMany
public Set<Equipment> getEqs() {
    return eqs;
}
...
```

Analiza

- Kod tworzący samochód z wyposażeniem

```
Car car = new Car();
car.setMake("Audi");
car.setModel("TT");
for(int i=0;i<ileWyp;i++) {
    Equipment eq = new Equipment();
    eq.setDesc("desc"+i);
    session.persist(eq);
    car.getEqs().add(eq);
}
session.persist(car);
```

Zadanie

- Dodanie asocjacji pomiędzy encjami

Użycie JUnit

- Narzędzie do tworzenia testów jednostkowych
- Budowa testu – metody z adnotacjami
- @BeforeClass, @AfterClass – przed i po wszystkich testach
- @Before, @After – przed i po każdym teście
- @Test – metoda testująca, bezparametrowa, zwracająca void