Zbigniew ROSIEK[1]

MAPOWANIE OBIEKTOWO-RELACYJNE (ORM) – CZY TYLKO DOBRA IDEA?

Streszczenie

Mapowanie obiektowo-relacyjne (ORM) jest nowoczesnym podejściem do zagadnienia współpracy z bazą danych, wykorzystującym filozofię programowania obiektowego. Na wielu forach dyskusyjnych pojawiają się głosy za i przeciw ORM. W artykule omówiona zostanie ogólna koncepcja ORM oraz niektóre elementy jej realizacji w oparciu o standard Java Persistence API (JPA). Podstawowe zalety i wady ORM zostaną przedstawione w oparciu o nie.

Abstract

Object-relational mapping is a modern approach to the problem of interaction with the database using the objective programming paradigm.There are many views for and against ORM on multiple discussion lists. The article briefly describes a general idea of object-relational mapping and some details of the ORM implementation which uses Java Persistence API (JPA) standard. Basing on this, we will try to point out benefits and disadvantages of the ORM approach.

1. PROBLEM

Koncepcja programowania obiektowego i projektowania systemów informatycznych w oparciu o paradygmat obiektowości (np. UML) okrzepła, ustabilizowała się i jest obecnie standardem. Dysponujemy dobrymi narzędziami do wspomagania projektowania (np. Rational Rose czy też Enterprise Architect), mamy dojrzałe obiektowe języki programowania (Java, C#). Możliwe jest płynne przejście od biznesowych założeń, przez specyfikację wymagań, projekt techniczny do gotowego programu. Czy na pewno?

Jak dotąd nie powstała powszechnie akceptowana koncepcja obiektowej bazy danych. Pojawiły się oczywiście różne koncepcje obiektowych baz danych:

– repozytoria XML (cała baza to jeden/wiele dokumentów XML),

[1] Mgr inż. Zbigniew Rosiek jest wykładowcą Warszawskiej Wyższej Szkoły Informatyki

– rozszerzenia do języków typu Java czy C# pozwalające na przechowywanie zserializowanych obiektów,
– kilka ciekawych, acz wymagających dalszych prac jak (np. ODRA – semistrukturalna baza danych z stosowym językiem zapytań rozwijana na P-JWSTK.

W praktyce wszyscy korzystają nadal z relacyjnych baz danych, gdyż systemy zarządzania relacyjnymi bazami danych są szybkie, stabilne i bezpieczne. Poważnym problemem jest przełożenie logicznej obiektowej struktury systemu na relacyjną bazę danych, podobnie nie jest oczywiste korzystanie z takiej bazy danych w obiektowej aplikacji.

Przykładowo, aby zmienić hasło użytkownika w bazie danych użytkowników musimy wykonać poniższą operację:

```
   public void changePassword(User u, String newPassword) {
          con.prepareStatement(„UPDATE tab_users SET password = ?
WHERE login = ?”).
             setString(1, u.getPassword()).
             setString(2, u.getLogin()).execute();
   }
```

Chcielibyśmy, aby taka operacja wyglądała następująco:

```
          public void changePassword(User u, String newPassword) {
            u.setPassword(newPassword);
          }
```

a więc bez konieczności „grzebania” w relacyjnej bazie danych.

Z pomocą przychodzą narzędzia działające w oparciu o zasady mapowania relacyjno-obiektowego (ORM), które pozwalają:

- zdefiniować odwzorowanie zawartości relacyjnej bazy danych na obiekty w używanym przez nas języku programowania,
- wykonywać operacje na danych w bazie danych tak jak na zwykłych obiektach języka programowania.

Podstawowe zasady pracy z narzędziami ORM są następujące:

1. Tworzymy model danych w obiektowym języku programowania.

2. Tworzymy schemat bazy danych odpowiadający temu modelowi (może się oczywiście okazać, że już mamy jakąś bazę danych, z którą musimy współpracować).
3. Definiujemy odwzorowanie bazy danych na model relacyjny.
4. Tworzymy aplikację obiektową operującą na modelu obiektowym (tworząc kod aplikacji w zasadzie nie musimy się zastanawiać jak nasze obiekty zostaną zapisane).
5. W razie konieczności pobrania obiektów z bazy danych, bądź utrwalenia jakiegoś nowo utworzonego obiektu, czy też usunięcia utrwalonego obiektu – posługujemy się odpowiednim API danego narzędzia ORM. To jest jedyne miejsce, w którym w naszej aplikacji przejmujemy się tym, że współpracujemy z jakąś bazą danych.

W dalszej części artykułu ograniczymy się do środowiska Java, co oczywiście nie oznacza, że tylko w nim ORM jest zastosowane. W najnowszej wersji VS 2010 jest również dostępny aparat ORM z bardzo wygodnym interfejsem programisty.

2. HISTORIA

- 1996 – pierwsze narzędzia do mapowania ORM dla Javy – TopLink,
- 1999 – powstaje J2EE – standard tworzenia wielowarstwowych aplikacji-komponentowych. W jego skład wchodzi standard Enterprise Java Beans (1.1). Standard J2EE jest skomplikowany, miejscami mało czytelny, a do tego wymaga kosztownych serwerów aplikacji (aczkolwiek z czasem pokazują się darmowe). J2EE znajduje zastosowanie w dużych aplikacjach korporacyjnych.
- 2001-2002 – pojawia się szereg bibliotek i narzędzi dla Javy, mających być substytutem różnych funkcjonalności J2EE. W dziedzinie mapowania obiektowo-relacyjnego powstają narzędzia takie jak Hibernate, TopLink(zostaje przejęty przez Oracla), iBatis. Największą popularność zdobywa Hibernate. Rozwiązania ORM trafiają do małych i średnich systemów.
- 2006 – opublikowanie standardu Java EE 5, kolejnej wersji J2EE. Java EE 5 jest znacznie uproszczona w stosunku do dawnego J2EE, w jej skład wchodzi definicja standardu JPA – standardu definiowania mapowania obiektowo-relacyjnego i korzystania z trwałych obiektów. JPA jest dużo prostsze od EJB 1.1, między innymi dzięki temu, że twórcy zaczerpnęli bardzo dużo pomysłów z narzędzi takich jak Hibernate.

- 2006 – powstaje Hibernate EntityManager, implementacja JPA oparta o dotychczasowego Hibernate. Pozostali twórcy narzędzi ORM również dostosowują się do nowego standardu.

3. MAPOWANIE RELACYJNO-OBIEKTOWE W OPARCIU O JPA

Java Persistence API nie jest jakimś konkretnym narzędziem. Jest to tylko standard, specyfikacja, interfejsy programistyczne określające jak ma działać narzędzie JPA. Różni dostawcy oprogramowania (SUN, IBM, Oracle, twórcy narzędzi OpenSource – Hibernate) stworzyli różne implementacje JPA. Obecnie wydaje się najbardziej popularna implementacja JPA na serwerze aplikacji Java EE 5.

Teoretycznie pisana przez nas aplikacja powinna działać tak samo, niezależnie od tego z jakiej implementacji JPA korzystamy. W praktyce nie ma gwarancji, że aplikacja „z marszu" zadziała na nowym serwerze aplikacji – z reguły konieczne są niewielkie poprawki.

Istotne cechy JPA poznamy tworząc prosty system bankowy, w którym zakładamy, że:

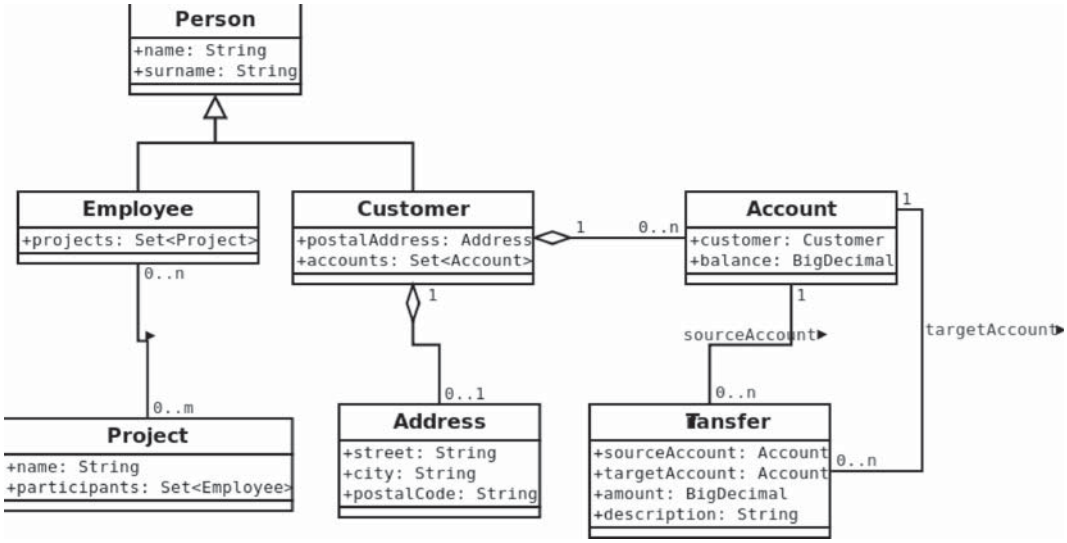
- bank ma wielu klientów,
- każdy klient ma jedno lub wiele kont,
- możliwe jest dokonywanie przelewów między kontami,
- konto ma aktualny bilans oraz stan (może być zamknięte, otwarte, zablokowane itd.).

Krok 1 – stworzenie modelu danych:

Zaczynamy od stworzenia modelu danych. Na razie nie ma nic związanego z JPA. Tworzymy zwykłe klasy JavaBeans zawierające interesujące nas informacje.

Standard klas JavaBeans może brzmi groźnie, natomiast sprowadza się do kilku prostych zasad:

- klasa musi mieć publiczny bezparametrowy konstruktor,
- dla każdego widocznego publicznie atrybutu klasy o nazwie xyz muszą istnieć metody getXyz() (isXyz dla typu boolean) i setXyz(), odpowiednio zwracające i ustawiające wartość tego atrybutu,
- odwołanie do atrybutów JavaBeana odbywa się za pomocą getterów i setterów a nie bezpośrednio.



Krok 2 – zdefiniowanie odwzorowania:

Odwzorowanie modelu obiektowego na model bazy relacyjnej można opisać w formie pliku XML (tak było w Hibernate na początku) lub za pomocą adnotacji Javy – bezpośrednio w kodzie klas definiujących model (każdy z tych sposobów ma swoich zwolenników). W kolejnych przykładach opierać się będziemy na adnotacjach. Mechanizm konfiguracji oparty o adnotacje hołduje zasadzie „convention over configuration", czyli jeśli środowisko JPA może się czegoś domyślić, to nie trzeba tego definiować w kodzie, określamy tylko ogólną strategię nazewnictwa (np. tabela w bazie nazywa się tak jak mapowana klasa). Poza tym musimy zdefiniować tylko to co jest nietypowe i niezgodne z obraną strategią. Aby klasa została odwzorowana na tabelę w bazie danych musimy dołączyć do niej adnotację *@Entity*. Wówczas każdy utrwalony obiekt tej klasy będzie zapisywany w tabeli o nazwie takiej jak nazwa klasy. Dla każdego obiektu przechowane będą wartości wszystkich atrybutów (oczywiście domyślnie atrybut o danej nazwie zostanie zapisany w formie kolumny o danej nazwie). Adnotacjami może zostać oznaczony zarówno atrybut, jak i jego getter lub setter – efekt będzie ten sam.

Trwały obiekt powinien mieć jakiś identyfikator – w przypadku bazy danych jest to ***klucz główny***. Możemy określić, który z atrybutów klasy ma być odwzorowany na klucz główny obiektu. W tym celu oznaczamy ten atrybut adnotacją ***@Id***. Zwykle klucze główne mają wartości nadawane przez jakiś generator. W JPA można ustalić, że przy utrwalaniu nowego rekordu wartość pewnych pól (oznaczonych adnotacją ***@GeneratedValue***) zostanie wygenerowana w oparciu o wybrany generator (tu już od

konfiguracji i bazy danych zależy, czy będzie to pole autonumerowane, sekwencja, wartość UUID (na tyle duża liczba losowa, że możemy być pewni, że nikt drugiej takiej nie wylosował – używana do generowania identyfikatorów w rozproszonych systemach – zwykle ma 128 bitów co daje 10^38 możliwych kombinacji).

Przykład 1:

```
@Entity
public class Account {
@Id
@GeneratedValue()
int identifier;
public int getIdentifier() {
          return number;
}
public void setIdentifier(int identifier) {
        this.identifier = identifier;
}
```

Domyślnie wszystkie atrybuty obiektu o prostych typach (int, string, boolean) są odwzorowywane na kolumny w tabeli. Możemy zmienić domyślne ustawienia mapowania dodając do atrybutu adnotację ***@Column***. Poszczególne parametry tej adnotacji pozwalają nam na określenie między innymi:

- nazwy kolumny w bazie danych odpowiadającej temu atrybutowi (parametr name),
- możliwość przyjmowania wartości null w kolumnie.

Rzeczywista siła narzędzi ORM tkwi w wygodnej obsłudze relacji między obiektami. W praktyce rozróżniamy trzy klasy relacji:

– jeden-do-jednego,
– wiele-do-jednego,
– wiele-do-wielu

JPA obsługuje każdy z powyższych rodzajów relacji. Przyjrzyjmy się najpopularniejszej relacji wiele-do-jednego na przykładzie relacji między obiektami ***Account*** a obiektami ***Customer*** naszego przykładu:

- Każdy klient może mieć pewną ilość kont (zakładamy że tych kont jest w miarę mało).

- Każde konto musi mieć referencję do jednego obiektu ***Customer*** definiującego użytkownika (mamy dwie relację wiele-kont-do-jednego-klienta).

W bazie relacyjnej taka referencja byłaby zapisana jako klucze główne rekordów w tabeli ***Customer.*** W JPA wystarczy, że zdefiniujemy atrybut „owner" typu ***Customer*** oraz oznaczymy go adnotacjami:

@JoinColumn – wartość danego pola jest identyfikatorem obiektu przechowywanego w tabeli ***Customer***

@ManyToOne – relacja zdefiniowana przez dane pole jest relacją wiele-do-jednego i my jesteśmy po stronie „wiele". Domyślnie obie te adnotacje nie wymagają żadnych argumentów.

Jeśli nie potrzebujemy, nie musimy definiować drugiej strony relacji. W przypadku klienta przydałaby się nam jednak możliwość dostępu do kont naszego klienta z poziomu obiektu ***Customer***. Aby zdefiniować drugą stronę relacji tworzymy w obiekcie ***Customer*** atrybut ***accounts*** typu ***Set<Account>.*** Atrybut ten oznaczamy adnotacją ***@OneToMany(mappedBy="owner").*** Tu ważny jest parametr ***„owner",*** który mówi, że dla danego obiektu ***Customer*** jego atrybut ***„accounts"*** reprezentuje listę wszystkich kont, które w swoim polu ***„owner"*** mają referencję do danego klienta.

W praktyce korzyść z odwzorowania strony (wiele) w relacji wiele-do-jednego jest niewielka. Gdy np. chcielibyśmy pobrać przelewy wykonane przez danego klienta i tak z reguły będziemy wykonywać jawnie zapytanie, które będzie wprowadzało jakieś warunki, filtrowanie, stronicowanie itd. Ze zdefiniowanych w ten sposób relacji będziemy też intensywnie korzystać przy wykonywaniu zapytań.

Relacje jeden-do-jednego z punktu widzenia użytkownika wyglądają dokładnie tak samo, jak relacje jeden-do-wielu. (tzn. każda ze stron relacji może mieć referencję do drugiej strony), nie będziemy się więc im szczegółowo przyglądać.

Oczywiście najciekawsza jest relacja wiele-do-wielu, ponieważ jej realizacja w SQL'u wymaga stworzenia dodatkowej, niewidocznej z punktu widzenia JPA tabeli. Wyobraźmy sobie relację pracownik-projekt. Pracownik może uczestniczyć w wielu projektach, w projekcie może uczestniczyć wielu pracowników. W wydaniu SQL'owym potrzebowalibyśmy do tego tabeli „assigned_projects" zawierającej kolumny ***„person_id"*** i ***„project_id".*** Rekord w tej tabeli to przypisanie pracownika do projektu. W JPA możemy zdefiniować takie odwzorowanie poprzez dodanie do klasy ***Person*** atrybutu ***assignedProjects*** typu ***Set<Project>*** i oznaczenie go następującymi adnotacjami:

```
@ManyToMany
@JoinTable(name="assigned_projects")
Set<Project> assignedProjects;
```

Adnotacje te oznaczają, że relacja wiele-do-wielu dotyczy osób (klasa ***Person***) i projektów (klasa ***Projects***). Deklaracja relacji wiele-do-wielu z drugiej strony jest opcjonalna. Jeśli potrzebny jest dostęp do listy osób zaangażowanych w projekt, wówczas do klasy ***Project*** musimy dodać następujący atrybut:

```
@ManyToMany(mappedBy="assignedProjects")
Set<Person> assignedPersons.
```

Obiekty zagnieżdżone – może się zdarzyć, że chcielibyśmy logicznie wyodrębnić jakiś podobiekt z obiektu modelu, natomiast nie chcielibyśmy tworzyć do tego celu oddzielnej tabeli (przykładem może być np. adres). Chcielibyśmy w „programie" pracować z obiektami typu ***Adres***, a więc:

1. Klasa ***Customer*** powinna mieć atrybut o nazwie ***adresKorespondencyjny*** typu ***Address*** (oddzielna klasa),
2. Nie tworzymy oddzielnej tabeli na adresy, lecz dodajemy w tabeli klientów dodatkowe kolumny ***adresKorespondencyjny_ulica***, ***adresKorespondencyjny_kodPocztowy***, ***adresKorespondencyjny_miasto***

Takie odwzorowanie można zrealizować w JPA za pomocą adnotacji ***@Embeddable*** (oznaczamy nią klasę składową obiektu złożonego) oraz ***@Embedded*** (stosujemy ją w atrybucie, który ma zostać zapisany jako obiekt złożony).

Przykład:

- Klasa **Address** i atrybut **Customer#address**.
- Domyślna strategia nazywania pól w obiektach ***Embedded*** to ***nazwaAtrybutuSkladowego***.

Przy użyciu odpowiednich adnotacji można nazwy tych pól zmienić.

JPA pozwala na odwzorowanie (w ograniczonym przez nas zakresie) w bazie danych struktury **dziedziczenia klas**. Wyobraźmy sobie, że mamy klasę ***Person*** oraz jej podklasy ***Customer*** i ***Employee***. JPA pozwala na swobodne definiowanie relacji zarówno z nadklasą jak i podklasami. Podobnie przy wykonywaniu zapytań możemy określić, czy interesują nas wszystkie osoby (obiekty klasy ***Person*** jak i wszystkich jej podklas), czy może konkretna podklasa.

Mechanizm odwzorowania dziedziczenia na bazę danych ma pewne ograniczenie w stosunku do w zwykłej aplikacji obiektowej, np. dodanie nowej podklasy w istniejącej aplikacji będzie najprawdopodobniej wymagało rozszerzenia struktury bazy danych. W praktyce JPA jest w stanie efektywnie obsługiwać raczej małe i płaskie hierarchie (jeden/dwa poziomy, kilka/kilkanaście klas). Należy więc uważać na zbyt swobodne tworzenie nowych podklas.

Oprócz omówionych podstawowych sposobów odwzorowań JPA pozwala na skonfigurowanie wielu nietypowych właściwości odwzorowania – specyficznych dla danego narzędzia ORM albo dla specyficznej bazy danych. Narzędzia implementujące JPA mogą być rozszerzone o dodatkowe mechanizmy nie związane z bazą danych np.:

- Cache'owanie (JPA buforuje obiekty w zewnętrznym cache'u – jeśli nie potrzeba, nie pyta bazy danych),
- Zaawansowana walidacja (np. dla NIP, PESEL),
- Indeks pełnotekstowy (oznaczamy, które obiekty i ich atrybuty mają być indeksowane w zewnętrznym indeksie pełnotekstowym). Odpowiedni moduł do JPA zapewni, że indeks będzie uaktualniany przy każdym zapisie do bazy.

4. PRACA Z JPA

W przypadku relacyjnej bazy danych każde zapytanie wykonywane jest w ramach określonego połączenia z bazą danych i w ramach określonej transakcji. Bardzo podobnie wygląda sprawa w przypadku JPA. Aby korzystać z JPA musimy utrzymać obiekt klasy EntityManager, który jest odpowiedzialny za zarządzanie cyklem życia obiektów i pozwala na wykonywanie operacji JPA takich jak: utrwalenie obiektu, pobranie trwałego obiektu, czy też wykonanie zapytania. W ramach określonego EntityManagera rozpoczynamy i kończymy transakcje – zatem za odpowiednik EntityManagera w relacyjnych bazach danych można przyjąć pojedyncze połączenie z bazą danych. W prostych aplikacjach WEB'owych sensownym założeniem jest przyjęcie, że obsługa pojedynczego wywołania HTTP jest realizowana przez jeden EntityManager i w ramach jednej transakcji.

Utworzenie obiektu trwałego sprowadza się do stworzenia obiektu klasy oznaczonej adnotacją ***Entity*** i wypełnieniu jego atrybutów pożądanymi wartościami. Aby utrwalić go w bazie danych wykonujemy operację:

```
entityManager.persist(obiekt);
```

W tym momencie obiekt (wraz z ewentualnymi zależnościami – patrz relacje) zostanie zapisany w bazie danych. Jeśli zawartość jakiegoś pola w obiekcie miała

być generowana w bazie danych (np. unikalny identyfikator rekordu), to jej wartość zostanie nadana podczas wykonania powyższej metody.

Modyfikacja obiektu trwałego polega po prostu na zmianie wartości jego atrybutów. Nie musimy w żaden sposób wykonywać jawnej operacji zapisu. JPA rejestruje wszystkie zmiany obiektów trwałych i w momencie zatwierdzenia transakcji zapisze zmiany w bazie danych.

```
account.setName('xxx');
```

Nie należy obawiać się o wydajność – nawet jeśli wykonaliśmy wiele operacji na danym obiekcie to JPA przetłumaczy to na jedno zapytanie „UPDATE".

Najprostszym sposobem **odczytania obiektu** z bazy danych jest pobranie go na podstawie klucza publicznego. Do tego celu służy metoda ***load()*** entityManagera.

```
Account account = entityManager.load(Account.class, '1234-5678');
<- pobieramy obiekt klas „Account" o kluczu głównym '1234-5678'
```

Jeśli obiekt o takim kluczu nie istnieje otrzymamy w wyniku null.

Usuwanie obiektu jest analogiczne do jego utrwalania – służy do tego metoda ***em.remove()*** EntityManagera.

Mając już zdefiniowane relacje możemy zacząć dokonywać operacji na bazie danych. Stworzenie konta i przypisanie go do istniejącego klienta wygląda następująco:

```
Customer customer = <pobieramy skądś istniejącego klienta>
Account newAccount = new Account();
newAccount.setNumber(„1234567");
newAccount.setOwner(customer);
entityManager.persist(newAccount);
```

Jeśli teraz np. chcemy pobrać sumaryczny bilans kont klienta możemy wykonać taką operację:

```
Customer customer=<pobieram skądś istniejącego klient>

BigDecimal balance = BigDecimal.ZERO;
for (Account account : customer.getAccounts()) {
        balance = balance.add(account.getBalance());
}
System.out.println(„Bilans wszystkich kont klienta" + customer.
getName() + „ wynosi " + balance);
```

Oczywiście stosując różne parametry można zwiększać efektywność działania aplikacji. Np. dodając parametr `fetch=FetchType.EAGER` do adnotacji ***ManyToOne*** w polu ***Account.owner*** spowodujemy, że dane o właścicielu konta będą pobierane od razu podczas pobierania konta. Wtedy JPA może w ramach optymalizacji użyć JOIN'a i wykonać jedno zapytanie zamiast dwóch.

JPA wprowadza swój własny **język zapytań** – EJB-QL. Na pierwszy rzut oka język jest podobny do SQL'a:

```
SELECT c FROM Customer c WHERE c.name = 'Jan' AND c.surname =
'Kowalski'.
```

Pierwsza różnica widoczna w przykładowym zapytaniu – obiektowy język zapytań zwraca obiekty a nie pojedyncze kolumny, stąd mamy „SELECT c" a nie „SELECT c.*" jak to wygląda w przypadku SQL'a. Druga sprawa – nie posługujemy się tu nazwami tabel i kolumn z bazy danych, a oczywiście nazwami klas i atrybutów. Większa różnica pojawia się kiedy chcemy pobrać dane na temat wielu obiektów powiązanych ze sobą – spróbujmy na przykład pobrać listę wszystkich kont, które mają debet, oraz właściciela o imieniu Jan. W SQL'u odpowiednie zapytanie miałoby postać:

```
SELECT a.* FROM Account a LEFT JOIN Customer c ON c.id =
a.owner_id WHERE a.balance < 0 AND c.name = „Jan"
```

W EJB-QL zapytanie jest dużo prostsze:

```
SELECT a FROM Account WHERE a.owner.name = „Jan" AND a.balance < 0
```

Nie ma w nim JOIN'ów. EJB-QL pozwala nam na dostęp do zagnieżdżonych atrybutów. W rzeczywistości zapytania EJB-QL są tłumaczone na SQL, czasem wychodzi to lepiej, czasem gorzej. Jeśli gorzej, to dobrze byłoby, abyśmy mogli skorzystać z zapytania w SQL. JPA zostawia nam furtkę pozwalającą na wywołanie „czystego" zapytania SQL i ręczne odwzorowanie jego wyniku na obiekty modelu. Do tego celu służy mechanizm JPA ***Native Queries***.

Można już chyba zastanowić się "jak to działa?". Zwykłe obiekty Javy po utrwaleniu za pomocą JPA zaczynają się zachowywać specyficznie, tzn. zmiana ich atrybutów pociąga za sobą pewne operacje na bazie danych. Generalnie obiekty trwałe zachowują się inaczej niż obiekty zwykłe. W praktyce (przynajmniej w przypadku Hibernate EntityManagera) za tą zmianą w działaniu obiektów w modelu stoi

biblioteka CGLIB. Pozwala ona na zmianę kodu klas Javy „w locie" – podczas pracy programu. Hibernate, za pomocą CGLIB zmienia treść metod getXX, setXX klas należących do modelu – dodając do nich mechanizm pobierania brakujących danych z bazy danych oraz informowanie EntityManagera związanego z danym obiektem modelu o tym, że obiekt uległ zmianie.

Jedynym sposobem w jaki narzędzia JPA dostają się do bazy danych jest mechanizm zapytań SQL. Każda instancja EntityManagera utrzymuje otwarte połączenie z bazą danych, a operacje otwarcia/zakończenia transakcji Entity Managera wiążą się z otwarciem/zamknięciem transakcji bazodanowej związanej z tym połączeniem. W praktyce oznacza to, że za pomocą JPA możemy zrobić tylko tyle co za pomocą zwykłych zapytań SQL. ***Warstwa JPA służy tylko temu aby było wygodniej, a kod był czytelniejszy.***

Poszczególne implementacje JPA pozwalają na wygodną i przezroczystą integrację np. z cachem w pamięci (wówczas zbuforowane obiekty nie muszą być pobierane z bazy danych). Istnieje rozszerzenie Hibernate JPA pozwalające na zintegrowanie aplikacji z biblioteką do tworzenia indeksów tekstowych ***Lucene***. Dzięki temu jesteśmy w stanie w bardzo łatwy sposób rozszerzyć aplikację o efektowny tekstowy interfejs wyszukiwawczy – jak w wyszukiwarce Google.

Rzućmy jeszcze okiem na problemy mogące wyniknąć z „sesyjności" współpracy z bazą danych. Każdy trwały obiekt, z którym pracujemy jest związany z konkretną instancją Entity Managera – tą w której obiekt pobraliśmy bądź utworzyliśmy. Praca z takim obiektem jest prosta i oczywista w momencie gdy Entity Manager związany z obiektem działa i utrzymuje połączenie z bazą danych. Problemy pojawiają się, gdy sesja Entity Managera w ramach której obiekt został utworzony/pobrany zostanie zamknięta. JPA posiada mechanizmy pozwalające na "odłączenie" obiektu od sesji (co wiąże się z pobraniem wszystkich interesujących nas atrybutów tego obiektu z bazy), a także na późniejsze "podłączenie" tego obiektu z powrotem w innej sesji. Przy "podłączeniu" następuje synchronizacja (potencjalnie zmienionego) odłączonego obiektu z zawartością bazy danych. Na tym etapie możliwe jest uaktualnienie bazy danych o zmiany dokonane na "odłączonym" obiekcie. Pozwala to między innymi na łatwe tworzenie interfejsów użytkownika, które nie wymagają ciągłego połączenia z bazą danych – np. Interfejs www.

W tym miejscu można już śmiało powiedzieć, że w JPA, a więc zapewne w każdym innym narzędziu umożliwiającym korzystanie z ORM, oprócz ewidentnych korzyści musimy się też liczyć z pewnymi niedogodnościami lub właściwościami pogarszającymi pewne cechy końcowego produktu. Jednym z nich jest tzw. **niedopasowanie impedancji.** Model obiektowy różni się nieco od modelu relacyjnego. Narzędzia ORM starają się te różnice zredukować, jednak czasem możemy mieć do

czynienia ze zbiorem danych w jednym z modeli, którego nie da się w sposób jednoznaczny przetłumaczyć na drugi model gdyż:

1. Baza danych ma **inny system typów niż używany przez nas język programowania** (np. w bazach danych typy tekstowe często mają ograniczoną długość);
2. **Polimorfizm** występujący w języku obiektowym nie ma prostego odwzorowania w bazie danych. Z drugiej strony niektóre odwzorowania poliformizmu w bazach relacyjnych pozwalają na operacje, które nie są dozwolone w najpopularniejszych językach obiektowych (np. zmiana typu obiektu – jeśli w bazie danych mamy pole „typ" i w ten sposób radzimy sobie z określeniem typu obiektu przechowywanego w tabeli, to możemy ten typ zmienić. W językach typu Java czy C# zmiana typu obiektu „w locie" jest trudna do odwzorowania i nieintuicyjna;
3. Inna jest **filozofia zapisu relacji** – w bazach relacyjnych relacja jest reprezentowana przez klucz obcy, w językach obiektowych obiekt może mieć referencję do innego obiektu (w językach obiektowych częściej spotykamy sytuację gdy relacja jest dostępna z obu stron, tzn. każdy z obiektów będących stroną relacji ma referencję do tego drugiego – co się powinno stać jeśli TYLKO z jednej strony relacji ustawimy wartość na „null"?);
4. Model obiektowy nie pozwala na pełne wykorzystanie możliwości bazy relacyjnej (widoki, złączenia nie po kluczach, bardziej skomplikowane zapytania, masowe operacje zmiany danych (n.p. ***UPDATE account SET closed = true WHERE balance < 0***));
5. **Model relacyjny nie pozwala na efektywną pracę z niektórymi danymi obiektowymi** (wielopoziomowe zagnieżdżenie obiektów, listy);
6. Zmiany dokonywane po stronie bazy danych – **jeśli w bazie danych zdefiniowane są jakieś wyzwalacze**, to system ORM nie jest w stanie dowiedzieć się, że wyzwalacz zmienił wartość jakiegoś obiektu – istnieje ryzyko, że wartość obiektu, z którym pracujemy w Javie zmieniła się. W praktyce użycie bardziej zaawansowanych mechanizmów relacyjnych bazach danych(widoki, procedury składowane itd.) jest utrudnione.

Oczywiście żaden z powyższych problemów nie dyskwalifikuje narzędzi ORM – po prostu należy być czujnym i mieć na względzie potencjalne problemy.

Jako że JPA jest kolejną warstwą nałożoną na mechanizm dostępu do bazy danych, więc wprowadza swój narzut wydajnościowy na wykonywane operacje. Aplikacja korzystająca z JPA zawsze będzie wolniejsza od aplikacji bezpośrednio wykonującej operacje SQL.

W przypadku bardziej skomplikowanych zapytań do baz danych – agregujących dane, intensywnie korzystających z indeksów, funkcji (często specyficznych dla określonego systemu bazodanowego) dodawanie dodatkowej warstwy w postaci języka EJB-QL, co do którego nie wiemy, jak jej zapytania zostaną na SQL'a przetłumaczone prawie na pewno spowoduje spadek wydajności. W takiej sytuacji wolelibyśmy bezpośrednio pisać kod w języku SQL, a więc idea ORM „traci rumieńce".

W tym momencie można więc zadać pytanie, czy aby „zbyt wiele pary nie idzie w gwizdek?". Podobne pytanie można by zadać także w przypadku Web serwisów czy stosowania komponentów zdalnych, a przecież są one bardzo popularne i powszechnie stosowane. Oczywistym jest, że tak Web serwisy jak komponenty zdalne czy narzędzia ORM w pewnych systemach będą poprawiały ich jakość, efektywność, zmniejszały koszty realizacji projektu – w innych wręcz przeciwnie. Są to z całą pewnością artefakty, które dobrze zastosowane przynoszą ewidentne korzyści, zaś rolą analityków i projektantów jest właściwe ich zastosowanie tam, gdzie te korzyści można będzie osiągnąć.

Literatura

[1] http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html – specyfikacja standardu JPA
[2] http://docs.jboss.org/hibernate/stable/entitymanager/reference/en/html/ – Hibernate – dokumentacja JPA
[3] http://download.oracle.com/docs/cd/B14117_01/appdev.101/b10807/toc.htm
[4] Piotr Błoch, Marek Wojciechowski, *Analiza porównawcza technologii odwzorowania obiektowo-relacyjnego dla aplikacji Java*, 2007, Politechnika Poznańska