Master rad

Tema: Razvoj Spring aplikacije primenom Hibernate okvira

Mentor: dr Gordana Pavlović-Lažetić

Student: Veljko Urošević
1009/2011, računarstvo i informatika

Matematički fakultet, Beograd
Septembar 2013.

Sadržaj

Apstrakt

Objektno/relaciono preslikavanje (engl. *object/relational mapping*, ORM) je aktuelan problem sa kojim se programeri sreću u savremenim aplikacijama. Jedan je od osnovnih izvora problema u velikom broju aplikacija. Java programski jezik nameće se kao prvi izbor za jezik u kome će se razvijati aplikacija, a podaci će se smeštati unutar relacione baze podataka. Za rešavanje problema neslaganja između objektno-orijentisane prirode programskog jezika, sa jedne strane, i relacione prirode baze podataka, sa druge, primenjuje se više različitih rešenja, a možda najefektivnije predstavlja okvir Hibernate, uz korišćenje pogodnosti koje omogućava Spring. Upravo će okvir Hibernate i njegov pristup rešavanju problema objektno/relacionog preslikavanja biti centralna tema ovog rada. Biće prikazana i drugačija rešenja (čisti JDBC, JDBC integrisan sa okvirom Spring). Cilj takvih digresija je potvrda elegancije i jednostavnosti koju sa sobom nosi okvir Hibernate.

1. Trajnost podataka i objektno/relaciono preslikavanje

1.1 Uvod

Opšta predstava o pojmu trajnosti podataka se bitno promenila sa pojavom Java Persistence API-ja (Application Programming Interface), koji je predstavljao novi standard za objektno/relaciono preslikavanje za Java Enterprise Edition, kao i za Java Standard Edition. Unutar Java zajednice se koristi određeni broj okvira koji implementiraju objektno/relaciono preslikavanje. Hibernate, kao vodeći među tim okvirima, je imao značajan uticaj na način na koji se u Javi ostvaruje trajnost podataka. Hibernate predstavlja direktan i pragmatičan pristup objektno/relacionom preslikavanju.
Relacione baze podataka su bez sumnje u samom jezgru svih modernih aplikacija. Dok moderni programski jezici, uključujući tu i Javu, pružaju intuitivni, objektno-orijentisani pogled i shvatanje elemenata koji čine poslovne aspekte aplikacija, podaci koji se nalaze u pozadini aplikacije su u suštini relacione prirode.
Bilo je mnogo pokušaja da se premosti jaz koji postoji između relacionih i objektno-orijentisanih tehnologija, ili da se pak jedna tehnologija zameni drugom. Upravo to neslaganje predstavlja izazov koji okvir Hibernate pokušava da reši – da predstavi most između podataka koji su relacioni i Java objekata.
Efektivna upotreba tehnologija za ORM (objektno/relaciono preslikavanje) u svim, osim u najprostijim okruženjima i alatima, zahteva razumevanje postupka i načina na koji se posreduje između relacionih podataka i objekata. To znači da programer treba da bude upoznat i sa zahtevima aplikacije i sa zahtevima koje sa sobom nose podaci, a takođe i upoznat sa SQL-om, relacionim strukturama, kao i sa mogućnostima za optimizaciju koje relaciona tehnologija nudi.
Ne samo da Hibernate pruža direktno, puno funkcionalno rešenje za sve ove zahteve, već je istovremeno i fleksibilna arhitektura podložna podešavanjima. Tim koji je razvijao Hibernate je to radio vodeći računa o modularnosti, ugradivosti, proširivosti i optimizacijama koje bi izvršili sami korisnici. Kao rezultat toga, Hibernate je za nekoliko godina od pojave postao jedna od vodećih ORM tehnologija koje koriste programeri.[1]
Ogroman broj kompanija koristi SQL baze podataka. Java je predstavljena u industriji kao prvi izbor za razvoj poslovnih (engl. *enterprise*)[1] aplikacija. Tabelarno predstavljanje podataka u relacionom sistemu je još uvek suštinski drugačije u odnosu na mrežu objekata koji se koriste u objektno-orijentisanim Java aplikacijama. Sa jedne strane Java programeri vide relacionu tehnologiju kao izvor neslaganja, a sa druge profesionalni korisnici baza podataka krive objektnu tehnologiju.

1.2 Trajnost podataka

Nekoliko godina trajnost podataka je predstavljala jednu od najbitnijih tema, oko koje su se vodile debate u Java zajednici. Mnogi programeri se čak ne slažu sa značajem koji se pridaje ovom problemu. Da li je trajnost podataka problem koji je već rešen korišćenjem relacione tehnologije i njenih proširenja, u koje spadaju pohranjene procedure (engl. *stored procedures*), ili je to više problem koji je sveprisutan, kome se mora pristupiti preko specijalnih Java komponenti, kao što je EJB (Enterprise Java Beans)? Da li je potrebno da programer kodira čak i osnovne CRUD (*create*, *read*, *update*, *delete*) operacije u SQL i JDBC, ili bi taj posao trebalo da bude automatizovan? Kako se postiže portabilnost ukoliko svaki sistem za upravljanje bazom podataka ima sopstveni SQL dijalekt?

[1] *Sintagma poslovna aplikacija je jedan od najčešće korišćenih termina u današnjem razvoju softvera. Ipak, kada neko izjavi da radi na razvoju poslovne aplikacije, svima je prva asocijacija koju imaju u glavi: informacija. Poslovne aplikacije se definišu potrebom da prikupljaju, transformišu i prijavljuju ogromne količine informacija. Naravno, ta informacija ne postoji jednostavno u eteru. Skladištenje i čitanje podataka je posao čija se vrednost meri milijardama dolara.[10]*

Da li treba u potpunosti napustiti SQL i usvojiti neku drugu tehnologiju baza podataka, kao što su, na primer, objektne baze? Debata se nastavlja, ali rešenje pod nazivom objektno/relaciono preslikavanje je široko prihvaćeno. Hibernate je otvoreni kod (engl. *open source*) implementacija ORM servisa.
Hibernate je ambiciozni projekat, koji ima za cilj da predstavlja kompletno rešenje za problem trajnosti podataka u Javi. Posreduje u interakciji aplikacije sa relacionom bazom, i ostavlja programeru vreme da se skoncentriše na poslovnu stranu problema. Od programera se ne zahteva da prati mnoga pravila koja su specifična za sam Hibernate i šablone dizajna (engl. *design patterns*) prilikom pisanja poslovne logike i trajnih klasa; samim tim, Hibernate se glatko integriše sa većinom novih i postojećih aplikacija i ne zahteva drastične promene koje utiču na ostatak aplikacije. Hibernate implementira Java Persistence i podržava standardizovana mapiranja, upite i API-je.

1.2.1 Šta je trajnost podataka?

Skoro sve aplikacije zahtevaju trajne podatke. Trajnost podataka je jedan od osnovnih koncepata prilikom razvoja aplikacija.
Relacione baze su široko rasprostranjene zato što predstavljaju veoma fleksibilan i robustan pristup kontrolisanja podataka.
Sistemi za upravljanje relacionom bazom nisu specifični za Javu, niti je relaciona baza podataka specifična za određenu aplikaciju. Ovaj važni princip je poznat pod nazivom nezavisnost podataka. Drugim rečima, podaci žive znatno duže nego što je to slučaj sa bilo kojom aplikacijom. Relaciona tehnologija pruža način za deljenje podataka između više aplikacija, ili između različitih tehnologija koje formiraju delove iste aplikacije. Relaciona tehnologija je zajednički imenilac različitih sistema i platformi. Prema tome, relacioni model podataka je često uobičajen za reprezentaciju poslovnih entiteta u poslovnim aplikacijama.

1.2.2 Trajnost podataka u objektno-orijentisanim aplikacijama

U objektno-orijentisanoj aplikaciji, trajnost podataka omogućava objektu da nadživi proces koji ga je kreirao. Stanje objekta će biti sačuvano na disku, i objekat sa tim stanjem će biti ponovo kreiran kada bude bio potreban.
Ovo nije ograničeno na pojedinačne objekte – čitave mreže međusobno povezanih objekata se mogu sačuvati i kasnije biti ponovo kreirane. Većina objekata nije trajna; objekat označen kao privremen (engl. *transient*) ima ograničen životni vek, koji je vezan za životni vek procesa koji ga je kreirao. Skoro sve Java aplikacije predstavljaju mešavinu trajnih i privremenih objekata; prema tome, javlja se potreba za sistemom koji rukuje trajnim podacima.
Kada se govori o trajnim podacima, misli se pre svega na:

- smeštanje, organizaciju, i pretragu podataka,
- konkurentnost i integritet podataka i
- deljenje podataka.

1.3 SQL u Javi

Kada se radi sa SQL bazom u Java aplikaciji, Java kod izdaje SQL naredbe bazi preko Java Database Connectivity (JDBC) API-ja. Bilo da je SQL pisan ručno i ugrađen u Java kod, ili generisan od strane samog Java koda, JDBC API se koristi za povezivanje argumenata kako bi se pripremili parametri upita, izvršavali upiti, prolazilo kroz tabelu rezultata upita, uzimale vrednosti iz skupa rezultata, itd. Sve to predstavlja zadatke niskog nivoa pristupanja podacima. Programere aplikacija više interesuje

poslovna logika problema kome je potreban pristup podacima. Ono čemu se teži jeste pisanje koda koji objekte – instance klasa – smešta u i uzima iz baze, bez detaljnih informacija o načinu na koji se to realizuje.
Međutim, programeri sve češće govore o neslaganju paradigmi. Ovo neslaganje objašnjava zašto se toliko truda ulaže u rešavanje zadataka koji se tiču trajnosti podataka u svakom poslovnom projektu. Paradigme o kojima je ovde reč su objektni model i relacioni model, ili se čak može reći i objektno-orijentisano programiranje u SQL-u.

1.3.1 Čisti JDBC

Tradicionalni način za pristupanje relacionoj bazi podataka je Java Database Connectivity (JDBC) API. Neki od uobičajenih problema koji se javljaju prilikom rada sa JDBC su:

- Programer mora voditi računa o svim konekcijama sa bazom podataka. Uvek postoji rizik da konekcije nisu zatvorene, što može uzrokovati probleme.
- Pisanje ogromne količine koda, zato što sve instancne promenljive koje su potrebne za umetanje, ažuriranje i upite moraju biti eksplicitno navedene.
- Programer vodi računa i o vezama među objektima. Ukoliko se radi sa kompleksnim podacima, ovo može predstavljati glavni problem.
- Kod nije portabilan na druge baze podataka.

1.3.2 JDBC inicijalizacija

Najpre je potrebno uključiti odgovarajuću *.jar datoteku u projekat. Pošto se u ovoj konkretnoj aplikaciji koristi PostgreSQL, potrebno je sa sajta `http://www.postgresql.org/` skinuti poslednju verziju drajvera. Nakon toga, potrebno je unutar Java klase u kojoj će biti implementirana konekcija, učitati JDBC drajver i ostvariti konekciju sa bazom podataka, pre nego što se može izvršiti bilo koja SQL naredba. JDBC drajver za PostgreSQL se nalazi u `postgresql-9.2-1003.jdbc4.jar` datoteci (poslednja dostupna verzija u vreme izrade ovog rada). Bitno je zatvoriti sve konekcije. Ovo je bitno zato što su konekcije zahtevne u pogledu resursa koje koriste – ukoliko nisu zatvorene nakon završetka rada, aplikacija može ostati bez resursa.

```
Class.forName("org.postgresql.Driver");
Connection connection =
DriverManager.getConnection("jdbc:postgresql://localhost:5432/postgres", "postgres",
"password");
try {
	//uzimanje podataka ili ažuriranje baze
} finally {
	connection.close();
}
```

1.3.3 Čitanje grafa objekata

Cilj je prikazati detalje o jednom igraču.

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM club_info,
player_info WHERE club_info.club_id = player_info.club_id AND person_id = ?");
statement.setInt(1, 8);
ResultSet resultSet = statement.executeQuery();

Igrac igrac = new Igrac();
if(resultSet.next()) {
```

```
        igrac.setOsobaId(resultSet.getInt("person_id"));
        igrac.setIme(resultSet.getString("name"));
        igrac.setBrojDresa(resultSet.getInt("jersey_no"));

        Klub klub = new Klub();
        klub.setKlubId(resultSet.getInt("club_id"));
        klub.setImeKluba(resultSet.getString("club_name"));
        igrac.setKlub(klub);
}

resultSet.close();
statement.close();

Set<Statistika> regularna = new HashSet<Statistika>();
statement = connection.prepareStatement("SELECT * FROM regular_season WHERE player_id =
?");
statement.setInt(1, 8);
resultSet = statement.executeQuery();
while(resultSet.next()) {
        Statistika statistika = new Statistika();
        statistika.setGodina(resultSet.getInt("year"));
        statistika.setApg(resultSet.getDouble("reg_apg"));
        statistika.setRpg(resultSet.getDouble("reg_rpg"));
        statistika.setPpg(resultSet.getDouble("reg_ppg"));

        regularna.add(statistika);
}

igrac.setStatistike(regularna);
```

Da bi bili kreirani objekti klasa `Igrac` i `Klub`, mora se proći kroz skup rezultata. Da bi se dobili podaci o statistici, potrebno je izvršiti još jednu SQL naredbu za polje `player_id`. Grupa međusobno povezanih objekata naziva se *graf objekta*.

1.3.4 Problemi na koje se nailazi kada se koristi čisti JDBC

Korišćenje JDBC API-ja znači mogućnost izvršavanja bilo koje SQL naredbe. Za jednostavan zadatak, moguće je višestruko ponavljanje mnogih naredbi pretrage, umetanja, ažuriranja i brisanja unutar koda. A to može rezultovati dodatnim problemima:

- Previše dupliranog koda: kada se vrši uzimanje objekata, potrebno je kopirati polja unutar `ResultSet`-a u instancne promenljive objekta. Kada se vrši trajno smeštanje objekta, potrebno je kopirati instancne promenljive objekta u parametre unutar `PreparedStatement`-a.
- Ručno kontrolisanje veza: prilikom uzimanja objekata, potrebno je vršiti spajanja nad tabelama ili čitati iz nekoliko tabela, kako bi se dobio graf objekta. Kada se vrši trajno smeštanje objekta, potrebno je ažurirati nekoliko odgovarajućih tabela.
- Zavisnost od baze podataka: SQL naredbe, koje se pišu za jednu bazu, ne moraju funkcionisati unutar neke druge baze.

1.4 Objektno/relaciono preslikavanje

1.4.1 Šta je objektno/relaciono preslikavanje?

Objektno/relaciono preslikavanje je automatizovan postupak smeštanja objekata Java aplikacije u tabele relacione baze podataka, korišćenjem metapodataka kojim se opisuje preslikavanje između objekata i baze.
ORM funkcioniše tako što transformiše podatke iz jedne reprezentacije u drugu. Sam postupak nagoveštava smanjenje performansi prilikom izvršavanja.[1] Ukoliko je ORM implementirano kao posredni softver, javljaju se mnoge prilike za optimizaciju. Upravljanje metapodacima koji kontrolišu transformaciju uzrokuje nagomilavanje u memoriji (engl. *overhead*). Međutim, i pored toga ORM rešenje je efikasnije od namenski kodiranog rešenja.
ORM alat se sastoji iz četiri celine:

- API-ja za izvršavanje osnovnih CRUD (*create*, *read*, *update*, *delete*) operacija na objektima trajnih klasa,
- jezika ili API-ja za specifikovanje upita koji se odnose na klase i atribute klasa,
- svojstva za specifikaciju metapodataka koji se odnose na preslikavanje i
- tehnike za implementaciju ORM-a kako bi se izvršila komunikacija sa transakcionim objektima i na taj način došlo do optimizacije funkcija koje uključuju proveru nepostojećih podataka i lenjo uzimanje asocijacija.

Postoje različiti načini na koje ORM može biti implementirano. Mark Fasel je definisao četiri nivoa kvaliteta ORM-a.[1] To su:

- Čitava aplikacija, uključujući i korisnički interfejs, je dizajnirana u odnosu na relacioni model i relacione operacije, čija je osnova u SQL-u. Ovaj pristup, uprkos svojim nedostacima za sisteme velikog obima, predstavlja odlično rešenje za jednostavne aplikacije, gde je nivo ponovne upotrebe koda nizak. Aplikacije iz ove kategorije često imaju potrebu za velikom upotrebom pohranjenih procedura.
- Entiteti se predstavljaju preko klasa koje se ručno preslikavaju u relacione tabele. Ručno kodirani SQL/JDBC je sakriven od poslovne logike korišćenjem poznatih šablona dizajna. Ovaj pristup je izuzetno široko rasprostranjen i uspešan je kod aplikacija sa malim brojem entiteta, ili kod aplikacija sa opštim i modelima podataka koji se zasnivaju na metapodacima.[1] Pohranjene procedure se mogu sresti u ovoj vrsti aplikacija.
- Dizajn aplikacije se zasniva na objektnom modelu. SQL se generiše tokom kompajliranja korišćenjem alata za generisanje koda, ili tokom izvršavanja od strane okvira koji se koristi za trajnost podataka. Mehanizam trajnosti podataka koji se primenjuje podržava veze među objektima, a upiti mogu biti specificirani korišćenjem objektno-orijentisanog jezika. Ovakvo preslikavanje je pogodno za aplikacije srednje veličine sa nekim kompleksnijim transakcijama, naročito kada je od značaja portabilnost između različitih implementacija baza podataka.
- Puno objektno preslikavanje podržava sofisticirano modeliranje objekata: kompoziciju, nasleđivanje i polimorfizam. Sloj preslikavanja implementira transparentni vid trajnosti podataka: trajne klase ne nasleđuju niti jednu specijalnu baznu klasu, niti implementiraju neki specijalni interfejs. Ovaj nivo funkcionalnosti teško može biti realizovan slojem trajnosti podataka koji je razvijao programer specijalno za potrebe aplikacije – zato što su u njega uložene godine rada i razvoja. Određeni broj komercijalnih i otvoreni kod Java ORM alata su dostigli ovaj nivo kvaliteta.[1]

1.4.2 Opšti ORM problemi

Sledeća lista predstavlja skup osnovnih problema koji se rešavaju upotrebom alata koji implementira puno objektno preslikavanje u Java okruženju.

- Kako izgledaju trajne klase? Koliko je transparentan alat za trajnost podataka? Da li je potrebno usvojiti određeni model programiranja i pridržavati se određenih konvencija za klase koje implementiraju poslovni domen?
- Kako je definisano preslikavanje metapodataka? Budući da je objektno/relaciona transformacija u potpunosti određena metapodacima, format i definsanje ovih metapodataka je od naročitog značaja. Da li ORM alat treba da obezbedi GUI (Graphical User Interface) interfejs preko koga bi se upravljalo tim metapodacima, ili postoje bolji pristupi definisanju metapodataka?
- Kako se identičnost i ekvivalencija među objekatima dovode u vezu sa primarnim ključem u bazi? Kako se preslikavaju instance klasa u njima odgovarajuće vrste u tabeli?
- Kako bi trebalo pristupiti preslikavanju nasleđivanja među klasama? Šta je sa polimorfizmom, apstraktnim klasama, i interfejsima?
- Na koji način logika trajnosti podataka vrši interakciju sa objektima poslovnog domena tokom izvršavanja? Ovo je problem generičkog programiranja, i postoje brojna rešenja koja uključuju generisanje izvornog koda, refleksiju tokom izvršavanja, generisanje bajtkoda tokom izvršavanja, kao i proširivanje bajtkoda tokom kompajliranja.[1]
- Koji je životni ciklus trajnih objekata? Da li je životni ciklus nekih objekata vezan za životni ciklus drugih? Kako prevesti životni ciklus objekta u životni ciklus odgovarajuće vrste u tabeli?
- Na koji način su omogućeni sortiranje, pretraga i agregacija?
- Kako efikasno uzeti podatke koji su povezani sa drugim podacima? Efikasan pristup relacionoj bazi se obično ostvaruje kroz operacije spajanja tabela. Objektno-orijentisane aplikacije pristupaju podacima kretanjem kroz mrežu objekata.[1]

Dolazi se do zaključka da alat koji implementira puno objektno preslikavanje mora ispuniti poprilično dugačku listu zahteva. ORM je napredna tehnika, namenjena programerima koji su pristupali problemu na teži način. Preduslov za efektivno korišćenje okvira Hibernate je poznavanje SQL jezika i njegovih naredbi.

Neka od svojstava koja se dobijaju korišćenjem okvira Hibernate su:

- Produktivnost – Kod kojim se ostvaruje trajnost podataka može biti najmanje elegantan kod u jednoj Java aplikaciji. Hibernate eliminiše veliki deo opšteg posla (čak i više nego što korisnik očekuje), i dopušta programeru da se posveti poslovnim aspektima problema.[1]
- Održivost – Manji broj linija koda utiče na to da sistem bude razumljiviji, zato što je naglasak na poslovnoj logici problema. Sistem sa manje koda je podložniji refaktorizaciji. Automatizovana objektno/relaciona trajnost podataka značajno smanjuje broj linija koda. Postoje i drugi razlozi zbog kojih je Hibernate aplikacija lakša za održavanje od aplikacija koje ga ne koriste. U sistemima u kojima je trajnost podataka kodirao programer, prisutno je neslaganje između relacione reprezentacije i objektnog modela koji implementira domen. Promene na jednom kraju uvek sa sobom povlače i promene na drugom kraju, pa je dizajn jedne reprezentacije ugrožen, kako bi se omogućilo bolje predstavljanje druge reprezentacije. (Ono što se u praksi gotovo uvek dešava je da je objektni model domena ugrožen.[1])
- Performanse – Neke optimizacije postupka trajnosti podataka je lakše realizovati primenom ručno kodiranih SQL/JDBC rešenja. Sa druge strane, mnoge optimizacije je znatno lakše realizovati primenom automatizovanog ORM alata. U projektima koji su vremenski ograničeni, ručno kodirana rešenja obično dozvoljavaju optimizacije koje su manjeg obima. Hibernate omogućava da se znatno više optimizacija iskoristi, nevezano za konkretan

problem. Programeri koji su implementirali ORM sistem, koji je izabran za rešavanje određenog problema, su imali mnogo više vremena za ispitivanje i istraživanje optimizacija koje se tiču performansi, nego što jedan programer koji razvija aplikaciju ima. Razvoj pristojnog ORM rešenja sa punim dijapazonom opcija može oduzeti nekoliko meseci rada. Hibernate čini oko 80000 linija koda, od kojih su neke mnogo ozbiljnije nego tipičan kod regularne aplikacije. Pored toga, treba uračunati i dodatnih 25000 linija test koda.[1]

- Nezavisnost od distributera – ORM apstrahuje aplikaciju od SQL baze i SQL jezika. Ukoliko alat ima podršku za veliki broj različitih baza (a većina alata ima), ovo pruža izvestan nivo portabilnosti aplikaciji koja se razvija. Jednostavnije je razvijati aplikaciju za više različitih platformi korišćenjem ORM alata. Nezavisnost od baze podataka omogućava programerima da u toku razvoja koriste jednostavniju lokalnu bazu, dok razvijaju aplikaciju koja će raditi sa drugom bazom podataka.

1.5 Razna ORM rešenja

Poput Spring-a (o kome će biti više reči u poglavlju *3. Spring*), i Hibernate je promenio razvoj softvera kada se pojavio na sceni. Tajming je bio idealan. Programeri su bili frustrirani preteranom kompleksnošću J2EE i nagomilavanjem u memoriji koji su pratili upotrebu EJB-a za perzistenciju.[7]
Hibernate se oslanja na korišćenje POJO-a (engl. *Plain Old Java Objects*). Ostali ORM okviri primoravaju programere da se pridržavaju određenih pravila tokom razvijanja domen modela, u šta spadaju alternativne i roditeljske klase, kao i objekti za prenos podataka (engl. *Data Transfer Objects*, *DTO*).
Sa filozofijom kod koje je naglasak bio na jednostavnosti i čistom kodu, Spring i Hibernate najavili su novo doba za poslovne Java aplikacije i trajno smeštanje podataka. Ovaj uzajamni uspeh je imao dramatičan uticaj na Java zajednicu, i predstavljao je katalizator za brojne promene, koja je prihvatila pristup razvoju aplikacija čiji cilj je jednostavnost.
Često postoji velika zabuna oko različitih opcija za trajno smeštanje koje postoje u Javi. U čemu je razlika između EJB 3, JPA, JDO i Hibernate? Odgovor na to pitanje se može naći u sledećem poglavlju.

1.5.1 Evolucija trajnosti podataka u Javi

JDBC je uključen od strane Sun Microsystem-a kao deo JDK 1.1 1997. godine. JDBC je API niskog nivoa orijentisan oko relacionih baza podataka. Omogućava metode za izvršavanje upita i ažuriranje podataka unutar baze. JDBC pruža veliki skup alata, ali najveći deo posla je na samom programeru, koji mora da piše SQL, da preslikava rezultate upita na objekte domena, da vodi računa o konekcijama i transakcijama itd. Većina drugih okvira za trajnost podataka je izgrađena kao apstrakcija iznad JDBC-a, kako bi smanjili količinu posla koji mora da obavi sam programer.
Počevši od 2000. godine, verzija 3.0 JDBC specifikacije je činila deo Java Community Process-a (JCP). JCP je nastao 1998. godine kao ideja koja bi okupila programere zainteresovane za određivanje ograničenja kojih bi se pridržavao budući razvoj Java platforme. U samom srcu JCP su Java Specification Requests (JSRs), koji su formalni dokumenti, i koji opisuju predložene dodatke ili promene na Java platformi. Svaki JSR ima jednu ili više osoba, koja predstavlja začetnika specifikacije (dok se tim članova naziva grupa eksperata), koji međusobno sarađuju kako bi njihov rad rezultirao krajnjim oblikom specifikacije. Konačni JSR takođe uključuje i referencu na odgovarajuću implementaciju.
Ovo razlikovanje specifikacije i implementacije je jedan od osnovnih izvora zabune među programerima, kada se vode polemike o različitim rešenjima koja se odnose na trajno smeštanje.[7]
Na primer, JPA je specifikacija, a Hibernate je samo jedan od mnogih projekata koji predstavljaju

implementaciju JPA specifikacije. Ostale implementacije JPA specifikacije uključuju OpenJPA, DataNucleus, i referencnu implementaciju, EclipseLink.

1.5.2 EJB, JDO, i JPA

Krajem devedesetih i tokom ranih dvehiljaditih, vodeća tehnologija za razvoj aplikacija velikog obima u Javi bio je EJB. Originalno začete od strane IBM-a 1997., EJB 1.0 i 1.1 specifikacije je usvojio Sun 1999. godine. Počevši od tada, EJB je proširivan kroz JCP. JSR 19 je poslužio kao osnova za EJB 2.0, koji je završen 2001. godine. EJB 2.0 specifikacija je postala najznačajnija komponenta u Sun Java 2 platformi, Enterprise Edition-u (J2EE), koji predstavlja referencnu implementaciju.
Problemi koje je EJB trebao da reši su kompleksni; među njih spada i ORM. EJB 1.0, 1.1, i 2.0 specifikacije su bile nepotrebno zakomplikovane proverenim izuzecima, interfejsima koji su bili obavezni, kao i velikom upotrebom apstraktnih klasa.[7]
Prvi zvanični pokušaj da se kreira jednostavan sloj apstrakcije, koji za osnovu ima JDBC, od strane JCP-a bio je JSR 12: Java Data Objects (JDO). Grupa eksperata koja je radila na JDO je 1999. postavila cilj da definiše standardni način za trajno smeštanje Java objekata u baze podataka. Kao dodatak tome, JDO je definisao sredstva za prevođenje podataka iz relacione baze u Java objekte. Dok je specifikacija završena 2002. godine, JDO je evoluirao u POJO zasnovan API koji je bio nezavisan od distribucije baze podataka. To je značilo da se JDO mogao koristiti sa raznim skladištima podataka, počevši od sistema za upravljanje relacionim bazama podataka do sistema datoteka (engl. *file system*), ili čak i sa objektno-orijentisanom bazom podataka. Interesantno je da glavni distributeri servera za aplikacije nisu prihvatili JDO 1.0, tako da nikada nije ostvario planirani potencijal.[7]
Između JDO i EJB se odvijala neka vrsta takmičenja dva standarda za kontrolu trajnog smeštanja, pri čemu nijedan od njih nije uspeo da ostvari prodor i uspeh, i da se nametne kao rešenje koje bi programeri prihvatili. Upravo to je ostavilo otvorena vrata za komercijalne igrače i otvoreni kod okvire. Hibernate je ORM okvir za koji ljudi obično misle da je zamena za EJB 2.0, ali drugo rešenje je zapravo bio začetnik svega.
ORM alat pod nazivom TopLink je originalno razvijao Top Object People za programski jezik Smalltalk. Portovan je u Javu i dodat u proizvodnu liniju kompanije 1998. Oracle je preuzeo TopLink 2002. godine. TopLink je predstavljao impresivan okvir, i njegove pogodnosti su igrale bitnu ulogu u oblikovanju specifikacija koje se odnose na trajnost podataka i koje su počele da se pojavljuju od tog trenutka unutar Java sveta. Kao komercijalni projekat, TopLink nikada nije dostigao nivo recepcije koji je dostigao EJB.[7] EJB je imao snažnu podršku među distributerima servera za aplikacije, kao što su bili IBM i BEA. A sa druge strane, nije bio u mogućnosti da se stvarno uključi u trku sa jednostavnim otvoreni kod okvirima koji su se pojvaljivali, među kojima je bio i Hibernate.
Gavin King je 2001. godine počeo sa radom na Hibernate projektu, kako bi pronašao rešenje za probleme koji su pratili upotrebu EJB 2 *entity beans*-a. Shvatio je da je provodio više vremena vodeći računa o trajnom smeštanju podataka, nego o problemima vezanim za poslovni model svojih klijenata. Namera okvira Hibernate bila je da se poveća produktivnost i da omogući programerima da se više posvete objektnom modeliranju, kao i da pojednostavi implementaciju logike trajnosti podataka. Hibernate 1.0 se pojavio 2002., Hibernate 2.0 2003., a Hibernate 3.0 2005. godine. Tokom tog perioda, Hibernate je stekao veliku prednost samim tim što je bio besplatni, POJO zasnovani ORM, koji je ispraćen dobrom dokumentacijom, a bio je i veoma pristupačan za programere. Hibernate je bio u mogućnosti da pruži sredstva za razvoj poslovnih aplikacija koje su bile praktične, jednostavne, elegantne, i najbitnije od svega, otvoreni kod.
Tokom prvih pet godina postojanja, Hibernate se nije formalno pridržavao niti jedne specifikacije, i nije bio deo niti jednog procesa standardizacije. Hibernate je bio samo otvoreni kod projekat koji se mogao koristiti za rešavanje problema i završavanje poslova. Tokom ovog perioda, mnogi programeri, uključujući tu i samog Gavin Kinga, su zajedničkim snagama otpočeli rad na JSR 220: Enterprise JavaBeans 3.0. Za cilj su imali kreiranje novog standarda koji bi ispravio bolne tačke koje

su dovođene u vezu sa EJB. Dok je grupa eksperata radila na EJB 3.0 specifikaciji, odlučeno je da će komponenta trajnosti podataka za interakciju sa sistemom za upravljanje relacionom bazom podataka biti odvojena u poseban API. Upravo to predstavlja nastanak Java Persistence API-ja (JPA). Zasnovan je na mnogim od ključnih principa koji su već bili implementirani i dokazani na polju ORM-a, a to su naravno TopLink i Hibernate. Kao deo JSR 220 grupe eksperata, Oracle je omogućio referencnu implementaciju JPA 1.0 svojim TopLink proizvodom. JSR 220 je završen 2006. godine i EJB 3 je odigrao ključnu ulogu u Sun-ovoj definiciji Java Enterprise Edition 5, ili JEE 5.
JPA 1.0 predstavlja ključnu prekretnicu za trajnost podataka u Javi. Međutim, mnoga svojstva koja su bila od posebnog značaja za programere koji su već koristili alate kao što je Hibernate nisu uključena u JPA zbog nedostatka vremena. JPA 2.0 je dodao mnogo novih svojstava, uključujući Criteria API, API za rad sa kešom, i proširenja za Java Persistence Query Language (JPQL), JPA-ov objektno-orijentisani upitni jezik. JPA 2.0 standard je završen u decembru 2009. godine kao novi, samostalni JSR koji je imao za cilj da bude uključen u Java EE 6 specifikaciju. Oracle je ustupio izvorni kod i razvojne resurse za TopLink Sun-u kako bi kreirali EclipseLink projekat. EclipseLink je zatim postao referencna implementacija za JPA 2.0. Hibernate 3.5 se pojavio u proleće 2010. sa punom podrškom za JSR 317: JPA 2.0. Hibernate 4 se pojavio u decembru 2011., a tokom 2012. je započet rad na razvoju Hibernate 5, koji će podržavati JPA 2.1.

1.5.3 Kako se Hibernate uklapa?

Potrebno je naglasiti razliku između Hibernate-a i EJB-a. EJB je specifikacija za pružanje distribuirane arhitekture za serversku stranu aplikacije kojom se upravlja korišćenjem kontejnera. EJB 3 enkapsulira nekoliko različitih specifikacija za olakšavanje mnogih stvari, uključujući upravljanje distribuiranim transakcijama, kontrolu konkurentnosti, slanje poruka, web servise, i npr. bezbednost. EJB 3 specifikacija pretpostavlja da upravljanje trajnosti podataka može biti preneto na JPA implementaciju.
Druga stvar, Hibernate nije JPA. Umesto toga, Hibernate je jedan od mnogih okvira koji pružaju standardizovanu implementaciju za JPA. Prva Hibernate verzija koja je podržavala JPA 1.0 bio je Hibernate 3.2, koji je postao dostupan u jesen 2006.
Često postoje specijalizovane pogodnosti koje pružaju okviri kao što je Hibernate, koji prevazilaze JPA specifikaciju. Kao takav, Hibernate se može shvatiti kao nadskup JPA. Sa jedne strane, Hibernate arhitektura omogućava upotrebu Hibernate jezgra bez korišćenja bilo kog dela JPA specifikacije uopšte. Sa druge, mogu se koristiti samo oni delovi okvira Hibernate koji ulaze u JPA specifikaciju. Striktno pridržavanje JPA specifikacije osigurava istinsku portabilnost na ostale JPA implementacije, kao što je Apache OpenJPA projekat.
Prilikom korišćenja okvira Hibernate, preporučljivo je da se programeri pridržavaju JPA specifikacije što je više moguće, ali naravno bez zalaženja u ekstreme. Budući da otvoreni kod projekti imaju tendenciju da evoluiraju znatno brže nego što je to slučaj sa JCP, okviri kao što je Hibernate će ponuditi rešenja za probleme, a koja nisu obuhvaćena procesom standardizacije. Ukoliko ova namenska rešenja olakšavaju posao programeru, cilj je iskoristiti ih i biti pragmatičan. Ovo je deo lepote procesa standardizacije – implementatori različitih specifikacija su slobodni da vrše inovacije, i najbolje, najuspešnije ideje će biti uključene u buduće revizije specifikacije.
JPA sepcifikacija definiše skup anotacija koje se mogu primeniti na klase domena, kako bi se izvršilo preslikavanje objekata u tabele baze, kao i instancnih promenljivih u kolone baze podataka.

2. Hibernate

Hibernate je potpuni alat za objektno/relaciono preslikavanje koji pruža sve opisane pogodnosti koje se i očekuju od jednog ORM alata. API sa kojim se radi u Hibernate-u je razvijan od strane Hibernate programera. Isto važi i za interfejse upita i upitne jezike, i za način na koji se definišu metapodaci za objektno/relaciono preslikavanje. Hibernate jezgro je poznato pod nazivom Hibernate. Predstavlja osnovni servis za trajnost podataka, sa nativnim API-jem. Metapodaci koji se odnose na preslikavanje smeštaju se unutar XML datoteka. Poseduje upitni jezik HQL (u velikoj meri sličan SQL-u), kao i programabilne interfejse upita za Criteria i Example upite. Postoji na stotine opcija i mogućnosti koje pokrivaju skoro sve, što Hibernate jezgro zaista čini temeljom i platformom na kojoj su svi ostali modeli zasnovani.[1]

2.1 Razlikovanje entiteta i vrednosnih tipova

Entiteti su trajni tipovi podataka koji predstavljaju objekte prvog nivoa, tj. objekte najvećeg značaja (termin objekat se ovde koristi u svom osnovnom značenju). Drugim rečima, neke od klasa i tipova koje se nalaze u aplikaciji su od većeg značaja, što implicira da su ostale manje važne.
Hibernate naglašava korisnost sofisticiranih klasa prilikom implementacije bezbednosti tipova i ponašanja. Na primer, veliki broj programera će implementirati e-mail adresu kao instancnu promenljivu tipa `String` unutar klase `Korisnik`. Sofisticiraniji pristup predstavlja definisanje klase `EmailAdresa`, koja omogućava semantiku višeg nivoa, kao i ponašanje – mogla bi da sadrži metodu `posaljiEMail()`.
Ukoliko objekat treba da podrži deljene reference tokom izvršavanja, on je tipa entitet. Ukoliko to nije slučaj, verovatnije je da je vrednosni tip; samim tim zavisi od reference koju na njega ima objekat koji je entitet.
Hibernate pravi sledeću razliku među objektima:

- Objekat je entitet ukoliko ima svoj identitet u bazi podataka (što se predstavlja kroz vrednost primarnog ključa). Referenca nekog objekta na instancu objekta koji je entitet se unutar baze preslikava takođe kao referenca (vrednost stranog ključa). Entitet ima sopstveni životni ciklus; može postojati nezavisno od bilo kog drugog entiteta.
- Objekat je vrednosnog tipa ukoliko nema svoj identitet u bazi podataka; pripada instanci entiteta i njegovo trajno stanje je ugrađeno (engl. *embedded*) u vrstu tabele odgovarajućeg entiteta, tj. entiteta kome pripada. Vrednosni tipovi nemaju ni identifikatore, ni njihova svojstva. Životni ciklus instance vrednosnog tipa je vezan za životni ciklus odgovarajućeg entiteta. Vrednosni tip ne podržava deljene reference. Najočigledniji primeri za vrednosne tipove su klase poput `String` ili `Integer`, premda se sve JDK (Java Development Kit) klase smatraju vrednosnim tipovima.[1]

Prilikom implementacije POJO-a za entitete, kao i vrednosne tipove, potrebno je voditi računa o sledećim stvarima:

- *Deljene reference* – potrebno je implementirati POJO klase na način koji bi omogućio da se izbegnu deljene reference na instance objekata koji su vrednosni tipovi.
- *Zavisnosti životnih ciklusa* – kao što je već spomenuto, životni ciklus instance vrednosnog tipa je vezan za životni ciklus instance odgovarajućeg entiteta. Ukoliko se objekat koji predstavlja entitet obriše, potrebno je obrisati i sve vrednosne tipove koji od njega zavise. Ne postoji odgovarajuća ključna reč u Javi kojom bi se moglo predstaviti ovakvo ponašanje objekata, ali sama aplikacija i korisnički interfejs moraju biti dizajnirani tako da reflektuju ovo ponašanje.
- *Identitet* – klase koje predstavljaju entitete moraju imati instancnu promenljivu koja će predstvaljati identifikator (u skoro svim slučajevima). Klase vrednosnih tipova koje definiše

sam korisnik (kao i JDK klase) nemaju instancnu promenljivu koja bi predstavljala identifikator, zato što se instance ovih klasa identifikuju kroz odgovarajuće entitete (entitete za koje su vezani, kojima pripadaju).

Instancna promenljiva koja predstavlja identifikator unutar klase je od posebnog značaja – njena vrednost je vrednost primarnog ključa u vrsti tabele kojim je predstavljen objekat te klase. Uobičajena praksa je izložiti identifikator u bazi kroz *getter* metode koje imaju `public` atribut pristupa.

```
@Id
private Long id;

public Long getId() {
	return id;
}

private void setId(Long id) {
	this.id = id;
}
```

Sa druge strane, *setter* metode za tu instancnu promenljivu se deklarišu sa `private` atributom pristupa, i sva odgovornost za generisanje i postavljanje vrednosti identifikatora je na Hibernate-u. Drugu mogućnost predstavlja preslikavanje kroz direktan pristup promenljivoj, pri čemu se implementira samo *getter* metoda. (Izuzetak od ovog pravila predstavljaju klase sa prirodnim ključem, gde se vrednost primarnog ključa postavlja od strane same aplikacije, pre nego što se izvrši trajno smeštanje objekta, ili se generiše od strane Hibernate-a.)
Anotacije koje se koriste unutar Java koda za preslikavanje identifikatora u klasi su:

```
@Entity
public class Igrac {

	private Long id;

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	@Column(name="IGRAC_ID")
	public Long getId() {
		return id;
	}

	private void setId(Long id) {
		this.id = id;
	}
```

`@Id` anotacija iznad *getter* metode označava odgovarajuću instancnu promenljivu kao identifikator, a `@GeneratedValue` sa `GenerationType.AUTO` opcijom se predstavlja nativna strategija za generisanje identifikatora. U slučaju da se ne definiše strategija, podrazumeva se `GenerationType.AUTO`, tako da je moguće izostaviti ovaj atribut. Takođe se može navesti ime kolone – u suprotnom će Hibernate iskoristiti ime same promenljive. Tip za preslikavanje identifikatora implicira tip Java promenljive – u ovom konkretnom slučaju, `java.lang.Long`.
Naravno, moguće je koristiti direktan pristup, preko samih instancnih promenljivih, uključujući i anotaciju identifikatora u bazi:

```
@Entity
public class Igrac {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
```

```
	@Column(name="IGRAC_ID")
	private Long id;

	public Long getId() {
		return id;
	}
```

Vrednosni tipovi se mogu javiti u nekoliko oblika. Izdvajaju se dva tipa: klase vrednosnih tipova koje se nalaze unutar JDK-a, kao što su `String` i primitivni tipovi, i klase vrednosnih tipova koje definiše programer.
Ukoliko se trajna klasa preslikava pomoću anotacija, podrazumeva se da se sve njene instancne promenljive postaju trajne. Moguće je označiti instancnu promenljivu sa `@javax.persistence.Transient` kako bi se sprečilo preslikavanje tog atributa, ili upotrebiti `transient` ključnu reč Java jezika. Ukoliko instancna promenljiva trajne klase nema anotaciju, primenjuju se sledeća pravila:

- ukoliko je instancna promenljiva tip iz JDK, preslikavanje se odvija automatski
- inače, ukoliko je klasa kojoj ta instancna promenljiva pripada označena sa `@Embeddable`, preslikava se kao komponenta odgovarajućeg entiteta.

Hibernate koristi termin komponenta za klasu koju je definisao korisnik (programer), a koja se preslikava unutar iste tabele kao i entitet kome pripada. Java Persistence specifikacija umesto termina komponente koristi ugrađene klase. Da bi se preslikala ugrađena klasa korišćenjem anotacija, potrebno je deklarisati instancnu promenljivu unutar entiteta kojem pripada vrednosni tip sa `@Embedded`:

```
@Entity
public class Igrac {
	...
	@Embedded
	private Statistika statistike;
```

U slučaju da se ne deklariše sa `@Embedded`, i takođe u slučaju da nije JDK tip, Hibernate će potražiti u odgovarajućoj klasi anotaciju `@Embeddable`. Ukoliko se ona nalazi tamo, svojstvo je automatski preslikano kao zavisna komponenta.

```
@Embeddable
public class Statistika {
	...
```

Postoje dva važna ograničenja u slučaju klasa koje se preslikavaju kao komponente. Prvo, deljene reference, kao i za sve vrednosne tipove, nisu moguće. Komponenta `statistike` nema svoj identitet u bazi (nema primarni ključ), pa prema tome ne može imati referencu od bilo kog objekta, osim onoga unutar kojeg je definisana.
Drugo, ne postoji elegantan način za predstavljanje `null` reference na objekat klase `Statistika`. Za razliku od bilo kog elegantnog pristupa, Hibernate predstavlja `null` komponentu kao `null` vrednosti svih kolona u koje se preslikavaju te komponente. Ovo znači da ako se trajno smešta komponentni objekat koji ima sve instancne promenljive sa vrednošću `null`, Hibernate vraća `null` komponentu kada se odgovarajući entitet uzme iz baze.

2.2 Nasleđivanje i proizvoljni tipovi

Preslikavanje nasleđivanja, tj. preslikavanje hijererhije klasa u tabele baze podataka predstavlja kompleksni zadatak. Za rešavanje tog problema se primenjuje nekoliko strategija.

Jednostavno rešenje može biti "jedna tabela za svaku od trajnih klasa". Ovaj pristup deluje jednostavno i funkcioniše prilično dobro, dok se ne naiđe na nasleđivanje.
Nasleđivanje je očigledno strukturno neslaganje između objektno-orijentisane i relacione paradigme zato što objektno-orijentisani sistemi modeliraju i *jeste* (engl. *is a*) i *ima* (engl. *has a*) veze između klasa u Javi. SQL zasnovani modeli pružaju samo *ima* veze među entitetima; sistemi za upravljanje bazama podataka ne podržavaju nasleđivanje tipova.[1]
Postoje četiri različita pristupa koji predstavljaju rešenja problema preslikavanja hijerarhije nasleđivanja:

- Tabela po konkretnoj klasi sa implicitnim polimorfizmom – ne koristi se eksplicitno preslikavanje nasleđivanja, ali se podrazumeva polimorfno ponašanje tokom izvršavanja.
- Tabela po konkretnoj klasi – iz SQL sheme kompletno odbaciti polimorfizam i veze dobijene nasleđivanjem.
- Tabela po hijerarhiji klasa – omogućiti polimorfizam denormalizacijom SQL sheme, i iskoristiti kolonu koja predstavlja diskriminator tipa i sadrži informacije o tipovima.
- Tabela po izvedenoj klasi – predstaviti *jeste* veze među klasama tj. nasleđivanje, kao *ima* (strani ključ) veze.

Uz pomoć nekih opcija koje se nalaze unutar okvira Hibernate, moguće je promeniti strategiju preslikavanja za određenu izvedenu klasu. Na primer, moguće je preslikati hijerarhiju klasa unutar jedne tabele, ali za određenu izvedenu klasu, preći na posebnu tabelu sa strategijom preslikavanja koja koristi strani ključ, kao što bi mogla biti strategija tabela po izvedenoj klasi.

2.2.1 Tabela po konkretnoj klasi sa implicitnim polimorfizmom

Predstavlja najjednostavniji način za rešavanje problema. Koristi se tačno jedna tabela za svaku (neapstraktnu) klasu. Sve instancne promenljive klase, uključujući i one koje se nasleđuju, mogu biti preslikane u kolone ove tabele. Nije potrebno ništa dodatno naglašavati u Hibernate-u kako bi se omogućilo polimorfno ponašanje. Hibernate je "svestan" natklase (ili bilo kog interfejsa), zato što skenira trajne klase prilikom pokretanja.
Glavni problem sa ovim pristupom je taj što ne pruža naročito dobru podršku za polimorfne asocijacije. U bazi podataka, asocijacije se obično predstavljaju kao veze preko stranog ključa. Ukoliko su sve izvedene klase mapirane u različite tabele, polimorfna asocijacija ka njihovim natklasama ne može biti predstavljena kao jednostavna veza preko stranog ključa.
Sledeći konceptualni problem sa ovom strategijom mapiranja je taj što nekoliko različitih kolona, ili nekoliko različitih tabela, dele potpuno istu semantiku. Ovim se znatno komplikuje evolucija sheme. Na primer, promena neke instancne promenljive u natklasi uzrokuje promene na mnogim kolonama. Takođe je znatno teže implementirati ograničenja integriteta baze podataka, koja bi se mogla primeniti na sve izvedene potklase.
Ukoliko programer odluči da se osloni na ovaj implicitni polimorfizam, preslikavanje konkretnih klasa se kao i obično vrši sa `@Entity`. Međutim, takođe je potrebno duplirati instancne promenljive natklase kako bi se izvršilo njihovo preslikavanje u sve tabele konkretnih klasa. Podrazumeva se da se instancne promenljive natklasa ignorišu i ne preslikavaju. Potrebno je izvršiti anotaciju natklase kako bi se omogućilo ugrađivanje njenih instancnih promenljivih u tabele određene potklase (izvedene klase). U ovu svrhu se koristi anotacija `@MappedSuperclass`.

2.2.2 Tabela po konkretnoj klasi

Razlikuju se dva slučaja. Prvi slučaj je kada je natklasa apstraktna (ili je u pitanju interfejs), a drugi slučaj je kada je natklasa neka konkretna klasa.

U JPA anotacijama, ova strategija je poznata kao TABLE_PER_CLASS.

```
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public class Osoba {
	...
}
```

Identifikator u bazi podataka, kao i njegovo preslikavanje se moraju nalaziti u natklasi; to je neophodno kako bi bilo moguće deljenje tog identifikatora kroz sve klase izvedene iz te natklase i u tabelama koje im odgovaraju. Sve što je potrebno uraditi je @Entity anotacija na svakoj od izvedenih klasa:

```
@Entity
public class Igrac extends Osoba {
	...
}
```

TABLE_PER_CLASS ima oznaku u JPA standardu kao opcioni atribut, pa ga ne moraju podržavati sve JPA implementacije. U slučaju da je natklasa konkretna klasa, potrebna je dodatna tabela u kojoj će se nalaziti instance te klase.

2.2.3 Tabela po hijerahiji klasa

Čitava hijerarhija klasa može biti preslikana unutar jedne jedine tabele. Ova tabela uključuje kolone za sve instancne promenljive svih klasa koje se nalaze u hijerarhiji. Konkretna izvedena klasa (tj. njen tip), koja je predstavljena određenom vrstom u velikoj tabeli se identifikuje pomoću vrednosti koja se nalazi u koloni čiji je tip diskriminator.
Međutim, postoji jedan veliki problem: kolone kojima se preslikavaju instancne promenljive koje su deklarisane unutar izvedenih klasa, moraju omogućiti postavljanje nedostajućih null vrednosti. Drugi bitan problem predstavlja normalizacija. Kreirane su funkcionalne zavisnosti između kolona koje nisu ključevi, pri čemu se narušava treća normalna forma. Denormalizacija zbog performansi može izvesti na pogrešan put, zato što se na taj način žrtvuju dugotrajna stabilnost, održivost, kao i integritet podataka zbog trenutnih dobitaka.[1]
U JPA anotacijama, strategija je poznatija pod imenom SINGLE_TABLE:

```
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
public class Osoba {
	...
}
```

2.2.4 Tabela po izvedenoj klasi

Četvrta opcija je predstavljanje nasleđivanja kao relacione asocijacije preko stranog ključa. Svaka klasa/natklasa koja deklariše instancne promenljive koje treba preslikati – uključujući apstraktne klase, i čak i interfejse – ima svoju sopstvenu tabelu. Za razliku od strategije tabela po konkretnoj klasi, koja je prva prikazana, u ovoj strategiji tabela sadrži kolone samo za instancnu promenljivu koja nije nasleđena (tj. za svaku instancnu promenljivu deklarisanu u samoj izvedenoj klasi), zajedno sa primarnim ključem koji je istovremeno strani ključ za tabelu kojom je predstavljena natklasa. Osnovna prednost ove strategije je normalizovana SQL shema.
U JPA anotacijama, za ovu strategiju se koristi naziv JOINED.

```
@Inheritance (strategy=InheritanceType.JOINED)
```

```
public class Osoba {
	...
}
```

2.3 Hibernate sistem tipova

Entiteti su najznačajnije klase u sistemu koji se razvija. Obično se svojstva sistema definišu preko entiteta koji se nalaze u tom sistemu. Sa druge strane, klase kojima su predstavljeni vrednosni tipovi se obično ni ne nalaze u poslovnim zahtevima – one se smatraju deklaracijama sofisticiranih klasa. Moguće je da se vrednosni tipovi pojave u deklaracijama instancnih promenljivih.
Formalnije rečeno, entitet je bilo koja klasa čije instance imaju svoj sopstveni identitet prilikom preslikavanja. Vrednosni tip je klasa koja nema identitet prilikom preslikavanja. To u praksi znači da su entiteti klase sa instancnom promenljivom koja predstavlja identifikator, a vrednosni tipovi su klase koje zavise od entiteta.
Prilikom izvršavanja, u aplikaciji se nalazi mreža instanci entiteta upletenih sa instancama vrednosnih tipova. Instance entiteta se mogu nalaziti u bilo kom od tri trajna stanja: privremeni (engl. *transient*), nepovezani (engl. *detached*) ili trajni (engl. *persistent*) (o stanjima će biti više reči u posebnom odeljku). Ova trajna stanja nemaju nikakav značaj za instance vrednosnih tipova.
Prema tome, entiteti imaju sopstveni životni ciklus. `save()` i `delete()` metode Hibernate `Session` interfejsa se primenjuju na instance klasa kojima su predstavljeni entiteti, a nikada na instance klasa kojima su predstavljeni vrednosni tipovi. Preslikavanje životnog ciklusa instance klase koja je vrednosni tip je u potpunosti povezan sa životnim ciklusom odgovarajućeg entiteta, i koji sadrži u sebi taj vrednosni tip kao instancnu promenljivu. Na primer, `username` postaje trajan kada se sačuva `user`; nikada se ne preslikava nezavisno od `user`-a.
Posmatrano iz ugla baze podataka, bilo koja tabela se smatra entitetom. Međutim, Hibernate omogućava izvesne konstrukte kako bi enkapsulirao određene entitete, koji postoje na nivou baze podataka, od Java koda. Na primer, *many-to-many* asocijacija preslikavanja sakriva posredničku tabelu asocijacija od aplikacije. Kolekcija String objekata (ili preciznije, kolekcija instanci klasa koje su vrednosni tipovi) se ponaša kao vrednosni tip posmatrano iz ugla aplikacije; međutim, preslikava se u sopstvenu tabelu.
Hibernate sadrži veliki broj različitih tipova preslikavanja koji definišu podrazumevane strategije za trajnost primitivnih Java tipova i određenih JDK klasa. Ugrađeni tipovi koji se koriste prilikom preslikavanja obično imaju isto ime kao Java tipovi koje preslikavaju. Međutim, moguće je da postoji više različitih Hibernate ugrađenih tipova za jedan određeni Java tip.

2.3.1 Preslikavanje primitivnih Java tipova

Osnovni tipovi preslikavanja omogućavaju trajnost Java primitivnih tipova (ili njihovih klasa koje ih enkapsuliraju) u odgovarajuće ugrađene SQL standardne tipove.

Tip za preslikavanje	Java tip	Standardni SQL ugrađeni tip
integer	int ili java.lang.Integer	INTEGER
long	long ili java.lang.Long	BIGINT
short	short ili java.lang.Short	SMALLINT
float	float ili java.lang.Float	FLOAT
double	double ili java.lang.Double	DOUBLE
character	java.lang.String	CHAR (1)
string	java.lang.String	VARCHAR
byte	byte ili java.lang.Byte	TINYINT
boolean	boolean ili java.lang.Boolean	BIT

Tabela 1. – Preslikavanje primitivnih tipova

Navedena imena tipova su imena koja odgovaraju ANSI-standardu. Mnogi proizvođači baza podataka ignorišu ovaj deo SQL standarda. Međutim, JDBC drajver pruža delimičnu apstrakciju SQL tipova koji su specifični za određenog proizvođača, i na taj način dozvoljava Hibernate-u da radi sa ANSI standardnim tipovima prilikom izvršavanja DML-a (Data Manipulation Language).
Hibernate sadrži i tipove koji su povezani sa datumom, vremenom, kao i vremenskim pečatom. U domen modelu aplikacije koja se razvija, može se izabrati predstavljanje podataka koji se odnose na datum i vreme korišćenjem java.util.Date, java.util.Calendar, ili nekom od klasa koje su izvedene iz java.util.Date, a koje su definisane u java.sql paketu.

Tip za preslikavanje	Java tip	Standardni SQL ugrađeni tip
date	java.util.Date ili java.sql.Date	DATE
time	java.util.Date ili java.sql.Time	TIME
timestamp	java.util.Date ili java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

Tabela 2. – Preslikavanje klasa vezanih za datum i vreme

2.4 Preslikavanje kolekcija i veza među entitetima

Dve važne i zahtevne teme su preslikavanje kolekcija i preslikavanje veza između klasa koje predstavljaju entitete.

2.4.1 Kolekcije vrednosnih tipova

Objekat vrednosnog tipa se ne može identifikovati u bazi podataka; pripada objektu koji je instanca entiteta. Njegovo trajno stanje je ugrađeno u vrstu tabele kojom je predstavljen objekat koji je instanca entiteta kome instanca vrednosnog tipa pripada (tj. u slučaju kada entitet ima referencu na jednu instancu vrednosnog tipa). Ukoliko klasa, kojom je predstavljen entitet, ima kolekciju vrednosnih tipova (ili kolekciju referenci na instance klasa vrednosnih tipova), potrebna je dodatna tabela, tzv. kolekcijska tabela (engl. *collection table*). Java poseduje bogat API koji se odnosi na kolekcije, tako da programer ima mogućnost izbora interfejsa kolekcije i implementacije koji najviše odgovaraju dizajnu domen modela.

Šablon koji se primenjuje na instancnu promenljivu koja predstavlja kolekciju u Java domen modelu je uvek isti:

```
private <<Interfejs>> slike = new <<Implementacija>>();
```

Poželjna je upotreba interfejsa prilikom deklaracije tipa instancne promenljive, a ne upotreba implementacije. Potrebno je odabrati implementaciju koja odgovara interfejsu, i inicijalizovati kolekciju odmah prilikom deklaracije; na taj način se izbegavaju neinicijalizovane kolekcije.
Primer za generički `Set` sa parametrom tipa je sledeći:

```
private Set<String> slike = new HashSet<String>();
```

Hibernate podržava većinu JDK interfejsa za kolekcije. Svaki interfejs ima odgovarajuću implementaciju, koju Hibernate podržava, i važno je da se upotrebi odgovarajuća kombinacija.

2.4.2 Osnovno preslikavanje kolekcija

Na sledeći način se preslikava jednostavna kolekcija elemenata klase `String`:

```
@ElementCollection (targetClass=java.lang.String.class)
@JoinTable (name="SLIKA_IGRACA", joinColumns=@JoinColumn(name="IGRAC_ID"))
@Column (name="DATOTEKA", nullable=false)
private Set<String> slike = new HashSet<String>();
```

Kolekcijska tabela će imati dve kolone; zajedno formiraju kompozitni primarni ključ. Hibernate može automatski da detektuje tip elementa, u slučaju da se koriste generičke kolekcije. Ukoliko se u kodu ne koriste generičke kolekcije, potrebno je navesti tip elementa sa atributom `targetElement` – dakle, u prethodnom primeru je taj atribut opcioni.

Za preslikavanje Liste, potrebno je dodati `@org.hibernate.annotations.IndexColumn` anotaciju, sa opcionom vrednošću za početnu vrednost indeksa (podrazumevana vrednost je 0):

```
@ElementCollection
@JoinTable (name="SLIKA_IGRACA", joinColumns=@JoinColumn(name="IGRAC_ID"))
@Column (name="DATOTEKA")
@IndexColumn(name="POZICIJA", base=1)
private List<String> slike = new ArrayList<String>();
```

U slučaju da se izostavi indeks kolona, lista će biti tretirana kao vreća.

2.4.3 Preslikavanje kolekcije ugnježdenih objekata

Moguće je preslikati kolekciju komponenti, elemenata koji su vrednosni tipovi koje definiše korisnik. Pretpostavka je da se želi preslikati komponentna klasa `Slika`, koja ima instancne promenljive sa imenom slike, veličinom, itd.
Potrebno je dodati anotaciju `@Embedded` iznad definicije te klase kako bi se omogućilo ugnježdavanje:

```
@Embeddable
public class Slika {
	@Column (nullable=false)
	private String datoteka;
	@Column (nullable=false)
```

```
        private int dimenzijaX;
        @Column (nullable=false)
        private int dimenzijaY;
        ...
```

Budući da su tabeli za kolekciju potrebne sve kolone koje ima komponenta za formiranje kompozitnog primarnog ključa, veoma je bitno da se te kolone preslikaju kao NOT NULL. Sada je moguće ugraditi ovu komponentu u mapiranje kolekcije, čak i predefinisati definicije kolona (u primeru koji je naveden predefinisan je naziv pojedinačne kolone tabele kolekcije za datu komponentu; svim ostalim je dodeljeno ime koristeći podrazumevanu strategiju):

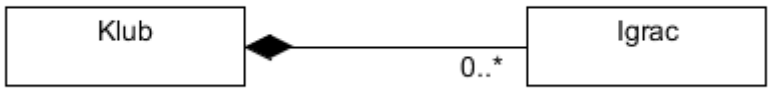
```
@ElementCollection
@JoinTable (name="SLIKA_IGRACA", joinColumns=@JoinColumn(name="IGRAC_ID"))
@AttributeOverride(name="datoteka", column=@Column(name="DATOTEKA_EXT"))
private Collection<Slika> slike = new ArrayList<Slika>();
```

Da bi se izbeglo navođenje komponenti sa `nullable` atributom, potreban je veštački primarni ključ na tabeli kolekcije. To se ostvaruje korišćenjem `@CollectionId` anotacije:

```
@ElementCollection
@JoinTable (name="SLIKA_IGRACA", joinColumns=@JoinColumn(name="IGRAC_ID"))
@AttributeOverride(name="datoteka",column=@Column(name="DATOTEKA_EXT"))
@GenericGenerator (name="hilo-gen", strategy="hilo")
@CollectionId (columns=@Column(name="SLIKA_ID"), generator="hilo-gen",
type=@Type(type="long"))
private Collection<Slika> slike = new ArrayList<Slika>();
```

2.5 Preslikavanje odnosa roditelj/dete

Kontrolisanje veza između klasa i veza između tabela je srce ORM-a. Do sada je prikazano preslikavanje veza između klasa vrednosnih tipova, sa različitim višestrukostima na krajevima tih veza. Višestrukost brojnosti jedan se preslikava kao jednostavna komponenta ili instancna promenljiva. Višestrukost brojnosti koja je veća od jedan zahteva kolekciju vrednosnih tipova. Sledeći korak je preslikavanje ovakvih višestrukosti, ali među klasama koje predstavljaju entitete. Naravno, aspekti koji se uzimaju u obzir prilikom rada sa entitetima, kao što su deljene reference i nezavisni životni ciklus, komplikuju preslikavanje ovakvih veza.



Slika 1. – Odnos između klasa `Klub` i `Igrac`

2.5.1 Višestrukost

Prilikom opisivanja i klasifikovanja veza, gotovo uvek se koristi termin višestrukost (engl. *multiplicity*). U ovom konkretnom primeru, višestrukost su samo dve informacije:

- Da li može biti više od jednog igrača za određeni klub?
- Da li može biti više od jednog kluba za određenog igrača?

Nakon pogleda na domen model, može se zaključiti da je veza iz klase `Igrac` ka klasi `Klub` *many-to-one*. S obzirom da su veze dvosmerne, inverzna veza iz klase `Klub` ka klasi `Igrac` je *one-to-many* veza.

2.5.2 Jednostavan primer

Veza od klase `Igrac` ka klasi `Klub` (i obrnuto) je primer najjednostavnije moguće veze između entiteta. U ovom konkretnom slučaju postoje dve instancne promenljive u dvema klasama. Jedna je kolekcija referenci, a druga je samo jedna jedina referenca.
Najpre, evo kako izgleda deo Java klase `Igrac`:

```
public class Igrac {
	...
	private String igracIme;
	private Klub klub;

	public Klub getKlub() {
		return klub;
	}

	public void setKlub(Klub klub) {
		this.klub = klub;
	}
	...
```

U JPA anotacijama, ova asocijacija se preslikava korišćenjem `@ManyToOne` anotacije:

```
public class Igrac {
	...
	@ManyToOne (targetEntity=domenModel.Klub.class)
	@JoinColumn (name="KLUB_ID", nullable=false)
	private Klub klub;
```

Postoje dva opciona elementa u ovom preslikavanju. Najpre, nije potrebno uključiti `targetEntity` za vezu; to se podrazumeva iz tipa polja. Eksplicitan `targetEntity` atribut je koristan u znatno kompleksnijim domen modelima – na primer, kada se vrši preslikavanje `@ManyToOne` *accessor* metode koja kao povratni tip ima izvedenu klasu, i predstavlja kopiju određenog interfejsa ciljanog entiteta.
Drugi opcioni element je `@JoinColumn`. Ukoliko se ne deklariše ime kolone za strani ključ, Hibernate automatski koristi kombinaciju imena ciljanog entiteta i imena identifikatora u bazi ciljanog entieta. Drugim rečima, ukoliko se ne doda `@JoinColumn` anotacija, podrazumevano ime za strani ključ je klub plus id, i međusobno su razdvojeni podvlakom (_). Međutim, pošto je cilj da je kolona kojom se predstavlja strani ključ NOT NULL, svejedno je anotacija potrebna kako bi se postavio atribut `nullable = false`.
Važno je primetiti da je moguće napisati kompletnu aplikaciju bez korišćenja bilo čega drugog. Nije potrebno preslikati drugu stranu ove asocijacije među klasama, i sve je već preslikano u SQL shemi (kolona za strani ključ).[1] Ukoliko je potrebna instanca klase `Klub` za koji igra određeni `Igrac`, dovoljno je pozvati `nekiIgrac.getKlub()`, i na taj način iskoristiti vezu među entitetima koja je upravo kreirana. Sa druge strane, ukoliko su potrebni svi igrači koji igraju za neki klub, moguće je napisati upit (u bilo kom jeziku koji Hibernate podržava).
Jedan od razloga zbog koga se koristi alat koji podržava puno objektno/relaciono preslikavanje kao što je Hibernate je želja da se izbegne pisanje takvih upita.

2.6 Stepenasta obrada stanja objekta

Pojam roditelja i deteta implicira da jedan vodi računa o drugome. U praksi, ovo znači da je potreban manji broj linija koda da se ostvari veza između roditelja i deteta, jer je neke stvari lakše odraditi

automatski. Kod koji sledi kreira novi Klub (koji smatramo roditeljem) i novu instancu klase Igrac (dete):

```
Klub klub = new Klub();
Igrac igrac = new Igrac();

klub.dodajIgraca(igrac);

session.save(klub);
session.save(igrac);
```

Drugi poziv session.save() metode deluje nepotreban, ukoliko bi se govorilo o istinskoj roditelj/dete vezi. Međutim, potrebno je setiti se entiteta i vrednosnih tipova: ukoliko su obe klase entiteti, njihove instance imaju potpuno nezavisne životne cikluse. Novokreirani objekti su privremeni i moraju preći u trajno stanje ukoliko se žele sačuvati u bazi. Njihova veza ne utiče na njihov životni ciklus, ukoliko su entiteti.[1] Na raspolaganju su tri opcije:

- Pobrinuti se za nezavisne instance i izvršavati dodatne pozive save() i delete() metoda na objektima klase Igrac kada je to potrebno – uz dodatni Java kod koji je potreban za kontrolisanje veza (dodavanje i izbacivanje referenci iz kolekcija).
- Pretvoriti klasu Igrac u vrednosni tip. Međutim, na taj način se gube aspekti koje ima jedan entitet, kao što su moguće deljene reference na jednu instancu.
- Treće rešenje koje nudi Hibernate je tranzitivna trajnost podataka, svojstvo kojim se smanjuje broj linija koda i dopušta Hibernate-u da automatski kontroliše životne cikluse međusobno povezanih instanci entiteta.

2.6.1 Tranzitivna trajnost podataka

Kada se instancira novi objekat klase Igrac i doda se klasi Klub, igrač bi trebao da bude trajan automatski. Želja je da se izbegne eksplicitna trajnost objekta klase Igrac sa dodatnom save() operacijom.
JPA podržava stepenastu obradu stanja instance entiteta u asocijacijama:

```
public class Klub {
	...
	@OneToMany (cascade={CascadeType.PERSIST, CascadeType.MERGE}, mappedBy="klub")
	private Set<Igrac> igraci = new HashSet<Igrac>();
	...
```

Atributi su za operacije za koje želimo da budu tranzitivne. Za nativni Hibernate, stepenuju se *save* i *update* operacije sa povezanim entitetima sa *persisit* i *merge*. Sada se može pojednostaviti kod koji povezuje klase Klub i Igrac, u nativnom Hibernate-u:

```
Klub klub = new Klub();
Igrac igrac = new Igrac();

klub.dodajIgraca(igrac);

session.persist(klub);
```

Svi entiteti u igraci kolekciji su sada takođe trajni, kao što bi bili da smo pozvali metodu save() za svaki od objekata klase Igrac.

2.7 Rad sa objektima

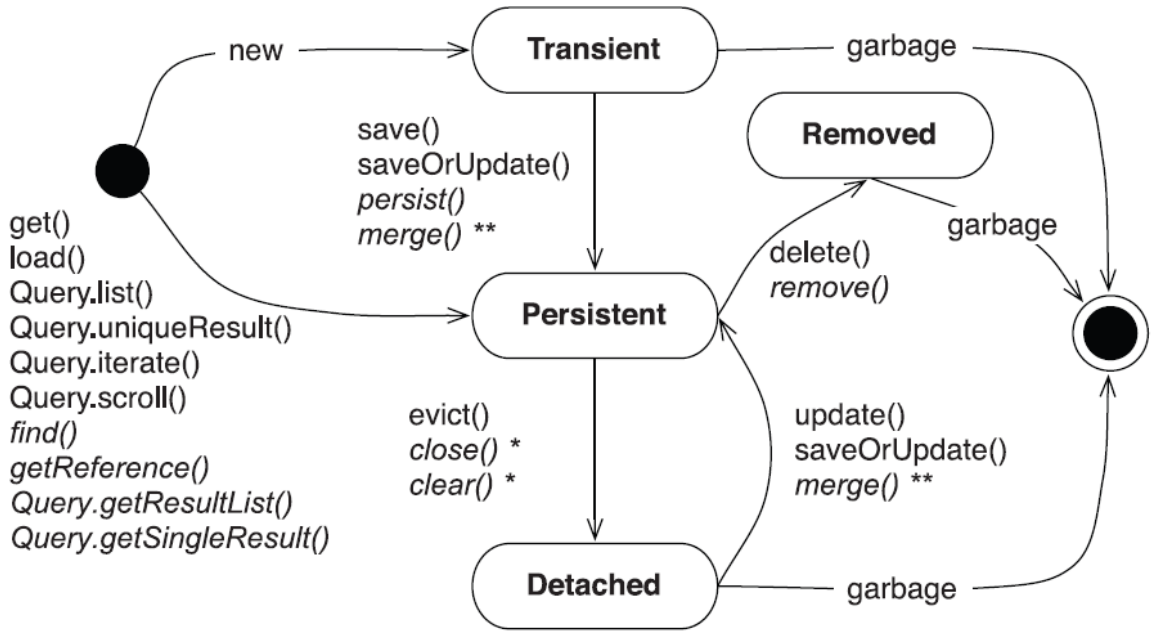
Temu od posebnog značaja predstavlja životni ciklus objekata – način na koji objekat prelazi u trajno stanje, kao i trenutak kada prestaje da se smatra trajnim – kao i pozivi odgovarajućih metoda kojima se ostvaruju ovi prelazi između stanja. Deo Hibernate-a, `Session`, vodi računa o trajnosti podataka.

2.7.1 Životni ciklus trajnih objekata

Budući da je Hibernate mehanizam za trajnost podataka koji je transparentan – što znači da klase nisu "svesne" trajnosti podataka – moguće je ostvariti logiku aplikacije kojoj nije od značaja da li objekti sa kojima operiše predstavljaju trajno stanje ili privremeno stanje (koje postoji samo u memoriji).
Bilo koja aplikacija sa trajnim stanjima mora komunicirati sa servisom za trajnost podataka, kad god je potrebno preneti stanje koje se nalazi u memoriji u bazu (ili obrnuto). Drugim rečima, moraju se koristiti Hibernate interfejsi za smeštanje i uzimanje objekata.
Trajnim životnim ciklusom se nazivaju stanja kroz koja objekat prolazi tokom svog životnog veka. Takođe se koristi termin jedinica rada: skup operacija koji se smatra jednom (obično atomičnom) grupom. Bitan element predstavlja kontekst trajnosti podataka, čija je uloga da pruži servis za trajnost podataka. Kontekst trajnosti podataka se može shvatiti kao keš koji vodi računa o svim modifikacijama i promenama stanja koja su se desila na objektu tokom određene jedinice rada.

2.7.2 Stanja objekta

Stanja objekta koja se interno koriste se mogu razlikovati od onih stanja koja su izložena aplikaciji koju koristi klijent. Hibernate definiše samo četiri stanja, sakrivajući kompleksnost internih implementacija od koda koji koristi klijent.
Stanja objekta koje definiše Hibernate i njihove tranzicije unutar dijagrama stanja su prikazani na slici. Takođe se mogu videti i pozivi metoda API-ja koji predstavlja kontroler trajnosti podataka i pokreće odgovarajuće prelaze. Taj API u Hibernate-u je `Session`.



Slika 2. – Stanja u kojima se Java objekat može naći

Tokom svog postojanja, objekat može preći iz objekta koji je privremen u trajno stanje, i zatim u nepovezani objekat.

2.7.3 Privremeni objekti

Objekti koji su instancirani korišćenjem operatora `new` nisu odmah trajni. Njihovo stanje je privremeno, što znači da nisu vezani za bilo koju vrstu tabele. Prema tome, njihovo stanje se gubi čim na njih više ne bude referenci od strane nekog drugog objekta. Ovi objekti imaju životni vek koji se efektivno navršava u tom trenutku, postaju nedostupni i kandidati da budu pokupljeni od strane sakupljača otpadaka.
Hibernate smatra da su sve instance objekata koje su privremene netransakcione; bilo koja modifikacija instance koja je privremena nije registrovana u kontekstu trajnosti podataka. To znači da Hibernate ne pruža *roll-back* funkcionalnost za objekte koji su privremeni.
Objekti na koje referišu samo privremene instance su, podrazumevano, takođe privremeni. Da bi instanca prešla iz stanja privremeno u trajno stanje, potreban je ili poziv kontroleru trajnosti podataka, ili kreiranje reference od neke instance koja je već trajna.

2.7.4 Trajni objekti

Trajna instanca je instanca klase koja je entitet, sa identifikatorom u bazi podataka koji predstavlja primarni ključ.
Trajne instance mogu biti objekti koje instancira aplikacija i onda preslikani pozivom jedne od metoda koju izvršava kontroler trajnosti podataka. To mogu biti i objekti koji postaju trajni kada se kreira referenca od nekog drugog objekta koji je trajan. Trajna instanca može biti instanca uzeta iz baze podataka izvršavanjem upita, pretragom po identifikatoru, ili kretanjem kroz graf objekata koji su trajni.
Trajne instance su uvek povezane sa kontekstom trajnosti podataka. Hibernate ih kešira i može ih detektovati bez obzira da li su promenjene u aplikaciji.

2.8 Pristupanje bazi korišćenjem okvira Hibernate

Kako Hibernate obavlja CRUD operacije, i kolika su poboljšanja koja on nudi prilikom rada sa objektima? Bez zalaženja u HQL i ostale egzotike koje Hibernate ima u rukavu, biće prikazane osnovne CRUD operacije unutar okvira Hibernate.
Unošenje podatka (objekta) u bazu je jednostavno uraditi pomoću sledećih naredbi:

```
Igrac igrac = new Igrac("Ray Allen", 34);
session.save(igrac);
```

Najpre kreiramo objekat klase `Igrac`. I sve što je potrebno da unesemo objekat u bazu je da pozovemo `save()` metodu. Hibernate će prepoznati klasu i upisati objekat u odgovarajuću tabelu.

Uzimanje objekta iz baze je jednostavno.

```
Igrac igrac = new Igrac();
igrac = (Igrac) session.get(Igrac.class, 8);
```

Argumenti `get()` metode su klasa objekta koji hoćemo da uzmemo iz baze, i primarni ključ objekta u odgovarajućoj tabeli. Operacije ažuriranja i brisanja su takođe slične (brisanje može biti malo komplikovanije), pa neće biti navedeni konkretni primeri.
Kao što se može videti, ORM alat, u ovom slučaju Hibernate, omogućava elegantan rad sa objektima unutar relacionih baza. Nema potrebe za ogromnim brojem linija koda, za naredbama koje se ponavljaju, za konekcijama i zatvaranjem istih. Za sve operacije koje su zahtevale linije i linije koda, par linija će završiti posao. Upravo u tome se ogleda osnovna prednost korišćenja ORM alata.

3. Spring

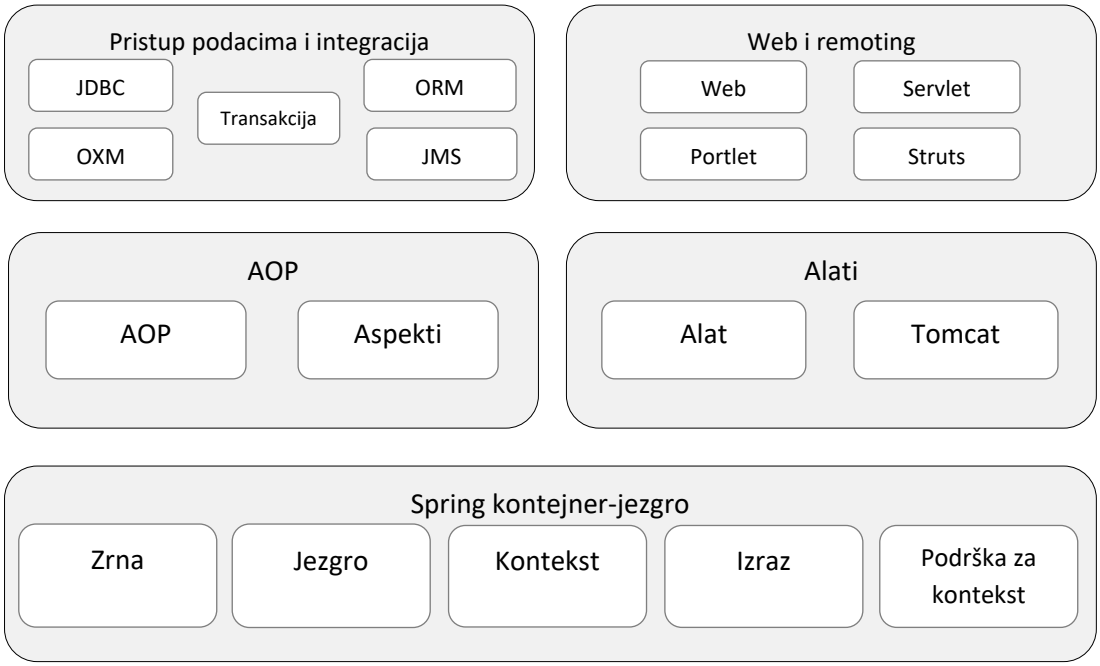
Spring je otvoreni kod okvir, koji je nastao sa ciljem da se reši problem kompleksnosti razvoja poslovnih aplikcija. Spring omogućava upotrebu čistih Java objekata, kako bi se realizovale stvari koje su ranije bile moguće jedino primenom EJB-a.[12] Spring komponenta može biti bilo koji tip POJO-a. Svaka Java aplikacija može imati koristi od primene Spring-a u pogledu jednostavnosti, testiranja i razdvajanja komponenti. Spring omogućava mnogo pogodnosti, ali kada se svede na osnovne elemente, Spring je kontejner i okvir koji pruža ubacivanje zavisnosti (engl. *dependency injection*, *DI*) i aspektno-orijentisan dizajn (engl. *apsect-oriented design*).[12] U osnovi Spring-a je nekoliko ključnih ideja, čiji je fokus na osnovnom cilju okvira Spring: pojednostavljeni razvoj Java aplikacija.

- Jednostavnost – Spring je jednostavan i u pogledu veličine i pogledu nagomilavanja u memoriji. Najveći deo okvira Spring se distribuira unutar jedne JAR datoteke. Nagomilavanje u memoriji uzrokovano Spring-om je zanemarljivo. Šta više, Spring je indirektan: objekti u aplikacijama koje koriste Spring često ne zavise od klasa koje su karakteristične za okvir Spring.
- Ubacivanje zavisnosti – Spring podržava razdvajanje komponenti kroz tehniku koja se naziva ubacivanje zavisnosti.[13] Kada se primenjuje DI, objektima se pasivno predaju elementi od kojih su zavisni, umesto da objekti kreiraju sami te elemente, ili da ih zahtevaju, kako bi mogli da se koriste u aplikaciji.
- Aspektno-orijentisan – Spring se isporučuje sa bogatom podrškom za aspektno-orijentisano programiranje (AOP), koja omogućava uporedno razvijanje i razdvajanje poslovne logike aplikacije od sistemskih servisa. Objekti aplikacije izvršavaju upravo ono što se od njih očekuje – izvode poslovnu logiku – i ništa više. Nisu odgovorni za (niti su "svesni") ostalih delova sistema, kao što su npr. logovi.
- Kontejner – Spring je kontejner u smislu da sadrži i kontroliše životni ciklus i konfiguraciju objekata unutar aplikacije. U Spring-u, moguće je deklarisati način kako bi svaki od objekata aplikacije trebao biti formiran, konfigurisan, i kako bi objekti trebali biti međusobno povezani.
- Okvir – Spring omogućava konfigurisanje i izradu kompleksnih aplikacija od jednostavnih komponenti. U Spring-u, objekti aplikacije se slažu na deklarativni način, obično u XML datoteci.[12]

Kada se Spring svede na gradivne elemente, ono što programer dobija je okvir, koji mu pomaže da razvija kod sa razdvojenim komponentama aplikacije na kojoj radi.

3.1 Spring moduli

Okvir Spring se sastoji od nekoliko dobro definisanih modula. Posmatrani kao celina, ovi moduli pružaju sve što je potrebno za razvoj poslovnih aplikacija. Nije neophodno zasnovati aplikaciju u potpunosti na okviru Spring. Programer ima slobodu da izabere module koji se najbolje uklapaju u njegovu aplikaciju, i da potraži druga rešenja kada ona koja Spring nudi ne mogu zadovoljiti potrebe konkretne aplikacije. Spring omogućava integraciju sa nekoliko drugih okvira i biblioteka, tako da ih programer ne mora sam pisati.



Slika 3. – Spring moduli

Kao što se može primetiti, svi Spring moduli su izgrađeni iznad kontejner-jezgra. Kontejner definiše kako su zrna kreirana, konfigurisana i kako se upravlja zrnima. Ove klase će se implicitno koristiti prilikom konfigurisanja aplikacije. Međutim, programerima će verovatno biti interesantniji neki drugi moduli koji utiču na servise koje ima kontejner. Ovi moduli će pružiti okvire pomoću kojih će biti izgrađeni servisi aplikacije, kao što su AOP i trajnost podataka. Sledi reč više o nekim modulima koji su od značaja za ovaj rad, među kojima su:

- Kontejner-jezgro (engl. *core container*) – U osnovi svih modula se nalazi Spring kontejner-jezgro. Upravo on omogućava fundamentalnu funkcionalnost okvira Spring.[12] Unutar ovog modula se nalazi `BeanFactory`, koji je osnovni Spring kontejner i temelj na kome se bazira ubacivanje zavisnosti.
- Modul konteksta aplikacije – Spring kontekst aplikacije je zasnovan na kontejner-jezgru. `BeanFactory` iz modula jezgra je ono što čini Spring kontejnerom, ali konteksni modul je ono što ga čini okvirom.[12] Pruža mnoge poslovne servise kao što su e-mail, JNDI (Java Naming and Directory Interface) pristup, EJB integracija, planiranje, itd.
- JDBC apstrakcija i DAO modul – Rad sa JDBC-om često uzrokuje mnogo standardnog koda kojim se ostvaruje konekcija, kreiraju naredbe, procesira skup rezultata, i zatim zatvara konekcija. Spring JDBC i *Data Access Objects* (DAO) modul apstrahuju standardni kod, tako da je kod koji se odnosi na rad sa bazom podataka čist i jednostavan, i sprečava probleme koji su uzrokovani npr. nemogućnošću zatvaranja resursa baze podataka. Ovaj modul donosi i sloj značajnih izuzetaka koji se javljaju zajedno sa porukama o grešci, koje proizvode neki serveri baza podataka. Na taj način se smanjuje potreba za dešifrovanjem SQL poruka o greškama.
- Modul integracije objektno/relacionog preslikavanja – Za programere koji preferiraju korišćenje alata za objektno/relaciono preslikavanje umesto čistog JDBC-a, Spring pruža ORM modul. Spring-ova podrška za ORM se zasniva na DAO podršci, i pruža ugodan način za izgradnju DAO za nekoliko ORM rešenja. Spring ne pokušava da implementira svoje ORM rešenje, već omogućava povezivanje sa nekoliko popularnih ORM okvira, kao što su Hibernate, Java Persistence API, Java Data Objects, kao i iBATIS SQL Maps.

3.2 Princip ubacivanja zavisnosti

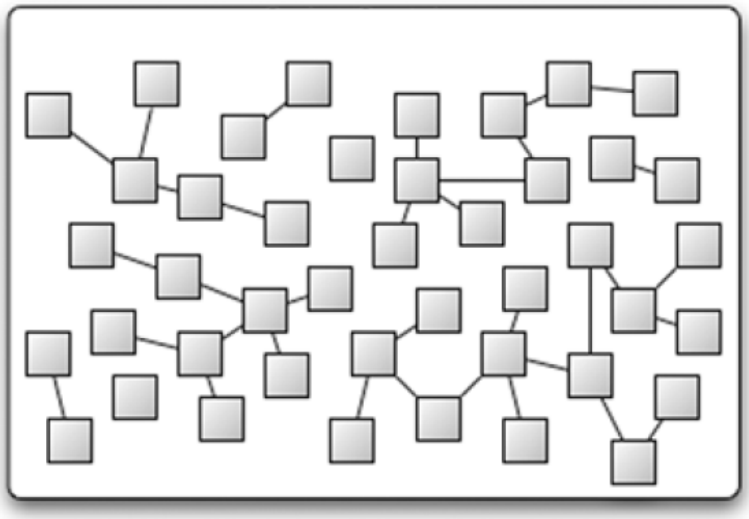
DI je samo srce okvira Spring. Upotrebom DI u projektima, kod će biti znatno jednostavniji, lakši i za razumevanje i za testiranje. Šta se krije iza termina ubacivanje zavisnosti?
Originalno, ubacivanje zavisnosti je bilo poznato pod drugim nazivom: inverzija kontrole. U jednom članku napisanom rane 2004. godine [12], Martin Fowler je postavio pitanje: koji aspekt kontrole se invertuje? Zaključio je da je ono što se invertuje dobijanje zavisnosti (tj. objekata sa kojima se operiše). Polazeći od tog otkrića, izveo je frazu ubacivanje zavisnosti, koja bolje opisuje ono što se dešava.
Bilo koja netrivijalna aplikacija (a tu se podrazumeva bilo šta kompleksnije od `HelloWorld.java` programa) se sastoji od dve ili više klasa koje sarađuju međusobno kako bi izvršile neku poslovnu logiku. Tradicionalno, svaki objekat je odgovoran za kreiranje referenci na objekte sa kojima sarađuje (njegove zavisnosti). Ovo može dovesti do jako povezanog koda, koji je ujedno veoma težak za testiranje.

3.3 Osnovno o povezivanju zrna

Objekti nisu zaduženi za pronalaženje ili kreiranje ostalih objekata koji su im potrebni da bi odradili svoj posao. Umesto toga, kontejner im daje reference na objekte sa kojima sarađuju. Postupak kreiranja ovih asocijacija između objekata u aplikaciji je suština ubacivanja zavisnosti, i obično se naziva povezivanje (engl. *wiring*).

3.3.1 Gde se nalaze zrna?

U aplikaciji zasnovanoj na okviru Spring, objekti aplikacije će živeti unutar Spring kontejnera. Kao što je prikazano na slici, kontejner će kreirati objekte, povezati ih, izvršiti njihovu konfiguraciju, i upravljati njihovim kompletnim životnim ciklusom.



Slika 4. – Spring kontejner

Ne postoji samo jedan Spring kontejner. Spring poseduje nekoliko implementacija kontejnera koji mogu biti podeljeni na dva različita tipa. Fabrike zrna (engl. *bean factories*), definisane `org.springframework.beans.factory.BeanFactory` interfejsom i kontekst aplikacije (engl.

application context), koji je definisan `org.springframework.context.ApplicationContext` interfejsom.

3.4 Rad sa kontekstom aplikacije

Fabrika zrna je pogodno rešenje za jednostavne aplikacije. Kako bi se iskoristila puna snaga okvira Spring, verovatno će se koristiti učitavanje zrna aplikacije korišćenjem naprednijeg kontejnera koji Spring pruža: konteksta aplikacije.
`ApplicationContext` je veoma sličan `BeanFactory`-ju. I jedan i drugi učitavaju definicije i povezuju zrna. Usled dodatnih funkcionalnosti koje pruža, ApplicationContext se koristi češće nego `BeanFactory` u skoro svim aplikacijama.[12] Među mnogim implementacijama `ApplicationContext`-a tri se najčešće koriste: `ClassPathXmlApplicationContext`, `FileSystemXmlApplicationContext` i `XmlWebApplicationContext`.
Uzimanje zrna korišćenjem `ApplicationContext`-a je isto kao i iz `BeanFactory`-ja: korišćenjem `getBean()` metode.
Ukoliko se dodatne funkcionalnosti koje omogućuje kontekst aplikacije ostave po strani, još jedna velika razlika između konteksta aplikacije i fabrike zrna je u tome kako se *singleton* zrna učitavaju. Fabrika zrna vrši lenju inicijalizaciju svih zrna, odlažući kreiranje samog zrna dok se ne pozove `getBean()` metoda. Kontekst aplikacije vrši predučitavanje svih *singleton* zrna po samom startovanju konteksta. Tim predučitavanjem *singleton* zrna, dobija se sigurnost da će biti spremna za upotrebu kada budu potrebna – aplikacija neće morati da čeka na njihovo kreiranje.

3.5 Kod vezan za JDBC

Iako JDBC pruža API koji je u radu blizak bazi podataka, programer ima obavezu da kontroliše sve što je povezano sa pristupanjem bazi. To uključuje kontrolu resursa baze podataka i rukovanje izuzecima.
Oslobađanje resursa i rukovanje greškama je ono što čini pristupanje podacima robusnim. Bez toga, greške bi se prošle nezapažene i resursi bi ostali otvoreni, što bi moglo dovesti do nepredvidivog koda i curenja resursa. Tako da ne samo da je ovo potrebno unutar koda, već je potrebno obezbediti i ispravnost. To je samo razlog više za prebacivanje kontrole na okvir, kako bi programer bio siguran da je kod napisan samo jednom, i da je ispravan.

3.5.1 Rad sa JDBC šablonima

Spring JDBC okvir će pročistiti JDBC kod tako što će preuzeti upravljanje resursima i rukovanje izuzecima. Ovo programeru omogućava da piše samo kod koji je neophodan za premeštanje podataka u i iz baze. Za JDBC, Spring ima tri klase šablona koje se mogu koristiti prilikom rada: `JdbcTemplate`, `NamedParameterJdbcTemplate` i `SimpleJdbcTemplate`.
Prvo poboljšanje u odnosu na tradicionalni JDBC pristup je da se mogu iskoristiti zrna koja Spring nudi, tako da se informacije o bazi i konekciji mogu smestiti unutar `spring.xml` datoteke, sa sledećim podacima:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
       <property name="driverClassName" value="org.postgresql.Driver" />
       <property name="url" value="jdbc:postgresql://localhost:5432/postgres" />
       <property name="username" value="postgres" />
       <property name="password" value="password" />
</bean>
```

Za potrebe ostvarivanja konekcije sa bazom, lako je generisati zrno pozivanjem `getBean()` metode (ili u ovom konkretnom slučaju, to se radi uz pomoć `@Autowired` anotacije). Spring takođe omogućava kreiranje objekata upotrebom klase koja implementira `RowMapper` interfejs. On programeru omogućava da se osloni na Spring kada je potrebno kreirati objekat na osnovu podataka iz baze. Sledi praktična implementacija, uz upotrebu `JdbcTemplate`-a.
`JdbcTemplate` se može inicijalizovati unutar *setter* metode `DataSource` objekta; to je moguće pošto se inicijalne vrednosti za ovaj objekat nalaze unutar `spring.xml` datoteke.

```
@Autowired
public void setDataSource(DataSource dataSource) {
	this.jdbcTemplate = new JdbcTemplate(dataSource);
}
```

Metoda pomoću koje dobijamo objekat klase `Igrac` je:

```
public Igrac getIgracForId(int igracId) {
	String sql = "SELECT * FROM player_info WHERE person_id = ?";
	return jdbcTemplate.queryForObject(sql, new Object[] {igracId}, new IgracMapper());
}
```

Unutar `RowMapper` klase je potrebno predefinisati `mapRow()` metodu:

```
private static final class IgracMapper implements RowMapper<Igrac> {
	@Override
	public Igrac mapRow(ResultSet resultSet, int rowNum) throws SQLException {
		Igrac igrac = new Igrac();
		igrac.setIme(resultSet.getString("name"));
		igrac.setOsobaId(resultSet.getInt("person_id"));
		igrac.setBrojDresa(resultSet.getInt("jersey_no"));

		return igrac;
	}
}
```

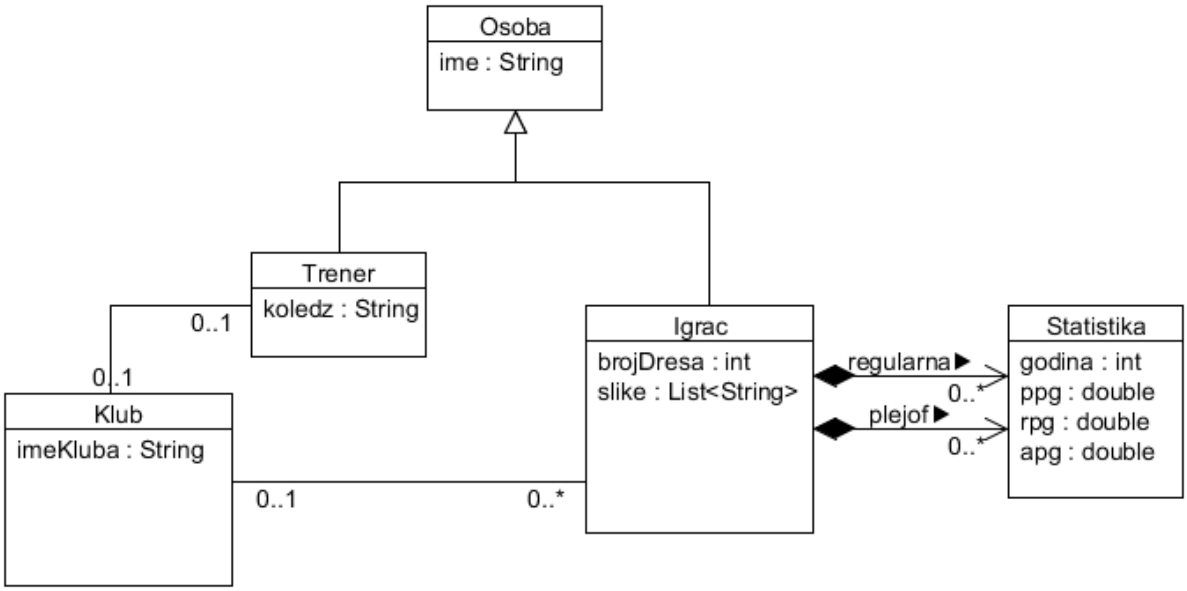
4. Pregled aplikacije

Aplikacija implementira većinu Hibernate svojstava opisanih u poglavlju *2. Hibernate*. Pogodnosti koje sa sobom donosi Hibernate se možda i ne mogu najbolje uočiti, budući da je aplikacija manjeg obima i prvenstveno služi kao ilustracija. Međutim, ušteda u pogledu vremena, broja linija koda i utrošenih sati tokom razvoja se znantno lakše može primetiti na poslovnim aplikacijama većeg obima.

Konfiguraciona XML datoteka za Hibernate ima sledeći sadržaj:

```
<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">org.postgresql.Driver</property>
        <property
name="connection.url">jdbc:postgresql://localhost:5432/postgres</property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">password</property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Disable the second-level cache  -->
        <property
name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <!-- Names the annotated entity class -->
        <mapping class="domen.model.Klub"/>
        <mapping class="domen.model.Osoba"/>
        <mapping class="domen.model.Trener"/>
        <mapping class="domen.model.Igrac"/>
    </session-factory>
</hibernate-configuration>
```

Najvažnije je svojstvo `hbm2ddl.auto`, koje je postavljeno na vrednost `create`. Upravo to omogućava kreiranje baze podataka.



Slika 5. – UML dijagram domen modela

Na osnovu anotacija prisutnih u klasama domen modela se vrši objektno/relaciono preslikavanje i generisanje baze. Slede definicije nekih klasa, kao i isticanje par interesantnih mesta.

```
@Entity
@Table (name="CLUB_INFO")
public class Klub {
	@Id
	@GeneratedValue (strategy=GenerationType.AUTO)
	@Column(name="CLUB_ID")
	private int klubId;
	@Column(name="CLUB_NAME")
	private String imeKluba;
	@OneToOne (cascade=CascadeType.PERSIST)
	@JoinColumn(name="COACH_ID")
	private Trener trener;
	@OneToMany (cascade=CascadeType.PERSIST)
	@JoinTable(name="CLUB_PLAYER",
			joinColumns=@JoinColumn(name="CLUB_ID"),
			inverseJoinColumns=@JoinColumn(name="PLAYER_ID"))
	private Set<Igrac> igraci = new HashSet<>();
...
```

Anotacijom `@JoinTable` i atributima `joinColumns` i `inverseJoinColumns` je omogućeno kreiranje `CLUB_PLAYER` tabele.

```
@Entity
@Table (name="COACH_INFO")
public class Trener extends Osoba {
	@Column(name="COLLEGE")
	private String koledz;
	@OneToOne (mappedBy="trener")
	private Klub klub;
...
```

`mappedBy` atribut `@OneToOne` anotacije prebacuje odgovornost za preslikavanje na klasu `Klub`, pa se unutar tabele za klasu `Trener` neće nalaziti kolona za strani ključ.

```
@Entity
@Table (name="PLAYER_INFO")
public class Igrac extends Osoba {
	@Column (name="JERSEY_NO")
	private int brojDresa;
...
@ElementCollection
@JoinTable (name="REGULAR_SEASON", joinColumns=@JoinColumn(name="PLAYER_ID"))
private Set<Statistika> statistike = new HashSet<Statistika>();

@AttributeOverrides ({
	@AttributeOverride (name="godina", column=@Column(name="PO_YEAR")),
	@AttributeOverride (name="ppg", column=@Column(name="PO_PPG")),
	@AttributeOverride (name="rpg", column=@Column(name="PO_RPG")),
	@AttributeOverride (name="apg", column=@Column(name="PO_APG"))
})
@ElementCollection
@JoinTable (name="PLAY_OFF", joinColumns=@JoinColumn(name="PLAYER_ID"))
private Set<Statistika> plejof = new HashSet<Statistika>();

@OneToOne
@JoinColumn (name="CLUB_ID")
private Klub klub;

@ElementCollection
@GenericGenerator(name="hilo-gen", strategy="hilo")
@CollectionId(columns = { @Column(name="PICTURE_ID") }, generator = "hilo-gen", type =
@Type(type="int"))
@JoinTable(name="PICTURE_INFO", joinColumns=@JoinColumn(name="PLAYER_ID"))
@Column (name="PICTURE")
private Collection<String> slike = new ArrayList<String>();
...
```

Unutar klase `Igrac` akcenat je na preslikavanju kolekcija. Hibernate generiše posebnu tabelu za svaku od navedenih kolekcija. Tabele `REGULAR_SEASON` i `PLAY_OFF` nemaju primarne ključeve, jer su u pitanju skupovi vrednosnih tipova, a uz to su implementirane kao `HashSet`. Sa druge strane, za tabelu `PICTURE_INFO` se generiše primarni ključ, budući da je to moguće pošto je kolekcija `slike` implementirana kao `ArrayList`.

Definicije zrna kojima se inicijalizuju tabele baze se nalaze unutar `spring.xml` datoteke. Na osnovu `<bean>` konfiguracije Spring kreira zrno u aplikaciji. Vrednost atributa `id` predstavlja identifikator pomoću koga Spring kontejner kontroliše zrno. Postoje dva načina da zrno dobije vrednosti, preko konstruktora i preko *setter* metoda. Ta dva pristupa su prikazana u zrnu `detroitT` i `detroit1`, respektivno. Pored toga, koriste se još i unutrašnja zrna i kolekcije.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
	<bean id="detroit" class="domen.model.Klub" >
		<property name="imeKluba" value="Detroit Pistons" />
		<property name="trener" ref="detroitT" />
		<property name="igraci" >
			<set>
				<ref bean="detroit1" />
				<ref bean="detroit2" />
				<ref bean="detroit3" />
				<ref bean="detroit4" />
			</set>
		</property>
	</bean>
```

```
	<bean id="detroitT" class="domen.model.Trener" >
		<constructor-arg index="0" value="Maurice Edward Cheeks" />
		<constructor-arg index="1" value="West Texas University" />
		<constructor-arg index="2" ref="detroit" />
	</bean>

	<bean id="detroit1" class="domen.model.Igrac" >
		<property name="ime" value="Josh Smith" />
		<property name="brojDresa" value="5" />
		<property name="klub" ref="detroit" />
		<property name="statistike">
			<set>
				<bean class="domen.model.Statistika">
					<property name="godina" value="2013" />
					<property name="ppg" value="17.5" />
					<property name="rpg" value="8.4" />
					<property name="apg" value="4.2" />
				</bean>
...
```

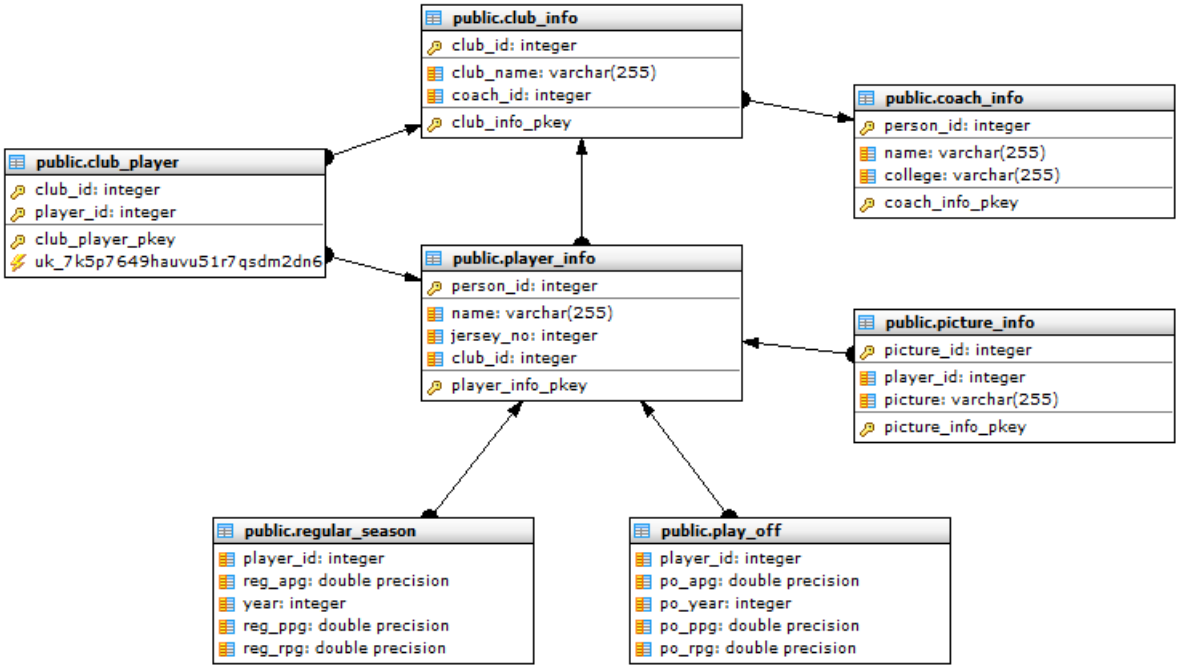
Deo klase unutar koje je realizovano povezivanje Hibernate i Spring okvira:

```
public static void main(String[] args) {
	SessionFactory sessionFactory = configureSessionFactory();
	Session session = sessionFactory.openSession();

	AbstractApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
	context.registerShutdownHook();

	session.beginTransaction();

	Klub detroit = (Klub) context.getBean("detroit");
	...
	session.persist(detroit);
	...
	session.getTransaction().commit();
	session.close();
	...
```

I na kraju, slede SQL naredbe koje Hibernate koristi prilikom generisanja baze:

```
create table CLUB_INFO (CLUB_ID int4 not null, CLUB_NAME varchar(255), COACH_ID int4,
primary key (CLUB_ID))
create table CLUB_PLAYER (CLUB_ID int4 not null, PLAYER_ID int4 not null, primary key
(CLUB_ID, PLAYER_ID))
create table COACH_INFO (PERSON_ID int4 not null, NAME varchar(255), COLLEGE varchar(255),
primary key (PERSON_ID))
create table PICTURE_INFO (PLAYER_ID int4 not null, PICTURE varchar(255), PICTURE_ID int4
not null, primary key (PICTURE_ID))
create table PLAYER_INFO (PERSON_ID int4 not null, NAME varchar(255), JERSEY_NO int4,
CLUB_ID int4, primary key (PERSON_ID))
create table PLAY_OFF (PLAYER_ID int4 not null, PO_APG float8, PO_YEAR int4, PO_PPG float8,
PO_RPG float8)
create table REGULAR_SEASON (PLAYER_ID int4 not null, REG_APG float8, YEAR int4, REG_PPG
float8, REG_RPG float8)
```

Shema generisane baze se može videti na slici:



Slika 6. – Shema generisane baze podataka

Aplikacija je razvijana u Eclipse okruženju na programskom jeziku Java. Iskorišćeni su okviri Hibernate i Spring, na sledeći način:

- Okvir Hibernate i objektno/relaciono preslikavanje koje on omogućava su dali jednostavan način generisanja sheme baze podataka.
- Okvir Spring i ubacivanje zavisnosti su iskorišćeni u svrhu inicijalizacije podataka unutar baze.

Aplikacija je razvijana sa PostgreSQL dijalektom, i prilikom testiranja je upotrebljena pgAdmin III aplikacija, koja pruža kontrolu nad bazom.
Cilj aplikacije je generisanje sheme baze podataka, na osnovu domen modela konstruisanog u programskom jeziku Java. Baza se generiše uz pomoć opcije koju nudi Hibernate okvir, hbm2ddl.auto.
Za pokretanje aplikacije je potrebno imati na računaru:

- Poslednju dostupnu verziju Java okruženja, koja se može naći na sajtu: http://www.oracle.com/technetwork/java/index.html.
- postgreSQL bazu podataka, koja se može naći na sajtu: http://www.postgresql.org/. (Prilikom instalacije, postaviti `username` i `password` na `postgres` i `password` vrednosti, respektivno.)

Zaključak

Objektni model koristi principe apstrakcije, enkapsulacije, modularnosti, hijerarhije, tipiziranosti, konkurentnosti, polimorfizma i trajnosti podataka. Objektni model omogućava kreiranje dobro struktuiranih, kompleksnih sistema. Unutar objektno modelovanog sistema, objekti su komponente sistema. Objekti su instance klasa, a veze između klasa su realizovane kroz nasleđivanje. Objekat ima identitet, stanje, i ponašanje. Objektni model pomaže u kreiranju okvira za aplikacije koje se mogu ponovo koristiti i sistema koji se mogu razvijati tokom vremena. Pored toga, objektno-orijentisani sistemi su obično manjeg obima od implementacija koje nisu objektno-orijentisane.
Sa druge strane, relacioni model definiše strukture podataka, manipulaciju tim podacima, kao i integritet podataka. Podaci su organizovani u vidu tabela, i različite tabele su povezane kroz sredstva referencijalnog integriteta (strani ključ). Ograničenja integriteta kao što su primarni ključ, ograničenja provere jedinstvenosti, kao i `not null` se koriste da održe integritet entiteta u relacionom modelu.
Relacioni model podataka nije fokusiran na podršku nasleđivanja tipova entiteta: polimorfizam koji se zasniva na entitetima iz objektnog modela se ne može prevesti u slične entitete u relacionom modelu. U objektnom modelu, koristi se stanje modela za definisanje jednakosti entiteta. U relacionom modelu, koristi se primarni ključ za definisanje jednakosti entiteta. Reference na objekte se koriste za povezivanje različitih objekata u objektnom modelu, a sa druge strane, strani ključ se koristi za uspopstavljanje veza u relacionom modelu. Budući da su ova dva modela suštinski različita, potrebno je pronaći način za adekvatno preslikavanje Java objekata u relacionu bazu podataka.
Okviri koji se bave objektno/relacionim preslikavanjem pomažu u iskorišćavanju svojstava koja su prisutna u objektnom modelu (kao što je Java) i relacionom modelu (kao što su sistemi za upravljanje bazama podataka). Uz pomoć ORM okvira, moguće je izvršiti preslikavanje Java objekata u relacione tabele korišćenjem metapodataka, koji definišu preslikavanje između objekata i baze. Metapodaci smanjuju kompleksnost direktnog rada sa SQL-om i pomažu u razvoju rešenja unutar poslovnog okvira aplikacije.
ORM rešenje može biti implementirano na različitim nivoima:

- *Čisto relaciono preslikavanje*: dizajn aplikacije se zasniva na relacionom modelu.
- *Lako objektno preslikavanje*: entiteti su predstavljeni kao klase i ručno se preslikavaju u relacione tabele.
- *Srednje objektno preslikavanje*: dizajn aplikacije se zasniva na objektnom modelu, i SQL kod je generisan tokom kompilacije.
- *Puno objektno preslikavanje*: ovo preslikavanje podržava sofisticirano modelovanje objekata, uključujući kompoziciju, nasleđivanje i polimorfizam.

Neke prednosti korišćenja ORM okvira:

- *Produktivnost*: iz razloga što se koriste metapodaci za preslikavanje i pretragu podataka, količina vremena potrebnog za razvoj se smanjuje, i samim tim se povećava produktivnost.
- *Održivost*: Zato što se većina posla odvija kroz konfiguraciju, broj linija koda je znatno manji i samim tim potrebe za održavanjem su takođe smanjene.
- *Nezavisnost od distributera*: ORM apstrahuje aplikaciju od SQL baze i SQL dijalekta. Na taj način se dobija portabilnost, koja se ogleda kroz podršku višestrukog broja baza podataka.

ORM okviri takođe imaju i neke mane:

- *Kriva učenja*: Moguće je susresti se sa strmom krivom učenja, dok se programer upozna sa suštinom procesa preslikavanja, i eventualno, nauči novi upitni jezik.
- *Nagomilavanje u memoriji*: Za jednostavne aplikacije koje koriste samo jednu bazu i podatke koji ne zahtevaju kompleksne upite, ORM može uzrokovati nepotrebno nagomilavanje u memoriji.
- *Smanjenje performansi*: Prilikom ažuriranja velikog obima, performanse su znatno slabije.

Hibernate je jedan od najrasprostranjenijih ORM okvira u industriji. Pruža sve pogodnosti koje nudi jedno ORM rešenje i implementira Java Persistence API (JPA) definisan unutar Enterprise JavaBeans (EJB) 3.0 specifikacije. U njegove osnovne komponenete spadaju:

- *Hibernate jezgro*: Jezgro generiše SQL i oslobađa programera od rukovanja Java Database Connectivity (JDBC) skupom rezultata i konverzijom objekata, dok su metapodaci definisani u jednostavnim XML datotekama. Jezgro nudi razne opcije za pisanje upita: čisti SQL, Hibernate Query Language (HQL), koji je specifičan za Hibernate, programabile kriterije, ili Query by Example (QBE).
- *Hibernate anotacije*: Od uvođenja anotacija u JDK 5.0, Hibernate pruža opciju definisanja metapodataka korišćenjem anotacija. Ovo skraćuje konfigurisanje u poređenju sa korišćenjem XML datoteka, i omogućuje jednostavno definisanje neophodnih metapodataka direktno unutar izvornog Java koda.
- *Hibernate EntityManager*: JPA specifikacija definiše interfejse za programiranje, pravila za životni vek za trajne objekte, i svojstva za upite. Hibernate implementacija za ovaj deo JPA standarda je omogućena kroz Hibernate `EntityManager`.

Na osnovu svega prikazanog, lako je zaključiti zbog čega ORM okviri i njihova integracija u savremene aplikacije imaju sve veći broj pristalica. Prednosti koje oni pružaju su ogromne, a mane se lako mogu prevazići korišćenjem nativnog SQL koda. Samo što je mnogo pogodnije pisati SQL CRUD naredbe na mestima gde je to neophodno umesto razvijati čitavu aplikaciju na taj način. Hibernate, i Java programski jezik, za koji je vezan, omogućavaju eleganciju i lepotu razvoja objektno-orijentisanih aplikacija koje koriste relacione baze podataka, na način koji opisuje svakog dobrog programera. Da bi programer bio uspešan, potrebno je da su njegova rešenja jednostavna i lako razumljiva.

Literatura

[1] Bauer C., King G., "Java Persistence with Hibernate", Manning Publications Co., 2007.

[2] Beighley L., "Head First SQL", O'Reilly Media, Inc., 2007.

[3] Eckel B., "Thinking in Java", Prentice-Hall, 2006.

[4] Pavlović-Lažetić G., "Uvod u Relacione Baze Podataka", Matematički Fakultet Beograd, 1999.

[5] Walls C., "Spring in Action", Manning Publications Co., 2011.

[6] Guruzu S., Mak G., "Hibernate Recipes: A Problem-Solution Approach", Apress, 2010.

[7] Tepper Fisher P., Murphy B., "Spring Persistence with Hibernate", Apress, 2010.

[8] Sierra K., Bates B., "Head First Java", O'Reilly Media, Inc., 2005.

[9] Minter D., Linwood J., "Beginning Hibernate: From Novice to Professional", Apress, 2006.

[10] Keith M., Schincariol M., "Pro EJB 3: Java Persistence API", Apress, 2006.

[11] Horton I., "Od početka... Java 2 JDK 5", CET, 2006.

[12] Walls C., Breidenbach R., "Spring in Action", Manning Publications Co., 2008.

[13] "Inversion of Control Containers and the Dependency Injection pattern", http://martinfowler.com/articles/injection.html, 15.09.2013.