Univerzitet u Beogradu
Matematički fakultet

Master rad

Alati za objektno–relaciono preslikavanje Entity Framework i Hibernate i njihovo poređenje

Mentor:
Prof. dr Nenad Mitić

Kandidat:
Ivana Duškić

Beograd, 2014.

Sadržaj

1.Uvod....2
2.Objektno–relaciono preslikavanje....2
3.Entity Framework....3
3.1.Arhitektura Entity Framework-a....3
3.2.Konfigurisanje Entity Framework-a....5
3.2.1. Database first....5
3.2.2. Model first....10
3.2.3. POCO....17
1.3.3. Delovi EDM modela....24
3.3.1. SSDL....25
3.3.2. CSDL....27
3.3.3. MSL....30
3.4.Entity Client Data Provider....31
3.5.Objektni servisi....32
3.6.LINQ to Entities....33
3.6.1. Upitna sintaksa....36
3.6.2. Metodna sintaksa....40
4.Hibernate....43
4.1.Arhitektura okvira Hibernate....43
4.1.1. Interfejs Session....44
4.1.2. Interfejs Transaction....48
4.1.3. Interfejs Query....49
4.1.4. Konfiguraciona klasa....50
4.1.5. Interfejsi Callback, UserType i CompositeUserType....50
4.2.Funkcije Hibernate-a 51
4.2.1. Keširanje....51
4.2.2. Kaskadno povezivanje....52
4.2.3. Lenjo učitavanje....52
4.3.Modularnost Hibernate-a....53
4.4.Konfiguracija okvira Hibernate....54
4.4.1. SessionFactory....55
4.5.Primer objektno-relacionog preslikavanja u Hibernate-u....56

4.6.Načini izražavanja upita u Hibernate-u..69
4.7.Hibernate upitni jezik HQL...70
4.8.Načini razvoja uz pomoć Hibernate-a..77
5.Poređenje Entity Framework-a i Hibernate-a...77
6.Zaključak...85
7.Literatura..85

1. Uvod

U okviru ovog rada biće prikazan koncept objektno-relacionog preslikavanja (*eng. Object Relation Mapping-ORM*) kao i alati koji se koriste za objektno-relaciono preslikavanje. Objektno-relacionim preslikavanjem se definiše prevođenje elemenata objektnog modela u elemente relacionog modela i obrnuto. Cilj objektno-relacionog preslikavanja je automatizovanje prenosa i konvertovanja podataka i struktura podataka.

Dizajn poslovnih aplikacija ima slojevitu arhitekturu softvera koja se najčešće sastoji od tri logička sloja: prezentacionog sloja, sloja poslovne logike i sloja podataka. Sloj podataka ima veliki značaj prilikom razvoja aplikacije. Implementiranje ovog sloja treba da pruži mogućinost preuzimanja podataka sa poslovnog sloja i smeštanje preuzetih podataka u bazu podataka i obrnuto, preuzimanje podataka iz baze i prosleđivanje višem sloju. Implementacija sloja podataka, koja se ostvaruje uspostavljanjem direktne komunikacije sa bazom podataka, ima nedostatke koji mogu da dovedu do složenijih izmena apliakcije. Ti nedostaci se ogledaju u zavisnosti aplikacije od konkretnog *DBMS*[1]-a, zavisnosti aplikacije od izmena u bazi podataka, itd. Rešenje ovih problema pružaju alati za objektno-relaciono preslikavanje koji vrše automatsko preslikavanje relacionog modela baze podataka na objektni model aplikacije. Oni omogućavaju da objektno-orijentisane aplikacije ne rade direktno sa tabelarnom reprezentacijom podataka već imaju svoj objektno-orijentisani model entiteta.

Ovaj rad je nastao kao rezultat znanja i iskustva stečenog na poslovnim projektima u kojima se koristio koncept objektno-relacionog preslikavanja. U prvom delu rada ukratko su navedeni razlozi potrebe za alatima koji se koriste za objektno-relaciono preslikavanje. U drugom delu opisane su osnovne karakteristike *Entity Framework*-a, alata za objektno–relaciono preslikavanje. Treći deo rada posvećen je opisu korišćenja alata *Hibernate* za objektno-relaciono preslikavanje. U četvrtom delu rada predstavljeno je poređenje ova dva alata prema različitim kriterijumima.

1 DBMS (*eng. Database Management System*) je sistem za upravljanje bazom podataka.

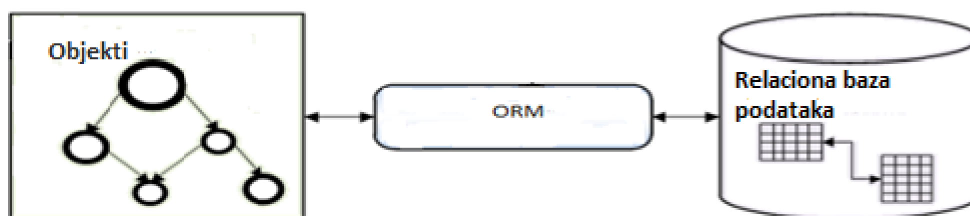
2. Objektno–relaciono preslikavanje

Danas mnoge objektno-orijentisane aplikacije koriste relacione baze za smeštanje podataka. Relacione baze podataka su pogodne za upotrebu zbog jednostavnosti njihovog formiranja, pristupa podacima u njima korišćenjem *SQL*-a, kao i jednostavnosti strukture modela podataka. Sistemi za upravljanje relacionim bazama podataka obezbeđuju konkurentan pristup podacima, njihovo ažuriranje, čuvanje integriteta podataka, kao i deljenje podataka od strane različitih korisnika i aplikacija.
Međutim, postoji problem u neskladu između objektne i relacione tehnologije. Njihova kombinacija može da dovede do konceptualnih i tehničkih problema. Ti problemi su poznati kao objektno–relaciono neslaganje (*eng. impedance mismatch*). Među najvažnije probleme spadaju:

1. neslaganje između objektne i relacione tehnologije - relacije predstavljaju činjenice o povezanosti podataka, dok objekti predstavljaju modele realnosti tj. sadrže i podatke
i ponašanje;
2. neslaganje pravila koja važe za jedinstvenost - dva reda u tabeli koji sadrže iste podatke smatraju se istim, tako da u bazu nije dozvoljeno unošenje dva ista reda, dok objekti koji sadrže iste podatke mogu biti različiti jer imaju različite memorijske adrese;
3. neslaganje hijerarhijskog uređenja podataka - među tabelama u relacionoj bazi ne postoji mogućnost izgradnje hijerarhije, dok se objekti mogu hijerarhijski urediti.

Da bi se prevazišli ti problemi, može se upotrebiti objektno–relaciono preslikavanje prilikom čuvanja objekata u relacionu bazu podataka.

Objektno-relaciono preslikavanje se realizuje na osnovu metapodataka koji opisuju preslikavanje između objekata i tabela baze podataka.



Slika 1: Objektno–relaciono preslikavanje

Da bi se olakšao i ubrzao proces objektno–relacionog preslikavanja, napravljen je niz alata. Ovi alati omogućavaju predstavljanje podataka relacione baze podataka pomoću objektno–orijentisanih koncepata u aplikaciji što je pogodnije za primenu poslovne logike. Cilj ovih alata je da smanje vreme potrebno za pravljenje aplikacije, naprave kod manje složenim i itljivijim, kao i da omoguće jednostavnije izmene aplikacije.

Većina rešenja za objektno–relaciono preslikavanje obuhvata četiri osnovna dela:

1. *API* koji izvršava *CRUD* (*create*, *read*, *update* i *delete*) operacije nad objektima;
2. Jezik ili *API* za zadavanje upita;

3. Tumač metapodataka za preslikavanje koji se koristi za opisivanje preslikavanja podataka iz objekata u relacije tj. povezivanje klasa sa tabelama baze podataka;
4. Modul pomoću kojeg *ORM* implementacija komunicira sa objektima i vrši funkcije koje se odnose na čuvanje i preuzimanje podataka.

U nastavku će biti predstavljena dva alata za objektno-relaciono preslikavanje, *Entity Framework* i *Hibernate*.

3. Entity Framework

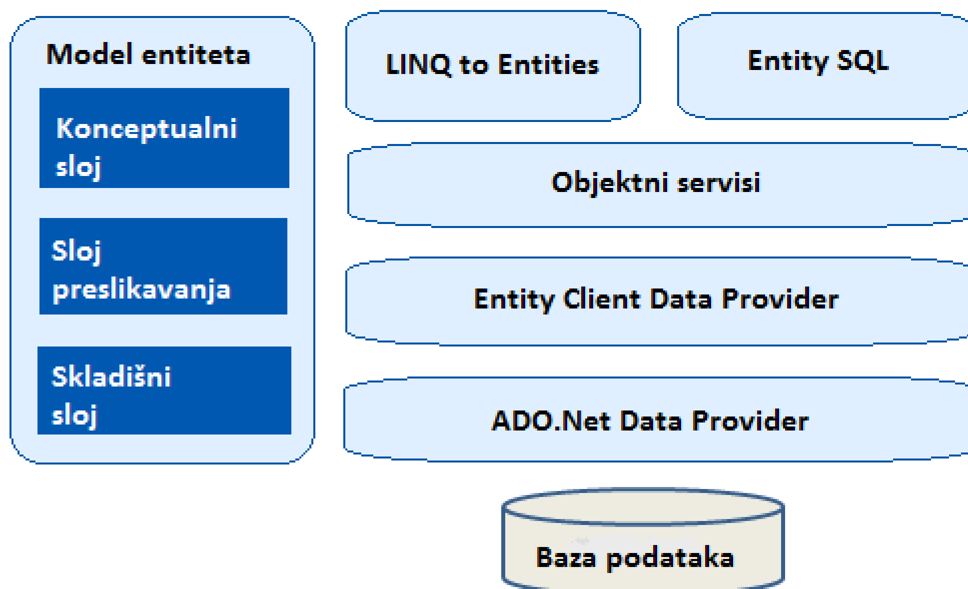
Entity Framework predstavlja *Microsoft*-ovu tehnologiju koja omogućava pristupanje relacionoj bazi podataka pomoću objektno-orijentisanih koncepata.

3.1. Arhitektura Entity Framework-a

U arhitekturi *Entity Framwork*-a razlikujemo:

1. Model entiteta (*eng. Entity Data Model (EDM)*) koji uspostavlja vezu između objektnog modela i baze podataka;
2. Objektne servise koji predstavljaju skup klasa koje se generišu i omogućavaju rad sa bazom podataka;
3. Sloj koji koristi sintaksu *.NET* programskih jezika za postavljanje upita nad bazom podataka.

Na slici 2 prikazana je arhitektura *Entity Framework*-a.



Slika 2: Arhitektura *Entity Framework*-a

EDM predstavlja jezgro *Entity Framework*-a. *Entity Framework* ga koristi za opisivanje svojih modela. *EDM* predstavlja *XML* datoteku sa ekstenzijom *.edmx* koja uspostavlja vezu između objektnog modela i baze podataka. U toj datoteci je definisano preslikavanje između objektnog modela i relacione baze podataka. Korišćenjem ove datoteke, *Entity Framework* pravi objektni model baze podataka što predstavlja objektne servise. Osnovni koncepti u *EDM*-u su entiteti (*eng. entity*) i asocijacije (*eng. assotiation*). Entiteti predstavljaju stvari ili objekte koji imaju identitet. Entiteti poseduju neka svojstva objekata:

- imaju tip i svojstva (*eng.properties*) koja mogu biti skalarne vrednosti ili reference na neke druge entitete;
- svaki entitet ima identitet (*eng. identity*).

Entiteti imaju sličnosti i sa podacima relacione baze:

- postoje unutar kolekcije;
- entitet ima veze (*eng. associations*) sa drugim entitetima;
- entiteti imaju primarne ključeve koji jedinstveno identifikuju entitet.

Asocijacije služe za povezivanje dva ili više entiteta.
EDM se sastoji iz tri sloja. To su:

1. Skladišni sloj (*eng. Storage layer*)
2. Konceptualni sloj (*eng. Conceptual layer*)
3. Sloj preslikavanja (*eng. Mapping layer*)

Ova tri sloja su definisana pomoću *XML* datoteka.
Skladišni sloj predstavlja šemu baze podataka i definisan je u *Store Schema Definition Language* (*SSDL*) *XML* datoteci.
Konceptualni sloj omogućava da se povezani podaci iz više tabela grupišu u objektno-orijentisane entitete, koji pružaju veću pogodnost za primenu poslovne logike. Konceptualni model je definisan u *Conceptual Schema Definition Language* (*CSDL*) *XML* datoteci. *CSDL* definiše entitete i veze na način na koji ih poznaje poslovni sloj aplikacije.
Sloj preslikavanja, koji se definiše korišćenjem *Mapping SchemaDefinition Language* (*MSDL*) *XML* datoteke, preslikava prethodna dva sloja jedan na drugi. Sloj za preslikavanje sadrži informacije o tome kako se konceptualni sloj preslikava u skladišni sloj i obrnuto.

3.2. Konfigurisanje Entity Framework-a

Da bi bilo moguće koristiti *Entity Framework* u projektu, potrebno je prvo konfigurisati taj projekat u *Visual Studio*-u, razvojnom i izvršnom okruženju za *.NET* tehnologije, što podrazumeva definisanje *EDM*-a, dodavanje projektnih referenci, definisanje parametara za povezivanje na bazu (*eng. connection string*). *Entity Framework* obezbeđuje tri načina za definisanje *EDM*-a:

1. *Database first* tj. definisanje *EDM*-a na osnovu baze podataka;
2. *Model first* tj. generisanje *EDM*–a, pa na osnovu njega generisanje baze podataka;
3. *Code only* tj .korišćenje *Plain Old CLR Objects* (*POCO*)[2] entiteta.

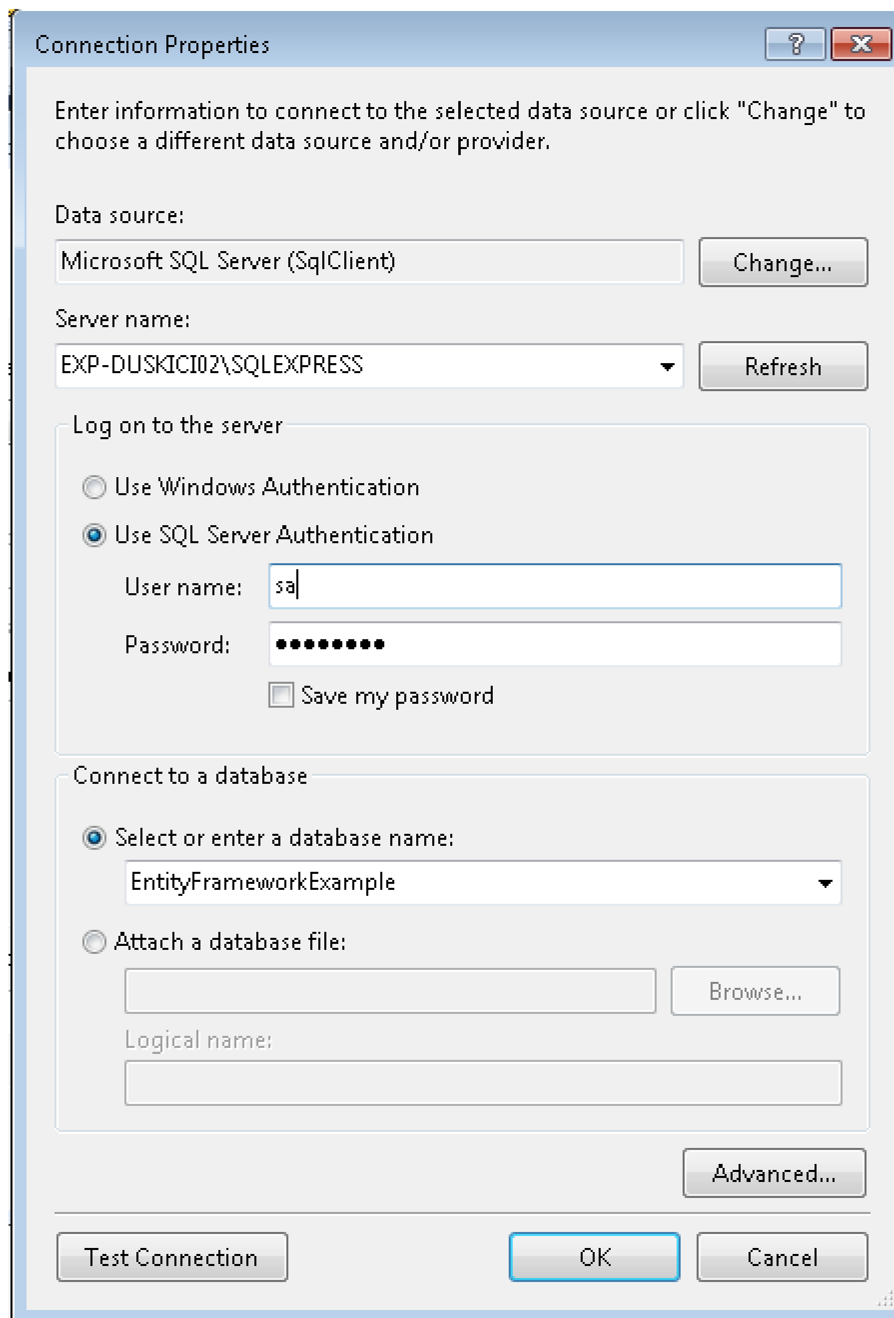
U nastavku će biti opisani načini definisanja *EDM*-a.

3.2.1. Database first

2 *Plain Old CLR Object* je klasa koja se sastoji isključivo od članova klase i odgovarajućih *get* i *set* metoda.

Konfiguracija će biti urađena korišćenjem alata *Entity Data Model Wizard*. Potrebno je otvoriti *Solution Explorer*, koji se nalazi u okviru *Visual Studio* okruženja, a zatim desnim klikom na projekat, izabrati *Add New Item* iz padajućeg menija. U listi instaliranih šablona (*eng. Templates*) sa leve strane dijaloga, izabrati nod *Data*. Pojaviće se lista svih *Data object* šablona, među njima i šablon *ADO.NET Entity Data Model* koji je potrebno izabrati.

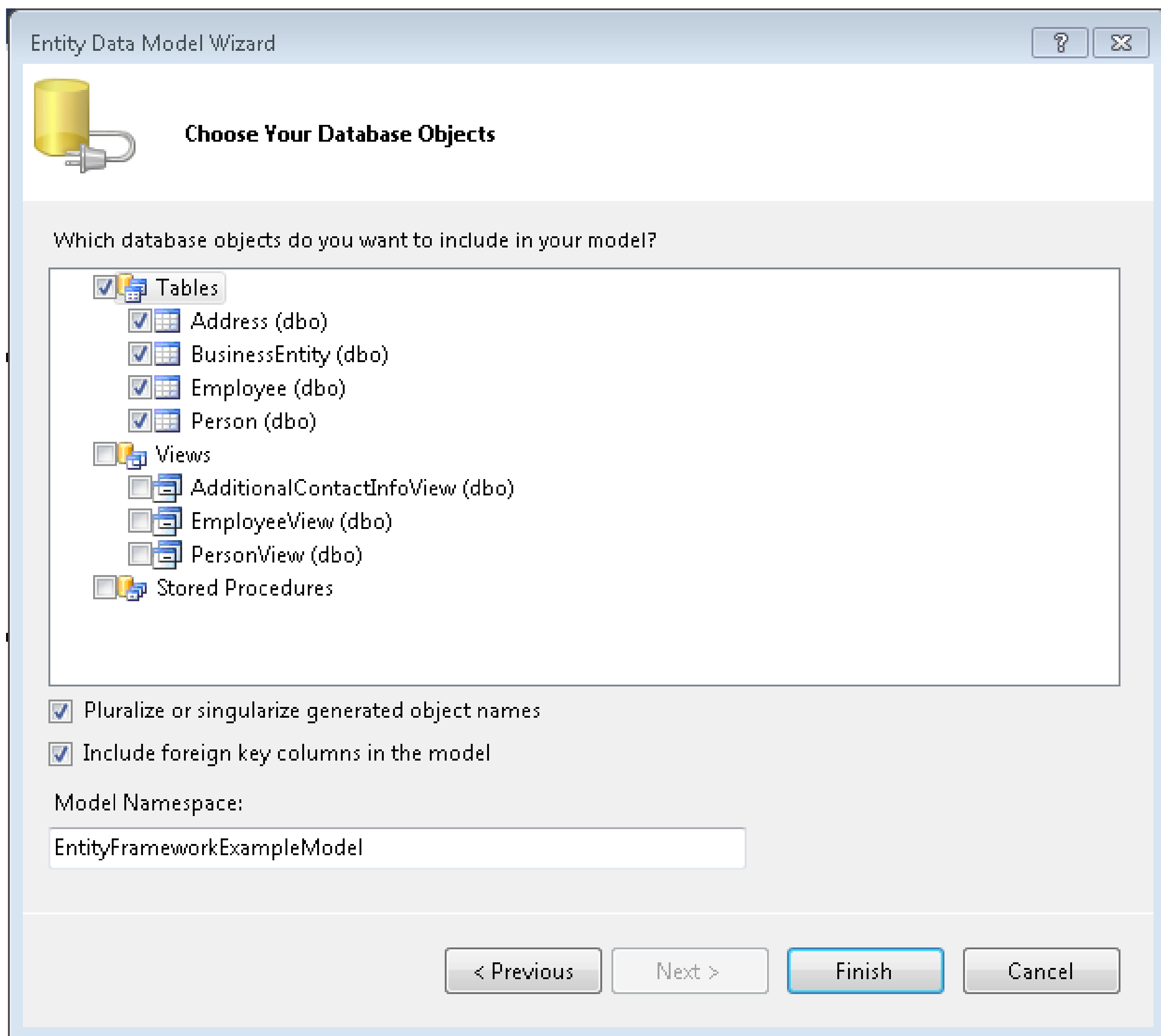
Nakon izbora šablona *ADO.NET Entity Data Model*, pojaviće se čarobnjak *Entity Data Model*. Potrebno je izabrati tip modela koji će se koristiti. U ovom koraku postoji opcija generisanja modela na osnovu baze podataka ili generisanje praznog modela. Prikazaćemo primer u kome se radi generisanje modela na osnovu baze podataka, pa je potrebno izabrati *Generate model from database*.
Sledeći korak omogućava da se odrede parametri za povezivanje na bazu podataka na osnovu koje se pravi model.
U ovom primeru izabraćemo testnu bazu podataka nazvanu *EntityFrameworkExample*.
Dijalog za izbor parametara za povezivanje na bazu podataka sadrži ime servera, ime baze i utvrđivanje identiteta korisnika koji pokušava da pristupi sistemu. Na slici 3 prikazan je dijalog za izbor parametara za povezivanje na bazu.



Slika 3: Izbor parametara za povezivanje na bazu podataka

Postoji opcija da se parametari za povezivanje na bazu sačuvaju u konfiguracionoj datoteci (*eng. App.config*) odakle će aplikacija moći da čita potrebne informacije o povezivanju na bazu. Ukoliko se ne izabere ova opcija, te informacije će trebati da se obezbede putem koda.

Sledeći korak omogu
ava izbor objekata baze koji će biti uključeni u *EDM*.



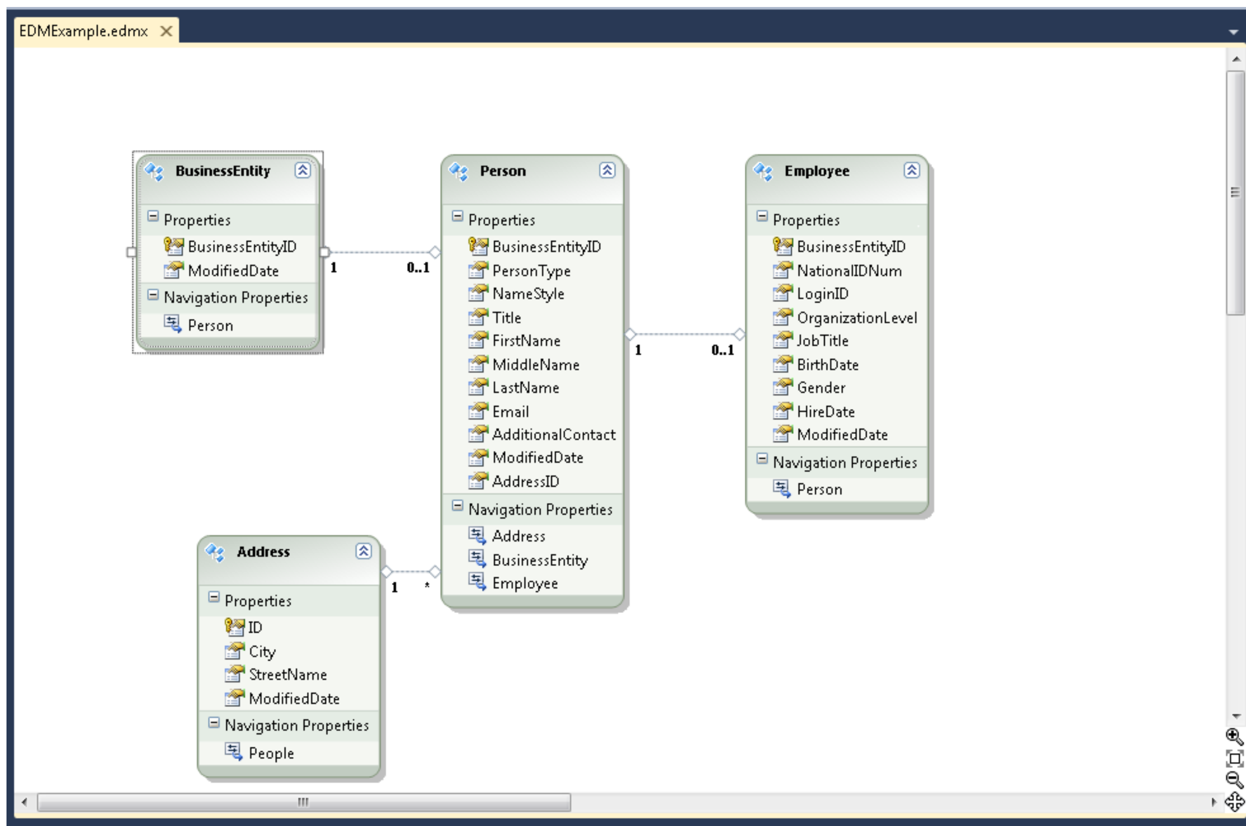
Slika 4: Izbor objekata koji će biti uključeni u *EDM*

U ovom primeru u model ćemo uključiti samo tabele. Moguće je uključiti još i poglede (*eng. views*) i uskladištene procedure (*eng. stored procedures*).

Opcija *Checkbox Pluralize or singularize generated object names* omogućava izbor pluralizacije ili singularizacije imena entiteta. To podrazumeva da će imena tipova entiteta biti u jednini, a imena *entityset*-ova u množini.

Opcija *Checkbox Include foreign key columns in the model* omogućava izbor generisanja svojstava na entitetskim tipovima koji odgovaraju kolonoma stranog ključa u bazi podataka.

Izborom dugmeta *Finish* na formi, dobija se:



Slika 5: Generisani *EDM*

Ovim je generisana datoteka *EDMX* koja definiše *EDM* i u sebi sadrži datoteke *CSDL*, *SSDL* i *MSL*.

Na osnovu *EDM*-a generisana je datoteka sa izvornim kodom za *C#* klase. U nastavku sledi primer generisane *C#* klase za entitet *BusinessEntity*:

```
[EdmEntityTypeAttribute(NamespaceName="EDMExample", Name="BusinessEntity")]
[Serializable()]
[DataContractAttribute(IsReference=true)]

//Generisana klasa koja odgovara entitetu BusinessEntity

public partial class BusinessEntity : EntityObject
{
   #region Factory Method

  //Metoda za pravljenje nove instance BusinessEntity

      public static BusinessEntity CreateBusinessEntity(global::System.Int32 businessEntityID,
      global::System.String modifiedDate)
   {
      BusinessEntity businessEntity = new BusinessEntity();
      businessEntity.BusinessEntityID = businessEntityID;
      businessEntity.ModifiedDate = modifiedDate;
      return businessEntity;
   }

   #endregion

   //definisanje svojstava entiteta
```

```
#region Primitive Properties

    //definisanje svojsvta BusinessEntityID; atribut EntityKeyProperty=true oznacava da je ovo
    svojstvo primarni ključ, dok atribut IsNullable označava da svojstvo ne može da ima vrednost
    null
[EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
[DataMemberAttribute()]
public global::System.Int32 BusinessEntityID
{
   get
   {
      return _BusinessEntityID;
   }
   set
   {
      if (_BusinessEntityID != value)
      {
         OnBusinessEntityIDChanging(value);
         ReportPropertyChanging("BusinessEntityID");
         _BusinessEntityID = StructuralObject.SetValidValue(value);
         ReportPropertyChanged("BusinessEntityID");
         OnBusinessEntityIDChanged();
      }
   }
}
private global::System.Int32 _BusinessEntityID;
partial void OnBusinessEntityIDChanging(global::System.Int32 value);
partial void OnBusinessEntityIDChanged();

#endregion


#region Navigation Properties

//definisanje navigacionog svojstva entiteta
[XmlIgnoreAttribute()]
[SoapIgnoreAttribute()]
[DataMemberAttribute()]
    [EdmRelationshipNavigationPropertyAttribute("EDMExample",   "BusinessEntityPerson",
    "Person")]
public Person Person
{
   get
   {
             return
            ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<Person>("E
            DMExample.BusinessEntityPerson", "Person").Value;
   }
   set
   {

            ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<Person>("E
            DMExample.BusinessEntityPerson", "Person").Value = value;
   }
}

[BrowsableAttribute(false)]
[DataMemberAttribute()]
public EntityReference<Person> PersonReference
{
   get
   {
```

```
                return
                ((IEntityWithRelationships)this).RelationshipManager.GetRelatedReference<Person>("E
                DMExample.BusinessEntityPerson", "Person");
            }
            set
            {
                if ((value != null))
                {

                        ((IEntityWithRelationships)this).RelationshipManager.InitializeRelatedReference
                        <Person>("EDMExample.BusinessEntityPerson", "Person", value);
                }
            }
        }

        #endregion
    }
```

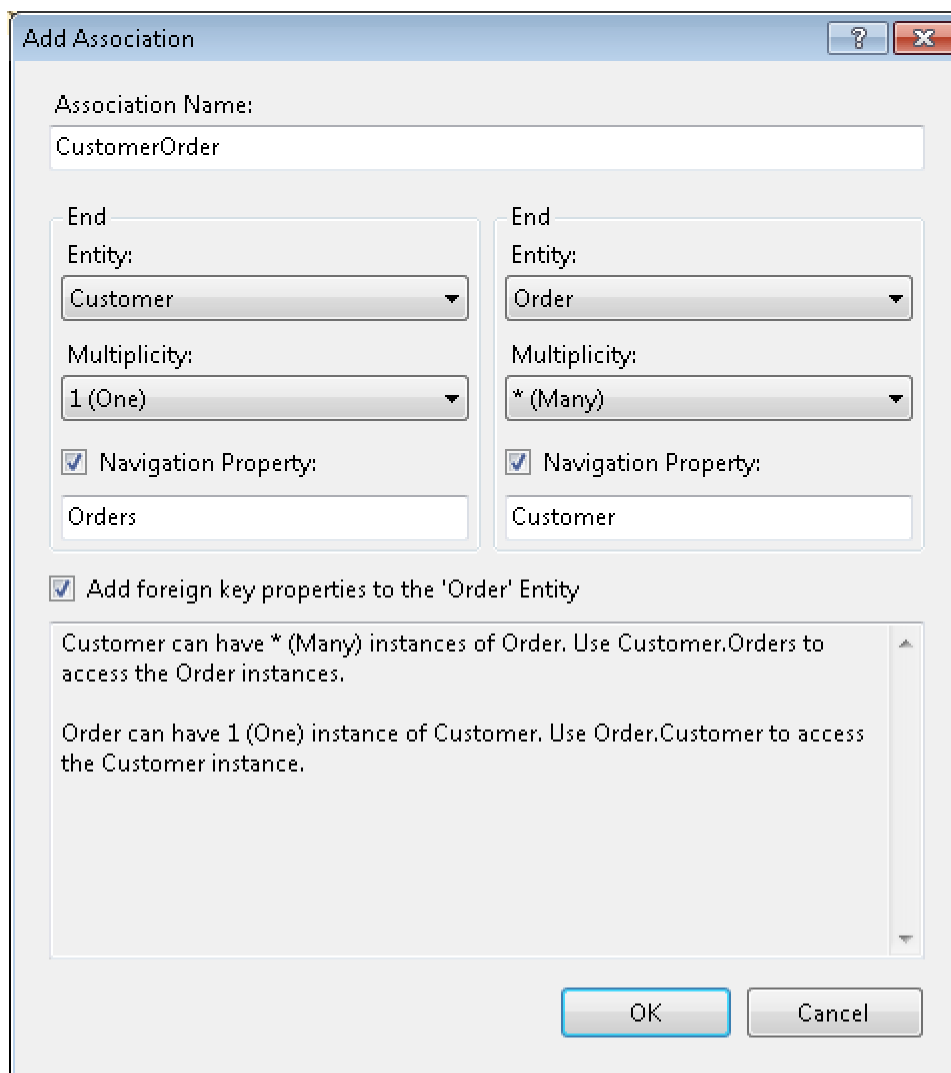
Pored datoteke sa *C#* klasama, u projekat su dodate reference na *System.Data.Entity*, *System.Runtime.Serialization*, *System.Security* i konfiguraciona datoteka.

3.2.2. Model first

Drugi pristup *EDM*-u omogućava pravljenje modela u dizajneru, a zatim generisanje baze podataka na osnovu tog modela. *Entity Framework* nudi grafički alat za pravljenje i izmenu *EDM*-a: *ADO.NET Entity Data Model Designer*. U ovom dizajneru možemo direktno videti konceptualni model, dodavati nove entitete, asocijacije i definisati nasleđivanje. Posebno je olakšano dodavanje novih objekata u model jer se oni mogu jednostavno prevući iz palete sa alatkama (*eng. Toolbox*).
Da bismo napravili prazan *EDM*, potrebno je pokrenuti čarobnjak *Entity Data Model* i izabrati obrazac *Empty model*. Izborom obrasca *Empty model* dobija se *Entity Data Model Designer*. Na dobijenom dizajneru postoji mogućnost izbora objekata i njihovog prevlačenja u model.
Razlikujemo tri tipa objekata:

1. Entitet - služi za pravljenje entiteta;
2. Asocijacija – omogućava pravljenje asocijacije između dva entiteta;
3. Nasleđivanje (*eng. Inheritance*) - omogućava uspostavljanje nasleđivanja između dva entiteta.

U dobijeni dizajner, prevlačenjem objekta *Entity* iz *Toolbox*–a, dodaćemo entitet *Customer* koji ima svojstvo (*eng. property*) *CustomerId*. *CustomerId* svojstvo je podešeno da bude primarni ključ za taj entitet. Postoji mogućnost postavke, kroz dizajner, nekog svojstva za primarni ključ tako što se vrednost svojstva *Entity key* postavi na *True*. Takođe, moguće je postaviti podrazumevanu vrednost za neko svojstvo, zatim izabrati da li to svojstvo može da ima *null* vrednost, kao i promeniti tip podatka. Dodatna svojstva se mogu dodati desnim klikom na entitet i izborom opcije *Add scalar property/complex property*. Složeno svojstvo (*eng. complex property*) predstavlja tip entiteta koji olakšava organizaciju skalarnih svojstava unutar entiteta.
Ovom entitetu dodaćemo još svojstava kao što su: *CustomerLastName* (tip: *String*), *CustomerAddress* (tip: *String*), *CustomerCity* (tip: *String*), *CustomerCode* (tip: *String*), *CustomerPhone* (tip: *String*), *ModifyDateTime* (tip: *DateTime*). Dodacemo još dva entiteta: *Item* i *Order*, a zatim među dodatim entitetima podesiti asocijacije. Dadavanje asocijacije se vrši izborom objekta *Association* iz *Toolbox*–a:



Slika 6: Dodavanje asocijacije u *EDM*

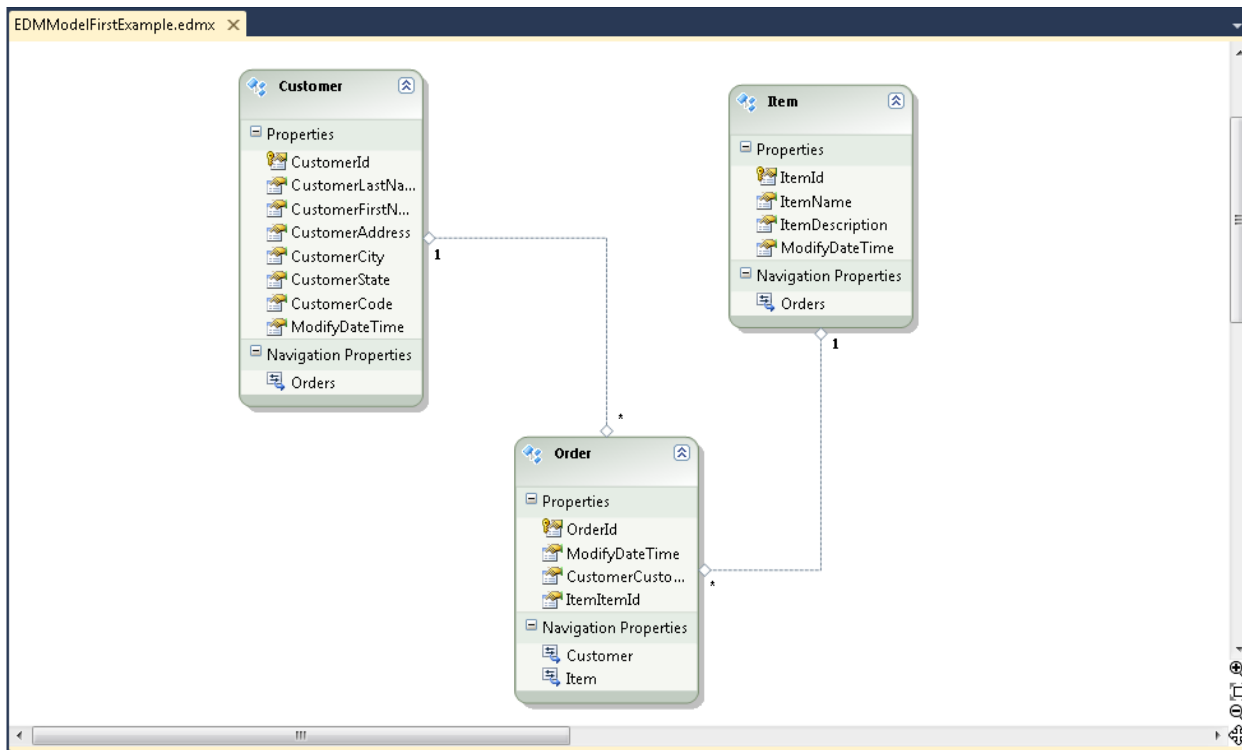
Prilikom pravljenja asocijacije između dva entiteta, postoji mogućnost odabira tipa veze između ta dva entiteta.

Mogući tipovi veza između tipova entiteta *X* i *Y* su:

- 1(*One*):1(*One*) ("jedan prema jedan") – Kod ovog tipa veze jedan entitet tipa *X* može biti pridružen najviše jednom entitetu tipa *Y* i jedan entitet tipa *Y* može biti pridružen najviše jednom entitetu tipa *X*;
- 1(*One*):*(*Many*) ("jedan prema više") – Ovaj tip veze omogućava da jedan entitet tipa *X* može biti pridružen većem broju entiteta tipa *Y*, dok jedan entitet tipa *Y* može biti pridružen najviše jednom entitetu tipa *X*;
- *(*Many*):*(*Many*) ("više prema više") – Kod ovog tipa veze jedan entitet tipa *X* može biti pridružen većem broju entiteta tipa *Y* i jedan entitet tipa *Y* može biti pridružen većem broju entiteta tipa *X*.

Napravićemo asocijaciju tipa 1(*One*):*(*Many*) između *Customer* i *Order* entiteta, i asocijaciju istog tipa između *Item* i *Order*.

Nakon tih podešavanja, dobija se model:



Slika 7: *EDM*

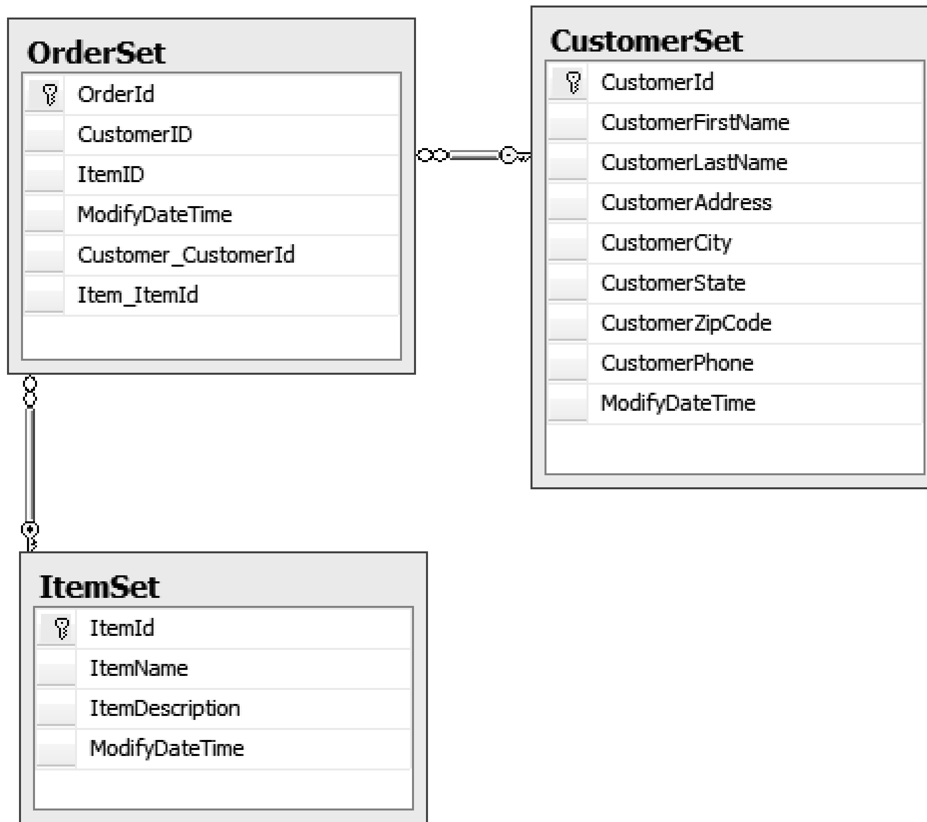
Na osnovu ovog modela, možemo napraviti šemu baze podataka izborom komande *Generate Database Script from Model*. Kao rezultat dobija se *SQL DDL* datoteka.

U nastavku sledi deo generisane *SQL DDL* datoteke koji se odnosi na generisanje šeme baze podataka za entitet *Customer*:

```
-- --------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, and Azure
-- --------------------------------------------------
-- Date Created: 09/03/2014 00:57:37
-- Generated from EDMX file: c:\users\ivana.duskic\documents\visual studio
2010\Projects\EDMModelFirstExample\EDMModelFirstExample\EDMModelFirstExample.edmx
-- --------------------------------------------------


SET QUOTED_IDENTIFIER OFF;
GO
USE [EntityFrameworkExample];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO
-- Generisanje tabela
-- --------------------------------------------------


-- Generisanje tabele 'Customers'
CREATE TABLE [dbo].[Customers] (
[CustomerId] int IDENTITY(1,1) NOT NULL,
[CustomerLastName] nvarchar(max)  NOT NULL,
[CustomerFirstName] nvarchar(max)  NOT NULL,
[CustomerAddress] nvarchar(max)  NOT NULL,
[CustomerCity] nvarchar(max)  NOT NULL,
[CustomerState] nvarchar(max)  NOT NULL,
[CustomerCode] nvarchar(max)  NOT NULL,
[ModifyDateTime] nvarchar(max)  NOT NULL
);
GO
```

```
-- --------------------------------------------------
-- Generisanje primarnih ključeva
-- --------------------------------------------------

-- Generisanje primarog ključa [CustomerId] u tabeli 'Customers'
ALTER TABLE [dbo].[Customers]
ADD CONSTRAINT [PK_Customers]
PRIMARY KEY CLUSTERED ([CustomerId] ASC);
GO

-- --------------------------------------------------
-- Generisanje stranih ključeva
-- --------------------------------------------------

-- Generisanje stranog ključa [CustomerCustomerId] u tabeli 'Orders'
ALTER TABLE [dbo].[Orders]
ADD CONSTRAINT [FK_CustomerOrder]
FOREIGN KEY ([CustomerCustomerId])
REFERENCES [dbo].[Customers]
 ([CustomerId])
ON DELETE NO ACTION ON UPDATE NO ACTION;

-- Generisanje neklasterovanog indeksa za strani ključ 'FK_CustomerOrder'
CREATE INDEX [IX_FK_CustomerOrder]
ON [dbo].[Orders]
([CustomerCustomerId]);
GO

-- --------------------------------------------------
-- Script has ended
-- --------------------------------------------------
```

Dobijena *SQL DDL* datoteka
e generisati šemu baze podataka prikazanu na slici 8.



Slika 8: Generisana šema baze podataka

Pored generisane baze podakaka, *Entity framework* automatski generiše skup klasa koje odgovaraju entitetima koji učestvuju u modelu. U nastavku sledi primer klase *Customer* koja odgovara entitetu *Customer* definisanom u *EDM*-u.

```
//Klasa Customer koja odgovara entitety Customer
[EdmEntityTypeAttribute(NamespaceName="EDMModelFirstExample", Name="Customer")]
[Serializable()]
[DataContractAttribute(IsReference=true)]
public partial class Customer : EntityObject
{
    #region Factory Method
   //metoda za pravljenje nove instance klase na osnovu svih svojstava
        public static Customer CreateCustomer(global::System.Int32 customerId,
        global::System.String customerLastName, global::System.String customerFirstName,
        global::System.String customerAddress, global::System.String customerCity,
        global::System.String customerState, global::System.String customerCode,
        global::System.String modifyDateTime)
    {
        Customer customer = new Customer();
        customer.CustomerId = customerId;
        customer.CustomerLastName = customerLastName;
        customer.CustomerFirstName = customerFirstName;
        customer.CustomerAddress = customerAddress;
        customer.CustomerCity = customerCity;
        customer.CustomerState = customerState;
        customer.CustomerCode = customerCode;
        customer.ModifyDateTime = modifyDateTime;
        return customer;
    }

    #endregion
```

```
    #region Primitive Properties

        //definisanje svojstva koje predstavlja primarni ključ i ne sme da ima null     vrednost
    [EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
    [DataMemberAttribute()]
    public global::System.Int32 CustomerId
    {
        get
        {
            return _CustomerId;
        }
        set
        {
            if (_CustomerId != value)
            {
                OnCustomerIdChanging(value);
                ReportPropertyChanging("CustomerId");
                _CustomerId = StructuralObject.SetValidValue(value);
                ReportPropertyChanged("CustomerId");
                OnCustomerIdChanged();
            }
        }
    }
    private global::System.Int32 _CustomerId;
    partial void OnCustomerIdChanging(global::System.Int32 value);
    partial void OnCustomerIdChanged();

    #endregion


    #region Navigation Properties

   //definisanje navigacionih svojstava
    [XmlIgnoreAttribute()]
    [SoapIgnoreAttribute()]
    [DataMemberAttribute()]
        [EdmRelationshipNavigationPropertyAttribute("EDMModelFirstExample", "CustomerOrder",
       "Order")]
    public EntityCollection<Order> Orders
    {
        get
        {
                        return
                ((IEntityWithRelationships)this).RelationshipManager.GetRelatedCollection<Order>("ED
                MModelFirstExample.CustomerOrder", "Order");
        }
        set
        {
            if ((value != null))
            {

                        ((IEntityWithRelationships)this).RelationshipManager.InitializeRelatedCollection
                        <Order>("EDMModelFirstExample.CustomerOrder", "Order", value);
            }
        }
    }


    #endregion
}
```

3.2.3. POCO

Do sada smo govorili o dva načina generisanja *EDM*-a: *database first* i *model first*. Postoji i treći način koji omogućava programerima da naprave svoj model korišćenjem *POCO* klasa. Ovaj pristup zahteva da se naprave *POCO* klase koje sadrže istu strukturu kao šema baze podataka sa kojom se uspostavlja preslikavanje. On omogućava potpunu odvojenost modela aplikacije definisanog pomoću *POCO* objekata od *Entity Framework*-a i ostalih delova aplikacije. *POCO* klase implementiraju atribute koji odgovaraju kolonama tabele, metode *get* i *set* koje se koriste za pristupanje atributima, konstruktor bez argumenata, konstruktor sa atributima koji odgovaraju kolonama tabele koje ne mogu da sadrže nedefinisane vrednosti, konstruktor sa svim atributima, atribut koji predstavlja skup objekata druge klase, interfejs *java.io.Serializable*, koji je potreban da bi se objekti mogli koristiti u sesijama.
Da bi se koristio *POCO* pristup, nakon sto se napravi *EDM*, potrebno je isključiti generator koda. Time se neće generisati klase na osnovu *EDM*-a već će biti potrebno napraviti sopstvene klase.
U nastavku sledi primer korišćenja *POCO* pristupa.
Najpre treba napraviti odgovarajuće *POCO* klase.

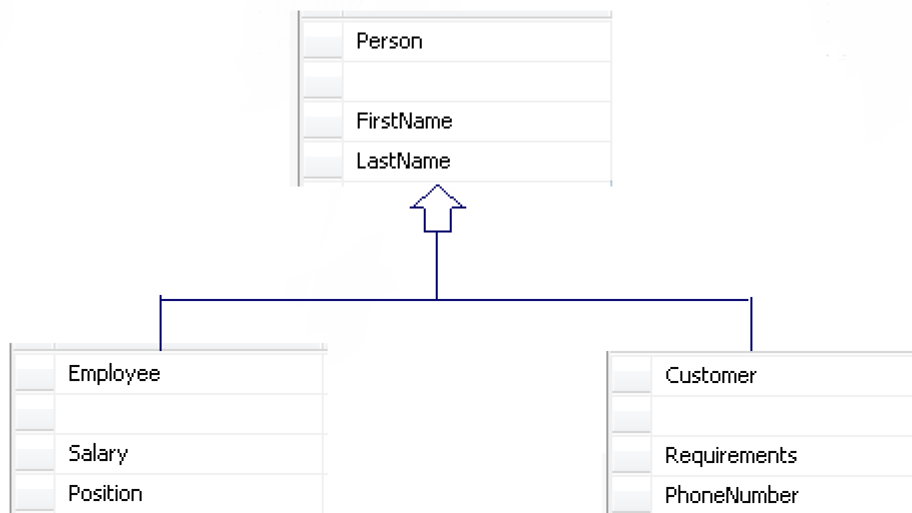
```
public class Person
{
        public Person() { }
        public int PersonID { get; set; }
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string Phone { get; set; }
        public ICollection<Employee> Employees { get; set; }
}

public class Employee
{
        public Employee() { }
        public int EmployeeID { get; set; }
        public string Position { get; set; }
        public Contact Contact { get; set; }
        public long Salary {get; set;}
}
```

S obzirom da je generator koda isključen, klasa *ObjectContext* neće biti generisana, tako da je potrebno napraviti klasu koja nasleđuje klasu *ObjectContext*. Ova klasa je potrebna za integraciju *POCO* klasa sa *ObjectContext*–om. Ta klasa se koristi da opiše oblik modela koji pravimo, zajedno sa mehanizmom za pristup *POCO* klasama.

```
public class POCOContext : ObjectContext //Klasa ObjectContext, upravlja POCO klasama
{
        private ObjectSet<Person> people;      //Deklarisanje ObjectSet svojstava
        private ObjectSet<Employee> employees;

        public POCOContext("name =”EntityFrameworkExampleEntities",”
EntityFrameworkExampleModelStoreContainer”)
      : base(connection)
        {
                people = CreateObjectSet<Person>(); //Nova instanca ObjectSet<TEntity>
        koja se koristi za pravljenje upita, dodavanje, izmenu i brisanje objekata navedenog
        tipa
                employees = CreateObjectSet<Employee>();
        }
```

```
public ObjectSet<Person> Person
{
        get
        {
            return people;
        }
}

public ObjectSet<Employee> Employee
{
        get
        {
            return employee;
        }
}}
```

Ukoliko među klasama postoji hijerarhijsko uređenje, potrebno je izabrati strategiju preslikavanja te hijerarhije klasa u tabele baze podataka. *Entity Framework* podržava tri pristupa preslikavanja hijerarhije klasa. To su: preslikavanje hijerarhije klasa u jednu tabelu (*eng. table per type hierarchy*), preslikavanje svake konkretne neapstraktne klase u određenu tabelu (*eng. table per concrete type*), preslikavanje svake klase/potklase (uključujući i apstraktne klase) u određenu tabelu (*eng. table per type*). U nastavku slede primeri preslikavanja hijerarhije klasa prikazane na slici 9 korišćenjem svake od navedenih strategija.



Slika 9: Hijerarhija klasa

- Primer preslikavanja hijerarhije klasa u jednu tabelu

  Hijerarhija klasa biće preslikana u tabelu pod nazivom *PERSON*. Da bi se izvršilo preslikavanje u tabelu *PERSON*, potrebno je dodati kolonu koja će predstavljati identifikatora (*PersonId*) i kolonu koja će određivati tip (*PersonType*). Kolona *PersonType* je potrebna da bi se odredio tip objekta koji je potrebno instancirati. Pored tih kolona, u tabelu je potrebno dodati i kolone koje odgovaraju poljima klase *Person* i poljima njenih potklasa. Na slici 10 prikazana je struktura tabele *PERSON*:

Column Name
PersonId
PersonType
FirstName
LastName
Salary
Requirements

Slika 10: Struktura tabele *PERSON*

- Primer preslikavanja svake konkretne neapstraktne klase u određenu tabelu

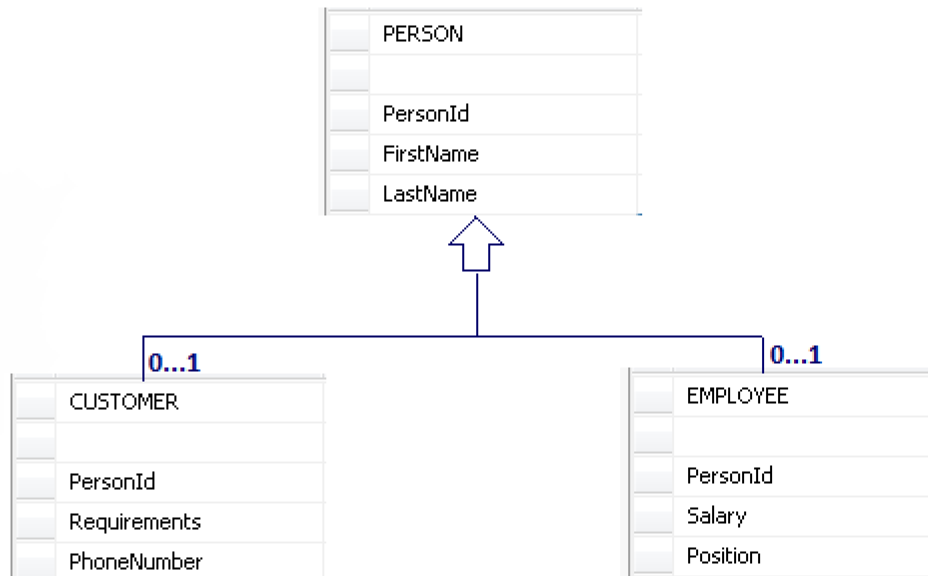
  Ovim pristupom pravi se jedna tabela za svaku klasu, pri čemu svaka tabela uključuje i atribute klase i one koji su nasleđeni. Na slici 11 prikazan je model podataka za hijerahiju klasa predstavljenu na slici 9 primenom ove strategije. Za klase *Customer* i *Employee* napravljene su posebne tabele, dok za klasu *Person*, koja je apstraktna, ne postoji posebna tabela. Svaka tabela ima svoj identifikator, polja klase *Person*, kao i polja specifična za svaku od klasa.

Column Name
CustomerId
FirstName
LastName
Requirements
PhoneNumber

Column Name
EmployeeId
FirstName
LastName
Salary
Position

Slika 11: Struktura tabela *CUSTOMER* i *EMPLOYEE*

- Primer preslikavanja svake klase/potklase (uključujući i apstraktne klase) u određenu tabelu (*eng. table per type*)
  Ovim pristupom pravi se tabela za svaku klasu sa po jednom kolonom za svako polje klase i kolonom za kontrolu istovremenog pristupa. Na slici 12 prikazana je struktura tabela dobijenih preslikavanjem hijerarhije klasa predstavljene na slici 9 strategijom preslikavanja svake klase/potklase u određenu tabelu. Podaci za klasu *Customer* sačuvani su u dve tabele, *CUSTOMER* i *PERSON*, tako da je za dobijanje svih podataka za objekat tipa *Customer* potrebno izvršiti povezivanje ovih tabela. Za razliku od prethodne strategije, ovde postoji razlika u korišćenju primarnih ključeva za sve tabele. Koristi se isti primarni ključ za sve tabele, koji je u potklasama i primarni i strani ključ. Pomoću njega tabele koje odgovaraju potklasama klase *Person* održavaju vezu sa tabelom koja odgovara apstraktnoj klasi *Person*.



Slika 12: Struktura tabela *PERSON*, *CUSTOMER* i *EMPLOYEE*

Pored preslikavanja svojstava i nasleđivanja, potrebno je realizovati i preslikavanje asocijacija. U *Entity Framework*-u razlikujemo: 1-1, * - * i 1 - * asocijacije. Asocijacije mogu da budu jednosmerne (*eng. unidirectional*) i dvosmerne (*eng. bidirectional*). Pod jednosmernom asocijacijom između dva objekta podrazumeva se veza kod koje samo jedan od objekata ima referencu na drugi objekat u vezi. Dvosmerna asocijacija podrazumeva postojanje reference kod oba objekta na objekat koji se nalazi na drugoj strani veze.

U nastavku slede primeri preslikavanja asocijacija korišćenjem anotacija (*eng. DataAnnotations)*. Anotacije predstavljaju jedan od standarda pomoću kog se vrši povezivanje *POCO* objekata sa tabelama baze podataka. Pored ovog standarda, *Entity Framework Code First* pristup obezbeđuje i konfigurisanje korišćenjem *Fluent API* standarda. Anotacije su opisni atributi koji se definišu uz klasu ili svojstvo klase koje je potrebno konfigurisati. *Fluent API* za konfiguraciju koristi posebne konfiguracione objekte definisane u posebnoj biblioteci. .

- Primer preslikavanja veze tipa 1-1 između entiteta *Employee* i *EmployeeAddress* korišćenjem anotacija:
  Veza 1-1 se ostvaruje definisanjem primarnog i stranog ključa u odgovarajućim tabeleama. U ovom primeru *EmployeeId* je primarni ključ u tabeli *Employee*, dok je u tabeli *EmployeeAddress* strani ključ.

```
public class Employee
{
        public Employee() { }
        public int EmployeeID { get; set; }
        public string Position { get; set; }
        public long Salary {get; set;}
        [Required]
        public virtual EmployeeAddress EmployeeAddress { get; set; }
}

public class EmployeeAddress
{
        [Key, ForeignKey("Employee")]
        public int EmployeeID { get; set; }
        public string Address { get; set; }
```

```
        public string State { get; set; }
        public string Country { get; set; }

        public virtual Employee Employee { get; set; }
    }
```

Anotacija *[Key, ForeignKey("Employee")]* se korsiti za označavanje polja koje predstavlja strani ključ.

- Primer preslikavanja asocijacije tipa 1-* između entiteta *Team* i *Employee* korišćenjem anotacija:

```
    public class Employee
    {
        public Employee() { }
        public int EmployeeID { get; set; }
        public string Position { get; set; }
        public long Salary {get; set;}
        [Required]
        public virtual Team Team { get; set; }
  }



    public class Team
    {
        public Team()
        {
            EmployeeList=new List<Employee>();
        }
        public int TeamID { get; set; }
        public string TeamName { get; set; }
        public string TeamDescription {get; set;}
        public virtual ICollection<Employee> Employees { get; set; }
    }
```

U klasi *Employee* potrebno je definisati svojstvo tipa klase *Team*, dok klasa *Team* treba da sadrži kolekciju objekata tipa *Employee*.

- Primer preslikavanja asocijacije tipa *-*:
  Da bi se realizovalo preslikavanje asocijacije *-* između klasa *Employee* i *Course*, klasa *Employee* treba da sadrži svojstvo koje predstavlja kolekciju objekata tipa *Course*, dok klasa *Course* treba da sadrži svojstvo koje predstavlja kolekciju objekata tipa *Employee*.

```
    public class Employee
    {
        public Employee() { }
        public int EmployeeID { get; set; }
        [Required]
        public virtual ICollection<Course> Courses { get; set; }
}



    public class Course
    {
        public Course()
        {
            this.Employees = new HashSet<Employee>();
        }
        public int CourseID { get; set; }
        public string CourseName { get; set; }
```

```
                public string TeamDescription {get; set;}
                public virtual ICollection<Employee> Employees { get; set; }
        }
```

Svaka klasa entiteta sadrži atribut koji predstavlja jedinstveni identifikator, preslikava se u primarni ključ tabele baze podataka i omogućava izdvajanje konkrene instance klase entiteta. Ovaj ključ može biti prost ili složen. Prosti ključevi su atributi klase entiteta, dok je prilikom preslikavanja složenih ključeva neophodno navesti redosled svojstava koja čine ključ. U nastavku sledi primer preslikavanja složenog ključa korišćenjem anotacija:

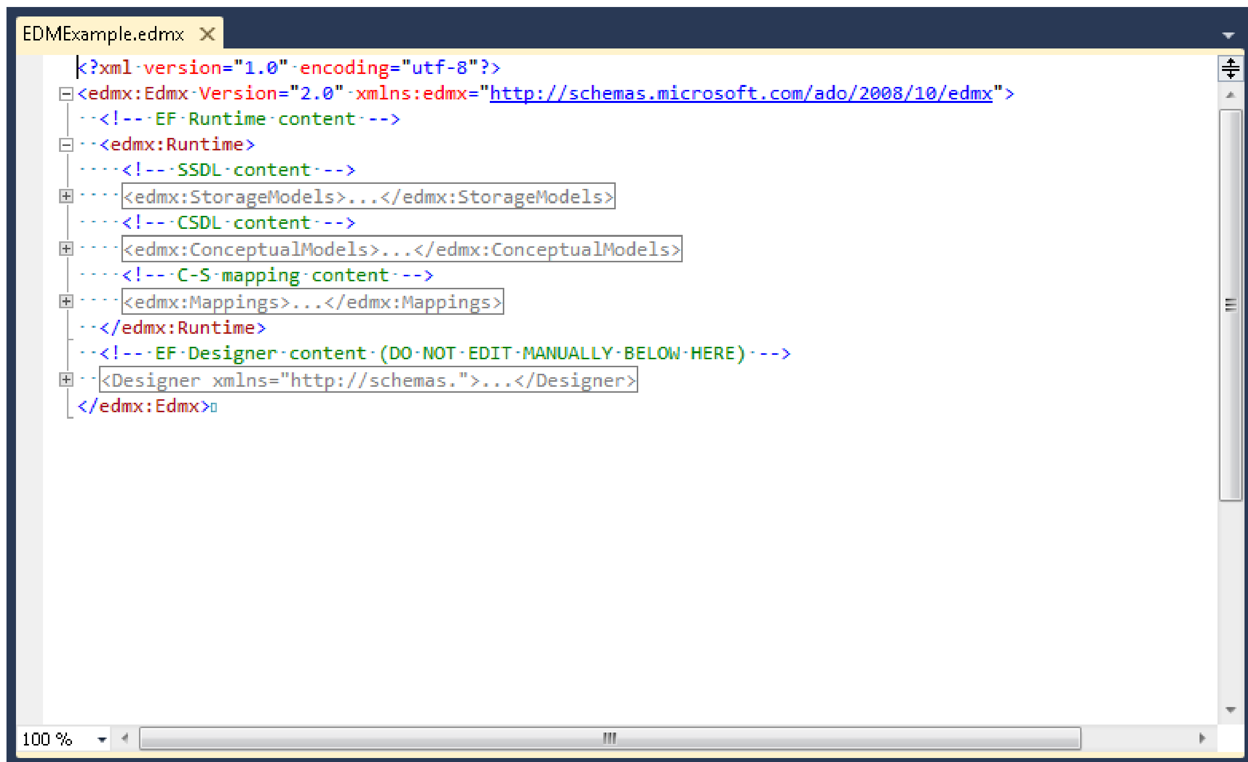
```
public class Person
{
        public Person() { }
        [Key]
        [Column (Order = 1)]
        public int PersonID { get; set; }
        [Key]
        [Column (Order = 2)]
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string Phone { get; set; }
}
```

Složeni ključ iz ovog primer se sastoji od dva svojstva: *PersonID* i *FirstName*.

1. 3.3. Delovi EDM modela

EDM sačinjavaju tri *XML* datoteke. Svaka od tih datoteka definiše deo modela i baze. *XML* datoteke koje čine *EDM* su:

1. *SSDL* (*eng. Storage Schema Definition Language*) poznata kao skladišni model;
2. *CSDL* (*eng. Conceptual Schema Definition Language*) poznata kao konceptualni model;
3. *MSL* (*eng. Mapping SchemaDefinition Language*) poznata kao sloj za preslikavanje.

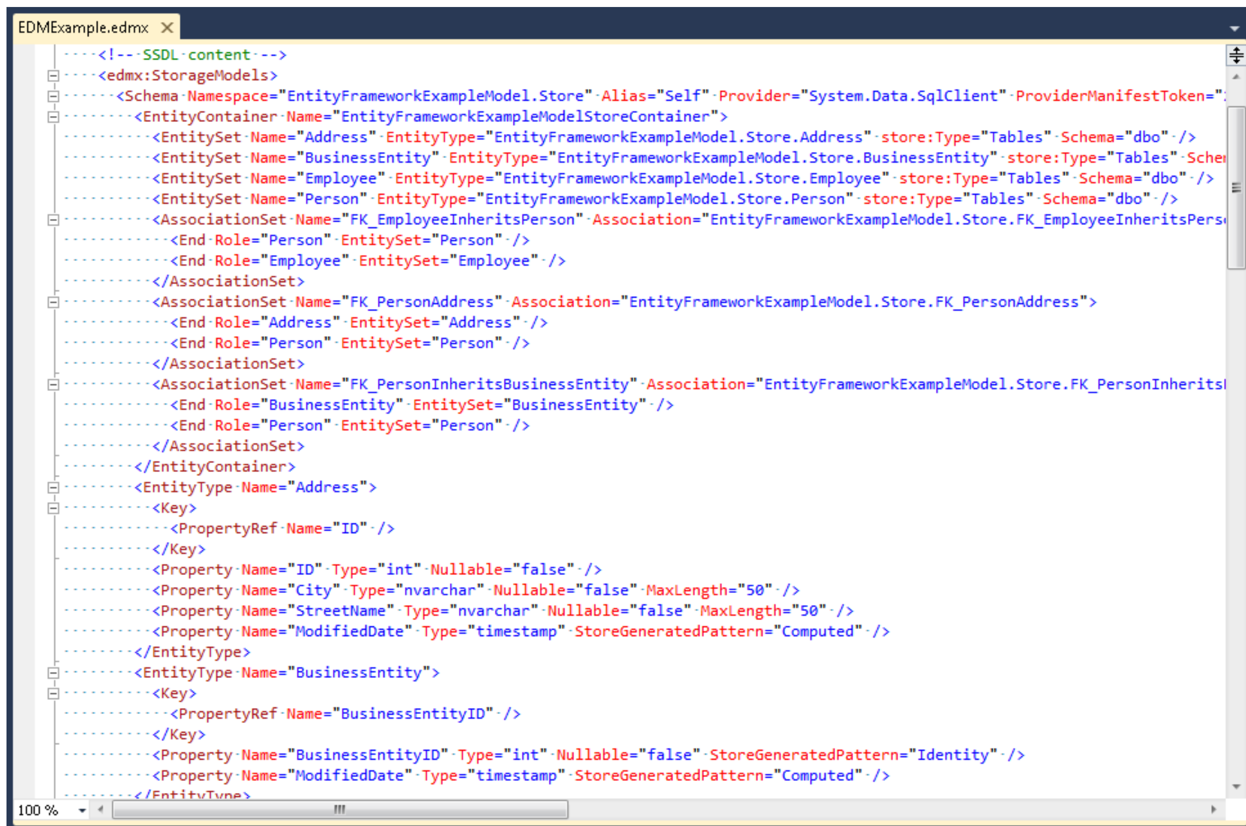


Slika 13: Delovi *EDM* - a

Svaki od ovih delova *EDM*-a će biti opisan u nastavku rada.

3.3.1. SSDL

SSDL je *XML* šematski prikaz preslikane baze podataka tj. *SSDL* opisuje tabele iz baze podataka, poglede, uskladištene procedure, itd. U primeru koji sledi dat je prikaz *EntityFrameworkExample* baze podataka.



Slika 14: *SSDL*

Prostor imena *SSDL* datoteke se bazira na imenu baze podataka koja se koristi. Element *EntityContainer* dobija naziv po šemi baze podataka. *EntityContainer* sadrži niz *EntitySet* i *AssociationSet* elemenata koji deklarišu instance tabela i veza predstavljenih kao *EntityType* i *Association*.

Primer <*EntitySet*>-a generisanog na osnovu tabele *Person*:

```
<EntitySet Name="Person" EntityType="EntityFrameworkExampleModel.Store.Person"
store:Type="Tables" Schema="dbo" />
```

Primer <*EntityType*> definisanog za *Person*:

```
<EntityType Name="Person">
<Key>
<PropertyRef Name="BusinessEntityID" />
</Key>
<Property Name="BusinessEntityID" Type="int" Nullable="false" />
<Property Name="PersonType" Type="nvarchar" Nullable="false" MaxLength="50" />
<Property Name="Title" Type="nvarchar" MaxLength="50" />
<Property Name="FirstName" Type="nvarchar" MaxLength="50" />
<Property Name="MiddleName" Type="nvarchar" MaxLength="50" />
<Property Name="LastName" Type="nvarchar" MaxLength="50" />
<Property Name="Email" Type="nvarchar" MaxLength="50" />
<Property Name="AdditionalContact" Type="nvarchar" MaxLength="50" />
<Property Name="ModifiedDate" Type="datetime" Nullable="false" />
<Property Name="AddressID" Type="int" Nullable="false" />
</EntityType>
```

Element *<Key>* određuje koja svojstva čine ključ tabele. Element *<Property>* predstavlja kolonu u tabeli baze podataka.
Element *<AssociationSet>* predstavlja strani ključ između dva *<EntitySet>*-a. Element *<Association>* u *SSDL*-u precizira kolone tabele koje u
estvuju u stranom klju
u. Dva potrebna elementa *<End>* u okviru ovog elementa određuju tabele koje učestvuju u asocijaciji.
AssociationSet za *Person* i *Employee* definisan je delom koda:

```
<AssociationSet Name="FK_EmployeeInheritsPerson"
Association="EntityFrameworkExampleModel.Store.FK_EmployeeInheritsPerson">
<End Role="Person" EntitySet="Person" />
<End Role="Employee" EntitySet="Employee" />
</AssociationSet>
```

Primer *Association*-a:

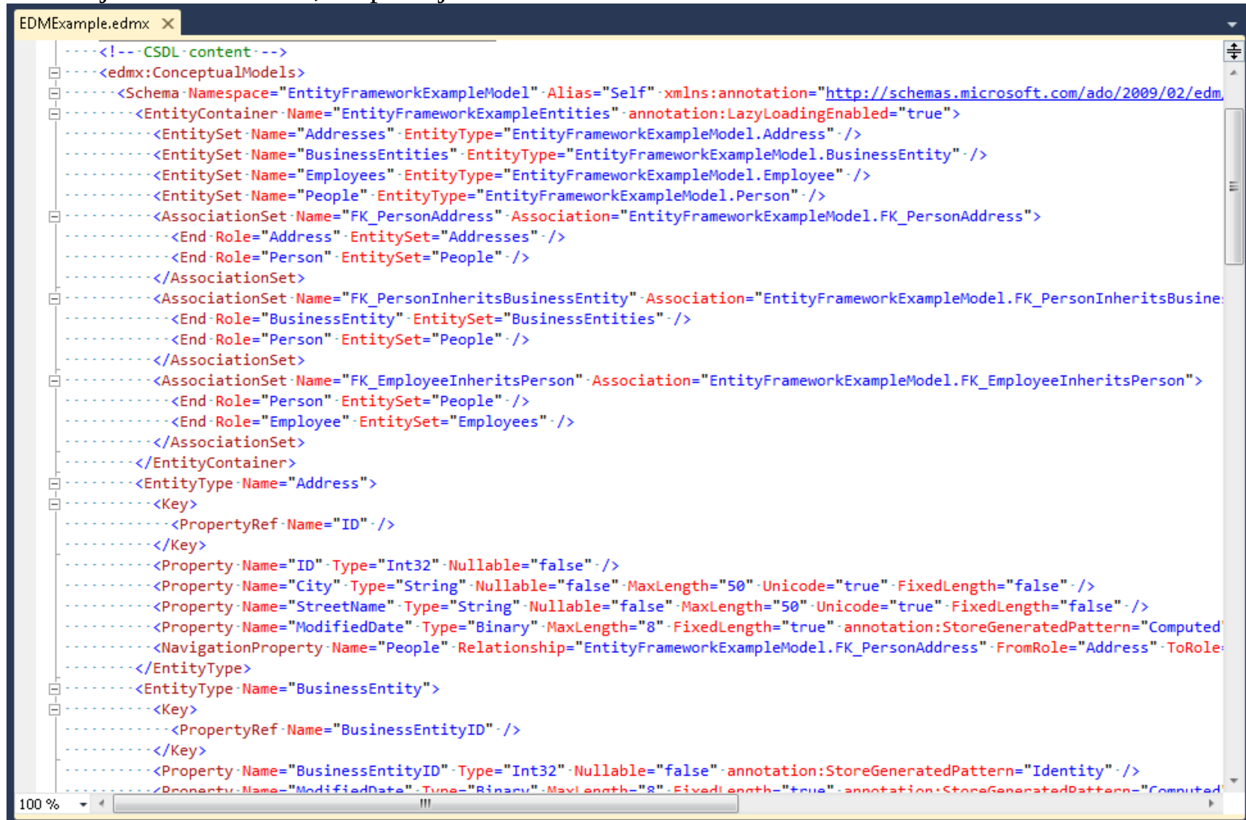
```
<Association Name="FK_EmployeeInheritsPerson">
<End Role="Person" Type="EntityFrameworkExampleModel.Store.Person" Multiplicity="1" />
<End Role="Employee" Type="EntityFrameworkExampleModel.Store.Employee"
Multiplicity="0..1" />
<ReferentialConstraint>
<Principal Role="Person">
<PropertyRef Name="BusinessEntityID" />
</Principal>
<Dependent Role="Employee">
<PropertyRef Name="BusinessEntityID" />
</Dependent>
</ReferentialConstraint>
</Association>
```

Opcioni element *<ReferentialConstraint>* predstavlja primarne i sekundarne elemente asocijacije kao i kolone koje učestvuju u ključu. Primarna tabela je ona u kojoj je kolona navedena u okviru elementa *<PropertyRef>* primarni ključ, a sekundarna tabela je ona u kojoj je ta kolona strani ključ. Ukoliko ne postoji element *<ReferentialConstratint>*, potrebno je koristiti element *<AssociationSetMapping>* za specifikaciju kolona koje se koriste u asocijaciji. Element *<Principal>* određuje tabelu koja je primarna u asocijaciji, dok element *<Dependent>* određuje tabelu koja je sekundarna u asocijaciji.

3.3.2. CSDL

Conceptual Schema Definition Language (*CSDL*) definiše kako se mogu slati upiti na model sa gledišta aplikacije tj. definiše klase koje predstavljaju bazu.
Metapodaci koji se nalaze u datoteci *CSDL* sadrže listu entiteta i asocijacija između njih. Za predstavljanje tipova tih entiteta koristi se element *<EntityType>*, a tipovi asocijacija predstavljeni su korišćenjem elementa *<Association>*. Entiteti mogu da sadrže jedan ili više atributa i članova koji opisuju strukturu entiteta. Ti članovi mogu biti skalarnog ili složenog tipa. Skalarni tipovi su npr. *integer*, *string*, dok su složeni tipovi sastavljeni od više skalarnih ili složenih tipova. Da bi se neki član predstavio kao ključ za određeni entitet koristi se element *<Key>*. Složeni ključevi su prikazani razdvajanjem imena svakog člana razmakom. Da bi se definisalo kako se vrši navigacija sa jednog entiteta na drugi entitet sledeći asocijaciju, koristi se tip člana nazvan navigaciono svojstvo (*eng. Navigation property*).

CSDL je sličan *SSDL*-u, ali postoje razlike u elementima i razlike u svrsi tih elemenata.



Slika 15: *CSDL*

Elementi <*EntityType*> služe za definisanje tipova domenskih entiteta. <*EntitySet*> predstavlja kolekciju entiteta u kojoj svi entiteti moraju biti istog (ili izvedenog) tipa. Jedan *EntitySet* je konceptualno sličan tabeli u bazi, dok je entitet sličan vrsti u tabeli. *AssociationSet* elementi predstavljaju instance asocijacija. *AssociationSet* identifikuje *EntitySet*-ove koji učestvuju u asocijaciji i konceptualno je sličan vezi između tabela u bazi.

Primer *EntitySet*–a:

```
<EntitySet Name="People" EntityType="EntityFrameworkExampleModel.Person" />
```

Primer *EntityType*–a:

```
<EntityType Name="Person">
<Key>
<PropertyRef Name="BusinessEntityID" />
</Key>
<Property Name="BusinessEntityID" Type="Int32" Nullable="false" />
<Property Name="PersonType" Type="String" Nullable="false" MaxLength="50"
Unicode="true" FixedLength="false" />
<Property Name="NameStyle" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="Title" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="FirstName" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="MiddleName" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="LastName" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="Email" Type="String" MaxLength="50" Unicode="true"
FixedLength="false"/>
```

```
<Property Name="AdditionalContact" Type="String" MaxLength="50" Unicode="true"
FixedLength="false" />
<Property Name="ModifiedDate" Type="String" MaxLength="50" Unicode="true"
    FixedLength="false" />
    <Property Name="AddressID" Type="Int32" Nullable="false" /><NavigationProperty
    Name="Address"         Relationship="EntityFrameworkExampleModel.FK_PersonAddress"
    FromRole="Person" ToRole="Address" />
    <NavigationProperty Name="BusinessEntity"
    Relationship="EntityFrameworkExampleModel.FK_PersonInheritsBusinessEntity"
    FromRole="Person" ToRole="BusinessEntity" />
    <NavigationProperty Name="Employee"
    Relationship="EntityFrameworkExampleModel.FK_EmployeeInheritsPerson" FromRole="Person"
    ToRole="Employee" />
    </EntityType>
```

Primer *AssociationSet*–a:

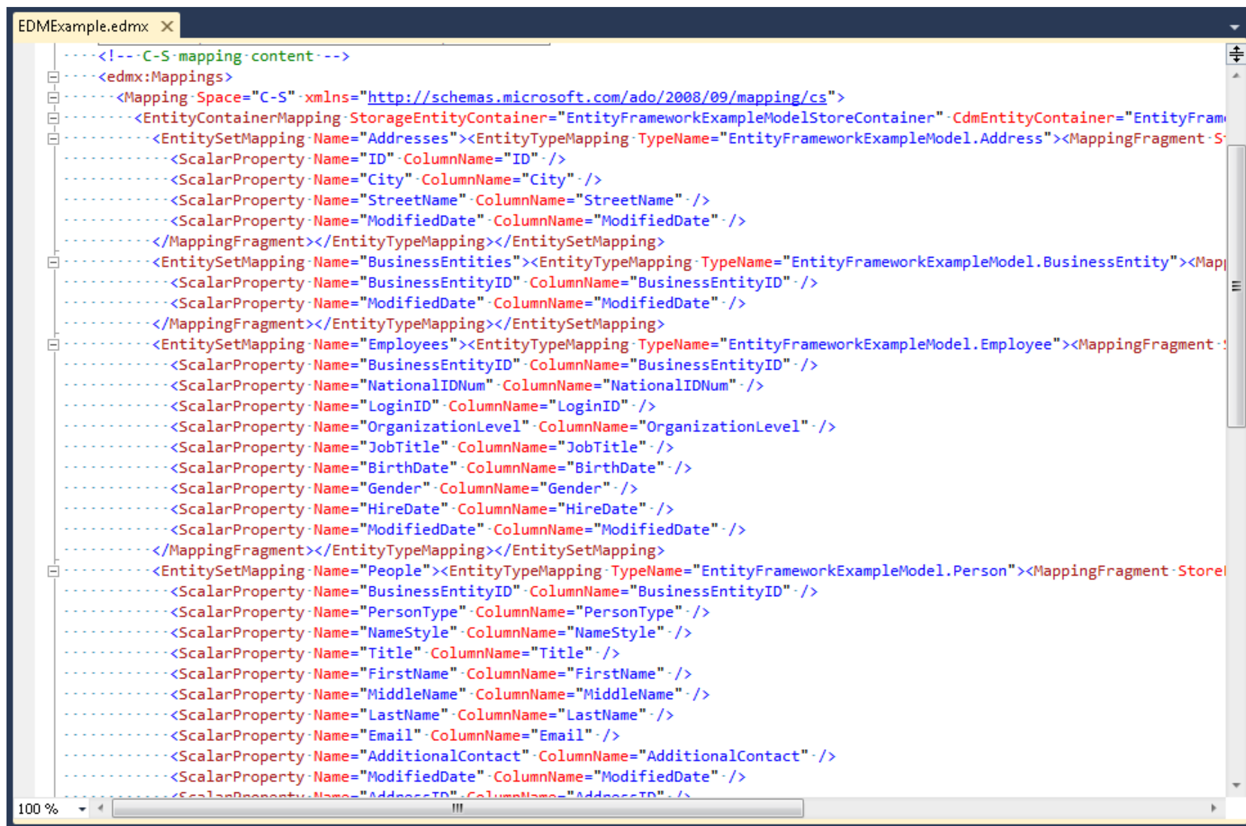
```
<AssociationSet Name="FK_EmployeeInheritsPerson"
Association="EntityFrameworkExampleModel.FK_EmployeeInheritsPerson">
<End Role="Person" EntitySet="People" />
<End Role="Employee" EntitySet="Employees" />
</AssociationSet>
```

Primer asocijacije u *CSDL*-u:

```
<Association Name="FK_EmployeeInheritsPerson">
<End Role="Person" Type="EntityFrameworkExampleModel.Person" Multiplicity="1" />
<End Role="Employee" Type="EntityFrameworkExampleModel.Employee"
Multiplicity="0..1" />
<ReferentialConstraint>
<Principal Role="Person">
<PropertyRef Name="BusinessEntityID" />
</Principal>
<Dependent Role="Employee">
<PropertyRef Name="BusinessEntityID" />
</Dependent>
</ReferentialConstraint>
</Association>
```

3.3.3. MSL

Ova datoteka se koristi za povezivanje *CSDL* tipova sa metapodacima baze podataka definisanih u *SSDL*-u.



Slika 16: *MSL*

Element <*EntityContainerMapping*> se koristi da preslika model (*CSDL*) na bazu (*SSDL*).

```
<EntityContainerMapping
StorageEntityContainer="EntityFrameworkExampleModelStoreContainer"
CdmEntityContainer="EntityFrameworkExampleEntities">
```

Atribut *StorageEntityContainer* prikazuje ime *EntiyContainer*-a u bazi dok atribut *CdmEntityContainer* prikazuje odgovarajući *EntityContainer* u modelu.

Preslikavanje *EntitySet*-a iz modela na *EntitySet* iz baze zahteva element <*EntitySetMapping*>.

```
<EntitySetMapping Name="People">
<EntityTypeMapping TypeName="EntityFrameworkExampleModel.Person">
<MappingFragment StoreEntitySet="Person">
<ScalarProperty Name="BusinessEntityID" ColumnName="BusinessEntityID" />
<ScalarProperty Name="PersonType" ColumnName="PersonType" />
<ScalarProperty Name="NameStyle" ColumnName="NameStyle" />
<ScalarProperty Name="Title" ColumnName="Title" />
<ScalarProperty Name="FirstName" ColumnName="FirstName" />
<ScalarProperty Name="MiddleName" ColumnName="MiddleName" />
<ScalarProperty Name="LastName" ColumnName="LastName" />
<ScalarProperty Name="Email" ColumnName="Email" />
<ScalarProperty Name="AdditionalContact" ColumnName="AdditionalContact" />
<ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate" />
<ScalarProperty Name="AddressID" ColumnName="AddressID" />
</MappingFragment></EntityTypeMapping></EntitySetMapping>
```

Atribut *Name* definiše ime *EntitySet*-a u modelu dok atribut *TypeName* definiše ime odgovarajućeg *EntitySet*-a u bazi. Svaki član u modelu je preslikan na bazu kroz element <*ScalarProperty*>.

3.4. Entity Client Data Provider

Komponenta *Entity Client Data Provider* se koristi za pristup konceptualnom modelu u *EntityFramework*-u. *Entity Client Data Provider* se nalazi u prostoru imena *System.Data.EntityClient* i predstavlja standardni *ADO.NET* provajder koji podržava pristup podacima opisanim u *EDM*-u.
Entity Client Data Provider za komunikaciju sa konceptualnim modelom koristi svoj jezik *Entity SQL*. *Entity SQL* je dijalekat *SQL*-a koji je nezavistan od baze podataka, koji se koristi direktno nad konceptualnim modelom i podržava karakteristike *EDM*-a.
Klasa *EntityCommand* se koristi da izvrši komandu *Entity SQL*-a nad modelom entiteta. Kada se konstruiše objekat *Sistem.Data.EntityClient.EntityCommand*, prosleđujemo mu ime uskladištene procedure ili tekst upita koji treba izvršiti. Iako postoji interakcija između *Entity Client Data Provider*-a i entiteta u *EDM*-u, on ne vraća instance tih entiteta već umesto njih vraća sve rezultate u objektu *DbDataReader*. *EntityCommand* za povezivanje na *EDM* koristi objekat *EntityConnection* koji može da prihvati parametare za povezivanje sa konceptualnim modelom ili vrednost atributa elementa *<connectionStrings>* definisanog u konfiguracionoj datoteci. Atribut *<conectionString>* sadrži listu datoteka sa metapodacima (datoteke *CSDL*, *MSL* i *SSDL*).
Primer korišćenja *Entity Client*-a:

```
var firstname = "";
var lastname = "";

using (EntityConnection conn = new EntityConnection("name =
EntityFrameworkExampleEntities"))
{
    conn.Open();
    var query = "SELECT p.FirstName, p.LastName FROM
    EntityFrameworkExampleEntities.People
    AS p WHERE p.LastName = 'King' Order by p.FirstName";

    EntityCommand cmd = conn.CreateCommand();
    cmd.CommandText = query;

    using (EntityDataReader rdr
    = cmd.ExecuteReader(System.Data.CommandBehavior.SequentialAccess))
    {
        while (rdr.Read())
        {
            firstname = rdr.GetString(0);
            lastname = rdr.GetString(1);
            listBox1.Items.Add(string.Format("{0} {1}", firstname, lastname));
        }
    }

    conn.Close();
}
```

Gde su u konfiguracionoj datoteci definisani parametri za povezivanje na bazu podataka:

```
<connectionStrings><add name="EntityFrameworkExampleEntities"
connectionString="metadata=.\naziv_csdl_datoteke|.\naziv_ssdl_datoteke|.\naziv_msl_datotek
e;provider=System.Data.SqlClient;provider connection string=&quot;Data
Source=baza_podataka;Initial Catalog=ime_tabele;Integrated Security=True&quot;"
providerName="System.Data.EntityClient" /></connectionStrings>
```

Entity Client komunicira sa *ADO.Net Data Provider*-om koji šalje ili preuzima podatke iz baze podataka.

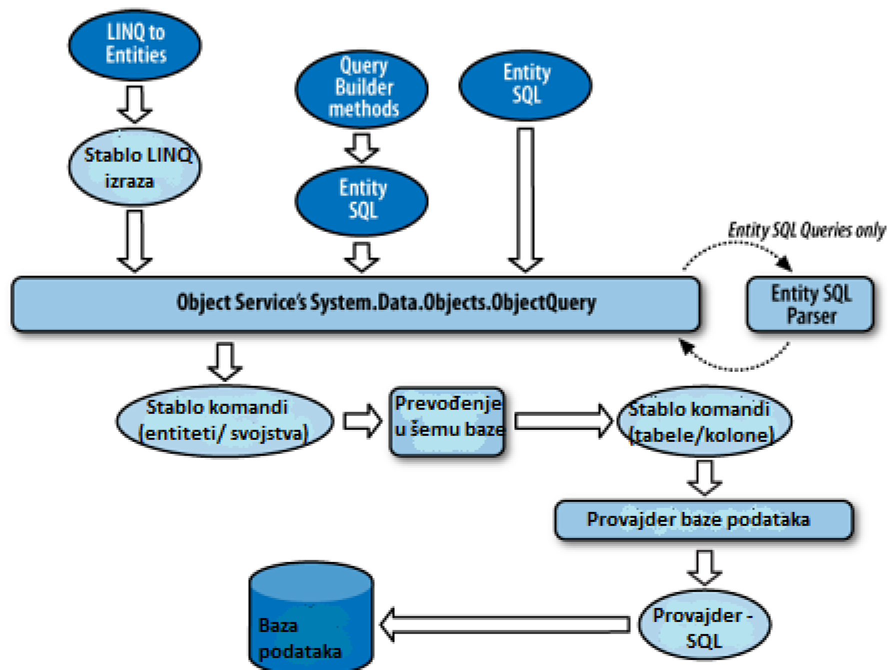
3.5. Objektni servisi

Objektni servisi *(eng. Object Services)* predstavljaju komponentu *Entity Framework*-a koja omogućava korišćenje upita za dodavanje (*eng. insert)*, menjanje (*eng. update)* i brisanje (*eng. delete*) nad podacima koji su predstavljeni kao objekti. Objektni servisi podržavaju *Language Integrated Query* (*LINQ*) i *Entity SQL* upite nad tipovima definisanim u *EDM*-u. Rezultujući podaci su u formi objekata, a promene nad objektima čuvaju se u bazi podataka. Takođe, objektni servisi obezbeđuju praćenje promena nad podacima učitanim iz baze (*eng. tracking changes*), povezivanje objekata sa kontrolama i upravljanje konkurentnošću.
Klasa *ObjectContext* predstavlja osnovnu klasu za interakciju sa podacima u formi objekata koji su primerci (instance) tipova entiteta definisanih u *EDM*-u. Objektni servisi čuvaju objekte podataka u memoriji i dozvoljavaju dodavanje, menjanje i brisanje objekata u *ObjectContext*-u. Kada se eksplicitno zahteva, objektni servisi sačuvaju te promene u bazu. U *EDM*-u, *EntityContainer* je predstavljen kao klasa nasleđena od *ObjectContext*-a. Klasa *ObjectContext* implementira interfejs *ObjectQuery<T>*, što omogućava pravljenje upita korišćenjem *Entity SQL* i *LINQ*. Instanca klase *ObjectContext* sadrži vezu ka tekućoj bazi podataka, metapodatke koji opisuju model tekuće baze podataka i objekat tipa *ObjectStateManager* koji rukovodi entitetima tokom operacija dodavanja, brisanja i izmene.
Klasa *ObjectContext* obezbeđuje:

1. Keširanje podataka
   Podaci iz baze se smeštaju u memoriju i tu se vrše operacije umetanja, izmene i brisanja. Postoje mehanizmi za praćenje izmena nad podacima. Podaci se iz memorije mogu trajno sačuvati u bazu.
2. Konkurentni pristup
   Klasa *ObjectContext* sadrži implementaciju strategija koje se koriste za upravljanje konkurentim pristupom.
3. Upravljanje vezama (konekcijama)
   Automatski obavlja otvaranje i zatvaranje veze ka bazi.

Osnovne funkcije objektnih servisa možemo podeliti na:

1. Obradu upita (*eng. Query processing*):
   Obrada upita u *Entity Framework*-u podrazumeva prevođenje *LINQ* ili *Entity SQL* upita u standardne *SQL* upite. Na slici 17 prikazani su koraci koje upit prolazi od *LINQ to Entities* forme do baze:

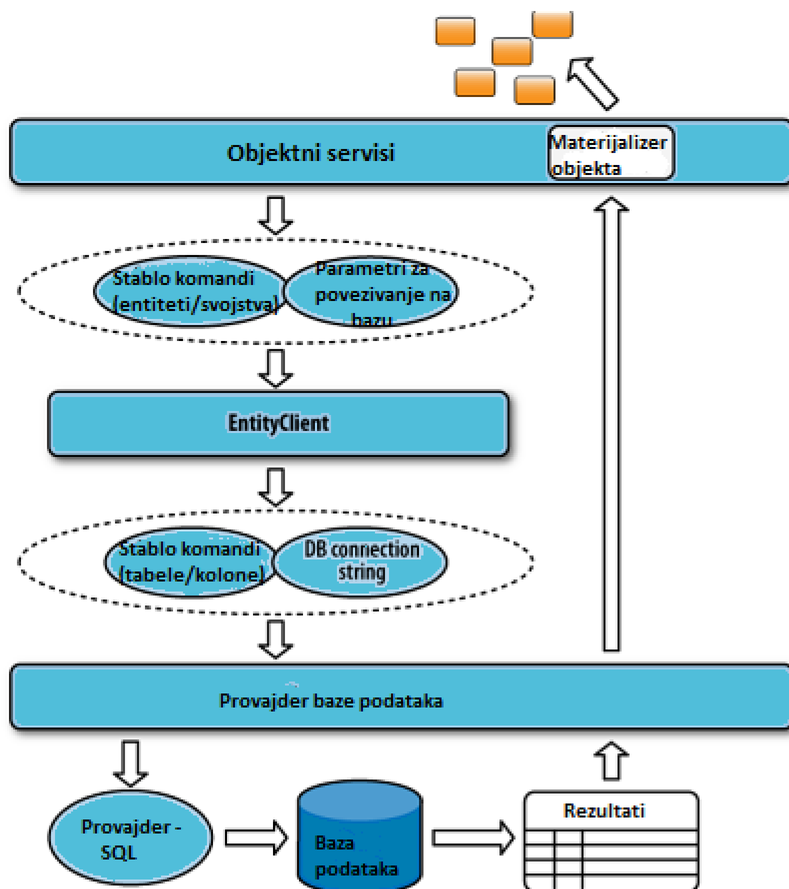


.

Slika 17: Prikaz koje korake prolazi *LINQ to Entities* upit dok ne dođe do baze podataka

LINQ prevodi *LINQ to Entities* upit u stablo *LINQ* izraza (*eng. LINQ Expression Tree*) koje je dobijeno prevođenjem *LINQ* operatora u pozive odgovarajućih metoda. Ovaj način predstavljanja *LINQ* upita poznat je kao metodna sintaksa. *LINQ Expression Tree* se dalje, pomoću objektnih servisa, raščlanjava u stablo komandi (*eng. Command Tree)* koje se sastoje od *LINQ* ili *Entity SQL* operatora ili funkcija primenjenih na entitete i svojstva. Nakon toga, korišćenjem metapodataka za opis preslikavanja između modela, vrši se prevođenje u stablo komandi nad tabelama i kolonama. Provajder prevodi upit u *SQL* koji je razumljiv bazi podataka.

2. Materijalizaciju objekata (*eng. Object materialization*).
   Materijalizacija predstavlja proces transformacije slogova baze podataka u objekte programa. Nakon sto *Entity Client* vrati rezultate iz baze podataka u *EntityDataReader*-u, objektni servisi transformisu ili materijalizuju rezultate u objekte.



Slika 18: Materijalizacija rezultata upita

3.6. LINQ to Entities

LINQ to Entities je jedna od *LINQ* implementacija koja pruža mogućnost za pravljenje upita. *LINQ* predstavlja deo *.NET framework*-a koji omogućava formiranje i izvršavanje upita nad različitim izvorima podataka (*eng. data sources*). *LINQ to Entities* podržava sve standardne *LINQ* operatore. *Entity Framework* pretvara *LINQ* upit u *SQL* upit koji je razumljiv bazi podataka.

LINQ query operacije se sastoje iz tri dela:

1. Obezbeđivanje izvora podataka
   Izvor podataka za *LINQ* je bilo koji objekat koji podržava generički interfejs *IEnumberable<T>* ili interfejs nasleđen od njega. Tipovi koji podržavaju *IEnumerable<T>* ili interfejse koji su izvedeni iz njega kao što je generički interfejs *IQueryable<T>* zovu se *queryable* tipovi. *Queryable* tipovi ne zahtevaju posebnu modifikaciju da bi služili kao izvor podataka za *LINQ*. Ako izvorni podaci nisu *queryable* tipovi, *LINQ* provajder mora da ih predstavi kao takve.

2. Pravljenje upita:
   Upit precizira koje informacije treba da se preuzmu iz izvora podataka. On takođe precizira kako informacije da se urede, grupišu i u kom obliku vrate. Upit se sastoji od tri

osnovne klauzule: *from*, *where* i *select*. Klauzula *from* navodi izvor podataka, *where* primenjuje filter, a klauzula *select* precizira tip povratnih vrednosti.

3. Izvršavanje upita.
   Stvarno izvršenje upita je odloženo dok se ne uđe u iteraciju u *foreach* petlji.

LINQ operatori se mogu koristiti nad različitim izvorima podataka. Ono sto je bitno za njih je da su podaci predstavljeni kao objekti. Operatori upita su definisani u klasi *System.Query.Sequence* kao *Extension* metode (metode koje proširuju funkcionalnost klasa) za *IEnumerable<T>* interfejs. Skup standardnih operatora koje definiše *LINQ* predstavljeni su kao *Standard Query Operator API*.

Posmatrajmo primer:

```
using (var context = new EntityFrameworkExampleEntities())
{
        var people = from p in context.People
                        where p.LastName == "Petrovic"
                        select p;

        foreach (var person in people)
        {
                Console.WriteLine(string.Format("{0} {1}", person.FirstName,
                person.LastName));
        }
}
```

Ovo je *LINQ To Entites* upit. *LINQ* upiti počinju klauzulom *from* i završavaju se klauzulom *select*. Pored ovih klauzula, koriste se jos i klauzule *join*, *group by*, *having*, *order by*,...

Prethodno naveden upit bi u *SQL*-u izgledao ovako:

```
select * from Person where LastName =‘Petrovic’
```

Klauzula *from* se prva obrađuje, dok se *select* obrađuje među poslednjim. Upotreba ostalih navedenih klauzula biće opisana kasnije.

U ovom primeru, klasa *EntityFrameworkExampleEntities* predstavlja *EntityContainer*. *EntityContainer* nasleđuje klasu *ObjectContext*. Klasa *ObjectContext* predstavlja osnovnu klasu koja omogućava uparavljanje podacima kao objektima definisanim u *EDM*-u.

Prva linija u ovom primeru:

```
using (var context = new EntityFrameworkExampleEntities())
```

obezbeđuje povezivanje na bazu podataka, kao i metapodatke koji opisuju model.

Deo koda:

```
var people = from p in context.People
                where p.LastName == "Tesla"
                select p;
```

predstavlja upit.

Izvršavanje upita i iteraciju kroz rezultate obavlja sledeći deo koda:

```
foreach (var person in people)
{
  Console.WriteLine(string.Format("{0} {1}", person.FirstName, person.LastName));
}
```

LINQ to Entities upiti mogu biti predstavljeni kroz dve različite sintakse: upitna sintaksa i metodna sintaksa. Osim sto postoji razlika u sintaksi između ova dva načina predstavljanja upita, postoji i razlika u načinu na koji *CLR*[3] obrađuje ove dve vrste upita. Kada je upit predstavljen pomoću upitne sintakse, prvo se vrši prevođenje operatora u skup poziva odgovarajućih metoda koje *CLR* razume dok kod metodne sintakse nema potrebe za prevođenjem.

3.6.1. Upitna sintaksa

Upitna sintaksa koristi deklarativni način pozivanja *LINQ* operatora, izgleda slično *SQL* jeziku (*from*, *where* i *select* naredbe imaju isto značenje u *LINQ*-u kao i u *SQL*-u) i ugrađena je u jezik *C#*. Kod upitne sintakse, upiti se pišu korišćenjem operatora i funkcija.

Primer korišćenja upitne sintakse je:

```
var people = from p in context.People
             where p.LastName == "King"
             select p;
```

U nastavku ćemo se upoznati sa operatorima koje se koriste u upitima predstavljenim korišćenjem upitnu sintaksu.

- *Select* i *SelectMany*:

  Operator *Select* vraća rezultat upita i određuje oblik i tip svakog od vraćenih elemenata.
  Primer korišćenja operatora *Select* (izdvajanje svih osoba kojima je prezime Petorvic):

```
var people = from p in context.People
             where p.LastName == "Petrovic"
             select p;
```

  Možemo odrediti da li će naš rezultat da se sastoji od kompletnih objekata, samo jednog člana ili podgrupe članova.
  Operator *SelectMany* vrši projekciju jednog na više elemenata sekvence. *SelectMany* se koristi kada postoji višestruki *from*.

- *Where*

  Where je operator restrikcije koji filtrira sekvencu u zavisnosti od predikata.
  Primer (izdvajanje svih osoba čije prezime počinje na slovo *P*):

```
var query = from p in context.People
            where p.LastName.StartsWith("P")
```

3 *CLR* (*eng. Common Language Runtime*) predstavlja izvršno okruženje koje upravlja *.NET* kodom u toku izvršavanja.

```
                    select new { p.LastName, p.FirstName, p.MiddleName, p.BusinessEntityID };
```

- *OrderBy/ThenBy*:

Operatori *Order by / Then by* vrše sortiranje sekvence na osnovu jednog ili više ključeva. *Order by* metoda određuje primarno sortiranje elemenata u rastućem redosledu. Za obrnuti redosled koristi se metoda *Order by descending*. Ostala sortiranja se određuju operatorima *Then by* i *Then by descending*.

U narednom primeru se vrši sortiranje po prezimenu a zatim po imenu:

```
var people = from p in context.People
                    where p.LastName == "Petrovic"
                            || p.FirstName == "Petar"
                    orderby p.LastName, p.FirstName
                    select new { p.FirstName, p.LastName };
```

- *Average / Max / Min / Sum / Aggregate*

Ovi operatori računaju prosečnu vrednost, maksimalnu vrednost, minimalnu vrednost, sumu i agregaciju na osnovu predikata koji vraća neku numeričku vrednost svakog elementa kolekcije.
Primer korišćenja *Average* (određivanje prosečne dužine imena svih osoba):

```
Person[] personArray = (from person in context.People
                                        Order by person.FirstName descending
                                        select person).ToArray();

double averageLength = personArray.Average(w => w.FirstName.Length);
```

- *Count*

Operator *Count* vraća broj elemenata kolekcije.
Primer:

```
var query = (from p in context.People
                    where p.LastName.StartsWith("P")
                    select new
        {
          p.LastName,
          p.FirstName,
          p.MiddleName,
          p.BusinessEntityID
         }).Count();
```

- *Take/Skip*

Operator *Take* vraća traženi broj elemenata sa početka sekvence. Iteracija kroz ulazne elemente se zaustavlja kad se vrati željeni broj elemenata ili se dostigne kraj sekvence. Operator *Skip* preskače navedeni broj elemenata sekvence, a onda vraća njen ostatak. Operatori *Take* i *Skip* su funkcionalno komplementni što znači da ako se na neku sekvencu primene. *Take(n)* i *.Skip (n)*, rezultat će biti identičan polaznoj sekvenci.
Primer korišćenja operatora *Take*:

```
var query = (from address in context.Addresses
                    from people in context.People
                    where address.ID == people.AddressID
                            && address.City == "Beograd"
                    select new
                    {
                            Name = people.FirstName,
```

```
                    City = address.City
                }).Take(3);
```

- *Join/GroupJoin*:

Operator *Join* vrši unutrašnje spajanje dve sekvence na osnovu identičnih ključeva elemenata. U terminima relacione baze podataka, operator *Join* odgovara unutrašnjem spajanju tabela (*eng. inner join*).
Primer upotrebe operatora *join*:

```
var people = from p in context.People
             join emp in context.Employees on p.BusinessEntityID equals emp.BusinessEntityID
            order by p.LastName, p.FirstName descending
            select new { p.FirstName, p.LastName, emp.HireDate };
```

GroupJoin radi slično kao operator *Join*. On proizvodi hijerarhijske rezultate. Ne postoji direktni ekvivalent ovog operatora sa tradicionalnim operatorima relacione baze podataka. Može se iskoristiti za pravljenje levog i desnog spoljašnjeg spajanja (*left/right outer join*).
Primer korišćenja operatora *GroupJoin*:

```
var query = from people in context.People
                join address in context.Addresses
                on people.AddressID     equals address.ID into orderGroup
                select new
                    {
                    PeopleID = people.BusinessEntityID,
                    OrderCount = orderGroup.Count()
                };
```

- *First*

Operator *First* vraća prvi element za koji prosleđeni predikat vraća tačnu vrednost. Ukoliko takav element ne postoji, generiše se izuzetak.

```
var query = (from p in context.People
                where p.LastName == "Petrovic"
                select p).First();
```

- *Group by*

Operator *Group by* grupiše elemente sekvence.
Primer korišćenja *Group by* operatora:

```
var people = from p in context.People
                orderby p.FirstName
                where p.LastName == "Petrovic"
            || p.LastName == "Petar"
                group p by p.LastName;
```

3.6.2. Metodna sintaksa

Metodna sintaksa koristi metode za pozivanje *LINQ*-u operatora i ugrađena je u *.NET framework* (*System.Linq*).
Iako je sintaksa drugačija, rezultat izvršavanja oba načina predstavljanja upita je jednak.
U nastavku slede primeri nekoliko upita u metodnoj sintaksi:

- *Select*:

```
var people = context.People
.Select(person =>new
{
        FirstName = person.FirstName,
        LastName = person.LastName
});
```

- *Where*:

```
var people = context.People
      .Where(person => person.FirstName == "Petar")
                  .Select(p =>new { p.FirstName, p.LastName });
```

- *Order by/ThenBy*:

```
IQueryable<Person> sortedPeople = context.People
                                .OrderBy(p => p.LastName)
                                .ThenBy(p => p.FirstName);
```

- *Count*:

```
var query = (from p in context.People
        where p.LastName.StartsWith("P")
                selectnew { p.LastName, p.FirstName, p.MiddleName,
                p.BusinessEntityID
                }).Count();
```

- *Take*:

```
IQueryable<Person> first5People = context.People.Take(5);
```

- *ToArray:*

```
Person[] personArray = (from person in context.People
                        orderby person.FirstName descending
                        select person).ToArray();
```

- *First*:

```
var query = context.People.First(people =>
        people.FirstName.StartsWith("Pet"));
```

- *GroupBy*:

```
var query = context.Addresses
                .GroupBy(address => address.City);
```

4. Hibernate

Hibernate je alat za objektno-relaciono preslikavanje kod *Java* okruženja. Predstavlja projekat otvorenog koda (*eng. open source*) koji služi za preslikavanje *Java* klasa u tabele relacione baze podataka i omogućava upite nad bazama podataka. Prva verzija *Hibernate*-a se pojavila 2001. godine. Potreba za uvođenjem *Hibernate*-a se javila jer postojeći alati nisu bili prilagodljivi složenim šemama podataka u aplikacijama. Njegovim autorom se smatra *Gevin King*.

Hibernate okvir je napisan u programskom jeziku *Java*, tako da se može izvršavati na svim operativnim sistemima na kojima se može izvršavati i *Java* (*Windows*, *Unix*, *Linux* i dr.). Podržava rad sa gotovo svim sistemima za upravljanje bazama podataka (*SUBP*) i ima ugrađene sve specifičnosti vezane za određeni sistem. Ukoliko se javi potreba za promenom *SUBP*-a, potrebno je samo promeniti odgovarajuće parametre u konfiguracionoj datoteci, obezbediti odgovarajući drajver i upotrebiti *Hibernate* za ponovno generisanje šeme baze podataka. Pri tome nije potrebno menjati aplikaciju.
Uloga *Hibernate*-a je da bude posrednik između aplikacije i baze podataka.

Slika 19: *Hibernate* kao posrednik između aplikacije i baze podataka

Preslikavanje između objekata koji čine domenski model i tabela u bazi podataka u *Hibernate*–u se može definisati u *XML* dokumentima, programski u *Java* kodu ili preko anotacija. Zahvaljujući modularnosti arhitekture *Hibernate*-a, ovaj alat se može na različite načine korisiti prilikom razvoja softvera. Osnovna svrha *Hibernate*-a kao *ORM* alata je da se koristi za objektno-relaciono preslikavanje. Međutim, *Hibernate* se može koristiti i za transakcije i keširanje entiteta i upita.

U nastavku rada biće predstavljena arhitektura okvira *Hibernate*.

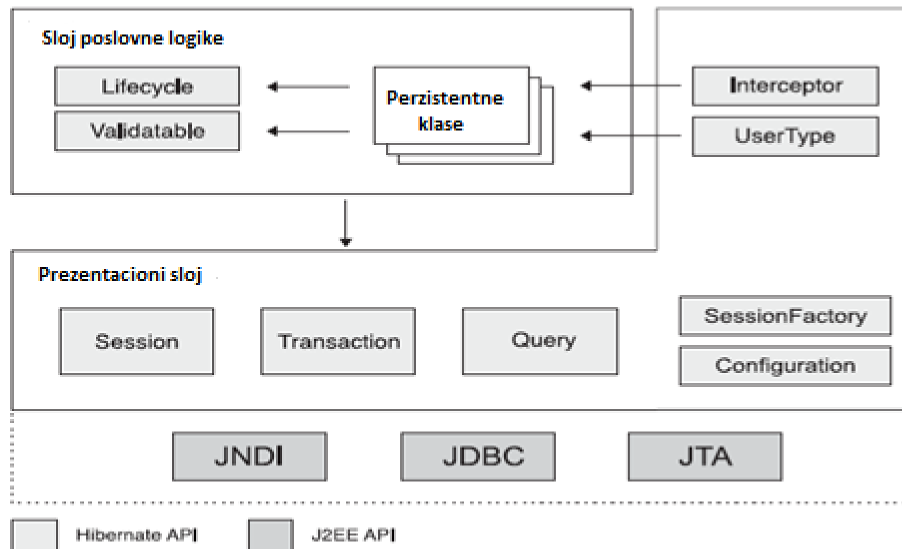
4.1. Arhitektura okvira Hibernate

Arhitektura okvira *Hibernate* se sastoji od sledećih komponenti:

1. Interfejsa: *Session*, *Transaction* i *Query*. Ove interfejse koristi aplikacija da bi izvršila osnovne *CRUD* operacije (*create*, *read*, *update*, *delete*);
2. Konfiguracione klase (*eng. Configuration)*. Ovu klasu aplikacija poziva radi konfigurisanja okvira *Hibernate*;
3. *Callback* interfejsa: *Interceptor*, *Lifecycle* i *Validatable*. Ovi interfejsi omogućavaju reakciju na određene događaje u aplikaciji.

4. Interfejsa: *UserType*, *CompositeUserType* i *IdentifierGenerator*. Ovi interfejsi omogućavaju nadogradnju (*eng. extension*) *Hibernate* funkcionalnosti za preslikavanje. Pomenuti interfejsi se implementiraju po potrebi u samoj aplikaciji.

Veza između navedenih komponenti prikazana je na slici 20:



Slika 20: Veza između komponenti *Hibernate* okvira

U daljem tekstu biće detaljnije opisane komponente okvira *Hibernate*.

4.1.1. Interfejs Session

Interfejs *Session* služi za formiranje sesije, glavnog interfejsa između *Java* aplikacije i okvira *Hibernate*. Svaka interakcija sa bazom se obavlja preko ovog interfejsa. Konkretna implementacija interfejsa *Session* se formira uz pomoć klase *SessionFactory*.
Pod kontrolom ovog interfejsa su: održavanje stanja *Java* objekata, upravljanje transakcijama i izvršavanje upita. Objekat *Session* ima svoj *API* za upravljanje perzistencijom, tako da može da čuva objekte u bazi, menja postojeće i preuzima podatke iz baze. U objektno–orijentisanoj aplikaciji trajnost (*eng. persistence*) omogućava podacima da nadžive procese koji su ih stvorili. Objekat *Session* sadrži red *SQL* naredbi koje treba da se sinhronizuju sa bazom podataka. Naredbe unutar *SQL* reda se raspoređuju tako da, bez narušavanja konzistentnosti, izvrše upit unutar baze. Podaci koji su dobavljeni preko *Session* objekta su virtualno sinhronizovani sa bazom podataka. To znači da će sve promene koje se izvrše nad podacima kojima rukuje *Session* uzrokovati dodavanje odgovarajuće naredbe u red naredbi. Ovaj postupak se zove kontrola sinhronizacije podataka (*eng. dirty checking*). Nakon sinhronizacije reda *SQL* naredbi sa bazom, sinhronizacija postaje stvarna.

Objekati *Session* u *Hibernate*-u nisu bezbedni za upotrebu prilikom paralelne obrade (*eng. thread safe*). Deljenje objekata *Session* između više programskih niti može da izazove gubitak podataka ili da dovede do međusobnog blokiranja (*eng. dead lock*) pa bi, iz tog razloga, bilo dobro da se napravi za svaku programsku nit po jedan objekat *Session*. Ukoliko objekat *Session* prijavi neki izuzetak, trebalo bi uništiti taj objekat i napraviti novi. Ovim se štite podaci u keš memoriji da ne postanu nekonzistentni u odnosu na podatke iz baze.
Objekat *Session* sadrži metode pomoću kojih se vrše osnovne operacije dohvatanja, čuvanja, izmene i brisanja podataka. To su:

1. *.load()* – Ova metoda dohvata jednu *n*-torku iz baze podataka. Metoda kao argument prima parametar objekat *class*, a kao rezultat vraća pojavljivanje date klase, čiji atributi su postavljeni odgovarajućim vrednostima, iz baze podataka. Ukoliko u bazi ne postoji odgovarajući red, metoda *load()* će prijaviti *HibernateExeption* izuzetak.

   Primer upotrebe metode *load()*:

```
public static Clan loadClana(Integer sifra)
{
        Session session = HibernateUtil.getSession();
        Transaction trans = null;
        Clan clan = null;
        try
        {
                trans = session.beginTransaction();
                clan = (Clan)session.load(Clan.class, sifra);
                trans.commit();
        }
        catch (RuntimeException e)
        {
                trans.rollback();
        }
        return clan;
}
```

2. *.save()* – zapisuje objekat u bazu podataka. Pomoću ove metode vrši se povezivanje objekta sa sesijom, odnosno prevođenje objekta u perzistentno stanje.

   Primer upotrebe metode *save():*

```
public static void  snimiClana(Clan clan)
{
        Session session = HibernateUtil.getSession();
        Transaction trans = null;
        try
        {
                if(!getListaClanova().contains(clan))
                {
                        trans = session.beginTransaction();
                        session.save(clan);
                        trans.commit();
                }
        catch (RuntimeException e)
        {
                trans.rollback();
        }}
```

3. *update()* – menja objekat u bazi podataka. Objekat prepoznaje na osnovu identifikatora objekta. Izmene se vrše u samom objektu i zato nisu potrebni nikakvi drugi argumenti. Prilikom korišćenja *update()* metode neophodno je voditi računa da u tekućoj sesiji ne postoji objekat sa istim identifikatorom kao i objekat koji se prosleđuje *update()* metodi.

   Primer upotrebe metode *update():*

```
public static void urediClana(Clan clan, Clan noviClan)
{
        Session session = HibernateUtil.getSession();
        Transaction trans = null;
        try
        {
                trans = session.beginTransaction();
                clan.setIme(noviClan.getIme());
                clan.setPrezime(noviClan.getPrezime());
                session.update(clan);
                trans.commit();
        }
        catch (RuntimeException e)
        {
                trans.rollback();
    }

        noviClan = clan;
}
```

4. *.delete()* – briše objekat iz baze podataka na osnovu identifikatora objekta. Pozivom *delete* metode brišu se podaci iz baze podataka, međutim, aplikacija i dalje može da zadrži referencu na objekat čiji su podaci obrisani.

   Primer upotrebe metode *delete()*:

```
public static void obrisiClana(Clan clan)
{
        Session session = HibernateUtil.getSession();
        Transaction trans = null;
        try
        {
                trans = session.beginTransaction();
                session.delete(clan);
                trans.commit();
        }
        catch (RuntimeException e)
        {
                trans.rollback();
        }
}
```

4.1.2. Interfejs Transaction

Interfejs *Transaction* je opcioni *API*. U *Hibernate* aplikacijama je moguće koristiti ovaj interfejs ili upravljanje transakcijama realizovati programski (realizovati ga u samom kodu). Ukoliko se on ne koristi, trebalo bi koristiti *flush()* metodu nad *Session* objektom i to svaki put kada su izvršene promene u bazi. Objekat *Transaction* vrši apstrakciju i skrivanje implementacije mehanizma izvršavanja transakcija od aplikacije i omogućava portabilnost *Hibernate* aplikacija između različitih izvršnih okruženja.
Ovaj objekat se bavi životnim vekom transakcije. Transakcija predstavlja logičku jednicu posla pri radu sa podacima. Nju čini niz radnji koje ne narušavaju uslove integriteta. Nakon izvršenja transakcije, stanje baze treba da bude konzistentno. Potvrda transakcije (*eng. commit*) označava uspešan završetak transakcije i ponovno uspostavljanje konzistentnog stanja u bazi. Poništavanje transakcije (*eng. rollback*) označava neuspešan završetak transakcije i poništenje svih efekata koje su proizvele radnje te transakcije. *Hibernate API* može postaviti ograničenja prenosa podataka tako da se razmena podataka sa bazom može dogoditi jedino unutar trajanja transakcije, koja započinje pozivom metode *.transactionBegin()*, a završava se pozivom metode *.commit()*. Sva komunikacija sa bazom podataka je virtualna i ostaje takva sve do poziva metode *.commit()*, kada se izvršavaju naredbe koje su sačuvane unutar objekta *Session*.

Primer korišćenja navednih metoda:

```
Session session = factory.openSession();
try
{
        session.beginTransaction();
        session.getTransaction().commit();
}
catch (HibernateException e)
{
        Transaction tx = session.getTransaction();
        if (tx.isActive())
        tx.rollback();
}
finally
{
        session.close();
}
```

Transakcija se karakteriše sledećim svojstvima (poznatim kao *ACID* svojstva):

1. atomičnost (*eng. atomicity*) - podrazumeva da se transakcija izvrši u potpunosti ili se uopšte ne izvrši;
2. konzistentnost (*eng. consistency*) – transakcija prevodi jedno konzistentno stanje baze u drugo konzistentno stanje baze što znači da transakcija ne sme da poništi pravila koja su definisana u bazi podataka. Ako se iz nekog razloga na kraju transakcije naruši konzistentnost baze podataka, ona se poništava. Za vreme izvršavanja transakcije dozvoljena je privremena nekonzistentnost baze. Transakcija se ponaša kao jedinstvena radnja sa tačke gledišta uslova integriteta, što ne važi za njene pojedinačne radnje;
3. izolacija (*eng. isolation*) – efekti izvršenja transakcije nisu vidljivi drugim transakcijama dok se ona ne izvrši uspešno;
4. trajnost (*eng. durability*) - efekti uspešno kompletirane transakcije su trajni, tj. mogu se poništiti samo drugom transakcijom.

Da bi se sprečili istovremeni pristupi podacima, transakcija zaključava podatke. Zaključavanje objekta predstavlja postupak koji transakciji obezbeđuje pristup objektu i sprečava druge transakcije da pristupe zaključanom objektu. Zaključavanje može trajati dok se naredba izvršava nad podacima ili do završetka transakcije. Na kraju izvršavanja, transakcija otključava objekte koje je zaključala.

4.1.3. Interfejs Query

Ovaj objekat služi za direktno postavljanje upita bazi podataka. Upiti se mogu pisati u objektno-orijentisanom *HQL*-u (*Hibernate Query Language*) ili u običnom *SQL*-u. Rezultati izvršavanja upita su objekti, koji se mogu potom dohvatiti iz objekta *Query* pojedinačno ili u obliku kolekcije (*eng. List*). Ovaj interfejs omogućava iterativnu obradu vraćenih rezultata upita (*eng*. *result set*). Pravljenje upita vrši se korišćenjem *createQuery()* metode tekuće sesije.
Primer korišćenja *Query* interfejsa:

```
public List ListCustomer
{
   Session sesion = HibernateUtil.getSessionFactory(). getCurrentSession();
sesion.beginTransaction();
 List listCustomer= sesion.createQuery("from Customer").list();
     sesija.getTransaction().commit();
     return listCustomer;
}
```

4.1.4. Konfiguraciona klasa

Konfiguraciona klasa se koristi za konfigurisanje i inicijalizaciju okvira *Hiberante*. Ovoj klasi je potrebno obezbediti konfiguracione podatke na osnovu kojih objekat *Configuration* formira instance klase *SessionFactory*.

4.1.5. Interfejsi Callback, UserType i CompositeUserType

Interfejsi *Callback* omogućavaju aplikaciji da primi informaciju kada se nad objektom izvrši neka operacija (učitavanje, čuvanje ili brisanje). *Hibernate* aplikacije ne moraju da implementiraju ove interfejse ali mogu da ih koriste za implementaciju nekih generičkih funkcionalnosti. Interfejsi *Lifecycle* i *Validatable* omogućavaju perzistentnim objektima da reaguju na događaje u skladu sa svojim perzistentnim životnim ciklusom. Perzistentni životni ciklus je određen *CRUD* operacijama objekta. Interfejs *Interceptor* omogućava aplikaciji da obrađuje pozive bez zahteva da perzistentne klase implementiraju *HibernateAPI*. Implementacije *Interceptor* interfejsa se prosleđuju perzistentnim instancama kao parametri.
Interfejsi *UserType* i *CompositeUserType* omogućavaju korisnički definisane (*eng. custom*) tipove.

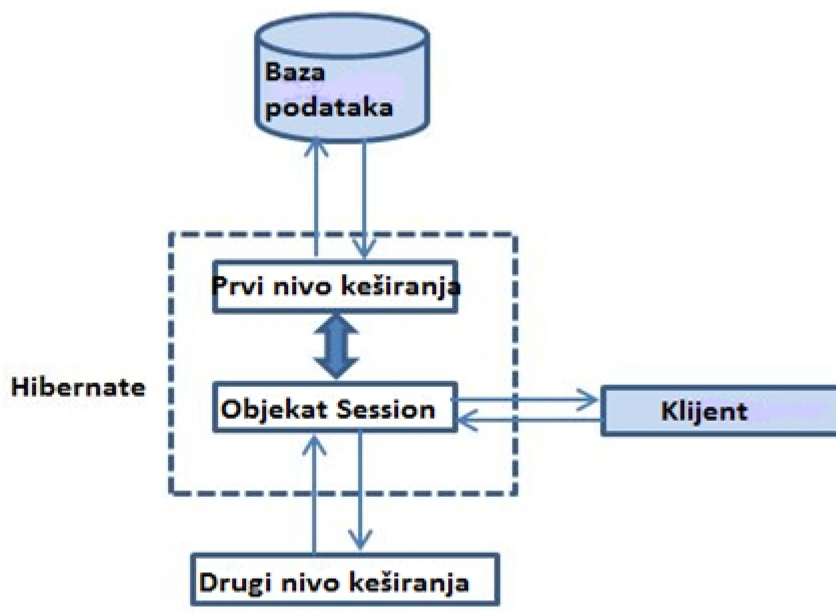
4.2. Funkcije Hibernate-a

Hibernate podržava: keširanje (*eng. caching*), kaskadno povezivanje (*eng. cascading*) i lenjo učitavanje (*eng. lazy loading*).

4.2.1. Keširanje

Keširanje predstavlja proces privremenog skladištenja kopija podataka na mestu gde im se može brže pristupiti. U *Hibernate*–u se keširanjem objekata postiže poboljšanje performansi. To omogućava da se, pri svakom sledećem zahtevu za istim objektima, umesto operacija čitanja podataka iz baze, ovi objekti čitaju iz keš memorije.

U *Hibernete*-u postoje dva nivoa keširanja.



Slika 21: Nivoi keširanja u *Hibernate*-u

Razlikujemo:

1. Prvi nivo keširanja u *Hibernate*–u je podrazumevani (*eng. default*). Prvi nivo keširanja je nivo koji svi zahtevi moraju da prođu. On za isti zahtev vraća objekte koji su u kešu, a odnose se na taj zahtev, čime se smanjuje broj pristupa bazi. Ukoliko se vrši više izmena nad objektom, sve te izmene se čuvaju u keširanoj kopiji objekta da bi se umesto pojedinačnog ažuriranja stanja u bazi podataka nakon svake promene stanja objekta, ažuriranje izvršilo na kraju izvršenja transakcije. Prvi nivo kešira samo one podatke kojima se pristupa u toku jedne sesije rada za bazom podataka.

2. Drugi nivo keširanja je opcioni keš. Pre nego sto se kontaktira drugi nivo keširanja, mora se proći kroz prvi nivo keširanja. Drugi nivo keširanja može koristiti više aplikacija ili

više distribuiranih aplikacija jer se nalazi eksterno, van okvira *Hibernate* i sve sesije dele isti drugi nivo keširanja. Na drugom nivou keširanja postoje dva načina keširanja: u okviru procesa (*eng. process scope*) i u okviru klastera (*eng. cluster scope*).

Na drugom nivou keširanja, *Hibernate* podržava i mehanizam keširanja rezultata upita (*eng. ResultSet*). Keširanje rezultata upita razlikuje se od keširanja objekata, jer se tom prilikom ne keširaju objekti koji se dobijaju iz upita, već samo identifikatori tih objekata.

Konfigurisanje keširanja podrazumeva određivanje pravila keširanja. Pod pravilima keširanja podrazumeva se da li je keširanje uključeno, koja strategija smeštanja keširanih objekata će se koristiti, način na koji će se keširati podaci i mesto gde će se keširani podaci smestiti.
Postoje četiri načina keširanja objekata:

1. *read only* - Ovo predstavlja najjednostavniji način keširanja gde se zna da se perzistentna klasa neće menjati;
2. *read/write* – Način keširanja koji se koristi za keširanje perzistentnih klasa koje će se tokom rada menjati;
3. *nonstrict read/write* – Ovaj način se koristi za keširanje perzistentnih klasa koje će se relativno retko menjati;
4. *transactional* – Ovaj način se koristi kod distribuiranih sistema.

4.2.2. Kaskadno povezivanje

Kaskadno povezivanje predstavlja pojavu da se promene u jednoj tabeli baze podataka odražavaju na promene u drugoj tabeli ili promene na jednom objektu u aplikaciji na druge objekte. Kaskadno povezivanje ima važnu ulogu u očuvanju integriteta podataka. Postoje tri vrste kaskadnog povezivanja. To su:

1. kaskadno brisanje (*eng. delete cascade*) - svi slogovi sa vrednošću stranog ključa jednakom primarnom ključu brišu se zajedno sa slogovima primarnog ključa;
2. kaskadno dodavanje (*eng. cascading inserts*) – prilikom dodvanja sloga u određenu tabelu, potrebno je dodati slog i u sve tabele koje imaju strani ključ na neku kolonu tabele u koju se dodaje slog;
3. kaskadne izmene (*eng. cascading updates*) - izmene podataka u jednoj tabeli, dovode do promena u tabelama koje imaju strani ključ na neku kolonu tabele u kojoj se vrši izmena sloga.

Pomenuta kaskadna povezivanja operacija u potpunosti su implementirana na strani *DBMS*-a, ali potrebno je realizovati njihovu implementaciju na samim objektima, kako ne bi došlo do konflikta sa ograničenjima integriteta na strani baze podataka. Automatizovani *ORM* sistemi prate promene na objektima i povezuju operacije na njima ako su povezani sa drugim objektima, zahtevajući izražavanje tih veza i na strani aplikacije. U nekim sistemima to se ostvaruje

asocijacijama i validacijama koje se na njima mogu postaviti, a koje pokreću povezane operacije na nivou objekata, pre nego što se sačuvaju u bazu podataka.

4.2.3. Lenjo učitavanje

Materijalizacija predstavlja proces transformacije slogova baze podataka u objekte programa. Kasna materijalizacija objekata omogućava da se materijalizacija objekta ili kolekcije objekata vrši u trenutku kada su oni potrebni (kada im se pristupi u kodu).

Hibernate koristi proksi obrazac za podršku lenjom učitavanju. Proksi obrazac je strukturni objekat koji se koristi kao posrednik za pristupanje drugom objektu radi ostvarivanja kontrole pristupa. On realizuje zamenu (*eng. surogat*) objekta koji kontroliše pristup originalnom objektu, odlaže formiranje i inicijalizaciju originala do trenutka kada je on potreban. Metode proksi objekta sadrže dodatnu logiku koja omogućava da se u vreme poziva metode pristupi bazi podataka i izvrši materijalizacija zahtevanih podataka. Prema primeni postoji više vrsta proksija:

a) Ukoliko je objekat skup za formiranje sa stanovišta procesorskih/vremenskih zahteva, možemo da odložimo njegovo formiranje za kasnije. Ova forma proksija poznata je pod nazivom virtualni proksi (*eng. virtual proxy*);
b) Ukoliko se objekat nalazi na udaljenoj (*eng. remote*) mašini, udaljeni proksi (*eng. remote proxy*) obezbeđuje lokalnog predstavnika objekta koji se nalazi u drugom adresnom prostoru;
c) Ukoliko je potrebno obezbediti bezbednosnu kontrolu pristupa do objekta, koristi se zaštitni proksi (*eng. protection proxy*). On treba da obezbedi različita prava pristupa;
d) Ukoliko je prilikom pristupa objektu potrebno izvršiti dodatne operacije, koristi se forma proksija poznata pod nazivom pametna referenca (*eng. smart reference*);

4.3. Modularnost Hibernate-a

Jedna od značajnijih karakteristika *Hibernete*-a je modularnost. *Hibernate* se sastoji od sledećih modula:

1. *Hibernate* jezgro (*eng. Hibernate Core*)
2. *Hibernate* anotacije (*eng. Hiberntate Annotations*)
3. *Hibernate* upravljač entiteta (*eng. Hibernate Entity Manager*)
4. *Hibernate* ljuštura (*eng. Hibernate Shards*)
5. *Hibernate* validator (*eng. Hibernate Validator*)

6. *Hibernate* pretraživač (*eng. Hibernate Search*)
7. *Hibernate* alati (*eng. Hibernate Tools*)

Ovi moduli se mogu koristiti pojedinačno ili kombinovano.
Hibernate jezgro predstavlja osnovni servis za perzistenciju podataka i uglavnom je jedini potreban modul za aplikaciju. Za preslikavanje koristi *XML* datoteke koje sadrže metapodatke za preslikavanje. On sadrži klase za preslikavanje između relacionog i objektnog modela, objektno-orijentisani jezik (*HQL*), itd. *Hibernate* jezgro je modul nezavistan od izvršnog okruženja na kom se koristi.

Hibernate anotacije obezbeđuju način za označavanje metapodataka koji su potrebni *Hibernate*-u radi preslikavanja (anotacije). Anotacije su ugrađene u *Java* kod i time omogućavaju nadogradnju ili eliminisanje dodatnih *XML* datoteka. *Hibernate* anotacije se sastoje od skupa osnovnih anotacija koje implementiraju *JPA*[4] specifikaciju i skupa novih, dopunskih anotacija koje se koriste za naprednija preslikavanja. Korišćenje anotacija ima prednost u odnosu na korišćenje *XML* datoteka jer je potrebno napisati manje koda za preslikavanje metapodataka.

Hibernate upravljač entiteta je opcioni modul koji implementira deo *JPA* specifikacije i obuhvata razne programske interfejse, upravljanje životnim ciklusom perzistentnih objekata i rad sa upitima. Najčešće se koristi u kombinaciji sa modulom *Hibernate* anotacije. *Hibernate* upravljač entiteta se može tretirati i kao omotač modula *Hibernate* jezgro.

Hibernate ljuštura je dizajnirana da obuhvati (*eng. encapsulate*) podatke i smanji složenost koja može da nastane ukoliko postoji prevelika količina podataka, koja zahteva distribuiranu arhitekturu sa višestrukim relacionim bazama podataka.

Hibernate validator omogućava da se uz pomoć anotacija postave domenska ograničenja koja će se provlačiti kroz sve nivoe sistema.

Hibernate pretraživač omogućava pretraživanje teksta u okviru perzistentnog modela pomoću anotacija i uobičajnog *API*-ja.

Hibernate alati se sastoje od sledećih komponenti:

a) Editor preslikavanja (*eng. Mapping Editor*) – Omogućava izmenu *Hibernate XML* datoteke za preslikavanje, podržava auto-kompletiranje (*eng. autocompletion*) imena klasa, tabela, kolona, i obeležavanje (*eng. highlighting*) sintakse;
b) Konzola (*eng. Console*) – *Hibernate* konzola omogućava konfiguraciju povezivanja na bazu podataka, prikaz klasa i njihovih međusobnih odnosa. Podržava izvršenje *HQL* upita nad bazom kao i iščitavanja rezultata upita;
c) Reverzno inženjerstvo (*eng. Reverse Engineering*) – Predstavlja jedno od značajnijih sposobnosti alata *Hibernate* koje na osnovu baze podataka generiše domenske klase i *Hibenate* datoteke preslikavanja, anotirane entitete i *HTML* dokumentaciju;

---

4 *Java Persistence API* (*JPA*) je deo *Java EE* platforme koji preslikava *Java* objekate u podatke relacione baze podataka i omogućava pristup bazi kroz *Java* objekte.

d) Čarobnjaci (*eng. Wizards*) – Obezbeđeno je nekoliko čarobnjaka, uključujući čarobnjaka za brzo generisanje *Hibernate* konfiguracione datoteke (*cfg.xml*) i *Hibernate* konfiguracione konzole;
e) Radni zadatak (*eng. Ant task*) – *Ant* zadaci omogućavaju generisanje šeme, generisanje preslikavanja ili generisanje *Java* koda.

4.4. Konfiguracija okvira Hibernate

Konfiguracija okvira *Hibernate* logički se deli u dve celine bez obzira na okruženje i bazu podataka sa kojom komunicira aplikacija. Prva celina predstavlja obezbeđivanje konfiguracionih podataka koji su neophodni okviru *Hibernate* da bi pristupio bazi podataka, a drugu celinu čine konfiguracioni podaci kojima se obezbeđuje preslikavanje između perzistentnih klasa aplikacije i odgovarajućih tabela u relacionoj bazi podataka.
Korišćenje okvira *Hibernate* započinje instanciranjem objekta *SessionFactory*, koji nastaje od objekta *Configuration*.

4.4.1. SessionFactory

Klasa *SessionFactory* je zadužena za pravljenje objekata klase *Session* prilikom svake interakcije sa bazom podataka. *SessionFactory* predstavlja veoma skup objekat jer njegovo višestruko instanciranje može da stvori probleme, a formiranje ovog objekta oduzima dosta vremena u toku izvršavanja (*eng. runtime*). *SessionFactory* je bezbedan za upotrebu prilikom paralelne obrade (*eng. thread safe*), što je dodatni razlog da se objekat instancira samo jednom i da sve niti u aplikaciji koriste taj objekat. Objekat *SessionFactory* se formira korišćenjem metode *.buildSessionFactory()*. Svaki objekat *SessionFactory* je konfigurisan za rad sa specifičnom vrstom baze podataka korišćenjem određenog *Hibernate* dijalekta. *SessionFactory* je objekat koji služi za uspostavljanje komunikacije sa bazom podataka. *SessionFactory* dobija potrebne podatke od objekta *Configuration* koji ga je instancirao.

Objekat *Configuration* se koristi pre svega za stvaranje objekta *SessionFactory*. Formiranje objekta *Configuration* se vrši tako sto se automatski otkriva da li postoji konfiguraciona datoteka:

```
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
```

Program na osnovu ove linije koda zna gde se nalazi konfiguraciona datoteka. Kada se pozove konstruktor *new Configuraion()*, *Hibernate* traži datoteku *Hibernate.properties* u glavnom direktorijumu projekta. Ukoliko pronađe datoteku *Hibernate.properties*, svojstva za *Hibernate* se učitavaju i dodaju objektu *Configuration*. Kada se pozove metoda *configure()*, *Hibernate* traži datoteku *Hibernate.cfg.xml* u glavnom direktorijumu projekta. Ukoliko ne nađe datoteku, prijaviće izuzetak. Datoteka *Hibernate.cfg.xml* se može postaviti i u neki drugi direktorijum, ali je onda obavezno u metodi *configure* navesti putanju do njegove datoteke. Pozivom metode *buildSessionFactory* formira se objekat *SessionFactory* klase. Da bi *SessionFactory* uspostavio komunikaciju sa *SUBP*–om, koristi se metoda *.openSession()*. Nakon toga kontrolu nad vezom predaje primerku objekta *Session* kog instancira.

Posmatrajmo primer formiranja konfiguracione datoteke za *Hibernate* aplikaciju. Napravićemo konfiguracionu datoteku *Hibernate.cfg.xml* i postavićemo je u izvorni (*eng. source*) direktorijum. Tako će ova datoteka ostati u glavnom direktorijumu projekta nakon kompajliranja i okvir *Hibernate* će je automatski naći. Konfiguracioni podaci potrebni za povezivanje *Hibernate*-a sa bazom podataka se mogu obezbediti pomoću *XML* datoteke (*Hibernate.cfg.xml*):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN""http://Hibernate.sourceforge.net/Hibernate-configuration-3.0.dtd"><Hibernate-
configuration>
<session-factory>
<property
name="Hibernate.dialect">org.Hibernate.dialect.MySQLDialect</property><property
name="Hibernate.connection.driver_class">com.mysql.jdbc.Driver</property><propertyname
="Hibernate.connection.url">jdbc:mysql://localhost:3306/HibernateTest</property><property
name="Hibernate.connection.username">root</property><property
name="Hibernate.connection.password">klapac</property>-
<mapping resource="domen/Programer.hbm.xml"/>
</session-factory>
</Hibernate-configuration>
```

Obavezni podaci koje ova datoteka mora da sadrži su:

1. *connection.driver_class*: program za izabrani *SUBP*;
2. *connection.url*: *JDBC URL* do baze podataka. *JDBC URL* se sastoji iz simboličkog imena programa (*jdbc:mysql*), *IP* adrese mašine na kojoj se nalazi *SUBP* (*localhost*), porta na kome je podignut *SUBP* i imena baze podataka sa kojom program treba da uspostavi konekciju;
3. *connection.username* : Korisničko ime za pristup bazi podataka;
4. *connection.password* : Lozinka za pristup bazi podataka;
5. *dialect* : Naziv *SQL* dijalekta za specifičnu bazu podataka.

Podaci kojima se opisuje preslikavanje između perzistentnih objekata i relacija u bazi podataka se mogu obezbediti ili *XML* datotekama za preslikavanje (*hbm.xml*) ili pomoću anotacija koje koriste modul *Hibernate* anotacije. Anotacije predstavljaju specijalan tip sintakse i omogućavaju uključivanje metapodataka direktno u izvorni kod aplikacije. Sledeći kod prikazuje deo *Entity* klase sa odgovarajućim anotacijama:

```
@Entity
@Table(name = "match", catalog = "world_cup")     // Preslikava naziv Entity klase u naziv
tabele baze podataka
public class Match implements java.io.Serializable { private Long id; private Team teamGuest;
@Id
@GeneratedValue(strategy = IDENTITY)
@Column(name = "id", unique = true, nullable = false)  // Preslikava naziv atributa klase u
naziv kolone u tabeli baze podataka
public Long getId() { return this.id; } public void setId(Long id) {
this.id = id;
}
@ManyToOne(fetch = FetchType.EAGER) // preslikava vezu između objekata u vezu između
tabela baze podataka
```

```
@JoinColumn(name = "team_guest_id",nullable = false) public Team getTeamGuest() { return
this.teamGuest; } public void setTeamGuest(Team teamGuest) { this.teamGuest = teamGuest;
}
```

Anotacija *@Entity* označava klasu kao entitet. Anotacija *@Id* služi za predstavljanje primarnog ključa. Anotacija *@Table* omogućava predstavljanje detalja tabele koji će biti korišćeni za čuvanje entiteta u bazi. Korišćenjem anotacije *@Column* omogućava se definisanje detalja kolone na koju će polje biti preslikano. Anotacije koje se koriste za predstavljanje veza među entitetima su: *@OneToMany* i *@MenyToOne*.

4.5. Primer objektno-relacionog preslikavanja u Hibernate-u

U sledećem primeru razmotrićemo preslikavanje klase *Person* u tabelu *PERSON* baze podataka. Da bi se realizovalo objektno–relaciono preslikavanje, *Hibernate* zahteva informacije o tome kako klasa *Person* treba da se perzistira u bazi (kako instance klase treba da se čuvaju u bazi podataka i učitavaju iz nje). Te informacije, koje predstavljaju metapodatke za *Hibernate*, upisuju se u *XML* dokument za preslikavanje. U tom dokumentu potrebno je definisati kako se članovi klase *Person* preslikavaju u kolone tabele *PERSON*.
Neka klasa *Person* ima sledeću definiciju:

```
public class Person
{
        private int BusinessEntityId;
        private String FirstName;
        private Employee Employee;

        Person()
        {
        }

        public Person(String fistName)
        {
                this.FistName = firstName;
        }

        public int getId()
        {
                return BusinessEntityId;
        }

        private void setId(int id)
        {
                this.BusinessEntityId = id;
        }

        public String getFistName()
        {
                return FistName;
        }
```

```
		public void setFistName(String name)
		{
			this.FistName = name;
		}

		public Employee getEmployee()
		{
			return nextPerson;
		}

		public void setEmployee(Employee employee)
		{
			this.Employee = employee;
		}
	}
```

Pre nego što definišemo *XML* datoteku, upoznaćemo se sa elementima svake datoteke za preslikavanje.

Osnovni element svake datoteke za preslikavanje je element <*Hibernate-mapping*>. Atributi ovog elementa služe za definisanje podešavanja koja se primenjuju za sve podelemente. Korišćenjem elementa <*cla*ss> vrši se deklaracija perzistentnih klasa. Unutar tog elementa definišu se sva preslikavanja između perzistentne klase i odgovarajuće tabele u bazi podataka. Svaka klasa koja se direktno definiše korišćenjem elementa <*class*> mora da sadrži element <*id*>. Ovaj element služi da se za svaku klasu definiše koja kolona u odgovarajućoj tabeli baze podataka predstavlja primarni ključ. Perzistentne *Java* klase poseduju atribut koji sadrži identifikatora svakog objekta. Elementom <*id*> se definiše preslikavanje između tog atributa i kolone primarnog kluča u tabeli. U okviru tela elementa <*id*> definiše se element <*generator*>. Ovim elementom se određuje na koji način se generiše primarni ključ svakog novog pojavljivanja klase tako sto se elementom <*generator*> definiše *Java* klasa koja vrši generisanje primarnih ključeva. Za svaki atribut klase koji je potrebno sačuvati u bazu podataka neophodno je definisati element <*property*> u datoteci za preslikavanje. Perzistentni atributi klase moraju da imaju odgovarajuće *set()* i *get()* metode.

Da bi se definisala *XML* datoteka, potrebno je uraditi analizu klase *Person*. Klasa *Person* ima sledeće članove: identifikatora (*BusinessEntityId*), ime i prezime kao i referencu na drugi objekat tipa *Employee*. Identifikator omogućava aplikaciji da pristupi primarnom ključu perzistentnog objekta u bazi podataka. Ako dve instance klase *Person* imaju istu vrednost identifikatora, to znači da predstavljaju isti zapis u bazi podataka. U klasi su definisane *get* i *set* metode kao i konstruktor bez argumenata. Konstruktor bez argumenata je neophodan jer *Hibernate* koristi refleksiju sa ovim tipom konstruktora da bi instancirao objekte.

Hibernate mora da ima podatke o tome kako instance klase treba da se čuvaju u bazi podataka i učitavaju iz nje. Ti metapodaci mogu da se upišu u *XML* dokument za preslikavanje u kome se, između ostalog, definiše kako se podaci članovi klase *Person* preslikavaju u kolone tabele *PERSON*.

```
<?xml version="1.0"?>
<!DOCTYPE Hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://Hibernate.sourceforge.net/Hibernate-mapping-3.0.dtd">
<Hibernate-mapping>
```

```
<class name="person.Person" table="PERSON">
<id name="BusinessEntityId" column="PERSON_ID">
<generator class="increment"/>
</id>
<property name="FirstName" column="PERSON_FIRSTNAME"/>
<many-to-one name="Employee" cascade="all"
column="Employee_ID" foreign-key="FK_Employee"/>
</class>
</Hibernate-mapping>
```

Na osnovu ove *XML* datoteke, *Hibernate* ima informacije da klasa *Person* treba da se perzistira u tabelu *PERSON*, da se podatak *BusinessEntiyId* preslikava u kolonu *PERSON_ID*, podatak *FirstName* preslikava u kolonu *PERSON_FIRSTNAME*, a da je podatak *Employee* sa njim u vezi tipa *many–to-one*. Kolona *Employee_ID* predstavlja strani ključ (*eng. foreign key*). *Hibernate* generiše šemu baze podataka i dodaje spoljni ključ nazvan *FK_Employee* u bazu podataka. *Hibernate* sada ima dovoljno informacija za generisanje svih *SQL* naredbi za umetanje, izmenu, brisanje i pronalaženje instanci klase *Person*.
Primer čuvanja novog objekat klase *Person* u bazu podataka i učitavanje svih objekata:

```
package person;
import java.util.*;
import org.Hibernate.*;
import persistence.*;

public class Person
{
public static void main(String[] args)
{
        Session sesionSave = HibernateUtil.getSessionFactory().openSession();
        Transaction transactionSave = sesionSave.beginTransaction();
        Person person = new Person("FirstName");
        int  personId = (int) sesionSave.save(person);
        transactionSave.commit();
        sesionSave.close();

        Session sesionList = HibernateUtil.getSessionFactory().openSession();
        Transaction transactionList = newSession.beginTransaction();
        List firstname = sesionList.createQuery("from Person p order by p.FirstName
        asc").list();
        transactionList.commit();
        sesionList.close();
        HibernateUtil.shutdown();
  }
}
```

U slučaju postojanja hijerarhijskog uređenja klasa, potrebno je izabrati strategiju preslikavanja hijerarhije klasa u tabele baze podataka. *Hibernate* podržava tri pristupa preslikavanja hijerarhije klasa. To su: preslikavanje cele hijerarhije klasa u jednu tabelu, preslikavanje svake konkretne neapstraktne klase u određenu tabelu i preslikavanje svake klase/potklase (uključujući i apstraktne klase) u određenu tabelu. U nastavku slede primeri preslikavanja hijerahije klasa prikazane na slici 9 korišćenjem prve i poslednje navedene strategije. Preslikavanje će biti realizovano korišćenjem anotacija.

- Primer preslikavanja hijerarhije klasa korišćenjem strategije jedinstvene tabele za celu hijerarhiju:

Korišćenjem ove strategije, hijerarhija klasa se preslikava u jednu tabelu sa kolonama koje odgovaraju članovima svih klasa koje čine hijerarhiju. Potrebno je uvesti diskriminatorsko polje na osnovu koga se prepoznaje konkretna izvedena klasa. Kolonama koje odgovaraju podacima deklarisanim u potklasama moraju biti dozvoljene nedefinisane (*null)* vrednosti. Ovaj zahtev je neophodan jer prilikom dodavanja objekta jednog tipa, polja koja su navedena u tabeli a odgovaraju objektu drugog tipa imaće vrednost *null*. Da bi se koristila ova strategija, prilikom definisanja anotacija vrednost aptirbuta *strategy* anotacije *@Inheritance* potrebno je postaviti na *InheritanceType.SINGLE_TABLE.* Za preslikavanje nasleđivanja ovim pristupom koriste se sledeće anotacije:

```
@Entity
@Table("PERSON")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="PersonType", discriminatorType
= DiscriminatorType.STRING)

public abstract class Person{
@Id @GeneratedValue
   @Column(name="PersonId")
    private Long id;                          @Column(name="FistName")
    private String firstName;               @Column(name="LastName")
    private String lastName;

@Entity
@Table("CUSTOMER")
@DiscriminatorValue("CP")

public class Customer extends Person {
@Column("Requirements")
    private String Requirements;
    @Column("PhoneNumber")
    private String PhoneNumber;


@Entity
@Table("EMPLOYEE")
@DiscriminatorValue("EP")

public class Employee extends Person {
    @Column(“Salary")
    private Long Salary;
    @Column(“Position")
    private String Position;
```

Polje diskriminator određuje koju potklasu predstavlja red u posmatranoj tabeli. To polje nije polje klase, već se samo koristi za potrebe *Hibernate*-a. U prethodnom navednom primeru, polje *PersonType* je diskriminator i može da ima dve vrednosti *CP* i *EP* na osnovu kojih se određuje da li je u pitanju klasa *Customer* ili *Employee*. *Hibernate* autmatski vrši upis i čitanje vrednosti diskriminatora. Ovaj način preslikavanja je najbolji u pogledu performansi.

- Primer preslikavanja korišćenjem strategije preslikavanja svake konkretne neapstraktne klase u određenu tabelu:

Korišćenjem ove strategije svaka klasa, uključujući i apstraktne klase, preslikava se u sopstvenu tabelu. Svaka tabela sadrži samo kolone koje odgovaraju članovima

odgovarajuće klase, zajedno sa primarnim ključem koji je istovremeno i spoljni ključ u tabeli koja odgovara natklasi. Da bi se koristila ova strategija, prilikom definisanja anotacija vrednost aptirbuta *strategy* anotacije *@Inheritance* potrebno je postaviti na *InheritanceType.JOINED*.

```
@Entity
@Table("PERSON")
@Inheritance(strategy=InheritanceType.JOINED)

public abstract class Person {
  @Id @GeneratedValue
  @Column(name="PersonId")
  private Long id;
  @Column(name="FirstName")
  private String firstName;
  @Column(name="LastName")
  private String lastName;
```

U tabelama koje odgovaraju izvedenim klasama ne mora da se zadaje kolona po kojoj se spajaju tabele ako kolona koja je primarni ključ tabele koja odgovara potklasi ima isto ime kao kolona koja je primarni ključ u tabeli koja odgovara natklasi. Ukoliko imaju različite nazive, kolona primarnog ključa u tabeli koja odgovara izvedenoj klasi označava se anotacijom *@PrimaryKeyJoinColumn*.

```
@Entity
@Table("CUSTOMER")
@PrimaryKeyJoinColumn(name="CustomerId")

public class Customer extends Person {
  @Column("Requirements")
  private String Requirements;
  @Column("PhoneNumber")
  private String PhoneNumber;
 }

@Entity
@Table(name="EMPLOYEE")
@PrimaryKeyJoinColumn("EmployeeID")

public class Employee extends Person {
@Column("Salary")
   private Long salary;
   @Column("Position")
  private String Position;
              }
```

Pored preslikavanja svojstava i nasleđivanja, postoji potreba za preslikavanjem odnosa među klasama tj. preslikavanjem asocijacija. *Hibernate*-u podržava: 1-1, * - * i 1 - * asocijacije. Asocijacije mogu da budu jednosmerne i dvosmerne. Sledi nekoliko primera preslikavanja jednosmernih asocijacija. Preslikavanje će biti definisano korišćenjem *XML* datoteke .

- Preslikavanje asocijacije tipa 1-1 može se ostvariti na nekoliko načina. Jedan od načina je korišćenje elementa *<component>*. Element *<component>* se koristi za predstavljanje zavisnog objekta koji je referenciran od strane perzistentnog objekta.

U nastavku sledi primer preslikavanja asocijacije tipa 1-1.

```
        <class name ="domen.Employee" table="EMPLOYEE">
 <id name="id">
                <generator class="native"/>
        </id>
        <component name="EMPLOYEEADDRESS" class="domen.EmployeeAddress">
                <property name="Address"/>
                <propery name=”Country”>
        </component>
                <property name="FirstName"/>
                <property name="LastName"/>
        </class >
```

Programski kod klasa korišćenih u prethodnom primeru:

```
        public class Employee {
        private int id;
                private EmployeeAddress employeeAddress;
                private string FirstName;
                private string LastName;
        }

        public class EmployeeAddress {
                private string Address;
                private string Country;
                }
```

U ovom slučaju postoji samo jedna tabela u bazi podataka: *Employee* sa kolonama**:** *id*, *FirstName*, *LastName*, *Address*, *Country***.**

Pored navednog načina ostvarivanja preslikavanja veze tipa 1-1, preslikavanje se može izvršiti korišćenjem elemenata *<many-to-one>* i *<one-to-one>*. Ukoliko se koristi element *<many-to-one>*, svaki od objekata između kojih se uspostavlja veza, posmatra se kao posebni perzistentni objekat i svaki ima odgovarajuću tabelu u bazi podataka. To preslikavanje se realizuje preko spoljnog ključa korišćenjem elementa *<many-to-one>*. Preslikavanje koje se ostvaruje korišćenjem elementa *<one-to-one>* realizuje se preko primarnog ključa. U nastavku sledi primer preslikavanja veze 1-1 korišćenjem elementa *<many-to-one>*:

```
  <class name="domen.Employee"
   table="Employee">
      <id name="Id">
        <generator class="native"/>
      </id>
      <property name="FirstName"/>
      <property name="LastName"/>
        <many-to-one name="EmployeeAddress"
        column="id_employeeaddress" unique="true"
        not-null="true"/>
 </class>

 <class name="domen.EmployeeAddress"
   table="EMPLOYEEADDRESS">
```

```
        <id name="Id">
          <generator class="native"/>



        </id>
        <property name="Address"/>
        <property name=”Country”>
    </class>
```

U ovom slučaju u bazi podataka postoji po jedna tabela za svaku od klasa.

Veza tipa 1-1 može se prevesti u vezu tipa *-1 ili u vezu 1-*. Da bi se prevela u vezu tipa *-1, potrebno je ukoniti *unique* ograničenje koje se odnosi na kolonu spoljnog ključa. Prevođenje veze 1-1 u 1-*, može se ostvariti tako sto se objektu koji se nalazi sa leve strane veze 1-* dodeli kolekcija objekata tipa objekta koji se nalazi sa desne strane veze.

- Primer preslikavanja dvosmerne veze tipa 1-*:
  Neka su objekti *Team* i *Employee* u dvosmernoj vezi tipa 1-*, objekti klase *Team* sadržaće listu objekata klase *Employee*. Preslikavanje liste objekta u *XML* datoteci realizuje se korišćenjem elementa *<list>*. U okviru elementa *<list>* potrebno je definisati atribut *name* koji predstavlja naziv atributa klase koji predstavlja kolekciju. Elementom *<key>* koji se nalazi u okviru elementa *<list>* opisuje se koja kolona sadrži strani ključ.

```
<class name="domain.Team" table="TEAM">
<id name="id">
        <generator class="native"/>
</id>
<property name="TeamName"/>
<property name="TeamDescription"/>
        <list name="listEmployee" table="EMPLOYEE" cascade="save-update"
        inverse="true"              lazy="false">
<key column="id_team"/> <index column="id"/>
<one-to-many class="domain.Employee"/> </list>
</class>
```

```
<class name="domain.Employee" table="EMPLOYEE">
        <id name="id">
        <generator class="native"/>
        </id>
        <property name="Salary"/>
        <property name="Position"/>
        <many-to-one name="employee" column="id_employee"/>
</class>
```

Sledi programski kod klasa korišćenih u prethodnom primeru.

```
public class Team
{
        private int Id;
private string TeamName;
        private string TeamDescription;

private List<Employee> listEmployee;
```

```
            public List<Employee> getEmployeeList()
        {
            return listEmployee;
        }
}


public class Employee
{
            private int Id;
            private long Salary;
            private string Position;
}
```

- Veza tipa *-* između klasa X i Y može se implementirati na dva načina. Prvi način podrazumeva dodavanje nove klase Z uz korištenje dve *many-to-one* asocijacije (X -> Y, te Y -> Z). Drugi način podrazumeva korištenje anotacije *@ManyToMany*, gde se automatski pravi nova tabela. Veza implementirana anotacijom *@ManyToMany* može biti jednosmerna i dvosmerna. U slučaju jednosmerne *-* veze, u klasi, u okviru anotacije *@ManyToMany*, potrebno je deklarisati listu objekata druge klase. U slučaju dvosmerne *-* veze**,** potrebno je izabrati nosioca veze koji se razlikuje od drugog člana po tome što ne sadrži atribut *mappedBy* u anotaciji *@ManyToMany*. Klasa koja je izabrana za nosioca veze treba da ima deklarisanu listu objekata tipa klase koja učestvuje u vezi u okviru anotacije *@ManyToMany*. Klasa koja nije izabrana za nosioca veze, u okviru anotacije *@ManyToMany* ima atribut tipa liste objekata deklarisanih u klasi koja je nosilac veze.

  Posmatrajmo primer uspostavljanja veze *-* između eniteta *Employee* i *Course*.
  Uvešćemo dodatnu tabelu *EmployeeCourse*. Tabela *EmployeeCourse* ima složeni primarni ključ, koji se sastoji od primarnih ključeva entiteta *Employee* i *Course*. Za preslikavanje složenih ključeva potrebno je definisati posebnu klasu za primarni ključ, čiji će atributi biti preslikani na kolone koje čine složeni ključ odgovarajuće tabele upotrebom *Column*, *JoinColumn* i *JoinColumns* anotacija. Ta klasa mora da poseduje sledeće karakteristike:

  - Modifikator pristupa klase mora biti javni (*eng. public*);
  - Klasa mora da ima implementiran *Serializable* interfejs;
  - Klasa mora imati definisan javni, podrazumevani konstruktor;
  - U okviru klase potrebno je implementirati *hashCode()* i *equals(Object other)* funkcije.

  Klasa koja odgovara kompozitnom ključu mora biti označena anotacijom *EmbeddedId*.

```
@EmbeddedId
public EmployeeCoursePK getEmployeeCoursePK() {
return this.employeeCoursePK;
}
```

```
@JoinColumn(name = "courseId", referencedColumnName = "courseId", insertable = false,
updatable = false)
@ManyToOne
public Course getCourse() {
        return this.course;
}
@JoinColumn(name = "EmployeeId", referencedColumnName = "EmployeeId", insertable = false,
updatable = false)
@ManyToOne
public Employee getEmployee() {
        return this.employee;
}
```

Složeni primarni ključ, *EmployeeCoursePK*, predstavljen je ključnom reči *@EmbeddedId*, i on predstavlja klasu, u konkretnom slučaju *EmployeeCoursePK()*, sa odgovarajućim poljima. U nastavku sledi primer preslikavanja klase koja odgovara složenom ključu:

```
@Embeddable
public class EmployeeCoursePK implements Serializable {
        @Column(name = "EmployeeId", nullable = false)
        public String getEmployeeId() {
                return this.employeeId;
        }
        @Column(name = "CourseId", nullable = false)
        public String getCourseId() {
                return this.courseId;
        }
```

4.6. Načini izražavanja upita u *Hibernate*-u

Hibernate podržava tri načina za izražavanje upita. To su:

1. *Criteria API* koji omogućava formiranje ugnježdenih upita, struktuiranih upitnih izraza u *Javi*. Pruža proveru sintakse u vreme kompajliranja što nije moguće kod *HQL* i *SQL* upitnih jezika. *Criteria* interfejs definiše nekoliko metoda kojima se mogu formirati upiti. Konkretnu implementaciju ovog interfejsa dobijamo preko *Session* objekta.

   Primer korišćenja *Criteria API*:

   ```
   session.createCriteria(Person.class).add(Restrictions.like("FirstName","Marin%"));
   ```

2. Direktan *SQL* sa automatskim preslikavanjem skupa rezultata u objekte ili bez njega. Za ovaj pristup potreban je *org.Hibernate.SQLInterface*, koji proširuje *org.Hibernate.Query* interfejs. Kroz aplikaciju *SQL* upit se formira metodom *createSQLQuery() Session* objekta. Nakon prosleđivanja teksta koji sadrži *SQL* upit, *createSQLQuery()* metodi, potrebno je povezati *SQL* rezultat sa nekim entitetom.

Primer:

```
session.createSQLQuery("select {p.*} from Person {p} where FirstName like Marin   o
%'").addEntity("p", Person.class);
```

3. *Hibernate Query Language* (*HQL*) koji je detaljnije opisan u narednom poglavlju.

4.7. Hibernate upitni jezik HQL

HQL je objektno-orijentisan upitni jezik. Sličan je *SQL*-u, ali pored operacija nad tabelama i kolonama, *HQL* radi sa perzistentnim objektima i njihovim atributima. *HQL*-om se mogu izdvajati podaci iz baze, za šta se koristi naredba *select*, ili se mogu menjati podaci u bazi za šta se koriste naredbe *inset*, *update* i *delete*. *HQL*-om se ne može promeniti struktura baze. *HQL* upiti se prevode u *SQL* upite, a *Hibernate* okvir ima mogućnost da direktno koristi *SQL* upite.

Za pripremu *HQL* upita koristi se metoda *createQuery()*. Metode *createQuery()* i *createSQLQuery()* objekta *Session* se koriste da bi se napravila nova instanca *Hibernate* objekta *Query*.

Primer pripreme *HQL* upita:

```
Query createHql = session.createQuery("from Person");
```

Hibernate vraća objekat *Query* koji se može iskoristiti za zadavanje načina izvršavanja upita.

Upit je spreman za izvršavanje nakon pripreme objekta *Query*. Za izvršavanje upita najcešće se koristi metoda *list()* interfejsa *Query*. Ova metoda vraća rezultate kao *java.util.List*:

```
List result = createHql.list();
```

Metoda *uniqueResult()* se koristi u slučaju kada znamo da će rezultat upita biti samo jedna instanca. Ako upit vrati više od jednog objekta, biće prijavljen izuzetak, dok će u slučaju da ne nađe ništa, metoda vratiti *null*.
Query i *Session* interfejs obezbeđuju *iterate()* metod, koji vraća isti rezultat kao *list()* ali koristi drugačiju strategiju vraćanja rezultata. Kada se koristi metod *iterate()* za izvršavanje upita, *Hibernate* vraća samo primarni ključ (identifikator) u prvom *SQL select*-u, pokušava da nađe ostatak objekta u kešu pre nego sto ponovo upita za ostatak *property* vrednosti. Metoda *iterate()* vraća rezultata tipa *Iterator*. Ako upit sadrži više redova rezultata, rezultat ce biti vraćen kao instanaca od klase *Object[]*.

Hibernate podržava tehniku imenovanih parametara (*eng. named parameters*). Korišćenje imenovanih parametara omogućava lakše pisanje upita koji se zasnivaju na ulaznim podacima korisnika.

Primer korišćenja imenovanog parametra:

```
String queryString = "from Item item where item.description like :search";
```

Imenovani parameter se označava dvotačkom. Imenovani parametar možemo povezati sa nekom vrednošću. Primer:

```
Query q = session.createQuery(queryString).
    setString(":search", searchString);
```

Metoda *setString()* interfejsa *Query* se koristi da bi se neka promenljiva, u ovom primeru *searchString*, povezala sa imenovanim parametrom (:*search*). Osim pomenute metode *setString()*, interfejs *Query* poseduje i druge metode za povezivanje argumenata većine tipova koje *Hibernate* podržava kao što su : *setInteger()*, *setTimestamp()*, *setLocale()*. Postoji metoda *setParameter()* koja automatski prepoznaje tip parametra pa time nije obavezno korišćenje prethodne tri metode.

Interfejs *Query* podržava i metod *setEntity()*, koji omogućava korišćenje nekog perzistentnog entiteta (preslikane klase).

Primer korišćenja *setEntity()* metode:

```
session.createQuery("from Person person where person.FirstName =
:Name").setEntity("Name", instancePerson);
```

U okviru upita može se koristiti vise imenovanih parametara. Na primer:

```
    String queryString = "from Item item"+ " where item.description like :search" + " and
    item.date > :minDate";
Query q = session.createQuery(queryString).
    setString("search", searchString).setDate("minDate", mDate);
```

Klauzula *Select* u *HQL*-u se koristi za izdvajanje svojstava objekta:

```
String hql=”select p.FirstName from Person p”;
Query query=session.createQuery(hql);
List results=query.list();
```

Klauzula *From* određuje izvor podataka nad kojim će se izvršavati upit:

```
from Person
```

Ovim upitom se generiše sledeći *SQL*:

```
select p.FistName. p.LastName, ... from Person p
```

Postoji mogućnost dodeljivanja aliasa klasi koja se koristi:

```
from Person as person
```

Za to se koristi ključna reč *as*. Korišćenje ključne reči *as* je opciono pa se prethodni zapis može predstaviti i ovako:

```
from Person person
```

Ukoliko je potrebno izdvojiti samo određene instance klase, klauzulom *where* se definiše restrikcija:

```
from Person p where p.FistName = 'Petra'
```

Ovim se generiše sledeći *SQL* upit:

```
select p. BusinessEntityId, p.FirstName, p.LastName
from Person p where p.FistName = 'Petra'
```

U izraze se mogu uključiti konstante. Često korišćene konstante u *HQL*-u su *true* i *false*:

```
from Person p where p.IsWoman = true
```

U *where* klauzuli se mogu koristiti sledeći izrazi:

1. *HQL* imenovani parametri;
2. *SQL* funkcije za datum i vreme: *current_time()*, *current_date()*, *current_timestamp()*;
3. *SQL* funkcije: *length()*, *upper()*, *lower()*, *ltrim()*, *rtrim()*, itd.

HQL podržava osnovne operatore za poređenje kao i *SQL*: =, <>, <, >, >=, <=, *between*, *not between*, *in* i *not in*.

Primer korišćenja operatora poređenja:

```
from Person p where p.BusinessEntityId between 200 and 1000
from Person p where p.BusinessEntityId > 1000
from Person p where p.FirstName in ('Petar', 'Petra')
```

HQL podržava operator *is[not] null*:

```
from Person p where p.BusinessEntityId is null
```

(u ovom primeru posmatramo sve osobe kojima je *BusinessEntityId* jednak *null*)

```
from Person p where p.BusinessEntityId is not null
```

(izdvajamo sve osobe sa *BusinessEntityId*–em koji je različit od *null*)

Operator *[not] like* podržava korišćenje znakova koji su isti kao u *SQL*-u: % i _ (procenat zamenjuje proizvoljan niz znakova, a _ jedan znak).

```
from Person p where p.FistName like 'Pet%'
```

Logički operatori (i zagrade za grupisanje) koriste se za kombinovanje izraza:

```
from Person p where p.FistName like 'Pet%'  or p.FirstName in ('Marija', 'Ana')
```

HQL podržava i pozivanje *SQL* funkcija. Primer za to je izraz koji izdvaja veličinu kolekcije:

```
from PeopleList pl where size(pl.Person) > 3
```

Slično kao u *SQL*-u, i *HQL* podržava klauzulu *Order by*.

```
from Person p order by p.FirstName
```

(ovaj upit vraća sve osobe sortirajući ih prema imenu)

Sortiranje je moguće izvršiti u rastućem ili opadajućem redosledu. Podrazumevani redosled je rastući, a ukoliko je potrebno opadajuće uređenje, potrebno je navesti ključnu reč. Ključna reč za rastući redosled je *asc*, a za opadajući *desc*.

Primer izdvajanja osoba uređujući ih prema imenu u opadajućem redosledu:

```
from Person p order by p.FirstName desc
```

Za uređivanje se može koristiti i više svojstava:

```
from Person p order by p.FirstName asc, p.LastName asc
```

HQL podržava klauzulu *GroupBy*. Ova klauzula omogućava dobijanje podataka iz baze i njihovo grupisanje na osnovu vrednosti atributa.

Primer:

```
String hql=”select sum(p.FirstName.Length), p.FistName from Person p” + “group by
p.FistName”;
Query query=session.createQuery(hql);
List results=query.list();
```

HQL podržava slične agregatne funkcije kao *SQL*. One čak i rade na sličan način, jedina razlika je u tome što se u *HQL*-u ove metode primenjuju nad poljima objekta. Ključna reč *distinct* broji samo objekte sa jedinstvenom vrednošću. Svi ovi upiti vraćaju ceo broj kao rezultat i za njegovo dobijanje se koristi metoda *uniqueResult()*. Dostupne funkcije u *HQL*-u su:

avg(ime polja): Prosečna vrednost polja;
count(ime polja ili *): Broj pojavljivanja polja u rezultatu;
max(ime polja): Maximalna vrednost svih polja;
min(ime polja): Minimalna vrednost svih polja;
sum(ime polja): Suma vrednosti svih polja.

U slučaju da u upitu postoje više od jedne agregatne funkcije, rezultat će biti lista objekata sa vrednostima traženih funkcija:

```
select min(person.FirstName.Length), max(person.FirstName.Length) from Person person
```

Operacija projekcija je još jedna od mogućnosti *Hibernate*. Na primer:

```
from Factory f, Person p
```

Ovaj upit daje uređene parove instanci *Factory* i *Person* kao listu *Object[]*. Na poziciji 0 je *Factory*, a indeks 1 je *Person*. Ovde se radi o Dekartovom proizvodu tabela, pa iz tog razloga rezultat sadrži sve moguće kombinacije zapisa *Factory* i *Person*.

```
Query q = session.createQuery("from Factory f,  Person p");
Iterator parovi = q.list().iterator();
while (parovi.hasNext() ) {
Object[] par = (Object[]) parovi.next();
Factory f = (Factory) par[0];
Person p = (Person) par[1];
```

Operacija *Join* je jedna od osnovnih prednosti relacionog modela podataka i predstavlja mogućnost spajanja podataka iz različitih tabela. Spajanje (*eng. join*) takođe omogućava dohvatanje nekoliko povezanih objekata i kolekcija u jednom upitu. *Join* se koristi za kombinovanje podataka u dve (ili više) relacija. *Hibernate* podržava *inner join, left outer join, right outer join, full join*.

4.8. Načini razvoja uz pomoć Hibernate-a

Hibernate podržava sledeće načine razvoja zahvaljujući svojim pomoćnim alatima:

1. Od vrha ka dnu (*eng. top–down*)
   Kod ovog tipa razvoja pravi se šema baze podataka od postojećeg modela koji je već implementiran u *Javi*. U ovom razvojnom tipu objekti određuju oblik šeme podataka u bazi;
2. Odozdo-navise (*eng. bottom–up*)
   Kod ovog načina razvoja kreće se od već definisane relacione šeme i modela, a zatim se korišćenjem alate izvlače metapodaci iz baze podataka koji služe za pravljenje datoteke za preslikavanje a zatim i objekata u *Javi*;
3. Od sredine *(eng. middle-out)*
   Kod ovog načina kreće se od *XML* datoteka za preslikavanje, a zatim pomoću alata prave šema i potrebne klase. Glavne izmene se vrše na datotekama za preslikavanje, a pomoću alata menja ostatak aplikacije;
4. Susret u sredini *(eng. meet – in-middle)*
   Ovaj način razvoja podrazumeva spajanje već postojeće *Java* klase sa već postojećim šemama podataka. U ovakvom slučaju *Hibernate* ne pomaže mnogo, već je potrebno prilagođavati i klase i šemu radi uspostavljanja veza. Datoteke za preslikavanje se u ovom slučaju formiraju ručno.

5. Poređenje Entity Framework-a i Hibernate-a

Entity Framework predstavlja alat koji se koristi kod .*NET* aplikacija, dok *Hibernate* predstavlja rešenje za objektno-relaciono preslikavanje kod *Java* okruženja. Zbog velike popularnosti i

korisničkih zahteva, *Hibernate* alat je proširen portom koji je nazvan *NHibernate* i koristi se za objektno-relaciono preslikavanje na .*NET* platformi.

U nastavku rada uporedićemo ova dva alata prema određenim kriterijumima.

- Istorijski razvoj

  Prva verzija *Entity Frameworka* (*EFv1*) objavljena je 1.avgusta 2008.godine i bila je uključena u pakete .*NET Framework 3.5.Serice Pack 1* i *VisualStudio2008ServicePack1*. Druga verzija *Entity Framework*-a, *Entity Framework 4.0*, objavljena je 12.aprila 2010.godine kao deo .*NET* 4.0 i, u odnosu na prethodnu verziju, sadržala je novine kao što su: *Model-first*, *POCO* i *Lazy loading*. Treće izdanje *Entity Framework*-a, verzija 4.1, objavljeno je 12.aprila 2011.godine sa podrškom za *Code First* pristup. Verzija 4.1 *Entity Framework*-a, nazvana *Entity Framework 4.1Update 1*, pojavila se 25.jula 2011.godine i sadržala je ispravke prethodne verzije. Verzija 4.3.1 objavljena je 29.februara 2012.godine, a zatim 11.avgusta 2012.godine verzija 5.0.0. Novine u odnosu na prethodnu verziju bile su: podrška za *Enum* tip podataka u *Code first* pristupu, *Enitity Framework* dizajner *itd*. Poslednja verzija, verzija 6.0, pojavila se 17.oktobra 2013.godine i donela je podršku za transakcije, *SQL* logovanje itd.

  Hibernate se prvi put pojavio 2001.godine. 2003.godine počelo se sa razvojem drugog izdanja *Hibernate*-a, koje je donelo značajnija poboljšanja u odnosu na prvo izdanje. 2005.godine, pojavila se verzija 3.0, koja je uključila nove *Interceptor/Callback*, korisnicki definisane interfejse, i *JDK 5.0* anotacije. Od 2010, *Hibernate 3*, verzija 3.5.0 i iznad nje, sadržale su *Java Persistence API 2.0*. U decembru 2010.godine počeo je da se koristi *Hibernate Core 4.0.0*. *Hibernate 4.1.9* počeo je da se koristi decembra 2012. Razvoj verzije *Hibernate 5* počeo je 2012.godine. *Hibernate* je 2006.godine portovan za .*NET* platformu pod nazivom *NHibernate*.

- Sesija podataka

  Sesija podataka upravlja učitavanjem objekata u memoriju i izmenama koje se vrše nad tim objektima, pri čemu se sinhronizacija ovih objekata sa bazom vrši jedino u slučaju eksplicitnog zahteva. Sesija podataka predstavlja klasu pri radu sa bazom podataka koja vrši automatsko upravljanje konekcijama, transakcijama i konkurentnim pristupom. *Entity Framework* i *Hibernate* sadrže načine za privremeno smeštanje preslikanih objekata u memoriju. *Entity Framework* koristi klasu *ObjectContext*, a *Hibernate* koristi klasu *Session* koja upravlja preslikanim objektima. Kada se preslikani objekat napravi u memoriji, povezivanje sa sesijom podataka vrši se eksplicitno. Pristupanje određenoj tabeli u bazi vrši se kroz odgovarajući objekat u memoriji, i pri tome se prate sve izmene koje se vrše nad objektom u memoriji do trenutka zahteva za sinhronizaciju sa bazom. Nakon što se sesija uništi, objekti nastavljaju da postoje u memoriji, ali ne postoji njihova sinhronizacija sa bazom. I kod *EntityFramework*-a i kod *Hibernate*-a inicijalizacija sesije podataka je skupa operacija u pogledu vremena potrebnog za izvršenje. Ova operacija podrazumeva uspostavljanje veze sa bazom podataka i pravljenje planova kako će se izvršavati upiti. Inicijalizacija sesije podataka ne obavlja se na isti način kod ova dva alata. *Entity Framework* implicitno vrši inicijalizaciju sesije podataka u trenutku kada treba da se izvrši prvi upit, dok *Hibernate* vrši inicijalizaciju eksplicitnim pozivom

određene metode. Rešenje koje *Hibernate* koristi za inicijalizaciju sesije smatra se fleksibilnijim rešenjem, jer omogućava pozivanje inicijalizacije sesije podataka u toku pokretanja programa. Korisniku aplikacije je bolje rešenje koje nudi *Hibernate* jer rešenje koje nudi *Entity Framework* podrazumeva čekanje od nekoliko sekundi prilikom prvog izvršenja upita nakon pokretanja aplikacije.

- Transakcije i konkurentni pristup

Entity Framework i *Hibernate* obezbeđuju način rada sa transakcijama koji je poznat kao *autocommit* način. Kod ovog načina rada, baza podataka svaku naredbu posmatra i izvodi kao zasebnu transakciju. *ACID* svojstva transakcija ostvaruju se korišćenjem transakcija *DBMS*-a, kao i njihovim ekvivalentima koji se formiraju na strani aplikacije. To osigurava njihovu obradu, a sam *DBMS* garantuje konzistentnost ograničenjima koja se implementiraju na starni *DMBS*-a. Ukoliko ta ograničenja nisu implementirana, potrebno je obezbediti validaciju podataka na strani aplikacije. Bitno svojstvo transakcije, izolaciju transakcije, treba obezbediti zbog problema istovremenog pristupa kod višekorisničkih aplikacija, ili kod sistema u kojima više aplikacija pristupa istoj bazi podataka. U oba slučaja najpouzdanije je koristiti sistem zaključavanja koji se implementira od strane *DBMS*-a. Sistem zaključavanja se može primeniti na celu bazu podataka, na tabelu, red tabele ili skup redova tabele. *Entity Framework* i *Hibernate* konkurentni pristup obezbeđuju kroz mehanizme za optimističko (*eng. optimistic locking*) i pesimističko zaključavanje (*eng. pesimistic locking*). Ovi mehanizmi uvek koriste zaključavanje na nivou baze (ne zaključavaju se objekti u memoriji). *Entity Framework* i *Hibernate* podrazumevano koriste optimističko zaključavanje. Kod optimističkog zaključavanja pretpostavlja se da više transakcija može da se izvrši bez uticaja jednih na druge. Sistem proverava da li više korisnika pokušava istovremeno da izmeni isti zapis, i ako utvrdi da je došlo do takve situacije, izmene jednog korisnika će biti prihvaćene, a izmene ostalih korisnika odbačene, i korisnici obavešteni o tome. Kod pesimističkog zaključavanja sistem kreće od pretpostavke da više korisnika menjaju istovremeno isti zapis u bazi, i sprečava tu mogućnost zaključavanjem zapisa. Ključevi se postavljaju na zapis čim se pristupi zapisu. U zavisnosti od tipa ključa koji se koristi ostali korisnici će možda moći da čitaju podatke čak iako je red zaključan.
.
Pristup bazi podataka je skupa operacija. Čak i za jednostavan upit, zahtev često mora biti poslat preko mreže do servera. Iz tog razloga, da bi se izbeglo često pristupanje bazi, *Entity Framework* i *Hibernate* podržavaju keširanje podataka. S obzirom da *Entity Framework* i *Hibernate* podržavaju keširanje podatka, ukoliko više aplikacija koristi istu bazu podataka, postavlja se pitanje kako određena aplikacija zna da je podatak izmenjen od strane neke druge aplikacije i da treba da pročitata novu verziju tog podatka a ne onu koja je keširana. *Entity Framework* i *Hibernate* obezbeđuju rešenje za ovakve situacije. *Entity Framework* koristi određenu kolonu za kontrolu konkurentnosti. Za ovu namenu uobičajno je korišćenje kolone tipa *Timestamp.* Nakon inicijalizacije nove transakcije, proverava se vrednost te kolone sa starom vrednošću. Ukoliko su te vrednosti jednake, biće potvrđeno uspešno izvršenje transakcije. U slučaju različitih vrednosti kolone, biće prijavljena greška. Baza podataka automatski menja vrednost ove kolone svaki put kada se dodaje novi slog ili menja postojeći.

U nastavku sledi primer upravljanja konkurentnošću u *Entity Framework*-u korišćenjem odgovarajuće kolone.

Neka dva korisnika pokušavaju da urade izmene na nekom entitetu:

```
Person person1WithUser1 = null;
Person person1WithUser2 = null;
//prvi korisnik učitava sve podatke o entitetu tipa Person čija je vrednost svojstva PersonId jednaka 1
using (var context = new EntityFrameworkExampleEntities())
{
        context.Configuration.ProxyCreationEnabled=false;
        person1WithUser1=context.People.Where(s=>s.PersonId==1).Single()
}
//drugi korisnik učitava sve podatke o entitetu tipa Person čija je vrednost svojstva PersonId jednaka 1
using (var context = new EntityFrameworkExampleEntities())
{
        context.Configuration.ProxyCreationEnabled=false;
        person1WithUser2=context.People.Where(s=>s.PersonId==1).Single()
}
//prvi korisnik menja svojstvo FirstName entiteta
person1WithUser1.FirstName=”Izmenjeno od strane prvog korisnika;
//drugi korisnik menja svojstvo FirstName entiteta
person1WithUser2.FirstName=”Izmenjeno od strane drugog korisnika;
```

Prvi korisnik je promenio podatak. Ako drugi korisnik pokuša da uradi izmene, biće prijavljena greska tipa *DbUpdateConcurrencyException.*

```
//prvi korisnik čuva izvršene izmene
using (var context = new EntityFrameworkExampleEntities())
{
    try
    {
        context.Entry(person1WithUser1).State=EntityState.Modified;
        context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        Console.WriteLine("Optimistic Concurrency exception occurred”);
}
}

//drugi korisnik pokušava da sačuva izmene
using (var context = new EntityFrameworkExampleEntities())
{
    try
    {
        context.Entry(person1WithUser2).State=EntityState.Modified;
        context.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
```

```
            Console.WriteLine("Optimistic Concurrency exception occurred”);
    }
```

Drugi korisnik neće uspeti da sačuva izmene zbog vrednosti kolone koja se koristi za kontrolu konkurentnog pristupa. Njena vrednost je automatski promenjena prilikom izmene entiteta od strane prvog korisnika .

Hibernate koristi verzioniranje kolone radi kontrole izmenjenih podataka prilikom konkurentnog pristupa. Kolona koja se koristi za verzioniranje označava se korišćenjem *@Version* anotacije ili taga *<Version>*, u zavisnosti da li se definisanje preslikavanja radi korišćenjem anotacije ili *XML* datoteke. U nastavku sledi primer:

```
@Entity
@Table(name = ”PERSON”)
public class Person{
        @Id
        private long id;
        @Version
        private int version;
        private String firstName;
        private String lastName;
}
```

Prilikom izmene entiteta, *Hibernate* menja vrednost verzionirane kolone korišćenjem sledećeg upita:

```
update PERSON
set firstName=?, lastName=?, version=?
where id=? and version=?
```

Ukoliko jedan korisnik pokuša da izvrši izmene u bazi izvršenjem sledećeg upita:

```
update PERSON
set firstName=?, lastName=?, version=2
where id=? and version=1
```

Izmene će biti prihvaćene i vrednost kolone koja se koristi za verzioniranje biće izmenjena.
Ukoliko drugi korisnik u tom trenutku pokuša da izvrši izmenu nad istim slogom u bazi:

```
update PERSON
set firstName=?, lastName=?, version=2
Where id=? and version=1
```

s obzirom da je prvi korisnik već izmenio vrednost kolone *version*, broj slogova u bazi koji zadovoljavaju uslov upita drugog korisnika jednak je nuli. U tom slučaju, *Hibernate* će prijaviti grešku tipa: *org.hibernate.StaleObjectStateException*.

- Strategije nasleđivanja
  Izbor korišćenja određenog pristupra za preslikavanje hijerarhije objekata zavisi od dubine hijerarhije nasleđivanja, preklapanja među klasama i potrebama za čestom promenom hijerahije. I *Entity Framework* i *Hibernate* podržavaju tri pristupa preslikavanja hijerarhije objekata. To su:

1. Preslikavanje cele hijerarhije klasa u jednu tabelu - predstavlja pristup kod kog se hijerarhija klasa preslikava u jednu tabelu baze podataka, koja ima posebnu kolonu na osnovu koje se određuje koju potklasu predstavlja red u toj tabeli. Ovaj pristup se koristi kod jednostavnijih ili plitkih hijerarhija.

   Prednosti ovog pristupa su:
   - jednostavan pristup;
   - brži pristup podacima jer se nalaze u jednoj tabeli;
   - jednostavnije je dodavanje nove klase uz dodavanja nove kolone za dodatne podatke.

   Nedostaci ovog pristupa su:
   - Promena u jednoj klasi utiče na tabelu i na prikaz ostalih klasa;
   - U slučaju veće hijerahije, broj zapisa u bazi se povećava.

   Ovaj pristup je podrazumevani prlikom preslikavanja hijerarhije objekata u *Entity Frarmework*-u i *Hibernate*-u.

2. Preslikavanje svake konkretne neapstraktne klase u određenu tabelu – ovaj pristup podrazumeva da se svi atributi klasa, uključujući i atribute koje klasa nasleđuje, preslikavaju u kolone jedne tabele. Koristi se kod retkih preklapanja među tipovima podataka ili promena nad njima.

   Prednosti ovog pristupa su:
   - brz pristup podacima jednog objekta;

   Nedostaci ovog pristupa su:
   - promene na klasi dovode do promena odgovarajuće tabele u bazi, ali i tabela koje odgovaraju potklasama;
   - kada dođe do promene tipa objekta, potrebno je izvršiti kopiranje podataka u drugu tabelu;
   - postoje problemi kod održavanja integriteta podataka za objekte koji imaju više uloga (tj. postoje slogovi u dvema tabelama za taj tip objekta).

3. Preslikavanje svake klase/potklase (uključujući i apstraktne klase) u određenu tabelu – ovaj pristup podrazumeva realizaciju hijerarhije nasleđivanja klasa preko spoljnih ključeva. Primarni ključevi u tabelama koje predstavljaju potklase su spoljni ključevi preko kojih se referencira tabela koja predstavlja natklasu. Koristi se kod češćih preklapanja među tipovima podataka i u slučajevima potrebe za češćim promenama tipova.
   Prednosti ovog pristupa su:

- jednostavnija izmene natklasa i dodavanje novih potklasa zbog postojanja jedne tabele;
- broj zapisa u bazi raste proporcijalno broju objekata;
- jednostavnija je šema baze podataka zbog preslikavanja 1-1.

Nedostaci ovog pristupa su:

- postojanje većeg broja klasa stvara potrebu za velikim brojem tabela u bazi, po jedna za svaku klasu;
- potencijalno usporenje pri čitanju i izmeni podataka zbog pristupa većem broju tabela.

- Načini postavljanja upita

Entity Framework i *Hibernate* obezbeđuju različite metode za postavljanje upita. Među najčešće korišćene metode spadaju: *SQL* izvedeni upiti i objektni upiti. *SQL* izvedeni upiti su proširenje standardnih *SQL* upita koji omogućavaju pravljenje upita nad preslikanim objektima umesto nad tabelama baze podataka. *Entity Framework* koristi *Entity SQL*, a *Hibernate HQL* izvedeni *SQL* jezik. Oba alata kao rezultat vraćaju listu preslikanih objekata. Ovaj način predstavljanja upita ima svoje nedostake, od kojih je najveći nemogućnost otkrivanja ispravnosti upita predstavljenog kao niska karaktera (*eng. string*). Greške u takvim upitima se mogu videti tek u toku izvršenja programa, što je teže za otkrivanje i ispravljanje grešaka. Pored izvedenih *SQL* upita, koristi se i drugi način za postavku upita. To su objektni upiti. Od objektnih upita, *Entity Framework* koristi *LINQ*, dok *Hibernate* koristi *Critera API*.

- Baze podataka

Entity Framework podržava samo *SUBP MS SQL Server*, ali postoje brojni *ADO.NET* provajderi za indirektan pristup nekim drugim bazama podataka kao što su *Oracle* i *MySQL*.
Hibernate podržava sledeće baze podataka i njihove *SQL* dijalekte: *Oracle*, *DB2*, *Sybase*, *MS SQL Server*, *PostgreSQL*, *MySQL*, *Hypersonic SQL*, *Mckoi SQL*, *SAP DB*, *Interbase*, *Pointbase*, *Progress*, *FrontBase*, *Ingres*, *Infomix* i *Firebird*.

- Asocijacije

Sto se tiče asocijacija, i *Entity Framework* i *Hibernate* podržavaju: 1-1, * - * i 1 - * asocijacije.

- Keširanje

I *Entity Framework* i *Hibernate* podržavaju prvi nivo keširanja. *Hibernate* podržava i drugi nivo keširanja, dok *Entity Framework* nema ugrađenu podršku za drugi nivo keširanja. Drugi nivo keširanja služi za keširanje upita. Rezultati *SQL* komandi se smestaju u keš, tako da se za iste *SQL* komande vraćaju podaci iz keša a ne vrši ponovno izvršenje upita sto utiče na povećanje performansi aplikacije. U *Entity Framework*-u postoji mogućnost podržavanja drugog nivoa keširanja korišćenjem određenih provajdera kao što je npr. *EFCachingProvider*-a.

- Strategije za generisanje primarnih ključeva

Entity Framework nudi tri strategije za generisanje primarnih ključeva. To su: korišćenje univerzalnog jedinstvenog identifikatora (*eng. Globally Unique Identifier-GUIDs*), ručno generisanje primarnih ključeva i korišćenje identifikacione (*eng. IDENTITY)* kolone pri čemu se automatski generiše primarni ključ. *Hibernate* nudi otprilike desetak strategija za generisanje primarnih ključeva. Pored ovih navedenih strategija za generisanje ključeva kod *Entity Framework*-a, *Hibernate* nudi još načina za generisanje primarnih ključeva kao što su: odabir adekvatne strategije, prema bazi koja se koristi, za generisanje vrednosti ključa (*eng. AUTO*), automatsko generisanje vrednosti ključa pre nego se objekat sačuva korišćenjem određene kolone (*eng. SEQUENCE*) ili automatsko generisanje korišćenjem standardne tabele u bazi za određivanje vrednosti ključa (*eng. TABLE*), itd.

6. Zaključak

U radu je razmatran problem predstavljanja relacionih baza podataka primenom objektno-orijentisanih koncepata unutar aplikacije. Predstavljene su mogućnosti koje nude alati *Entity Framework* i *Hibernate* kao i poređenje ovih alata.
Entity Framework predstavlja jednu od glavnih .*NET* tehnologija za rad sa podacima. Kao i ostali *ORM* alati omogućava pristupanje relacionoj bazi podataka pomoću objektno–orijentisanih koncepata u aplikaciji što je pogodnije za primenu poslovne logike. Korisnik više ne mora sam da preslikava strukturu baze podataka u klase aplikacije, nego to za njega radi *Entity Framework*. Na taj način se generiše veliki deo koda koji je ranije morao da piše sam programer, čime se ubrzava proces izrade aplikacija. Zahvaljujući brojnim poboljšanjima, *Entity Framework* je jedna od glavnih .*NET* tehnologija za objektno-relaciono preslikavanje.

Hibernate je od strane programera ocenjen kao odličan okvir za implementaciju *ORM*-a, moćan i jednostavan za upotrebu, uz zanemarljive nedostatke u performansama. On omogućava rukovanje redovima i kolonama iz baze podataka na objektno-orijentisan način bez velikog korišćenja standardnog *SQL*-a. Pravilnom upotrebom *Hibernate*-a od strane programera, bez obzira na način razvoja i zahteve softvera, može se uštedeti mnogo vremena, dajući kodu preglednost, omogućavajući lako održavanje izmena koda, što predstavlja i glavne karakteristike svake dobro razvijene aplikacije.

7. Literatura

Julia Lerma