IEGULDĪJUMS TAVĀ NĀKOTNĒ!

Eiropas Reģionālās attīstības fonds

Prioritāte: 2.1. Zinātne un inovācijas

Pasākums: 2.1.1. Zinātne, pētniecība un attīstība

Aktivitāte: 2.1.1.1. Atbalsts zinātnei un pētniecībai

Projekts: ”Multi - modeļu izstrādes tehnoloģija .NET pielietojumu projektiem”

Projekta sākuma datums: 2014.gada 1.janvāris.

Projekta beigu datums: 2015.gada 30.jūnijs.

Līguma Nr. 2013/0031/2DP/2.1.1.1.0/13/APIA/VIAA/010

ESF finansējuma saņēmējs: SIA, SWH SETS

Sadarbības partneris: Elektronikas un datorzinātņu institūts (EDI)

Projekta aktivitātes Nr.3.4 "MS Entity Framework integrācija" progresa pārskats

Pārskats Nr.15. par periodu no 2014.gada 1.jūlija līdz 2014.gada 31.decembrim.

SATURS

1. Kopsavilkums

Pārskata periodā (2014-07-01 – 2014-12-31.) projekta „Multi - modeļu izstrādes tehnoloģija .NET pielietojumu projektiem" aktivitātes Nr.3.4 "MS Entity Framework integrācija" ietvaros ir pabeigti iepriekšējā pārskata periodā iesāktie darbi.

2. Ievads

Šis pārskats ir veltīts projekta apakšaktivitātes Nr.3.4 "MS Entity Framework integrācija" ietvaros paveiktajam šajā un iepriekšējā pārskata periodā.

Tās ietvaros ir aprakstīta Object-Relational Mapping tehnoloģija, dažādās pieejas darbam ar Entity Framework, kā arī pamatota konkrētās pieejas izvēle. Pielikumā ir pievienots POCO objekta piemērs, kā arī Sakarību un īpašību definīcijas koda fragments, izmantojot Fluent API interfeisu.

3. Object-Relational Mapping

Object-Relational Mapping (ORM) ir programmēšanas tehnoloģija, kurā datus no datubāzēm sasaista ar objektiem pēc objektorientētās programmēšanas principiem, rezultātā iegūstot virtuālu objektu datubāzi [1]. Izmantojot ORM, lietotājprogrammu izstrādātājam nav jāraksta zema līmeņa SQL vaicājumi, bet tiek izmantotas objekta metodes. ORM tiek izmantots kā datu glabātuves abstrakcija.
Eksistē daudzi rīki un bibliotēkas dažādām programmēšanas vidēm, ar kuru palīdzību var vaidot lietotājprogrammas datu pieejas līmeni, izmantojot ORM tehnoloģiju.
Kā ORM priekšrocības var minēt:

- **Produktivitāte:** Datu pieejas līmenis ir svarīga un arī koda apjoma ziņā ievērojama lietojumprogrammas daļa, kas arī aizņem ievērojamu daļu laika no lietojumprogrammas izstrādes. ORM izmantošana ļauj samazināt laiku, kas nepieciešams datu pieejas līmeņa programmēšanai.
- **Lietojumprogrammas dizains:** ORM tehnoloģija liek programmētājiem turēties pie labas programmēšanas prakses, datubāzes pieejas līmeni atdalot no biznesa loģikas.
- **Koda atkalizmantošana:** Izvietojot datu pieejas līmeni, izmantojot OMR tehnoloģiju un ievietojot atsevišķā bibliotēkā, to iespējams izmantot vairākās lietojumprogrammās.
- **Lietojumprogrammas uzturēšana:** Ar ORM koda ģenerācija ir labi testēta. Tas nozīmē, ka mainot datubāzes struktūru, nav jāuztraucas par to, kā tas ietekmēs datu pieejas līmeni.

Kā trūkumu ORM rīkiem varētu minēt iespējamo veiktspējas samazināšanos, lietojot ar ORM rīkiem ģenerētu datu pieejas bibliotēku. Ģenerētais kods ir sarežģītāks, salīdzinot ar kodu, kuru būtu rakstījis programmētājs. Tas nozīmē, ka ģenerētais kods arī izpildīsies lēnāk. Labi ORM rīki šo veiktspējas zudumi ir minimizējuši.
Nākošajā tabulā ir apkopoti populārākie ORM rīki .NET videi:

Bibliotēka	**Licence**
ADO.NET Entity Framework	Apache License v2 (brīvi izplatāms atvērtais kods)
Nhibernate	GNU Lesser General Public License brīvi izplatāms atvērtais kods)
DataObjects.Net	Commercial
Telerik Data Access	Telerik End User License Agreement for Data Access (brīva)
Subsonic	Subscription based

Pirms Microsoft radīja ADO.NET Entity Framework [2], populārākais ORM rīks .NET videi bija NHibernate [3]. NHibernate ir portēts no Hibernate, kas ir ORM rīks Javai.
Internetā ir atrodami daudzi raksti, kuros salīdzināti ADO.NET Entity Framework un NHibernate [4] [5]. Lai arī vēl pavisam nesen NHibernate pārspēja ADO.NET Entity Framework iespēju ziņā, līdz ar pēdējo Entity Framework versiju iznākšanu, abu rīku iespējas ir izlīdzinājušās. Gan Entity Framework, gan NHibernate datubāzes pieejai lieto

ADO.NET datu sniedzējus (Data Provider). ADO.NET datu sniedzēji ir visām populārākajam datu bāzēm [6].
Projektā datu pieejas līmeņa veidošanai tika izvēlēts ADO.NET Entity Framework. Kā galvenos argumentus šim lēmumam var minēt:

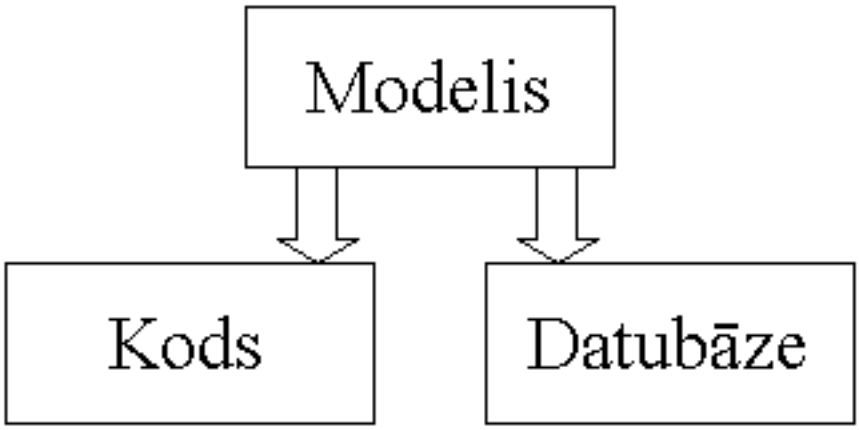
- Entity Framework izstrādā Microsoft, regulāri tiek izlaistas jaunas versijas, ir skaidrs plāns, kā produkts tālāk attīstīsies [7].
- NHibernate izstrādā atvērtā koda kopiena. Projektā aktivitāte samazinājusies un samazinās, kā arī nav plāna par produkta tālāko attīstību.
- Entity Framework atbalsta arī jaunākās .NET 4.5 versijas iespējas, bet NHibernate tikai .NET 3.5.
- Entity Framework ir daudz labāk dokumentēta, salīdzinot ar NHibernate.

4. Dažādas pieejas darbam ar Entity Framework

Strādājot ar Entity Framework var izmantot dažādas pieejas. Apskatīsim šīs pieejas sīkāk.

4.1. Model First

Model First pieejā vispirms ar speciāla grafiskā redaktora, kurš ir integrēts Microsoft Visual Studio, palīdzību tiek izveidots modelis. Pēc tam no modeļa tiek ģenerēta gan datubāze, gan objektu klašu struktūra.

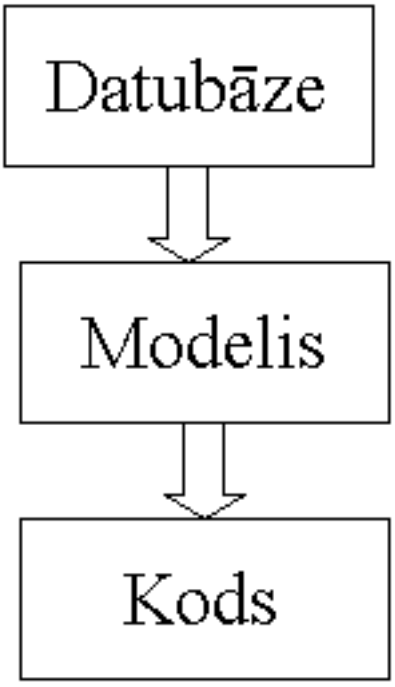


Zīmējums 1

Modelis faktiski ir ORM definīcija XML faila veidā. Tajā tiek aprakstīts, kādā veidā objekti tiek piesaistīti datubāzes tabulām. Modelis tiek glabāts failā ar paplašinājumu .edmx.

4.2. Database First

Database First pieeja nozīmē, ka no eksistējošas datubāzes tiek ģenerēts kods, kurš nodrošina darbību ar datubāzi. Vispirms no datubāzes tiek ģenerēts modelis, pēc tam no modeļa tiek ģenerēts kods (klašu struktūra), kurš nodrošina darbību ar objektiem..



Zīmējums 2

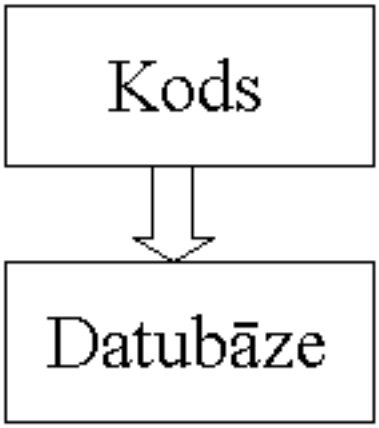
Gadījumā ja datubāzē tiek izdarītas izmaiņas, modelis un objektu klašu struktūra Ir jāpārģenerē.

4.3. Code First

Ja abās iepriekš aprakstītajās pieejās objektu hierarhija tika bāzēta uz klases, kas ir definēta Entity Framework bibliotēkā, tad **Code First** pieejā objektu klases ir neatkarīgas. Tajās tiek definēti tā saucamie **POCO** objekti (**Plain Old CLR Object**).
Code First pieeja neizmanto edmx modeli. Modeļa definīcija tiek iekodēta programmu izejas kodā. Code First pieeju var izmantot gan gadījumā, ja tiek veidota lietojumprogramma ar jaunu datubāzi, gan gadījumā, ja datubāze jau eksistē.

4.3.1. Code First ar datubāzes ģenerāciju

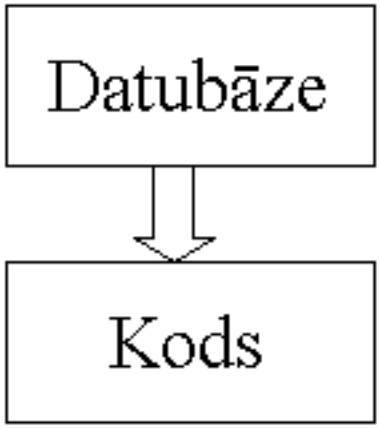
Sākotnēji šī pieeja tika saukta par **Code Only** pieeju. Tas ir tāpēc, ka vienīgais, ko ir nepieciešams izdarīt - ir jāuzraksta kods (POCO objektu klašu definīcijas). Datubāze šajā pieejā tiek ģenerēta izpildes laikā.
Šo pieeju var uztvert kā kaut ko līdzīgu Model First pieejai, tikai, atšķirībā no Model First pieejas, kurā modelis tiek definēts speciālā xml failā, Code First pieejā modeli definē ar koda palīdzību – ar datu anotācijām(atribūtiem) pie POCO objektiem un **Fluent API** interfeisa palīdzību [8].



Zīmējums 3

4.3.2. Code First ar eksistējošu datubāzi

Code First pieeju un POCO objektus, līdzīgi kā Database First pieejā, var izmantot arī gadījumā, ja datubāze jau eksistē. Šajā gadījumā ir jānodefinē datubāzes tabulām atbilstoši POCO objekti un, jānodefinē korekta atbilstība starp datubāzi un POCO objektiem, izmantojot datu anotācijas un Fluent API interfeisu.



Zīmējums 4

4.3.3. Migrācija

Sākot no Entity Framework 4.3 versijas, izmantojot Code First pieeju, gadījumā, ja ir mainījies datubāzes modelis, ir iespējams veikt automātisku vai manuālu datubāzes migrāciju, nezaudējot lietotāja datus [9]. Tas nozīmē, ja, piemēram, kādā tabulā nepieciešams definēt jaunu lauku, t. i. ir mainījies modelis, ir iespējams panākt automātisku datubāzes migrāciju, vai izveidot skriptus, kas veic šādu datubāzes migrāciju, nezaudējot datus. Šī Entity Framework iespēja atvieglo izstrādājamās lietojumprogrammas uzturēšanu.

5. Pieejas izvēle

Kā jau tika minēts, Model First un Database First pieejās objektiem ir virsklase, kura ir definēta Entity Framework bibliotēka. Turpretī POCO objektiem nav nepieciešama virsklase. Tas nozīmē, ka POCO objektiem:

- Minimizēta sarežģītība un atkarība no citiem līmeņiem.
- Ir vienkāršāka serializācija un vienkāršāka objektu nodošana starp dažādiem līmeņiem.
- Vienkāršāka testēšana.

Izmantojot POCO objektus, iespējams veidot gan jaunu datu bāzi, gan veidot datu pieejas līmeni, izmantojot jau eksistējošu datu bāzi.

Ņemot vēra mūsu uz modeļiem balstīto pieeju, lietojot POCO objektus, koda ģenerācija ir vienāda gan *Code First ar datubāzes ģenerācijas*, gan *Code First ar eksistējošu datubāzi* gadījumā. Vajadzīgs tikai atbilstošs modelis. Tādēļ projektā tiek izmantots ADO.NET Entity Framework ar Code First pieeju.

Koda ģenerators atbilstoši datubāžu tabulu metamodelim [6] ģenerē POCO objektu klases un atbilstošo sakarību un īpašību definīcijas (var teikt ORM modeli), izmantojot datu anotācijas un Fluent API interfeisu.

6. Rezultāti

Aktivitātes ietvaros tika izpētīts Entity Framework, izpētītas dažādas pieejas darbam ar Entity Framework, kā arī izstrādāts koda ģenerators atbilstoši datubāžu tabulu metamodelim [6] ģenerē POCO objektu klases un atbilstošo sakarību un īpašību definīcijas (var teikt ORM modeli), izmantojot datu anotācijas un Fluent API interfeisu.

7. Literatūras saraksts

[1] Object-relational mapping - http://en.wikipedia.org/wiki/Object-relational_mapping
[2] Entity Framework (EF) Documentation - http://msdn.microsoft.com/en-us/data/ee712907.aspx
[3] NHibernate Forge - http://nhforge.org/
[4] Entity Framework 6 vs NHibernate 4 - http://www.devbridge.com/articles/entity-framework-6-vs-nhibernate-4/
[5] The State of Entity Framework and NHibernate - http://weblogs.asp.net/ricardoperes/the-state-of-entity-framework-and-nhibernate
[6] Projekta aktivitātes Nr.3.3 "Datu bāzes meta modeļa izstrāde un pētniecība, kas ietver dažādu datu bāzu vadības sistēmu atbalsta izpēti" progresa pārskats
[7] Entity Framework Roadmap - https://entityframework.codeplex.com/wikipage?title=Roadmap
[8] Configuring/Mapping Properties and Types with the Fluent API - http://msdn.microsoft.com/en-us/data/jj591617.aspx
[9] Code First Migrations - http://msdn.microsoft.com/en-us/data/jj591621.aspx

8. Pielikumi

8.1. POCO objekta piemērs

```
public partial class Person
{
        [Required]
        [MaxLength(200)]
        public string @name { get; set; }
        public string @persCode { get; set; }
        public string @CIFcode { get; set; }
        public string @juridical { get; set; }
        public int? @uniqueID { get; set; }
        public bool? @isZombie { get; set; }
        public string @phoneMain { get; set; }
        public string @phoneMain2 { get; set; }

        [MaxLength(200)]
        public string @emergencyContacts { get; set; }
        public string @postAddress { get; set; }
        public string @email { get; set; }

        [MaxLength(100)]
        public string @legalAddress { get; set; }
        [Key()]
        public int @id { get; set; }
        // Navigation properties
        public virtual Subject myBase_Subject_O  {get;set;}
        public virtual ICollection<CourtDecision> against_CourtDecision_O
{get;set;}
        public virtual Heir myBase_Heir_O  {get;set;}
        public virtual SUauthorizedP myBase_SUauthorizedP_O  {get;set;}
        public virtual ICollection<Person_Act> idt_Person_Act_O  {get;set;}
}
```

8.2. Sakarību un īpašību definīcijas koda fragments, izmantojot Fluent API interfeisu

```
modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
          modelBuilder.Entity<Subject>().HasKey(t => new {t.myBase });
      modelBuilder.Entity<Subject>().Property(x => x.income).HasPrecision(28, 4);
      modelBuilder.Entity<Subject>().Property(x =>
x.dateOfDeath).HasColumnType("date");
      modelBuilder.Entity<Subject>().Property(x =>
x.insolvStartDate).HasColumnType("date");
       // Navigation properties
      modelBuilder.Entity<Subject>().HasRequired(e => e.myBase_O).WithOptional(c
=> c.myBase_Subject_O).WillCascadeOnDelete(false);
      modelBuilder.Entity<Subject>().HasOptional(e =>
e.overdueReason_O).WithMany(c => c.overdueReason_Subject_O).HasForeignKey(f => new
{f.overdueReason}).WillCascadeOnDelete(false);
```