LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

LINQ MODEĻU REPOZITORIJIEM

BAKALAURA DARBS

Autors: **Ivans Tabernakulovs**
Studenta apliecības Nr.: it12036
Darba vadītājs: docente Dr. dat. Elīna Kalniņa

RĪGA 2016

ANOTĀCIJA

Modeļu vadīta pieeja programmatūras izstrādes pasaulē pēdējā laikā kļūst arvien populārāka un attiecīgi arī aug pieprasījums pēc rīkiem, kas spēj efektīvi strādāt ar modeļiem. “LINQ modeļu repozitorijiem” darba ietvaros uzmanība tiek pievērsta iespējām strādāt ar modeļu repozitorijiem .NET satvarā C# programmēšanas valodā. Darbā tiek izpētītas LINQ adapteru izstrādes iespējas, modeļu repozitoriju piekļuves iespējas un izstrādāts LINQ adapteris modeļu repozitorijiem, kas balstās uz universālu RA API, kas ļauj izmantot dažādus modeļu repozitorijus, piemēram, populāro EMF.

Atslēgvārdi: .NET C#, LINQ, modeļu repozitoriji.

ABSTRACT

Model-driven approach in the software development world lately gains popularity and accordingly grows the need of tools that are able to work with models effectively. In the context of "LINQ to model repositories" thesis, the attention has been directed on the ways to work with model repositories in .NET framework C# programming language. LINQ adapter development options and model repository accessibility options has been researched and LINQ adapter to model repositories has been developed, based on universal RA API, that enables usage of different model repositories, for example, popular EMF.

Keywords: .NET C#, LINQ, model repositories.

SATURS

APZĪMĒJUMI

Tabula 1.1.

Termini un pieņemtie apzīmējumi

Definīcija, apzīmējums, saīsinājums	Paskaidrojums
.NET	Microsoft izstrādāta lietojumprogrammatūras platforma [1].
C#	Microsoft izstrādāta vienkārša, moderna, plaša pielietojuma, objektorientēta programmēšanas valoda [2].
LINQ [Language-Integrated Query]	.NET satvara komponente, kas papildina .NET programmēšanas valodas ar vienotu vaicājumu veidošanas un apstrādes iespējām [3].
SQL [Structured Query Language]	Vaicājumu valoda datubāžu piekļuvei un manipulācijai [4].
XML [eXtensible Markup Language]	Paplašināmā iezīmēšanas valoda, kas atvieglo strukturēta teksta un informācijas koplietošanu Internetā [5].
IntelliSense [Intelligent code completion]	Pirmkoda pabeigšanas iespēja dažās programmēšanas vidēs, kas paātrina pirmkoda rakstīšanas procesu, samazinot rakstīšanas un citu izplatīto kļūdu skaitu [6].
EMF	Modelēšanas satvars un pirmkoda ģenerācijas mehānisms [7].
Mii_rep	Latvijas Universitātes Matemātikas un informātikas institūtā izstrādāts modeļu repozitorijs.
JR	Latvijas Universitātes Matemātikas un informātikas institūtā izstrādāts modeļu repozitorijs.
EF [Entity Framework]	Atklātā pirmkoda satvars objektu relāciju kartēšanai .NET sistēmās [8].

EDM ģenerators [Entity Data Model generator]	Lietojumprogramma, kas "Entity Framework" satvarā pirms lietotāja darba uzsākšanās pieslēdzas pie datu avota un uzģenerē konceptuālu modeli un mērķa programmēšanas valodas klases [9].
Database-first	Viena no "Entity framework" satvara piedāvātām izstrādes iespējam, kas ļauj konstruēt modeli no eksistējošas datubāzes [10].
Model-first	Viena no "Entity framework" satvara piedāvātām izstrādes iespējam, kas ļauj izveidot datubāzi no uztaisīta modeļa [11].
Code-first	Viena no "Entity framework" satvara piedāvātām izstrādes iespējam, kas ļauj lietotājam pašam uzrakstīt klases pēc kurām tiks izveidota jauna datubāze vai kuras tiek izmantotas, lai strādātu ar eksistējošu datubāzi [12].
API [Application programming interface]	Specifikācija, kas nosaka, kā atšķirīgām programmatūras komponentēm vajadzētu savstarpēji sazināties [13].
RA API [Repository Access Application Programming Interface]	Sergeja Kozloviča izstrādāts modeļu repozitoriju piekļuves API [14].
Java	"Sun Microsystems" izstrādāta objektorientēta programmēšanas valoda [15].
Re-linq	Bibliotēka, kas var palīdzēt veidot LINQ piegādātājus (providers) [16].
WPF [Windows Presentation Foundation]	Grafiskā sistēma, kas paredzēta lietotāju saskarnes izrādīšanai Microsoft Windows-bāzētās lietojumprogrammās [17].
Visual Studio	Microsoft izstrādāta integrētā izstrādes vide, paredzēta lietojumprogrammu izstrādei dažādām platformām [18].

IEVADS

Pieaugot modeļu vadītas pieejas programmatūras izstrādei popularitātei pasaulē, pieaug arī pieprasījums pēc rīkiem, kas spēj efektīvi strādāt ar modeļiem, jeb veikt dažādas lasīšanas un transformācijas operācijas ar dažādiem modeļu repozitorijiem.

Ir nepieciešami gan tādi rīki, kas ir savrupas programmas ar vizuālu saskarni, gan tādi, kas ļauj strādāt ar modeļu repozitorijiem programmējot, tas ir, rakstot pirmkodu kādā programmēšanas valodā.

Šis darbs ir veltīts iespējām strādāt ar modeļu repozitorijiem .NET satvarā [1] C# programmēšanas valodā [2], jo tā ir viena no populārākām programmēšanas valodām, kas, kā autors uzskata, jau vistuvākajā laikā kļūs vēl populārāka.

Pirms šī darba izstrādes neeksistēja C# bibliotēka, ar kuru varētu ērti strādāt ar modeļu repozitorijiem, tāpēc tika pieņemts lēmums radīt C# bibliotēku, kas būtu pietiekami augstā abstrakcijas līmenī un varētu tikt galā ar šo uzdevumu. Par pietiekamu abstrakcijas līmeni tika izvēlēta LINQ adaptera izstrāde.

Darba struktūra

Darba pirmajā nodaļā tiek aplūkoti pamatjēdzieni – LINQ, modeļu repozitoriji un Entity Framework, izpratne par kuriem ir nepieciešama, lai labāk saprastu darba nākamās nodaļas.

Darba otrajā nodaļā tiek aplūkota esošā situācija pasaulē, saistībā ar modeļu transformāciju rīkiem C# programmēšanas valodā.

Darba trešajā nodaļā tiek aplūkota darba autora izstrādātā “LINQ to RA API” sistēma. Tāpat tiek aplūkotas dažādi tehnoloģiskie risinājumi, kas tika izmantoti sistēmas izstrādes laikā.

Darba ceturtajā nodaļā tiek demonstrētas izstrādātās sistēmas pielietošanas iespējas vienkāršiem modeļu transformāciju uzdevumiem.

1. PAMATJĒDZIENI

1.1. LINQ

LINQ ir līdzekļu kopums, kas paplašina un standartizē vaicājumu veidošanas iespējas C# programmēšanas valodā. LINQ piedāvā viegli izmantojamu, viegli apgūstamu un, pats galvenais, vienādu sintaksi, lai veidotu vaicājumus potenciāli jebkura tipa datu krātuvei [3]. Sākot no .NET satvara 3.5 versijas, tajā ir atbalsta komplekts, kas ļauj izmantot LINQ, lai veiktu 4 standarta tipa operācijas (veidošana, lasīšana, rediģēšana, dzēšana) ar .NET satvara kolekcijām, SQL datubāzēm [4], ADO.NET datu kopām un XML dokumentiem [5].

Bez LINQ veikt vaicājumus ārējai datu krātuvei varēja, tikai sastādot vienkāršā teksta vaicājumus mērķa glabātuves vaicājumu valodā un manuāli pārveidot atbildi iepriekš sagatavotā objektā. Izmantojot šādu metodi kompilators nevar iepriekš zināt datu tipu un nevar pārbaudīt datu tipu atbilstību, pirms dati faktiski atnāk, koda izpildes laikā, kas nozīmē, ka arī IntelliSense [6] nevar palīdzēt programmētajiem ar vaicājumu sastādīšanu.

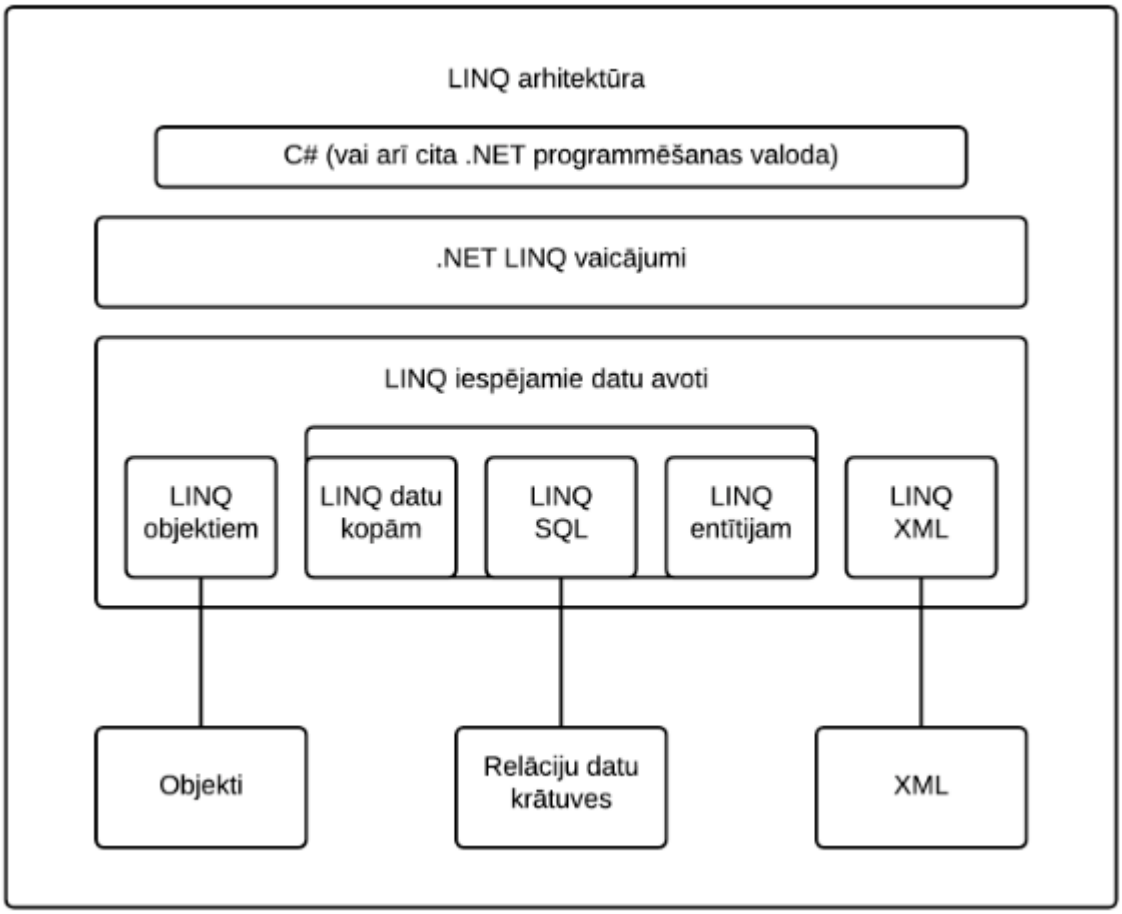
LINQ atrisina visas minētas un daudzas citas izplatītas vaicājumu veidošanas pieejas problēmas, kā arī standartizē vienotu sintaksi, neatkarīgi no mērķa datu krātuves. Tā, piemēram, aplūkosim LINQ vaicājumu, kas atrod kolekcijas vidu un tad sameklē pirmos trīs elementus, kas satur burtu “a”. Vaicājuma sintakse nemainīsies neatkarīgi no tā vai tas tiks jautāts “List<string>” tipa objektu vai XML dokumentu ar tādu pašu struktūru (pirmkods 1.1.).

```
var result = items.Skip(items.Count() / 2).Where(o => o.Contains("a")).Take(3).ToList();
```

Pirmkods 1.1. **LINQ vaicājumu sintakse**

Abos gadījumos, rakstot šo vaicājumu, var paļauties uz IntelliSense palīdzību un kompilators brīvi varēs pārbaudīt datu tipu pareizību.

LINQ sintakse ir kļuvusi ļoti populāra un tagad programmētājiem ir grūti iedomāties dzīvi bez tās, tāpēc trešās puses izstrādātāji ir izveidojuši vairākus LINQ adapterus daudzām citām datu krātuvēm.



Attēls 1.1. **LINQ arhitektūra [19]**

LINQ galvenās sastāvdaļas ir standartizētas vaicājumu metodes (Where, First, Select, Count u.t.t.) un adapteris, kas pārtulko vaicājumus uz mērķa krātuves vaicājumu valodu un saņemto atbildi pārkartē atpakaļ objektos (attēls 1.1.).

1.2. LINQ implementācija

Implementējot linq kādai datu glabātuvei, galvenais un grūtākais uzdevums ir iztulkot lietotāja vaicājumu tā, lai tas ir saprotams mērķa datu krātuvei [20]. Jādomā arī par datu pieprasīšanas veidu, jo operācijas, kas tiek izpildītas ar tiešā atmiņā pieejamiem datiem, nebūs optimāli izpildīt ar datiem, kas ir izkaisīti pa ārējas datu krātuves entītijām. Tāpēc ir svarīgi parūpēties par to, lai vaicājums būtu sastādīts tā, lai rezultātā no datu krātuves atnāk tikai tie dati, kuri lietotājam ir nepieciešami (kurus lietotājs tālāk izmanto) un, lai pēc iespējas vairāk filtru būtu pielietots datu krātuves pusē.

```
var thirtyYearsOld = db.Person.Where(o => o.Age == 30).ToList();
```

Pirmkods 1.2. **LINQ vaicājuma piemērs**

Piemēram, vaicājumu, kas no entītijas "Person" saņem tikai tos ierakstus, kur kolonnas "Age" vērtība ir vienāda ar 30 (pirmkods 1.2.), nebūtu vēlams izpildīt, vispirms saņemot visus "Person" entītijas ierakstus, un tikai tad pielietot filtru. Daudz labāk būtu, ja šo filtru varētu pielietot pašā datu krātuves līmenī.

1.3. Modeļu repozitoriji

Mūsdienās programmatūras izstrādes jomā kļūst populāra modeļ-vadīta pieeja, kur visas programmas klases glabājas izstrādes platformas neatkarīgā vietā, modelī. Modelis ar klasēm sastāv no divām galvenām daļām: no metamodeļa ar informāciju par visām klasēm, to atribūtiem un asociācijām starp tiem un no klašu objektiem. Lai modelī varētu veikt izmaiņas, jeb pāriet no viena modeļa uz citu, ir nepieciešams veikt modeļu transformācijas, par ko ir atbildīgi vairāki tām paredzēti rīki.

Ir vairāki modeļu repozitoriji, kas tādus modeļus var glabāt. Šī darba kontekstā izmantotā RA API spēj strādāt ar trim modeļu repozitorijiem: "EMF" [7], "Mii_rep" un "JR".

Pārsvarā "LINQ to RA API" bibliotēka tika testēta uz "EMF" modeļu repozitorijas. "EMF" projekts ir modelēšanas satvars un pirmkoda ģenerācijas mehānisms, lai izstrādātu programmatūru, balstoties uz strukturētu datu modeli. Izmantojot dotā modeļa specifikācijas aprakstu, "EMF" nodrošina instrumentus un atbalstu programmas izpildes laikā, lai ražotu Java klases [7].

1.4. Entity Framework

Entity Framework (EF), kura struktūra ir daļēji līdzīga šajā darbā izstrādātajai sistēmai, ir satvars objektu relāciju kartēšanai .NET sistēmās [8], to arhitektūras galvenā daļa ir iepriekš aprakstītais LINQ adapteris. Bez tam, EF nodrošina arī EDM ģeneratoru [9], kas ir lietojumprogramma, kas pirms lietotāja darba uzsākšanās pieslēdzas pie datu avota un uzģenerē konceptuālu modeli un mērķa programmēšanas valodā klases, kas atspoguļo datu avotam vismaz līdzīgu datu struktūru. EF arī piedāvā trīs pieejas, kā tiek veidotas minētais modelis un klases:

- "Database-first" [10] – Izmanto, kad lieto gatavu datubāzi. Gan Modelis, gan klases tiek uzģenerētas automātiski. Šī pieeja ir visvieglākā un rekomendējama iesācējiem.

- “Model-first” [11] – Izmanto, kad datubāzes vēl neeksistē. EF piedāvā datubāzes dizaineri, ar kuru var no nulles uztaisīt datubāzes shēmu un palaist uzģenerētu SQL skriptu jau pašā datubāzē. Klases šādā pieejā arī tiek uzģenerētas automātiski.
- “Code-first” [12] – Izmanto pieredzējuši lietotāji. Šajā pieejā vispār netaisa konceptuālo modeli un klases veido manuāli. Šī pieeja ir derīga gan eksistējošai, gan jaunajai datubāzei.

2. PAŠREIZĒJĀS SITUĀCIJAS IZKLĀSTS

Kopš 2000. gada modeļu-orientēta pieeja kļūst arvien populārāka, attiecīgi attīstās nepieciešamība efektīvi veikt modeļu transformācijas. Eksistē jau vairāki modeļu transformāciju rīki, tomēr pašlaik nav pietiekami ērta un pieejama veida, lai strādātu ar modeļu repozitorijiem C# programmēšanas valodā pietiekoši augstā abstrakcijas līmenī, kas ir piemērots mūsdienām.

2.1. Esošie risinājumi

Sergejs Kozlovičs sava "Transformāciju vadītā arhitektūra un tās grafiskie prezentācijas dziņi" darba [21] kontekstā izstrādāja vienotu repozitoriju piekļuves abstrakcijas slāni, RA API (Repository Access API [13]) [14], kas vienā bibliotēkā nodrošina iespēju strādāt ar dažādiem modeļu repozitorijiem. Sākotnējā bibliotēka ir Java programmēšanas valodā [15], tomēr ar vēl vienu slāni no uzģenerētām C# klasēm, var izmantot šo bibliotēku arī C# valodā.

Strādāt ar šo bibliotēku nav ļoti ātri un viegli, jo tās abstrakcijas līmenis ir diezgan zems. Lietotājam nākas strādāt pārsvarā ar 64 bitu skaitļu objektiem, kas reprezentē dažādus repozitorija elementu identifikatorus. Piemēram, zināms, ka repozitorijā ir klase "Planet" ar pieciem atribūtiem. Izmantojot RA API, lai vienkārši uzzinātu šo atribūtu nosaukumus, lietotājam ir jāvēršas pie RA API vismaz 14 reizes (pirmkods 2.1.).

```
var attributes = new List<string>();
var classID = tdaKernel.FindClass("Planet");
var iterator = tdaKernel.GetIteratorForAllAttributes(classID);
var currentAttribute = tdaKernel.ResolveIteratorFirst(iterator);
while (currentAttribute != 0)
{
    attributes.Add(tdaKernel.GetAttributeName(currentAttribute));
    currentAttribute = tdaKernel.ResolveIteratorNext(iterator);
}
tdaKernel.FreeIterator(iterator);
```

Pirmkods 2.1. **RA API vienkārša uzdevuma izpildīšanas piemērs**

Tomēr ar šo API potenciāli var izpildīt visas nepieciešamās darbības ar modeļu repozitorijiem, tāpēc, ja izdotos pietiekami paaugstināt to abstrakcijas līmeni, varētu nodrošināt ērtu un vieglu darbu ar modeļu repozitorijiem.

3. LINQ MODEĻU REPOZITORIJIEM

Lai atvieglotu darbu ar modeļu repozitorijiem C# valodā, tika pieņemts lēmums paplašināt RA API C# reprezentāciju, paaugstinot tā abstrakcijas līmeni līdz pilnvērtīgam LINQ adapterim.

3.1. Dažādas idejas un pieejas

Pirms darba uzsākšanas, kā arī darbā gaitā, darba autoram bija vairākas idejas un tika aplūkotas vairākas pieejas, kā realizēt ideju un nonākt līdz uzstādītajam mērķim. Šī nodaļa apraksta būtiskākās no tām.

3.1.1. Palīgbibliotēkas

Pielāgota LINQ piegādātāja (provider) implementācija tiek uzskatīta par sarežģītu programmēšanas uzdevumu, tāpēc eksistē bibliotēkas, kas var palīdzēt to realizēt.

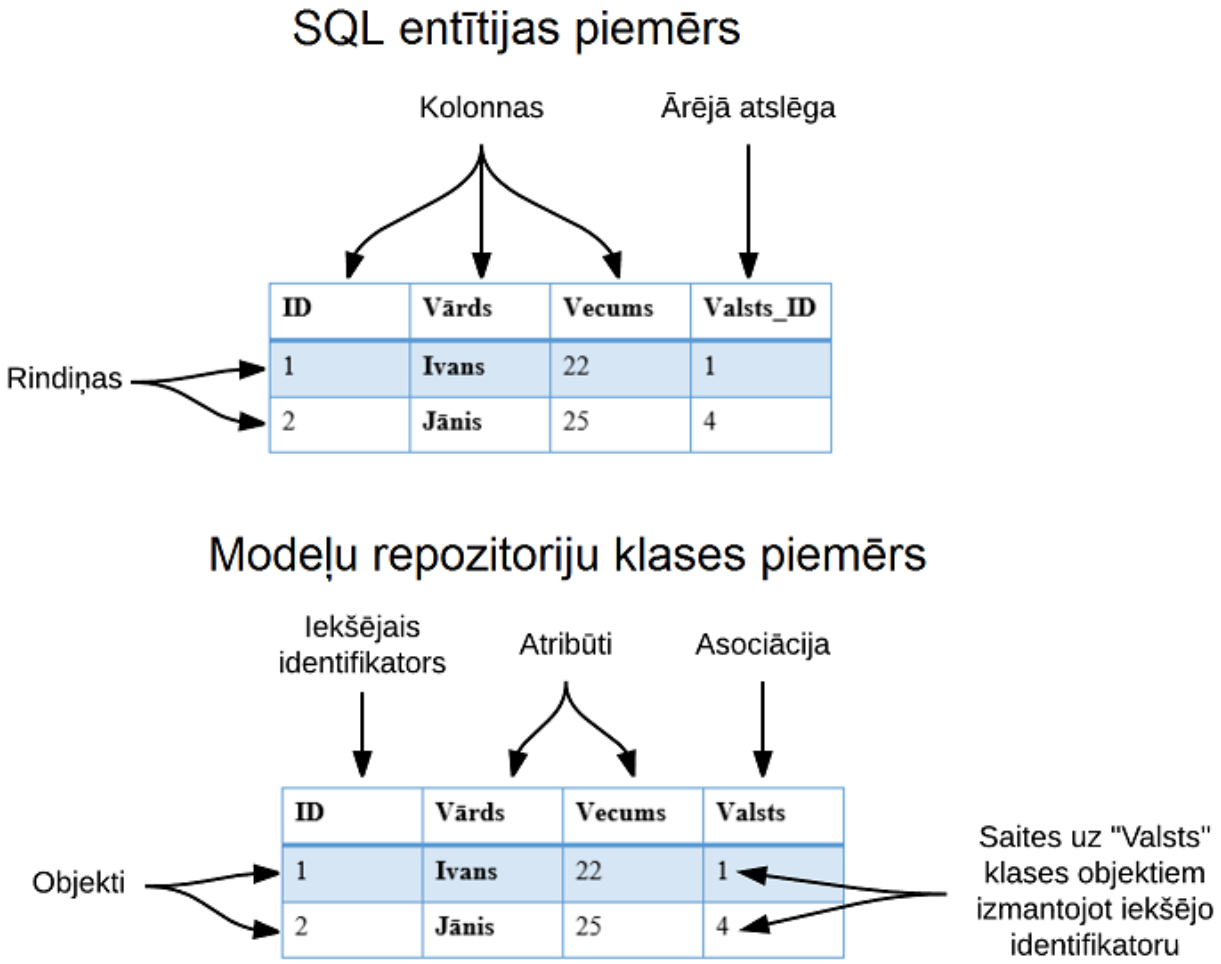
Vispazīstamākā tāda bibliotēka ir *re-linq* [16]. Tomēr šī bibliotēka ir pārāk maz dokumentēta un tās versijas diezgan būtiski atšķiras, kas padara vairākus atrodamos izmantošanas piemērus neaktuālus. Pie tam, tā ir vairāk piemērota gadījumiem, kad, lai saņemtu datus, jākonstruē pieprasījumus kādā citā vaicājumu valodā (piemērām SQL), tāpēc *re-linq* ir vairāk vērsta uz pieprasījumu tulkošanu mērķa datu krātuves piekļuves valodā.

Šī darba mērķa sasniegšanai nav nepieciešams veikt tulkošanu citā vaicājumu valodā, jo API, kura tiek izmantota lai piekļūtu modeļu repozitorijam jau ir C# valodā.

Tāpēc tika pieņemts lēmums neizmantot nekādas palīgbibliotēkas, taču izmantot dažas pieejas , ko izmanto palīgbibliotēkas, implementējot savu LINQ bibliotēku. Šīs lēmums arī pozitīvi ietekmē galīgo izejas datņu skaitu, kuram vajadzētu būt pēc iespējas mazākam, lai nepiesārņotu jeb neliktu pārāk daudz datņu lietotāja resursu direktorijā. Jebkādas trešās puses bibliotēkas (tādu, kas neeksistē pašā .NET satvarā) izmantošana nozīmētu, ka lietotajam būtu jātur gan tā bibliotēkas datne, gan jāpievieno atsauce uz to savā projektā.

3.1.2. EF līdzīgu pieeju izmantošana

Starp modeļu repozitoriju un SQL datubāžu struktūrām var, vismaz idejiski, novilkt vairākas paralēles. Modeļu repozitoriju klases var iedomāties kā SQL entītijas ar kolonnām kā modeļu repozitoriju atribūtiem, ārējām atslēgām kā modeļu repozitoriju asociācijām un rindiņām kā modeļu repozitoriju objektiem, piemēru var apskatīt attēlā 3.1.



Attēls 3.1. **SQL un modeļu repozitoriju struktūru salīdzinājums**

Lai ērti strādātu ar SQL datubāzēm .NET satvarā jau eksistē gan plaši pazīstams un izmantojams LINQ adapteris, "LINQ-to-entities" adapteris [22], gan bibliotēka, "Entity Framework", kas to izmanto.

Šī darba uzdevuma prasības lielā mērā ir līdzīgas tām, ko izpilda EF. Darbības, ko EF izpilda ar SQL datubāzēm, ļoti lielā mērā (varbūt pat pilnībā) sakrīt ar tām, kas ir jāimplementē "LINQ to RA API" projekta ietvaros. Kā arī, ļoti iespējams, ka lietotāji, kuri izmantos "LINQ to RA API" bibliotēku būs pieraduši strādāt ar EF un viņiem būs daudz ērtāk, ja darbs ar "LINQ to RA API" bibliotēku vismaz sintaktiski būtu līdzīgs darbam ar EF.

Tāpēc tika pieņemts lēmums, taisot "LINQ to RA API" bibliotēku balstīties uz EF un "LINQ-to-entities" tehnikām, principiem un sintaksi.

Saistībā ar 1.4. nodaļā minētajām EF koda un konceptuāla modeļa ģenerācijas pieejām, var teikt kā "LINQ to RA API" sistēmā tiek izmantota pieeja, kas ir kaut kas pa vidu starp "Code-first" un "Database-first" pieejām, jo vizuālā konceptuālā modeļa projekta ietvaros nav, tomēr klases tiek uzģenerētas automātiski, par ko vairāk nākamajā nodaļā.

3.1.3. Provizoriska klašu ģenerācija

C#, kā objektorientētā programmēšanas valodā, vienmēr labāk ir strādāt ar objektiem nekā ar dinamiskajiem tipiem. Tāpēc, strādājot ar modeļu repozitorijiem būtu daudz ērtāk, ja katrai repozitorijas klasei eksistētu attiecīga C# klase. Piemēram, ja repozitorijā ir klase "Planet" ar atribūtiem "Name" un "Radius", tad C# arī jābūt klasei "Planet" ar īpašībām "Name" un "Radius" (pirmkods 3.1.).

```
class Planet
{
    public string Name { get; set; }
    public string Radius { get; set; }
}
```

Pirmkods 3.1. **Uzģenerētas klases piemērs**

Lai tādas klases būtu pieejamas, pirms strādāt ar katru konkrēto repozitoriju, ir nepieciešams sagatavošanās darbs. Lai neliktu rakstīt šīs klases pašam lietotajam, tās var uzģenerēt. Klašu ģenerāciju var veikt neliela atsevišķa programma ar minimālistisku lietotāju saskarni, kur lietotājam nākas tikai norādīt repozitorijas atrašanās vietu datņu sistēmā un palaist klašu ģenerāciju, rezultātā saņemot DLL tipa datni, kas arī ir "LINQ to RA API" bibliotēka darbam ar konkrēto repozitoriju.

Līdzīgu pieeju izmanto arī EF, piedāvājot lietotājiem vai nu uzģenerēt konceptuālo modeli ("database-first" pieeja), vai nu uzrakstīt klases pašiem ("code-first" pieeja).

Alternatīva būtu izmantot dinamiskā tipa objektus ar atslēgvārdu "dynamic", bet tad kompilators nevarēs veikt savu darbu līdz koda izpildes momentam un IntelliSense arī nevarēs palīdzēt kodu rakstīšanā, kas pārkāpj divus 1.1. nodaļā minētos galvenos LINQ principus, un, manuprāt, šāds rezultāts nevar tikt uzskatīts par pilnvērtīgu LINQ adapteri.

3.1.4. LINQ sintakse

Tradicionāli eksistē divi veidi kā rakstīt LINQ vaicājumus: izmantojot vaicājumu sintaksi vai izmantojot metožu sintaksi [23]. Pirmkodā 3.2. var redzēt vienu un to pašu vaicājumu, uzrakstītu divos dažādos veidos.

Vaicājumu sintakses piemērs

```
var querySyntax = from number in numbers
                  where number < 5
                  orderby number
                  select number;
```

Metožu sintakses piemērs

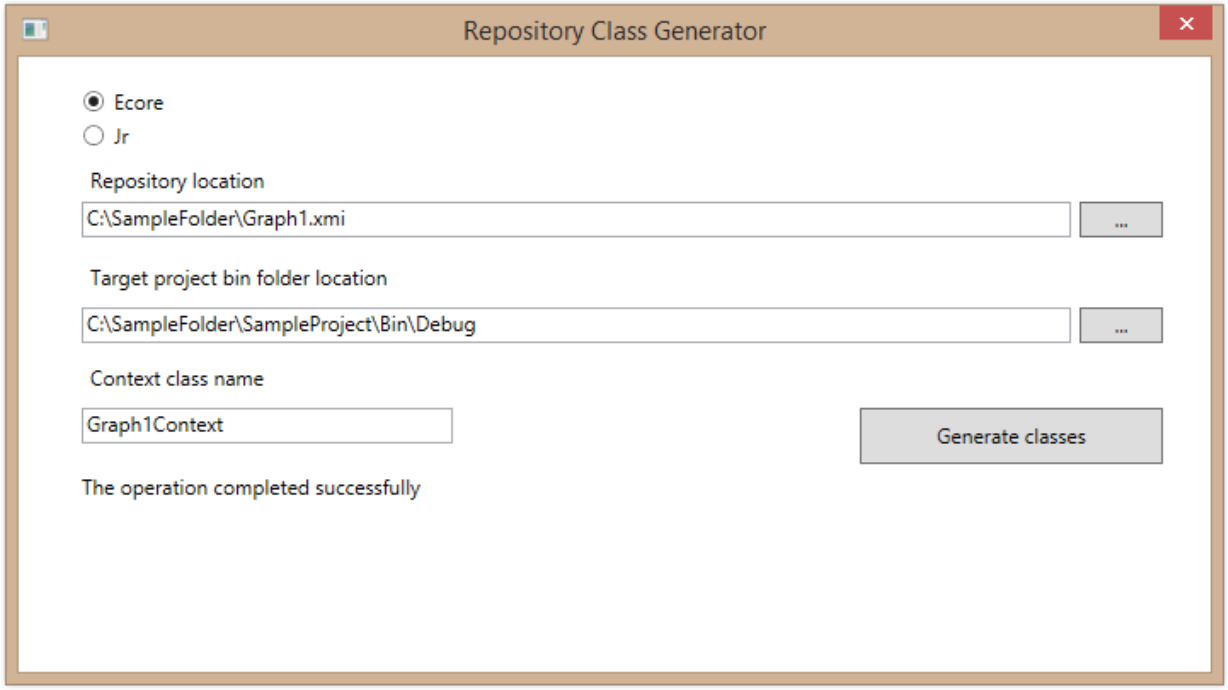
```
var methodSyntax = numbers.Where(o => o < 5).OrderBy(o => o);
```

Pirmkods 3.2. **Divu dažādu LINQ sintakšu piemēri**

Tomēr "LINQ to RA API" bibliotēkā tika realizēts tikai viens veids – izmantojot metožu sintaksi. Autors uzskata, ka izmantot vaicājumu sintaksi ir jēga varbūt tikai strādājot ar SQL datubāzi, jo tās vaicājumu valodas sintakse ir ļoti līdzīga un programmētāji, kuri ir vairāk pieraduši strādāt ar to, nekā ar objektorientētas valodas draudzīgu metožu sintaksi, sākumā varbūt izmantos vaicājumu sintaksi. Vaicājumi, kas tiek veikti modeļu repozitorijiem, autora prāt, nemaz neatgādina SQL vaicājumus, tāpēc autors izlēma nerealizēt šīs sintakses atbalstu "LINQ to RA API" bibliotēkā.

3.2. Sagatavošanās darbam instrukcija

Lai sāktu strādāt ar konkrētu modeļu repozitoriju savā C# projektā, vispirms jāizpilda daži sagatavošanas darbi. Tika izstrādāta neliela WPF programma [17], kas ar to palīdz. Lietotājam jāpalaiž "Repository Class Generator" programma, kuru saskarnes ekrānuzņēmumu var redzēt 3.2. attēlā.



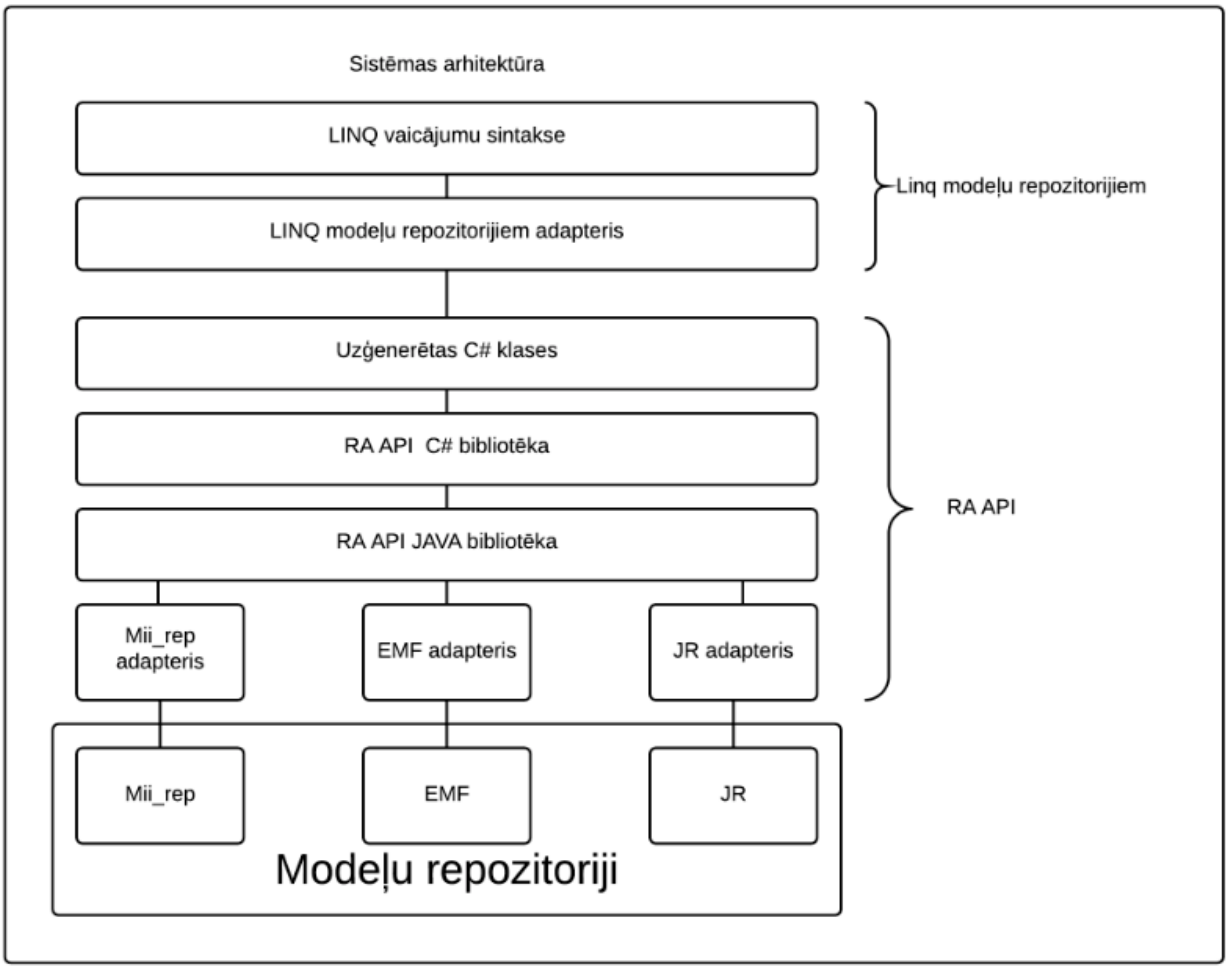
Attēls 3.2. **"Repository Class Generator" programmas lietotāja saskarne**

Programmā jāizvēlas repozitoriju tips, jānorada repozitorijas direktorija, jānorada sava projekta izpilddirektorija, jāizvēlas konteksta klases nosaukums un jānospiež uz "Generate Classes". Kad zemāk parādīsies "The operation completed successfully" paziņojums, noradītajā projekta direktorijā jāparādās 12 jaunām datnēm, viena no kurām ir "LINQ to RA API" bibliotēka, bet pārējas datnes ir nepieciešamas lai strādātu RA API.

Tālāk savā projektā jāpievieno saite uz "LINQ to RA API" bibliotēku, kas ir datne ar nosaukumu, kas sastāv no noradītas konteksta klases nosaukuma un "Linq.dll". Visual Studio [18] rīkā to var izdarīt uzklikšķinot uz projekta "References" nodaļu ar labo peles pogu, izvēloties "Add reference…" un pēc tam "Browse…" atrodot "LINQ to RA API" bibliotēkas datni un nospiežot "Add" un "OK".

3.3. Sistēmas arhitektūra

"LINQ to RA API" uzdevums ir nodrošināt lietotājam iespēju ar pazīstamu, EF līdzīgu sintaksi veikt vaicājumus uz RA API, kas savukārt tos pārveido un nodod modeļu repozitorijam, saņemot atpakaļ pieprasītos datus, vai pieprasītās darbības rezultātu (attēls 3.4.).



Attēls 3.4. **Sistēmas arhitektūra**

Lai to izdarītu, sistēmā sadarbojas vairākas komponentes. Katru konkrētu repozitoriju atspoguļo konteksta klases instance, visas darbības ar repozitoriju notiek caur šo instanci. Konteksta klasei, klašu ģenerācijas rezultātā, ir meta informācija par visām repozitorijas klasēm. Izmantojot šo meta informāciju, lietotājs var noformulēt, ar kādu repozitorijas klasi viņš vēlas strādāt, un, izvēloties vienu no implementētām LINQ vaicājumu veidošanas metodēm (sarakstu var redzēt 3.6. nodaļā) lietotājs noformulē, ko viņš vēlas izpildīt. Pēc tām sāk strādāt "LINQ to RA API" bibliotēkas LINQ adapteris, kas pārveido vaicājumu par RA API izsaukumiem un atgriež rezultātu.

3.4. Konteksta klase

"LINQ to RA API" bibliotēkas pamatā ir konteksta klase, kas ir līdzīga EF "DBContext" klasei [24].

Arī līdzīgi tam, kā tas notiek EF, klases nosaukumu lietotājs izvēlas pats, bibliotēkas ģenerācijas procesā.

Konteksta klase ir svarīga daļa no "LINQ to RA API" bibliotēkas, tas saista uzģenerētas C# klases un repozitoriju. Šī klase ir atbildīga par mijiedarbību ar repozitorijas datiem kā ar C# objektiem.

Konteksta klase ir atbildīga par sekojošām darbībām:

- **Repozitoriju klašu kopas:** konteksta klase satur repozitoriju klašu kopas (TDAQueryable<TEntity>) katrai uzģenerētai klasei.
- **Izmaiņu pārvaldība:** konteksta klase glabā norādi uz visiem objektiem, kas tika iegūti repozitorijas konteksta klases dzīves cikla ietvaros, tāda veidā realizējot iespēju jebkurā laikā objektus salīdzināt ar attiecīgajiem repozitorijā esošiem objektiem un pamanīt izmaiņas.
- **Izmaiņu realizēšana:** konteksta klase veic izmaiņu saglabāšanu repozitorijā, jeb atjaunošanas, pievienošanas un dzēšanas operāciju realizēšanu.
- **Sakaru pārvaldība:** konteksta klase pārvalda sakarus starp LINQ metodēm un RAAPI, kas reāli sadarbojas ar repozitoriju.

3.5. TDAQueryable klase

Pieprasījumus veido klase TDAQueryable, līdzīga IQueryable un DBSet EF klasēm, Klasei ir visas nepieciešamās īpašības, lai turētu atmiņā visus iepriekšējos vaicājumu ķēdes elementus: filtrus un nepieciešamās repozitoriju klases meta informāciju. Kad vaicājumu konstruēšana tiek pabeigta un pienāk laiks to izpildīt, klase nodrošina arī to, ņemot vērā arī vaicājumā specificētos nosacījumus un to kā vaicājuma rezultātam jābūt noformētam.

TDAQueryable klase ir atbildīga par sekojošām darbībām:

- **Vaicājumu konstruēšana un veikšana:** TDAQueryable klase vada vaicājumu veidošanas procesu, konvertē pieprasījumu uz vairākiem RA API izsaukumiem.
- **Objektu materializācija:** TDAQueryable klase pārveido neapstrādātus repozitoriju datus par C# objektiem.

3.6. TDAQueryable paplašinājuma metodes

Šī nodaļa satur visas implementētās TDAQueryable klases metodes, kuras tieši var izmantot lietotājs pēc konteksta klases instances izveidošanas. Tā kā šīs visas ir TDAQueryable paplašinājuma metodes pirmais parametrs vienmēr būs ar tipu TDAQueryable.

Metodes var sadalīt divās daļās:

- Metodes, kurās notiek datu materializācija, jeb tādas metodes, kurās reāli notiek darbības ar datiem no repozitorijām (Tabula 3.1);
- Metodes, kas tikai sastāda vaicājumu, pievienojot filtrus, kas būs jāpielieto datiem, kad tiks izsaukta viena no augstāk minētajām pirmās daļas metodēm (Tabula 3.2). Tādas metodes vienmēr atgriezīs TDAQueryable<T> tipa objektu.

Tabula 3.1.

TDAQueryable<T> pirmās daļas paplašinājuma metodes.

Metode	Parametri	Atgriežamais tips	Apraksts
Any<T>	Func<T, bool> predicate	bool	Nosaka, vai kaut viens kolekcijas elements apmierina nosacījumu.
Count<T>	-	int	Atgriež kolekcijas elementu skaitu.
Count<T>	Func<T, bool> predicate	int	Atgriež kolekcijas elementu, kas apmierina nosacījumu skaitu.
FirstOrDefault<T>	-	T or default	Atgriež pirmo kolekcijas elementu, vai "null", ja kolekcija ir tukša.
FirstOrDefault <T>	Func<T, bool> predicate	T or default	Atgriež pirmo kolekcijas elementu, kas apmierina nosacījumu vai "null", ja tādu elementu kolekcijā nav.
First<T>	-	T	Atgriež pirmo kolekcijas elementu.
First<T>	Func<T, bool> predicate	T	Atgriež pirmo kolekcijas elementu, kas apmierina nosacījumu.
SingleOrDefault<T>	-	T or default	Atgriež vienīgo kolekcijas elementu (gadījumā, ja elementu ir vairāk par

			vienu, atgriež kļūdu, vai "null", ja kolekcija ir tukša.
SingleOrDefault<T>	Func<T, bool> predicate	T or default	Atgriež vienīgo kolekcijas elementu, kas apmierina nosacījumu (gadījumā, ja elementu ir vairāk par vienu, izraisa kļūdu), vai "null", ja kolekcija ir tukša.
Single<T>	-	T	Atgriež vienīgo kolekcijas elementu (gadījumā, ja elementu ir vairāk par vienu, izraisa kļūdu).
Single<T>	Func<T, bool> predicate	T	Atgriež vienīgo kolekcijas elementu, kas apmierina nosacījumu (gadījumā, ja elementu ir vairāk par vienu, izraisa kļūdu).
GroupBy<T, TResult>	Func<T, bool> selector	IEnumerbale<IGrouping <TResult, T>>	Atgriež "IGrouping" tipa kolekciju, sagrupētu pēc dotā šķirotāja
Select<T, TResult>	Func<T, bool> selector	IEnumerbale<TResult>	Atgriež kolekciju atlasītu pēc dotā šķirotāja.
ToList<T>	-	List<T>	Atgriež visus elementus List<T> formātā.
Add<T>	T newObject	bool	Pievieno jaunu objektu repozitorijas klasei un nosaka, vai to izdevās izpildīt.
Remove	T objectToRemove	bool	Izdzēš objektu no repozitorijas un nosaka, vai to izdevās izpildīt.

Tabula 3.2.

TDAQueryable<T> otrās daļas paplašinājuma metodes.

Metode	**Parametri**	**Atgriežamais tips**	**Apraksts**
Skip<T>	int skipAmount	TDAQueryable<T>	Atgriež ieejas TDAQueryable<T> parametru ar papildus filtru – izlaist

			noteiktu elementu skaitu un atstāt pārējos.
Take<T>	int takeAmount	TDAQueryable<T>	Atgriež ieejas TDAQueryable<T> parametru ar papildus filtru – atstāt noteiktu elementu skaitu un izlaist pārējos.
Where<T>	Func<T, bool> predicate	TDAQueryable<T>	Atgriež ieejas TDAQueryable<T> parametru ar papildus filtru – atstāt tikai tos elementus, kas apmierina nosacījumu un izlaist pārējos.

3.7. Izmaiņu pārvaldība

Lai repozitorijā eksistējošo objektu izmainīšanai nevajadzētu izmantot nekādas speciālas papildus metodes un varētu veikt izmaiņas vienkārši piešķirot jaunas vērtības tieši no repozitorijas iegūtiem objektiem, ir nepieciešams izmaiņu pārvaldības mehānisms.

"LINQ to RA API" projektā tas tika realizēts sekojošā veidā. Konteksta klasē ir privāta īpašība "trackedObjects" ar tipu List<object>. Katru reizi, kad lietotāja pieprasīti repozitorijas objekti tiek saņemti un atdoti lietotājam, šajā mainīgajā tiek saglabātas norādes uz katru objektu. Kad lietotājs, pēc visu nepieciešamo izmaiņu veikšanas, izsauc konteksta klases "SaveChanges" metodi, katrs objekts, uz kuru "trackedObjects" mainīgajā ir norāde, tiek salīdzināts ar attiecīgu oriģinālu repozitorijas objektu, un, atšķirības gadījumā, visu to izmainīto atribūtu vērtības tiek pārrakstītas ar jaunām vērtībām.

Tādā veidā nav nepieciešamības izsekot objektu vērtību piešķiršanas momentiem, kas nozīmētu, ka katrai automātiski uzģenerētai klašu īpašībai būtu jādefinē pielāgotu "set" īpašību, kas palielinātu uzģenerēto koda apjomu vairākas reizes.

3.8. Objektu pievienošana un dzēšana

"LINQ to RA API" arī ļauj pievienot jaunus objektus repozitorijā, kā arī dzēst eksistējošus. Līdzīgi EF, tam ir paredzētas divas metodes, attiecīgi "Add" un "Remove", kas ir "TDAQueryable"

klases paplašinājuma metodes, parametros saņem vienu objektu, kuru attiecīgi jāpievieno vai jānodzēš.

3.9. Zināmie trūkumi

Tā kā ar RA API nav iespējams uzzināt repozitoriju asociāciju galu kardinalitāti, attiecīgi to nevar arī "LINQ to RA API" bibliotēka, tāpēc visas asociācijas tiek uzskatītas kā "Many-To-Many". Tas nekādā mērā neierobežo to funkcionalitāti un iespējamo darbību skaits nemainās, tomēr lietotājam katrā "Many-To-One" asociācijas gadījumā būs jāizsauc vienu no metodēm, kas atgriež vienu kolekcijas elementu, piemērām "FirstOrDefault", kā tas arī tiek izdarīts 6. nodaļā 4. datu lasīšanas uzdevuma risinājumā.

Pašlaik "LINQ to RA API" bibliotēkā nepastāv iespēja repozitorijā eksistējošam objektam labot asociāciju saites. Ja to ir nepieciešams izdarīt, var izdzēst esošu objektu un pievienot to no jauna, gan ar vecajām asociāciju saitēm, gan ar jaunajām (kā piemērām 4. nodaļas 1. datu transformācijas uzdevuma risinājumā). Šo trūkumu noteikti var izlabot, izmaiņu pārvaldības mehānismā, kas pašlaik izseko izmaiņas tikai objektu atribūtos, implementējot arī asociāciju saišu izmainu pārvaldību.

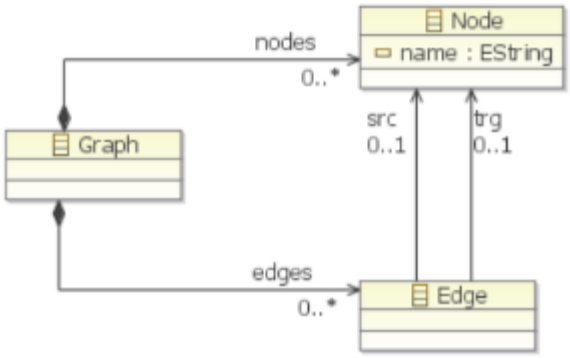
Izmaiņu pārvaldības mehānisms strādā nekorekti, kad vienu un to pašu repozitorijas objektu saņem (un attiecīgi pievieno izsekošanas objektam) vairākas reizes. Tādā gadījumā izmaiņu saglabāšanas momentā objekts repozitorijā tiks pielīdzināts pēdējam saņemtajam no repozitorijas un izmaiņas ar pārējiem netiek ņemtas vērā. Lai izvairītos no tādas problēmas, ieteicams izsaukt izmaiņu saglabāšanas metodi uzreiz pēc izmaiņu veikšanas un pirms tam, kad šis pats repozitorijas objekts tiks saņemts atkārtoti.

Projekta sagatavošanu darbam ar kādu repozitoriju izmantojot "LINQ to RA API" bibliotēku veic neatkarīga atsevišķa programma, kas nav nekādā mērā saistīta ar Visual Studio rīku. Būtu daudz labāk, ja sagatavošanas process varētu tikt iebūvēts Visual Studio rīkā pēc NuGet pakas instalācijas, kā tas notiek piemēram "Entity Framework" sistēmā.

4. PIELIETOŠANAS PIEMĒRI

Lai pārbaudītu bibliotēku darbībā tika atrisināti daži uzdevumi no "Hello World!" pasākuma, kur dalībniekiem ar savām transformāciju darbarīkiem bija jāizpilda vienkārši uzdevum, kas veica veidošanas, lasīšanas, rediģēšanas un dzēšanas operācijas ar modeļu repozitorijiem.

Uzdevumus bija jāveic uz konkrētas repozitorijas, kuras meta modeli var apskatīt attēlā 4.1.



Attēls 4.1. **Repozitorijas meta modelis**

4.1. Datu lasīšanas uzdevumi

Uzdevums 1: Veikt vaicājumu, kas saskaita grafam mezglus.

Risinājumu var apskatīt pirmkodā 4.1.

```
using (var context = new TDAContext(repositoryLocation))
{
    int nodeCount = context.Graphs.FirstOrDefault().nodes.Count();
}
```

Pirmkods 4.1. **1. datu lasīšanas uzdevuma risinājums**

Atbilde: `nodeCount` mainīga vērtība.

Uzdevums 2: Veikt vaicājumu, kas saskaita izolētus mezglus, jeb mezglus kas nav ne izeja, ne ieeja nevienai šķautnei.

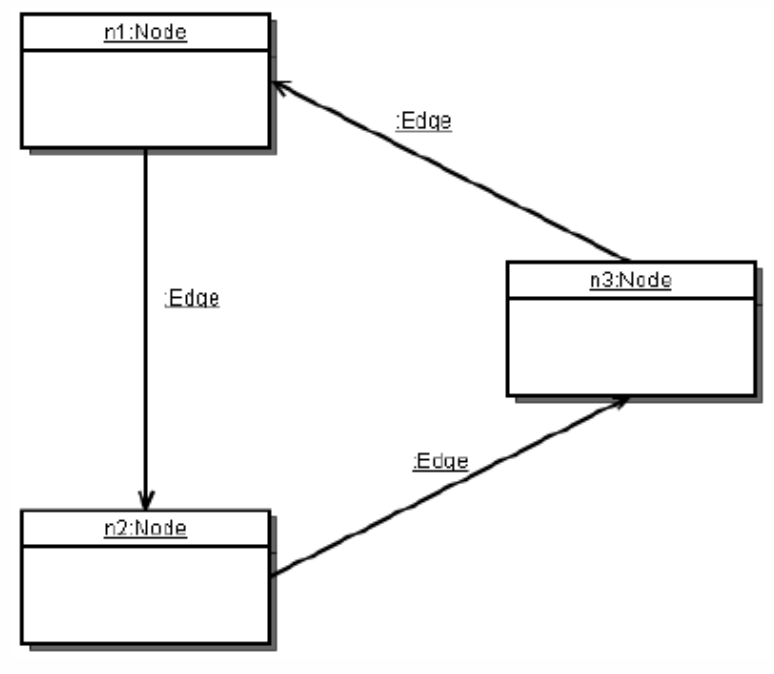
Risinājumu var apskatīt pirmkodā 4.2.

```
using (var context = new TDAContext(repositoryLocation))
{
    int isolatedNodeCount = 0;
    foreach (var node in context.Graphs.FirstOrDefault().nodes)
    {
        var isolated = true;

        foreach (var edge in context.Edges)
        {
            if (edge.src.Any(o => o == node) || edge.trg.Any(o => o == node))
            {
                isolated = false;
                break;
            }
        }

        if (isolated) isolatedNodeCount++;
    }
}
```

Pirmkods 4.2. **2. datu lasīšanas uzdevuma risinājums**

Atbilde: `isolatedNodeCount` mainīga vērtība.

Uzdevums 3: Veikt vaicājumu, kas saskaita riņķus, kas sastāv no trim atšķirīgiem mezgliem, kuru šķautnes norāda no pirmā uz otru, no otrā uz trešo un no trešā uz pirmo (Attēls 4.2.).



Attēls 4.2. **Riņķa no trim mezgliem piemērs**

Risinājumu var apskatīt pirmkodā 4.3.

```
using (var context = new TDAContext(repositoryLocation))
{
    int circleCount = 0;
    foreach (var node1 in context.Nodes)
    {
        var edges1 = context.Edges.Where(o => o.src.FirstOrDefault() == node1).ToList();

        foreach (var edge in edges1)
        {
            var node2 = edge.trg.FirstOrDefault();
            if (node2 == node1) continue;

            var edges2 = context.Edges.Where(o => o.src.FirstOrDefault() == node2).ToList();

            foreach (var edge2 in edges2)
            {
                var node3 = edge2.trg.FirstOrDefault();
                if (node3 == node2 || node3 == node1) continue;

                var edges3 = context.Edges.Where(o => o.src.FirstOrDefault() == node3);

                if (edges3.Any(o => o.trg.FirstOrDefault() == node1))
                    circleCount++;
            }
        }
    }
    circleCount /= 3;
}
```

Pirmkods 4.3. **3. datu lasīšanas uzdevuma risinājums**

Atbilde: `circleCount` mainīga vērtība.

Uzdevums 4: Veikt vaicājumu, kas saskaita šķautnes, kurai nav vai nu ieejas vai nu izejas mezgla.

Risinājumu var apskatīt pirmkodā 4.4.

```
using (var context = new TDAContext(repositoryLocation))
{
    int danglingEdgeCount = context.Edges.Count(o => o.src.FirstOrDefault() == null || o.trg.FirstOrDefault() == null);
}
```

Pirmkods 4.4. **4. datu lasīšanas uzdevuma risinājums**

Atbilde: `danglingEdgeCount` mainīga vērtība.

4.2. Datu transformācijas uzdevumi

Uzdevums 1: Veikt transformāciju, kas maina virzienu visām šķautnēm uz pretējo.

Risinājumu var apskatīt pirmkodā 4.5.

```
using (var context = new TDAContext(repositoryLocation))
{
    var edges = context.Edges.ToList();
    foreach (var edge in edges)
    {
        var newSource = edge.trg;
        var newTarget = edge.src;

        context.Edges.Remove(edge);
        context.Edges.Add(new Edge() { src = newSource, trg = newTarget });
    }

    context.SaveChanges();
}
```

Attēls 4.5. **1. datu transformācijas uzdevuma risinājums**

Uzdevums 2: Izdzēst mezglu ar atribūta "name" vērtību "n1" un visas šķautnes, kurām izeja vai ieeja ir šīs mezgls.

Risinājumu var apskatīt pirmkodā 4.6.

```
using (var context = new TDAContext(repositoryLocation))
{
    var n1 = context.Nodes.SingleOrDefault(o => o.name == "n1");
    var n1Edges = context.Edges.Where(o => o.src.FirstOrDefault() == n1 || o.trg.FirstOrDefault() == n1).ToList();

    context.Nodes.Remove(n1);
    foreach (var n1Edge in n1Edges)
    {
        context.Edges.Remove(n1Edge);
    }

    context.SaveChanges();
}
```

Pirmkods 4.6. **2. datu transformācijas uzdevuma risinājums**

5. SECINĀJUMI

"LINQ to RA API" bibliotēka tika veiksmīgi izstrādāta, tā izpilda visas pamatprasības, kas varētu būt modeļu transformāciju rīkam. Bibliotēka izturēja testēšanu, veiksmīgi atrisinot uzdevumus no "Hello world!" pasākuma. Neviens no 3.9. nodaļā minētajiem trūkumiem nav kritisks un ar laiku var tikt novērsts.

LINQ adaptera izstrāde ir diezgan apjomīgs darbs, tomēr iespējams sākumā implementēt pamata funkcionalitāti un tā jau nesīs lielus ieguvumus. Pēc tam ir iespējams veltīt laiku specifiskākai funkcionalitātei, kas uzlabo vai nu lietotāja izmantošanas pieredzi, vai nu sistēmas veiktspēju.

Saistībā ar šīs sistēmas attīstīšanu, LINQ sintakse iekļauj vairākas metodes, kas varētu tikt izmantotas, strādājot ar modeļu repozitorijiem un pašas svarīgākās no tām tika implementētas, nodrošinot iespēju veikt ar modeļu repozitorijiem visas nepieciešamas datu lasīšanas un transformācijas operācijas. Tomēr noteikti ir vēl dažas neimplementētas metodes, kuras var implementēt un kuras vēl vairāk atvieglotu darbu ar repozitorijām. Piemērām, "Last", "LastOrDefault", "ElementAt", "SkipWhile", "SelectWhile", "OrderBy", "OrderByDescending" un citas.

6. IZMANTOTĀ LITERATŪRA UN AVOTI

1. .NET [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://www.microsoft.com/net
2. C# [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/kx37x362.aspx
3. LINQ: .NET Language-Integrated Query [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/bb308959.aspx
4. Introduction to SQL [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://www.w3schools.com/sql/sql_intro.asp
5. Extensible Markup Language (XML) [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://www.w3.org/XML/
6. Using IntelliSense [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/hcw1s69b.aspx
7. What every Eclipse developer should know about EMF [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://eclipsesource.com/blogs/tutorials/emf-tutorial/
8. Entity Framework [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://www.asp.net/entity-framework
9. EDM Generator (EdmGen.exe) [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/bb387165(v=vs.100).aspx
10. Database First [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/data/jj206878.aspx
11. Model First [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/data/ff830362.aspx
12. Code First to a New Database [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/data/jj193542.aspx
13. API - application program interface [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://www.webopedia.com/TERM/A/API.html
14. Transformation-Driven Architecture, RA API dokumentācija [tiešsaiste – pārbaudīts 26.05.2016]. Pieejams:

http://tda.lumii.lv/documentation.shtml

15. What is Java technology and why do I need it? [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://java.com/en/download/faq/whatis_java.xml
16. Re-linq [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://relinq.codeplex.com/
17. Introduction to WPF [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/mt149842(v=vs.110).aspx
18. Introducing Visual Studio [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/fx6bk1f4(v=vs.90).aspx
19. Understanding LINQ (C#) [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://www.codeproject.com/Articles/19154/Understanding-LINQ-C
20. The Pain of Implementing LINQ Providers [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://cacm.acm.org/magazines/2011/8/114934-the-pain-of-implementing-linq-providers/fulltext
21. Sergejs Kozlovičs. Transformāciju vadītā arhitektūra un tās grafiskie prezentācijas dziņi. Rīga, 2012 [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://tda.lumii.lv/doc/thesis/Kozlovics_summary_lv.pdf
22. LINQ to Entities [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/bb386964(v=vs.100).aspx
23. Query Syntax and Method Syntax in LINQ (C#) [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: https://msdn.microsoft.com/en-us/library/bb397947.aspx
24. DBContext [tiešsaiste – pārbaudīts 27.05.2016]. Pieejams: http://www.entityframeworktutorial.net/EntityFramework4.3/dbcontext-vs-objectcontext.aspx

7. PIELIKUMI

7.1. Koda fragmenti

"TDAQueryable" klase.

```
public class TDAQueryable<T>
    where T : new()
{
    private T[] items = new T[0];

    internal List<Condition<T>> Conditions { get; set; }

    internal TDAContext context { get; set; }

    public EnumSimulator GetEnumerator()
    {
        return new EnumSimulator(Extensions.ToList(this).ToArray());
    }

    public class EnumSimulator
    {
        public T[] Items;
        int position = -1;

        public EnumSimulator(T[] list)
        {
            Items = list;
        }

        public bool MoveNext()
        {
            position++;
            return (position < Items.Length);
        }

        public void Reset()
        {
            position = -1;
        }

        public T Current
        {
            get
            {
                try
                {
                    return Items[position];
                }
                catch (IndexOutOfRangeException)
                {
                    throw new InvalidOperationException();
                }
            }
        }
```

```
        }
    }


    public class Condition<T>
        where T : new()
    {
        public Condition() { }

        public Condition(Enums.ConditionTypes type, Func<T, bool> predicate)
        {
            this.Type = type;
            this.Predicate = predicate;
        }

        public Func<T, bool> Predicate { get; set; }
        public Enums.ConditionTypes Type { get; set; }
        public int Amount { get; set; }
    }

    public class Enums
    {
        public enum ConditionTypes
        {
            Where,
            Skip,
            Take
        };
    }
```

"TDAQueryable" klases paplašinājuma metode "ToList".

```
public static List<T> ToList<T>(this TDAQueryable<T> source)
    where T : new()

{
    var context = source.context;
    var result = new List<T>();
    var conditions = source.Conditions;
    var className = TranslateClassName(typeof(T).Name);
    var properties = typeof(T).GetProperties(BindingFlags.Instance |
BindingFlags.NonPublic | BindingFlags.Public).Where(o => o.PropertyType.Name !=
typeof(TDAQueryable<>).Name).ToArray();
    var tdaQueryableProperties = typeof(T).GetProperties().Where(o => o.PropertyType.Name
== typeof(TDAQueryable<>).Name).ToArray();
    var classAttributeInstances = new List<List<string>>();

    foreach (var property in properties)
    {
        classAttributeInstances.Add(context.ClassAttributeObjects(className,
property.Name));
    }

    var data = GetData<T>(classAttributeInstances, properties);

    foreach (var item in data)
```

```
    {
        InitializeAssociationProperties<T>(context, tdaQueryableProperties, item,
className);
    }

    ApplyFilters<T>(data, conditions);

    foreach (var item in data)
    {
        context.TrackObject(item);
        result.Add(item);
    }

    return result;
}



public static List<long> GetObjectReferenceIDs<T>(TDAQueryable<T> source)
    where T : new()
{
    var context = source.context;
    var result = new List<long>();
    var conditions = source.Conditions;
    var className = TranslateClassName(typeof(T).Name);
    var properties = typeof(T).GetProperties(BindingFlags.Instance |
BindingFlags.NonPublic | BindingFlags.Public).Where(o => o.PropertyType.Name !=
typeof(TDAQueryable<>).Name).ToArray();
    var classAttributeInstances = new List<List<string>>();

    foreach (var property in properties)
    {
        classAttributeInstances.Add(context.ClassAttributeObjects(className,
property.Name));
    }

    var data = GetData<T>(classAttributeInstances, properties);

    ApplyFilters<T>(data, conditions);

    foreach (var item in data)
    {
        var objectReferenceIDs =
Int64.Parse((string)item.GetType().GetProperty("ObjectReferenceID", BindingFlags.Instance
| BindingFlags.NonPublic | BindingFlags.Public).GetValue(item, null));
        result.Add(objectReferenceIDs);
    }

    return result;
}

private static void InitializeAssociationProperties<T>(TDAContext context, PropertyInfo[]
tdaQueryableProperties, T inputItem, string className)
    where T : new()
{
    var ObjectID = Convert.ToInt64(typeof(T).GetProperty("ObjectReferenceID",
BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public).GetValue(inputItem,
null));
```

```
    foreach (var prop in tdaQueryableProperties)
    {
        var typeOfProp = prop.PropertyType.GetGenericArguments().FirstOrDefault();
        var tdaQueryableType = typeof(TDAQueryable<>);
        Type[] tdaQueryableTypeArgs = { typeOfProp };
        var typeForProp = tdaQueryableType.MakeGenericType(tdaQueryableTypeArgs);
        object mainProperty = Activator.CreateInstance(typeForProp);

        var internalProperties = typeForProp.GetProperties(BindingFlags.Instance |
BindingFlags.NonPublic | BindingFlags.Public);

        var linkedObjectIDs = context.GetAssociatedObjects(className, prop.Name,
ObjectID);

        var conditionType = typeof(Condition<>);
        Type[] conditionTypeArgs = { typeOfProp };
        var typeForCondition = conditionType.MakeGenericType(conditionTypeArgs);
        object condition = Activator.CreateInstance(typeForCondition, new object[]
{ Enums.ConditionTypes.Where, new Func<dynamic, bool>(o => linkedObjectIDs.Any(m => m ==
o.ObjectReferenceID)) });

        var listType = typeof(List<>);
        Type[] conditionListTypeArgs = { typeForCondition };
        var typeForConditionList = listType.MakeGenericType(conditionListTypeArgs);
        object listOfConditions = Activator.CreateInstance(typeForConditionList);

        // Adding 'where' predicate
        listOfConditions.GetType().GetMethod("Add").Invoke(listOfConditions, new[]
{ condition });

        // Setting new List<Condition<TYPE>>
        internalProperties[0].SetValue(mainProperty, listOfConditions, null);

        // Setting context
        internalProperties[1].SetValue(mainProperty, context, null);

        // Seeeting TDAQueryable<TYPE>
        prop.SetValue(inputItem, mainProperty, null);
    }
}

private static InternalTDAQueryable<T> GetData<T>(List<List<string>>
classAttributeInstances, PropertyInfo[] properties)
    where T : new()
{
    var data = new InternalTDAQueryable<T>();
    var instanceCount = classAttributeInstances.FirstOrDefault().Count;

    for (var i = 0; i < instanceCount; i++)
    {
        var t = new T();

        var counter = 0;
        foreach (var propInfo in properties)
        {
            var value = classAttributeInstances[counter++][i];
            propInfo.SetValue(t, value, null);
        }
```

```
            data.Add(t);
        }

        return data;
    }

    private static void ApplyFilters<T>(InternalTDAQueryable<T> transformedInput,
    List<Condition<T>> conditions)
        where T : new()
    {
        foreach (var condition in conditions)
        {
            var counter = 0;
            var itemsToRemove = new InternalTDAQueryable<T>();

            foreach (var inputItem in transformedInput)
            {
                if (condition.Type == Enums.ConditionTypes.Where
    && !condition.Predicate(inputItem))
                {
                    itemsToRemove.Add(inputItem);
                    continue;
                }
                else if (condition.Type == Enums.ConditionTypes.Skip && counter++ <
    condition.Amount)
                {
                    itemsToRemove.Add(inputItem);
                    continue;
                }
                else if (condition.Type == Enums.ConditionTypes.Take && counter++ >=
    condition.Amount)
                {
                    itemsToRemove.Add(inputItem);
                    continue;
                }
            }

            foreach (var itemToRemove in itemsToRemove)
            {
                transformedInput.Remove(itemToRemove);
            }
        }
    }

    private static string TranslateClassName(string className)
    {
        return className.Replace("__", "::").Replace("_", " ");
    }
```

Dokumentārā lapa

Bakalaura darbs "***LINQ modeļu repozitorijiem***" izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: (*personiskais paraksts*) Ivans Tabernakulovs

Rekomendēju darbu aizstāvēšanai

Vadītāja: docente Dr. dat. Elīna Kalniņa (personiskais paraksts) 30.05.2016.

Recenzents: profesors Dr. hab. dat. Audris Kalniņš

Darbs iesniegts Datorikas fakultātē 30.05.2016.

Dekāna pilnvarotā persona: metodiķe Ārija Sproģe (*personiskais paraksts)*

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

__.06.2016. prot. Nr. __.

Komisijas sekretār__: _____ (*personiskais paraksts)*