UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Danijel Hajdinjak

UPORABA LINQ-A PRI RAZVOJU POSLOVNIH APLIKACIJ

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: dr. Igor Rožanc

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00488/2009
Datum: 15.11.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DANIJEL HAJDINJAK**

Naslov: **UPORABA LINQ-A PRI RAZVOJU POSLOVNIH APLIKACIJ**
USING LINQ IN BUSINESS APPLICATION DEVELOPMENT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Poslovne aplikacije zahtevajo učinkovit dostop do podatkovnih zbirk. Za to se uporabljajo vmesniki, ki so pogosto nerodni za uporabo in namenjeni le za izbrane platforme. LINQ je sodoben dodatek v .NET ogrodju, ki omogoča enoten dostop do različnih podatkovnih virov na enostavnejši način.

V diplomski nalogi predstavite LINQ na teoretičen in praktičen način. Najprej predstavite možnosti uporabe LINQ poizvedb pri delu s podatki, nato pa z uporabo LINQ pristopa zasnujte in razvijte manjšo poslovno aplikacijo za obdelavo računov v manjšem podjetju. Nalogo zaključite z oceno uporabnosti tovrstnega pristopa.

Mentor:

viš. pred. dr. Igor Rožanc

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Franc Solina

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00488/2009
Datum: 15.11.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **DANIJEL HAJDINJAK**

Naslov: **UPORABA LINQ-A PRI RAZVOJU POSLOVNIH APLIKACIJ**
USING LINQ IN BUSINESS APPLICATION DEVELOPMENT

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

Poslovne aplikacije zahtevajo učinkovit dostop do podatkovnih zbirk. Za to se uporabljajo vmesniki, ki so pogosto nerodni za uporabo in namenjeni le za izbrane platforme. LINQ je sodoben dodatek v .NET ogrodju, ki omogoča enoten dostop do različnih podatkovnih virov na enostavnejši način.

V diplomski nalogi predstavite LINQ na teoretičen in praktičen način. Najprej predstavite možnosti uporabe LINQ poizvedb pri delu s podatki, nato pa z uporabo LINQ pristopa zasnujte in razvijte manjšo poslovno aplikacijo za obdelavo računov v manjšem podjetju. Nalogo zaključite z oceno uporabnosti tovrstnega pristopa.

Mentor:
viš. pred. dr. Igor Rožanc

Fakulteta za računalništvo in informatiko 2 UNIVERZA V LJUBLJANI

Dekan:
prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a DANIJEL HAJDINJAK,

z vpisno številko 63010208,

sem avtor/-ica diplomskega dela z naslovom:

UPORABA LINQ-A PRI RAZVOJU POSLOVNIH APLIKACIJ

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  viš. pred. dr. Igor Rožanc

  in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

Zahvala

Najprej bi se rad zahvalil staršem za možnost študija. Ženi in hčerkama za potrpežljivost ob pisanju tega dela. Posebno bi se rad zahvalil tudi mentorju dr. Igorju Rožancu za strokovno pomoč in nasvete ob izdelavi diplomske naloge.

Kazalo vsebine

Slike

Seznam uporabljenih kratic in simbolov

.NET	Ogrodje .NET - Microsoftovo ogrodje za razvoj programske opreme
ADO.NET	ActiveX Data Objects - Vmesnik za dostop do podatkov v ogrodju .NET
DBML	DataBase Markup Language - Razširljiv označevalni jezik podatkovnih baz
DDV	Davek na dodano vrednost
EDM	Entity Data Model - Entitetni podatkovni model
LINQ	Language Integrated Query - Komponenta ogrodja .NET za delo s podatki
SQL	Structured Query Language - Strukturirani povpraševalni jezik
XML	Extensible Markup Language - Razširljiv programski jezik
XPath	XML Path Language - Označevalni jezik za iskanje vozlišč v XML dokumentih

Povzetek

LINQ je komponenta ogrodja .NET, ki omogoča izvajanje poizvedb v .NET programskih jezikih.

LINQ bi bil lahko manjkajoči člen med podatkovnim svetom ter svetom splošno usmerjenih programskih jezikov. Združuje dostop do podatkov ne glede na vir le-teh in dovoljuje skupno uporabo podatkov različnih vrst. Dovoljuje podobne poizvedbene operacije in operacije za shranjevanje, kot jih omogoča pri delu s podatkovnimi bazami jezik SQL.

Ključni vidik razvoja LINQ-a je bil zagotoviti enoten programski model za uporabo nad vsemi tipi objektov ter podatkovnih virov in poenostaviti vmesnik za dostop in branje podatkov iz teh podatkovnih virov. S takim pristopom LINQ omogoča uporabo vseh jezikov, ki so podprti v ogrodju .NET na različnih tipih podatkovnih virov na enak način. S tem omogoča enostavno zamenjavo vrste podatkovnega vira brez večjih sprememb v programski kodi aplikacije.

LINQ je postal splošna zbirka orodij za poizvedbe, ki je hkrati vgrajena v programski jezik. Ta zbirka orodij se lahko uporablja za dostop do objektov v spominu, podatkovnih baz, XML dokumentov, sistemskih datotek kot tudi ostalih virov.

V osrednjem delu diplomske naloge bomo poskušali prikazati celoten potek razvoja aplikacije od podatkovnega modela do končnega produkta. Opisali bomo uporabniške zahteve, predstavili posamezne postopke razvoja in orisali posamezne nivoje aplikacije. Podrobno bomo predstavili kje v aplikaciji je bil uporabljen LINQ in kako so bile izdelane posamezne vnosne maske.

Ključne besede:

- LINQ
- Ogrodje .NET
- C#
- SQL
- delo s podatkovnimi bazami
- poslovne aplikacije

Abstract

LINQ is a component of the .NET framework, which brings the possibility of querying to the .NET languages.

LINQ could be the missing link between the world of data and the world of generally oriented programming languages. It combines access to data regardless of its source and allows sharing of data of different types. It allows similar querying updating operations as operations that are used in SQL for working with databases.

Key aspect of development of LINQ was to provide a unified programming model that could be used with all types of objects and data sources as well as simplifying reading from these sources. Such an approach lets you use all languages supported in the .NET framework languages on different data sources in the same way. This makes it easy to change the type of data source without major changes in the source code of applications.

LINQ became general set of tools for queries and it is a part of programming languages. This toolkit can be used to access objects in memory, databases, XML documents, system files, as well as other sources.

The central part of the graduation thesis will attempt to show the full path of development of applications from data modelling to the final product. We'll describe the user requirements, present the development of procedures and outline specific tiers of applications. We will present in detail where LINQ was used and how each of the input masks was made.

Keywords:

- LINQ
- .NET framework
- C#
- SQL
- Work with databases
- Business applications

1. Uvod

Ker je razvoj poslovnih aplikacij v praksi zahteven, glavni del razvoja pa je povezan s podatki, smo se odločili v diplomskem delu preučiti novejšo tehnologijo za delo s podatki na ogrodju .NET. Predstavili bomo LINQ (ang. Language Integrated Query), sestavo te tehnologije v teoriji in predstavili njene glavne konstrukte. Opisali bomo tudi, kateri konstrukti so nastajali znotraj ogrodja .NET v obdobju njegovega razvoja, in kako jih LINQ uporablja za svoje delovanje.

Preverili bomo tudi, kje so bile glavne slabosti do sedaj uporabljenih vmesnikov za dostop do podatkov ter kako jih v tem nadomešča LINQ.

Namen diplomskega dela je:

- pridobiti nova znanja za razvoj aplikacij,
- se naučiti obravnavati uporabnikove zahteve in želje, ter
- prikazati razvoj poslovne aplikacije z uporabo izbrane tehnologije in izbranih prijemov.

Cilji diplomskega dela so:

- osvojiti novo tehnologijo na področju razvoja aplikacij,
- utrditi znanje in sposobnosti s področja dela z uporabniki,
- izdelati aplikacijo in raziskati možne razširitve za nadaljnji razvoj.

V teoretičnem delu diplomske naloge bomo poskušali zajeti vse prijeme, ki jih morajo razvijalci osvojiti za uporabo LINQ-a v praksi. Posamezne prijeme bomo prikazali tudi na primerih in le-te nekoliko razložili.

V nadaljevanju bomo predstavili v diplomski nalogi uporabljena programska orodja in njihovo uporabo ter glavne lastnosti.

V praktičnem delu bomo na primeru razvoja poslovne aplikacije prikazali celoten potek razvoja, od ideje do končnega produkta. Opisali bomo uporabniške zahteve, predstavili posamezne postopke razvoja in orisali posamezne nivoje aplikacije. Podrobno bomo predstavili, kje v aplikaciji je bil uporabljen LINQ, in kako so bile izdelane posamezne vnosne maske.

Na koncu bomo predstavil še probleme, ki smo jih srečali pri razvoju ter opisali izboljšave, ki so se razkrile med razvojem in ob dejanski implementaciji aplikacije v delovnem okolju.

2. LINQ v teoriji

2.1. Razvoj poslovnih aplikacij

Uporaba poslovnih aplikacij je v praksi enostavna, razvoj le-teh pa zapleten. Eden od pomembnejših delov razvoja poslovnih aplikacij je razvoj metod za delo s podatkovnimi strukturami. Izbiramo lahko med različnimi programskimi jeziki, njihova izbira pa je odvisna od različnih dejavnikov, kot so: specifične zahteve poslovne aplikacije, znanje programerja, ki aplikacijo razvija, trenutni trend, operacijski sistem aplikacije, itd.

Ne glede na to, za kateri jezik se odločimo, vedno pridemo do točke, ko se srečamo z obdelavo podatkov. Ti podatki lahko izhajajo iz različnih virov, od tabel aplikacije do podatkov v datotekah ali celo podatkov z interneta.

Glede na razširjenost uporabe dela s podatki, bi pričakovali, da je pri večini programskih jezikov delo s podatki enostavno in dobro definirano. .NET kot eno novejših programskih ogrodij omogoča velik nabor vmesnikov za delo s podatki, vendar je očitno, da je mogoče delo s podatki še veliko bolj poenotiti ter poenostaviti glede na različne tipe virov podatkov.

Tukaj nastopi nov način razmišljanja ter nov način dela s podatkovnimi strukturami z imenom LINQ (ang. Language Integrated Query).

2.2. Ogrodje Microsoft .NET

Microsoft .NET ogrodje (ang. Microsoft .NET Framework) je programsko ogrodje, ki je namenjeno uporabi v operacijskih sistemih Microsoft Windows [1]. Vsebuje široko zbirko knjižnic (ang. libraries) za razvoj aplikacij in virtualni stroj (ang. virtual machine), ki upravlja z izvrševanjem programov, ki so namenjeni temu ogrodju.

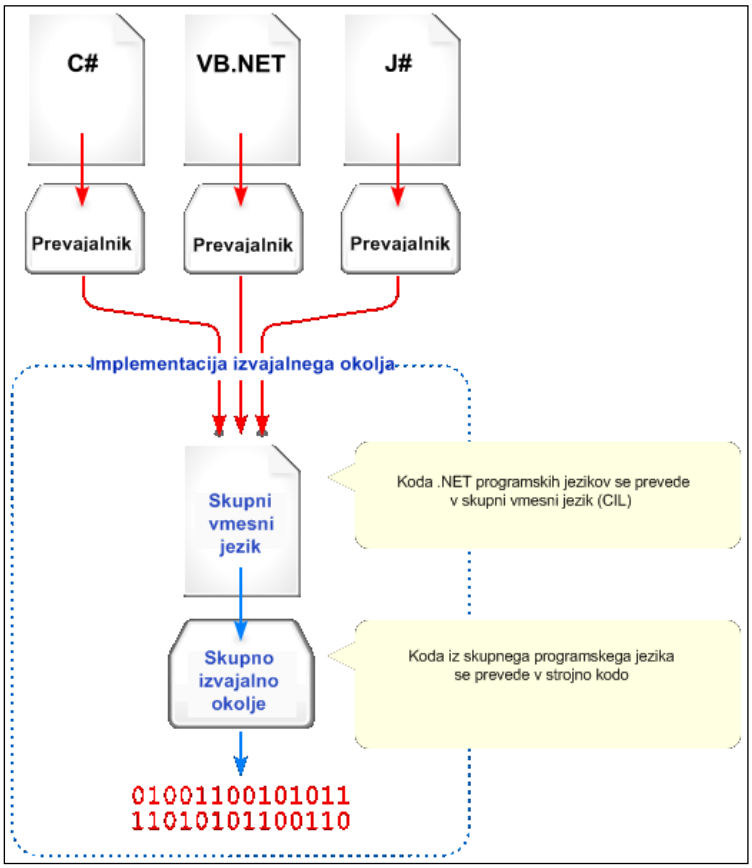
V dobrih petih letih si je sledilo več izdaj.

- Prva izdaja .NET ogrodja, (verzija 1.0) je bila izdana februarja 2002.
- .NET 1.1 se pojavi v aprilu 2003. del te druge izdaje programa pa je tudi Microsoft Visual Studio .NET.
- Ogrodje .NET 2.0 je bilo izdano 2005 v sklopu programa Visual Studio 2005 in Microsoft SQL Server 2005.
- Dobro leto pozneje izide verzija 3.0 (november 2006).
- Zadnja izdana verzija je verzija 3.5 (novembra 2007) .

Delo v diplomski verziji je zasnovano na .NET ogrodju verzije 3.5, ki vsebuje knjižnice za delo z LINQ-om [2].

Ogrodje .NET uporablja vmesnik, ki je skupen vsem podprtim jezikom (ang. Common Language Infrastructure) in je zato lažje obvladljiv, saj ponuja vsem trem jezikom (C#, J# in Visual Basic .NET) enak nabor knjižnic in zelo podobno uporabo le-teh. Slika 1 prikazuje

kako ti trije jeziki uporabljajo skupni vmesnik, ki preko skupnega izvajalnega okolja (ang. Common Language Runtime) pretvori kodo v strojno kodo [1].



Slika 1: Uporaba skupnega prevajalnega vmesnika

2.3. Kaj je LINQ in zakaj se uporablja

LINQ je komponenta .NET ogrodja, ki omogoča izvajanje poizvedb (ang. query) v .NET programskih jezikih.

LINQ bi lahko bil manjkajoči člen med podatkovnim svetom ter svetom splošno usmerjenih programskih jezikov [3]. Združuje dostop do podatkov, ne glede na vir le-teh, in dovoljuje skupno uporabo podatkov različnih vrst. Dovoljuje podobne poizvedbene operacije in operacije za shranjevanje kot jih omogoča jezik SQL za delo s podatkovnimi bazami, in je zato zelo nazoren z vidika uporabe. Poleg tega definira tudi množico operatorjev, ki so na voljo za poizvedovanje in filtriranje podatkov v tabelah, XML dokumentih (ang. Extensible Markup Language), relacijskih podatkovnih zbirkah in zunanjih podatkovnih virih.

Poizvedbe so vgrajene v .NET jezike (C# in Visual Basic) in s pomočjo razširitve teh jezikov, ki so nastale pri vključevanju LINQ-a. Te razširitve bomo podrobno opisali v nadaljevanju diplomskega dela.

Pred nastankom LINQ-a smo morali za različne podatkovne jezike (SQL, XML, XPath, itd) uporabljati različne vmesnike, kot je sta na primer vmesnika ADO.NET ali System.Xml. Tak

pristop je seveda botroval velikim težavam pri uporabi. LINQ povezuje te svetove med seboj ter se tudi izogiba težavam, na katere bi naleteli pri združevanju teh jezikov.

Eden ključnih vidikov je bil zagotoviti enoten programski model za uporabo nad vsemi tipi objektov ter podatkovnih virov, in zagotoviti dosleden programski model za uporabo na njih. Sintaksa in koncepti so v LINQ-u enaki ne glede na trenutno uporabo. Ko osvojimo uporabo na enem od podatkovnih virov ali objektov, smo dejansko osvojili tudi vse koncepte in sintaktična pravila tega poizvedbenega jezika, ki se uporabljajo na objektih ostalih tipov.

Drugi pomemben vidik je uporaba strogega preverjanja tipov (ang. strongly typed). To omogoča preverjanje poizvedb v času prevajanja in uporabne napotke pomočnika za kodiranje (ang. IntelliSense) v Visual studiu.

LINQ je znatno spremenil tudi nekatere vidike obdelovanja podatkov v aplikacijah. Njegova uporaba je korak naprej v deklarativnem programskem modelu. Izvirna ideja, in hkrati motivacija za nastanek LINQ-a je izhajala iz konceptualnih in tehničnih težave pri uporabi podatkovnih baz v programskih jezikih .NET. Z LINQ-om je hotel Microsoft predstaviti rešitev objektno-relacijskega mapiranja, kot tudi poenostaviti prenos podatkov med objekti in podatkovnimi viri.

LINQ je postal splošna zbirka orodij za poizvedbe, ki je vgrajena v programski jezik. Ta zbirka orodij se lahko uporablja za dostop do objektov v spominu, podatkovnih baz, XML dokumentov, sistemskih datotek kot tudi ostalih virov.

Trenutna orodja so v primerjavi z LINQ neuporabna ali manj uporabna. Kot smo že omenili, je večina poslovnih aplikacij narejena tako, da se uporablja v povezavi s podatki. Za razvoj take aplikacije ni dovolj, da se le naučimo jezika, s katerim bomo napisali programsko kodo, temveč se moramo naučiti tudi jezika za delo na želenem podatkovnem viru, na primer jezika SQL za delo z bazami. Hkrati s tem se moramo tudi naučiti uporabljati vmesnike, ki jih izbrani jezik ponuja za povezovanje z bazo oziroma podatki.

V nadaljevanju bomo izpostavili osnovne težave, ki nastanejo pri uporabi vmesnika, ki ga v osnovi ponuja ogrodje .NET.

2.4. Osnovni vmesnik

Pogosta uporaba podatkovnih struktur je zahtevala od ogrodja .NET, da vpelje knjižnice za delo na podatkih. Tako uporablja več knjižnic za dostop do podatkov. Ena najbolj uporabljanih je ADO.NET, ki vsebuje vmesnik za dostop in uporabo relacijskih baz ter za prenos relacijskih podatkov v pomnilnik, do katerega lahko dostopa programski jezik.

Ta vmesnik je sestavljen iz razredov `SqlConnection`, `SqlCommand`, `SqlReader`, `DataSet` in `DataTable`. Problem teh razredov je, da programerja silijo v delo neposredno s tabelami, vrsticami in stolpci podatka, medtem ko moderni programski jeziki (Visual Basic, C#, itd.) uporabljajo objektno usmerjeno programiranje. Zato programerji porabijo preveč časa za povezovanje teh objektov na podatke v relacijski bazi, torej na posamezne tabele. Če bi razvoj tega dela lahko poenostavili, bi po ocenah lahko pridobili od 30 do 40 % časa razvoja. Pri tem nam lahko pomaga LINQ.

Pri tem ne gre samo za večjo učinkovitost, ampak tudi za večjo kvaliteto. Mapiranje med objekti in relacijsko bazo namreč predstavlja verjetno mesto napak. Kot primer naj navedem, da se za pridobitev podatka iz posamezne vrstice v tabeli navede kar ime stolpca ali celo zaporedna številka stolpca, brez preverjanja pravilnosti imena stolpca med prevajanjem programske kode. S tem pa seveda obstaja možnost, da bo med izvajanjem programa na tem mestu prišlo do napake. Prikaz uporabe osnovnega vmesnika ogrodja .NET prikazuje slika 2 spodaj.

```
using (SqlConnection connection = new SqlConnection("...")) {
  connection.Open();
  SqlCommand command = connection.CreateCommand();
  command.CommandText =
    @"SELECT Name, Country          1. SQL ukaz
        FROM Customers        <---  v tekstu
       WHERE City = @City";
  command.Parameters.AddWithValue("City", "Paris"); <--- 2. Nepovezani
  using (SqlDataReader reader = command.ExecuteReader()) {  parametri
    while (Reader.Read()) {
      string name = reader.GetString(0);   <--- 3. Stolpci brez
      string country = reader.GetString(1);      znanega tipa
      ...
    }
  }
}
```

Slika 2: Uporaba osnovnega vmesnika za dostop do podatkov

Že s hitrim pregledom kode lahko opazimo nekaj omejitev modela:

- Čeprav želimo izvršiti enostavno opravilo, je potrebno za to kar nekaj korakov.
- Poizvedbe predstavljajo ukaz, ki je vpisan neposredno v kodi (1. točka). Mogoče stavek SQL ni pravilen in vsebuje sintaktično napako? Kaj če se stolpec v tabeli preimenuje?
- Enako velja za točki 2. in 3. s slike. Sta stolpca v tabeli pravega tipa? Uporabljamo pravo število parametrov za dodajanje zapisa?
- Ta razred se lahko uporablja samo za delo s SQL bazami in ga ni mogoče uporabiti na ostalih tipih podatkovnih virov.

Seveda obstajajo tudi druge rešitve in orodja, ki omogočajo generiranje objektov, ki se nato mapirajo na podatkovno strukturo, vendar imajo tudi ta orodja svoje omejitve. Večina jih recimo zna uporabljati samo en tip podatkovnega vira.

Poleg tega si veliki proizvajalci (kot je Microsoft) lahko privoščijo, da vmesnike za dostop do podatkov vgradijo neposredno v programski jezik, česar si seveda manjši proizvajalci orodij za mapiranje ne morejo privoščiti.

Slika 3 prikazuje primer uporabe LINQ-a, kjer je lahko vir v ozadju relacijska baza, podatki v pomnilniku, XML dokument ali kaj drugega.

```
from customer in customers
where customer.Name.StartsWith("A") && customer.Orders.Count > 0
orderby customer.Name
select new { customer.Name, customer.Orders }
```

Slika 3: Enostavna poizvedba z uporabo LINQ-a

Največja težava vseh orodij je mapiranje objektnega modela na relacijski model, saj že v osnovi modela nista sestavljena iz enakih primitivnih tipov. Kot primer naj navedem, da besedilo v objektnem modelu, ni omejeno z dolžino, medtem ko v relacijskem je, zato se mapiranje besedila v polje v tabeli ne izvede vedno uspešno.

Predvidevamo lahko, da nobena od rešitev ni optimalna. Glavne omejitve so:

- Potrebno je dobro poznati posamezno orodje, da lahko s pravilno uporabo zadostimo performančnim zahtevam.
- Optimalna uporaba še vedno zahteva zelo dobro poznavanje relacijskih baz oziroma ustreznega drugega tipa virov podatkov.
- Orodja za mapiranje niso vedno tako učinkovita kot namensko napisana koda.
- Vsa orodja niso podprta s preverjanjem kode med prevajanjem.

2.5. C# konstrukti v LINQ-u

Če želimo razumeti delovanje LINQ-a moramo najprej pojasniti nekaj konstruktov jezika C# [4]. Najprej si poglejmo spremembe, ki so bile vpeljane s .NET 2.0.

2.5.1. Konstrukti dodani v C# 2.0

2.5.1.1. Generiki

Z njihovo pomočjo lahko razredom v katere vpisujemo več podatkov, vnaprej definiramo, kakšen tip vsebujejo. S tem se znebimo nepotrebnega eksplicitnega pretvarjanja v želeni tip (slika 4).

```
List<int> list = new List<int>();
list.Add(3);
int i = list[0];
```

Slika 4: Uporaba generikov

2.5.1.2. Delegati

Delegati ponazarjajo podpis metode. Obnašajo se kot kazalec na metodo, ki jo opisuje delegat. Če obstaja njegova instanca, lahko pokličemo metodo, na katero kaže kazalec (slika 5).

```
delegate void TwoParamsDelegate(string name, int age);

public class DemoDelegate {
  void MethodC (string x, int y) {...}

  void CreateInstance() {
    TwoParamsDelegate c = MethodC;
  }
}
```

Slika 5: Uporaba delegatov

2.5.1.3. Anonimne metode

Anonimne metode omogočajo, da na mesto, kjer bi v C# 1.0 uporabili delegate, vpišemo kar blok kode in delegata ni potrebno posebej deklarirati.

Primer uporabe anonimnih metod je predstavljen na sliki 6.

```
public class DemoDelegate {
  void Repeat10Times (TwoParamsDelegate callback) {
    for (int i = 0; i < 10; i++) {
      callback("Linq book", i);
    }
  }

  void Run3() {
    Repeat10Times(delegate(string text, int age) {
      Console.WriteLine("{0}{1}", text, age);
    }
  }
}
```

Slika 6: Uporaba anonimnih metod

2.5.1.4. Iteratorji

Iteratorji (ang. Enumerators) omogočajo sprehod po vrednostih s pomočjo metode `MoveNext()` (slika 7).

```
public void DemoCountDown () {

  CountDown countdown = new CountDown();
  IEnumerator i = countdown.GetEnumerator();
  while (i.MoveNext()) {
    int n = (int) i.Current;
    Console.WriteLine(n);
  }
}
```

Slika 7: Uporaba iteratorjev

2.5.2. Konstrukti dodani v C# 3.0

2.5.2.1. Implicitno določanje tipa lokalnim spremenljivkam

Ta funkcionalnost omogoča, da lokalnim spremenljivkam naknadno definiramo tip. Spremenljivki se namreč njen tip nastavi ob prvem prirejanju vrednosti. To nam omogoča lahkotnejše programiranje, saj ni potrebno vnaprej vedeti, katerega tipa bo spremenljivka. `Var` spremenljivke je mogoče uporabljati samo lokalno (slika 8).

```
var a = 2;          // Spremenljivka je deklarirana kot int
object b = 2;       // b je tipa objekt, ki vsebuje vrednost tipa int 2
int c = a;          // c-ju lahko priredimo vrednost a, saj je tipa int
int d = (int) b;    // eksplicitno moramo povedati, da
                    // želimo int vrednost spremenljivke b
```

Slika 8: Implicitno določanje tipa

Šele z uvedbo te funkcionalnosti je možno definirati anonimne tipe.

2.5.2.2. Lambda izrazi

Lambda izrazi omogočajo funkcionalen zapis anonimnih metod. V C# 2.0. je bilo potrebno metodo, ki sprejema delegate zapisati takole [5] (slika 9):

```
customers.Where(delegate(Customer c) {
      return c.Name == "John";
});
```

Slika 9: Metoda z delegatom v C# 2.0

V C# 3.0. pa lahko isto metodo zapišemo kar takole (slika 10):

```
customers.Where(c => c.Name == "John");
```

Slika 10: Lambda izraz v C# 3.0

Iz primera je razvidno, da je izraz lambda sestavljen iz parametrov, ki jim sledi operator => in nato izraz ali blok, ki se izvede ob izračunavanju vrednosti izraza lambda.

2.5.2.3. Anonimni tipi

Anonimni tipi ponujajo možnost, da razvijalec v kodi definira razred, ne da bi za to potreboval formalno deklaracijo le-tega. Primer uporabe si poglejmo na sliki 11.

```
var products =
  from p in db.Products
  where p.Price > 50
  select new {Id = p.Id, name = p.Name};
```

Slika 11: Uporaba anonimnih tipov

2.5.2.4. Razširitvene metode

Razširitvene metode (ang. Extension methods) lahko dodajamo razredom, ne da bi morali spreminjati njihovo izvorno kodo. Na sliki 12 je prikazana uporaba teh metod.

```
public static Filled(this string s) {
  return s != "";
}

...

  // Sedaj imajo vse spremenljivke tipa string to metodo,
  // ki preverja če je spremenljivna napolnjena
  string sText = "Test";
  if (sText.Filled) {
    ...
  }
```

Slika 12: Uporabe razširitvenih metod

2.5.2.5. Inicializatorji objektov

Kreiranju objektov (ang. Object initializers) v C# pogosto sledi nastavljanje posameznih lastnosti novega izvoda objekta. Razvijalci zato precej pogosto pišemo kodo kot kaže slika 13.

```
Customer c = new Customer();
c.Name = "John";
c.LastName = "Tex";
```

Slika 13: Inicializacija objektov

Po novem lahko to kodo zapišemo tako kot prikazuje slika 14:

```
Customer c = new Customer {Name = "John", LastName = "Tex"};
```

Slika 14: Inicializacija objektov v C# 3.0

2.5.2.6. Poizvedbeni izrazi

Z novimi rezerviranimi besedami lahko v kodi C# zapišemo poizvedbo za LINQ [4]. Sintaksa je podobna jeziku SQL. Koda se nato pretvori v navadno kodo C# 3.0. in pri tem se uporabijo posebni razredi in metode, ki so del LINQ knjižnic. Primer je prikazan na sliki 15.

```
var customers = new [] {
  new {Name = "Marco", Discount = 4.5},
  new {Name = "Paolo", Discount = 3.0},
  new {Name = "Tom", Discount = 3.5}
};

var query =
  from c in customers
  where c.Discount > 3
  orderby c.Discount
  select new {c.Name, Perc = c.Discount / 100};
```

Slika 15: Primer poizvedbenega izraza

2.6. Sintaktična pravila

V tem podpoglavju bomo poskušali opisati osnovne razrede in operatorje, na katerih je LINQ zasnovan. S tem bomo razumeli njegovo osnovno arhitekturo in sintakso. Kot smo že zapisali v prejšnjih poglavjih, je LINQ namenjen uporabi na različnih tipih objektov. V tem delu bomo skušali prikazati uporabo na objektih (ang. LINQ to Objects) ter pri uporabi na SQL bazah.

2.6.1. LINQ poizvedbe

LINQ je zasnovan na množici operatorjev, ki so definirani kot razširitvene metode, in je uporaben na vseh objektih, ki implementirajo `IEnumerable<T>` (več o tem tipu je že bilo napisano v poglavju 2.5.1.4) [4]. S tem pristopom postane LINQ splošno uporaben, saj večina seznamov že implementira ta tip, poleg tega si pa lahko vsak razvijalec razvije svoje sezname, ki to implementirajo. Prav tako si lahko vsak razvijalec zaradi dodatnih razširitev metod razvije svoje metode, ki so specializirane za delo na specifičnih podatkih. Tak primer je v bistvu tudi LINQ za SQL (ang. LINQ to SQL), ki je le razširitev metod, ki jih ponuja LINQ za objekte (ang. LINQ to Objects).

2.6.2. Sintaksa

Primer poizvedbenega stavka prikazuje slika 16. Ta primer bo v nadaljevanju tudi podrobneje opisan.

```
public class Developer {
  public string Name;
  public string Language;
  public int Age;
}

public class DeveloperDemo {
  static void DeveloperTest {
    Developer[] developers = new Developers[] {
      new Developer {Name = "Paolo", Language = "C#"},
      new Developer {Name = "Marco", Language = "C#"},
      new Developer {Name = "Frank", Language = "VB.NET"}
    }

    IEnumerable<string> developerUsingCsharp =
      from d in developers
      where d.Language == "C#"
      select d.Name
  }
}
```

Slika 16: Primer LINQ sintakse poizvedbenega izraza

V tem primeru je najprej deklariran tip `Developer` [4]. Recimo, da potrebujemo poizvedbo na seznamu tega tipa, s katero bi radi dobili seznam razvijalcev, ki programirajo v jeziku C#. Rezultat poizvedbe v primeru sta razvijalca `Paolo` in `Marco`. Sintaksa je v tem primeru podobna SQL stavku z nekaj razlikami. Za boljše razumevanje bomo kodo razdelil na dele.

Rezultat je definiran z rezervirano besedo `select` in podatkom, ki bo vrnjen (slika 17):

```
select d.Name
```

Slika 17: Rezultat pri poizvedbi

Poizvedba se bo izvršila na podatkih iz podatkovnega vira. Rezervirana beseda, ki vpeljuje vir je `from`, za njo je naveden vir. Vir je lahko katerakoli struktura, ki implementira `IEnumerable<T>`. Primer je na sliki 18:

```
from d in developers
```

Slika 18: Navedba vira v poizvedbi

Filter na podatkih definiramo s besedo `where` in vsebino filtra:

```
where d.Language == "C#"
```

Slika 19: Filter pri poizvedbi

Pogoj v filtru se enostavno prevede z uporabo `Where` razširitvene metode, ki je ena od metod `Enumerable` razreda in je definirana v imenskem prostoru `System.Linq`. Dve od teh metod sta tudi `Select` in `From`.

Vsaka poizvedba se začne z ukazom `from` in konča z ukazom `select` ali `group`. Razlog, da se začne s `from` namesto s `select` (kot je navada pri sintaksi jezika SQL) je ta, da lahko okolje za razvoj ugotovi kaj nam vir podatkov ponuja. S tem je namreč omogočeno, da se lahko koda že med pisanjem samodejno dokončuje in samodejno ponuja metode in lastnosti, ki so na voljo za ta vir.

Rezultat `select` stavka je vedno objekt, ki implementira razred `IEnumerable`, kar omogoča nadaljnje poizvedbe na dobljenem objektu. Tudi ukaz `group` vrne seznam skupin glede na pogoje v `group` ukazu, kjer je vsaka novo ustvarjena skupina spet implementacija `IEnumerable` razreda.

Prvemu `from` ukazu lahko sledi eden ali več ukazov `from`, `let` ali `where`. Ukaz `let` doda rezultatu naziv, medtem ko `where` ukaz definira filter, ki bo na podatkih izveden. Vsak `from` stavek predstavlja generator iteracij na nekem viru na katerem želimo izvršiti poizvedbo. Ukazu `from` lahko sledijo ukazi `join`. Pred zadnjim `select` ali `group` stavkom je lahko postavljen še `orderby` stavek, ki definira vrstni red zapisov v končnem rezultatu [4].

2.6.3. Poizvedbeni operatorji

Zadnji del v tem podpoglavju opisuje glavne metode in generične delegate, ki jih ponuja imenski prostor `System.Linq` za poizvedovanje.

2.6.3.1. Operator `Where`

`Where` operator je namenjen filtriranju podatkov, kar omogoča, da iz celotnega nabora izločimo samo želene podatke.

Za uporabo imamo na voljo dve metodi, ki sta prikazani na sliki 20. Prva metoda je namenjena osnovni uporabi, druga pa omogoča še določitev indeksa v seznamu, ki nas zanima. Tako lahko omejimo končni nabor podatkov, če nas recimo zanima samo prvih nekaj zapisov. Kot je razvidno iz primera pri drugi metodi ni mogoča uporaba LINQ poizvedbene sintakse.

```
var expr =
      from c in customers
      where c.Country == Countries.Italy
      select new {c.Name, c.City};

    var expr =
      customers
      .Where((c, index) => (c.Country == Countries.Italy && index >= 1))
      .Select(c => c.Name)
```

Slika 20: Uporabe operatorja `where`

2.6.3.2. Operatorji za vračanje rezultatov

Ta del opisuje uporabo operatorjev za vračanje rezultatov. S temi operatorji preberemo podatke iz vira v končni nabor podatkov v pomnilniku.

- **Operator `Select`**
  Je eden izmed operatorjev, s katerimi vračamo rezultate, ki jih lahko spet uporabljamo za naslednje poizvedbe, ker le-ti implementirajo `IEnumerable` razred. V `select` operatorju določimo tudi tip rezultata. Kot primer vzemimo spodnji predikat (slika 21):

```
var expr = customers.Select(c => c.Name)
```

Slika 21: Uporaba operatorja `select`

Rezultat te poizvedbe bo seznam objektov tipa `IEnumerable<string>`. Če za primer vzamemo poizvedbo kot je na sliki 22:

```
var expr = customers.Select(c=> new {c.Name, c.City});
```

Slika 22: Uporaba operatoja select s seznamom objektov

pa kot rezultat dobimo seznam objektov anonimnega tipa, ki so definirani kot naziv in mesto za vsako stranko v seznamu.

- **Operator `SelectMany`**
  Recimo, da želimo dobiti seznam vseh naročil za stranke iz Italije. Lahko bi napisali poizvedbo kot na sliki 23.

```
var orders =
  customers
  .Where(c => c.Country == Countries.Italy)
  .Select(c => c.Orders);
```

Slika 23: Poizvedba za naročila strank iz Italije

S tem bi kot rezultat dobili seznam naročil tipa `IEnumerable<Order[]>` za vsako stranko. Če pa bi želeli dobiti seznam naročil za vse stranke v enem seznamu lahko uporabimo operator `SelectMany`, katerega uporaba je prikazana na sliki 24.

```
var orders =
 customers
 .Where(c => c.Country == Countries.Italy)
 .SelectMany(c => c.Name)
```

Slika 24: Uporaba operatorja `SelectMany`

2.6.3.3. Operatorji za sortiranje

Z njimi določamo vrstni red podatkov v končnem rezultatu.

- **`OrderBy` in `OrderByDescending`**

Včasih je uporabno, če podatke v seznamu uredimo po vrsti. LINQ lahko podatke sortira v naraščajočem ali padajočem vrstnem redu kot je prikazano na sliki 25.

```
var expr =
  from c in customers
  where c.Country == Countries.Italy
  orderby c.Name descending
  select new {c.Name, c.City};
```

Slika 25: Poizvedba s sortiranje podatkov

Razlika med `OrderBy` in `OrderByDescending` je v vrstnem redu podatkov, saj prvi operator uporablja naraščajoče sortiranje, drugi pa padajoče. Za primerjanje uporabljata oba operatorja privzeti `Comparer`, katerega ima implementiranega razred `IEnumerable`. Ena od možnih uporab omogoča tudi izbiro, kateri `IComparer` naj metoda uporabi.

- **`ThenBy` in `ThenByDescending`**
  Ta dva operatorja se uporabljata takrat, ko bi podatke radi sortirali po več lastnostih.

  Slika 26 prikazuje primer uporabe, kjer bi podatke radi sortirali po nazivu padajoče, znotraj tega pa še po mestu naraščajoče.

```
var expr = customers
  .Where(c => c.Country == Countries.Italy)
  .OrderByDescending(c => c.Name)
  .ThenBy(c => c.City)
  .Select(c => new {c.Name, c.City});
```

Slika 26: Uporabe operatorja `ThenBy`

2.6.3.4. Operatorji za grupiranje

Včasih v poizvedbi potrebujemo rezultat grupiran pod določenimi kriteriji. To omogoča operator `GroupBy` (slika 27).

```
var expr =
  customers
  .GroupBy(c => c.Country, c => c.Name);

//Rezultat poizvedbe
Country: Italy
  Paolo
  Marco
Country: USA
  James
  Frank
```

Slika 27: Uporaba operatorja `GroupBy`

2.6.3.5. Operatorji za združevanje

Ti operatorji se uporabljajo za definiranje razmerij med podatki v tabelah in LINQ poizvedbo. Pri SQL poizvedbah vsebuje skoraj vsaka poizvedba združevanje več tabel. Zato ima tudi LINQ več takih operatorjev.

- **Operator `Join`**
  Primer uporabe operatorja `Join`, ki nam omogoča stikanje več tabel, prikazuje slika 28.

```
var expr =
     customers
     .SelectMany(c => c.Orders)
     .Join(products,
           o => o.IdProduct,
           p => p.IdProduct,
           (o, p) => new {o.Month, o.Shipped, p.IdProduct, p.Price});
```

Slika 28: Uporaba operatorja `Join`

- **Operator `GroupJoin`**
  Ko želimo uporabiti nekaj podobnega kot sta v SQL-u `LEFT JOIN` in `RIGHT JOIN` moramo v LINQ-u uporabiti `GroupJoin` operator. Prikaz uporabe se nahaja na sliki 29.

```
var expr =
     products
     .GroupJoin(customers.SelectMany(c => c.Orders),
           p => p.IdProduct,
           o => o.IdProduct,
           (p, orders) => new {p.IdProduct, Orders = orders});
```

Slika 29: Uporaba operatorja `GroupJoin`

2.6.3.6. Operatorji za delo na nizih

- **Operator `Distinct`**
  Recimo, da bi želeli poizvedbo, ki vrne vse produkte, ki so bili kdajkoli naročeni. Seveda nas ne zanima to, na koliko naročilih je izdelek naveden. V tem primeru lahko uporabite operator `Distinct`, kot je prikazano na sliki 30.

```
var expr =
      (from c in customers
          from o in c.Orders
          join p in products
                    on o.IdProduct equals p.IdProduct
          select p
       ).Distinct();
```

Slika 30: Uporaba operatorja `Distinct`

- **Operatorji `Union`, `Intersect` in `Except`**
  Ti operatiroji skrbijo za pregled unije, preseka in razlik.
  Primeri uporabe so prikazani na sliki 31:

```
var expr = customers[1].Orders.Union(customers[2].Orders);
var expr = customers[1].Orders.Intersect(customers[2].Orders);
var expr = customers[1].Orders.Except(customers[2].Orders);
```

Slika 31: Primeri uporabe operatorjev `Union`, `Intersect` in `Except`

2.6.3.7. Ostali operatorji

Na tej točki bomo našteli le nekaj operatorjev, ki jih ne bomo posebej obdelovali, saj za prikaz uporabe LINQ- niso bistvenega pomena.

- `Count` in `LongCount`: preštejejo zapise
- `Sum`: sešteje vrednosti vseh zapisov
- `Min` in `Max`: izbere najmanjšega oz. največjega med podatki
- `Average`: izračuna povprečje vrednosti podatkov

2.7. LINQ in delo s podatki

2.7.1. LINQ za delo s SQL bazami

Prva in najbolj samoumevna je bila izvedba LINQ-a za delo na relacijskih bazah. LINQ za SQL je LINQ-ova komponenta, ki ponuja možnost izvajanja poizvedb na relacijskih bazah, ob tem pa ponuja objektni model, ki je zasnovan na objektih te konkretne baze. Z drugimi besedami omogoča, da iz baze pridobimo množico objektov, na katerih lahko nato poganjamo poizvedbe. LINQ-ove knjižnice pri tem poskrbijo, da se poizvedbe pretvorijo v jezik SQL in poženejo na bazi. Na sliki 32 je prikazan primer poizvedbe v LINQ-u, ki razkrije, kakšna je dejanska poizvedba, ki se nato požene.

```
var query =
      from c in Customers
      where c.Country == "USA"
            && c.State == "WA"
      select new {c.CustomerID, c.CompanyName, c.City}

  //SQL stavek
  SELECT CustomerID, CompanyName, City
    FROM Customers
   WHERE Country = 'USA'
     AND State  = 'WA'
```

Slika 32: Poizvedba v LINQ in primer SQL stavka, ki pri tem nastane

2.7.1.1. Poizvedbe

Poizvedba pri LINQ za SQL je poslana na bazo šele takrat, ko program dejansko potrebuje podatke. To pomeni, da če poizvedbo napišemo, vendar rezultata nikjer ne uporabimo, se poizvedba ne bazi sploh ne sproži. Paziti moramo na to, da vsak klic metode `GetEnumerator`, ki se sproži ob sprehodu po podatkih, ki naj bi jih poizvedba vrnila,

sproži novo poizvedbo na bazi. Zato moramo biti pazljivi pri uporabi `foreach` stavkov. Veliko bolj primerno je, da pred sprehodom podatke shranimo v pomnilnik. Primer nepotrebnih klicev prikazuje slika 33.

```
var query =
      from c in Customers
      where c.Country == "USA"
            && c.State == "WA"
      select new {c.CustomerID, c.CompanyName, c.City}


  foreach(var row in query) {
    Console.WriteLine(row);
  }
```

Slika 33: Primer LINQ poizvedbe

2.7.1.2. Dodajanje in spreminjanje podatkov

LINQ sledi vsem instancam entitet s pomočjo servisa za urejanje identitet zato, da ima ena vrstica podatkov vedno samo eno identiteto med entitetami. Za to skrbi razred `DataContext`. S tem, ko ima vsaka vrstica samo en zapis, lahko brez skrbi obdelujemo podatke pomnilnika, saj servis poskrbi, da so ti podatki enaki kot na bazi. Kadar popravimo podatke v neki entiteti, servis poskrbi za to, da se generirajo stavki SQL, ki na bazi naredijo enake spremembe. To prikazuje slika 34.

```
var customer db.Customers.Single(c => c.CustomerID = "FRANS");
customer.ContactName = "Marco Russo";

UPDATE [Customers] SET [ContactName] = "Marco Russo"
WHERE  [CustomerID] = "FRANS"

var newCustomer = new Customer {CustomerID = "DLEAP", CompanyName = "DevLeap"}
db.Customers.Add(newCustomer);

INSERT INTO [Customers] (CustomerID, CompanyName)
VALUES ("DLEAP", "DevLeap")
```

Slika 34: Primer spreminjanja in dodajanja podatkov

2.7.2. LINQ za delo s podatkovnimi nizi

Opisali bomo delo z LINQ za podatkovne nize (ang. `LINQ to DataSet`). Osnovni .NET `System.Data.DataSet` je predstavitev podatkov, ki so že v pomnilniku. Uporaben je za pridobitev samostojnih podatkov, ki prihajajo iz zunanjega vira. Ne glede na vir podatkov sledi predstavitev le-teh v podatkovnem nizu (ang. `DataSet`) relacijskega modela. Ta vsebuje tabele in relacije med njimi. Z drugimi besedami si lahko predstavljamo podatkovni niz kot nek tip relacijske baze, ki je shranjena v pomnilniku. Tak tip podatkov je primerna tarča za uporabo LINQ-a za nize podatkov.

2.7.2.1. Polnjenje podatkovnih nizov

Podatke lahko polnimo z izvršitvijo poizvedbe na neki relacijski bazi. Ena od možnosti je, da uporabimo razred `DataAdapter`. Primer polnjenja podatkovnega niza si oglejmo na sliki 35.

```
DataSet ds = new DataSet("CustomerOrders");
SqlDataAdapter da = new SqlDataAdapter(QueryORders, ConnectionString);
da.SelectCommand.Parameters.AddWithValue("@CustomerID", "QUICK");
da.TableMappings.Add("Table", "Orders");
da.TableMappings.Add("Table1", "OrderDetails");
da.Fill(ds);
```

Slika 35: Uporaba `DataAdapterja` za polnjenje podatkovnega niza

2.7.2.2. Poizvedbe na podatkovnih nizih

Na tabelah iz podatkovnih nizov lahko z LINQ-om poizvedujemo enako kot na vseh ostalih seznamih, ki implementirajo `IEnumerable<T>`. V resnici je tako, da poizvedbe ne moremo poganjati neposredno na tabeli tipa `DataTable`, ampak je potrebno uporabiti razširitveno metodo `AsEnumerable`, ki je bila dodana temu tipu. Seznam še vedno vsebuje podatke tipa `DataRow`. Če želimo dostopati do podatkov, moramo uporabljati lastnosti, ki jih ta tip ponuja (slika 36).

```
DataSet ds = LoadDataSetUsingDataAdapter();
DataTable orders = ds.Table["Orders"];
DataTable orderDetails = ds.Table["OrderDetails"];

var query =
  from o in orders.AsEnumerable();
  where o.Field<DateTime>("OrderDate").Year >= 1998
  orderby o.Field<DateTime>("OrderDate") descending
  select o;
```

Slika 36: Poizvedba na nizu podatkov

2.8. Poizvedbe na entitetah

Vse poizvedbe doslej so bile izvrševane na določeni tabeli z razmerjem ena proti ena, glede na rezultate poizvedb. Seveda se v resničnem svetu poslovnih aplikacij zahteva abstrakcija na fizični predstavitvi podatkov, da bi zagotovili neodvisnost posameznih platform.

Kadar delamo z entitetnim modelom (ang. Entity Data Model - EDM) raje pripravimo abstrakcijo podatkov na konceptualnem nivoju kot na podatkovnem. LINQ za entitete (ang. LINQ to Entities) omogoča poizvedovanje na tem nivoju abstrakcije. Entitete definira kot instance, ki so strukturirane in zgrajene iz katerega koli objektnega nivoja, vendar neodvisno od njega. Entitete so grupirane v nize z uporabo relacijskih tipov. Lahko jih definiramo ročno, lahko pa tudi uporabimo orodje, ki ga za to ponuja .NET, za generiranje entitetnega modela iz drugih podatkovnih virov.

Na takem modelu se nato poizvedbe pišejo enako kot tiste na viru podatkov iz baze.

2.9. Sklep

Spoznali smo kaj nam LINQ ponuja, katere operatorje imamo na voljo in kako se uporabljajo. Sedaj bomo prikazali še njegovo uporabo v praksi na primeru razvoja konkretne poslovne aplikacije.

3. Razvoj poslovne aplikacije

Za prikaz uporabe LINQ-a v praksi smo izbrali izdelavo poslovne aplikacije, ki omogoča vnos in izdelavo računov za manjša podjetja. Prikazali bomo razvoj celotne aplikacije, od ideje do končnega produkta.

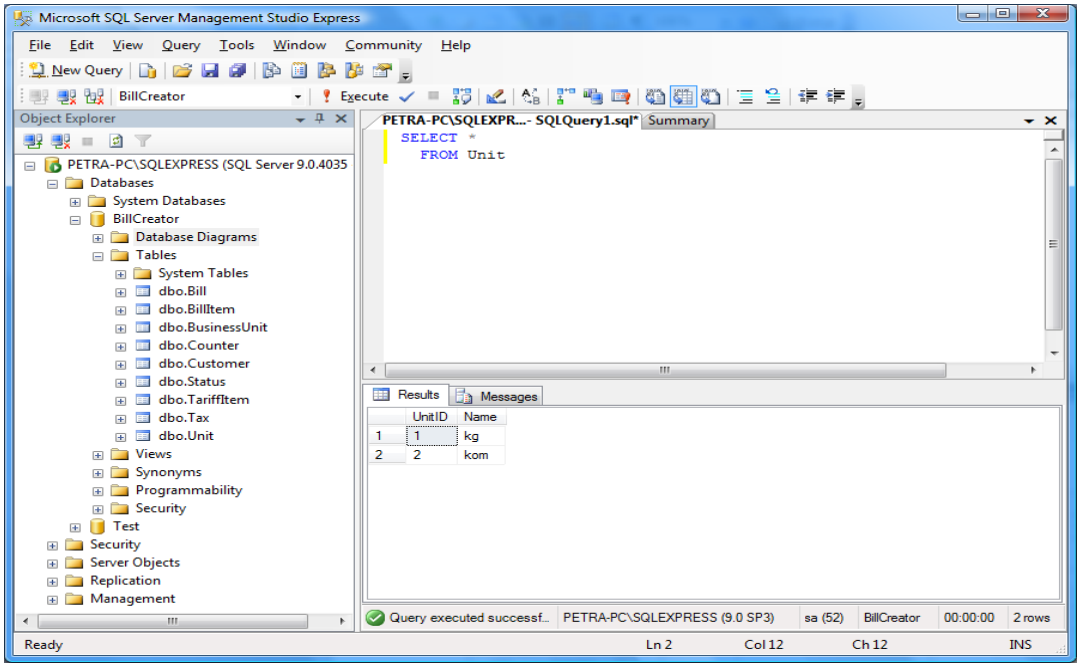
Aplikacija ni nastajala v okvirih dela, ki ga opravljam v podjetju, kjer sem zaposlen, smo pa poskušali uporabiti podobne prijeme in priporočila kot so določeni v tem podjetju. Zato smo imena objektov v podatkovnem modelu in programsko kodo v celoti napisali v angleškem jeziku, z izjemo komentarjev v programski kodi, ki so napisani v slovenščini. Prav tako smo standarde upoštevali pri oblikovanje programske kode in poimenovanju spremenljivk [9]. Po standardu je namreč zahtevana uporaba madžarske notacije (ang. Hungarian Notation) [7]. Upoštevali smo tudi standard za pisanje stavkov SQL in poimenovanje stolpcev, tabel, primarnih ter tujih ključev [9].

3.1. Uporabljena orodja in tehnologije

V nadaljevanju bomo opisali dve orodji, ki smo ju potrebovali za izdelavo aplikacije.

3.1.1. Microsoft SQL Server 2005 Express

Glede na uporabnikove zahteve o stabilnosti podatkovne strukture smo se odločili za uporabo podatkovnega strežnika Microsoft SQL Server 2005 Express, ki ga je mogoče brezplačno prenesti in uporabljati [6]. Je podatkovni strežnik, ki omogoča postavitev podatkovne baze in njeno vzdrževanje. Gre za relacijsko bazo podatkov, glavni jezik za poizvedovanje pa je Transact-SQL. Glavna funkcionalnost strežnika je zajem podatkov iz baze, pri čemer strežnik s pomočjo plana poizvedbe poskuša poiskati najhitrejši način za pridobitev želenih podatkov.

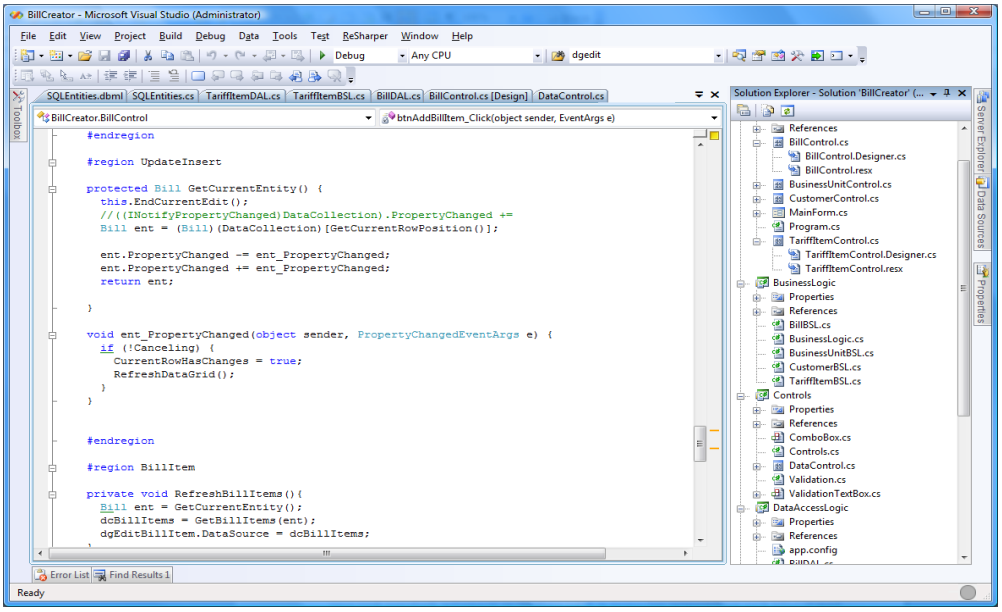


Slika 37: Poizvedbeno okno v Microsoft SQL Server Management Studio okolju

Z dodatno namestitvijo programa Microsoft SQL Server Management Studio Express, ki je še eno od Microsoftovih brezplačnih orodij, je mogoča tudi izdelava podatkovnega diagrama in generiranje baze iz njega. Ena od možnosti je tudi direktno pisanje poizvedb v poizvedbena okna, ki jih aplikacija ponuja, kar prikazuje slika 37.

3.1.2. Microsoft Visual Studio 2008

Je orodje za razvoj aplikacij v različnih tehnologijah od spletnih strani, spletnih storitev, uporabniških aplikacij in konzolnih aplikacij. Vsebuje urejevalnik kode, v katerem je vgrajen pomočnik za kodiranje (ang. IntelliSense). Poleg tega omogoča razhroščevanje in analizo programske kode. Omogoča razvoj v več programskih jezikih (Visual Basic, C++, C#, J#). Omogoča tudi grafično urejanje vnosnih mask in spletnih strani. Na sliki 38 je prikazan primer urejevalnika programske kode.



Slika 38: Urejevalnik programske kode v orodju Microsoft Visual Studio 2008

3.2. Predstavitev problema

Idejo za razvoj take aplikacije smo dobili po tem, ko je nekaj samostojnih podjetnikov in obrtnikov izrazilo željo po preprosti aplikaciji za izdelavo računov. Zaenkrat se je le eden od teh podjetnikov odločil, da bo sodeloval pri razvoju. Je pa še nekaj podjetij, ki bi aplikacijo rado preizkusilo, in se nato odločilo, če je le-ta primerna zanje.

Ti podjetniki trenutno uporabljajo za izdelavo računov svoje predloge. Običajno so to dokumenti, ki so narejeni s pomočjo programa Microsoft Word (urejevalnik besedil s podporo za oblikovanje tekstov) ali pa s programom Microsoft Excel (orodje za oblikovanje tabel). Oba sta del pisarniške zbirke Microsoft Office. Hkrati morajo podjetja voditi knjigo izdanih računov. Tukaj seveda lahko prihaja do odstopanj med podatki v knjigi računov in samimi računi. Zgodi se namreč, da se račun pozabi vpisati v knjigo, dodeli se mu napačno številko, pride do neujemanja zneskov, ali kakšne izmed ostalih možnih napak, saj nad podatki računa, ni nobene kontrole, razen seveda preverjanja tik pred pošiljanjem stranki. Naslednji problem je seznam strank, katerim se pošiljajo računi. Če so računi izdelani z zgoraj opisanima programoma, je potrebno imeti za vsako stranko svoj dokument, ali pa tvegati, da se imena in

naslovi strank napačno vpišejo na račun. Zato bi bilo smiselno omogočiti vnos imen in naslovov v neko podatkovno strukturo.

Pred izdajo računov je potrebno poiskati ceno posamezne storitve iz cenika. Velikokrat je celo tako, da imajo posamezne stranke svoj cenik, zato bi bilo potrebno vpeljati še možnost vnosa cenika za posamezno stranko.

Obstaja veliko programov, ki to že ponujajo, vendar so navadno zelo obširni in vsebujejo tudi komponente, ki so za manjša podjetja nepotrebne. Drugi problem je seveda cena, saj je večina takih programov zelo dragih, in si jih manjša podjetja ne morejo privoščiti.

3.3. Zahteve uporabnika

Programske zahteve uporabnika so, da aplikacija deluje na operacijskem sistemu Microsoft Windows. Shranjevanje podatkov mora biti stabilno in zaščiteno pred izgubo. Aplikacija naj ima sodobno strukturo in uporablja najnovejšo programsko opremo. Vsebinske zahteve uporabnika bomo poskušali razdeliti na posamezne sklope zato, da bo bolj razvidno, kako je aplikacija nastajala.

3.3.1. Poslovna enota

Vpisani morajo biti podatki poslovne enote, za katero se računi izdajajo. V glavi vsakega računa se morajo nahajati naslednji podatki:

- naziv poslovne enote,
- naslov in hišna številka,
- pošta in mesto,
- država,
- telefon in faks,
- bančni račun za nakazila,
- banka pri kateri je račun odprt in
- davčna številka.

3.3.2. Stranke

Zahteva se vnos podatkov o vseh strankah na enem mestu, za stranke mora biti omogočen vnos naslednjih podatkov:

- naziv stranke (polni naziv stranke oziroma podjetja),
- naslov in hišna številka,
- poštna številka,
- mesto,
- država,
- davčna številka in
- rok plačila (število dni, v roku katerih mora stranka plačati račun).

3.3.3. Cenik

Cenik storitev omogoča vnos cene za posamezno storitev pri posamezni stranki. Vnosna maska mora omogočati vnos naslednjih podatkov:

- naziv storitve,
- stranko, na katero se cenik storitve nanaša in
- privzeto ceno storitve (na končnem računu mora biti ceno še mogoče spreminjati).

3.3.4. Računi

Vnos podatkov o računih mora najprej omogočiti vnos osnovnih podatkov, nato pa še vnos leteh o posameznih postavkah z računa. Poleg osnovnih podatkov naj bo mogoč še vnos naslednjih podatkov:

- številka računa (zaporedna številka računa),
- izbira stranke, za katero se račun izdaja,
- kraj izdaje,
- datum izdaje, datum storitve in rok plačila, ki naj se privzeto prebere iz izbrane stranke ter roka plačila izbranega na stranki,
- vrednost računa z DDV (davek na dodano vrednost), ki se izračuna glede na vnesene postavke ter
- skupaj DDV (izračuna se iz vrednosti vnesenih postavk brez DDV).

Podatki o posameznih postavkah naj omogočajo vnos naslednjih podatkov:

- zaporedna številka postavke na posameznem računu,
- izbira postavke za izbrano stranko iz cenika,
- količino enot,
- enoto (kom, kg, ura),
- ceno na enoto,
- vrednost brez DDV (izračuna naj se iz količine in cene),
- stopnjo DDV in znesek DDV ter
- vrednost z DDV.

Statusi, v katerih se lahko nahaja račun, so:

- priprava: račun se še ureja,
- poslano: račun je že bil poslan naslovniku,
- plačano: račun je bil že poravnan in
- stornirano: račun je bil storniran zaradi napake pri vnosu.

3.3.5. Izpis računa

Seveda je potrebno omogočiti tudi izpis računa. Le-ta mora vsebovati glavo z logotipom poslovne enote, naslov pošiljatelja ter naslov prejemnika.

Poleg tega mora prikazovati še vse potrebne podatke z računa, kot so datum ter kraj izstavitve, ter datum storitve in rok plačila.

Podatki o posameznih postavkah morajo biti zapisani v tabeli, ki vsebuje podatke z zaporedno številko, vrsto blaga oziroma storitve, količino in enoto, v kateri je zapisana, ceno na enoto, vrednostjo brez DDV-ja, stopnjo DDV-ja in znesek DDV-ja ter vrednostjo z DDV-jem. Spodaj mora biti še seštevek vrednosti z DDV-jem in vrednost DDV-ja.

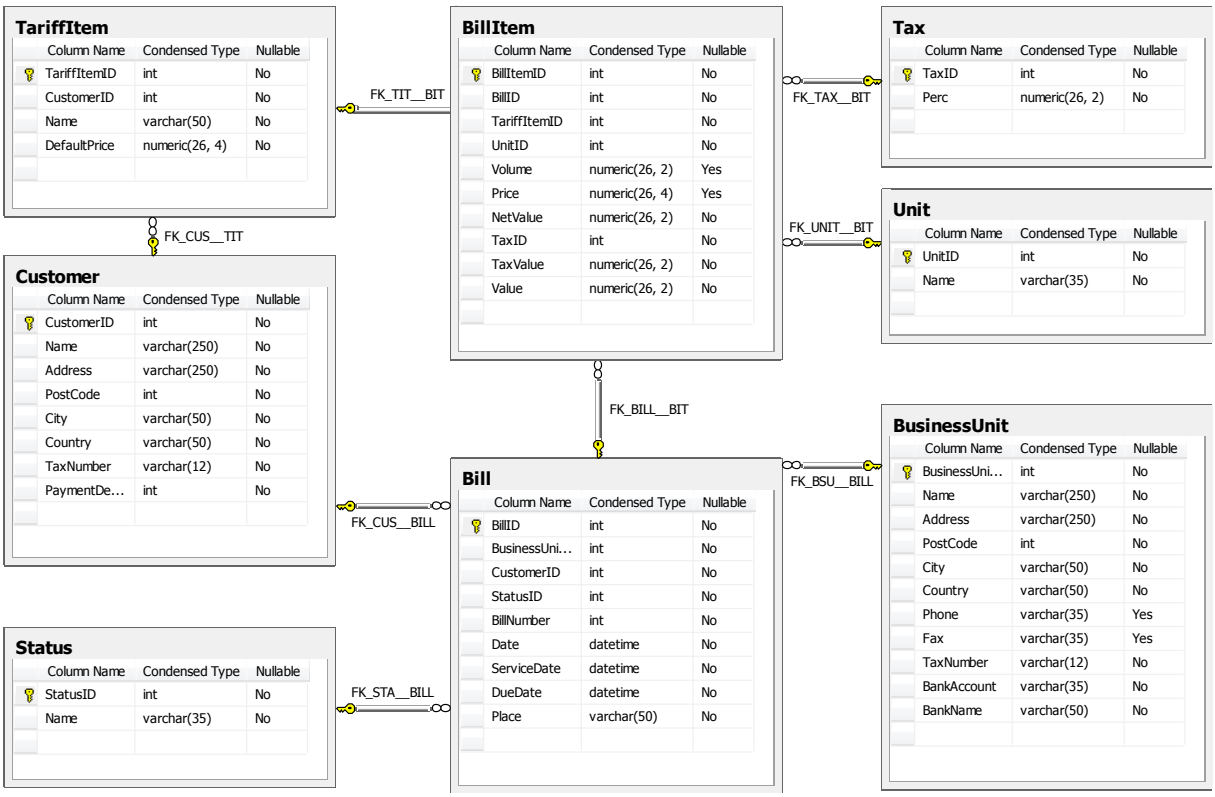
Spodaj na poročilu mora biti še tekst z besedilom o zaračunavanju zakonitih zamudnih obresti.

3.4. Predstavitev podatkovnega diagrama

S pomočjo SQL Server-ja smo najprej ustvarili nov podatkovni diagram, iz katerega smo nato ustvarili skripto za izdelavo podatkovne strukture `BillCreator`. Program omogoča vnos

vseh potrebnih informacij za izdelavo tabel in postavitev primarnih ter tujih ključev med tabelami.

Podatkovni diagram si lahko ogledate na sliki 39.



Slika 39: Podatkovni diagram aplikacije za izdelavo računov

Kot je razvidno iz slike je diagram sestavljen iz osmih tabel. Posameznih tabel ne bomo opisovali, saj so zelo podobne samim zahtevam uporabnika. Razlog za to je večkratno preverjanje zahtev uporabnika in dogovarjanje z njim, kaj dejansko želi ter kaj so glavni razlogi za posamezne podatke. Bazo smo normalizirali z vpeljavo nekaj tabel, ki predstavljajo tuji ključ v glavnih tabelah (`Tax`, `Status`, `Unit`).

3.5. Prikaz strukture projekta

Glede na programska priporočila (ang. Programing best practices) smo se odločili za izdelavo tri nivojske arhitekture [10].

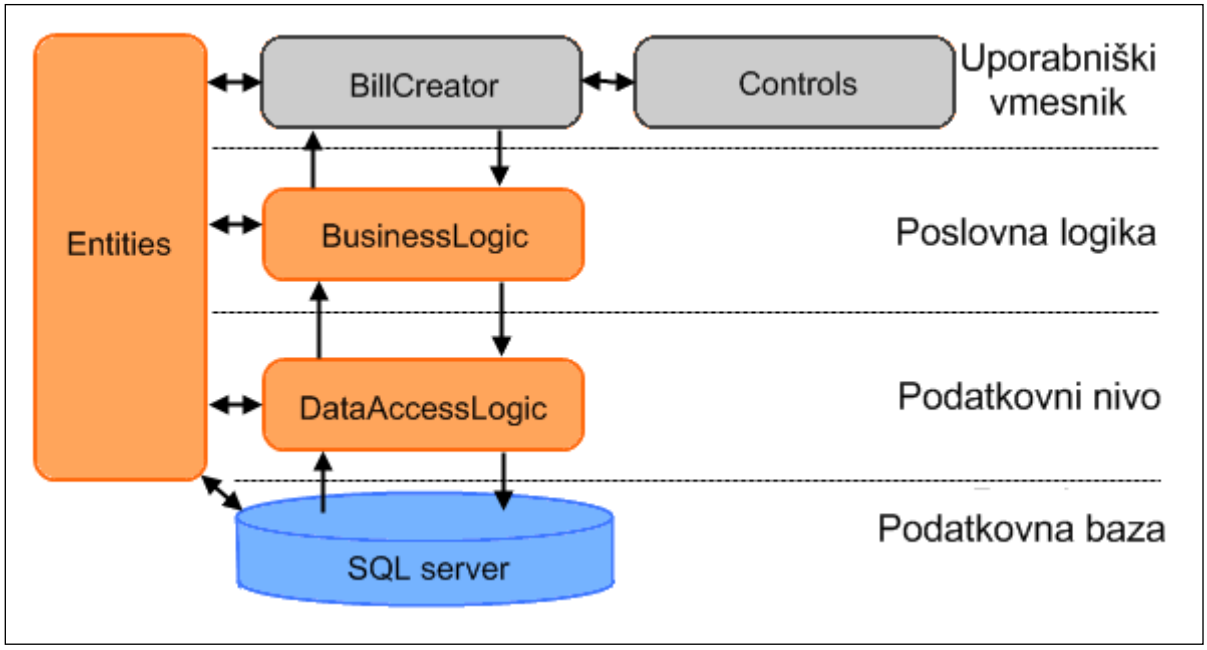
Najnižji je podatkovni nivo. Za to skrbi projekt `DataAccessLogic`. Njegova naloga je branje podatkov iz podatkovne baze in pisanje le-teh vanjo. Na tem nivoju naj se vsebinski izračuni ne bi delali, in vse mora biti resnično prirejeno samo za povezavo na podatkovno bazo. V bistvu se zaradi uporabe LINQ za SQL to zadnje nekoliko spremeni. Z generiranjem entitet iz baze ter s servisi, ki LINQ-u pomagajo pri usklajevanju podatkov med entitetami in bazo se namreč zgodi, da preko entitet že dostopamo neposredno na bazo. Podatkovni nivo smo vseeno obdržali, in tu delamo vse poizvedbe na podatkih, kot da bi ti bili v podatkovni bazi.

Drugi nivo je vmesni nivo, ki predstavlja poslovno logiko. To je projekt `BusinessLogic`, ki skrbi za vsebinske preračune ter prenos podatkov klienta v bazo ter iz baze do klienta. Ta nivo ne sme nikoli dostopati direktno na podatkovno bazo.

Tretji in zadnji nivo je predstavitveni nivo, v katerem se nahajajo vnosne maske, poročila in kontrole vnosa podatkov. Ta nivo prestavljata projekta `BillCreator` in `Controls`.

Edini preostali projekt je `Entities`, ki predstavlja podatke iz baze. Večina entitet v tem projektu je bilo s pomočjo LINQ-a generiranih direktno iz baze. Po potrebi se v ta projekt dodajajo tudi ostale podatkovne strukture. Do tega projekta namreč dostopajo vsi nivoji aplikacije: podatkovni za vračanje podatkov iz baze, vmesni za obdelavo podatkov in predstavitveni za prikaz teh podatkov. Hkrati ima ta projekt tudi dostop do baze, saj LINQ s svojimi servisi skrbi za sinhroniziranje podatkov med temi entitetami in podatki v bazi.

Slika 40 prikazuje trinivojsko arhitekturo aplikacije.



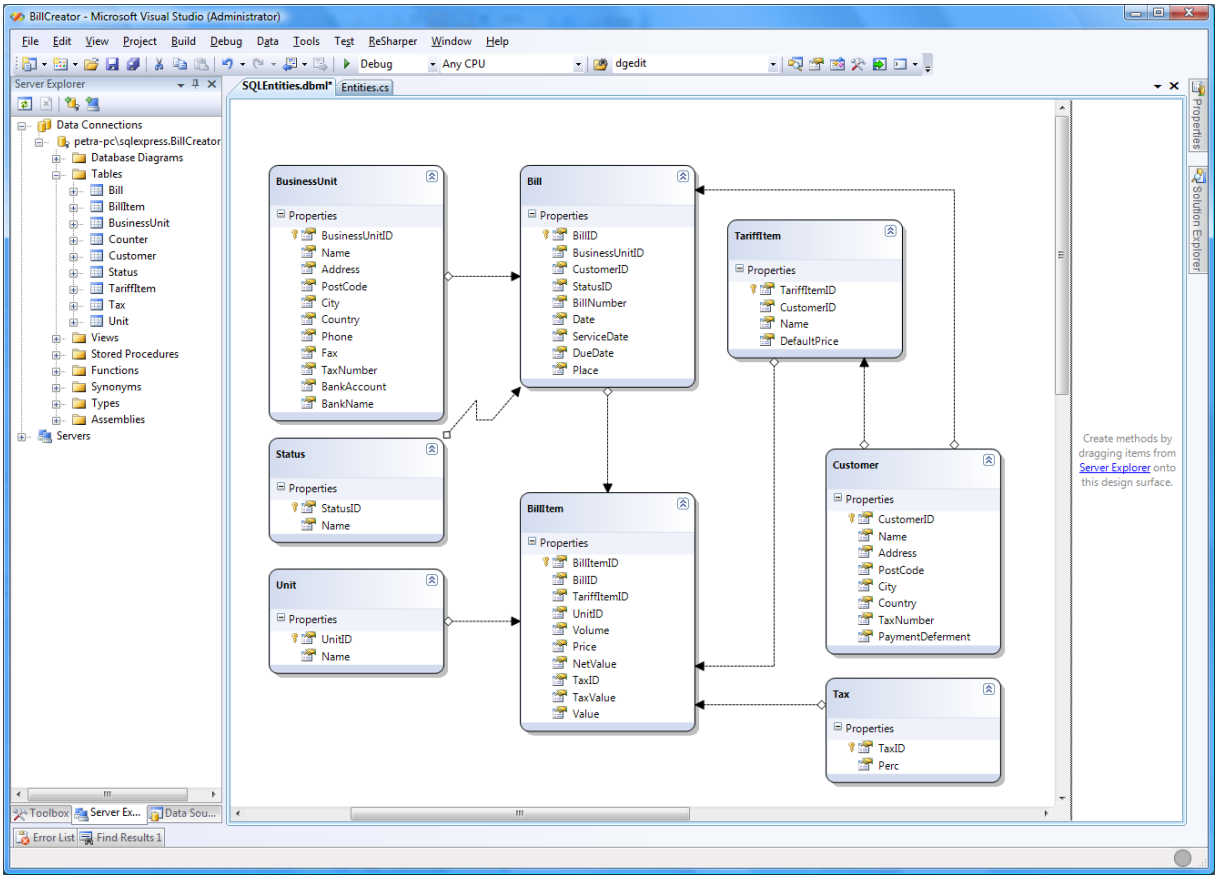
Slika 40: Trinivojska arhitektura aplikacije

3.6. Izdelava entitet iz baze

Namesto standardne uporabe ADO.NET vmesnika za dostop do baze, smo se odločili za uporabo nove tehnologije, ki jo ponuja Microsoft v njihovem .NET ogrodju. Glede dosedanjih izkušenj z verzijo 1.1. smo potrebovali kar nekaj časa, da smo spoznali, kaj vse verzija 3.5 ponuja. Ena od teh stvari je tudi generiranje entitet neposredno iz baze s pomočjo LINQ za SQL grafičnega vmesnika.

V ta namen najprej ustvarimo novo povezavo na podatkovno bazo. Vse kar moramo nato narediti je, da v želenem projektu izberemo »Dodaj nov« (ang. Add new) in tip `LINQ to SQL Classes`. Ustvari se nam razred s končnico `dbml` (ang. database markup language). To je grafični vmesnik, s pomočjo katerega nato s tehniko povleci in spusti dodajamo tabele.

Ko tabelo dodamo, se nam dodajo razredi posameznih tabel z vsemi relacijami iz podatkovne strukture. Ko zaključimo postopek dodajanja tabel, imamo objektni relacijski model. Primer je prikazan na sliki 41.



Slika 41: Primer objektnega relacijskega modela

3.7. Podatkovni nivo

Ko imamo podatke v aplikaciji, je dostop do podatkov v entitetah enak dostopu do podatkovne baze, zato lahko začnemo implementirati podatkovni nivo.

Na tem mestu ne bomo prikazali vseh tipov poizvedb in spreminjanj ter dodajanj podatkov, temveč zgolj nekaj primerov. Opozoriti moramo še na to, da seveda vrstni red razvoja le ni povsem enak predstavljenemu, saj smo sproti razvijali tudi posamezne vnosne maske in po potrebi dopolnjevali vse nivoje aplikacije.

3.7.1. Prikaz poizvedb na entitetah iz baze

Glede na potrebe vnosne maske za vnos strank smo potrebovali dve poizvedbi. Eno, ki vrne vse stranke, in drugo, ki vrne samo posamezno stranko. Za začetek si poglejmo naslednja dva primera, ki sta prikazana na sliki 42.

```
/// <summary>
/// Pridobi seznam vseh strank v obliki IQueryable
/// </summary>
/// <returns></returns>
public IQueryable<Customer> GetCustomers(){
  Table<Customer> Customers = db.GetTable<Customer>();
  return from c in Customers
         select c;
}

/// <summary>
/// Pridobi seznam vseh strank v obliki IBindingList
/// </summary>
/// <returns></returns>
public IBindingList GetCustomerUnfiltered() {
  IQueryable<Customer> query = GetCustomers();
  return new BindingList<Customer> (query.ToList());
}

/// <summary>
/// Pridobi seznam ene stranke v obliki IBindingList
/// </summary>
/// <param name="_iCustomerID">ID stranke</param>
/// <returns></returns>
public IBindingList GetCustomerFiltered(int _iCustomerID) {
  var qCustomers = GetCustomers();
  if (_iCustomerID != 0){
    qCustomers = from c in qCustomers
                 where c.CustomerID == _iCustomerID
                 select c;
  }
  return new BindingList<Customer> (qCustomers.ToList());
}
```

Slika 42: Poizvedbe za pridobitev podatkov o strankah

S prvo poizvedbo najprej preberemo vse stranke iz entitet. Kot je videti iz primera jih vrnemo kot seznam tipa `IQueryable`, na katerem so možne nadaljnje poizvedbe. To metodo uporabljata ostali dve metodi kot seznam podatkov o vseh strankah. Uporablja se tudi kot metoda za prikaz strank v izbirniku na računu. Kar zadeva vnosne maske za stranke, bi lahko ostali dve poizvedbi pridobili podatke neposredno iz entitete.

Druga poizvedba prebere podatke s pomočjo prve poizvedbe, nato pa ta seznam pretvori v seznam tipa `IBindingList`. Da se to zgodi obstajata dva razloga. Prvi je ta, da potrebujemo na vnosni maski tip `BindingList` zato, da lahko posamezne kontrole povežemo (ang. binding) na podatke, kar omogoča že samo .NET ogrodje. Drugi razlog je ta, da pretvorbo v navaden seznam (`query.ToList()`) povzročimo, da podatki v seznamu niso več neposredno povezani (ang. reference) na entitete, saj ne želimo, da uporabnikova sprememba podatka povzroči spremembo tudi v bazi. Spremembe in dodajanja podatkov v bazo bi namreč radi izvajali samo, kadar se uporabnik odloči, da želi podatke shraniti.

Tretja poizvedba je enaka drugi, le da je stranka omejena po ključu, in je zato v seznamu samo ena. Še vedno potrebujemo seznam zato, da lahko povežemo nanj kontrole na vnosni maski. Tudi ta poizvedba se uporablja na ostalih vnosnih maskah.

3.7.2. Dodajanje in spreminjanje podatkov

Tukaj smo najprej naleteli na pred tem neznano težavo, ko je med izvajanjem programa le-ta neprestano sporočal, da podatek, ki ga spreminjamo ali dodajamo ne more biti dodan, ker se ne nahaja v tem podatkovnem kontekstu (ang. data context). Kar precej časa smo porabili, da smo dojeli, da morajo biti vse poizvedbe na entitetah pognane na enem samem podatkovnem kontekstu, kar smo rešili s tem, da smo naredili prednika vsem razredom na podatkovnem nivoju, in v njem ustvaril statičen kontekst, preko katerega nato vsi berejo podatke.

```
public static SQLEntitiesDataContext db = new
                                SQLEntitiesDataContext(ConnectionString);
```

Odslej vse poizvedbe za dostop do podatkov uporabljajo ta kontekst. To je razvidno tudi že na sliki 42.

Ko je bilo to urejeno je delo lahko normalno potekalo naprej. Poglejmo si primera, kako spreminjamo in dodajamo stranko v podatkovno bazo (slika 43).

```
/// <summary>
/// Shrani spremembe na stranki
/// </summary>
/// <param name="_entCustomer"></param>
/// <returns></returns>
public bool SaveChanges(Customer _entCustomer) {
  try{
    db.Customers.Attach(_entCustomer);
    db.SubmitChanges();
    return true;
  } catch{
    return false;
  }
}

/// <summary>
/// Doda novo stranko
/// </summary>
/// <param name="_entCustomer"></param>
public void AddNew(Customer _entCustomer){
  db.Customers.InsertOnSubmit(_entCustomer);
  db.SubmitChanges();
}
```

Slika 43: Dodajanje in spreminjanje strank

Iz zgornjih primerov je razvidno, kako LINQ za SQL poenostavi delo s podatkovno bazo. Prvi primer je spreminjanje podatka. Ker LINQ ve, kateri podatek imamo v trenutni entiteti, ki jo spreminjamo (pozneje bomo, ob prikazu delovanja vnosnih mask razložili kako dobimo entiteto, ki je povezana z entitetami iz baze, saj se namreč pri poizvedbi načrtno povezava prekine), enostavno v bazo shrani spremembe s klicem metode `SubmitChanges`.

Podobno deluje tudi dodajanje v bazo, saj je vse, kar je potrebno narediti to, da kličemo metodo `InsertOnSubmit` in ji kot parameter dodamo novo entiteto za stranko. Seveda v tem primeru nova entiteta še ne sme obstajati v podatkovni bazi, LINQ-ovi servisi namreč skrbijo za pravilno podeljevanje identitet podatkom, in v primeru obstoja entitete, bi servis vnos preprečil z napako.

3.8. Vmesni nivo − poslovna logika

Na tem nivoju nismo imeli posebnih primerov logike. Večina metod je bila samo vmesni klic med predstavitvenim nivojem in podatkovnim nivojem, zato tega na tem mestu ne bomo posebej opisovali.

3.9. Predstavitveni nivo

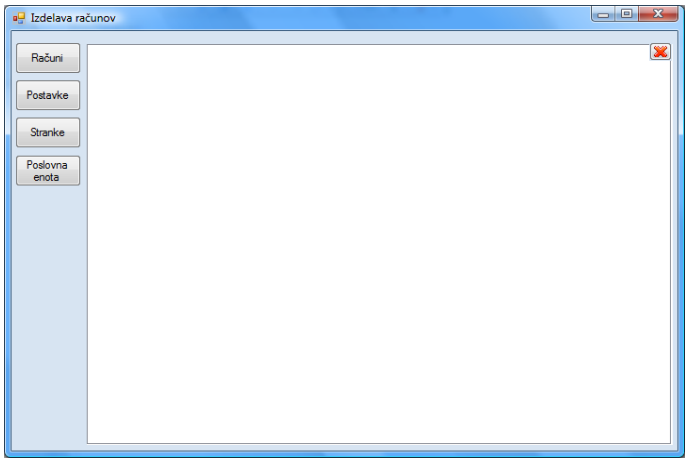
Uporabili smo razvoj vnosnih mask z dedovanjem (ang. form inheritance). Princip govori o tem, da se za posamezen sklop vnosnih mask naredi po enega ali več prednikov, ki naredijo večino potrebnih opravil, hkrati pa sprožijo potrebne dogodke (ang. event) na naslednikih. Naslednik nato ob dogodku naredi del svojega opravila in vrne nek rezultat preko argumentov, ki so parametri dogodka. Nato lahko prednik nadaljuje s svojim delom do naslednjega dogodka.

Pri razvoju aplikacije smo uporabili dedovanje kontrol, saj se le-te dinamično prikazujejo na osnovni (ang. main) maski, kar bom podrobneje opisal v nadaljevanju.

3.9.1. Osnovna maska

Osnovna maska skrbi za prikazovanje in skrivanje zavihkov za vnos posameznih podatkov. V bistvu ponuja gumbe za odpiranje vnosnih mask za vnos podatkov o računih, postavkah, strankah in poslovni enoti. Druga stvar, za katero skrbi ta maska je, da se ob zapiranju posameznih vnosnih mask in pri zapiranju celotne aplikacije preverja, če so podatki na vnosnih maskah shranjeni, saj v nasprotnem primeru aplikacija to javi in zahteva, da naj uporabnik podatke shrani ali spremembe zavrže.

Ta maska skrbi tudi za to, da se ob pritisku na kombinacije tipk za bližnjice sprožijo pravilni dogodki na trenutno prikazani vnosni maski. Tako se morajo recimo pri pritisku na tipko F5 osvežiti podatki na prikazani vnosni maski, seveda samo ob predpostavki, da se trenutni zapis na tej maski ni spreminjal oziroma, da so spremembe že shranjene. Na sliki 44 je prikazan izgled osnovne maske s pripadajočimi gumbi.



Slika 44: Izgled osnovne maske

3.9.2. Preverjanje vnosa podatkov

Kot se je naknadno izkazalo, je bilo potrebno uvesti tudi kontrolo nad vnesenimi podatki pri shranjevanju. Ker nismo želeli na vsaki maski posebej preverjati pravilnost vseh postavk, ali je podatek sploh vnesen, in ali je pravega tipa, smo si pripravili naslednike nekaterih osnovnih kontrol, ki jih ponuja ogrodje .NET. Tako smo na primer naredili naslednika kontrole `TextBox`, ki smo ga poimenovali `ValidationTextBox`. Kontroli smo dodali lastnost, ki pove, ali je vnos podatka zahtevan, in kakšnega tipa podateka.

Druga izmed kontrol, ki smo jim naredili naslednika, je bil izbirnik (ang. `ComboBox`). Tudi pri tej kontroli smo dodali metode za preverjanje pravilnosti vnosa podatkov.

3.9.3. Prednik vnosnih mask

Nato smo začeli z izgradnjo glavnega elementa vnosnih mask (ang. `DataControl`), torej prednika vseh nadaljnjih mask, pri čemer naj bi ta prednik poskrbel za vse potrebne dogodke, nasledniki pa bi nato implementirali samo za njih specifične stvari. Ne bomo se spuščali v podrobnosti opisa kaj vse prednik dela, našteli bomo le nekaj stvari, za katere poskrbi.

Kontrola je razdeljena na dva dela. Zgornji del vsebuje prikazovalnik podatkov iz razreda `DataGridView`, ki omogoča prikaz podatkov iz različnih tipov izvora podatkov. V tem prikazovalniku bomo vedno prikazovali glavne podatke iz podatkovne baze, seveda s podatki, za katere bo končni naslednik narejen. Zato dejanski podatkovni vir in podatke v stolpcih nastavimo šele na nasledniku, podrobneje bo to prikazano v nadaljevanju.

Ob premikanju po prikazovalniku se izbira trenutno izbrani zapis, na katerega se nanašajo podatki spodaj (opisano v nadaljevanju). Spodnji del je vnosni del. Kontrola tu poskrbi povezovanje (ang. binding) podatkov v kontrolah na podatke, ki so trenutno izbrani v zgornjem prikazovalniku. Ta del tudi skrbi, da se ob shranjevanju naredi pregled pravilnosti vnosa.

Seveda pa ta kontrola skrbi še za veliko drugega. Ena od teh zahtev je tudi to, da se ob premikanju po zgornjem seznamu podatkov vedno tudi preverja, če podatki niso spremenjeni, saj se v tem primeru premik ne sme zgoditi, temveč mora aplikacija vprašati, če želi uporabnik shraniti spremenjene podatke.

Poleg tega kontrola ponuja v srednjem delu še orodno vrstico, kjer so podprte operacije (gumbi za izbiro) za dodajanje novega zapisa, shranjevanje obstoječega, osvežitev podatkov s svežimi podatki iz baze ter tiskanje. Te gumbe je pri naslednikih mogoče skrivati, onemogočiti, ali tudi dodati nove. Na desni strani imamo na voljo še dva večja gumba za shranjevanje in preklic sprememb. Razpored gradnikov je prikazan na sliki 45.

Slika 45: Razporeditev gradnikov na predniku vnosnih mask

Nekaj primerov proženja dogodkov na naslednikih:

- Klik na gumb »`Osveži`«: Najprej preverimo, če gre za spremenjene podatke ali dodajanje novega zapisa. Če gre za to, aplikacija najprej zahteva potrditev uporabnika, da želi podatke shraniti oziroma preklicati spremembe. Če uporabnik potrdi shranjevanje, najprej pokličemo dogodek za shranjevanje ali preklic na nasledniku. Šele nato lahko naredimo osvežitev podatkov.
- Klik na gumb »`Dodaj`«: najprej preverimo ali je vse shranjeno, šele nato lahko sprožimo dogodek na nasledniku, ki nastavi privzete vrednosti na gradnike.
- Klik na gumb »`Shrani`«: najprej se sproži preverjanje vnesenih podatkov na nasledniku; če je le to uspešno, se podatki shranijo. Ko so podatki shranjeni se ponovno poveže gradnike iz naslednika na podatkovno strukturo. Na koncu se naredi še osvežitev podatkov na vnosni maski s podatki iz baze.

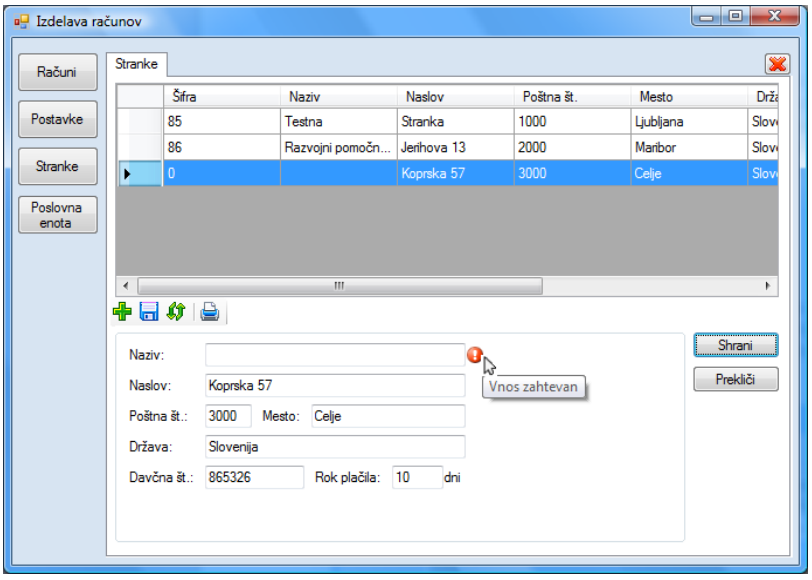
3.10. Vnosne maske

Pri vnosnih maskah smo poskušali slediti enostavnemu vnosu in preprečevanju vnosa napačnih podatkov. Seveda je to le deloma mogoče, saj vsebinsko preverjanje razen nekaj manjših izjem ni mogoče. Vse maske so narejene na podoben način, kot določa že prednik vnosnih mask. Pri vsaki vnosni maski bomo tudi nekoliko opisali možne izboljšave glede na pripombe uporabnikov.

3.11. Stranke

Stranke so poslovni partnerji, za katere podjetje izdeluje račune. Pri vnosu podatkov o strankah je potrebno zajeti podatke strank, ki so potrebni na izpisu računa. Ti podatki so naziv (vpišemo polni naziv podjetja za katerega se račun izdeluje), naslov (vpišemo polni naslov z ulico in hišno številko), poštna številka (štirimestna številka pošte v Sloveniji), mesto poštne številke, država naslova, davčna številka in rok plačila. Rok plačila je število dni, ki predstavljajo zamik, do katerega je posamezna stranka upravičena. Zahtevan je vnos vseh podatkov, saj so vsi potrebni za pravilno izdelavo računa. Za to je na sliki 46 prikazano opozorilo, ki nakazuje, da nek podatek ni vpisan. Aplikacija v tem primeru zapisa ne pusti

shraniti in zahteva vnos podatkov. S tem uporabnika prisilimo, da vnese vse podatke, ki so potrebni za izdelavo posameznega računa.

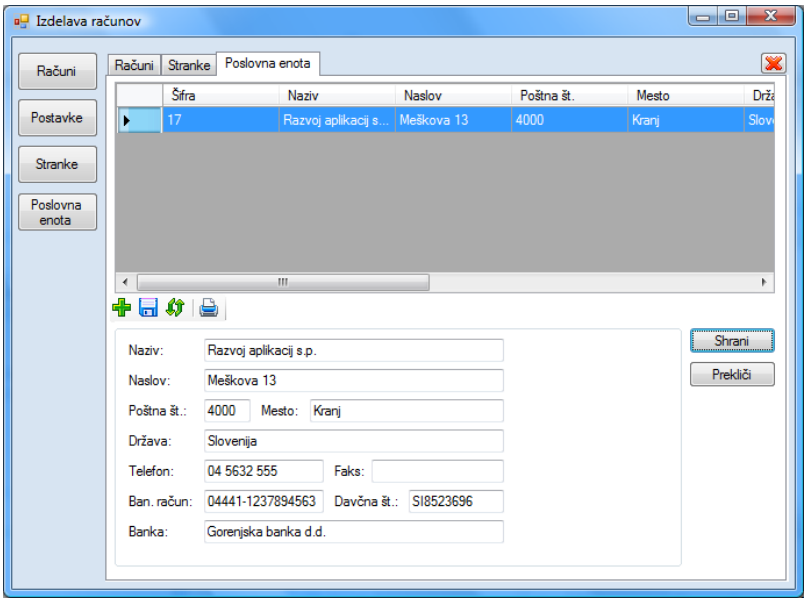


Slika 46: Preverjanje vnosa podatkov pri dodajanju strank

V naslednji nadgradnji aplikacije se že pripravljajo spremembe vnosne maske. Zaželeno je namreč, da bi se mesto in država samodejno izpolnila glede na izbrano poštno številko, kar v trenutni verziji še ni podprto. Prav tako bi bilo dobro omogočiti še vnos nekaterih drugih podatkov, kot so telefonska številka, faks in mogoče tudi vnos opomb za posamezno stranko.

3.12. Poslovna enota

Naslednja vnosna maska je maska za vnos podatkov o poslovni enoti. Poslovna enota je enota, pri kateri se izdajajo računi. Vnos podatkov je podoben kot pri strankah, le da nekateri podatki niso zahtevani (slika 47).

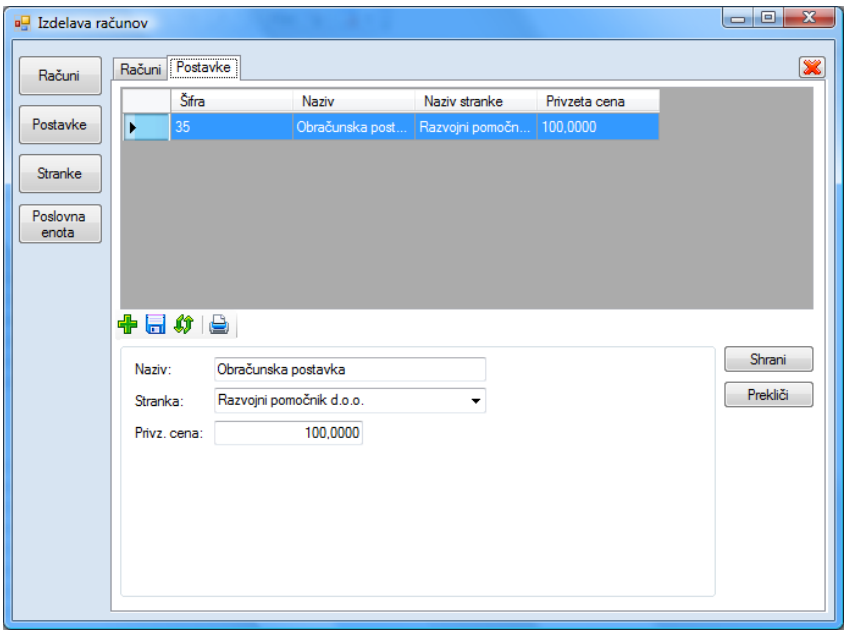


Slika 47: Vnosna maske za vnos podatkov o poslovni enoti

3.13. Ceniki

Zasnova cenika je taka, da za posamezno stranko lahko vnesemo več različnih postavk. Posamezni postavki se najprej določi ime, nato izbere, za katero stranko je postavka, na koncu pa je potrebno še vpisati privzeto ceno. To ceno je na računu še mogoče spreminjati, saj predstavlja samo osnovo, ki se nato glede na posamezno storitev še naknadno spreminja (možni popusti, dodatna zahtevnost dela, itd.).

Na tej maski je uporabljen tudi izbirnik strank, kjer si uporabnik iz seznama vseh strank vpisanih v aplikaciji izbere enega, ta pa se nato kot tuji ključ vnese v podatkovno bazo. Primer vnosa postavke cenika za stranko si lahko pogledamo na sliki 48.



Slika 48: Vnos postavke za cenik stranke

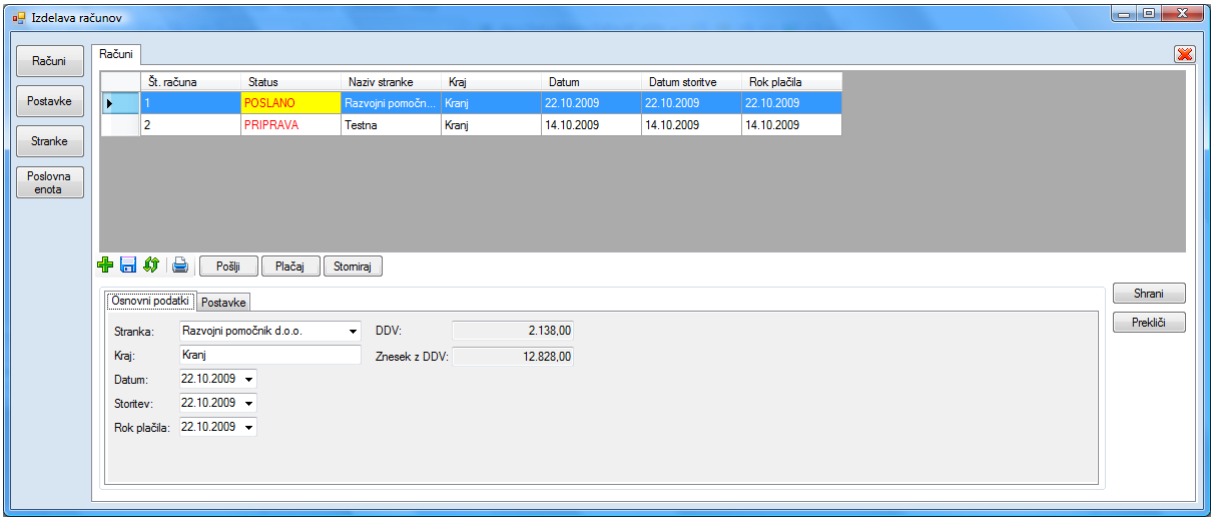
3.14. Računi

Pri računih smo vnosno masko razdelili na dva zavihka (ang. tab). Prva stran omogoča vnos osnovnih podatkov kot so stranka, kraj izdaje, datum izdaje, datum storitve in rok plačila. Rok plačila se izračuna glede na zamik plačila kot je označeno na stranki. DDV in znesek z DDV-jem se izračunata glede na podatke, ki so vneseni na zavihku s postavkami. DDV predstavlja seštevek davka na vseh postavkah, znesek z DDV-jem pa predstavlja seštevek vseh končnih zneskov na posameznih računih. Teh dveh podatkov ni mogoče spreminjati.

V zgornji seznam smo dodali tudi polje s podatkom o statusu. Status je mogoče spreminjati z gumbi, ki se nahajajo pod zgornjim seznamom. Ob dodajanju računa dobi le-ta najprej status »priprava«. Ko je račun natisnjen in pripravljen na pošiljanje, se ga odpošlje, in takrat se status nastavi na »poslano«. V trenutku, ko je iz nakazil razvidno, da je bil račun plačan, se ga lahko prestavi v status »plačano«. Podatke o računih je mogoče spreminjati le, če je račun v statusu »priprava«.

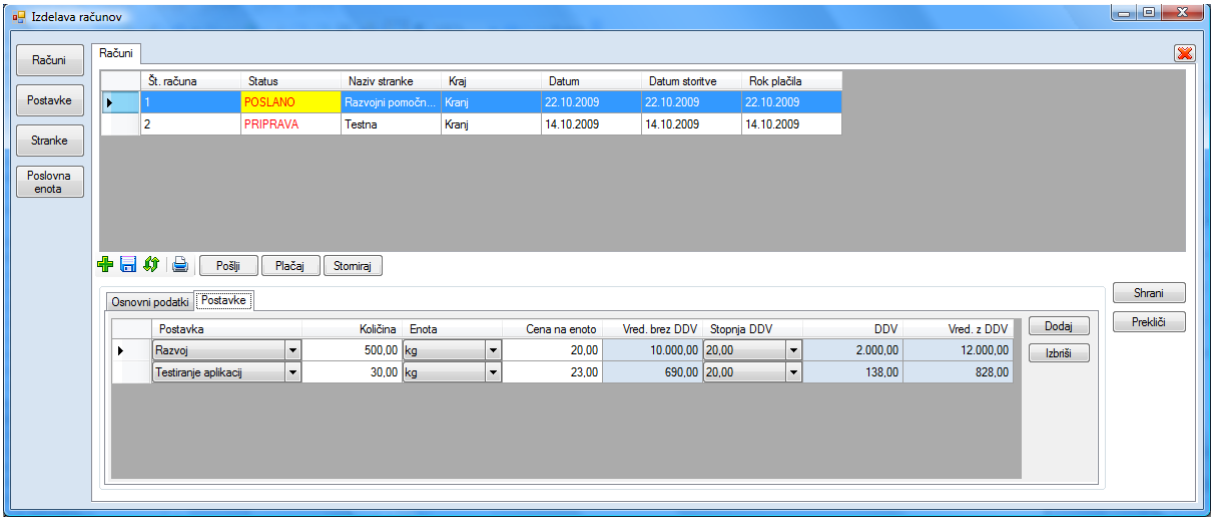
Zadnji status, ki je na voljo, je status »stornirano«, kar pomeni, da je pri izdelavi računa prišlo do napake in le-ta ni primeren za oddajo. Druga možnost za razveljavitev računa je ta, da stranka iz kateregakoli razloga to zahteva.

Na sliki 49 je prikazana vnosna maska za vnos osnovnih podatkov o računih.



Slika 49: Vnos osnovnih podatkov o računu

Drugi zavihek na tej maski omogoča vnos podatkov o posameznih postavkah iz računa (slika 50). Ker je teh lahko več, je to v bistvu podseznam na glavnem seznamu z računi. Najprej moramo izbrati postavko iz cenika za stranko, ki je izbrana na računu. Ob tem se nam samodejno nastavi privzeta cena s postavke, ki jo je naknadno še mogoče spreminjati. Naslednji podatek, ki ga moramo vnesti je podatek o količini storitve. Ko je tudi ta vnesen, se glede na stopnjo DDV, ki je privzeto že nastavljena na 20%, izračunajo podatki iz stolpcev vrednost brez DDV-ja, DDV in vrednost z DDV-jem. Posamezne postavke je mogoče dodajati z gumbom »`dodaj`«. Z gumbom »`izbriši`« pa je mogoče postavke odstranjevati.



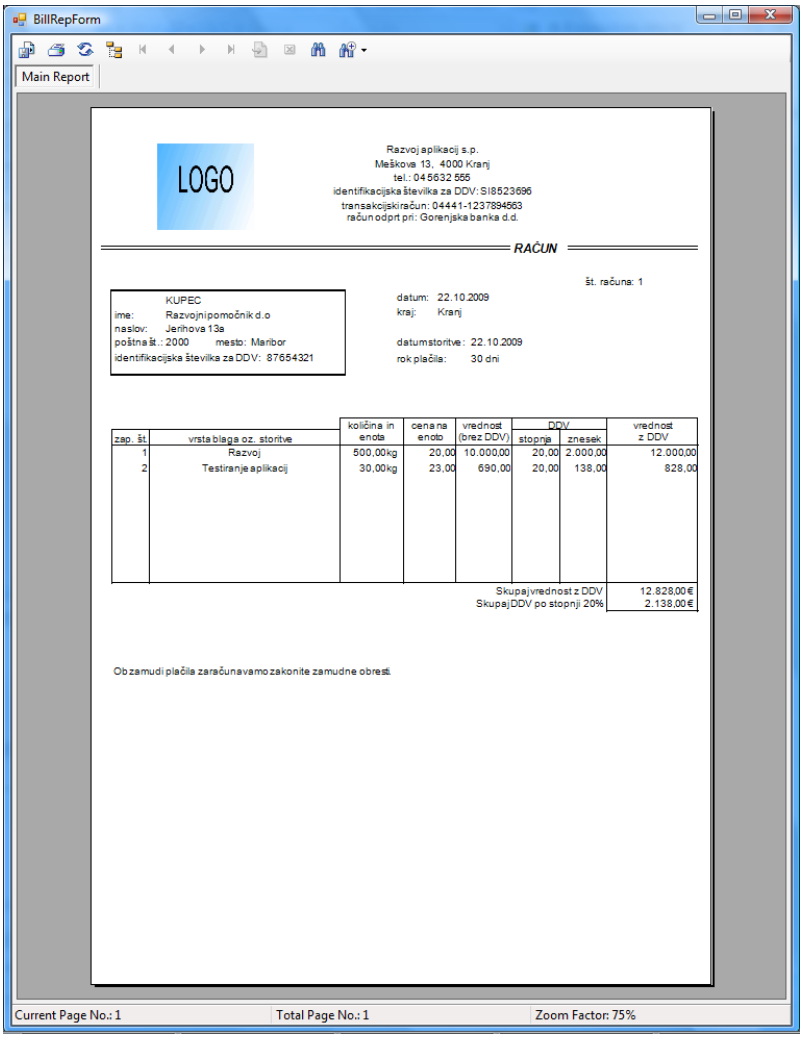
Slika 50: Vnos posameznih postavk na računu

3.15. Izpis računa

Najpomembnejši in hkrati najmanj zahteven del razvoja te aplikacije je bila izdelava izpisa računa. V ta namen smo ustvarili novo prikazovalno masko. Za prikaz izpisa smo uporabili orodje Crystal Reports [8], ki je vključeno v razvojno orodje Visual Studio 2008. To orodje omogoča, da z uporabo njihovega pregledovalnika prikažemo izpis na zaslonu, potrebno je le pripraviti podatke in mu jih dodeliti. Razvoj poteka tako, da na izpis dodajamo posamezna polja s pomočjo tehnike povleci in spusti (ang. drag and drop). Rišemo lahko črte, dodajamo slike in poljubno urejamo tekst. Dodajamo lahko tudi formule za preračun in urejanje podatkov ter delamo vmesne ter končne vsote po tabelah. Prikaz izpisa je predstavljen na sliki 51.

Izpis je pripravljen za končnega uporabnika in se lahko spreminja glede na njegove želje. Želja je sicer to, da bi bilo mogoče izpis spreminjati brez sodelovanja razvijalca in med delovanjem samega programa, kar bo verjetno tudi vključeno v eno izmed naslednjih verzij aplikacije.

Sam izpis mora prikazovati vse potrebne podatke po zahtevanih računovodskih standardih. Ostale zahteve, kot je oblika izpisa in postavitev posameznih podatkov, se lahko spreminjajo glede na želje končnega uporabnika.



Slika 51: Izpis računa

4. Sklepne ugotovitve

Razvoj aplikacij s pomočjo LINQ-a je na podatkovnem nivoju zares enostaven. LINQ-ovi servisi dobro skrbijo za sinhronizacijo podatkov in le-ta nam ni predstavljala posebnih težav. Nekaj jih je bilo le na začetku, dokler nismo poskrbeli za to, da so se poizvedbe izvrševale znotraj istega podatkovnega konteksta. Uporabnost na nivoju zajema podatkov nas je popolnoma prevzela, saj je bilo pisanje poizvedb zelo podobno pisanju SQL stavkov kot smo jih vajeni. Tudi preverjanje poizvedb v času pisanja programske kode se je izkazalo kot zelo uporabno. Glede na osnovni vmesnik za delo s podatki, ki ga ponuja .NET ogrodje, je razvoj s pomočjo LINQ-a veliko hitrejši in enostavnejši, kar omogoča prihranek na času razvoja aplikacije, in hkrati možnost uporabe različnih virov podatkov brez bistvenih sprememb programske kode.

Najzahtevnejši dela razvoja aplikacije pa je bila izdelava prednika vnosnih mask. Tukaj smo imeli največ težav s povezavo vnosnih gradnikov na podatke, ki jih preko poizvedb vrača LINQ, saj so se tukaj pokazale določene posebnosti in zato je bil pristop povezovanja gradnikov na podatke pridobljene z LINQ-om drugačen, kot ob uporabi podatkov iz podatkovnih nizov, ki smo jih vajeni ob uporabi osnovnega vmesnika za dostop do podatkov.

Med razvojem smo prišli do spoznanja, da je LINQ zelo uporaben tudi na ostalih nivojih poslovnih aplikacij, in ne samo na podatkovnem nivoju kot smo bili prepričani pred razvojem. In tukaj se resnično lahko pokaže dejanska vrednost te tehnologije, saj lahko enak pristop kot za delo na podatkovnih virih uporabimo tudi na vseh ostalih seznamih znotraj poslovne logike in celo uporabniškega vmesnika.

Ob izdelavi tega diplomskega dela smo se začeli bolj zanimati za to tehnologijo, glede na uporabno vrednost razvoja z LINQ-om, tudi v podjetju, kjer sem zaposlen. Začeli smo preučevati možnost uporabe v obstoječih kot tudi nastajajočih projektih, saj LINQ resnično poenostavlja dostop do podatkov.

Aplikacija, ki je bila prikazana kot primer, se trenutno nahaja v testnem okolju pri eni izmed strank, ki je sodelovala ob razvoju. Predvidena je uporaba v delovnem okolju, po odpravi napak in dopolnitvah, ki se bodo izkazale kot potrebne med testiranjem.

Med razvojem in ob testiranju uporabnika se je pojavilo nekaj zamisli, kako aplikacijo izboljšati oziroma nadgraditi. Na vnosne maske bi bilo dobro vpeljati še možnost iskanja po podatkih z nastavljanjem filtrov. Pri posameznih podatkovnih strukturah bo potrebno dodati še dodatne podatke kot so telefon pri strankah, elektronska pošta in drugo. Namen naslednje verzije programa je te zamisli vgraditi v aplikacijo in razširiti funkcionalnosti še na izdelavo naročilnic in dobavnic.

Viri

[1] Wikipedia - .NET Framework. Dostopno na: http://en.wikipedia.org/wiki/.NET_Framework

[2] Wikipedia - Language Integrated Query. Dostopno na: http://en.wikipedia.org/wiki/Language_Integrated_Query

[3] Fabrice Marquerie, Steve Eichert, Jim Wooley, LINQ in Action. Manning Publications Co., 2008

[4] Paolo Pialorsi, Marco Russo, Introducing Microsoft LINQ, Microsoft Press, 2007

[5] Monitor, Kaj je novega v C# 3.0. Dostopno na: http://www.monitor.si/clanek/kaj-je-novega-v-c-3-0/

[6] Microsoft, Microsoft SQL Server 2005 Express. Dostopno na: http://www.microsoft.com/Sqlserver/2005/en/us/express.aspx

[7] Wikipedia- Hungarian notation. Dostopno na: http://en.wikipedia.org/wiki/Hungarian_notation

[8] Crystal Reports. Dostopno na: http://www.crystalreports.com/

[9] Evolve d.o.o., Standardi za razvoj aplikacij, 2005

[10] Simple-talk, NET Application Architecture . Dostopno na: http://www.simple-talk.com/dotnet/.net-framework/.net-application-architecture-the-data-access-layer/