Jihočeská univerzita v Českých Budějovicích

Katedra informatiky Pedagogické fakulty

Technologie LINQ

LINQ technology

Autor:

Ondřej Černý

Vedoucí práce:

RNDr. Hana Havelková

České Budějovice 2011

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně, pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. V platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě pedagogickou fakultou, elektronickou cestou ve veřejně přístupné části databáze STAG, provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne

Abstrakt (Česky)

Práce se zabývá problematikou sjednoceného programovacího modelu LINQ a jeho použití v C#. Zkoumá jeho dopady na konvenční způsoby pracování s daty v objektově orientovaném programování. Obsahuje základní informace o všech hlavních způsobech implementace LINQ, ale většina textu je věnována implementaci LINQ pro SQL. Dále se zabývá jak s LINQ pracovat při tvorbě aplikací v C# a pro tyto účely poskytuje učební příručku, kde jsou vysvětleny klíčová fakta o LINQ, jeho syntaxe a výhody oproti klasickému přístupu k datům. Tato příručka však vyžaduje pro pochopení alespoň základní znalosti programovacího jazyka C#.

Abstract(English)

Purpose of this thesis is to overview unified programming model LINQ and its application in C#. It examines effects of LINQ utilization on conventional methods used while working with data in object oriented programming. Major part of this thesis discuss the implementation of LINQ to SQL, however basic information about all the significant methods are also included. Furthermore, it is engaged in development of C# applications using LINQ and may also serve as a learning guide, containing all vital facts about LINQ, its syntax, and its advantages compared to classic principles of data handling; however fully understanding this guide may be conditioned by, at least, basic knowledge of programming language C#.

Obsah

1 Úvod **6**

2 LINQ obecně **6**

2.1 Testovací databáze 7

2.2 Mapování relační databáze 8

2.3 Sestavení dotazovacího výrazu 9

3 Klíčová slova LINQ **13**

3.1 Řazení dat 13

3.2 Seskupování dat 14

3.3 Vnitřní spojení tabulek 16

3.4 Skupinové spojení tabulek 18

3.5 Vnější spojení tabulek 20

4 Operátory **21**

4.1 Dělící operátory 22

4.1.1 Take 22

4.1.2 Skip 22

4.1.3 TakeWhile 23

4.1.4 SkipWhile 24

4.2 Logické operátory 24

4.2.1 Distinct 24

4.2.2 Union 25

4.2.3 Concat 25

4.2.4 Intersect 26

4.2.5 Except 26

4.3 Operátory konverze 27
4.3.1 ToArray 27
4.3.2 ToList 27
4.3.3 ToDictionary 28
4.3.4 OfType 28
4.4 Operátory pro elementy 29
4.4.1 First a FirstOrDefault 29
4.4.2 Single 29
4.5 Kvantifikační operátory 30
4.5.1 Any 30
4.5.2 All 31
4.5.3 Contains 31
4.6 Agregační operátory 31
4.6.1 Count a LongCount 32
4.6.2 Sum 32
4.6.3 Min a Max 32
4.6.4 Average 33

5 Správa dat **33**

5.1 Vložení dat 33
5.2 Úprava dat 35
5.3 Mazání dat 35

6 Závěr **36**

7 Seznam bibliografických záznamů **38**

8 Seznam příloh **39**

1 Úvod

Dnes už prakticky každá aplikace, která pracuje s větším objemem dat, je spojena s databází. V tomto modelu však musíme vždy pracovat s dvěma druhy dat. Na jedné straně stojí objektově orientovaný jazyk a na druhé straně relační databáze. V programu si tak musíme vytvořit třídu pro každou tabulku a objekt pro každý řádek tabulky. Tento proces se nazývá objektově relační mapování.

Pokud si chceme tuto práci značně urychlit a zjednodušit, můžeme použít programovací model LINQ to SQL.

Cílem této práce je tedy sestavit příručku, která čtenáře, na názorných příkladech, seznámí s používáním LINQ to SQL ve svých aplikacích. Současně s příručkou lze také použít pracovní listy, umístěné v příloze.

2 LINQ obecně

Language Integrated Query, zkráceně LINQ je sjednocený programovací model pro práci s daty. Poprvé se objevil jako součást .NET 3.5 frameworku v listopadu 2007. V první verzi LINQ se tak objevili celkem tři implementace: LINQ to Objects, LINQ to SQL a LINQ to XML. V dnešní době má LINQ další rozšíření jako je LINQ to Entity nebo LINQ to Amazon.

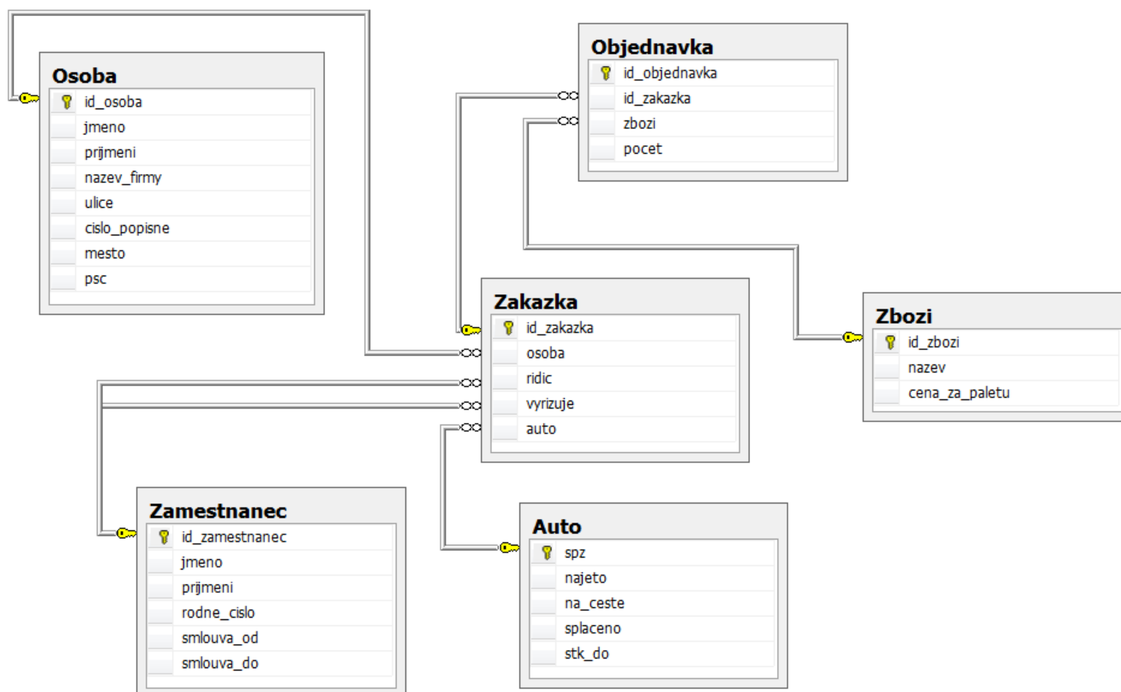
Hlavní přínos LINQ je, že sjednocuje syntax ostatních přístupů k datům. Obecně vzato, programátor může poslat dotaz do SQL databáze, do XML souboru nebo vyhledávat v poli objektů, ale musí v každém z těchto případů použít jiný přístup. To, že v LINQ použije pouze jeden, byl také důvod, proč vznikl.

LINQ přináší programátorům nový způsob, jak pracovat s daty v relační databázi. Vkládá do klasického programování další prvek, dotazovací výraz. Tento výraz vypadá jako klasický dotaz SQL, spojený se syntaxí C#. LINQ ho přeloží a pošle do SQL serveru, kde se provede požadovaná operace. Jedna z užitečných vlastností LINQ je, že výraz se neprovede do té doby, než je o data zažádáno. Proto můžeme tak výraz upravovat a měnit v průběhu aplikace, a nezatěžovat tak paměť ani procesor.

2.1 Testovací databáze

Pro potřeby této práce byla vytvořena testovací databáze „Doprava“ a byla naplněna daty. Tato databáze reprezentuje firmu, dopravující materiál zákazníkovi. Je důležité zmínit, že se nejedná o komplexní model a slouží pouze k představení LINQ.

Databáze byla zhotovena v Microsoft SQL Server Management Studio 2008, stejně jako diagram 2.1, který ukazuje vazby mezi jednotlivými tabulkami.



Obrázek 2.1: Databázový diagram

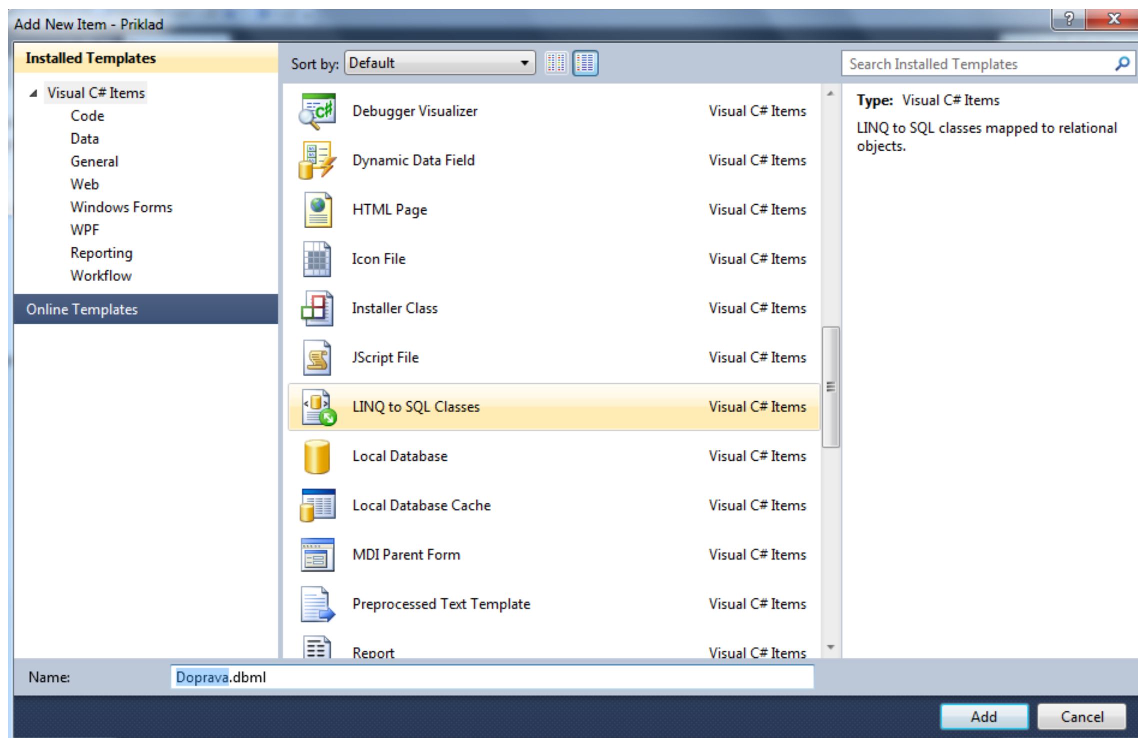
Tabulka *Osoba* reprezentuje zákazníka nebo firmu a uchovává informace o tom, kde se nachází. *Zamestnanec* obsahuje základní informace o zaměstnancích. Auto má informace o autech a o tom, zda je zrovna na cestě, jestli je splaceno atd... Z těchto tabulek se vytvoří *Zakazka*, ta se poté odkáže do *Objednavka*. *Zbozi* je soubor všech možných druhů zboží, které společnost dopravuje. *Objednavka* a *Zakazka* jsou odděleny, protože si zákazník může objednat několik různých zboží v jedné objednávce.

2.2 Mapování relační databáze

Ve Visual Studiu[1] existuje velmi mocný nástroj pro mapování relační databáze. V následující podkapitole se dozvíme, jak ho používat a jak se následně vyznat ve výsledku.

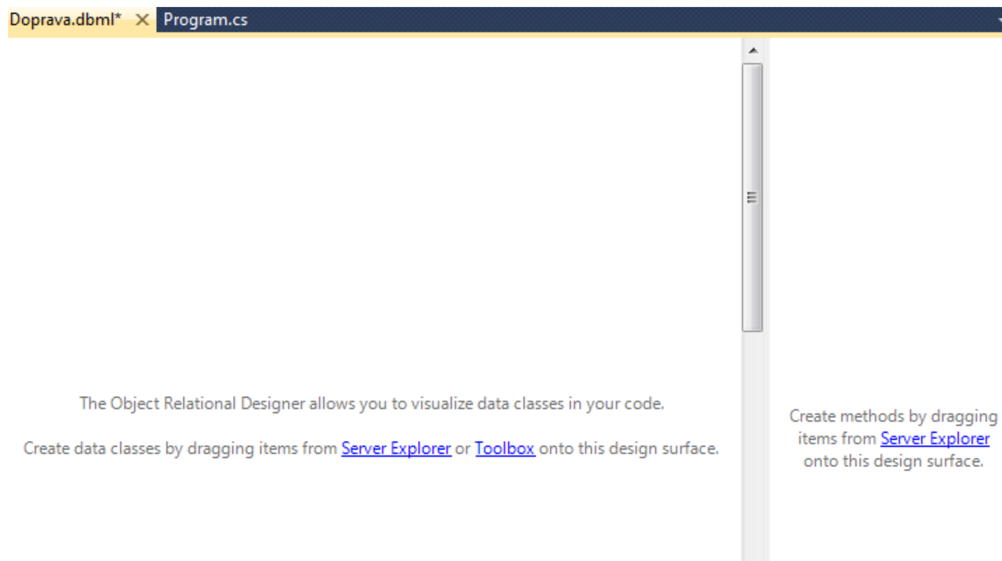
Object Relational Designer umožňuje ukládat tabulky, procedury nebo pohledy do dbml souboru. Podmínkou pro tento postup je mít správně připojenou databázi v Server Exploreru. Pokud je tato podmínka splněna, můžeme v Solution Exploreru vybrat pomocí pravého tlačítka myši položku Add > New Item...

Jak je vidět na obrázku 2.2, zde můžeme najít položku s názvem LINQ to SQL Classes. Tato položka reprezentuje samotné třídy, které se nám vygenerují po použití Object Relational Designer.



Obrázek 2.2: Přidání dbml souboru

[1] Tato podkapitola ukazuje tvorbu dbml souboru ve Visual Studiu 2010, avšak stejný postup se dá použít i ve Visual Studiu 2008



Obrázek 2.3: Prázdný dbml soubor

Obrázek 2.3 ukazuje prázdný Object Relational Designer, připravený pro použití. Levá strana je určena pro vložení tabulek a pravá strana pro pohledy a procedury. V properties pak můžeme vidět název DataContextu, který budeme potřebovat při psaní samotných výrazů. Jestliže je jméno příliš dlouhé nebo nevhodné, je dobré ho změnit. Ne vždy se zobrazí panel pro vložení pohledů a procedur. Aktivujeme ho buď pomocí klávesové zkratky Ctrl+1, nebo vybráním možnosti Show Methods Pane po stlačení pravého tlačítka myši.

Na plochu designeru přetáhneme tabulky, popřípadě procedury a uložíme. Tak se nám vytvoří celkem tři soubory. *Název*.dbml.layout je XML soubor, kde je uloženo rozvržení prvků na ploše designeru. *Název*.dbml je schéma samotné. *Název*.designer.cs je sestava tříd, kterou nám automaticky vygeneroval Object Relational Designer a o které se dozvíme něco blíže později.

2.3 Sestavení dotazovacího výrazu

V následující podkapitole se blíže podíváme na dotazovací výraz, na jeho strukturu. K tomu nám poslouží ukázkový příklad, který vrátí všechny zaměstnance se smlouvou na neurčito:

```
DopravaDataContext db = new DopravaDataContext();

var priklad = from z in db.Zamestnanec
              where z.smlouva_do.Equals(null)
              select new {z.jmeno, z.prijmeni};

foreach(a in priklad)
{
    Console.writeline(a);
}
```

Tento výraz vrací stejný výsledek jako následující SQL dotaz.

```
SELECT jmeno, prijmeni
FROM dbo.Zamestnanec
WHERE smlouva_do IS NULL
```

Z těchto příkladů můžeme poznat, že LINQ používá stejná klíčová slova jako SQL. Rozdíl je však u použití objektů, které u SQL nenajdeme.

Pro správný dotazovací výraz musíme dodržet pořadí klíčových slov. Jejich výskyty jsou znázorněny v tabulce 2.1 (1, s. 132).

Tabulka 2.1: Možné výskyty klíčových slov ve výrazu

První řádka	from s datovým zdrojem a proměnnou intervalu
Prostřední řádky	from, where, orderby, join, let nebo group_by
Poslední řádka	select nebo group_by

Klíčové slovo from

From definuje datový zdroj výrazu a proměnnou intervalu. V tomto případě *db.Zamestnanec* je rozdělen na dvě části: *db* je datacontext, o kterém si povíme něco později a *Zamestnanec* reprezentuje tabulku v relační databázi. Z je proměnná intervalu, její specifickou vlastností je nepovinné zadávání datového typu. Kompilátor si sám určí, jaký typ bude třeba při vyhodnocování výrazu, *z* tedy může například být typu string nebo int, ale v LINQ to SQL se nejčastěji setkáme s objektem.

LINQ samozřejmě podporuje i explicitní zadání datového typu. S tím se setkáme obzvláště při implementaci LINQ to Objects. Toto použijeme v případech, kdy kompilátoru není z jakéhokoli důvodu jasné, jaký typ potřebujeme (nepodaří se nám výraz zkompilovat).

```
from int z in Zamestnanci
```

V tomto příkladě donutíme kompilátor, aby o z uvažoval jako o int, i kdyby to byl objekt.

Klíčové slovo where

Where je podmínka výrazu. V LINQ funguje stejně jako v SQL, tedy zobrazí výsledky splňující podmínku. Můžeme použít logické operátory ($<$, $>$, $==$, && apod.), nebo v tomto případě operátor *Equals()* a jiné.

Klíčové slovo select

Select také funguje stejně jako v SQL. Použitím select vybíráme sloupce, které se mají ve výsledku objevit. V tomto případě ze *Zamestnanec* vybíráme pouze jméno a příjmení zaměstnance. Pokud zvolíme jen jednu složku, nemusíme vytvářet novou instanci operátorem *new*.

Odložené zpracování výrazu

Dotazovací výraz, uložený v proměnné *priklad*, se uskuteční, zeptáme-li se na data, která má vrátit. Můžeme tedy k výrazu přidávat další specifikace, aniž bychom zbytečně zatěžovali server.

Uložení výrazu v proměnné má svůj důvod. Kdyby výraz byl volně v kódu, program by ho bral řádek po řádku, a jak už víme, na prvním místě se nachází klíčové slovo *from.* V tomto případě se tabulka načte celá, protože program ještě neví o podmínce *where.*

Výrazy se nejčastěji prochází cyklem *foreach* pro svoji jednoduchost.

Datacontext

Datacontext je třída, která stojí mezi LINQ a databází. Nachází se zde popis všech tabulek a všechny základní metody na práci s nimi. [2]

```
public partial class DopravaDataContext :
        System.Data.Linq.DataContext
{
        public DopravaDataContext(string connection)
        {
        }

        public System.Data.Linq.Table<Zbozi> Zbozi
        {
        }
}
```

Jak můžeme vidět výše, tento výřez *DopravaDataContext* je odvozen od třídy *DataContext* ve jmeném prostoru System.Data.Linq. Každý objekt typu Datacontext obsahuje atribut *connection*, kde je uložena cesta k databázi. Přes tuto třídu se realizují veškerá spojení s databází, ať už ukládáme data nebo je mažeme.

[2] Bohužel LINQ to SQL podporuje jen databáze Microsoft SQL serveru (2, s. 120)

3 Klíčová slova LINQ

V této kapitole si projdeme všechny klíčová slova, která LINQ to SQL pro dotazovací výrazy nabízí.

3.1 Řazení dat

Stejně jako u klasických SQL dotazů, potřebujeme data vhodně seřadit. Pro tyto účely nám LINQ umožňuje řadit data hned několika způsoby. Následující výraz ukazuje, jak seřadit zaměstnance podle jména:

```
var seradit = from z in db.Zamestnanec
              orderby z.jmeno
              select new { z.jmeno, z.prijmeni,
                 z.rodne_cislo };
```

V tomto jednoduchém příkladě jsme použili klíčové slovo *orderby.* Jeho syntaxe se liší podle toho, jestli chceme data seřadit sestupně nebo vzestupně. *Z.jmeno* je klíč řazení a prakticky nezáleží na jeho datovém typu. V tomto případě je řadíme vzestupně (ascending), tedy od písmena A. To je defaultní hodnota, a nemusíme proto zadávat ani operátor ascending.

Chceme-li seřazovat sestupně, použijeme orderby takto:

```
orderby z.jmeno descending
```

Data můžeme řadit podle klíče, který se nakonec nezobrazí na výstupu. Lze tedy zaměstnance řadit podle jména a zobrazit jejich rodná čísla.

Jestliže chceme použít víc než jeden klíč, můžeme za čárku přidat další. Následující výpis ukazuje názorné použití, seřazení zaměstnanců podle jmen a dále podle příjmení:

```
orderby z.jmeno, z.prijmeni
```

Samozřejmě můžeme používat sestupné nebo vzestupné řazení, pomocí *ascening* nebo *descending*.

3.2 Seskupování dat

V následující podkapitole se dozvíme, jak seskupovat data. V práci s daty se často setkáme s případem, kdy potřebujeme data rozdělit do skupin, podle nějakého klíče. Například, máme-li u každého zákazníka město, můžeme ho použít jako klíč. SQL nám všechny stejná města spojí do jednoho a k nim přiřadí jednotlivé zákazníky. V následující primitivní ukázce si předvedeme syntaxi klíčového slova *Group_by*, která toto umožňuje.

```
var seskupit = from z in db.Osoba
               group z by z.mesto;

foreach(var seskup in seskupit)
{
    Console.WriteLine(seskup.Key);
}
```

Zde je na první pohled vidět trochu odlišná syntaxe, než u předchozích klíčových slov. *Group_by* se skládá ze dvou částí: za *group* následuje označení seskupovaného objektu (tabulky) a za *by* následuje specifikace sloupce. Tímto jsou data seskupena, ale abychom je mohli vidět, musíme při vypisování určit, čeho chceme dosáhnout. Tento příklad nám tedy vypíše všechna města u všech osob v databázi (každé jen jednou).

Pro lepší představu si ukážeme ještě několik alternativních použití tohoto klíčového slova. Pokud chceme ke každému z měst také vypsat, kdo tam bydlí, můžeme použít následující příklad.

```
foreach(var seskup in seskupit)
{
    Console.WriteLine(seskup.Key);
    foreach(var ses in seskup)
    {
            Console.WriteLine(ses.prijmeni);
    }
}
```

Jak jsme si již řekli, *group_by* seskupuje data podle klíče a co je důležité, přiřazuje k nim odpovídající záznamy. Toho využívají tyto vnořené cykly. První cyklus dělá to samé jako v předešlém příkladě. Druhý v každém cyklu zobrazí všechny příjmení k danému městu. Povšimněte si také, propojení cyklů, můžeme to vyjádřit jako „všechny (ses v seskup) v seskupit“.

Jestliže chceme *group_by* použít vevnitř výrazu (nikoli na konci), musíme použít další klíčové slovo *into*. Nyní rozdělíme města tak, že klíč bude tvořit vždy první písmeno daného města a osoby v nich budou spočítány.

```
var seskupit = from z in db.Osoba
               group z by z.mesto[0] into skupina
               select new { mesta = skupina.Key,
                   pocetLidi = skupina.Count() };

foreach (var q in seskupit)
{
    Console.WriteLine(q);
}
```

Zde je *group_by* použit uvnitř výrazu a tvoří tak jakousi hranici. Ve většině případů se proměnná intervalu nedostane přes tuto hranici, pracujeme za ní už s jinými daty. Například, *z* v tomto případě nemůžeme použít v *select*. *Into* tedy

vytvoří novou proměnnou intervalu, se kterou jsme schopni dále pracovat. Lze říci, jakmile se proměnná intervalu nemůže dostat přes *group_by*, převezme její průběžné výsledky proměnná intervalu zadeklarovaná pomocí klíčového slova *into* a stále s nimi pracuje (1, s. 141).

Klíčové slovo let

Pokud chceme ve svých výrazech používat proměnné, můžeme využít klíčové slovo *let*. To nám dovolí založit novou proměnnou intervalu. Použití *let* je spíše pro ulehčení práce, ale lze najít případy, kdy je velice užitečné. Následující výraz sleví všechno zboží o 15 %.

```
var promenna = from z in db.Zbozi
               let x = (z.cena_za_paletu / 100)*85
               orderby z.cena_za_paletu
               select new {cena_po_sleve = x,
                puvodni_cena=z.cena_za_paletu};
```

Pro založení proměnné tedy použijeme klíčové slovo let následujícím způsobem: *let* název_proměnné = operace.

3.3 Vnitřní spojení tabulek

V této podkapitole si představíme způsob, jakým se v LINQ spojují data, přesněji tabulky. V každé větší relační databázi, máme tabulky na sebe navázány, abychom udrželi kontinuitu dat. Tyto vazby (cizí klíče) LINQ sice vidí, ale neví, zda je má uplatnit. Tento výrok lze dokázat následujícím příkladem.

```
var pokus = from z in db.Zbozi
            from o in db.Objednavka
            select new {z.nazev, o.pocet};
```

Výraz se nám sice podaří zkompilovat, ale nad jeho užitečností se dá polemizovat. Pokud ho totiž projdeme cyklem, výsledkem bude výpis veškerého zboží v tabulce *Zbozi* (pro každou objednávku v tabulce *Objednavka*). Z toho lze usoudit, že LINQ o vazbě ví, ale nezná její využití. Ale stačí do výrazu přidat následující řádek, tabulky se korektně připojí

```
where z.id_zbozi == o.zbozi
```

S takto propojenými tabulkami již můžeme pracovat, podle svého. Ve shrnutí, primární klíč v tabulce *Zbozi* se rovná cizímu klíči v tabulce *Objednavka*. Tento způsob funguje na stejném principu, jako propojení tabulky v SQL pomocí *WHERE*. Stejně jako v SQL však existuje i jiná, a při větším počtu tabulek přehlednější, cesta.

Následující kód ukazuje, jak se v LINQ používá klíčové slovo *join*.

```
var spojit = from o in db.Objednavka
             join z in db.Zbozi on o.zbozi equals
                z.id_zbozi
             select new { o.id_objednavka, z.id_zbozi,
                z.nazev };
```

Když porovnáme tento a minulý příklad, zjistíme, že se v mnohém neliší. V prvním případě *from* vytvoří nové proměnné intervalu a *where* je spojí. V druhém případě, jednu proměnnou vytvoří *from* a druhou *join*. Join však využívá operátor *equals* místo „==“, jiná syntaxe ani není přípustná.

Takto se vytvoří spojení typu inner join, tedy vnitřní spojení, kdy musí být záznam uveden v obou tabulkách, aby byl zobrazen. Při používání klíčového slova *join* si musíme dávat pozor, v jakém pořadí píšeme objekty databáze kolem klíčového slova *equals*.

```
Objekt_z_původní_tabulky equals objekt_z_připojované_tabulky
```

V opačném případě nás naštěstí kompilátor upozorní o prohození objektů.

Jestliže chceme spojit více než dvě tabulky, musíme pro každou další aplikovat jeden *join*. Následující příklad ukazuje takové spojení.

```
var spojit = from o in db.Objednavka
             join z in db.Zbozi on o.zbozi equals
                z.id_zbozi
             join k in db.Zakazka on o.id_zakazka equals
                k.id_zakazka
             select new { o.id_objednavka, z.id_zbozi,
                z.nazev, k.vyrizuje };
```

Tímto stylem můžeme spojit tabulek, kolik potřebujeme. Opět platí posloupnost objektů kolem klíčového slova *equals*.

Více tabulek lze propojit i pomocí klíčového slova where, ale už u třech tabulek to znamená výrazně složitější a větší kód.

3.4 Skupinové spojení tabulek

Tato podkapitola pojednává o způsobu spojování tabulek v relační databázi pomocí tzv. group join. Tato metoda používá klíčové slovo *into* a nejvíce se podobá spojení typu *outer join*. Poté jsou výsledky ještě seskupeny s použitím klíče. A celkové výsledky výrazu se uspořádají hierarchicky. Neexistuje žádný SQL dotaz, který by vracel podobné výsledky, jedná se tak o specialitu LINQ (1, s. 147).

Tento poněkud složitější výraz zobrazí zakázky ke každému zaměstnanci, které vyřizuje.

```
var spojit = from z in db.Zamestnanec
             join zak in db.Zakazka on z.id_zamestnanec
                equals zak.vyrizuje into skupina
             select new { vyrizuje = z.prijmeni,
                skupinaSpojeni = skupina };

foreach (var spoj in spojit)
{
     Console.WriteLine(spoj.vyrizuje);
     foreach (var s in spoj.skupinaSpojeni)
     {
        Console.WriteLine(s.id_zakazka);
     }
}
```

Výsledky takového výrazu, jak bylo již zmíněno, jsou hierarchické. Proto použijeme vnořené cykly pro získání dat. Jelikož chceme seskupit výsledky, musíme zadat co je klíč takové skupiny. Zde je to příjmení zaměstnanců. Skupina poté obsahuje klíč a k němu přiřazené hodnoty.

Ve vnějším cyklu vypisujeme příjmení (klíč). Ve vnitřním cyklu vypisujeme id zakázky. Výsledky vypadají následovně.

```
Brožová
2
6
Cicvárek
12
Dvorská
11
Elman
```

Jestli chceme místo id zakázek zobrazit jména zákazníků, provedeme vnořený výraz ve skupině spojení.

```
var spojit = from z in db.Zamestnanec
             join zak in db.Zakazka on z.id_zamestnanec
                equals zak.vyrizuje into skupina
             select new
             {
                 Vyrizuje = z.prijmeni,
                 Spojeni = from s in skupina
                           join o in db.Osoba
                             on s.osoba equals o.id_osoba
                           select o.prijmeni
             };
```

Vnořený výraz se provádí nad výsledky předešlého výrazu a upraví je tak, že se následně spojí s tabulkou *Osoba* a vybere příjmení zákazníků. Takto spojené tabulky procházíme vnořenými cykly. Rozdílem je, že u druhého cyklu vypisujeme jen objekty z vnořeného výrazu, tedy *o.prijmeni*. Nyní výsledky vypadají následovně.

```
Brožová
   Janský
   Černý
Cicvárek
   Poskočil
Dvorská
   Koutenský
Elman
```

3.5 Vnější spojení tabulek

Někdy potřebujeme, aby se ve výsledku objevovaly i data, která nejsou přiřazena k datům v jiných tabulkách. Přesněji řečeno, všechny položky na levé straně takového spojení se zobrazí ve výsledku, i kdyby nebyly přiřazeny k položce na pravé straně spojení. Klasicky se toto používá, když chceme vypsat všechny zákazníky, bez ohledu na to, jestli si něco objednali. Pro tyto účely existuje v SQL left outer join a samozřejmě existuje i v LINQ.

Následující výpis kódu ukazuje možnost propojení tabulky jako left outer join.

```
var spojit = from o in db.Osoba
             join z in db.Zakazka on o.id_osoba
                equals z.osoba into spojeni
             from s in spojeni.DefaultIfEmpty()
             select new {jmeno= o.prijmeni,
                Zakázka = s == null ? "neobjednáno" :
                s.id_zakazka.ToString() };
```

Spojení těchto tabulek vypadá velice podobně jako v předešlých případech. Jediný rozdíl je v použití klíčového slova *from*, které vlastní jako datový zdroj skupinu definovanou o řádek výše a na kterou je použita metoda *DefaultIfEmpty*().

Podívejme se na to, jak se LINQ chová v následujícím příkladě: V databázi máme uloženého zákazníka Erika Elmana, který nemá žádnou objednávku. Když sestavíme dotazovací výraz na zjištění všech zákazníků a jejich objednávek, v konzoli se nám u pana Elmana ukáže prázdný string. Ale v případě, že se zeptáme na jeho první objednávku, LINQ již tuto situaci nezvládne a zobrazí chybu „*ArgumentNullException*“. Chybu odstraníme metodou *DefaultIfEmpty()*, jenž zaměňuje *null* za defaultní stav, který již zobrazit lze. V druhém přetížení této metody lze také nadefinovat vlastní defaultní stav.

Aby ve výsledku nebyl u zákazníků bez objednávek prázdný string, ale třeba slovo „neobjednáno“, můžeme použít podmínkový operátor „?“, jako v našem příkladě.

4 Operátory

Následující kapitola pojednává o operátorech, které nám LINQ poskytuje. V LINQ je celkem 49 operátorů, ale bohužel některé nejdou použít na implementaci LINQ to SQL. Tento nedostatek je většinou zapříčiněn faktem, že LINQ překládá výrazy na SQL pseudo-dotaz. Nelze-li tedy některá funkce nelze v SQL udělat, nelze ani v LINQ.

4.1 Dělící operátory

Tyto operátory slouží k výběru pouze specifických prvků databáze. Patří sem *Take*, *Skip*, *TakeWhile* a *SkipWhile*.

4.1.1 Take

Operátor *Take* umožňuje vybrat pouze prvních n záznamů v tabulce. Jeho syntaxe je vidět v následujícím příkladě.

```
var operatory = (from z in db.Auto
                 select z.spz).Take(3);
```

Tento výraz vrátí první tři záznamy v tabulce *Auto*. Místo trojky můžeme samozřejmě dát jakékoli jiné celé číslo.

Jak je vidět, operátor se v LINQ používá až na výsledek výrazu. Tuto syntax můžeme rozepsat do následující formy.

```
var operatory = from z in db.Auto
                select z.spz;
var promenna = operatory.Take(3);
```

Takto jsme schopni jedním výrazem získat dva rozdílné výsledky. Jestliže projdeme cyklem *operatory*, získáme všechny poznávací značky našich vozidel. Projdeme-li *promenna*, získáme pouze první tři.

4.1.2 Skip

Skip je pravý opak k *Take*. Přeskakuje prvních n záznamů v tabulce. Jeho využití je v následujícím příkladě.

```
var operatory = (from z in db.Auto
                 select z.spz).Skip(3);
```

Nyní máme vybrány všechny poznávací značky, až na první tři.

4.1.3 TakeWhile

TakeWhile vybírá prvky z databáze, dokud je splněna podmínka. To je potenciálně užitečný operátor, ale zde LINQ naráží na problém. LINQ musí tento požadavek přeložit na pseudo SQL dotaz, ale v SQL žádná taková funkce není, proto operátor *TakeWhile* podporuje pouze LINQ to Objects. Pokud se pokusíme zkompilovat výraz, který tento operátor používá, zobrazí se nám chyba „The query operator 'TakeWhile' is not supported.“. Samozřejmě je tu cesta, jak dosáhnout stejného výsledku.

```
var operatory = (from z in db.Auto
                 select z).ToList()
                 .TakeWhile(e => e.na_ceste == false);

foreach (var q in operatory)
{
    Console.WriteLine(q.spz);
}
```

Takto napsaný výraz prochází tabulku *Auto* a vrátí nám všechny poznávací značky, než narazí na auto, které není na cestě. Jako parametr *TakeWhile* se používají tzv. Lambda výrazy, jejich zjednodušený tvar je:

```
nazev_promene => podminka/vyraz
```

Proč nám jde výraz zkompilovat, když není operátor podporován? Použitím operátoru *ToList* převedeme výsledky výrazu na generické pole objektů, tedy ve výsledku pracujeme v LINQ to Objects. Ovšem síla LINQ to SQL se tím ztrácí, protože abychom mohli *TakeWhile* použít, musíme načíst celou tabulku do paměti a to u větších databází nepřipadá v úvahu.

4.1.4 SkipWhile

SkipWhile přeskakuje prvky v tabulce do té doby, než nenarazí na prvek splňující podmínku. Tento operátor nemá obdobu v SQL, a proto není LINQ to SQL podporován. Naštěstí lze stejným způsobem, jako v předchozím příkladě, nedostatek obejít. Následující kód ukazuje jeho použití.

```
var operatory = (from z in db.Auto
                select z).ToList()
                .SkipWhile(e => e.na_ceste == false);

foreach (var q in operatory)
{
    Console.WriteLine(q.spz);
}
```

Výraz přeskakuje prvky v tabulce, dokud nenajde první auto, které není na cestě.

4.2 Logické operátory

Logické operátory spojují, diskretizují nebo jinak upravují a třídí data. Do této skupiny operátorů patří: *Distinct*, *Union*, *Concat, Intersect* a *Except.*

4.2.1 Distinct

Tento operátor diskretizuje data, vybírá každý prvek pouze jednou. Jeho použití lze vidět na následujícím příkladě.

```
var operatory = (from o in db.Osoba
                 select o.mesto).Distinct();
```

Výraz vrátí všechny města našich zákazníků, každé jen jednou.

4.2.2 Union

Union funguje jako logické sjednocení $a \cup b$. Po vypsání výsledků výrazu, zůstanou jen unikátní prvky. Stejně jako u logické funkce, se union provádí mezi dvěma datovými zdroji. Následující výrazy ukazují syntaxi.

```
var prijmeni1 = from o in db.Osoba
                select o.prijmeni;

var prijmeni2 = from z in db.Zamestnanec
                select z.prijmeni;

var prijmeni = prijmeni1.Union(prijmeni2);
```

Tento výraz vrátí sjednocená příjmení zákazníků a zaměstnanců za podmínky, že se vyskytuje stejné příjmení v obou tabulkách, je zobrazeno pouze jedno.

4.2.3 Concat

Concat provádí kompletní sloučení dvou datových zdrojů. Pokud provedeme *Concat* na dvě tabulky obsahující stejná data, výsledkem bude tabulka se zdvojenými záznamy. Následující příklad vyjadřuje použití tohoto operátoru.

```
var prijmeni1 = from o in db.Osoba
                select o.prijmeni;

var prijmeni2 = from z in db.Zamestnanec
                select z.prijmeni;

var prijmeni = prijmeni1.Concat(prijmeni2);
```

4.2.4 Intersect

Intersect funguje jako logický průnik $a \cap b$. V obou datových zdrojích se najdou stejné prvky a vypíší se. Následující příklad ukazuje alternativní způsob použití *Intersect*, lze použít i dva oddělené výrazy.

```
var prijmeni = (from o in db.Osoba
                select o.prijmeni).Intersect
                (
                from z in db.Zamestnanec
                select z.prijmeni
                );
```

Výsledkem jsou všechny příjmení zákazníků, která jsou zároveň v tabulce zaměstnanců.

4.2.5 Except

Tento operátor se používá jako jednoduchý komparátor. Máme-li dvě tabulky příjmení, vybereme všechny z první až na ty, co se vyskytují ve druhé tabulceToto lze vidět v následujícím příkladě.

```
var prijmeni1 = from o in db.Osoba
                select o.prijmeni;

var prijmeni2 = from z in db.Zamestnanec
                select z.prijmeni;

var prijmeni = prijmeni1.Except(prijmeni2);
```

4.3 Operátory konverze

Tato skupina operátorů je zaměřena na konvertování výsledků výrazů do různých tvarů. Konverzní operátory jsou: *ToArray*, *ToList*, *ToDictionary* a *OfType*.

4.3.1 ToArray

ToArray konvertuje výsledek do pole typu *ArrayList*. Jeho použití vidíme v nadcházejícím výpisu kódu.

```
var operatory = (from z in db.Zbozi
                select z).ToArray();
```

Tímto se z proměnné *operatory* stane pole, s kterým již můžeme pracovat, dle uvážení.

4.3.2 ToList

Tento operátor funguje stejně jako předešlý, ale konvertuje výsledky do generického pole *List<T>*.

```
var operatory = (from z in db.Zbozi
                 select z).ToList();
```

4.3.3 ToDictionary

ToDictionary vytvoří z výsledků výrazu slovník. Jak se tento operátor používá lze vidět v následujícím příkladě.

```
var operatory = (from z in db.Zbozi
                 select z).ToDictionary
                 (e => e.id_zbozi, e => e.nazev);

Console.WriteLine(operatory[1]);
```

ToDictionary potřebuje parametrem předat informace o klíči a o datech. První z lambda výrazů určuje klíč slovníku a druhý data. V tomto příkladě vypisujeme položku pod klíčem 1.

4.3.4 OfType

Tento operátor vybírá pouze prvky určeného datového typu. *OfType* je primárně určen pro LINQ to Objects a protože v SQL nelze zadat do jednoho sloupce dva datové typy, jeho užitečnost v LINQ to SQL nechápu. Pro úplnost, následuje příklad použití *OfType*.

```
var operatory = (from z in db.Zbozi
                  select z.nazev).OfType<string>();
```

Takto napsaný výraz vybere všechny názvy v tabulce *Zbozi*, jejichž datový typ je *string*.

4.4 Operátory pro elementy

Tato skupina operátorů se zaměřuje na práci s elementy v tabulce. Patří sem: *First*, *FirstOrDefault* a *Single*.

4.4.1 First a FirstOrDefault

Operátor *First* vrací první záznam v daném výrazu. Jeho syntaxe je vidět na následujícím výpisu kódu.

```
var operatory = (from z in db.Zbozi
                 select z.nazev).First();

Console.WriteLine(operatory);
```

Jak můžeme vidět výše, použitím operátoru *First*, zmenšíme výstup pouze na jeden prvek. Není proto třeba procházet výraz cyklem, jako obvykle.

U *First* můžeme velice jednoduše narazit na chybu „InvalidOperationException“. To se nám stane, když první záznam bude prázdný. Naštěstí existuje operátor *FirstOrDefault*, jestliže najde na prvním místě *null*, vrátí defaultní hodnotu. Otázkou zůstává, proč použít *first*, když stejný výsledek vrátí *take(1)*.

4.4.2 Single

Výsledkem operátoru *Single*, jako v předešlém příkladě, je jediný záznam. Odlišnost je v tom, že single očekává pouze jeden záznam, pokud jich dostane víc, zobrazí chybu „InvalidOperationException“. Následující příklad ukazuje použití *Single*.

```
var operatory = (from z in db.Zbozi
                 where z.id_zbozi == 2
```

```
                  select z.nazev).Single();

Console.WriteLine(operatory);
```

Stanovením takto specifické podmínky, je zajištěno, že se k *Single* dostane pouze jeden záznam.

4.5 Kvantifikační operátory

Tato skupina operátorů se zaměřuje na zkoumání prvků v tabulce. Kvantifikační operátory jsou: *Any*, *All* a *Contains*.

4.5.1 Any

Any lze použít dvěma způsoby. První z nich vrací logickou hodnotu true/false, pokud existuje výsledek daného výrazu. Druhým způsobem se ptáme, zda výsledek obsahuje daný prvek. Oba tyto způsoby použití jsou uvedeny v následujícím příkladě.

```
var operatory = (from z in db.Zbozi
                 select z.nazev).Any();

var operatory2 = (from z in db.Zbozi
                  select z.nazev).Any(e => e == "Vápno");

Console.WriteLine(operatory);
Console.WriteLine(operatory2);
```

V druhém případě použití je lambda výraz, určující požadovaný řetězec.

4.5.2 All

Operátor *All* vrací logickou hodnotu true/false, na základě splnění podmínky, předané lambda výrazem. Podmínka se aplikuje na výsledky celého výrazu a logickou hodnotu true vrací jen, když jí všechny prvky splňují. Použití *All* je vidět na následujícím příkladě.

```
var operatory = (from z in db.Zamestnanec
                 select z).All(e => e.smlouva_do != null);
```

Pokud si vypíšeme výsledky tohoto výrazu, zjistíme, zda všichni zaměstnanci mají smlouvu na určitou dobu.

4.5.3 Contains

Contains vrací logickou hodnotu true/false, pokud prvek obsahuje daný řetězec nebo číslo. V následujícím příkladě je *Contains* použit jako podmínka výrazu.

```
var operatory = from a in db.Auto
                where a.spz.Contains("STA")
                select a.spz;
```

Nyní se do výsledku dostanou jen auta, jejichž poznávací značka obsahuje řetězec „STA“.

4.6 Agregační operátory

Tyto operátory provádějí jednoduché matematické a statistické výpočty ve výsledku výrazu. Mezi agregační operátory patří: *Count*, *LongCount*, *Sum*, *Min*, *Max* a *Average*.

4.6.1 Count a LongCount

Tyto dva operátory vrací celkový počet výsledků ve výrazu. *Count* však nemůže spočítat více prvků než je maximální rozsah typu Integer.[3] Pro tyto účely existuje *LongCount*, jenž místo datového typu Integer, používá Long.[4] Další příklad ukazuje použití těchto operátorů.

```
var operatory = (from o in db.Osoba
                 select o).Count();

var operatory2 = (from o in db.Osoba
                 select o).LongCount();
```

4.6.2 Sum

Sum je ekvivalent k matematické sumě $\sum$. Následující příklad ukazuje jeho použití.

```
var operatory = (from a in db.Auto
                 select a.najeto).Sum();

var operatory2 = (from a in db.Auto
                 select a).Sum(e => e.najeto);
```

Oba tyto výrazy vrátí součet najetých kilometrů u všech aut. Pokud preferujeme lambda výrazy, použijeme druhý z nich.

4.6.3 Min a Max

Tyto operátory vracejí nejmenší a největší číselnou hodnotu v daném objektu databáze. Jejich použití ukazuje následující příklad.

[3] Velikost Integer = 2^{32} tj. 0 až 4 294 967 295.
[4] Velikost Long = 2^{64} tj. 0 až 18 446 744 073 709 551 615

```
var operatory = from a in db.Auto
                select a.najeto;

Console.WriteLine("Nejméně najeto: {0},
                   Nejvíce najeto: {1}",
                   operatory.Min(), operatory.Max());
```

Zde nám konzole vypíše nejvíce a nejméně ojeté auto.

4.6.4 Average

Average vrací průměrnou hodnotu čísel v daném objektu databáze. V následujícím příkladě budeme zjišťovat, kolik mají naše auta průměrně najeto kilometrů.

```
var operatory = (from a in db.Auto
                 select a.najeto).Average();
```

5 Správa dat

V této kapitole si ukážeme, jak se pracuje s daty v LINQ to SQL. Při provozu téměř jakékoli databáze potřebujeme tři čtyři základní funkce: zobraz, vlož, změň a smaž. Protože dotazovací výrazy nahrazují funkci zobraz, zbývají nám jen tři. LINQ tyto tři funkce překládá na SQL klíčová slova: INSERT, UPDATE a DELETE.

5.1 Vložení dat

V LINQ to SQL je operace vložení dat rozdělena na čtyři kroky. Prvním krokem je vytvoření nové instance požadované tabulky.

```
Auto aut = new Auto();
```

Tento nový objekt je zatím prázdný, takže dalším krokem je ho naplnit daty.

```
aut.spz = textBox1.Text;
aut.na_ceste = checkBox1.Checked;
aut.najeto = Convert.ToInt32(textBox4.Text);
aut.splaceno = checkBox2.Checked;
aut.stk_do = dateTimePicker1.Value;
```

Tímto jsme vytvořili již plnohodnotný objekt, který může být vložen do naší databáze. Zatím je však uložen v paměti počítače a LINQ neví, co s ním chceme udělat. V dalším kroku řekneme, že chceme tento objekt vložit do tabulky.

```
db.Auto.InsertOnSubmit(aut);
```

Metoda *InsertOnSubmit()*, které jsme parametrem předali nový objekt, nevloží auto do databáze ihned. Čeká na potvrzení změn a toho dosáhneme následujícím kódem.

```
db.SubmitChanges();
```

Pokud je v databázi nastaven autoinkrement, LINQ si nejprve zjistí, jaká má být další hodnota a doplní jí za nás. Pokud nemáme, musíme si hodnotu zadat sami.

5.2 Úprava dat

V praxi se velice často setkáme se situací, kdy potřebujeme upravit již stávající záznam v databázi. Pro tyto účely existuje v LINQ velice jednoduchý způsob. Následující kód ukazuje jak upravit data v tabulce *Zbozi*.

```
var uprav = (from z in db.Zbozi
             where z.id_zbozi == 5
             select z).Single();

uprav.nazev = "Vápno";

db.SubmitChanges();
```

V prvním kroku jsme si vybrali požadovaný záznam v databázi. Operátorem *Single* je zajištěno, že se nám vybere pouze jeden záznam. Dalším krokem je samotná změna záznamu, v tomto případě přejmenováváme název produktu pod id_zbozi 5. Jako poslední krok je opět potvrzení změn metodou *SubmitChanges()*.

5.3 Mazání dat

Mazání dat probíhá v LINQ velmi podobně jako vkládání. Následující příklad ukazuje výraz, který smaže záznam v databázi.

```
var smaz = (from z in db.Zbozi
            where z.id_zbozi == 5
            select z).Single();

db.Zbozi.DeleteOnSubmit(smaz);

db.SubmitChanges();
```

O mazání v databázi se stará metoda *DeleteOnSubmit()*, která stejně jako všechny operace v této kapitole potřebuje potvrzení změn metodou *SubmitChanges()*. Pokud chceme smazat více záznamů najednou, můžeme použít metodu *DeleteAllOnSubmit()*, využívající se stejně jako *DeleteOnSubmit()*.

6 Závěr

Cíl této práce byl uvést čtenáře, co nejméně nenásilnou formou, do problematiky LINQ to SQL. Tento cíl byl splněn, avšak pro plné pochopení jak LINQ pracuje uvnitř a jak vytvářet složitější výrazy, je zapotřebí dalšího studia odborné literatury. Vzhledem ke stránkovému omezení této práce, nelze projít všechny možné použití LINQ v praxi.

Jak bylo možno vidět na předcházejících stránkách, zaimplementovat LINQ do stávajících projektů není nic těžkého. Použitím této technologie získáme přehledný kód, efektivnější práci s daty a ještě k tomu, pracujeme v čistě objektovém kódu.

Na závěr jsem odzkoušel aplikace, které jsem vytvořil v C#. První z nich jsem napsal za pomoci LINQ to SQL technologie a druhou starším přístupem, objektově relačním mapováním.

Objektově relační mapování

Tato aplikace poskytuje uživateli možnost přidávat, upravovat a mazat záznamy v databázi *Doprava*. Její vývoj byl znatelně časově náročnější, než u druhé verze, využívající technologii LINQ to SQL. Při vytváření pomocných tříd a metod jsem byl odkázán pouze na svoje schopnosti, jinak skvělý nástroj IntelliSense, zde napomáhal jen v několika málo případech. Aplikace má cca 1200 řádků pro pouze šest tabulek.

LINQ to SQL

Díky LINQ byl vývoj této aplikace asi o polovinu rychlejší a přehlednější. Skvělou vlastností LINQ je, že kompilátor nás upozorní na chyby ještě před samotným

překladem aplikace. V ORM přístupu jsem musel vždy čekat, jak bude odesílaný dotaz reagovat na danou situaci. Výsledný kód má cca 700 řádků, které jsem psal ručně. LINQ to SQL je vskutku reloluce v přístupu k datům.

Bohužel, dnešní česká literatura nevěnuje LINQ moc velkou pozornost, a proto je těžké se dostat k informacím, které potřebujeme. Naštěstí existují alternativy v podobě cizojazyčné literatury, odborných článků, MSDN dokumentace, diskusních fór atd.

Pro zlepšení představy o LINQ to SQL doporučuji projít žákovské listy, které se nacházejí v příloze.

7 Seznam bibliografických záznamů

1. CALVERT, Charlie; KULKARNI, Dinesh. Essential LINQ. Crawfordsville (Indiana) : Addison-Wesley, 2009. 564 s. ISBN 978-0-321-56416-0, ISBN-10: 0-321-56416-2.

2. PIALORSI, Paolo; RUSSO, Marco. Microsoft LINQ : Kompletní průvodce programátora. Vyd. 1. Brno : Computer Press, 2009. 615 s. ISBN 978-80-251-2735-3.

3. Visual C# Developer Center [online]. 2011 [cit. 2011-04-28]. 101 LINQ Samples. Dostupné z WWW: <http://msdn.microsoft.com/en-us/vcsharp/aa336746.aspx>.

4. MSDN [online]. 2011 [cit. 2011-04-28]. LINQ (Language-Integrated Query). Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/bb397926.aspx>.

8 Seznam příloh

1. Žákovské listy.
2. Projekt databázové aplikace využívající technologii LINQ to SQL.
3. Projekt databázovévyužívající objektově relační mapování.
4. Testovací databáze *Doprava*.

Vytvoření jednoduchého dotazu v LINQ to SQL

1 Zadání

Doplňte následující kód o dotazovací výraz tak, aby se vybrala všechna auta, kterým letos končí platnost STK. Ve výsledku potom zobrazte poznávací značku a datum, do kdy má auto platnou STK. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5       DopravaDataContext db = new DopravaDataContext();
6
7               var STK_do =
8
9       }
10  }
```

Nápověda

1. *STK_do* je ve formátu *DateTime*.
2. Pro porovnávání roku musíte nejdříve zkonvertovat *STK_do* na formát, který lze porovnat s číselnou hodnotou.
3. Můžete využít například vlastnost *Year*.
4. V kapitole 2.3 je popsáno z čeho se dotazovací výraz skládá a jak se používá.

2 Řešení

Pomocí vlastnosti *Year* můžeme z datového formátu DateTime použít jen rok. To už je číselná hodnota, která lze porovnávat.

Cyklem *foreach* potom projdeme výraz a vypíšeme ho na konzoli.

```
1          var STK_do = from a in db.Auto
2                       where a.stk_do.Year == 2011
3                       select new { a.stk_do, a.spz };
4
5            foreach (var s in STK_do)
6            {
7              Console.WriteLine(s);
8            }
9          Console.ReadKey();
```

Vytvoření jednoduchého dotazu v LINQ to SQL

1 Zadání

Vytvořte dotazovací výraz, který vybere všechny zákazníky ze Strakonic. Výsledek seřaďte vzestupně podle příjmení. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1    class Program
2    {
3        static void Main(string[] args)
4        {
5        DopravaDataContext db = new DopravaDataContext();
6
7                var priklad =
8
9        }
10   }
```

Nápověda

1. Zákazníky můžeme filtrovat pomocí klíčového slova *where*.
2. Seřazování provádíme klíčovým slovem *orderby*.
3. Řazení dat se řeší v kapitole 3.1.

2 Řešení

Nejprve si pomocí *where* stanovíme podmínku, kterou projdou jen zákazníci ze Strakonic. Použitím *orderby* se výsledek seřadí vzestupně. Dotazovací výraz poté projdeme cyklem *foreach*.

```
1              var priklad = from o in db.Osoba
2                            where o.mesto == "Strakonice"
3                            orderby o.prijmeni
4                            select new { o.jmeno, o.prijmeni };
5
6                foreach (var p in priklad)
7                 {
8                   Console.WriteLine(p);
9                 }
10             Console.ReadKey();
```

Seskupování dat v LINQ to SQL

1 Zadání

Vytvořte dotazovací výraz, který vypíše pro každou sekretářku identifikační čísla jejich zakázek. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1     class Program
2     {
3         static void Main(string[] args)
4         {
5         DopravaDataContext db = new DopravaDataContext();
6
7                 var vyraz =
8
9         }
10    }
```

Nápověda

1. Nejjednodušeji dosáhnete cíle použitím klíčového slova *group_by*.
2. Pro správné zobrazení výsledků použijte vnořené cykly.
3. V kapitole 3.2 je posáno, jak se seskupují data.

2 Řešení

V samotném výrazu stačí pouze seskupit výsledek podle sloupce *vyrizuje*. Toho dosáhneme použitím klíčového slova *group_by*. Výraz poté procházíme vnořenými cykly *foreach*. V prvním z nich vypisujeme klíč seskupení a v druhém identifikační čísla zakázek.

```
1             var vyraz = from z in db.Zakazka
2                         group z by z.vyrizuje;
3
4               foreach (var vy in vyraz)
5                {
6                  Console.WriteLine(vy.Key);
7                  foreach(var v in vy)
8                  {
9                      Console.WriteLine(v.id_zakazka);
10                 }
11               }
12            Console.ReadKey();
```

Spojování tabulek v LINQ to SQL

1 Zadání

Doplňte následující kód o výraz, který zobrazí řidiče ke každému obsazenému autu. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5       DopravaDataContext db = new DopravaDataContext();
6
7               var vyraz =
8
9       }
10  }
```

Nápověda

1. Pro zobrazení korektních výsledků spojte tabulky *Zamestnanec*, *Zakazka* a *Auto.*
2. Pokud nevíte jak jsou tabulky propojeny, použijte databázový diagram na straně 7.
3. V kapitole 3.3 je posáno, jak se spojují tabulky.

2 Řešení

Pomocí klíčového slova *join* spojíme tabulky přes cizí a primární klíče. Protože toto spojení je typu inner join, zobrazí se pouze auta, která mají přiřazené řidiče. Výraz opět projdeme cyklem *foreach.*

```
1    var vyraz = from z in db.Zamestnanec
2                join za in db.Zakazka on z.id_zamestnanec
3                    equals za.ridic
4                join a in db.Auto on za.auto equals a.spz
5                select new { z.prijmeni, a.spz };
6
7     foreach (var v in vyraz)
8     {
9         Console.WriteLine(v);
10    }
11    Console.ReadKey();
```

Spojení tabulek v LINQ to SQL

1 Zadání

Vytvořte dotazovací výraz, který vypíše pro každého řidiče identifikační čísla jeho zakázek. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1    class Program
2    {
3        static void Main(string[] args)
4        {
5        DopravaDataContext db = new DopravaDataContext();
6
7                var vyraz =
8
9        }
10   }
```

Nápověda

1. Pro tento příklad není třeba použít group join.
2. Pokud nevíte jak jsou tabulky spojeny, na straně 7 je databázový diagram.
3. V kapitole 3.3 je posáno, jak se spojují tabulky..

2 Řešení

Pro tento příklad stačí použít klasický inner join. Klíčové slovo *join* nám zajistí toto spojení. Výraz poté projdeme cyklem *foreach*.

```
1   var vyraz = from z in db.Zamestnanec
2               join za in db.Zakazka on z.id_zamestnanec
3                    equals za.ridic
4               select new { z.prijmeni, za.id_zakazka };
5
6   foreach (var v in vyraz)
7   {
8       Console.WriteLine(v);
9   }
10  Console.ReadKey();
```

Použití operátorů v LINQ to SQL

1 Zadání

Doplňte následující kód o dotazovací výraz, který vybere prvních 8 zákazníků z Českých Budějovic. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5       DopravaDataContext db = new DopravaDataContext();
6
7               var vyraz =
8
9       }
10  }
```

Nápověda

1. Pro vybrání zákazníků z Českých Budějovic použijte odpovídající podmínku *where.*
2. Vybrání prvních osmi záznamů proveďte operátorem *Take().*
3. V kapitole 4.1.1 je popsán operátor *Take().*

2 Řešení

Výraz vybírá všechny zákazníky z Českých Budějovic, na který je poté aplikován operátor *Take().* Výsledky výrazu vypíšeme cyklem *foreach.*

```
1            var vyraz = (from o in db.Osoba
2                         where o.mesto == "České Budějovice"
3                         select o.prijmeni).Take(8);
4
5              foreach (var v in vyraz)
6              {
7                 Console.WriteLine(v);
8              }
9            Console.ReadKey();
```

Použití operátorů v LINQ to SQL

1 Zadání

Napište výraz, který bude vypisovat názvy zboží do té chvíle, než narazí na položku, která je dražší než 2000. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5       DopravaDataContext db = new DopravaDataContext();
6
7               var vyraz =
8
9       }
10  }
```

Nápověda

1. Vzpomeňte si, že operátor *TakeWhile()* není podporován v LINQ to SQL.
2. Operátorem *ToList()* převedete výsledky do pole.
3. V kapitole 4.1.3 lze najít příklad použití *TakeWhile()*

2 Řešení

Nejprve si vybereme všechno zboží jednoduchým výrazem. Na tento výraz aplikujeme operátor *ToList()* abychom získali pole objektů. Do lambda výrazu operátoru *TakeWhile()* vyplníme podmínku. Nakonec vypíšeme výsledky cyklem *foreach*.

```
1              var vyraz = (from z in db.Zbozi
2                           select z).ToList()
3                           .TakeWhile(e => e.cena > 2000);
4
5                foreach (var v in vyraz)
6                {
7                   Console.WriteLine(v.nazev);
8                }
9              Console.ReadKey();
```

Použití operátorů v LINQ to SQL

1 Zadání

Dopňte následující kód o dotazovací výraz, který provede logický průnik mezi příjmení zákazníků a zaměstnanců. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1     class Program
2     {
3         static void Main(string[] args)
4         {
5         DopravaDataContext db = new DopravaDataContext();
6
7                 var vyraz =
8
9         }
10    }
```

Nápověda

1. Logický průnik provádí operátor *Intersect()*.
2. *Intersect()* se provádí mezi dvěma datovými zdroji.
3. V kapitole 4.2.4 lze najít příklad použití *Intersect()*

2 Řešení

V prvním výrazu si vybereme všechna příjmení zaměstnanců a v druhém všech zákazníků. Poté provedeme průnik mezi již zmíněnými výrazy pomocí *Intersect()*. Výsledek vypíšeme v cyklu *foreach*.

```
1          var vyraz = (from z in db.Zamestnanec
2                        select z.prijmeni).Intersect
3                       (from o in db.Osoba
4                        select o.prijmeni);
5
6           foreach (var v in vyraz)
7            {
8              Console.WriteLine(v.nazev);
9            }
10         Console.ReadKey();
```

Použití operátorů v LINQ to SQL

1 Zadání

Napište výraz, který vypíše všechny firmy, jejichž název obsahuje řetězec s.r.o. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1    class Program
2    {
3        static void Main(string[] args)
4        {
5        DopravaDataContext db = new DopravaDataContext();
6
7                var vyraz =
8
9        }
10   }
```

Nápověda

1. Pro výběr specifických řetězců se používá operátor *Contains()*.
2. V kapitole 4.5.3 lze najít příklad použití tohoto operátoru.

2 Řešení

Určíme si podmínku pomocí klíčového slova *where*, na kterou použijeme operátor *Contains()*. Jako parametr tohoto operátoru napíšeme řetězec „s.r.o.“. Výraz projdeme pomocí cyklu *foreach*.

```
1             var vyraz = from o in db.Osoba
2                         where o.nazev_firmy.Contains("s.r.o.")
3                         select o.nazev_firmy;
4 
5               foreach (var v in vyraz)
6               {
7                 Console.WriteLine(v.nazev);
8               }
9             Console.ReadKey();
```

Správa dat v LINQ to SQL

1 Zadání

Doplňte následující kód tak aby po spuštění smazal zaměstnance s příjmením Koutný. Předpokládejte, že do projektu je zapojena databáze dopravní firmy, ketrá je popsána na straně 7.

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5       DopravaDataContext db = new DopravaDataContext();
6
7               var vyraz =
8
9       }
10  }
```

Nápověda

1. Mazání se v LINQ to SQL provádí metoda *DeleteOnSubmit()*.
2. Nezapomeňte potvrdit změny pomocí metody *SubmitChanges()*.
3. Kapitola 5.3 se zabývá mazáním dat.

2 Řešení

Nejprve si vybereme jednoduchým výrazem požadovaného zaměstnance. Použitím operátoru *Single()*, se ujistíme, zda máme vybraný jen jeden prvek databáze. Poté výraz předáme jako parametr metody *DeleteOnSubmit()* datacontextu. Nakonec potvrdíme změny metodou *SubmitChanges()*.

```
1            var vyraz = (from z in db.Zamestnanec
2                         where z.prijmeni == "Koutný"
3                         select z).Single();
4
5            db.Zamestnanec.DeleteOnSubmit(vyraz);
6            db.SubmitChanges();
```