Popis vývoje perzistentního frameworku QuickPersistor v jakyzu C-Sharp

Description of development of persistent framework QuickPersistor in C-Sharp language

Bc. Filip Kovář

Diplomová práce
2011

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2010/2011

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Filip KOVÁŘ**
Osobní číslo:	**A09708**
Studijní program:	**N 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Popis vývoje perzistentního frameworku QuickPersistor v jakyzu C-Sharp**

Zásady pro vypracování:

1. **Analyzujte přístupy k perzistenci objektů.**
2. **Vypracujte požadavky na perzistentní framework.**
3. **Zpracujte analýzu frameworku.**
4. **Realizujte navržený framework.**
5. **Implementujte datové vrstvy na reálné aplikaci.**
6. **Zhodnoťte přínos a navrhněte další rozvoj frameworku.**

Rozsah diplomové práce:
Rozsah příloh:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **Microsoft. MSDN. MSDN. [Online] http://msdn.microsoft.com**
2. **Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner, C-Sharp 2008 Programujeme profesionálně. Computer Press, 2009. 978-80-251-2401-7**
3. **Bill Evjen, Scott Hanselman, Devin Rader. Professional ASP.NET 3.5 In C-Sharp and VB. Indianapolis : Wiley Publishing, 2008. 978-0-470-18757-9**
4. **Miroslav Virius. C-Sharp Hotová řešení. Computer Press, a.s. 80-251-1084-2**
5. **Jürgen Bay. C-Sharp 2005 Velká kniha řešení. Computer Press, a.s., 2007. 978-80-251-1620-3**
6. **Zabir, Omar AL. Building a Web 2.0 Portal with ASP.NET 3.5. Sebastopol : O?Reilly Media, 2007. 0-596-51050-0**
7. **The Code Poject. The Code Project. [Online] http://www.codeproject.com/**

Vedoucí diplomové práce: **Ing. Radek Šilhavý, Ph.D.**
Ústav počítačových a komunikačních systémů
Datum zadání diplomové práce: **24. února 2011**
Termín odevzdání diplomové práce: **18. května 2011**

Ve Zlíně dne 24. února 2011

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Cílem této práce je návrh, popis a příklad použití perzistentního frameworku QuickPersistor napsaného v jazyku C#. Tento framework obstarává obecnou datovou vrstvu n-vrstvých objektových aplikací. Na začátku jsou popsány různé příklady perzistence objektů s výčtem výhod a nevýhod jednotlivých řešení. Dále je specifikován seznam požadavků na perzistentní framework. Na základě seznamu požadavků je vytvořena analýza pomocí UML diagramů a následná realizace frameworku. Na závěr práce je uveden příklad implementace datové vrstvy na reálné aplikaci.

Klíčová slova: Objektově relační mapování, n-vrstvá aplikace, perzistentní framework, reflexe, relační databáze, jazyk c#, .Net Framework.

ABSTRACT

The aim of this work is a design, description and example of a persistent framework QuickPersistor use. This framework was created in C# language. QuickPersistor is designed for data layer of n-layers object applications. At its beginning, this thesis describes different ways of object persistency, mentioning their advantages and disadvantages. Furthermore, it specifies a list of requirements for persistent framework. Based on a list of requirements an analysis by UML diagrams is described and realized. At the end of this work, there is a demonstration of framework implementation to real application.

Keywords: Object relational mapping, n-layers application, persistent framework, reflection, relational database, c# language, .Net Framework.

Chtěl bych poděkovat vedoucímu své diplomové práce, Ing. Radku Šilhavému, Ph.D. za velmi vstřícný přístup. Dále bych chtěl poděkovat rodině a přátelům za jejich podporu.

Motto:

Fantazie je důležitější než vědění.

Vědění je ohraničené, fantazie pojme celý svět.

Albert Einstein

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně 5.5.2011 ……………………
podpis diplomanta

OBSAH

ÚVOD **11**

I TEORETICKÁ ČÁST **12**

1 MAPOVÁNÍ OBJEKTŮ DO RELAČNÍ DATABÁZE **13**

1.1 TŘÍDA - TABULKA 13

1.2 DĚDIČNOST 13

1.3 ASOCIACE 16

1.4 KOMPOZICE 1:1 A 1:N 17

1.5 AGREGACE 19

1.6 ASOCIAČNÍ TŘÍDA (ASOCIACE 1:N) 19

2 PŘÍSTUPY K PERZISTENCI OBJEKTŮ **21**

2.1 RUČNÍ VYTVOŘENÍ DATOVÉ VRSTVY 21

2.1.1 RDB Perzistor 21

2.1.1.1 Koncept 21

2.1.1.2 Příklad 22

2.1.1.3 Zadání 22

2.1.1.4 Návrh business vrstvy 23

2.1.2 Zhodnocení 24

2.2 GENERÁTORY DATOVÉ VRSTVY 25

2.3 DATOVÁ VRSTVA ZALOŽENA NA REFLEXI 25

2.3.1 Reflexe 26

2.3.2 Implementace datové vrstvy založená na reflexi 27

3 PŘÍKLAD EVIDENCE VOZIDEL **29**

3.1 ZADÁNÍ 29

3.2 ŘEŠENÍ 29

3.2.1 Implementace metody GetObject (Read) 30

3.2.2 Implementace metody Save (Create, Update) 33

3.2.3 Implementace metody Delete 37

3.3 DALŠÍ POŽADAVKY 37

3.3.1 Atributy 38

3.3.1.1 Vlastní atributy 38

3.3.2 Návrh řešení 39

II PRAKTICKÁ ČÁST **40**

4 POŽADAVKY NA PERZISTENTNÍ FRAMEWORK. **41**

4.1 POŽADAVKY NA PERZISTENCI OBJEKTŮ 41

4.2 POŽADAVKY NA GENERÁTOR SQL SKRIPTŮ Z MODELU 42

5 ANALÝZA **43**

5.1 UC001 - ULOŽENÍ PRIMITIVNÍCH DATOVÝCH TYPŮ DO DATABÁZE 43
Vstupní podmínky 43
Hlavní úspěšný scénář 43
5.2 UC002 - VYHLEDÁNÍ A NAPLNĚNÍ OBJEKTU Z DATABÁZE DLE ID (PRIMITIVNÍ DATOVÉ TYPY) 43
5.3 UC003 - VÝMAZ OBJEKTU Z DATABÁZE 43
Vstupní podmínky 43
5.4 UC004 - VYHLEDÁNÍ A NAPLNĚNÍ KOLEKCE OBJEKTŮ Z DATABÁZE DLE KRITÉRIÍ 44
5.5 UC005 - LAZY LOAD KOLEKCE 44
5.6 UC006 - OZNAČENÍ VLASTNOSTÍ JAKO NEPERZISTENTNÍCH 44
5.7 UC007 - VLASTNÍ IMPLEMENTACE PERZISTENTNÍCH METOD 45
5.8 UC008 - PERZISTENCE DĚDIČNOSTI 45
6 REALIZACE 47
6.1 PROJEKT QUICKPERSISTOR 47
6.1.1 Vize 47
6.2 DEFINICE ROZŠIŘUJÍCÍCH INFORMACÍ PRO PERZISTENCI 47
6.2.1 Atribut Peristent 47
6.2.1.1 Rozsah použití 47
6.2.1.2 Vlastnosti a jejich význam 47
6.2.2 Atribut NonPersistent 48
6.2.2.1 Rozsah použití 48
6.2.3 Atribut Inheritance 48
6.2.3.1 Rozsah použití 48
6.2.3.2 Vlastnosti a jejich význam 48
6.2.4 Atribut Association 48
6.2.4.1 Rozsah použití 49
6.2.4.2 Vlastnosti a jejich význam 49
6.3 IMPLEMENTACE KOMUNIKACE S DATABÁZÍ 50
6.3.1 Rozhraní IPersistor 50
6.3.2 Implementace konkrétního databázového stroje 51
6.3.3 Implementace poskytovatele pro Microsoft Sql Server 51
6.3.3.1 Důležité části perzistoru 52
6.4 VYTVOŘENÍ SCHÉMA APLIKACE PRO KOMUNIKACI MEZI VRSTVAMI 53
6.4.1 Realizace jmenného prostoru ApplicationSchema 53
6.4.1.1 Popis schéma 54
6.5 IMPLEMENTACE TŘÍD PŘEDKA PERZISTENTNÍCH OBJEKTŮ 57
6.5.1 Třída PersistentBase 57
6.5.1.1 Metody pro práci s perzistencí objektů 57
6.5.2 Třída StoredObject 58
6.6 IMPLEMENTACE DALŠÍCH FUNKCÍ 59
6.6.1 Lazy load kolekce 59
6.6.2 Podpora windows forms bindingu 60

6.6.3 Tvorba jazyku pro objektové dotazy a následná implementace parseru na sql dotazy 60
6.6.4 Generování a aktualizace databáze, tabulek, primárních a cizích klíčů dle tříd 61
7 IMPLEMENTACE DATOVÉ VRSTVY V APLIKACI VK-PORTAL 63
7.1 VIZE PROJEKTU VK-PORTAL 63
7.2 BUSINESS LOGIKA APLIKACE 63
7.3 IMPLEMENTACE FRAMEWORKU QUICKPERSISTOR 65
7.4 POUŽITÍ QUICKPERSISTORU V APLIKACI VK-PORTAL 68
8 ZHODNOCENÍ PŘÍNOSU A SEZNAM POŽADAVKŮ PRO BUDOUCÍ VERZI. 72
8.1 SEZNAM POŽADAVKŮ PRO BUDOUCÍ VERZI 72
ZÁVĚR 74
ZÁVĚR V ANGLIČTINĚ 76
SEZNAM POUŽITÉ LITERATURY 78
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 79
SEZNAM OBRÁZKŮ 80
SEZNAM PŘÍLOH 82

ÚVOD

Perzistentní vrstva je základní vrstvou snad všech třívrstvých business aplikací. Tato vrstva lze napsat mnoha způsoby. Od ručního napsání všech potřebných perzistentních metod pro všechny třídy, až po použití různých placených či neplacených frameworků. Napsání první datové vrstvy může být pro programátora objevnou a velmi zábavnou činností. Ovšem každá opakovaná práce vede člověka k hledání určitého zjednodušení. Řekl bych dokonce, že u programátorů to platí dvojnásob. Ve druhé fázi programátor začne vytýkat opakující se části do různých pomocných tříd „helperů“ nebo tříd předků. Stále ovšem zůstává napsat spoustu podobného či duplicitního kódu. Ve třetí fázi sáhne programátor pravděpodobně po frameworku třetí strany. Je ovšem velmi těžké najít framework, který by vyhovoval všem specifickým potřebám programátora a jeho projektům. Fázemi, které jsou zde popsány, jsem si také prošel. Bohužel v době, kdy jsem začal hledat nástroj či framework pro platformu .NET, byl výběr opravdu nevelký. Proto jsem začal vyvíjet svůj vlastní framework, který mi byl jak zdrojem a inspirací pro pochopení nekonečné problematiky datové vrstvy, tak i pro mé budoucí projekty.

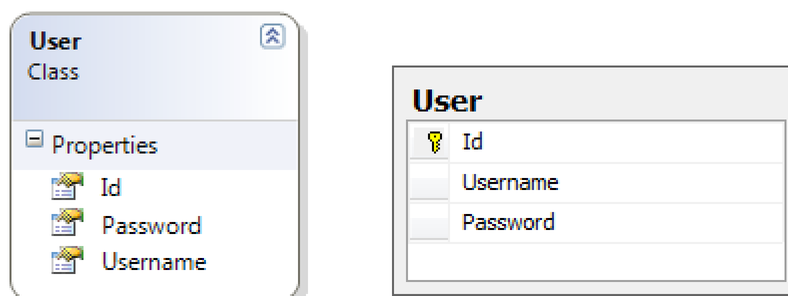
I. TEORETICKÁ ČÁST

1 MAPOVÁNÍ OBJEKTŮ DO RELAČNÍ DATABÁZE

Na úvod do problematiky tvorby datové vrstvy je nutné popsat pravidla pro ukládání objektů do relační databáze. Také zde budou zavedeny důležité pojmy pro perzistenci objektů, jako je „asociace“, „asociační třída“, „kompozice“ a „agregace“. Nemůžou zde být popsány základní pojmy objektově orientovaného programování, protože by byl přesažen rozsah práce. Pro pochopení této problematiky existuje mnoho zdrojů, například publikace, přednášky a školení pana Ilji Kravala [1].

1.1 Třída - Tabulka

Třída bude nejčastěji v objektově relačním mapování představovat tabulku. Budou, ale existovat případy, kdy bude více tříd uloženo v jedné tabulce. A to hlavně z důvodů optimalizace. Klasickým příkladem je dědičnost. Jistě je velmi přehledné oddělit společné části do třídy předka (použít takzvanou generalizaci-specializaci) v business logice. Ovšem v případě databáze (tabulek, dotazů a nahrávání objektů) může být situace složitější a občas přílišné členění a fragmentace, může být na škodu věci.

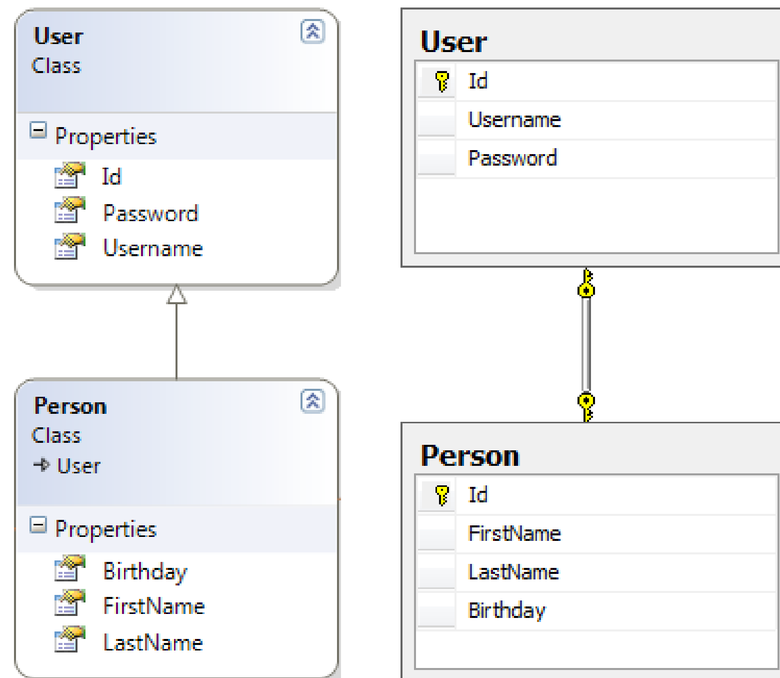


Obrázek 1. Mapování, Třída - Tabulka

1.2 Dědičnost

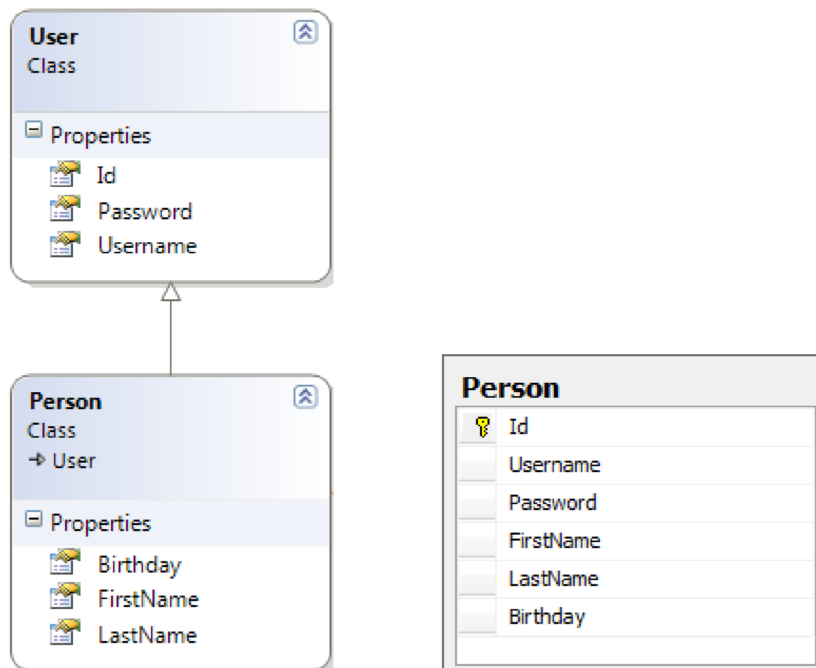
Dědičnost může být mapována následujícími způsoby:

1) 1:1 - Každá třída je mapována na jednu tabulku.



Obrázek 2. Mapování dědičnosti 1:1

2) N:1 – Více tříd je mapováno do jedné tabulky. Tato technika se používá pro optimalizaci a zjednodušení databázového schématu.

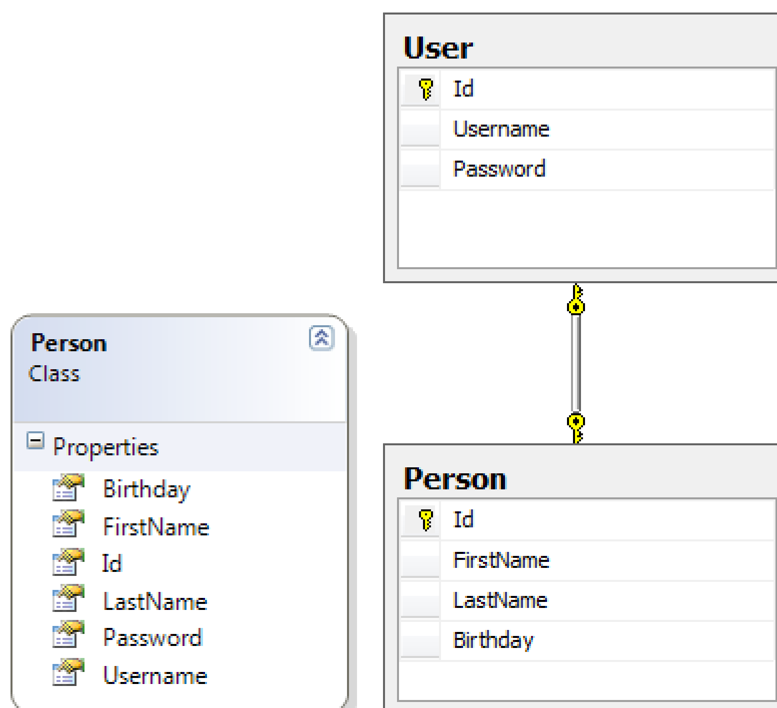


Obrázek 3. Mapování dědičnosti N:1

3) 1:N – Téměř se nepoužívá. Možné použití v případě třídy s mnoha atributy, které například přesahují omezení databázového stroje (počet sloupců).

Implementace v relační databázi:

Tabulka potomka obsahuje cizí klíč na tabulku předka. Často se dědičnost implementuje tak, že hierarchicky nejvyšší předek obsahuje sloupec Id, který se automaticky generuje. Tabulky potomků obsahují také sloupec Id, který ovšem není automaticky generován, ale přebírá hodnotu z tabulky předka.



Obrázek 4. Mapování dědičnosti 1:N

Ukázka třídy v jazyku C#

V ukázce můžeme vidět příklad dědičnosti. Třída Person dědí z třídy User. Zde může být diskutabilní, zda je tento návrh správný, ale jedná se pouze o ukázkový příklad.

```
public class Person : User
{
    #region Properties
    public IList<Address> Addresses
    {
        get;
        set;
    }

    public string FirstName
    {
        get;
        set;
    }
```

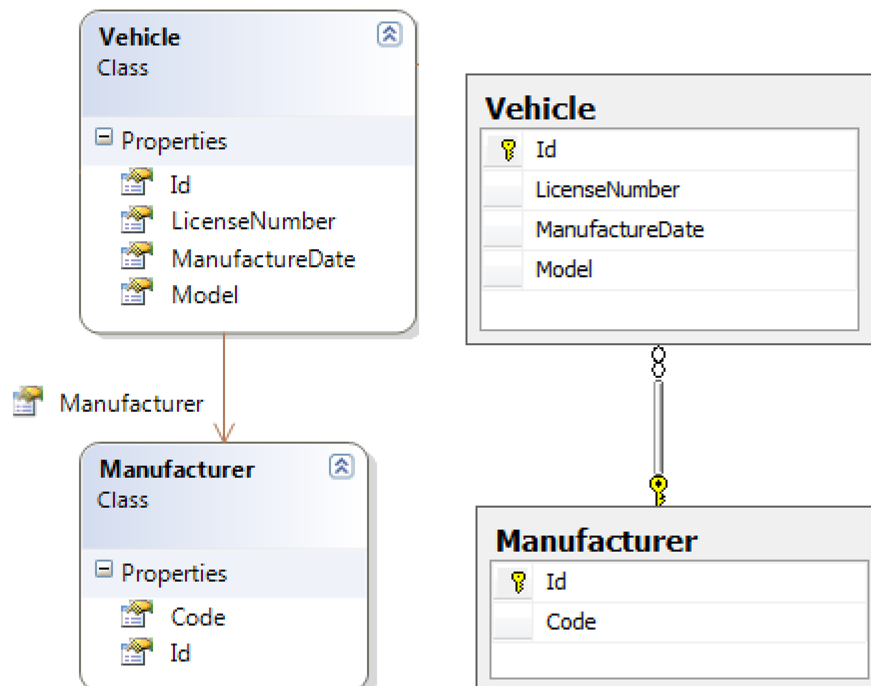
Obrázek 5. Ukázka dědičnosti

1.3 Asociace

Asociace představuje vztah, kdy asociovaný objekt neví o objektech, které jej používají. Typickým příkladem jsou číselníky (barva, značka,atp.). Existuje také speciální případ asociace a to obousměrná asociace.

Implementace v relační databázi:

Příklad: Třída A používá třídu B. **Řešení:** Tabulka A, obsahuje cizí klíč na tabulku B.



Obrázek 6. Mapování asociace

Ukázka třídy v jazyku C#

Jak vidíme, z kódu nelze poznat, o jaký vztah se jedná (Asociace, kompozice). Proto nemůže být framework implementován automaticky bez dalších nutných informací. Definici těchto informací implementuje každý framework po svém.

```
public class Vehicle
{
    #region Properties
    public int Id
    {
        get;
        set;
    }

    public Manufacturer Manufacturer
    {
        get;
        set;
    }
```

Obrázek 7. Ukázka asociace

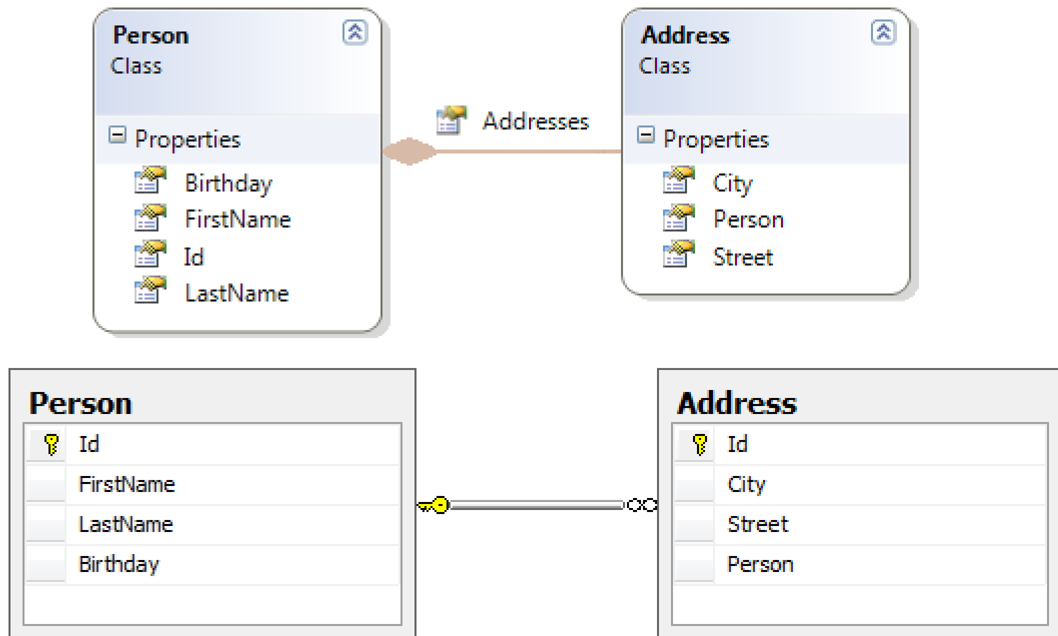
1.4 Kompozice 1:1 a 1:N

Kompozice (skládání) vyjadřuje vztah typu nadřízený a podřízený objekt. Nadřízený objekt je zodpovědný za vznik i zánik podřízeného objektu. Multiplicita 1:1 vyjadřuje, že jeden nadřízený objekt se skládá z jednoho podřízeného. V případě multiplicity 1:N je nadřízený objekt zodpovědný za vznik i zánik všech podřízených objektů. U kompozice zná podřízený objekt svůj nadřízený. V případě že toto chování není vyžadováno, je implementována takzvaná jednosměrná kompozice.

Implementace v relační databázi:

Literatura doporučuje v případě multiplicity 1:1 udržovat cizí klíč podřízené tabulky v tabulce nadřízené. U multiplicity 1:N musí být cizí klíč nadřízené tabulky v tabulce podřízené. Je to hlavně z důvodu možnosti implementovat jednosměrnou kompozici 1:1. Dle mého názoru je možné a pravděpodobně výhodnější mapovat multiplicitu 1:1 stejně jako 1:N. V případě změny návrhu aplikace z 1:1 na 1:N odpadá nutnost převodu dat. Na druhou stranu změna návrhu z kompozice na asociaci se stejnou multiplicitou je

podmíněna nutností převodu dat. Protože asociovaný objekt nesmí znát nadřízený objekt. Tento případ změny návrhu je ale méně častý.



Obrázek 8. Mapování kompozice

Ukázka třídy v jazyku C#

Z ukázky lze vypozorovat multiplicitu vztahu pomocí typu objektu. Pokud objekt implementuje rozhraní IEnumerable, ICollection, IList, atp. jedná se o vztah 1:N. U tohoto vztahu lze předpokládat i typ vztahu a to kompozici. Asociace 1:N se vytváří pomocí asociačních tříd.

```
public class Person : User
{
    #region Properties
    public IList<Address> Addresses
    {
        get;
        set;
    }

    public string FirstName
    {
        get;
        set;
    }
```

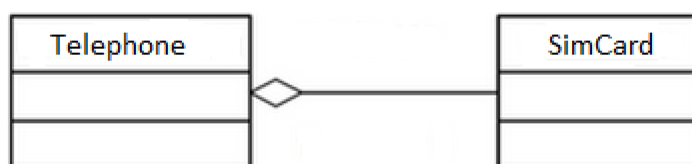
Obrázek 9. Ukázka kompozice 1:N

1.5 Agregace

Agregace je speciální typ asociace. Je to vztah podobný kompozici (skládání) s tou výjimkou, že podřízený objekt je možno sdílet mezi jinými objekty. Nejlépe si to můžeme ukázat na příkladu. Představme si SIM kartu. Tato karta může být jak součástí telefonu, tak například modemu.

Implementace v relační databázi:

Agregace se implementuje v relační databázi stejně jako asociační třída (viz asociační třída). Je to vlastně vazba mezi objektem a jeho vlastníky. Agregovaný objekt může mít N vlastníků.



Obrázek 10. Class diagram agregace

Ukázka třídy v jazyku C#

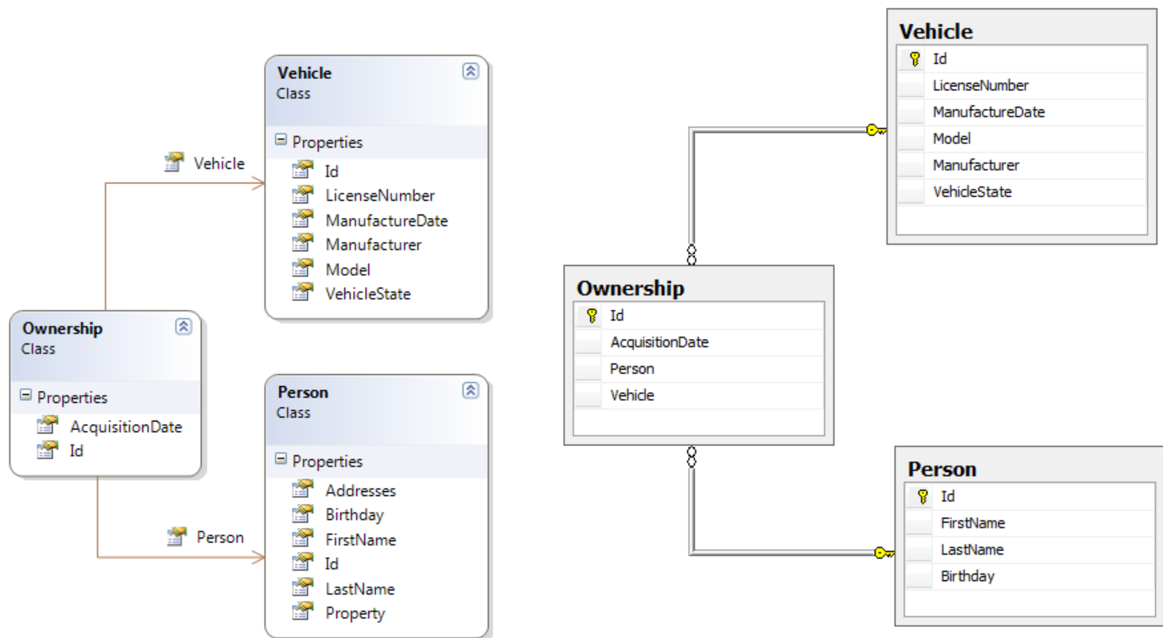
Implementace je stejná jako asociační třída. S tím, že se musí definovat typ vztahu (agregace).

1.6 Asociační třída (Asociace 1:N)

Asociační třída představuje řešení asociace 1:N. Víme, že asociovaný objekt nesmí znát objekt, jenž ho používá. Otázkou tedy je, jak řešit asociační vazbu M:N pokud asociovaný objekt nesmí znát svého „nadřízeného“. U kompozice můžeme přidat vazbu z podřízeného na nadřízený. V případě asociace musíme zavést další třídu představující vztah mezi objekty. Například u evidence vozidel a jejich vlastníků. Každé vozidlo může mít N vlastníků a každý vlastník může mít N vozidel. Zavedeme proto novou třídu, kterou můžeme pojmenovat například vlastnictví „Ownership“. U toho příkladu je diskutabilní, zda se jedná o kompozici nebo asociační třídu. Nad výsledkem není nutné příliš dlouho přemýšlet, protože oba vztahy se mapují stejně.

Implementace v relační databázi:

Nově vytvořená tabulka představující vztah obsahuje cizí klíče na tabulky, se kterými je ve vztahu.



Obrázek 11. Mapování asociační třídy

Ukázka třídy v jazyku C#

```
public class Ownership
{
    #region Properties
    public Person Person
    {
        get;
        set;
    }

    public Vehicle Vehicle
    {
        get;
        set;
    }

    public DateTime AcquisitionDate
    {
        get;
        set;
    }
```

Obrázek 12. Ukázka asociační třídy

Většina informací pro tuto kapitolu byla čerpána z knihy pana Ilji Kravala [6].

2 PŘÍSTUPY K PERZISTENCI OBJEKTŮ

V této kapitole budou popsány různé způsoby a přístupy k perzistenci objektů. Perzistencí zde bude myšleno ukládání objektů do relační databáze. U každého způsobu budou uvedeny výhody a nevýhody. Nelze s určitostí říct, který způsob je nejlepší. Hodnocení bude tedy ponecháno na čtenáři.

2.1 Ruční vytvoření datové vrstvy

Pokud se rozhodne programátor začít psát třívrstvé aplikace, měl by se porozhlédnout po nějakých návrhových vzorech či vzorových řešení. Ať už se jedná o návrh architektury, jmenné konvence, nebo vzor datové vrstvy. V dnešní době je spoustu návrhových vzorů pro datovou vrstvu. Dříve bylo možno vycházet například ze vzoru pana Ilji Kravala [1]. Tento člověk je jedním z největších propagátorů a šiřitelů myšlenek objektově orientovaného programování v České republice. Mezi jeho vydanými skripty a knihami lze najít i materiály věnované datové vrstvě. V knize popisuje takzvaný RDB perzistor a uvádí čtenáře do problematiky ukládání objektů do relační databáze. Součástí jsou i ukázky v různých programovacích jazycích (C#, Visual Basic, Java, atp.).

2.1.1 RDB Perzistor

V této podkapitole bude čtenář seznámen s návrhem datové vrstvy podle pana Kravala. Vědomě není nazýván frameworkem, protože datová vrstva RDB Perzistoru je specifická pro každou aplikaci.

2.1.1.1 Koncept

Předpokladem je, že je vytvořena business logika aplikace. Následně je potřeba „naučit“ třídy perzistenci. Začne se tím, že bude vytvořeno rozhraní předepisující CRUD metody pro každou třídu. Tyto metody by měly obsahovat parametry primitivních datových typů. Toto rozhraní je zavedeno hlavně z důvodů možnosti implementace více databází. To znamená, že aplikace nebude závislá na jednom databázovém stroji. Určitě je zbytečné implementovat na počátku aplikace veškeré známé databáze, hlavní výhodou je otevřenost řešení pro odlišné implementace. Každá vrstva n-vrstvé aplikace by měla být pokud možno co nejjednodušeji nahraditelná jinou implementací, aniž by to ostatní vrstvy „poznaly“.

Po definici rozhraní přichází na řadu implementace pro danou databázi. Zde je nutné dodržet předepsané metody a vlastnosti. Pokud toto nebude dodrženo, překladač bude hlásit chyby a aplikaci nebude možno spustit. Po implementaci metod specifických pro danou databázi bude nutno naprogramovat volání z business logiky. V RDB Perzistoru je doporučováno vytvoření předka pro všechny business třídy a do něj implementovat společné vlastnosti a metody. Například „Id“ (GUID), „State“,“ Save“, „Load“, „Delete“, atp. Za zmínku stojí metoda „Save“ místo metody „Insert“ a „Update“. Je to proto, že objekt zná svůj stav a na základě něj dokáže rozhodnout, zda bude z datové vrstvy volat „Insert“ nebo „Update“.

Po implementaci metod je vhodné definovat způsob, jakým bude možno vytvářet správnou instanci datové vrstvy pro danou databázi. Například návrhový vzor Factory.

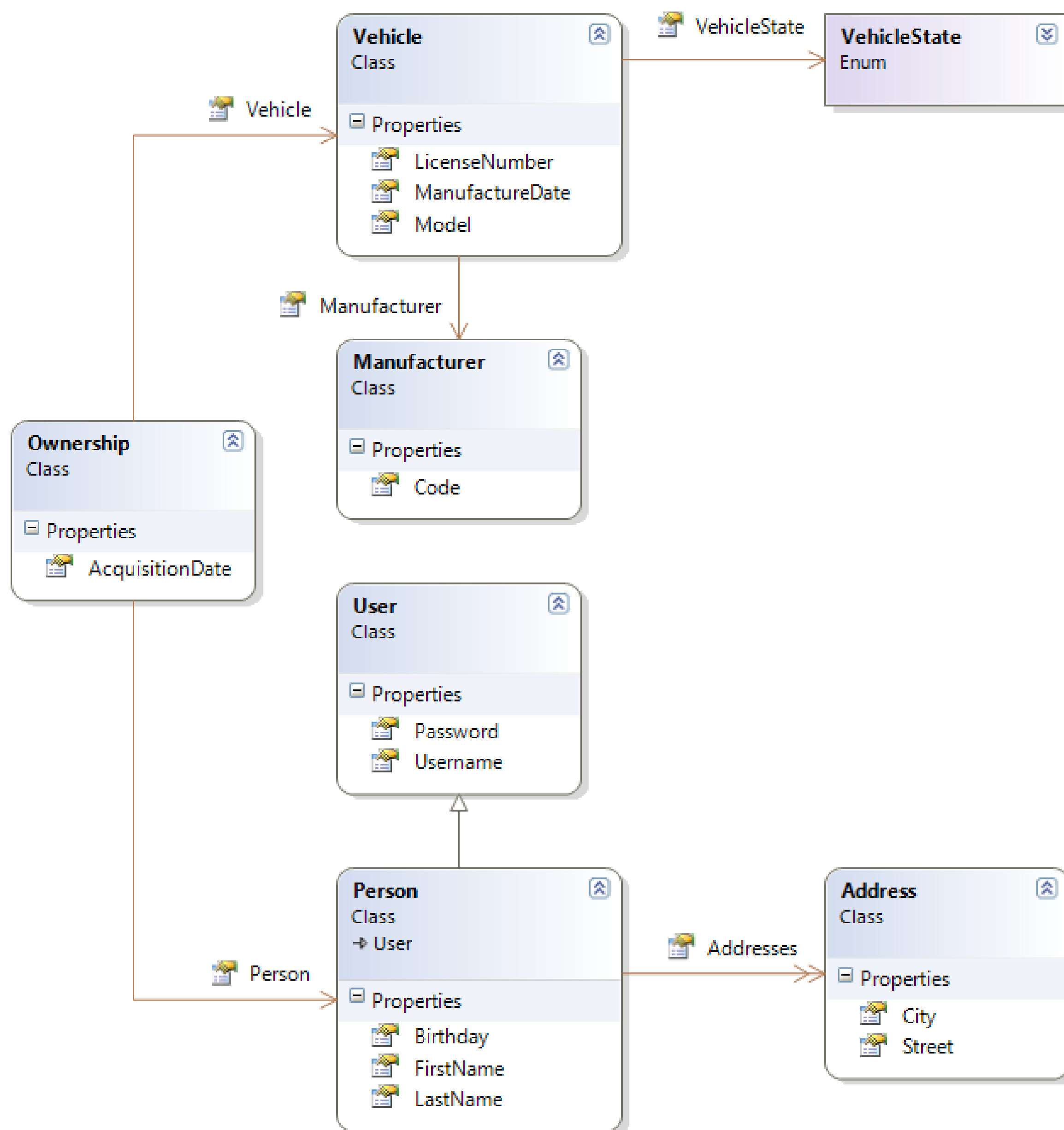
2.1.1.2 Příklad

Jako příklad bude uvedena velmi zjednodušená aplikace pro evidenci vozidel.

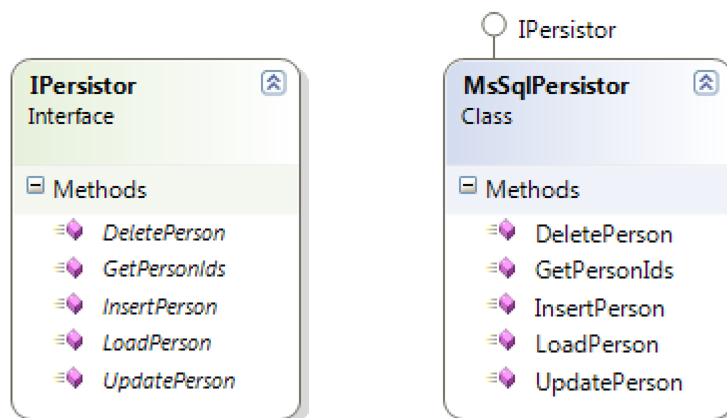
2.1.1.3 Zadání

Požadavkem je evidovat osoby a jejich aktuální a minulé vlastnictví vozidel. U osob bude evidováno: Jméno, příjmení, datum narození a adresa. U vlastnictví datum pořízení a u vozidla datum výroby, státní poznávací značka, výrobce, model a informace zda bylo vozidlo zlikvidováno nebo vyřazeno z evidence.

2.1.1.4 Návrh business vrstvy



Obrázek 13. Class diagram aplikace bez implementace datové vrstvy



Obrázek 14. Ukázka datové vrstvy

2.1.2 Zhodnocení

Z obrázků je patrné, že tvorba datové vrstvy bude velmi náročný úkol. Pro každou třídu, vztah, vlastnost bude nutno implementovat metody do rozhraní a databázově specifických tříd. Při přidání nové vlastnosti nebo třídy či vztahu bude nutno implementovat toto chování v dalších třídách. Většina operací se v konečném důsledku stane opakovanou činností typu „Copy&Paste”. Je více než pravděpodobné, že tento způsob vývoje datové vrstvy je náchylný na chyby.

Výhody:

1) **Jednodušeji se řeší individuality v aplikaci.**
2) **Celý kód datové vrstvy je pod kontrolou programátora.**
3) **Jednoduše pochopitelný kód.**

Nevýhody:

1) **Spoustu duplicitního kódu.**
2) **Pomalý vývoj a úpravy.**
3) **Náchylné k chybám typu „Copy&Paste”.**

2.2 Generátory datové vrstvy

Existuje celá řada frameworku, které více či méně dobře generují třídy a metody pro přístup k databázi. Některé obsahují různé návrháře a průvodce pro nastavení generovaných tříd a operací, jiné jsou otázkou vnitřní konfigurace. Všechny generátory vygenerují třídy a metody na základě databázového schéma (schema-first) nebo tabulky na základě modelu (model-first). Kromě základních CRUD metod generují další užitečné třídy a metody. Například načtení dat z databáze do objektů dle určitých omezení, filtrů, řazení, seskupení atp. Některé dokonce umožňují data stránkovat nebo nabízí kompatibilní datové zdroje s prvky windows a web formulářů.

Generátory datové vrstvy jsou v podstatě zautomatizování ruční implementace datové vrstvy. Mnohem snadněji se zde implementují speciality dané požadavky na budoucí systém. Nevýhodu je zde opět duplicita kódu, i když pod kontrolou frameworku či programu generátoru. Problémem také může být kombinace generovaného kódu s ručními zásahy. To lze v celku elegantně řešit pomocí takzvaných „partitial classes“ či dědičnosti jako je tomu například u návrháře ve windows forms. Mezi nesporné výhody (stejně jako u ostatních frameworků) je rychlejší vývoj aplikací. Programy pro UML návrhy obsahují generátory tříd a dokonce funkcionalitu reverse engineering. Toto umožňuje z diagramů tříd generovat třídy v daném jazyku a naopak. Některé generátory dokonce nevytvářejí kód v programovacím jazyku, ale řeší perzistenci převážně nad databází ve formě uložených procedur, pohledů a triggerů.

Výhody:

1) **Jednodušeji se řeší individuality v aplikaci.**
2) **Jednoduše pochopitelný kód.**

Nevýhody:

1) **Spoustu duplicitního kódu.**

2.3 Datová vrstva založena na reflexi

Třetí typ vzorů a frameworků pro tvorbu datové vrstvy jsou ty založené na tzv. reflexi. Jedná se v dnešní době o velmi oblíbenou skupinu jak pro menší aplikace, tak pro rozsáhlá řešení. Obecně použití frameworku má velkou výhodu v dodržení jednotných principů.

Pokud je část aplikace sjednocená a postavené na stejných principech je jednodušší vytvářet další obecné prostředky pro zrychlení vývoje. Představme si například framework, který k business logice doplní datovou vrstvu a navíc obsahuje datové prvky pro windows, web a mobile řešení. V podstatě se dá aplikace „naklikat“ bez nutnosti speciálních programovacích dovedností. To vše při zachování čistoty návrhu n-vrstvé aplikace. Samozřejmě toto asi nebude platit u všech typů aplikací.

2.3.1 Reflexe

Sestavení obsahují metadata, včetně podrobností o všech typech a členech těchto typů definovaných v daném sestavení. Proto lze k těmto metadatům přistupovat prostředky programovacích jazyků. Tento postup označovaný jako reflexe přináší zajímavé možnosti. Vyplývá z něj totiž, že řízený kód může analyzovat jiný řízený kód, nebo dokonce zkoumat sám sebe a zjišťovat informace o příslušném kódu. Nejčastěji se používá k získání podrobností atributů, ačkoli pomocí reflexe můžete mimo jiné nepřímo vytvářet instance tříd nebo volat metody tak, že uvedete názvy těchto tříd nebo metod jako řetězce. Tímto způsobem je možno na základě vstupu od uživatele vybrat třídy, které vytvoří instance

metod pro volání za běhu, nikoli v době kompilace (dynamická vazba).

Reflexe je obecný pojem, který popisuje možnost kontrolovat součásti programu a manipulovat s nimi za běhu. Reflexe například umožňuje:

- Vytvořit výčet složek určitého typu
- Vytvořit instanci neznámého typu
- Spustit metodu objektu
- Zjistit informace o typu
- Vyhledat informace o sestavení
- Prozkoumat vlastní atributy aplikované na typ
- Vytvořit a přeložit nové sestavení

Tento seznam představuje mnoho různých nástrojů a zahrnuje některé z nejsilnějších a nejsložitějších možností, které knihovna tříd platformy .NET Framework poskytuje. V této kapitole není dostatek místa, aby molo být popsáno toto téma komplexně. [4]

2.3.2 Implementace datové vrstvy založená na reflexi

Pokud máme naprogramovány třídy. Je nutné definovat perzistentní atributy a vztahy jako jsou například: Vztah je typu asociace, kompozice, vlastnosti je perzistentní a nemůže být null atp.

Každý framework implementuje tyto vlastnosti po svém. Například NHibernate požaduje nadefinování všech informací do xml konfigurace. Vyplnění takové konfigurace může být velmi náročnou a chybovou operací. Jiné frameworky mají definovány atributy, které se přímo vkládají do tříd. V tomto případě stačí dát například nad třídu atribut, který určuje, do jaké tabulky se budou data ukládat atp. Toto řešení má tu nevýhodu, že porušuje pravidlo „Single responsibility principle“. To znamená, že do třídy business logiky je vkládán další kód, který nesouvisí s funkcionalitou objektu. Každý objekt by měl být zodpovědný pouze za jednu věc. [5]

Ve frameworku implementující návrhový vzor ActiveRecord Martina Fowlera by mohla vypadat implementace datové vrstvy asi nějak takto:

```
[Persistent]
public class Person : User
{
    #region Properties
    public int Id
    {
        get;
        set;
    }

    [Persistent(AssociationType.Composition)]
    public IList<Address> Addresses
    {
        get;
        set;
    }
```

Obrázek 15. Možné řešení perzistence

Tento kód je ve vymyšleném frameworku a ilustruje doplnění chybějících informací pro perzistenci.

Výhody:

1) **Veškerý kód datové vrstvy je v samostatné knihovně.**

2) **Velmi jednoduchá implementace.**

Nevýhody:

1) **Porušení principu Single responsibility principle.**

2) **Kód datové vrstvy není plně pod kontrolou programátora.**

3 PŘÍKLAD EVIDENCE VOZIDEL

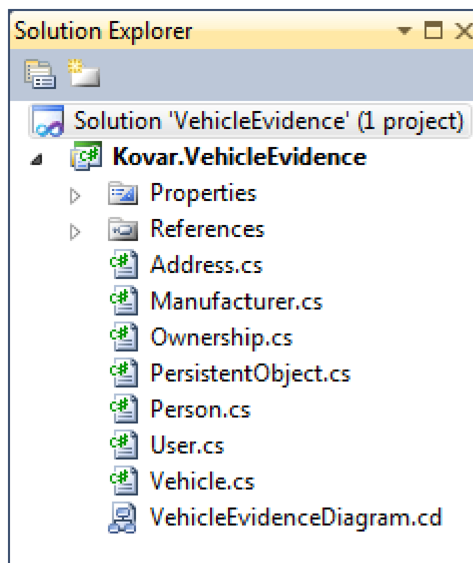
V této kapitole bude vytvořen příklad a budou definovány požadavky pro ideální perzistentní framework. Některé principy budou implementovány. Dále bude prototypován vlastní framework. Prototypování je velmi užitečnou technikou (metodou) na odhalení největších neznámých a hrozeb nového projektu. U Prototypů bývá uváděna nevýhoda, že se v rámci ušetření nákladu a času použije kód z prototypu. To většinou může způsobit problémy, protože kód prototypu je psán v rychlosti, přímočaře a bez jakéhokoliv návrhu.

3.1 Zadání

Pro příklad evidence vozidel bude použito zadání z druhé kapitoly. Pro tento případ nebude řešena prezentační vrstva, postačí pouze business logika a velmi zjednodušená datová vrstva.

3.2 Řešení

Podle zadání vytvoříme class diagram. Snahou je v něm podchytit veškeré entity, atributy a vztahy. V tomto příkladu je obsažena: Dědičnost, asociace, kompozice. Založíme si nové řešení (solution) ve vývojovém prostředí Visual Studio. Dále je nutno logicky rozčlenit řešení na knihovny. V našem jednoduchém příkladu bude zavedena pouze jedna knihovna pro business logiku. Budeme se držet podle jmenných konvencí Microsoftu. V solution založíme nový projekt typu class library pojmenované Kovar.VehicleEvidence. První název je vhodné volit například podle firmy, pro kterou program vytváříme. Na základě názvu je pak jednoduché zjistit, autora knihovny. V našem projektu vytvoříme z class diagramu třídy a vlastnosti. Nyní se podíváme na solution ve vývojovém prostředí Visual Studio.



Obrázek 16. Okno řešení ve vývojovém prostředí Microsoft Visual Studio 2010

Pomalu začneme uvažovat nad implementací datové vrstvy. Výše jsme uvedli, že budeme prototypovat datovou vrstvu. Založíme si nyní projekt typu class library pro obecnou datovou vrstvu s názvem „Kovar.DataAccessLayer“. V tomto projektu si založíme obecnou třídu, do které budeme implementovat všechny společné vlastnosti pro perzistentní třídy. Tato třída bude použita jako předek pro všechny perzistentní třídy. Určitě bychom měli začít s CRUD operacemi a založení identifikátoru. Bylo by vhodné zavést možnosti jak s „Id“, tak i bez něj (Ne každý jej bude chtít používat). Například pomocí další obecné třídy předka.

Založíme tedy třídu pojmenovanou „PersistentObject“. Přidáme vlastnost typu Int32 s názvem Id. Dále v projektu „VehicleEvidence“ přidáme referenci na „Kovar.DataAccessLayer“ a podědíme všechny třídy (kromě třídy Person, která je děděná ze třídy User) ze třídy „PersistentObject“.

3.2.1 Implementace metody GetObject (Read)

Pro zjednodušení budeme uvažovat o tom, že všechny vlastnosti objektů poděděného z „PersistentObject“ budou perzistentní, tabulka bude mít shodný název s třídou a sloupce v tabulce budou mít shodný název s vlastnostmi. Metoda „GetObject“ bude generická a bude obsahovat parametr Id. Tuto metodu budeme implementovat do třídy

„PersistentObject“. Navíc přidáme interface „IPersistor“ a další třídu s názvem „MsSqlPersistor“. Interface bude obsahovat předpis pro metody a vlastnosti, které bude muset obsahovat každý perzistor specifický pro databázový stroj. Třída „MsSqlPersistor“ bude obsahovat kód specifický pro databázi „Microsoft Sql Server“. Budeme počítat s možnosti jednoduše implementovat další databázové stroje. Pro náš příklad nám bude stačit pouze jeden a pro zjednodušení se vyhneme i návrhovému vzoru factory pro vytvoření správné instance perzistoru. Přidáme ještě další testovací projekt typu WPF application s názvem VehicleEvidence.

Metoda GetObject bude implementována následovně:

```
T loadedObject = null;
using (SqlConnection connection = new SqlConnection(_ConnectionString))
{
    using (SqlCommand command = new SqlCommand(
        string.Format("SELECT * FROM {0} WHERE Id=@Id"
        , ReflectionHelper.GetTableName<T>()), connection))
    {
        SqlParameter parameterId = command.CreateParameter();
        parameterId.ParameterName = "Id";
        parameterId.Value = id;
        command.Parameters.Add(parameterId);

        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            if (!reader.HasRows) return loadedObject;
            loadedObject = ReflectionHelper.CreateAndFillObject<T>(reader);
            reader.Close();
        }
    }
}
return loadedObject;
```

Obrázek 17. Ukázka základní metody pro naplnění objektu z databáze

Tento kód není nic jiného, než standardní dotaz do databáze pomocí ADO.NET. Zajímavé je až volání metody, která pomocí reflexe vytvoří novou instanci objektu a naplní jeho vlastnosti daty z databáze.

Vytvoření instance objektu a naplnění daty

```
T loadedObject = null;
if (reader.Read())
{
    loadedObject = Activator.CreateInstance<T>();
    for (int i = 0; i < reader.FieldCount; i++)
    {
        loadedObject.SetPropertyValue(reader.GetName(i), reader.GetValue(i));
    }
    loadedObject.PersistentState = PeristentState.Loaded;
}


return loadedObject;
```

Obrázek 18. Ukázka vytvoření nové instance objektu a naplnění daty

Zde jsou procházeny všechny sloupce databáze a pomocí reflexe jsou přiřazeny jejich hodnoty do vlastností objektu.

Přiřazení hodnoty do vlastnosti objektu pomocí reflexe

```
PropertyInfo property = this.GetType().GetProperty(propertyName);
if (property == null) return;
if (value != null && value != DBNull.Value)
{
    property.SetValue(this, Convert.ChangeType(value, property.PropertyType), null);
}
else
{
    property.SetValue(this, null, null);
}
```

Obrázek 19. Ukázka přiřazení hodnoty do vlastnosti

V kódu je ukázáno získání instance objektu typu „PropertyInfo“ pomocí názvu vlastnosti. Třída „PropertyInfo“ obsahuje spoustu užitečných informací a metod o vlastnosti. Například název, datový typ, modifikátory přístupu, atributy, atp.

Nyní otestujeme náš kód. Do databáze si naplníme testovací data, vytvoříme instanci „MsSqlPersistoru“ a zavoláme metodu „GetObject“. Zde je potřeba upozornit, že by bylo vhodné implementovat návrhový vzor „Singleton“ (Jedináček) pro vytvoření pouze jediné instance daného typu perzistoru.

```
IPersistor persistor = new MsSqlPersistor(
@"server=xxxxxx\SQLEXPRESS;Integrated Security=false; user=xxxxxx;
password=xxxxx; database=VehicleEvidence;Persist Security Info=true");
Manufacturer manufacturer = persistor.GetObject<Manufacturer>(1);
if (manufacturer != null)
{
    MessageBox.Show(manufacturer.Code);
}
```

Obrázek 20. Ukázka inicializace frameworku a získání objektu z databáze

První řádek kódu by bylo vhodné nahradit návrhovým vzorem „Factory“, který na základě konfigurace vytvoří správnou instanci konkrétního databázového perzistoru. Připojovací řetězec by bylo vhodné také umístit do konfiguračního souboru. Pro čistotu návrhu je správné pracovat s databází pouze přes interface „IPersistor“. Další řádek ukazuje, jak metodu „GetObject“ volat. Do budoucna je možné implementovat různá přetížení pro tuto metodu, například vyhledání dle kritérií. Bylo by vhodné také implementovat „Linq“.

3.2.2 Implementace metody Save (Create, Update)

Metoda Save bude odpovědná za uložení objektu do databáze. Zde bude vhodné odstínit programátora od nutnosti volat metody „Insert“ a „Update“. Tyto metody bude volat framework na základě vnitřních stavů objektu. Tato myšlenka nás vede k vytvoření výčtu perzistentních stavů objektů. Vytvoříme tedy ve třídě PersistentObject výčet s názvem „PersistentState“ a naplníme ho hodnotami „New“, „Loaded“.

```
public enum PeristentState
{
    New,
    Loaded
}
```

Obrázek 21. Stavy objektu

Při vytvoření instance perzistentního objektu mu bude automaticky nastavena hodnota „PersistentState“ na hodnotu „New“. V případě, že dojde k uložení objektu do databáze nebo jeho naplnění z databáze, bude jeho stav roven „Loaded“. Metoda „Save“ se na základě tohoto stavu bude rozhodovat, zda volat Insert nebo Update.

```
if (_PersistentState == PeristentState.New)
{
    Id = Persistor.Instance.InsertObject(this);
}
else
{
    Persistor.Instance.UpdateObject(this);
}
PersistentState = PeristentState.Loaded;
```

Obrázek 22. Uložení objektu

Provedli jsme zjednodušenou implementaci návrhového vzoru Singleton, pro jednodušší práci s datovou vrstvou.

Implementace metody Insert (Create)

```
using (SqlCommand command = new SqlCommand())
{
    StringBuilder columnNames = new StringBuilder();
    StringBuilder values = new StringBuilder();
    foreach (PropertyInfo property in
        persistentObject.GetType().GetProperties())
    {
        if (property.Name.Equals("PersistentState") ||
            property.Name.Equals("Id")) continue;

        if (columnNames.Length > 0) columnNames.Append(",");
        columnNames.Append(property.Name);

        if (values.Length > 0) values.Append(",");
        values.Append("@").Append(property.Name);

        SqlParameter sqlParameter = command.CreateParameter();
        sqlParameter.ParameterName = property.Name;
        object value = property.GetValue(persistentObject, null);
        if (value == null) value = DBNull.Value;
        sqlParameter.Value = value;
        command.Parameters.Add(sqlParameter);
    }

    using (SqlConnection connection =
        new SqlConnection(_ConnectionString))
    {
        command.CommandText = string.Format(
            "INSERT INTO {0} ({1}) VALUES({2}) ;SELECT @@IDENTITY;"
               , ReflectionHelper.GetTableName(persistentObject),
               columnNames.ToString(), values.ToString());
        command.Connection = connection;
        connection.Open();
        object result = command.ExecuteScalar();
        int newId = -1;
        if (result != null)
            int.TryParse(Convert.ToString(result), out newId);
        return newId;
    }
}
```

Obrázek 23. Implementace metody pro založení objektu v databázi

Zde stojí za povšimnutí záměna „null“ hodnoty za „DbNull.Value“ a získání nového „Id“ objektu pomocí příkazu „SELECT @@IDENTITY“. Dále stojí za zmínku vynechání vlastností „Id“ a „PersistentState“. Zde by se mělo zvážit, jakým způsobem definovat, které vlastnosti jsou perzistentní a které ne.

Implementace metody Update

```
using (SqlCommand command = new SqlCommand())
{
    StringBuilder sqlUpdatePart = new StringBuilder();

    foreach (PropertyInfo property in
        persistentObject.GetType().GetProperties())
    {
        if (property.Name.Equals("PersistentState")
            || property.Name.Equals("Id")) continue;

        if (sqlUpdatePart.Length > 0) sqlUpdatePart.Append(",");
        sqlUpdatePart.Append(property.Name).Append("=")
            .Append("@").Append(property.Name);

        SqlParameter sqlParameter = command.CreateParameter();
        sqlParameter.ParameterName = property.Name;
        object value = property.GetValue(persistentObject, null);
        if (value == null) value = DBNull.Value;
        sqlParameter.Value = value;
        command.Parameters.Add(sqlParameter);
    }

    SqlParameter parameterId = command.CreateParameter();
    parameterId.ParameterName = "Id";
    parameterId.Value = persistentObject.Id;
    command.Parameters.Add(parameterId);

    using (SqlConnection connection =
        new SqlConnection(_ConnectionString))
    {
        command.CommandText =
            string.Format("UPDATE {0} SET {1} WHERE Id=@Id"
                , ReflectionHelper.GetTableName(persistentObject)
                , sqlUpdatePart.ToString());
        command.Connection = connection;
        connection.Open();
        command.ExecuteNonQuery();
    }
}
```

Obrázek 24. Implementace metody pro aktualizaci objektu v databázi

Metoda „Update“ v podstatě vychází z metody „Insert“, pouze mění sql příkaz.

Ukázka volání metody insert a update

```
Manufacturer newManufacturer = new Manufacturer();
newManufacturer.Code = "Mercedes";
newManufacturer.Save();
newManufacturer.Code = "Mercedes Edited";
newManufacturer.Save();
```

Obrázek 25. Práce s perzistencí objektu

První „Save“ volá příkaz insert, nastaví objektu nově vygenerované „Id“ a poté nastavuje stav objektu na „Loaded“. Druhé volání metody Save zjistí, že objekt je již uložen v databázi a volá příkaz Update.

3.2.3 Implementace metody Delete

Poslední z CRUD metod je metoda „Delete“. V podstatě se neliší od ostatních metod. Řešení je stejné, až na sql příkaz.

```
using (SqlCommand command = new SqlCommand())
{
    SqlParameter parameterId = command.CreateParameter();
    parameterId.ParameterName = "Id";
    parameterId.Value = persistentObject.Id;
    command.Parameters.Add(parameterId);

    using (SqlConnection connection = new SqlConnection(_ConnectionString))
    {
        command.CommandText = string.Format("DELETE FROM {0} WHERE Id=@Id"
               , ReflectionHelper.GetTableName(persistentObject));
        command.Connection = connection;
        connection.Open();
        command.ExecuteNonQuery();
    }
}
```

Obrázek 26. Implementace mazání objektu z databáze

Jak vidíme z ukázky, kód je velmi jednoduchý.

3.3 Další požadavky

Určitě jste zaregistrovali, že tento framework zatím neřeší žádné vazby a vztahy. Nyní je potřeba rozhodnout, jakým způsobem budou tyto informace uchovávány. Buďto budou vytvořeny třídy pro konfiguraci, kde budeme uchovávat seznam perzistentních tříd,

vlastností a vztahů nebo to vyřešíme pomocí atributů. Práce s atributy je jistě komfortnější a ušetří nám čas při vývoji. V této kapitole budou popsány základy práce s atributy a bude vysvětleno, jak budou v budoucím frameworku používány.

3.3.1 Atributy

A tributy budou povědomé vývojářům, kteří v jazyku C++ píší komponenty COM (na základě jejich použití v jazyku COM IDL - Interface Definition Language - společnosti Microsoft). Atributy původně vznikly proto, aby poskytovaly dodatečné informace o některých položkách v programu, které by mohl využít kompilátor.

Podpora atributů je k dispozici v platformě .NET a nyní tedy i v jazycích C++, C# a Visual Basic 2008. Atributy v prostředí .NET však nově nabízejí mechanismus, s jehož pomocí můžete ve svém zdrojovém kódu definovat vlastní atributy. Spolu s těmito uživatelsky definovanými atributy budou umístěna metadata pro odpovídající datové typy nebo metody. Tuto vlastnost lze využít například kvůli dokumentaci, kdy mohou atributy spolu s technologií reflexe zajišťovat její vygenerování. V souladu s filozofií nezávislosti jazyků v technologii .NET, lze také atributy definovat ve zdrojovém kódu v jednom jazyce a číst je může kód, který je napsán v jiném jazyce. [1]

3.3.1.1 Vlastní atributy

Platforma .NET Framework také dovoluje definovat vlastní atributy. Tyto atributy pochopitelně nijak neovlivní proces překladu, protože překladač o nich standardně nemá žádné informace. Když však vlastní atributy aplikujete na součásti programu, uloží se jako metadata do přeloženého sestavení. Sama o sobě mohou být tato metadata užitečná pro účely dokumentace. Skutečná síla atributů se však projeví díky reflexi, jež umožňuje programu tato metadata načíst a za běhu se podle nich rozhodovat. Z toho vyplývá, že vlastní atributy, které sami definujete, mohou mít přímý vliv na činnost vašeho kódu. Pomocí vlastních atributů, lze například povolit deklarativní bezpečnostní kontroly přístupu ke kódu pro třídy vlastních oprávnění, připojit k prvkům programu informace, se kterými mohou pracovat testovací nástroje, nebo je využít při vývoji rozšiřitelné architektury, která dovoluje načítání doplňkových či jiných modulů. [1]

3.3.2 Návrh řešení

Po definování pojmu atribut bude ukázáno a vysvětleno jeho praktické využití. Je potřeba definovat tyto základní informace:

- Označení tříd a vlastností jako perzistentní.
- Typ vztahu (Asociace, kompozice) a jeho pojmenování.

U atributů je možné definovat „AttributeUsage“. Tato vlastnost určuje, kde může být atribut použit. Slouží v podstatě pro kontrolu, že bude atribut správně použit. Například atribut s „AttributUsage = Class“ nelze přidat nad vlastnost. Pokud bychom atribut chybně použili, překladač nám zahlásí chybu a řešení nepůjde sestavit. „AtributUsage“ může nabývat následujících hodnot : All, „Assembly, Class, Constructor, Delegate, Enum, Event, Field, GenericParameter (pouze .NET 2.0), Interface, Method, Module, Parameter, Vlastnost, ReturnValue a Struct.

Dále se bude muset implementovat pro dané vztahy také chování. To platí hlavně u vztahu typu kompozice. Z definice kompozice vyplývá, že nadřízený objekt se musí postarat o ten podřízený. Proto musí být do nahrání objektu z databáze, jeho uložení a smazání přidán kód, který bude zjišťovat práci s podřízenými objekty.

V metodě „GetObject“ bude tedy definováno chování pro podřízené objekty. To by teoreticky nemuselo být tak složité, protože jsou všechny perzistentní objekty děděné ze třídy „PersistentObject“, tudíž je známo jejich „Id“.

V metodě „Delete“ bude všechny podřízené objekty volána metodu „Delete“. A u metody „Save“ bude situace stejná. Kód bude muset být velmi odolný proti zacyklení, protože objekty mohou mít na sebe vzájemnou referenci.

Jistě bude na náš framework kladeno spoustu požadavků, protože aplikace které musí podporovat, mohou být různorodé a mohou mít své specifické požadavky. Nyní bude ukončena teoretická část práce. Dále se budeme věnovat tvorbě perzistentního frameworku s názvem „QuickPersistor“.

II. PRAKTICKÁ ČÁST

4 POŽADAVKY NA PERZISTENTNÍ FRAMEWORK.

V této kapitole budou definovány požadavky na ideální perzistentní framework. To v žádném případě neznamená, že musí být všechny požadavky zařazeny do vývoje. Navíc se bude vycházet z agilního iterativního způsobu vývoje (MSF-Microsoft solution framework, SCRUM, atp.). Není zde prostor popisovat agilní metodiky a rozdíly oproti vodopádovému modelu, takže jen ve zkratce bude popsáno, jak bude tento vývoj probíhat. Nemůžeme samozřejmě říci, že budeme vycházet z určité agilní metodiky, protože tyto metodiky jsou určeny pro tým lidí a ne pro jednotlivce. Postup vývoje bude následující:

- Definice seznamu požadavků.
- Jednoduchá analýza v podobě scénářů (popřípadě User stories).
- Vývoj.
- Testování.
- Výběr požadavků pro další verzi (Mohou být zařazeny i požadavky z předchozí verze, které byly nedokonale implementovány nebo rozšířeny).

4.1 Požadavky na perzistenci objektů

Framework bude vytvářen hlavně z důvodů zrychlení práce programátora business aplikací a aplikací, které uchovávají svá data v relační databázi. Použití by mělo být co nejvíce intuitivní a nemělo by být podmíněno složitou nebo nelogickou konfigurací. Perzistor se bude snažit z kódu vytěžit pokud možno co nejvíce informací. Výchozí nastavení bude umožňovat rychlou implementaci. Nyní bude definován seznam požadavků na framework.

- Ukládání primitivních datových typů do databáze.
- Vyhledání a naplnění objektu z databáze dle Id (Primitivní datové typy).
- Výmaz jednotlivých objektů z databáze.
- Vyhledání a naplnění kolekce objektů z databáze dle kritérií.
- Lazy load kolekce.
- Označení vlastností jako neperzistentních.
- Možnost vlastní implementace perzistentních metod.

- Možnost perzistovat dědičnost.
- Definování typu vztahu u vazby 1:1 a 1:N.
- Implementace práce s kolekcemi objektů.
- Ukládání komplexních datových typů do databáze.
- Naplnění komplexních datových typů z databáze.
- Mazání komplexních datových typů v databázi.
- Podpora windows forms bindingu.
- Tvorba jazyku pro objektové dotazy a následná implementace parseru na sql dotazy.
- Možnost definovat další atributy nad vlastnostmi („Not null“, datový typ).

4.2 Požadavky na generátor sql skriptů z modelu

Pokud je vytvořen model a jsou definovány vztahy tak podle pravidel objektově relačního mapování jsou v podstatě k dispozici veškeré informace o schématu databáze. Proč tedy těchto informací nevyužít pro generování a aktualizaci databázových objektů? Nyní zde bude uveden seznam požadavků na generování a aktualizaci schéma databáze.

- Vytvoření databáze (pokud neexistuje) z připojovacího řetězce.
- Vytvoření tabulek dle tříd.
- Vytvoření sloupců dle datového typu a dalších informací v atributu vlastnosti.
- Přidávání nových sloupců do tabulek.
- Vytvoření cizích a primárních klíčů.

5 ANALÝZA

V této kapitole budou vypsány vybrané případy užití a jejich scénáře. Popis všech případů užití je nad rámec této práce.

5.1 UC001 - Uložení primitivních datových typů do databáze

Vstupní podmínky

Programátor má nad třídou definovány atributy, které určují, zda se jedná o perzistentní třídu. Nad vlastnostmi, u kterých nechce, aby byly perzistentní, má speciální atribut. Má instanci dané třídy naplněnou daty.

Hlavní úspěšný scénář

1. Programátor zavolá na objektu metodu pro uložení do databáze.
2. Systém se postará o zavolání metody „Insert“ v případě, že je objekt nový. Pokud je objekt již nahrán z databáze, postará se o zavolání metody „Update“. Nastaví objektu stav „Loaded“. Systém ignoruje vlastnosti typu nebo podtypu „PersistentObject“.

5.2 UC002 - Vyhledání a naplnění objektu z databáze dle Id (Primitivní datové typy)

Hlavní úspěšný scénář

1. Programátor zavolá metodu pro vydání objektu z databáze a předá ji parametr „Id“.
2. Systém vyhledá v databázi řádek tabulky s daným „Id“, vytvoří novou instanci objektu a naplní ho daty. Systém ignoruje vlastnosti typu nebo podtypu „PersistentObject“.

5.3 UC003 - Výmaz objektu z databáze

Vstupní podmínky

Programátor má perzistentní objekt ve stavu „Loaded“.

Hlavní úspěšný scénář

1. Programátor zavolá na objektu metodu pro výmaz z databáze.
2. Systém vyhledá v databázi řádek s daným „Id“, provede výmaz z databáze a nastaví objektu stav „Deleted“.

5.4 UC004 - Vyhledání a naplnění kolekce objektů z databáze dle kritérií

Hlavní úspěšný scénář

1. Programátor zavolá metodu pro vydání generického listu objektů z databáze. Předá ji kritéria a řazení.
2. Systém si vyžádá z databáze seznam Id dle kritérií a řazení. Pro každé Id zavolá UC002.

5.5 UC005 - Lazy load kolekce

Hlavní úspěšný scénář

1. Programátor zavolá metodu pro vydání generického listu objektů z databáze. Předá ji kritéria a řazení.
2. Systém kompletně implementuje všechny vlastnosti a metody kolekce. Systém si vyžádá z databáze seznam „Id“ dle kritérií a řazení a uchová si je v privátním seznamu. Objekty plní z databáze, až při prvním použití (Indexer, cyklus atp).

5.6 UC006 - Označení vlastností jako neperzistentních

Hlavní úspěšný scénář

1. Programátor přidá nad vlastnost atribut, který určuje, že má být ignorován pro perzistenci.
2. Systém při CRUD operacích bude tuto vlastnost ignorovat.

5.7 UC007 - Vlastní implementace perzistentních metod

Hlavní úspěšný scénář

1. Programátor může přepsat na objektu CRUD metody.
2. Systém bude prioritně volat tyto metody.

5.8 UC008 - Perzistence dědičnosti

Hlavní úspěšný scénář

1. Programátor zavolá metodu pro naplnění dat do objektu z databáze, který existuje ve stromu dědičnosti.
2. Systém zjistí, že se jedná o dědičnost a dále zjistí, jakého typu je objekt uložený v databázi. (Společně s dědičností, musí být perzistována i informace o skutečném typu objektu). Vytvoří instanci správného typu a naplní ji daty z jedné nebo více tabulek dle nastavení.

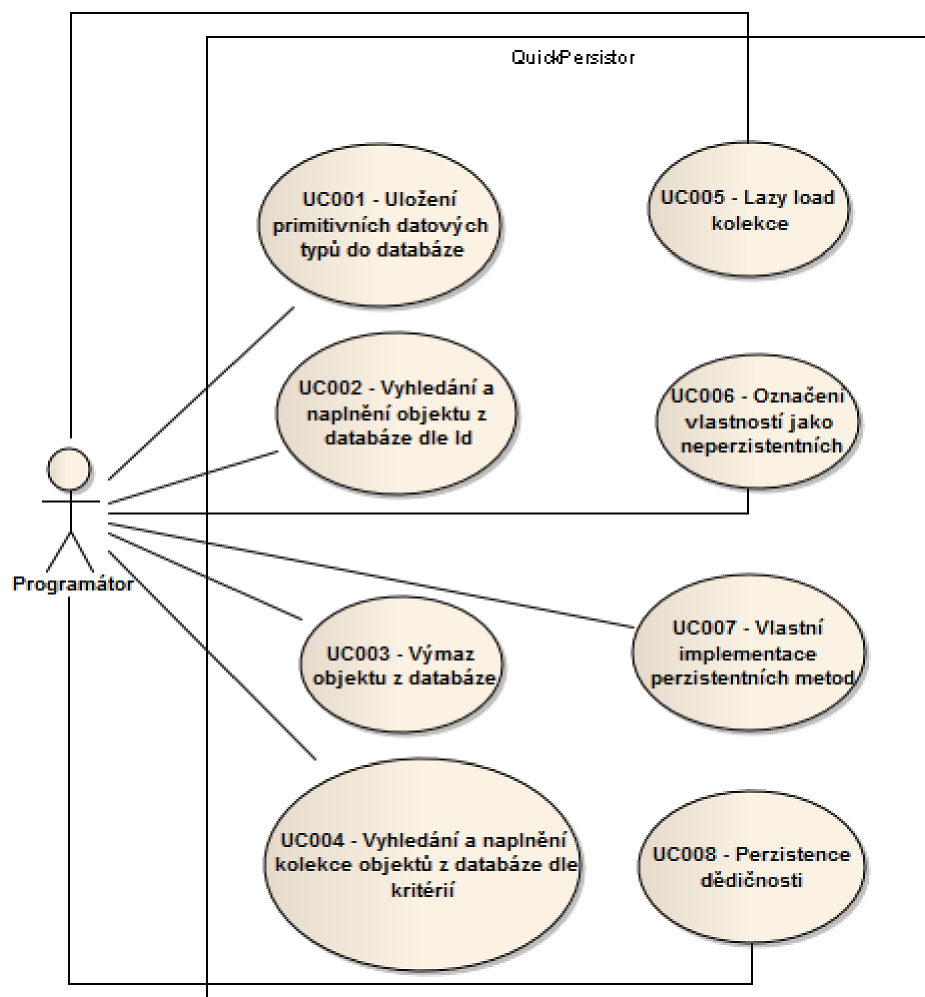
Alternativní scénáře

1a Programátor zavolá metodu pro uložení objektu do databáze.

1. Programátor zavolá metodu pro uložení dat objektu do databáze, který existuje ve stromu dědičnosti.
2. Systém zjistí, že se jedná o dědičnost a zajistí uložení dat do databáze dle zvoleného mapování (1:1 nebo 1:N) a dále zjistí, uložení informace o skutečném typu do tabulky s hierarchicky nejvyšším předkem.

1b Programátor zavolá metodu pro vymazání objektu z databáze.

1. Programátor zavolá metodu pro vymazání objektu z databáze, který existuje ve stromu dědičnosti.
2. Systém zjistí, že se jedná o dědičnost a zajistí výmaz všech dat ze všech tabulek ve stromu dědičnosti, dle zvoleného mapování (1:1 nebo 1:N).



Obrázek 27. Use Case model

6 REALIZACE

6.1 Projekt QuickPersistor

Nyní jsme se konečně dostali k praktickému vytvoření perzistentního frameworku QuickPerzistor. Tato a následující kapitola bude plná úvah a fragmentů řešení. Popsat zde celý framework by přesahovalo možnosti a rozsah této práce.

6.1.1 Vize

Bude vytvořen framework, který rapidně urychlí práci na nových projektech. Nahradí zdlouhavé a chybové psaní perzistentní vrstvy. Framework bude umožňovat jednoduchou implementaci dalších databázových strojů a objektové dotazy do databáze.

6.2 Definice rozšiřujících informací pro perzistenci

Jak již bylo v předchozích kapitolách řečeno, bude potřeba uchovávat rozšiřující informace pro perzistenci objektů do databáze. Tyto rozšiřující informace budou řešeny pomocí atributů platformy .NET. Nyní si definujme parametry, jejich vlastnosti a rozsah použití.

6.2.1 Atribut Peristent

Jedná se o stěžejní atribut. Jeho použití říká, že daná třída, vlastnost nebo výčet je perzistentní. Na jednu stranu je nevýhoda sdílet jeden atribut mezi třídou a vlastností, protože může obsahovat určité atributy, které pro jednu nebo druhou entitu nemusí být použitelné. Na druhou stranu se tak výrazně zvyšuje čitelnost a jednoduchost kódu. Třídám a vlastnostem, které jsou nazývány perzistentní, bude přidán atribut Persistent.

6.2.1.1 Rozsah použití

Třída (Class), vlastnost (Property), Výčet (Enum) – není implementován.

6.2.1.2 Vlastnosti a jejich význam

Name: Tato vlastnost určuje název databázového objektu. U třídy se jedná o název tabulky a u vlastnosti název sloupce v tabulce.

Length: Tato vlastnost určuje délku u sloupců typu omezený text.

ExplicitDatabaseColumnType: Tato vlastnost určuje použití vlastního typu pro sloupec místo implicitního (Vybraného frameworkem dle CLR typu). Nejedná se o konkrétní název například NVarchar(MAX), protože by to porušilo nezávislost na databázi. Implementovány jsou tyto typy: Default, UnlimitedText, UnicodeText, UnicodeUnlimitedText.

6.2.2 Atribut NonPersistent

Tento atribut určený pro třídu a vlastnost určuje, že daná třída nebo vlastnost není perzistentní a nebude proto reflektována do databáze. Atribut nemá žádné vlastnosti.

6.2.2.1 Rozsah použití

Třída (Class), vlastnost (Property).

6.2.3 Atribut Inheritance

Název atributu napovídá, že se bude jednat o mapování dědičnosti do relační databáze. Tento atribut je výhradně určen pro třídy.

6.2.3.1 Rozsah použití

Třída (Class).

6.2.3.2 Vlastnosti a jejich význam

- **InheritanceMapType** – Tento výčet určuje typ mapování dědičnosti. Vztah mezi třídou a tabulkou. Zatím bude pouze implementován typ OwnTable (Mapování 1:1).

6.2.4 Atribut Association

Tento atribut definuje typ vazby z pohledu objektově relačního mapování. Je určen výhradně pro vlastnosti děděné z bázového perzistentního objektu a kolekci těchto objektů. Multiplicitu zde definuje typ. Pokud třída implementuje rozhranní nebo dědí z objektu představujícího kolekci z pohledu OOP, bude multiplicita 1:N, v opačném případě 1:1. Tento atribut v sobě zahrnuje vztah asociace i kompozice. Rozlišení o jaký typ vztahu se definuje parametrem.

6.2.4.1 Rozsah použití

Vlastnost (Property).

6.2.4.2 Vlastnosti a jejich význam

AssociationType: Tento výčet určuje o jaký typ asociace se jedná. Může nabývat hodnot „Association“ (Jednoduchá asociace) a „Composition“ (Kompozice). Jedná se o velmi důležitý atribut z pohledu perzistence, na jeho základě se určuje chování nadřízeného objektu k podřízenému. Například vztah kompozice znamená, že nadřízený objekt se musí postarat o vznik i zánik svých podřízených. Nastavování tohoto parametru musí být velice dobře zváženo. V případě chybného použití se může například stát, že se budou objekty odmazávat z databáze, aniž by to bylo vyžadováno.

Name: Jedná se o další důležitou vlastnost, která slouží pro identifikaci vztahu mezi třídami. Nejlépe si ji vysvětlíme na příkladu. Nutno upozornit, že se jedná pouze o ukázkový příklad, který by se pravděpodobně v případě reálného systému řešil jinak. Představme si následující scénář. Zákazník chce evidovat v systému E-Shop objednávky. U každé objednávky chce kromě jiného držet dvě adresy. Adresu doručovací a fakturační. Ze zadání jednoduše odvodíme třídu „Objednávka“ a „Adresa“. Máme tedy dvě třídy, nyní si musíme definovat vztahy. Ve třídě „Objednávka“ si vytvoříme kromě jiného dvě vlastnosti typu adresa, první se jménem „Doručovací“ a druhou „Fakturační“. Dále definujeme atribut „Association“ typu kompozice. Je známo, že u kompozice zná podřízený objekt svůj nadřízený. Proto v podřízeném objektu definujeme vlastnosti typu „ObjednávkaKDoručovací“ a „ObjednávkaKFakturační“. Nyní musíme definovat unikátní název vztahu aby framework mohl správně naplnit objekty daty. Proto vyplníme do vlastnosti „Name“ tento název.

StorageName: Tato vlastnost určuje název sloupce v databázi. V případě že není vyplněna, bude pro název databázového použit název vlastnosti.

MappingType: Tato vlastnost určuje, jak se budou objekty mapovat do tabulek. Tzn. do tabulky třídy, ve které je vlastnost definována nebo do tabulky nadřízeného objektu. V této verzi je implementována pouze možnost „OwnTable“ (vlastní tabulka).

6.3 Implementace komunikace s databází

V této kapitole bude popsáno jakým způsobem má QuickPersistor tvořenu databázovou část a jak je připraven pro implementaci dalších databázových strojů.

6.3.1 Rozhraní IPersistor

Základ tvoří rozhraní „IPersistor“, které obsahuje vše, co musí konkrétní databázová vrstva implementovat. V případě, že je vyžadována implementace dalšího databázového stroje, stačí vytvořit novou třídu, která implementuje rozhraní „IPersistor“ a napsat metody pro konkrétní databázi. Nyní budou popsány nejdůležitější metody tohoto rozhraní.

Initialize: Tato metoda se volá při spuštění aplikace a jako parametry obdrží připojovací řetězec k databázi. V tento okamžik by si konkrétní implementace měla uchovat řetězec v paměti pro tvorbu nových instancí konkrétní třídy „Connection“.

Insert: Metoda slouží pro založení nového záznamu. Jako vstupní parametr má název tabulky a kolekci objektů typu „FieldSchema“. Tato třída bude pospána v dalších kapitolách. Jako výstupní parametr je nově vygenerovaný identifikátor „Id“.

Update: Metoda slouží pro aktualizaci již založeného záznamu v databázi. Jako vstupní parametr opět přebírá název tabulky, kolekci objektů typu „FieldSchema“ a „Id“.

Load: Metoda slouží pro naplnění objektu z databáze. Jako vstupní parametr obsahuje kolekci inicializovaných objektů typu „FieldSchema“, kolekci názvů tabulek a „Id“. Kolekce tabulek znamená, že tato metoda dokáže nahrávat kompletní dědičnost. Výstupní parametr je typu „bool“ a určuje, zda se podařilo objekt kompletně naplnit z databáze.

Delete: Metoda slouží pro odstranění objektu z databáze. Jako vstupní parametr má „Id“ a název tabulky.

GetColId: Tato metoda slouží pro naplnění kolekce identifikátorů. Rozhraní obsahuje různá přetížení této metody. Asi nejsložitější přetížení obsahuje jako parametry kolekci tabulek, podmínku a řazení.

Rozhraní „IPersistor“ není příliš složité, proto implementace dalších databázových strojů, není nejnáročnější operací z celého frameworku. Nejnáročnější je implementace správného chování objektů, jejich vztahů a rozložení na atomické operace definované ve třídách implementující toto rozraní.

6.3.2 Implementace konkrétního databázového stroje

V předchozí kapitole bylo popsáno rozhraní, které musí implementovat každá třída pracující s konkrétním databázovým strojem. Konkrétní databázové poskytovatele (Providery) vytváříme v projektu QuickPersistor ve jmenném prostoru „vknet.QuickPersistor.Data.Provider“.

Třída konkrétního poskytovatele bude založena ve jmenném prostoru definovaného v předchozí kapitole. Tato třída bude instanční a bude implementovat rozhranní „IPersistor“. S implementací si můžeme ve vývojovém prostředí Microsoft Visual Studio pomoci příkazem „Implementace rozhraní“. Tento příkaz vytvoří potřebné metody a vlastnosti s tím, že do těla vloží vyvolání výjimky typu „System.NotImplementedException“. Již nyní můžeme předpokládat, že bude nutno někde uchovávat informace o připojovacím řetězci a metody inicializace. Z tohoto důvodu bude tedy založena třída „Persistor“ ve jmenném prostoru „vknet.QuickPersistor.Data“. Tato abstraktní třída bude předkem pro všechny konkrétní implementace databázových poskytovatelů. Kromě vlastnosti uchovávající připojovací řetězec k databázi bude implementovat návrhové vzory „Signleton“ a „Factory“. „Singleton“ se postará o uchování pouze jedné jediné instance konkrétního persistoru a „Factory“ pro vytvoření konkrétní instance na základě konfigurace. V případě, že je vyžadována práce s datovou vrstvou, zavolá se „get“ vlastnosti Persistor. V případě, že dosud není „zrozena“ instance konkrétního poskytovatele, zavolá se metoda, která vytvoří na základě konfigurace správnou instanci. Konfigurace může být ruční (vyplnění informací o typu poskytovatele atp.) nebo automatická (načtení předem definovaných atributů z konfiguračního souboru).

Bylo definováno rozhraní „IPersistor“ a třída předka „Persistor“ společná pro všechny konkrétní poskytovatele. Nyní je potřeba provést implementaci jednotlivých metod.

6.3.3 Implementace poskytovatele pro Microsoft Sql Server

Jako první bude vytvořen poskytovatel pro Microsoft Sql Server. Bude založena instanční veřejná třída MsSqlPersistor ve jmenném prostoru „vknet.QuickPersistor.DataProvider“. Z předchozí kapitoly vyplývá, že je nutné implementovat rozhraní „IPersistor“ a podědit třídu ze třídy „Persistor“. Nyní zbývá implementovat jednotlivé metody rozhraní, pomocné metody, proměnné a konstanty.

6.3.3.1 Důležité části perzistoru

Je vhodné si nadefinovat konstanty pro čísla databázových chyb. Tyto chyby bohužel nemají v C# své třídy výjimek. Budou definovány tyto konstanty:

DATABASE_DOES_NOT_EXIST_ERROR = 4060

TABLE_DOES_NOT_EXIST_ERROR = 208

KEY_IS_EXIST_ERROR = 2714;

Na tyto chyby budeme v případě jejich vyvolání řádně reagovat. Například pokud při databázové akci „SELECT COUNT" nastane výjimka neexistující tabulky, budeme vracet 0 záznamů a tuto chybu ignorovat.

```
catch (SqlException ex)
{
    if (ex.Number == TABLE_DOES_NOT_EXIST_ERROR)
    {
        return 0;
    }
    else
    {
        throw;
    }

}
```

Obrázek 28. Odchycení databázové výjimky

V dalším kroku bude naprogramována metoda „Initialize". Tato metoda bude na třídě předka přiřazovat připojovací řetězec do vlastnosti „ConnectionString". Ve třídě potomka bude krom jiného kontrolovat, existenci databáze.

Za zmínku stojí metoda „Insert". V této metodě se budeme potýkat s problémem, jak správně získat identifikátor nově založeného objektu. Řešením tohoto problému je volat najednou dva příkazy. První se samotným příkazem „Insert" ukončený středníkem a druhý pro získání identifikátoru („;SELECT @@IDENTITY"). Akci provedeme zavoláním metody „ExecuteScalar". Tato metoda provede příkazy a vrátí první sloupec prvního řádku. V našem případě vygenerovaný identifikátor.

Metoda „Load" s parametrem přebírajícím kolekci tabulek slouží pro získání všech dat z celé hierarchie dědičnosti. Pokud třída neobsahuje dědičnost, bude kolekce obsahovat pouze jeden záznam. Příkaz „SELECT" pro dědičnost bude generován dle pravidla, že tabulka v nejvyšší úrovni hierarchie bude mít zapnuto automatické generování „Id".

Podřízené tabulky budou mít sloupec „Id“ s vypnutou funkcí automatického generování. Sloupec „Id“ podřízené tabulky přebírá hodnotu z tabulky nadřízené.

Metoda „Count“ bude používat stejnojmennou sql funkci. Pro vykonání příkazu bude použita metoda třídy SqlCommand „ExecuteScalar“.

Dále zde existují následující pomocné metody:

GetParameter. Vytvoří novou instanci třídy SqlParametr a naplní ji jménem sloupce.

SetParameterValue. Naplní sql parametr hodnotou. Navíc správně převede hodnotu „null“ na hodnotu „DBNull.Value“.

GetSelectCommandText. Tato metoda vygeneruje dle názvu tabulky příkaz „SELECT“.

IsTableExist. Tato metoda ověřuje, zda existuje tabulka. Toto ověření dělá na základě metody „GetSchema“ třídy „SqlConnection“.

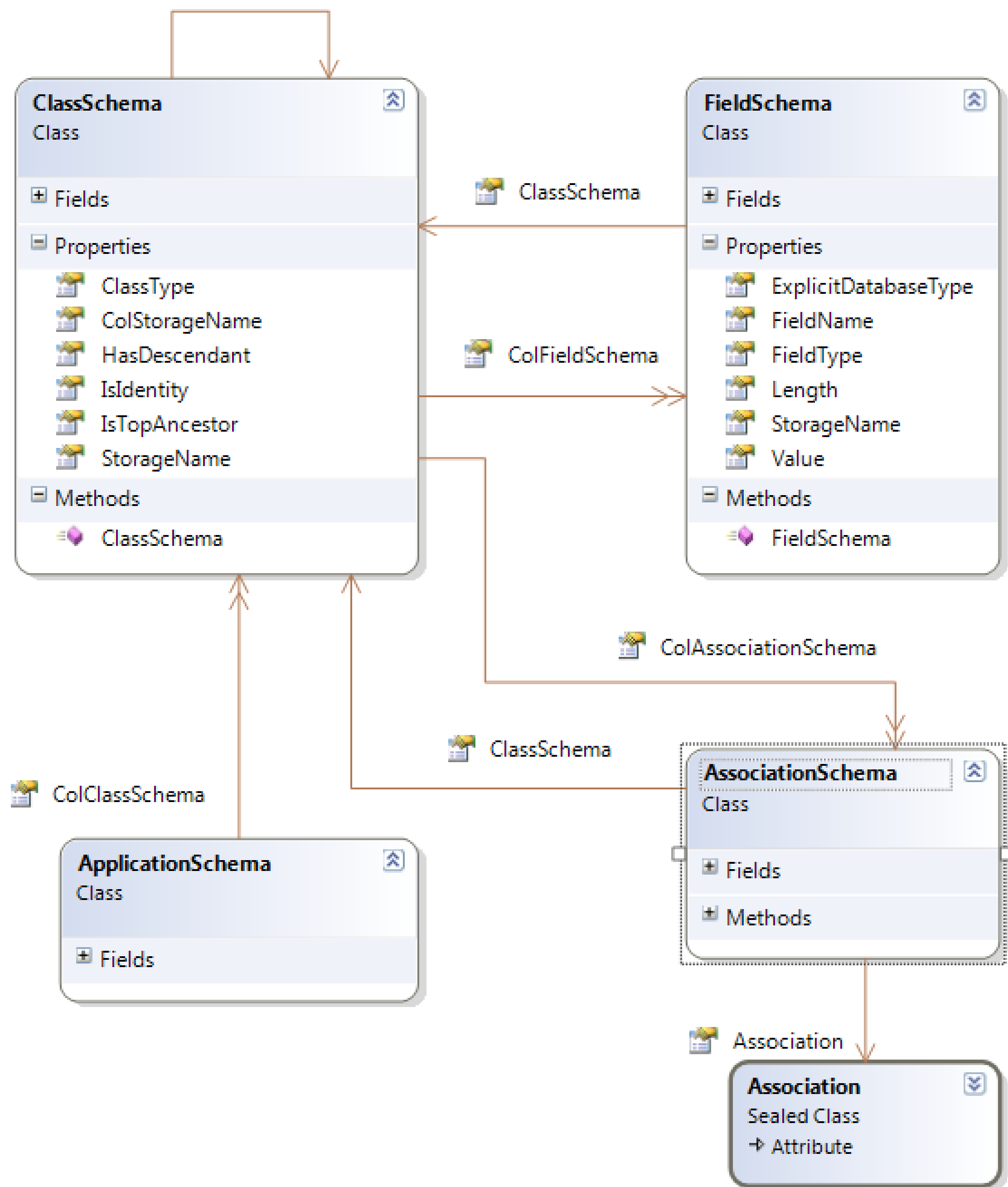
GetSqlCommand. Tato metoda vytvoří novou instanci třídy SqlCommand navíc nastaví parametr „CommandTimeout“ na hodnotu definovanou v konfiguraci aplikace. Toto nastavení slouží pro aplikace, které používají příliš složité příkazy. V tomto případě se osvědčuje výhoda Frameworku, jenž umožňuje globální nastavení.

6.4 Vytvoření schéma aplikace pro komunikaci mezi vrstvami

Jak již bylo naznačeno v kapitole věnované komunikaci s databází, je použito speciálních tříd pro předávání parametrů. V případě že by nebyl použit framework tak by bylo nutné v metodách persistoru („Load“, „Insert“ a „Update“) definovat tolik vstupních parametr, kolik chceme uložit nebo nahrát z databáze. Další možností, také doporučovanou například v knize Martina Fowlera Refaktoring „Zlepšení existujícího kódu“ [2] by bylo použití speciálních tříd místo desítek vstupních nebo výstupních parametrů. Tedy pro předávání parametrů a další pomocné účely byl vytvořen následující objektový graf.

6.4.1 Realizace jmenného prostoru ApplicationSchema

Požadavkem je jednoduše zachytit schéma aplikace. Nejjednodušší bude si nejprve ukázat diagram tříd.



Obrázek 29. Diagram tříd zachycující schéma databáze

Diagram mapuje do tříd strukturu aplikace.

6.4.1.1 Popis schéma

- Třída „ApplicationSchema“ představuje v podstatě virtuální aplikaci, která obsahuje kolekci „ClassSchema“.
- Třída ClassSchema představuje v naší virtuální aplikaci třídu. Obsahuje následující vlastnosti:

Type: Uchovává datový typ skutečné třídy.

StorageName: Určuje název databázového objektu (Tabulky).

IsTopAncestor: Určuje informaci o tom, zda je třída nejvyšší předek z hierarchie dědičnosti.

HasDescendant: Určuje, zda třída obsahuje potomky.

IsIdentity: Určuje, zda tabulka obsahuje sloupec „Id“ se zapnutou funkcí autoidentity neboli generování identifikátorů.

BaseClassSchema: Mapuje dědičnost tříd do instancí typu ClassSchema. Uchovává informace o nadřízené třídě z pohledu dědičnosti.

ColStorageName: Je to vlastnost pouze ke čtení a vrací seznam všech tabulek z hierarchie dědičnosti.

ColFieldSchema: Uchovává seznam vlastností (atributů, sloupců) dané třídy.

ColAssociationSchema: Uchovává seznam vazeb mezi třídami.

- Třída FieldSchema představuje vlastnost virtuální třídy. Obsahuje tyto vlastnosti:

  FieldName: Určuje název vlastnosti (property) reálné třídy.

  FieldType: Určuje datový typ vlastnosti.

  Value: Uchovává hodnotu z databáze nebo určenou pro zapsání do databáze.

  StorageName: Určuje název sloupce v databázi.

  ExplicitDatabaseType: Určuje, že bude použit jiný typ než explicitně přiřazený. Například místo neomezeného textu je požadováno použití neomezeného textu pro uchování unicode.

  Length. Určuje délku sloupce obsahující řetězec (NVarchar, VarChar, atp.).

- Třída AssociationSchema uchovává informace o vztahu mezi třídami. Obsahuje pouze jednu vlastnost.

  Association. Jedná se v podstatě o stejný atribut, který je definovaný na vztahu mezi třídami. Tento atribut je popsán v předchozích kapitolách. Zopakujeme si jen že obsahuje veškeré informace o daném vztahu.

- Statická třída ReflectionHelper obsahuje spoustu pomocných metod pro práci s reflexemi. Jak bylo řečeno na začátku reflexe jsou základem tohoto frameworku. Popsat všechny metody je nad rámec této práce.

- Poslední třídou ze jmenného prostoru „vknet.QuickPersistor.ApplicationSchema“ je Mapper. Tato třída obsahuje logiku, která z reálné aplikace vytvoří výše popsané schéma aplikace. Napsání této třídy bylo velmi složité. Musely se implementovat všechna pravidla pro mapování do relační databáze. Navíc jsou zde použity pokročilé metody reflexe. Velmi obezřetně se musí ošetřovat vzájemná reference tříd. Pokud nebude vše řádně ošetřeno, může dojít k zacyklení a přetečení zásobníku. Celé vytvoření schéma aplikace musí proběhnout jen jednou a velmi rychle při spuštění aplikace. Zdržení při startu aplikace je další daní při použití frameworku. V QuickPeristoru bylo dosaženo velmi dobrých výsledků při inicializaci aplikace. Nyní budou velmi zhruba popsány nejdůležitější metody této třídy.

  GetApplicationSchema: Tato metoda se volá pro inicializaci aplikace a má za úkol kompletně naplnit výše definované struktury (ApplicationSchema, ClassSchema, FieldSchema,..). Na začátku načte všechna potřebná sestavení pro analýzu aplikace pomocí metody GetAssembliesReferringTo s parametrem „vknet.QuickPersistor“. Postupně prochází jednotlivé datové typy, a pokud jsou děděny z PersistentBase tak je analyzuje pomocí metody GetClassSchema. Výsledkem této metody je kompletně naplněné schéma databáze.

  GetAssembliesReferringTo: Jedná se o klíčovou metodu při inicializaci aplikace. Tato metoda projde všechna sestavení (Assembly), která mají referenci na projekt vknet.QuickPersistor. Každé toto sestavení přidá do kolekce, která bude dále analyzovaná.

  GetClassSchemaFromCache: Tato metoda vydá z paměti schéma tříd dle datového typu. Důležité je volat v průběhu aplikace pouze tuto metodu. První metoda GetClassSchema je totiž velmi náročná operace, která se volá na začátku aplikace.

6.5 Implementace tříd předka perzistentních objektů

Jak bylo napsáno v úvodu této práce, musí být perzistentní framework jednoduchý na implementaci a musí šetřit čas na vývoj datové vrstvy. Z důvodů rychlé implementace perzistence na objektech vznikla myšlenka řešení pomocí dědičnosti ze základní třídy. Nyní bude popsáno, které třídy tvoří základ perzistence objektů a jakou obsahují nejdůležitější funkcionalitu.

6.5.1 Třída PersistentBase

Tato abstraktní třída tvoří základní funkcionality perzistence. Je zodpovědná za základní operace (Save, Delete,atp.). Tato třída zatím neobsahuje vlastnost „Id“ a proto je možné do budoucna implementovat i objekty, které obsahují jiný typ identifikátoru. Například složený klíč nebo Guid. V této kapitole bude popsána nejdůležitější funkcionalita této třídy.

Každá instance třídy obsahuje instanci konkrétní datové vrstvy, jež implementuje „IPersistor“. O zrod správného typu je zodpovědný návrhový vzor „Factory“. Tím že má každý objekt svou instanci perzistoru je vyřešena konkurence ve více vláknové aplikaci.

Pro možnost reagovat na databázové operace je zde vytvořena událost. Pro tuto událost je definován event handler typu „PersistentActionEventHandler“. Tento handler předepisuje dva parametry „source“ typu object a „e“ typu PersistentActionEventArgs. Využití této události je jistě mnoho. Například možnost validovat před uložením, mazáním atp. Určitě najde své využití i při tvorbě GUI.

Tato třída obsahuje další pomocné metody, které není nutno popisovat.

Je nutné zmínit i pravidlo bez, kterého se neobejde žádný perzistentní framework. Jedná se o nutnost tříd implementovat bezparametrický konstruktor. Toto pravidlo sice narušuje čistotu objektového návrhu, ale je to jediná možnost jak programově vytvářet instance tříd. V další kapitole si popíšeme metody pro práci s perzistencí objektů. Tímto bude kapitola o třídě PersistentBase ukončena.

6.5.1.1 Metody pro práci s perzistencí objektů

- **Load** – Jedná se o metodu, která danou instanci objektu naplní daty z databáze na základě vyplněného „Id“. Tato metoda je na rozdíl od té ve třídě „Persistor“

komplexní a naplní kompletní objektový graf. Zkusíme si zjednodušené popsat, jak postupuje.

Vyžádá si konkrétní typ perzistoru (Návrhový vzor „Factory“).

Vyžádá si od třídy „Mapper“ schéma třídy včetně vazeb a vlastností. (GetClassSchemaFromChache).

Přidá odkaz na sebe do session aplikace pokud v ní neexistuje. Session je implementována ve tříde „DataSession“ a bude popsána dále.

Vybere data z databáze a prochází jednotlivé vztahy, pro tyto vztahy volá v rekurzi „Load“. Tento algoritmus je velmi složitý a nebude zde detailně popsán. Jsou v něm řešeny všechny typy mapování.

- **Save** – Metoda se postará o uložení objektového grafu dle pravidel pro mapování objektů do relační databáze. Výjimku tvoří uložení objektů, které jsou ve vztahu jednoduché asociaci a nejsou uloženy do databáze. Tyto objekty budou uloženy do databáze, aby se předešlo chybám opomenutí volání „Save“ na asociovaných objektech. Opět se jedná o velmi složitý algoritmu, který zde nemůže být popsán.
- **Delete** – Metoda se postará o odstranění objektu a jeho podřízených objektů z databáze. Tento výmaz musí proběhnout ve správném pořadí, nejprve musí být smazány podřízené objekty a teprve poté nadřízené. V opačném případě by nastal konflikt cizích klíčů v databázi. Výjimku zde tvoří vztah kompozice 1:1, v tomto mapování si nese nadřízený objekt „Id“ podřízeného. Tudíž před vymazáním podřízeného objektu musí být „Id“ ve vazbě nastaveno na hodnotu „null“. Teprve poté bude smazán podřízený objekt. Ani u této metody nebudeme popisovat detailně algoritmus.

6.5.2 Třída StoredObject

Tato třída rozšiřuje předka o důležité vlastnosti a metody pro urychlení vývoje aplikací. Pokud programátor použije předka, musí spoustu vlastností a metod implementovat. Nyní budou vyjmenovány nejdůležitější části této třídy.

Mezi ty nejdůležitější vlastnosti této třídy patří stav perzistence. Tento stav může nabývat následujících hodnot:

OnlyId: Tento stav určuje, že je objekt zatím není nahrán z databáze, ale má již vyplněnu hodnotu identifikátoru pro budoucí vyhledání a naplnění z databáze.

Loaded: Určuje, že objekt již byl naplněn daty z databáze.

NewIsSaving: Objekt je nový a zároveň probíhá ukládání do databáze.

LoadedIsSaving: Objekt již v databázi existuje a zároveň probíhá ukládání do databáze.

IsLoading: Objekt se plní daty z databáze.

IsDeleting: Objekt se maže z databáze.

NotFound: Objekt má naplněnu hodnotu identifikátoru, ale nebyl v databázi nalezen.

Stavy v průběhovém čase jsou hlavně používány samotným frameworkem.

Dále objekt obsahuje vlastnost „Id“. Tato vlastnost slouží pro identifikaci objektu. Za zmínku také stojí asociace na třídu „GenSpecType“. Díky této třídě může programátor nahrát z databáze konkrétní typ potomka z typu předka. Tato věta říká, že programátor může z databáze nahrát všechny objekty typu nejvyššího předka a systém se postará o vytvoření správného typu potomka, který byl ukládán do databáze. Dále obsahuje pomocné metody na přiřazení a vrácení hodnoty z vlastnosti dle názvu pomocí reflexí. Třída „StoredObject“ také přepisuje standardní metody a operátory ze „System.Object“. Jedná se o metody „Equals“, „GetHashCode“ a operátory „==“ a „!=“. To vše z důvodu porovnání dle datového typu a identifikátoru.

6.6 Implementace dalších funkcí

V této kapitole budou popsány další důležité funkce definované v požadavcích na produkt.

6.6.1 Lazy load kolekce

Nahrávání celého objektového grafu může být velmi náročné na prostředky počítače. Navíc je uživatel aplikace zdržován nahráváním dat, které pravděpodobně ani nevyužije. Řešením tedy bude vytvoření vlastní kolekce, která se musí postarat o správné načasování nahrávání objektů z databáze. Bude vytvořena nová třída „StoredCollection“, která si bude udržovat informace o filtru, řazení a vnitřní kolekci obsahující pouze identifikátory objektů. Třída implementuje nebo přepisuje všechny vlastnosti a metody kolekce, které přistupují k objektům. V případě, že je vyžádán první objekt nebo počet záznamů, je kolekce

inicializována. Inicializace spočívá v naplnění vnitřní kolekce identifikátorů dle daného filtru a řazení. V případě, že je vyžadováno vydání objektů, ať už dle indexu nebo enumerací, je objekt nahrán z databáze a vydán. „StoredCollection“ je také využívána pro vztah typu kompozice 1:N. Tento vztah v podstatě není nic jiného, než filtrovaná kolekce objektů.

6.6.2 Podpora windows forms bindingu

Další podporou pro vývojáře v technologii Windows Forms je komponenta „QuickBingingSource“. Je děděna ze třídy „BindingSource“ a rozšiřuje ji o automatické nahrávání a ukládání dat do databáze. Práce s touto komponentou je následující: Na windows formulář vývojář přidá komponentu „QuickBindingSource“ z panelu nástrojů. V návrháři (Windows Forms Designer) ve vlastnostech komponenty vybere třídu, s jejichž daty bude chtít pracovat. Poté přidá na formulář ovládací prvek umožňující datovou vazbu (Binding) například prvek „DataGridView“. Jako datový zdroj vybere instanci komponenty „QuickBindingSource“ z formuláře. Nyní je možno přidávat vlastnosti nejen z této třídy, ale i vnořené. Takto je možné přidat například sloupec gridu, který je vázaný na vlastnost „Trvalé Bydliště.Město.Poštovní směrovací číslo“ třídy „Osoba“. Po tomto jednoduchém nastavení je možné aplikaci spustit a zobrazit formulář. Při zobrazení jsou automaticky nahrána data z databáze (Díky lazy load kolekci jen zobrazené záznamy) a zobrazena ve sloupcích gridu. Díky „QuickBindingSource“ je možné přidávat a mazat řádky editovat sloupce a všechny tyto akce jsou provázány přes business logiku přímo do databáze. To vše je již možné vytvářet ve vývojovém prostředí, ale pouze s těsnou vazbou na databázi bez střední vrstvy. Tato komponenta velice urychlí práci vývojářům, bez nutnosti opustit třívrstvou architekturu.

6.6.3 Tvorba jazyku pro objektové dotazy a následná implementace parseru na sql dotazy

V předchozích kapitolách byla popsána datová vrstva a její metody na získávání dat. Vstupní parametry byly často sql dotazy. Tyto dotazy je nutné napsat například v logice aplikace. Tyto dotazy, ale není vhodné psát přímo pro danou databázi, protože jedním z požadavků frameworku byla podpora více databázových strojů. Vznikl tedy požadavek na komponentu překladače z určitého obecného jazyka do sql jazyka dané databáze.

Obecný jazyk by měl být velice jednoduchý a intuitivní. Dotazy v tomto jazyku by měly vypadat dle následujícího příkladu:

„Vyber všechny osoby, jejichž město trvalého bydliště má psč je 777555.“ – „Query.Translate<Person>(MainAddress.City.Zip=‘75505‘)“

Z příkladu je patrné, že obecný jazyk je objektový a velmi intuitivní.

Implementace požadavku si vyžádá nové rozhraní. Situace je stejná jako u databázové vrstvy. Zde bylo vytvořeno obecné rozhraní, které musí implementovat perzistory pro konkrétní databáze. Rozhraní pro dotazy se bude jmenovat „IQuery“ a bude umístěno do jmenného prostoru „QuickPersistor.Data“. Bude obsahovat následující metody.

Metoda „Translate“ slouží pro překlad z objektového jazyka do jazyka daného databázového stroje. Obsahuje různá přetížení (řazení, podmínka, atp). Dále obsahuje vlastnosti „CommandText“ – přeložený příkaz konkrétního jazyka, „Condition“ – Přeložená podmínka konkrétného jazyka, „Top“ – Počet požadovaných záznamů, „UseTop“ – určuje zda bude použito omezení „Top“.

Objektový dotaz obecného jazyka se musí rozdělit na požadované sloupce, tabulky, podmínky a řazení. O toto rozdělení se stará obecná třída „Parser“, která obsahuje jedinou metodu „Parse“ se vstupním parametrem typu řetězec. Tato metoda obsahuje kompletní rozdělení sql dotazu na jednotlivé části (operandy, relační operátory, logické operátory, hodnoty). Rozdělení je implementováno pomocí regulárních výrazů.

Implementace pro konkrétní databázový stroj obdrží datovou tabulku (DataTable) rozdělených částí od třídy „Parser“ a postará se o jejich překlad pro konkrétní databázi. Nejsložitější část je správné poskládání jednotlivých spojení tabulek.

6.6.4 Generování a aktualizace databáze, tabulek, primárních a cizích klíčů dle tříd

Další velmi užitečnou částí “QuickPersistoru” je generování a aktualizace databázového schéma dle tříd a jejich vztahů. Tato část musí být opět otevřená pro implementace různých databázových strojů. Proto bude opět vytvořen interface nazvaný „IDatabaseCreator“. Tento interface obsahuje následující metody: *„CheckDatabase“* – Metoda zjišťuje, zda existuje daná databáze. V případě že neexistuje tak ji založí. *„IsTableExist“* – Metoda zjišťuje, zda existuje daná tabulka. *„GetMissingColumns“* – Metoda má za úkol vrátit chybějící sloupce v dané tabulce oproti třídě. *„CreateTable“* – Metoda vytvoří tabulku

v databázi. *“AddForeignKey“* – Metoda vytvoří cizí klíč mezi tabulkami. *“AddColums“* – Metoda přidá do tabulky dané sloupce.

Implementace konkrétního databázového stroje má za úkol dané příkazy implementovat. Tyto příkazy volá obecná business logika dle schéma tříd.

Za zmínku stojí, že třída pouze sloupce a tabulky přidává, ale nikdy neodebírá. Tato operace by byla velmi nebezpečná a mohla by způsobit ztrátu dat.

7 IMPLEMENTACE DATOVÉ VRSTVY V APLIKACI VK-PORTAL

První aplikací, která vznikla nad frameworkem „QuickPersistor“ byl webový portálový server VK-Portal.

7.1 Vize projektu Vk-Portal

Požadavkem bylo vytvořit program, který by umožňoval uživatelsky vytvářet nad jednou instalací jednotlivé portály. Tyto portály by měly představovat samostatné webové prezentace. Struktura a obsah by měla být také uživatelsky definovatelná. Obsah by si měl uživatel skládat pomocí jednotlivých modulů, které by mohly být do aplikace instalovány. Aplikace má umožňovat spravovat uživatele a definovat jim práva. Jednotlivé stránky by se měly zobrazovat na adresách vyhovujících SEO požadavkům. Aplikace by měla také obsahovat další možnosti, jejichž popis by však byl nad rámec této práce.

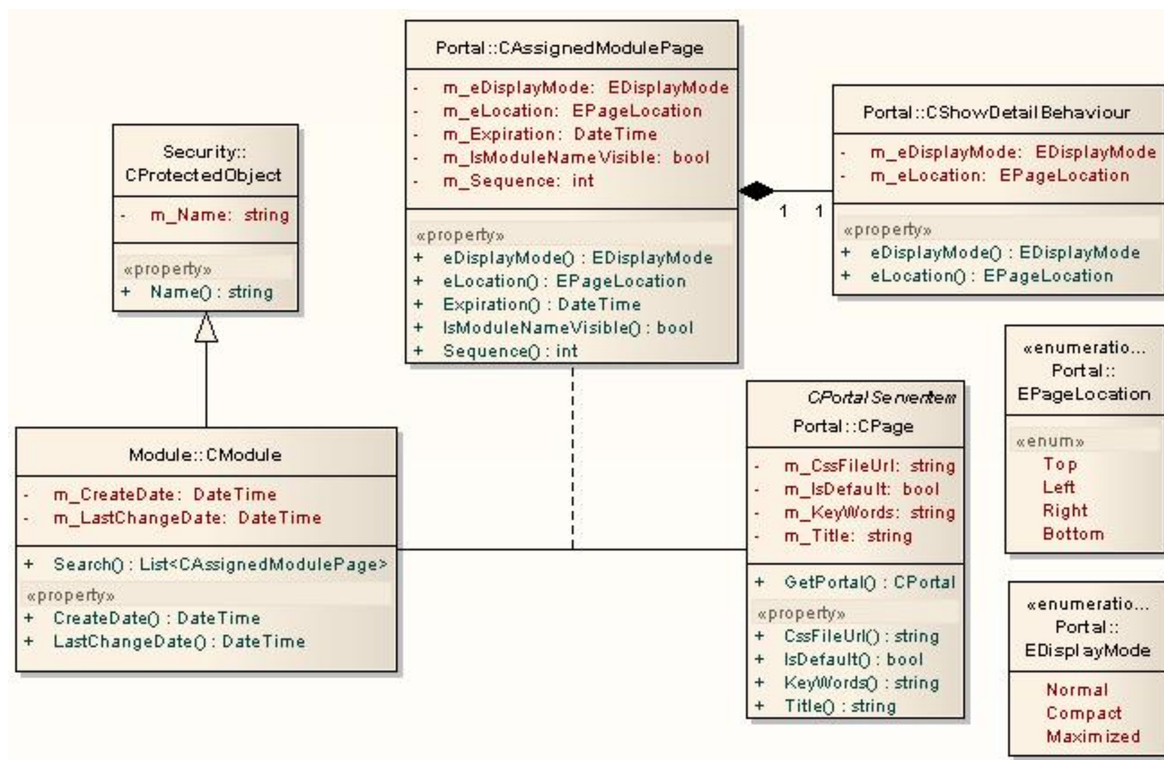
7.2 Business logika aplikace

V této kapitole bude popsána business logika aplikace. Nejjednodušší bude popsat návrh na diagramu tříd. Poté bude následovat model komponent.

Obrázek 30. Část diagramu tříd popisující základní strukturu strukturu

Diagram bude nejvhodnější popsat od třídy „PortalServer“. Ten představuje základní část celého řešení, kromě vlastností společných pro všechny portály také obsahuje v kompozici kolekci portálů. Portál si lze představit jako webovou prezentaci se svou strukturou a obsahem. Z diagramu lze dále vidět, že portál obsahuje neomezený počet sekcí (kategorií). Každá sekce obsahuje jednu stránku nebo další podsekce. Dále zde existuje dědičnost z důvodů implementace práv. Nejvyšší předek v tomto diagramu představuje chráněný objekt.

Nyní bude popsána struktura obsahu jednotlivých stránek.



Obrázek 31. Diagram tříd představující řešení obsahu jednotlivých stránek

Z diagramu je patrné, že stránka může obsahovat jednotlivé moduly. Vazbu mezi stránkou a modulem představuje asociační třída „AssignedModulePage“. Tento vztah také říká, že jeden modul může být sdílen mezi více stránkami. Další třídy už jen určují jakým způsobem bude řízeno zobrazení a řazení modulů na stránce.

Pro ilustraci zde bude kompletní diagram tříd business logiky aplikace, který je součástí přílohy P1.

Z diagramu lze vypozorovat, že se zde vyskytuje většina typů vazeb snad jen kromě agregace. Všechny tyto vazby musí umět implementovat datová vrstva frameworku „QuickPersistor“.

7.3 Implementace frameworku QuickPersistor

V této kapitole budou ukázky zdrojových kódů, které budou ilustrovat jakým způsobem je do aplikace implementována datová vrstva „QuickPersistor“.

Nyní zde bude představena implementace tříd dle obrázku (Obrázek 30).

```
namespace vknet.Portal
{
    [Persistent("vknet_Portal_PortalServer")]
    public class PortalServer : PortalServerItem
    {
        #region Fields
        private string _SmtpServerAddress = "";
        private StoredCollection<Portal> _ColPortal = new StoredCollection<Portal>();
        private string _CssFileUrl = "";
        private Accessor _Guest;
        #endregion

        #region Properties
        public string SmtpServerAddress
        {
            get { return _SmtpServerAddress; }
            set { _SmtpServerAddress = value; }
        }

        public string CssFileUrl
        {
            get { return _CssFileUrl; }
            set { _CssFileUrl = value; }
        }

        [Association("PortalServer_Portal", Association.AssociationTypeEnum.Composition)]
        public StoredCollection<Portal> ColPortal
        {
            get { return _ColPortal; }
            set { _ColPortal = value; }
        }

        /// <summary>
        /// Přistupovatel dle kterého budo brána prava pro nepřihlášeného uživatele
        /// </summary>
        public Accessor Guest
        {
            get { return _Guest; }
            set { _Guest = value; }
        }
        #endregion
```

Obrázek 32. Implementace QuickPersistoru ve třídě PortalServer

Nad třídou je definován atribut typu „Persistent“, který určuje, že třída bude ukládána do databáze do tabulky z parametru tohoto atributu. Nejvyšší předek této třídy je typu „StoredObject“. Za zmínku stojí mapování vztahu kompozice 1:N pro seznam portálů. Vlastnost „Guest“ typu „Accessor“ je také persistentní, aniž by obsahovala atribut „Peristent“. Všechny veřejné vlastnosti ve třídách, které je potomky „StoredObject“ jsou totiž ve výchozím stavu persistentní, protože se předpokládá, že většina vlastností je i v databázi. Pokud není žádoucí vlastnost perzistovat, stačí nad ní definovat atribut typu „NonPersistent“.

Nyní bude představena asociační třída pro přiřazení modulu ke stránce.

```
[Persistent("vknet_Portal_AssignedModulePage")]
public class AssignedModulePage:StoredObject
{
    #region Fields
    private PageLocation _PageLocation;
    private Module _Module;
    private Page _Page;
    private DateTime? _Expiration;
    private bool _IsModuleNameVisible;
    private int _Sequence;
    private DisplayMode _DisplayMode = DisplayMode.Maximized;
    private ShowDetailBehavior _ShowDetailBehaviour;
    #endregion

    #region Properties
    public Module Module
    {
        get { return _Module; }
        set { _Module = value; }
    }

    public Page Page
    {
        get { return _Page; }
        set { _Page = value; }
    }

    public PageLocation PageLocation
    {
        get { return _PageLocation; }
        set { _PageLocation = value; }
    }

    public DateTime? Expiration
    {
        get { return _Expiration; }
        set { _Expiration = value; }
    }
```

Obrázek 33. Asociační třída pro přiřazení modulu ke stránce

Třída je v obdobná jako v předchozím případě. Asociace je zde implementována automaticky. Je to výchozí stav vlastnosti podtypu „StoredObject“. Z vlastnosti „Expiration“ je patrné, že „QuickPersistor“ podporuje nullable typy.

Předchozí příklady ukazují, že implementace datové vrstvy je velmi snadná. V další kapitole bude ilustrováno, jak se s frameworkem pracuje.

7.4 Použití QuickPersistoru v aplikaci VK-Portal

Opět není v možnostech této práce popsat kompletní zdrojový kód aplikace. Proto budou popsány pouze vzorové části.

Při použití frameworku „QuickPersistor“ není nutné explicitně inicializovat aplikaci. Vytvoření správné instance perzistoru dle konfigurace a vytváření schéma aplikace je činností, která se provádí automaticky.

Po přístupu na adresu aplikace je automaticky předána výchozí stránka „Default.aspx“ tato stránka obsahuje ovládací prvek portálu „wucPage“. Tento prvek řídí veškerá zobrazování obsahu stránek. Nyní bude představen kód metody „OnInit“ která je volána jako první při požadavku na zobrazení. V této metodě je možno dynamicky vytvářet ovládací prvky stránky. V události „Load“ již toto není možné.

```
protected override void OnInit(EventArgs e)
{
    try
    {
        base.OnInit(e);

        wucPageAdministrationPanel1.Visible = false;
        if (PortalSession.LoggedUser != null)
        {
            IsViewAdministrationTools = true;
            wucPageAdministrationPanel1.Visible = true;
        }

        vknet.Portal.Page page = null;

        if (PortalSession.PortalServer != null)
        {
            if (PortalSession.AllowShow(PortalSession.PortalServer))
            {
                if (!wucPageAdministrationPanel1.Visible && PortalSession.LoggedUser != null &&
                    PortalSession.HasPermission(PortalServerItem.Permissions.Edit, PortalSession.PortalServer))
                {
                    wucPageAdministrationPanel1.Visible = true;
                    IsViewAdministrationTools = true;
                }

                Portal portal = PortalSession.GetCurrentPortal();
                if (portal != null)
                {
                    if (PortalSession.AllowShow(portal))
                    {
                        //Nastaveni odkazu Home na horním panelu
                        if (!wucPageAdministrationPanel1.Visible && PortalSession.LoggedUser != null &&
                            PortalSession.HasPermission(PortalServerItem.Permissions.Edit, portal))
                        {
                            wucPageAdministrationPanel1.Visible = true;
                            IsViewAdministrationTools = true;
                        }

                        hlMainPageLink.NavigateUrl = PortalSession.GetCurrentPortalRootPath();

                        page = PortalSession.GetCurrentPage();
```

Obrázek 34. Část kódu, který generuje obsah stránky

Na začátku kódu dojde k ověření, zda je přihlášen uživatel z důvodu povolení nebo zakázání obsahu. Dále je vyhledán „PortalServer“, „Portal“ a nakonec konkrétní instance stránky „Page“. Zatím zde není žádný kód, který by volal metody perzistentního frameworku. Konkrétní volání je v pomocné třídě „PortalSession“, která uchovává data v rámci jedné „HttpSession“. Další ukázkou zdrojového kódu bude metoda, která zjišťuje, zda má objekt, přistupující k jinému, potřebná práva. Tato metoda ilustruje použití objektových dotazů.

```
public bool HasAssignedPermission(Enum permissionEnum, Accessor accessor)
{
    try
    {
        Type permissionEnumType = permissionEnum.GetType();
        AssignedPermission assignedPermission = DataSession.GetObject<AssignedPermission>(
            Query.Translate<AssignedPermission>(
            "Accessor=" + accessor.Id + " AND ProtectedObject=" + Id +
            " AND Permission.Code='" + Enum.GetName(permissionEnumType, permissionEnum) +
            "' AND Permission.ProtectedObjectType.AssemblyQualifiedName='" +
            permissionEnumType.DeclaringType.AssemblyQualifiedName + "'"));
        return assignedPermission !=null;
    }
    catch
    {
        throw;
    }
}
```

Obrázek 35. Příklad objektového dotazu.

Metoda z obrázku se dotazuje na přiřazení práva ze vstupního parametru mezi objektem typu „Accessor“ (Přistupující objekt) a „ProtectedObject“(Chráněný objekt). Volání je velmi jednoduché a intuitivní na rozdíl od výsledného sql dotazu. Následující kódy představí kompletní CRUD operace jednoduchého modulu „SimpleText“, který slouží pro jednoduchou tvorbu obsahu pomocí html editoru.

```
protected void Page_Load(object sender, EventArgs e)
{
    try
    {
        if (!this.IsPostBack)
        {

            if (CurrentModuleId > -1)
            {
                SimpleTextModule simpleTextModule =
                    DataSession.GetObject<SimpleTextModule>(CurrentModuleId);
```

Obrázek 36. Vyžádání objektu z databáze dle identifikátoru

```
protected void btnAccept_Click(object sender, ImageClickEventArgs e)
{
    try
    {
        if (CurrentModuleId > -1)
        {
            SimpleTextModule simpleTextModule =
                DataSession.GetObject<SimpleTextModule>(CurrentModuleId);
            if (simpleTextModule != null)
            {
                simpleTextModule.Text = txtSimpleText.Value;
                simpleTextModule.Save();
```

Obrázek 37. Uložení objektu do databáze

```
protected void btnDelete_Click(object sender, ImageClickEventArgs e)
{
    try
    {
        if (CurrentModuleId > -1)
        {
            SimpleTextModule simpleTextModule =
                DataSession.GetObject<SimpleTextModule>(CurrentModuleId);
            if (simpleTextModule != null)
            {
                simpleTextModule.Delete();
```

Obrázek 38. Odstranění objektu z databáze

8 ZHODNOCENÍ PŘÍNOSU A SEZNAM POŽADAVKŮ PRO BUDOUCÍ VERZI.

Perzistentní framework „QuickPersistor“ podstatně urychlil práci na aplikaci VK-Portal. Bez jeho pomoci by byla implementace jednotlivých modulů velmi pomalá a nezáživná činnost. Velkou roli hraje použití frameworku i při chybovosti programátora. Odpadá nutnost implementace velmi podobných funkcí, které jsou náchylné na chyby typu „Copy&Paste“ (Chyba vzniklá kopírováním a opomenutím změnit části kopírovaného kódu). Další nespornou výhodou je standardní přístup k perzistenci. Díky tomu je možno vytvářet obecně použitelné komponenty a spoustu duplicitní práce dělat za programátora. Příkladem může být právě komponenta „QuickBindingSource“ popisovaná v této práci. Díky tomu, že má framework ve výchozím stavu implementován takzvaný „Lazy Load“, jsou aplikace velmi rychlé a šetrné k prostředkům. Aplikace VK-Portal Server byla testována na statisících funkčních (naplněných daty) portálů. Díky „Lazy Load“ nebylo pozorováno zpomalení. Funkce postupného načítání byla nutným požadavkem pro tvorbu frameworku. Stačí si představit nahrání objektového grafu samotného portálu. Testovaný příklad byl: Jeden portál server obsahoval sto tisíc portálů, každý portál obsahoval stromovou strukturu, až sto sekcí. Každá sekce obsahovala jednu stránku, na kterou bylo přiřazeno 10-15 modulů. V případě, že by se při dotazu na portálový server nahrál celý objektový graf z databáze, nebylo by možné aplikaci vůbec provozovat.

Mezi jasné nevýhody tohoto frameworku patří těsná vazba business a prezentační vrstvy na „QuickPersistoru“. Tato těsná vazba by se dala zrušit mapovací konfigurací a nepoužitím dědičnosti. Aplikace by byla sice „čistější“, ale vývoj by byl opět pomalejší. Další nevýhodou a také požadavkem na novou verzi je absence databázových transakcí. I v tomto případě si dokáže programátor celkem snadno poradit a to použitím Microsoft Distributed Transaction.

8.1 Seznam požadavků pro budoucí verzi

V budoucí verzi by měly být implementovány následující požadavky: Interní práce v perzistentní vrstvě automaticky zařazena do transakcí, podpora dalších databázových strojů, rozšíření možností mapování na již stávající databázové schéma, možnost použití GUID jako identifikátoru, implementace konkurence (zamykání záznamů), sada Windows

Forms, Wpf a webových uživatelských prvků s rozšířenou možností bindingu. A nakonec možnost oddělení business logiky od datové vrstvy.

ZÁVĚR

Na začátku této práce byla popsána teorie mapování objektů do relační databáze. Poté byly uvedeny možné přístupy k perzistenci objektů s výčtem výhod a nevýhod jednotlivých řešení. Po úvodu do reflexí byl definován ideální perzistentní framework. Tento framework byl ukázán na teoretickém příkladu.

V praktické části byl definován seznam požadavků na perzistentní framework. Tento seznam byl rozdělen na dvě části. První část obsahuje požadavky na datovou vrstvu. Ve druhé části jsou požadavky pro generátor sql skriptů. Požadavky byly analyzovány pomocí diagramu případu užití a scénářů. Byly uvedeny pouze některé scénáře včetně jejich alternativ. Po fázi analýzy byla popsána samotná realizace.

Realizace byla také rozdělena na dvě části. První část je věnovaná podrobnějšímu popisu řešení perzistentního frameworku. Obsahuje jednotlivé problémy, které při vývoji vznikly, včetně jejich řešení. Některá řešení byla tak komplexní, že by přesáhla rámec této práce. V tomto případě byly pospány jen nejdůležitější fragmenty. Na příkladech bylo demonstrováno použití reflexe pro tvorbu dynamických aplikací, práce s databází a návrh rozhraní pro oddělení logiky od samotné implementace. Samostatnou kapitolu tvoří realizace objektového jazyka. Zde byl cíl splněn a vznikl intuitivní jazyk pro tvorbu dotazů do databáze. Tento jazyk není závislý na použitém databázovém stroji.

Druhá část obsahuje popis implementace vyvinutého frameworku na komplexní portálové aplikaci VkPortal. V této kapitole bylo velmi zjednodušeně popsáno použití datové vrstvy na webové aplikaci. Na příkladu bylo uvedeno rozšíření aplikační logiky o potřebné atributy. Dále bylo ukázáno získání dat z databáze pomocí objektového dotazu a použití CRUD operací.

V závěru diplomové práce bylo provedeno zhodnocení a byl definován seznam požadavků pro budoucí verze.

Mým úkolem bylo vytvořit perzistentní framework, jehož cílem je rychlá a intuitivní implementace datové vrstvy do objektových aplikací. Tento cíl byl dle mého názoru splněn a vznikl perzistentní framework „QuickPersistor“.

Tvorba této aplikace mi dala rozsáhlé znalosti z oblasti relačních databází, objektově relačního mapování, reflexí, regulárních výrazů a implementace velmi složitého návrhu.

Při implementaci na první skutečné aplikaci, bylo potřeba spoustu věcí vyladit nebo upravit. Návrh systému byl prověřen implementací pro databázový stroj Oracle programátorem, který se na projektu nepodílel. K implementaci mu stačil navržený interface. „QuickPersistor“ je implementován v několika aplikacích, které bez problému fungují. Aplikace VK-Portal je nasazena u mnoha zákazníků a funguje bez nutnosti oprav.

ZÁVĚR V ANGLIČTINĚ

At the beginning, the thesis described a theory of mapping objects into the relational database. Furthermore, itspecified ways of persistency approach. Every approach contains a list of advantages and disadvantages. After the reflection description an ideal of persistent framework was defined. This framework was demonstrated on a theoretical example.

The practical chapter defined a list of persistent framework requirements. This list was divided into the two parts. The first part contains data layer requirements and the second part contains requirements for generator of sql scripts. Requirements were analyzed by case diagram and scenarios. This document contains main scenarios with their alternative. After the analysis, realization was described.

The realization was divided into two parts, too. The first part deals with a more detailed description of persistent framework solution. This part contains development problems and their solutions. Some problems were very complex. In this case only the most important resolution fragments were described. The reflection was demonstrated on examples of dynamical applications. Other expamples were about interface used for separate logic and implementation. A very important chapter focuses on object oriented language realization which is independent of a database engine. This intuitive language was successfully developed.

Next part contains description of developed framework implementation into the VkPortal application. In this chapter, use of data layer in web application was described. In this example business logic extension for needed attributes was shown. Furthermore, CRUD operations and using of object oriented language for loading data from database were shown here.

At the end of this work, evaluation was executed and a list of requirements for future versions was defined.

My task was to create a presistent framework. The goal of a framework is to quickly and intuitively implement data layer into the object oriented applications. I think that aim was accomplished and a persistent framework „QuickPersiastor“ was successfully developed.

Persistent Framework creation gave me wide knowledge of relational databases, Object-relational mapping, reflections, regular expressions and designing of complex systems.

Many things had to be fixed in the first implementation of this framework to a real system. The design was checked by implementation of Oracle database system. This implementation was made by a programmer from outside of this project. The programmer only implemented the required interface. The „QuickPersistor“ was used in many applications. These applications were running without any problems. The VK-Portal was deployed for many customers and has been functioning sufficiently and successfully.

SEZNAM POUŽITÉ LITERATURY

[1] **Kraval, Ilja.** *Server objektových technologií.* [Online] http://www.objects.cz/.

[2] **kolektiv, Martin Fowler a.** *Refaktoring - Zlepšení existujícího kódu.* s.l. : Grada Publishing, 2003. 80-247-0299-1.

[3] **Klein, Scott.** *Professional WCF Programming .NET Development with the Windows® Communication Foundation.* s.l. : Wiley Publishing, Inc., 2007. 978-0-470-08984-2.

[4] **Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, Morgan Skinner.** *C# 2008 Programujeme profesionálně.* s.l. : Computer Press, 2009. 978-80-251-2401-7.

[5] **Augustýn, Michal.** Moderní programování v C#. *Augiho web.* [Online] http://www.augi.cz.

[6] **Kraval, Ilja.** *Základy objektově orientovaného programování.* Praha : Computer Press, 1998. 80-7226-047-2.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

GUI Graphical User Interface (Grafické uživatelské rozhraní).

CRUD Databázové operace (Create, Read, Update, Delete).

ID Identifikace

SEZNAM OBRÁZKŮ

Obrázek 1. Mapování, Třída - Tabulka ... 13
Obrázek 2. Mapování dědičnosti 1:1 ... 14
Obrázek 3. Mapování dědičnosti N:1 ... 14
Obrázek 4. Mapování dědičnosti 1:N ... 15
Obrázek 5. Ukázka dědičnosti ... 16
Obrázek 6. Mapování asociace ... 16
Obrázek 7. Ukázka asociace ... 17
Obrázek 8. Mapování kompozice ... 18
Obrázek 9. Ukázka kompozice 1:N ... 18
Obrázek 10. Class diagram agregace ... 19
Obrázek 11. Mapování asociační třídy ... 20
Obrázek 12. Ukázka asociační třídy ... 20
Obrázek 13. Class diagram aplikace bez implementace datové vrstvy ... 23
Obrázek 14. Ukázka datové vrstvy ... 24
Obrázek 15. Možné řešení perzistence ... 27
Obrázek 16. Okno řešení ve vývojovém prostředí Microsoft Visual Studio 2010 ... 30
Obrázek 17. Ukázka základní metody pro naplnění objektu z databáze ... 31
Obrázek 18. Ukázka vytvoření nové instance objektu a naplnění daty ... 32
Obrázek 19. Ukázka přiřazení hodnoty do vlastnosti ... 32
Obrázek 20. Ukázka inicializace frameworku a získání objektu z databáze ... 33
Obrázek 21. Stavy objektu ... 33
Obrázek 22. Uložení objektu ... 34
Obrázek 23. Implementace metody pro založení objektu v databázi ... 35
Obrázek 24. Implementace metody pro aktualizaci objektu v databázi ... 36
Obrázek 25. Práce s perzistencí objektu ... 37
Obrázek 26. Implementace mazání objektu z databáze ... 37
Obrázek 27. Use Case model ... 46
Obrázek 28. Odchycení databázové výjimky ... 52
Obrázek 29. Diagram tříd zachycující schéma databáze ... 54
Obrázek 30. Část diagramu tříd popisující základní strukturu strukturu ... 64
Obrázek 31. Diagram tříd představující řešení obsahu jednotlivých stránek ... 65

Obrázek 32. Implementace QuickPersistoru ve třídě PortalServer 66
Obrázek 33. Asociační třída pro přiřazení modulu ke stránce 67
Obrázek 34. Část kódu, který generuje obsah stránky 69
Obrázek 35. Příklad objektového dotazu. 70
Obrázek 36. Vyžádání objektu z databáze dle identifikátoru 70
Obrázek 37. Uložení objektu do databáze 71
Obrázek 38. Odstranění objektu z databáze 71

SEZNAM PŘÍLOH

P1 CLASS DIAGRAM PROGRAMU VK-PORTAL.

PŘÍLOHA P I: CLASS DIAGRAM PROGRAMU VK-PORTAL