VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

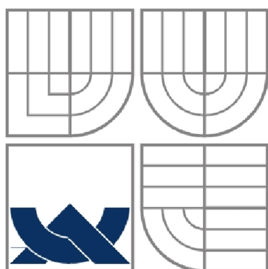
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR DATABÁZOVÉ VRSTVY APLIKACÍ

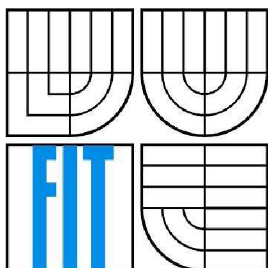
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAROSLAV KUBOŠ
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERÁTOR DATABÁZOVÉ VRSTVY APLIKACÍ

APPLICATION DATABASE LAYER GENERATOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAROSLAV KUBOŠ
AUTHOR

VEDOUCÍ PRÁCE Ing. RADEK BURGET, Ph.D.
SUPERVISOR

BRNO 2007

Zadání

Generátor databázové vrstvy aplikací

Application Database Layer Generator

Vedoucí:

Burget Radek, Ing., Ph.D., UIFS FIT VUT

Zadání:

1. Seznamte se s jazykem C#, zejména s prací s relačními databázemi a reflexí
2. Prostudujte existující způsoby persistence objektů
3. Navrhněte způsob popisu metadat pro persistenci objektů s použitím reflexe
4. Navrhněte knihovnu pro podporu tvorby databázové vrstvy aplikace
5. Implementujte knihovnu v jazyce C#
6. Zhodnoťte použitelnost a srovnejte s jinými přístupy

Část požadovaná pro obhajobu SP:

Body 1 až 4.

Kategorie:

Databáze

Implementační jazyk:

C#

Literatura:

- Robinson, S., et al: C# Programujeme profesionálně, Computer Press, 2003, ISBN 8025100855

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Cílem projektu je návrh a implementace knihovny pro podporu vývoje databázové/perzistenční vrstvy aplikací psaných v jazyce C#. Knihovna se snaží o maximální snadnost použití při zachování elegance zápisu. Knihovna podporuje rysy objektového programování typu dědičnost a kolekce. Z dalších rysů lze jmenovat podporu verzování a opožděného načítání. Pro získávání metadat o objektech používá reflexi poskytovanou .NET frameworkem. Knihovna nepoužívá žádné literály pro identifikaci (tříd, atributů) a to ani v objektových dotazech. Většinu kontrol proto provede už překladač.

Klíčová slova

objektově-relační mapování, perzistence, ORM, hibernace, C#, DB, relační databáze, SQL, verzování, reflexe

Abstract

This diploma thesis deals with the design and implementation of a framework for the database persistence layer development. This framework is easy to use while keeping the code elegance. It supports object oriented programming features such as inheritance and collections. Other features include versioning of objects and lazy loading. The object metadata are obtained through reflection provided by the .NET framework. The framework is not using any literal for identification (classes, attributes) even in object queries. Most of checks are done by compiler.

Keywords

object-relational mapping, persistence, ORM, hibernation, C#, DB, relational database, SQL, versioning, reflection

Citace

Jaroslav Kuboš: Generátor databázové vrstvy aplikací, diplomová práce, Brno, FIT VUT v Brně, 2007

Generátor databázové vrstvy aplikací

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Burgeta, Ph.D.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jaroslav Kuboš
10.5.2007

Poděkování

Poděkování za odborné vedení a korekce patří panu Ing. Radku Burgetovi Ph.D.

© Jaroslav Kuboš, 2007.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah ... 1
1 Úvod ... 4
2 Současný stav ... 5
2.1 Rozdělení ORM knihoven ... 5
2.2 DataObjects.NET ... 7
2.3 NHibernate ... 7
2.4 Sisyphus Persistence Framework ... 7
2.5 Vlastní implementace ... 8
2.6 Další ... 8
3 Teorie a použité technologie ... 9
3.1 Ontologie ... 9
3.1.1 Popis ... 9
3.1.2 Ontologický popis řešené domény ... 10
3.1.3 Export datového modelu ... 10
3.2 Perzistentní třída/objekt ... 11
3.3 C# a relační databáze ... 11
3.4 C# a šablony ... 12
3.5 C# a reflexe ... 13
3.6 C# a přetěžování operátorů ... 14
3.6.1 Přetížení operátoru "." ... 15
3.7 Parciálně definované třídy ... 15
3.8 Modifikátor přístupu "internal" ... 16
4 Cíle ... 17
4.1 Všechny perzistenční informace uložené v kódu ... 17
4.2 Podpora OOP vztahů perzistentních objektů ... 18
4.2.1 Dědičnost ... 18
4.2.2 Kolekce ... 18
4.3 Abstrakce úložiště perzistence ... 18
4.4 Přehledná a definovaná struktura úložiště ... 20
4.5 Transparentnost a elegance ... 20
4.6 Události v životě objektů ... 21
4.7 Další pokročilé rysy ... 21
4.7.1 Verzování ... 21
4.7.2 Odložené načítání ... 22

4.7.3 Generování/úpravy schématu databáze 22
4.7.4 Transakce 22
4.7.5 Identifikace instance 23
4.7.6 Objektové dotazy 23
4.7.7 Objekty a vyrovnávací paměť 24
4.7.8 Inicializace vlastností, referencí a kolekcí 24
5 Návrh 26
5.1 Všechny perzistenční informace uložené v kódu 26
5.2 Podpora OOP vztahů perzistentních objektů 26
5.2.1 Dědičnost 26
5.2.2 Kolekce 27
5.3 Abstrakce úložiště perzistence 30
5.4 Přehledná a definovaná struktura úložiště 30
5.5 Transparentnost a elegance 31
5.6 Události v životě objektů 31
5.7 Další pokročilé rysy 31
5.7.1 Verzování 31
5.7.2 Odložené načítání 32
5.7.3 Generování/úpravy schématu databáze 32
5.7.4 Transakce 33
5.7.5 Identifikace instance 33
5.7.6 Objektové dotazy 33
5.8 Objektový návrh 33
5.8.1 Schéma modulů 33
5.8.2 Modul "Common" 34
5.8.3 Modul "Ontology" 35
5.8.4 Modul "Perzistence" 36
6 Implementace 41
6.1 Prostředí, přenositelnost 41
6.2 Použitý databázový systém 41
7 Ukázka použití 43
7.1 Práce s knihovnou ve Visual Studiu 43
7.2 Definice perzistentních typů 44
7.3 Příprava použití knihovny 46
7.4 Záložka "Oddělení" 47
7.5 Záložka "Zaměstnanci" 48
7.6 Záložka "Projekty" 49

7.7 Záložka "Přiřazení zaměstnanců" 51
7.8 Záložka "Zobrazení přiřazení" 53
8 Závěr 55
Literatura 57
Seznam příloh 58

1 Úvod

Tento dokument popisuje proces návrhu a tvorby knihovny pro podporu tvorby databázové perzistenční vrstvy aplikací psaných v programovacím jazyce C#. Idea, vytvořit tuto knihovnu, mě napadla po několikáté implementaci databázové vrstvy různých programů v různých programovacích jazycích. Vždy se jednalo o zdlouhavou, jednotvárnou práci s velkým sklonem k chybám vzniklým nepozorností.

Základní motivací je vytvořit velmi snadno použitelnou knihovnu, která ale bude řešit všechny často řešené problémy (verzování, reference, dotazy, transakce, atd.). Pro popis perzistentních objektů budou použity prostředky jazyka C#. Nebude potřeba vytvářet jakékoliv externí specifikace (XML apod.).

Popis kapitol:

- "Současný stav" – Popisuje aktuální knihovny pro vytváření perzistentních objektů v C#.
- "Teorie a použité technologie" – Seznam použité teorie a technologií spolu s jejich krátkým popisem
- "Cíle" – Uvádí výčet vlastností a funkcí zamýšleného řešení. Zamýšlím se nad hlavními rysy celého mnou navrženého řešení. Uvádím důvody proč je rys důležitý, případně příklady toho, jak by vypadala implementace bez něj.
- "Návrh" – Popisuje (zejména objektový) návrh knihovny.
- "Implementace" – Popisuje implementaci knihovny a použité nástroje.
- "Ukázka použití" – Demonstrace schopností na testovací aplikaci.
- "Závěr" – Obsahuje popis současného stavu projektu a jeho budoucnost. Shrnuje poznatky získané při jeho tvorbě a snaží se vysvětlit, jak by jich šlo využít při práci na nové verzi. Ukazuje také problémy, které vznikly v důsledku snahy o dokonalou transparenci perzistence.

Tato diplomová práce vychází z poznatků nabytých při vytváření mé bakalářské práce [7]. Vnitřně generátor používal velmi jednoduchou perzistenci PHP objektů do relační databáze MySQL. Ta nedokázala řešit problémy typu ukládání dědičnosti či relací mezi objekty. Dále tato diplomová práce navazuje na semestrální projekt stejného jména. V něm byl vytvořen návrh knihovny, tedy prvních pět kapitol (ty byly rozšířeny) a prototyp systému.

2 Současný stav

Jako seznam řešení perzistence objektů v C# dostupných jako open source jsem použil informační zdroj [2]. Vybral jsem několik systémů (nejen "open source"), jež zběžně popíši a uvedu rozdíly oproti mnou navrženému řešení. Nejprve je ale potřeba specifikovat hlavní parametry, ve kterých se knihovny liší.

2.1 Rozdělení ORM knihoven

Na trhu existuje mnoho implementací ORM (Object-Relational Mapping) knihoven, ať už komerčních, či "open source". Liší se v mnoha parametrech, ale dá se vypozorovat několik hlavních skupin.

Podle stupně automatizace:

- Autonomní – knihovna implementuje nejen ORM, ale přidává i pokročilé funkce (snaží se udělat maximum práce za uživatele):
  - Verzování
  - Překladatelné objekty
  - Přístupová práva
  - Transakce
  - Automatická tvorba databázové struktury
- Poloautonomní – knihovna implementuje pouze základní ORM a očekává se, že si uživatel doplní pokročilé funkce sám
  - Výhoda – funkcionalita "pod kontrolou", možnost ošetřit i specifické problémy
  - Nevýhoda – mnoho leckdy zbytečné práce

Podle úložiště metadatových informací (tedy informací, které popisují perzistentní objekty a vztahy mezi nimi):

- XML soubory
- Získání pomocí reflexe
- Získání z databázových metadat (import neznámých typů objektů?)
- Jiný způsob uložení (v databázi, jiných formátech souborů, …)

Dalším kritériem může být způsob práce s typy a atributy (jako všude jinde platí, že statický přístup je bezpečnější, ovšem dynamický je mocnější):

- Dynamický – existence atributů, typů a vazeb se kontroluje až za běhu

- Vhodné pro skripty, použitelné i pro překládané programy
- Skript:

```
class Person : Persistent
{
	String Name;
}
Person p = new Person();
p.Name = "Jarek";
p.Test = 34;	//chyba za běhu
p.save();
```

- Překládaný program:

```
GenericObject o = new GenericObject("Person");
o.setAttr("Name","Jarek");
o.setAttr("Test","34");		//chyba za běhu
o.save();
```

- Za běhu může selhat kvůli mnoha faktorům:
  - Chybějící typ, atribut nebo vazba
  - Chybám dat (viz problémy statického přístupu)
- Za běhu je možné přidávat a modifikovat typy, vazby i atributy

* Statický – atributy, vazby a typy jsou známy v době překladu
  - Vhodné pro kompilované programy – překladač provádí kontrolu jejich existence

```
class Person : Persistent
{
	String Name;
}
Person p = new Person();
p.Name = "Jarek";
p.Test = 34;	//chyba překladu
p.save();
```

  - Je jisté že daný typ/atribut za běhu bude existovat
  - Selhat může za běhu kvůli chybám v datech – například:
    - neexistující OID při načítání objektu
    - chybný formát úložiště (neexistující tabulka, tabulka s chybějícím sloupcem, …)
    - neexistující vazba
  - Nelze přidávat a modifikovat typy za běhu

2.2 DataObjects.NET

Tato komerčně šířená knihovna se jednoznačně nejvíce podobá tomu, čeho bych chtěl docílit. Dala by se zařadit do kategorie "autonomní". Používá metodu dědění databázových objektů z obecného předka dodaného knihovnou. Hlavní rysy:

- Podpora dědičnosti, kolekcí, referencí
- Vyhledávání
- Verzování objektů
- Překladatelné objekty
- Práva
- Podpora dotazování
- Nepoužívá žádné externí konfigurační programy
- Podpora transakcí/konkurence

Jediná větší odchylka oproti mému řešení je v tom, že používá jako členských proměnných běžné objekty. Tím vyvstávají 2 problémy:

- Nelze určit, které vlastnosti jsou perzistentní a které ne (lze vyřešit atributem)
- Horší dynamické schopnosti – například nemožnost reagovat na nastavení vlastnosti okamžitě (nemožnost přidat objektu validaci stavu okamžitě po nastavení vlastnosti)

2.3 NHibernate

Tato open source knihovna si bere za vzor knihovnu Hibernate z jazyku Java. Tato knihovna by se dala zařadit do kategorie "autonomní". Hlavní rysy:

- Používá externí XML mapovací soubory (vlastnosti na sloupce databáze, specifikace typů)
- Podpora dotazování
- Podpora reakce na události objektu (vytvoření, uložení, smazání)
- Podpora transakcí/konkurence
- Podpora dědičnosti, kolekcí, referencí
- Podpora objektových dotazů

2.4 Sisyphus Persistence Framework

Velmi jednoduchá perzistenční knihovna v kategorii "poloautonomní". Nabízí pouze základní funkcionalitu. Stylem podobná knihovně kterou chci vytvořit.

2.5 Vlastní implementace

V neposlední řadě je potřeba zmínit vlastní (pro každou aplikaci zvláštní) implementaci mapování objektů do databáze. Taková implementace je ovšem časově náročná a náchylná k chybám. Výhodou je možnost optimalizací – při návrhu jsou známy všechny aspekty řešeného problému.

2.6 Další

Mezi další známé knihovny patří například ObjectBroker, db4o, Persist.NET, csopf.

3 Teorie a použité technologie

V následujících podbodech jsou uvedeny nejdůležitější části jazyka C# a frameworku .NET, které budou použity v knihovně. Dále jsou teoreticky popsány pojmy použité dále v textu. Každý podbod shrnuje nejdůležitější vlastnosti pro tvorbu knihovny v dané oblasti a zdůrazňuje výhody při použití.

3.1 Ontologie

Ontologie je filozofickou vědou která studuje jsoucno, bytí a základní pojmy. Použití ontologie v informatice je sporné pro svou nejasnou definici a vyvolává řadu diskuzí a polemik.

Samotná ontologie má poměrně dlouhou historii ve filozofii ve vztahu k existenci. Převzetí ontologie z filozofie do informatiky rozhodně není 1:1. Spíše bych řekl, že ontologie v informatice je konkretizací ontologie ve filozofii. V informatické ontologii jsou přesně stanoveny základní elementy, s nimiž ontologie pracuje. Při použití v informatice je také její úloha poněkud specifičtější – uchování znalostí týkajících se určité problematiky.

3.1.1 Popis

V našem případě se jedná získání a uchování znalostí o uživatelem definovaných typech, které mají být perzistentní. Tyto typy a další informace jsou získány pomocí reflexe. Uchovávání znalostí nebude perzistentní, ale jejich doba života bude shodná s dobou běhu aplikace. Důvodem je to, že uživatel může své typy změnit, což musí reflektovat naše znalosti.

Nejprve si uvedeme znalosti které chceme uložit a pak definujeme prostředky, které ontologie nabízí k jejich popisu. Posledním krokem je identifikace prostředků pro popis konkrétních znalostí. Získatelné znalosti jsou:

- Existence třídy
- Atributy tříd
- Relace dědičnosti
- Asociace tříd (kolekce – 1:N, reference – 1:1)

Pro popis těchto znalostí nám dává ontologie k dispozici následující čtyři elementy, přičemž ne všechny musíme nutně použít:

- Třída (koncept) – třída, nebo též koncept, je množina všech jedinců určitého typu
- Jedinec – základní stavební prvek, může představovat jak věci konkrétní, tak abstraktní
- Atribut – popisuje určitou vlastnost, charakteristiku či parametr jedince
    - Atribut je dán přinejmenším názvem a hodnotou

    - Může být specifikován typ atributu (a je otázkou, zda typ může nebo musí být typu třída)
- Vztahy – jedná se vlastně o použití atributů, kde je jako hodnota dosazen jiný objekt
    - Dědění může být reprezentováno jako atribut "ZděděnoZ" s hodnotou nastavenou na objekt, z něhož tento objekt dědí

3.1.2 Ontologický popis řešené domény

Jádro knihovny popisuje objekty určené k perzistenci pomocí ontologie:

- typy objektů – třídy
- atributy objektů – jsou definovány pro typy, ale za běhu realizovány na objektech
- typy vztahů mezi objekty – popis vztahů mezi objekty (tj. vazby jsou definovány jako předpis na typech, ovšem za běhu jsou realizovány na objektech daných typů)
    - asociace
    - vztah dědičnosti

Všechny potřebné informace lze za běhu vyčíst pomocí reflexe z metadat programu. V praxi vše funguje tak, že je definován kořenový objekt, který je předkem všech perzistentních objektů. Pak se od něj postupuje po všech děděných objektech a vytváří se popisy relace dědičnosti. Toto procházení se děje rekurzivně, čili po průchodu všemi přímo zděděnými objekty se procházejí všechny objekty děděné z nich. Po ukončení této části už existuje seznam perzistentních tříd a také vazeb dědičnosti mezi nimi.

V dalším kroku se zjišťují atributy definované na třídách. Protože všechny atributy musí být vytvořeny jako instance šablony třídy "Property<T>", opět se jedná o poměrně přímočarou záležitost.

Posledním krokem je čtení všech vlastností objektů, které jsou typu "Reference<T>" nebo "Collection<T>" které definují vztahy mezi objety typu asociace.

3.1.3 Export datového modelu

Protože v aplikaci existuje kompletní ontologický popis datového modelu aplikace, není problém tento model exportovat v různých formátech, například:

- UML diagram
- Neformální popis – dokumentace (docbook, HTML, …)
- Formální popis pomocí některého z ontologických jazyků

Takovýto exportovaný model může sloužit jako dokumentace ke knihovně. Zvláště po doplnění možnosti komentovat perzistentní objekty pomocí atributů. Například:

```
[ClassComment("Tato třída reprezentuje osobu")]
class Person : Persistent
```

```
{
        [PropertyComment("Tato vlastnost reprezentuje jméno")]
        String Name;
}
```

3.2 Perzistentní třída/objekt

Perzistentní třída je taková třída, jejíž objekty mohou mít dobu života delší, než jeden běh aplikace. Objekt musí mít implementován mechanizmus, který zaručí jeho uložení a načtení z/do úložiště. Úložištěm může být soubor, XML dokument, relační databáze, objektová databáze, …

3.3 C# a relační databáze

Hlavní metodou přístupu k relačním databázím (a nejen k nim) je v jazyce C# kolekce tříd a rozhraní ADO.NET obsažená v .NET frameworku. Jedná se o pokračovatele knihovny ADO (Active Data Objects) firmy Microsoft. Knihovnu lze chápat jako odpověď Microsoftu na javovou knihovnu JDBC. Základní vlastnosti:

- Různé datové zdroje – unifikovaná podpora různých datových zdrojů, nejen relačních databází
- Odpojené aplikace – podpora práce bez otevřeného připojení k databázi
- Podpora továrních tříd ("class factory") – používající aplikace pouze definuje třídu, pomocí které se získávají přetížené objekty – to vytváří absolutní nezávislost na databázové platformě
- Knihovna je postavena na rozhraních, pro specifickou databázi pak stačí požadované rozhraní implementovat

Pro jednotlivé druhy připojení definuje ADO.NET objekty s definovaným rozhraním. Přidání dalšího zdroje dat je pak realizováno pomocí podědění specifických objektů. Jednotlivé objekty jsou uložené ve jmenném prostoru "System.Data".

- Connection a ConnectionStringBuilder
- Command a CommandBuilder
- DataReader
- Transaction
- DataTableCollection a DataTable
- DataSet

3.4 C# a šablony

Ve verzi 2.0 přibyla do jazyku podpora generických typů. Návrháři Microsoftu vyslyšeli volání uživatelů, kteří generické typy postrádali. Nejvíce byla absence šablon vidět u kontejnerů/kolekcí. Ty byly součástí frameworku a umožňovaly ukládání referencí na objekty typu "Object". Protože každý objekt v C# je poděděný od objektu "Object", šly do kolekcí uložit jakékoli objekty, dokonce různých typů současně. Při vkládání do kolekce se objekty přetypovávat nemusely, protože je definován implicitní přetypovávací operátor na předka – "Object". Horší situace nastala při získávání referencí zpět. Implicitní přetypovávací operátor z předka na potomka neexistuje a bylo proto nutné získané reference přetypovávat ručně, což bylo velmi nepříjemné. Řešením bylo snad jen zapouzdření kolekce do svého objektu.

Vytváření generických typů je velmi podobné vytváření generických typů v C++. Ovšem v C# rozšířené, zejména v omezeních typových parametrů. Příklad definice vlastnosti perzistentního objektu, která reprezentuje kolekci referencí na "Persistent" nebo potomků:

```
public class CPersObjectPropertyColl<T> where T : Persistent
{
…
public T getAt(int index){return null;}
…
}
```

Generická třída se definuje tak, že se po názvu třídy uvedou závorky <> a do nich identifikátory typů (jeden nebo několik) které se pak používají v těle třídy místo typů návratových hodnot, proměnných, členských proměnných, atd. Za závorky se může uvést omezení typů. Například z které třídy musí být poděděny, který interface musí obsahovat, zda musí mít deklarován bezparametrický konstruktor, atd.
Generické programování v C# se neomezuje na třídy – šablonou mohou být rozhraní, delegáty a metody. Protože se to ale netýká mé práce, nebudu je zde obšírněji rozvádět.

Generické typy si také velmi dobře rozumí s reflexí – pomocí reflexe lze z instance třídy zjistit jakého typu jsou typové parametry šablony použité pro vytvoření instance. Tuto vlastnost zužitkuji při návrhu a implementaci knihovny.

Implementační zvláštností je, že v .NET frameworku jsou instance kódu šablon generovány až za běhu pomocí MSIL (Microsoft Intermediate Language). Pokud dojde k prvnímu pokusu o vytvoření instance pomocí instance šablony (šablona s nadefinovaným typem), která ještě není zkompilována, vytvoří se kód v MSIL a ten se zkompiluje. Generování šablon při kompilaci by totiž znamenalo vytvořit instance šablony pro všechny dosaditelné typy, protože díky podpoře reflexe nelze dopředu omezit množinu typů (jen použité – jako v C++).

3.5 C# a reflexe

Před definicí pojmu reflexe je nutné nejdříve definovat data, ze kterých čerpá. Základní spustitelnou jednotkou platformy .NET je assembly. Každá assembly je složena z těchto částí:

- Manifest – metadata o samotné assembly (název, verze, seznam propojených assembly)
- Zdroje – obrázky, texty a jiná libovolná data uložená v assembly a přístupná pomocí identifikátoru
- Kód assembly v jazyce MSIL (Microsoft Intermediate Language) – což umožňuje použití na kterékoliv platformě schopné MSIL interpretovat nebo přeložit do nativního kódu
- Metadata typů – popisuje všechny (veřejné/exportované) typy, které se vyskytují v assembly
    - To mimo jiné znamená, že pro použití knihovny není třeba žádných hlavičkových souborů (C/C++), nebo definic rozhraní (COM)
    - Protože je popis rozhraní definován platformou, není problém užít knihovnu z jiných jazyků

Reflexí se rozumí sada objektů, které umožňují dynamický přístup k metadatům uloženým společně s assembly. Tyto třídy jsou uloženy ve jmenném prostoru "System.Reflection". Pomocí reflexe se za běhu programu dají zjistit i velmi podrobné detaily o objektech:

- Jméno typu
- Předci
- Implementovaná rozhraní
- Členské proměnné
- Vlastnosti
- Metody
- Atributy
- Typy dosazené do šablony

Reflexe se ovšem nespokojuje s pouhým popisem metadat, ale umožňuje jejich použití (opět za běhu aplikace):

- Tvorba instance typu, jehož popis vlastníme
- Volání metod
- Nastavování vlastností a členských proměnných
- Tvorba nových typů

3.6 C# a přetěžování operátorů

Tvůrci jazyka se při jeho vytváření většinou snaží o to, aby uživateli znemožnili vytvářet určité typy konstrukcí. Jedná se zejména o konstrukce které by byly potenciálně náchylné k chybám, případně se neshodovaly s názorem tvůrců na správný programátorský styl. Například vyjmutím příkazu "goto" se dá zabránit "špagetovému" stylu programování a donutit programátora zapisovat kód strukturovaně.

Jedním z dobrých příkladů sporných konstrukcí je přetěžování operátorů, převzaté do C# z jazyka C++. V C++ je přetěžování operátorů velmi benevolentní, naopak v C# poměrně omezené a například Java jej neumožňuje vůbec. Vedou se diskuze o vhodnosti přetěžování operátorů ve kterých jsou používány různé argumenty:

- Pro
  - Umožňuje velmi elegantní zápisy
  - Zvyšuje transparentnost objektů vzhledem k objektům jazyka (vlastní typ čísel lze také sčítat pomocí operátoru "+" a není třeba používat například "a.add(b))"
- Proti
  - Lze vytvářet sémantické nesmysly (operátor "-" který násobí)
  - Nejlépe jsou aplikovatelné na matematické objekty (které podporují většinu dostupných operátorů), těch je ovšem málo

Výsledkem podobných diskuzí bývá obecně přijímané tvrzení, že pokud se operátory přetěžují s rozumem, jsou dobrým nástrojem k tvorbě kvalitních programů. Jsou ovšem případy, kdy mají tyto nebezpečné konstrukce svůj smysl. Nejedná se o běžné použití programátorem, ale o tvorbu systémových knihoven. V nich lze ospravedlnit použití některých konstrukcí, protože se dá předpokládat, že je bude psát člověk s hlubokou znalostí problematiky. Dále v textu rozvedu operátor, který by se hodil mi při tvorbě knihovny pro perzistenci. Já jsem narazil na jeden, pravděpodobně by se jich dalo najít více při tvorbě jiných typů knihoven.

Bylo by užitečné, kdyby programovací jazyk obsahoval dvě úrovně oprávnění – programátor a systémový programátor. Běžnému programátorovi by pak nebyly přístupné některé aspekty jazyka. Toto rozdělení ale virtuálně přece jen existuje. Například v jazyce Java bylo potřeba nějakým způsobem implementovat operátor "+" na základních typech ("Integer", "String", "Double", …). Čili systémový programátor je k dispozici měl.

V jazyce C# lze přetěžovat většinu klasických operátorů ("= - + * / % () [] & |"), ale jazyk vytváří určitá omezení. Například při přetížení operátoru "==" je nutné zároveň přetížit operátor "!=". V tomto případě měl tvůrce na mysli 2 věci:

1. Optimalizace
   - Překladač se může rozhodnou použít i párový operátor bez strachu, že má jiný sémantický význam
2. Chyby uživatele

- Pokud uživatel přetíží pouze jeden operátor, druhý zůstane standardní – může docházet k velmi špatně odhalitelným chybám
    - U typu "Uživatel" může operátor "==" porovnávat přihlašovací jméno a podle něj usuzovat na ekvivalenci objektů. Předdefinované chování je ale porovnávání referencí – to by se uplatnilo při použití operátoru "!=".

Tato omezení nevnášela do mé práce žádné potíže, spíše mi pomáhala. Bohužel jsem ale narazil na operátor který přetížit nejde, což vedlo k určitým nedokonalostem při použití knihovny. Jak píši v kapitole "Návrh", byla jedním z mých cílu transparence objektů. Čili aby použití perzistentních objektů vypadalo jako použití jakýchkoliv jiných objektů.

3.6.1 Přetížení operátoru "."

Abych mohl popsat problém, budu muset předběhnout kapitolu návrh a popsat typ atributu, který bude sloužit k ukládání vlastností objektů. Jedná se o šablonový typ "Property<T>", který jako parametr přijme jakoukoliv třídu. Má přetížen implicitní operátor přetypování na typ T. Proto se může použít na místě, kde se očekává T. Problém ovšem nastane při přístupu k atributům a metodám. Například:

```
Property<String> p = "test";
String s = p;
int length = p.Length;    //chyba – hledá se metoda/atribut šablony Property, ne metoda typu T (String)
int length1 = ((String)p).Length;            //explicitní přetypování zabere
```

Explicitní přetypování ovšem ničí transparentnost použití objektů. Pokud by šel přetypovat operátor ".", bylo by možné jej nastavit tak, aby vracel typ T a transparentnost by zůstala zachována. Je ovšem třeba zmínit, že pokud by měl být operátor "." přetypovatelný, musel by jazyk obsahovat rozšíření, kterým by šlo přistupovat k původnímu typu.

3.7 Parciálně definované třídy

Tato vlastnost ("parcial classes") jazyka C#, uvedená ve verzi 2 umožňuje definovat třídu v několika souborech – po částech. Toho se využívá například při navrhování formulářů, kdy v předchozích verzích editoval návrhář rozhraní (program) stejný soubor se zdrojovým kódem jako programátor (člověk). Nyní je třída formuláře rozdělena na 2 soubory. Jeden pro programátora a jeden pro návrháře rozhraní.

Já tuto možnost používám pro rozdělení dlouhých zdrojových souborů. Například třída "Persistent" je velmi rozsáhlá a proto je rozdělena na:

- "persistent.cs" - společné metody a všechny atributy

- "persistent_save.cs" - metody související s ukládáním objektů
- "persistent_load.cs" - metody související s načítáním objektů
- "persistent_delete.cs" - metody související s mazáním objektů

3.8 Modifikátor přístupu "internal"

Jako programátor, který napsal většinu kódu v C++ jsem velmi ocenil schopnosti tohoto modifikátoru. Zejména při omezování metod objektů. Často se totiž vyskytnou objekty, které mají být součástí volně dostupného rozhraní třídy, ovšem obsahují metody, které by neměly být volány uživatelem knihovny, ovšem je potřeba je zpřístupnit pro vnitřní objekty knihovny. Toho nelze dosáhnou pomocí modifikátorů "private" a "protected", pokud nechcete použít dědičnost. Jedinou možností v C++ je použití "friend" deklarací, ovšem tato cesta není doporučována.

Použitím tohoto modifikátoru vznikají dokonalá rozhraní, která nenabízejí žádné metody, kterými by uživatel knihovny mohl zneužít pro vytvoření nevalidního (invariantního) vnitřního stavu knihovny.

4 Cíle

Cílem diplomové práce by měla být knihovna napsaná v jazyce C#, která usnadní perzistenci objektů v C#. Primárním úložištěm budou relační databáze, ale knihovna by měla být připravena i pro ukládání v jiných úložištích – XML, objektové databáze.

V následujících odstavcích popíši požadavky, které jsem si vytyčil pro svou knihovnu. Vycházejí jednak z mých zkušeností, ale také ze studia existujících knihoven. Jsou to zejména tyto:

- všechny perzistenční informace uložené v kódu
- podpora OOP vztahů perzistentních objektů
  - dědičnost
  - kolekce (vazba 1:N)
  - reference (vazba 1:1)
- abstrakce úložiště perzistence
- přehledná a definovaná struktura úložiště
- transparentnost a elegance použití
- události v životech objektů
- objektového dotazy bez literálů sloužících k identifikaci typů
- další pokročilé rysy
  - verzování
  - odložené načítání
  - generování/úpravy schématu databáze
  - transakce
  - identifikace instance

4.1 Všechny perzistenční informace uložené v kódu

Jedním z hlavních rysů knihovny by mělo být uložení veškerých perzistenčních informací přímo ve zdrojovém kódu aplikace. Tento rys činí knihovnu odlišnou například od knihovny NHibernate, která používá externí XML dokumenty k uložení pomocných dat.

Důvodem je snaha integrovat všechny údaje potřebné pro perzistenci do jednoho místa – to je pomáhá udržet konzistentní. Pokud totiž uživatel změní údaje pouze v jednom z míst, musí myslet i na změnu v druhém dokumentu. Další výhodou je, že uživatel nemusí distribuovat další soubory se svou přeloženou aplikací (nebo je schovávat – šifrovat, vkládat do přeložené aplikace, …).

Data pro perzistenci by mělo jít získávat pomocí reflexe. Pro uložení údajů pro perzistenci by se mělo co nejvíce používat nativních prostředků jazyka, nikoliv popisů uložených v literálech. Tedy nikde by se nemělo objevit jméno třídy či atributu ve formě literálu. To platí zejména pro:

- Objektové dotazy
- Přístup k atributům

Není třeba mít obavy z nesnadné tvorby dokumentace – jednou ze služeb knihovny by mohl být export popisů perzistentních tříd a vazeb mezi nimi do souboru, který by šel otevřít v běžném UML editoru (UML eXchange Format, …).

4.2 Podpora OOP vztahů perzistentních objektů

4.2.1 Dědičnost

Pokud si uživatel vytvoří perzistentní třídu, a rozhodne se vytvořit potomka dané třídy děděním, měla by mu knihovna maximálně ušetřit práci. Uživatel pouze přidá nové atributy do poděděné třídy. Knihovna by měla uživatele maximálně odstínit od implementačních detailů uložení potomkových objektů do databáze – je na ní, jakou strategii vybere. Zároveň by uživatel neměl být omezen maximálním počtem dědění perzistentních objektů a neměla by utrpět rychlost řešení.

4.2.2 Kolekce

Kolekce objektů se budou v knihovně objevovat ve dvou případech:

- Jako atribut třídy
  - Podpora relací 1:1, 1:N
- Jako výsledek objektového dotazu
  - Načítání včetně děděných tříd

4.3 Abstrakce úložiště perzistence

Uživatel knihovny by neměl mít pocit, že knihovna je jen jakési zjednodušení přístupu k datům pomocí SQL. Naopak by mu měla knihovna pomoci oprostit se od relačních databází – v našem případě je SQL relační databáze pouze nástrojem (jedním z možných) pro uložení dat objektů. Data mohou být stejně dobře uložena pomocí XML, objektové databáze, textových souborů, binárních souborů, atd. Odrazující příklad použití knihovny "Gentle.NET":

```
[TableName("Users")]
```

```
public class User
{
    private int userId;
    private string userName;

    public User( int userId, string userName )
    {
        this.userId = userId;
        this.userName = userName;
    }

    [TableColumn("UserId"), PrimaryKey]
    public int Id
    {
        get{ return userId; }
        set{ userId = value; }
    }

    [TableColumn(NotNull=true)]
    public string Name
    {
        get{ return userName; }
        set{ userName = value; }
    }
}
```

Za povšimnutí stojí výrazy SQL jako "PrimaryKey" a "NotNull" zavlečená do popisu perzistence. Například "PrimaryKey" je výrazně implementační záležitost a při perzistenci do XML vůbec nemusí dávat smysl (například při vazbě 1:N mohou být objekty uloženy jako podstromy majitelského objektu). Místo "NotNull" by bylo vhodnější uvést "MustBeDefined". To už je ale věcí názoru na nomenklaturu.

Jako každá abstrakce, i tato, snižuje výkon řešení a omezuje uživatele v použití specializovaných vlastností úložiště. Některá rozhodnutí uživatele musí být postihnuta logikou perzistenční vrstvy a nemohou proto dosáhnout výkonu uživatelem vytvořeného ad-hoc řešení ("triggery", uložené dotazy, uložené procedury). Záleží ovšem na schopnostech perzistenční vrstvy. Existuje podobná analogie mezi psaním assemblerového kódu programátorem a tvorbou překladačem

vyššího jazyka. Překladač generuje lepší běžný kód, než většina programátorů, ale v případě nějakého specializovaného výpočtu může programátor vytvořit lepší kód (apriorní znalosti, vyšší inteligence).

4.4 Přehledná a definovaná struktura úložiště

Pokud bude knihovna například provádět perzistenci objektů do relační databáze (nejen), musí být exaktně definováno, do jakých struktur se budou ukládat data. Do jakých tabulek se bude ukládat objekt, jak se uloží jeho potomci, jak se uloží kolekce a vazby (1:1, 1:N). Tím se ulehčí programátorovi tvorba ad-hoc dotazů pomocí SQL. Knihovna by ovšem měla být natolik mocná, aby nebylo vůbec potřeba SQL dotazy vytvářet.

4.5 Transparentnost a elegance

Knihovna musí být při použití velmi elegantní, zejména nesmí uživatele nutit vytvářet zdlouhavé a nepřehledné konstrukce. Dále by měla být většina možných chyb ošetřena překladačem – to by mělo být zajištěno tím, že uživatel nebude nucen používat literály k identifikaci. Má představa o téměř ideálním perzistentním objektu je asi takováto (toto je celá definice, tedy žádné dodatky v XML):

```
class CCustomer : CPersistent
{
  public Collection<CAddress> Addresses;
  public Property<int> Age;
  public Property<string> Comment;
  public Property<string> Name;
}
```

Použití takto definovaného objektu by mělo být opět velmi snadné:

```
CAdress addr = new CAdress();
addr.Street = "Mifkova";

CAdress addr1 = new CAdress();
addr1.Street = "Rybkova";

CCustomer cust = new CCustomer();
cust.Name = "Pepa";
cust.Age = 14;
cust.Adresses.Add(addr);
cust.Adresses.Add(addr1);
```

```
cust.save();

CCustomer cust1 = new CCustomer();
cust1.load(cust.Id);

Filter<CCustomer> filter = new Filter<CCustomer>();
filter.Condition = (filter.Def.Name=="Jarek" || filter.Def.Age>5);
foreach (CCustomer c in filter.loadCollection())
{
	System.Console.Write(c.Name);
}
```

Z ukázky je vidět, že se s objektem pracuje stejně jako s jakýmkoliv jiným objektem vytvořeným v jazyce C#. Pro přístup k atributům a tvorbu objektových dotazů nejsou použity literály. Tím mnoho případných chyb identifikuje už překladač.

4.6 Události v životě objektů

Knihovna musí poskytnou uživateli možnost specifikovat delegáty, které budou volány při důležitých událostech života objektu. Umožní to uživateli kontrolovat validitu objektů, případně přidat funkcionalitu závislou na události. Například po vytvoření zákazníka se mu automaticky vystaví první poplatek za registraci. Jsou to zejména tyto události:

- Před a po uložení
- Před a po vytvoření
- Před a po zrušení
- Po načtení
- Při validaci

4.7 Další pokročilé rysy

4.7.1 Verzování

Knihovna by měla umožňovat uchovávání historie objektů pomocí verzování. Verzování musí být automatické a transparentní. Také musí být prostorově nenáročné.

4.7.2 Odložené načítání

Pokud jsou v databázi pouze jednoduché objekty a je jich málo, pak není problém je pokaždé všechny načíst z databáze a vytvořit z nich objekty v paměti. Pokud je objektů příliš mnoho nebo jsou velké, nastává problém. Odložené načítání spočívá v načítání pouze těch nejdůležitějších údajů (id, verze). Až pokud se objekt opravdu použije, jsou transparentně načtena všechna data objektu.

Odložené načítání se týká zejména kolekcí. Uplatní se zejména v úlohách typu: Kolekce zákazníku, z nichž každý má kolekci faktur. Rozhodneme se, načíst kolekci všech zákazníků, protože se nám v ní bude dobře indexovat – nemusíme tvořit dotazy pro každého zákazníka zvlášť. Pokud nepoužijeme odložené načítání kolekcí, bude načtení kolekce zákazníku znamenat načtení všech údajů o nich, čili i všech faktur každého zákazníka. To bude trvat velmi dlouho a navíc ani nevíme, zda budeme opravdu potřebovat všechny faktury všech zákazníků. Pokud použijeme odložené načítání, pak se kolekce faktur načte z databáze až ve chvíli, kdy se k ní pokusí uživatel přistoupit.

Odložené načítání řeší problém pomalého načtení po dotazu, ovšem přidává jiný – problém pomalých reakcí při práci s načtenými objekty. Pokud se totiž data načtou kompletní, nehrozí žádná režie při jejich použití. Pokud je načtení odloženo, může vzniknout nežádoucí prodleva při prvním použití.

4.7.3 Generování/úpravy schématu databáze

Protože úložiště by mělo být abstraktní, musí knihovna zajistit automatické tvoření jeho struktury. V případě databáze tvorbu tabulek, jejich sloupců a vazeb. Pro XML úložiště může vytvořit DTD popis- V ideálním případě by knihovna rovněž měla reagovat na změny tříd změnou struktury (přidání atributu objektu, přidání objektu, …). Pokud by ovšem hrozila ztráta dat, musí knihovna výjimkou požádat uživatele o explicitní potvrzení (odebrání vlastnosti objektu – ztráta dat).

4.7.4 Transakce

Pokud se zaměříme na podporu transakcí které by měla knihovna podporovat, můžeme identifikovat dva typy. Knihovna by měla podporovat oba přístupy.

První typ je implicitní a měl by zajistit, aby byly všechny operace s objekty vnitřně atomické – tedy buď se provedou a změny se uloží, nebo se provádění nepodaří a databáze se nezmění. Například v případě ukládání objektu s kolekcí to znamená, že se uloží všechny objekty z kolekce i objekt, nebo nic.

Další typ transakcí je explicitní. Zajišťuje atomičnost na úrovni několika operací nad objekty. Jejich začátek a konec určí uživatel. Například při převodu peněz z účtu na účet se transakce smí provést jen v případě, že se podaří uložit oba objekty účtů.

4.7.5 Identifikace instance

Každá instance třídy by měla mít unikátní identifikátor v rámci databáze (v duchu paradigmat objektových databází), nebo alespoň třídy. Verze s jedinečností v rámci třídy bude rychlejší, protože nebude obsahovat dlouhý jedinečný identifikátor, či základní tabulku všech typů. V našem případě bude jedinečný identifikátor v rámci třídy dostatečný.

Pomocným identifikátorem může být verze objektu – po každé změně v objektu je uložena nová verze s novým číslem verze. Objekt se pak identifikuje složeným klíčem [id, číslo verze].

4.7.6 Objektové dotazy

Pro perzistentní objekty je důležitá otázka načítání kolekce objektů jedné třídy podle zadané podmínky (testující atributy objektu). Existují dva protichůdné požadavky:

- Abstrakce úložiště – možné použití různých typů úložišť za cenu snížení výkonu (použití pouze podmnožiny možných funkcí)
- Efektivita dotazů - načtení celé kolekce objektů do paměti a zkoumání jejich atributů není efektivní

Řešením je výběr funkcí společných pro různé úložiště a vytvoření mapování z těchto funkcí na funkce úložiště. Toto mapování je potřeba vytvořit pro každé úložiště zvlášť (SQL, XQuery, XML-QL, …). V existujících řešeních se objevují tři způsoby vytváření těchto dotazů.

- Textové s nahrazováním parametrů (JDO – Java Data Object, Hibernate)
  - Samotný dotaz je tvořen řetězcem, ve kterém jsou povoleny názvy atributů objektu, operátory a parametry (ty se nahradí při použití podle aktuálně definovaných hodnot)
  - Pseudokód:

```
Query q = new Query(A.class, "Name='$Par1' AND ISNULL(Surname) ");
q.setParam("$Par1", "Jarek");
Coll coll = new Coll(q);
```

  - Potíž je, že se řetězec musí rozparsovat na jednotlivé výrazy a ty přeložit do výsledného dotazovacího jazyka a navíc je zápis dost nepřehledný
- Procedurální
  - Dotaz je vytvořen postupným voláním metod
  - Pseudokód:

```
Query q = new Query();
q.operatorEq("Name", "Jarek");
q.and();
q.isNull("Surname");
Coll coll = new Coll(q);
```

  - Výhodou je, že se chyba parametrů (např. jméno neexistujícího atributu) je odhalena ihned při volání parciální tvořící funkce
- Překládaný dotaz
  - Dotaz je tvořen zápisem podmínky přímo v cílovém jazyce
  - Dotaz je v rámci možností kontrolován překladačem

```
Filter<CCustomer> filter = new Filter<CCustomer>();
filter.Condition = (filter.Def.Name=="Jarek" && filter.Def.Surname==null);
foreach (CCustomer c in filter.loadCollection())
{
        System.Console.Write(c.Name);
}
```

  - Protože "Filter<T>" je šablonová třída, definuje atribut typu T, na kterém jde pomocí operátoru "." vybrat atribut dotazované třídy. Přetížený operátor "==" (a ostatní porovnávací operátory) pak převede tento zápis na podmínku.

Cíl je jasný: „Maximální abstrakce od konkrétních úložišť při maximálním výkonu a maximální vyjadřovací schopnost takových dotazů.“

4.7.7 Objekty a vyrovnávací paměť

Většina perzistenčních knihoven tuto techniku nepoužívá. I já jsem se rozhodl její podporu neimplementovat. Důvody, proč se nepoužívá jsem našel 2:

- Knihovny nemohou odhadnout, jestli bude přístup k datům pouze jednouživatelský, nebo ne. Proto se nechává na uživateli, aby si vyrovnávací paměť vytvořil sám. Pokud by totiž objekty v databázi změnil, staly by se ty v paměti zastaralé a pokus o změnu a uložení jejich stavu by znamenal výjimku.
- Navíc pokud si vypůjčíme koncept navigace po objektech z objektových databází, není vyrovnávací paměť často vůbec potřeba, protože uživateli postačí držet si referenci na objekty, na nichž navigace začíná (kořeny navigačních stromů) a tím v paměti zůstanou všechny objekty, přes které už se někdy navigoval. V tomto případě je ovšem potřeba implementovat odkládání nepotřebných objektů po čase, protože jinak by mohlo být v paměti příliš mnoho dat.

4.7.8 Inicializace vlastností, referencí a kolekcí

Aby bylo definování objektů přímočaré, ale zároveň nebylo potřeba specifikovat hodnoty všech vlastností, referencí a kolekcí, nastavují se hodnoty nenastavených atributů automaticky. Vytvoří se pro ně nový objekt s vnitřní hodnotou "null". Např.

```
class Person : Perzistent
{
	Reference<Address> Adress;  //C# inicializuje tuto vlastnost na "null"
}

Person p = new Person();
if ( p.Address.ref_obj==null )
{
	//todo
}
```

Pokud by nebyla automatická inicializace implementována, pokus o přistoupení k "p.Address" by skončil pádem aplikace, protože p.Address by byla automaticky inicializována na hodnotu "null".

5 Návrh

Návrh klade důraz na velmi důkladnou analýzu domény problému za podpory modelovacích technik UML. Návrh musí pokrýt hlavní problémy a určit způsoby řešení. V ideálním případě zachytí i problémy podružné, i když ne všechny – ty ale nejsou podstatné pro úspěch projektu. Čtenář této části by měl mít na paměti, že jsem mnoho témat rozebral a mnoho rozhodnutí učinil už v kapitole "Cíle" tohoto dokumentu.

5.1 Všechny perzistenční informace uložené v kódu

Všechny potřebné informace k perzistenci jsou popsány pomocí prostředků jazyka C# bez použití jakýchkoli literálů, které by nešly kontrolovat pomocí překladače. Původně jsem se domníval, že budou použity atributy ("Attribute" třídy v .NET), ale při návrhu jsem nenašel problém, který by nešel lépe popsat bez jejich pomoci.

5.2 Podpora OOP vztahů perzistentních objektů

5.2.1 Dědičnost

Mapování dědičnosti do relační databáze se dá zařídit v podstatě 3 způsoby (v našem modelovém příkladu existuje objekt Zvíře s atributy [Jméno, Pohlaví] a zděděný objekt Pes s atributem [Rasa]):

- Každá třída má vlastní tabulku – ta obsahuje všechny sloupce vlastní a v případě poděděných i poděděné. Při načítání kolekce předků včetně potomků se pak musí projít všechny tabulky vzniklé děděním dané třídy. Příklad:

zvíře:

id	jméno	pohlaví
1	Pepa	on

pes:

id	jméno	pohlaví	rasa
23	Čuk	on	husky

- Každá třída má vlastní tabulku – ta obsahuje vlastní sloupce a sloupce referencí na děděné objekty. Pokud buňka v tomto sloupci obsahuje hodnotu NULL, jedná se o předka, jinak o potomka (a ten má v referované tabulce jen vlastní sloupce). Jeden objekt je tedy rozložen do více tabulek. Příklad:

  zvíře:

id	jméno	pohlaví	ref_pes
58	Jolana	ona	NULL
59	Asta	ona	25

  pes:

id	rasa
25	doga

- Všechny třídy v hierarchii dědičnosti v jedné tabulce – atributy všech tříd v hierarchii dědičnosti sdílí jednu tabulku. Třída objektu je určena hodnotou speciálního sloupce s výčtem typů. Příklad:

  zvíře a pes:

id	jméno	pohlaví	rasa	typ
0	Žeryk	on	špicl	Class_Pes
1	Esmeralda	ona		Class_Zvíře

Z možných variant jsem si vybral druhou, tady přidané sloupce při dědění do zvláštní tabulky. Tato technika je podobná implementaci dědičnosti v programovacích jazycích a přijde mi nejvíce intuitivní.

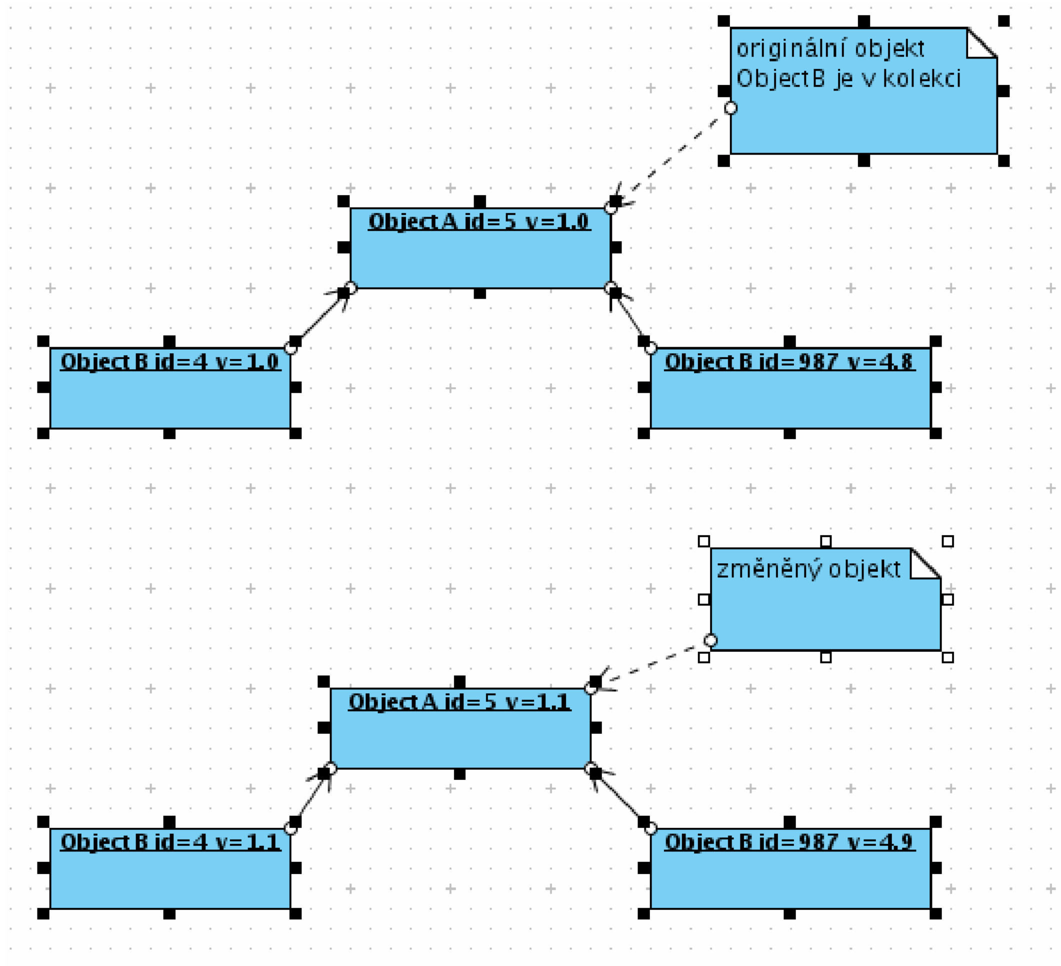
5.2.2 Kolekce

U kolekcí vznikají hlavní problémy díky verzování. Při výběru nevhodné strategie můžou vzniknou problémy s objemem dat, případně ztrátou části historie. Navíc, pokud je to možné, musí být zapnuta referenční integrita kontrolovaná databázovým strojem a všechny kritické operace musí být implicitně prováděny v transakci. Napadají mě 3 strategie ukládání kolekcí:

5.2.2.1 Kompletní historie

Nová verze objektu obsahujícího kolekci způsobí vyrobení nových verzí objektů v kolekci, které jsou referovány novou verzí:

- hodně dat
- nebezpečí rekurzivního verzování – změna způsobí změnu verze objektu v kolekci a ta vyvolá novou verzi objektu s kolekcí, …vznikne při křížové referenci
- 100% historie

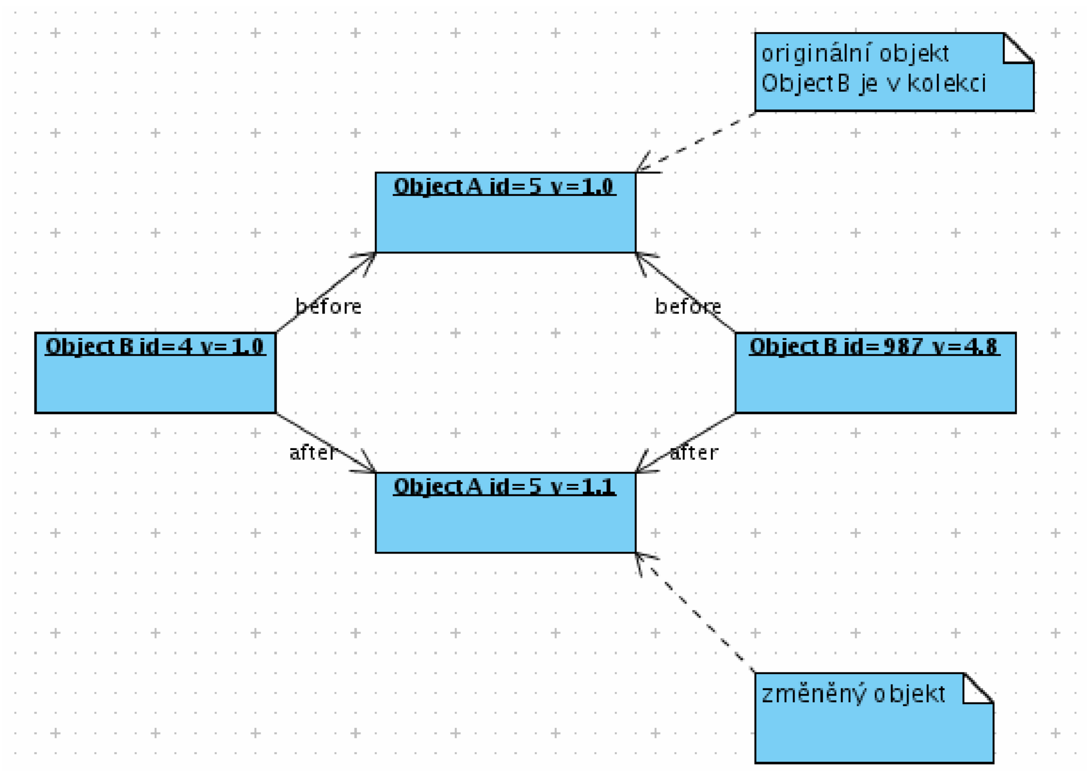


Obr. 5-1 Ukázka vytváření nové verze

5.2.2.2 Historie objektů bez historie referencí

Nová verze objektu s kolekcí způsobí změnu referencí posledních verzí objektů v kolekci na novou verzi

- ztráta části historie
- menší velikost



Obr. 5-2 Ukázka vytváření nové verze

5.2.2.3 Vlastní způsob

Protože mi ani jedna z výše popsaných strategií nevyhovuje rozhodl jsem se vytvořit vlastní. Měla by umožňovat ukládání téměř kompletní historie včetně referencí ovšem bez velké datové režie. Každá kolekce (či reference) bude reprezentována tabulkou:

coll_order	from_id	from_version	to_id	to_version

Jméno tabulky je vytvořeno podle vzoru "obj_" + [jméno třídy na které je kolekce definována] + "_coll_" + [jméno kolekce] + "_ref_obj_" + [jméno referované třídy]. Tedy například "obj_project_coll_involved_ref_obj_employee". Všechny znaky jsou převedeny na malé. Význam jednotlivých sloupců je následující:

- "coll_order" – pořadí referovaného objektu v kolekci
  - Má význam pro kolekce 1:N kde definuje pořadí v kolekci (aby se objekty do kolekce načetly v pořadí ve stejném pořadí v jakém do ní byly uloženy)
- "from_id" a "from_version" – je cizí klíč do tabulky objektu, který kolekci obsahuje
- "to_id" a "to_version" – je cizí klíč do tabulky objektu, který je kolekcí referován

Při změně objektu s kolekcí (jeho atributů nebo kolekcí či referencí) se uloží nová verze objektu a vytvoří kopie předchozí verze kolekce (pouze referující na nejnovější verze objektů). Tedy například:

coll_order	from_id	from_version	to_id	to_version
0	5	0	33	1
1	5	0	65	7
0	5	1	33	1
1	5	1	65	8

Při změně referovaného objektu k žádné změně v referenční tabulce nedojde. Při načítání objektu pak může dojít ke dvěma situacím:

- Uživatel zadá id i číslo verze
  - Do kolekce se načtou přesně ty objekty, včetně verzí, které jsou uloženy v tabulce
- Uživatel zadá pouze id (chce načíst poslední verzi)
  - Do kolekce se načtou objekty referované v poslední uložené verzi kolekce, ovšem "to_version" je ignorováno a načtou se poslední verze objektů

5.3 Abstrakce úložiště perzistence

Abstrakce úložiště bude vyřešena pomocí použití knihovny ADO.NET – součásti .NET frameworku. Pomocí ní bude uživatel naprosto nezávislý na použitém databázovém stroji. Pokud se vyskytnou implementační potíže (různé verze standardu SQL) pak budou vyřešeny uvnitř knihovny. Tomuto stavu předejdu používáním co nejomezenější množiny SQL dotazů.

5.4 Přehledná a definovaná struktura úložiště

Tabulky se budou jmenovat stejně jako perzistentní třídy, ovšem zapsané pouze malými písmeny a předponou "obj_". První tabulka v hierarchii dědičnosti bude obsahovat jednoznačný identifikátor (název "_id") a identifikaci verze (název "_version"). ID a verze budou dohromady tvořit složený primární klíč. Dále bude obsahovat příznak smazání "_deleted". Tato jména budou vždy stejné.

V odvozených třídách bude primární klíč části objektu "_part_id" na který bude vytvořen cizí klíč v rodičově tabulce. Jméno cizího klíče bude začínat prefixem "inh_" a pokračovat jménem zděděné třídy. Názvy (atributů) sloupců budou tvořeny pouze malými písmeny (bude potřeba ošetřit kolizi jmen – "Jmeno" a "jmeno"). Na ukázce je vše demonstrováno:

"obj_person"

[PK]_id	[PK]_version	_deleted	name	surname	[FK]inh_employee
5	1	0	Jarek	Kuboš	58
6	1	0	Petr	Šťastný	NULL

"obj_employee"

[PK]_part_it	City
58	Vlčovice

5.5 Transparentnost a elegance

Transparentnost a elegance výsledné knihovny byla naznačena v části "Cíle" tohoto dokumentu.

5.6 Události v životě objektů

Objekty budou vybaveny možností předefinování virtuálních událostních metod (například "beforeSave"). Pokud se uživatel rozhodne události nepoužívat, použije se základní funkcionalita – prázdná metoda.

Uvažoval jsem, že k řešení tohoto problému použiji delegáty, ale práce s delegáty implikuje 2 přístupy:

- Delegát pro celou třídu (volat pro každý objekt)
- Delegát pro objekt (volat pro konkrétní objekt)

Pokud bych se rozhodl pro delegáty, musely by být deklarovány jako statické (1 delegát pro všechny objekty třídy) i členské proměnné tříd (delegát pro každý objekt zvlášť). To by přineslo zmatky při inicializaci. Navíc pokud se uživatel rozhodne delegáty přidat, není to pomocí připravených metod problém. Pokud se takto nerozhodne, ušetří si zápis.

5.7 Další pokročilé rysy

5.7.1 Verzování

Vztahy verzování k jednoznačnému identifikátoru a kolekcím už byly rozebrány výše. Při návrhu sytému verzování objektů jsem musel zohlednit tato kritéria:

- kvalita historie – lze opravdu vrátit každou změnu?
- smysl historie I. – lze vymyslet případ použití pro všechny možnosti obnovení historických verzí objektů? Například: „Při návratu objektu O na verzi X mají být jím referované objekty v poslední verzi, verzi ve které byly do referencí vloženy nebo ve verzi těsně před další změnou objektu O?“

- smysl historie II. - má být historie pouze informativní (zjištění předchozí hodnoty atributu) nebo bude použita pro zálohování dat (tj. obnovení kompletních dat pro definovaný čas, daný verzí objektu)?
- velikost databáze – pokud malá změna vyvolá uložení velkého množství dat do databáze, bude knihovna použitelná jen pro malý objem dat.
- časová náročnost – při uložení objektu v nové verzi se nesmí v provést mnoho dalších dotazů, jinak se stane knihovna nepoužitelnou pro pomalost.
- kruhové reference, zacyklení – při špatném návrhu by se mohlo stát, že se při vytváření nové verze objektu A vytvoří nová verze objektu B, protože je referovaná A. Jenže pokud i B referuje A pak vyvolá novou verzi A, což vyvolá novou verzi B, … Ošetřit takovouto chybu je problematické. Na její nalezení sice stačí nalézt smyčky v orientovaném grafu, jenže graf je tvořen objekty (ne třídami) a jeho načtení do paměti je by bylo neúnosným plýtváním zdroji.

5.7.1.1 Historické verze objektů

Při načtení historické verze objektu nesmí být možné změny na takovémto objektu uložit (existují novější verze). Proto musí být obnovení historického objektu realizováno jako uložení nové verze objektu s atributy nastavenými hodnotami atributů historické verze. Tím se také zaručí podmínka vratnosti všech operací nad objekty. Čili i návrat ke starší verzi musí jít vrátit zpět.

5.7.2 Odložené načítání

Odložené načítání je z implementačního hlediska velmi jednoduché díky použití "proxy" šablon (Property<T>, Reference<T>, Collection<T>). Při načítání objektu se pouze načte jeho jednoznačná identifikace (id+verze) a s načtením jeho členských proměnných (nejen atomické proměnné, ale i kolekce) se počká do prvního přístupu k nim. Zda už je objekt celý v paměti bude určovat příznak "_notYetLoaded". Díky tomu, že pro perzistentní atributy nebudou použity přímo jejich datové typy, ale instance šablony nebude problém vytvořit zpětnou vazbu zajišťující načtení objektu ve chvíli prvního přístupu k jeho hodnotě.

5.7.3 Generování/úpravy schématu databáze

Generování schématu databáze je relativně snadná operace, která se bude vykonávat po startu aplikace používající knihovnu. Kontrola korektnosti databáze se provede při nastavení továrního objektu pro práci s databází.

Všechna data potřebná k vytvoření schématu si aplikace získá pomocí reflexe objektů zděděných od základního objektu knihovny.

5.7.4 Transakce

Pro transakce jsem se rozhodl použít koncept nezanořovaných transakcí (zanořované nejsou široce podporovány v databázových strojích). K zanoření transakcí může dojít snadno – uživatel ukládá skupinu objektů z nichž každý, který obsahuje kolekci jiných objektů použije další – implicitní – transakci. Použití nezanořených transakcí je i logické – chceme uložit vše, nebo nic a obsluha neprovedených transakcí by stejně pravděpodobně zrušila i rodičovskou transakci. Těžko si představit, že by aplikaci postačil částečný úspěch při ukládání mnoha objektů.

Standardně se použijí implicitní transakce, které zaručují atomicitu operací typu "save". V rámci těchto implicitních transakcí se pak provede uložení celé hierarchie objektů. Pokud uživatel chce, může transakci specifikovat, což mu umožní zaručit atomicitu celého bloku operací.

5.7.5 Identifikace instance

Pro identifikaci instance jsem se rozhodl použít jednoduché, inkrementované celé číslo "INT(4) ". Stejný typ bude použit pro číslo verze. Pokud by v budoucnu vyvstala potřeba lepší identifikace (např. "GUID" – "Generated Unique ID") není problém ji doplnit.

5.7.6 Objektové dotazy

Pro načítání kolekcí jsem se rozhodl použít "překládané dotazy" vyložené v sekci 4.7.6. Bližší návrh schopností bude definován v UML diagramu tříd níže.

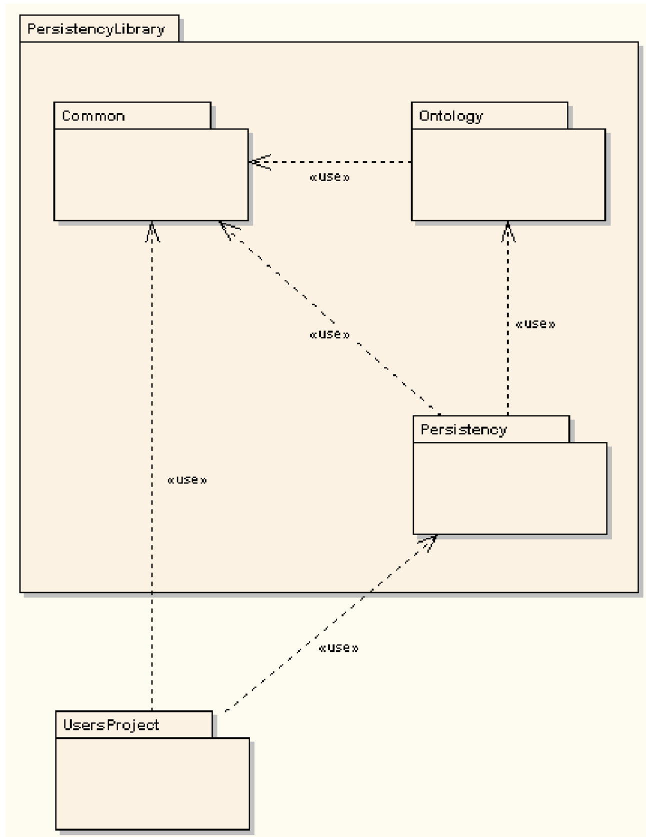
5.8 Objektový návrh

5.8.1 Schéma modulů

Při návrhu jsem identifikoval tři logické části knihovny, které jsem poté implementoval jako tři "assembly" (dll knihovny). Jsou to tyto:

- "Common" – tato část obsahuje třídy používané napříč aplikací
    - Typ výjimky
    - Pomocné třídy pro práci s ADO.NET
    - Základní nastavení za běhu
- "Ontology" – tato část obsahuje třídy použité k ontologickému popisu uživatelských dat
    - Popis perzistentních tříd
    - Popis relací mezi třídami – reference, dědění
    - Tvorba databázového schématu
    - Nastavení dostupných "assembly" s perzistentními třídami

- "Perzistence" – tato část obsahuje zejména typ perzistentního objektu, který musí dědit všechny podřízené objekty
    - Rodičovský perzistentní objekt
    - Filtry a podmínky pro objektové dotazy
    - Typy atributů – reference, kolekce a běžný datový atribut



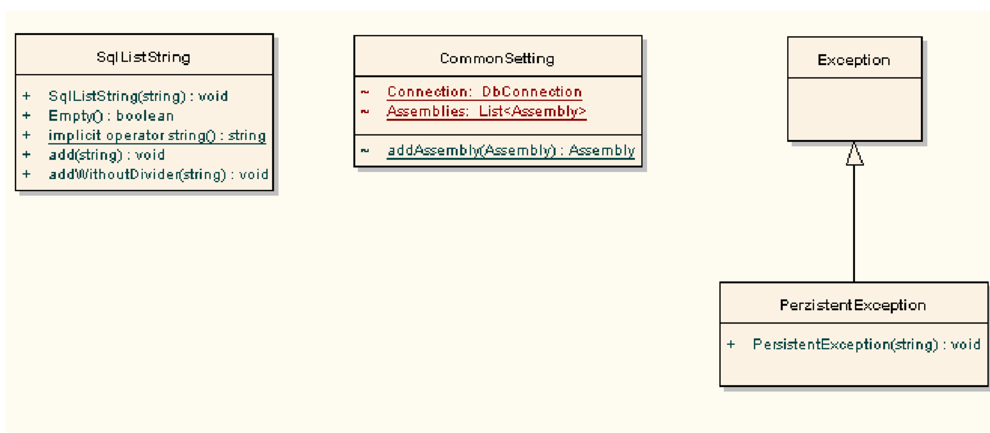
Obr. 5-3 Schéma modulů

5.8.2 Modul "Common"

Modul "Common" obsahuje" následující třídy:

- "SqlListString"
    - Třída pro ulehčení tvorby částí SQL dotazů
    - Provádí konkatenaci dodávaných řetězců pomocí nastaveného oddělovače
    - Vhodné pro výrazy typu "SELECT name, surname, age, city, street, id, version"
    - Zjednodušuje zápis, protože není pořád dokola explicitně nutné testovat, zda je řetězec ještě prázdný a nebo už se má použít oddělovač
- "CommonSetting"
    - Vně neviditelná třída, která zpřístupňuje obsahuje globální připojení k databázi a také seznam "assemblies" ve kterých se mají hledat perzistentní třídy
- "PerzistentException"

    - Třída výjimek, které mohou vzniknou n knihovně
    - Nepřináší žádnou novou funkcionalitu oproti výchozí třídě "Exception" ale dovoluje uživateli určit, zda je výjimka očekávaná (knihovna ví o chybě a dává ji najevo uživateli) nebo neočekávaná (chyba práce s nenastavenou referencí, chyba dotazu do databáze, …)



Obr. 5-4 Modul "Common"

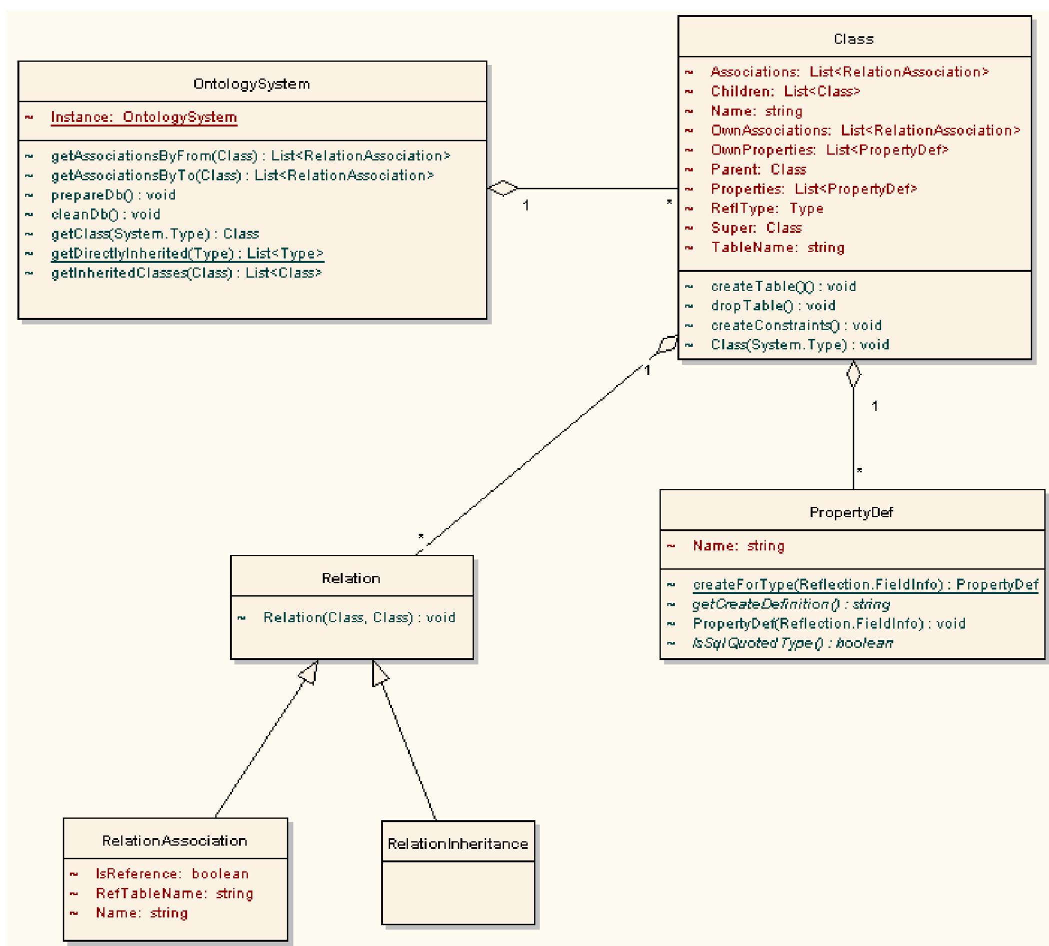
5.8.3 Modul "Ontology"

Modul "Ontology" obsahuje následující třídy:

- "OntologySystem" – objekt této třídy existuje v instanci aplikace jako singleton
    - Udržuje informace o dostupných perzistentních třídách a relacích mezi nimi
    - Vytváří a odstraňuje databázovou strukturu
    - Definuje pomocné metody pro prohledávání dostupných tříd
        - Hledání referencí když je známá referující třída
        - Hledání referencí když je známá referovaná třída
        - Hledání tříd, které jsou zděděné ze zadané třídy
        - Překlad systémového typu "Systém.Type" na "Class" objekt
- "Class" – třída která reprezentuje jeden typ perzistentního objektu definovaného uživatelem
    - Udržuje informace o atributech, třídách ze kterých dědí, třídách které ji dědí a referencích
        - Atributy a reference dovoluje rozdělit na vlastní (definované v této třídě) a zděděné
    - Dovoluje zjistit nejvyšší objekt v hierarchii dědičnosti této třídy
    - Vytváří a odstraňuje tabulky a omezení ("db constraints") pro tuto třídu
- "PropertyDef" a odvozené – tato třída definuje specifické chování různých primitivních datových typů při ukládání do databáze (sloupec jakého typu vytvořit, jak uložit hodnotu, …)
    - Třída "PropertyDef" obsahuje metodu která se pokusí na základě reflexního typu pole zkonstruovat správný odvozený objekt

- Pomocí odvození této třídy je možné ukládat do databáze uživatelská data (například georeferencovaná, objekty serializované do XML, …)

- "Relation" – tato třída definuje základní relaci – vztah dvou objektů
  - Sémantika není definována – jedná se vlastně o abstraktní třídu
  - Obsahuje reference na dva "Class" objekty (označení "from" a "to") – jedná se tedy o orientovanou hranu
  - Sémantický význam hrany je určen až v odvozených typech
- "RelationAssociation" – tato třída popisuje reference a kolekce – tedy vazby 1:1 a 1:N
  - Je možné zjistit zda se jedná o referenci nebo kolekci
  - Tvoří jméno referenční tabulky
- "RelationInheritance" – definuje pouze sémantiku relace



Obr. 5-5 Modul "Ontology"

5.8.4 Modul "Perzistence"

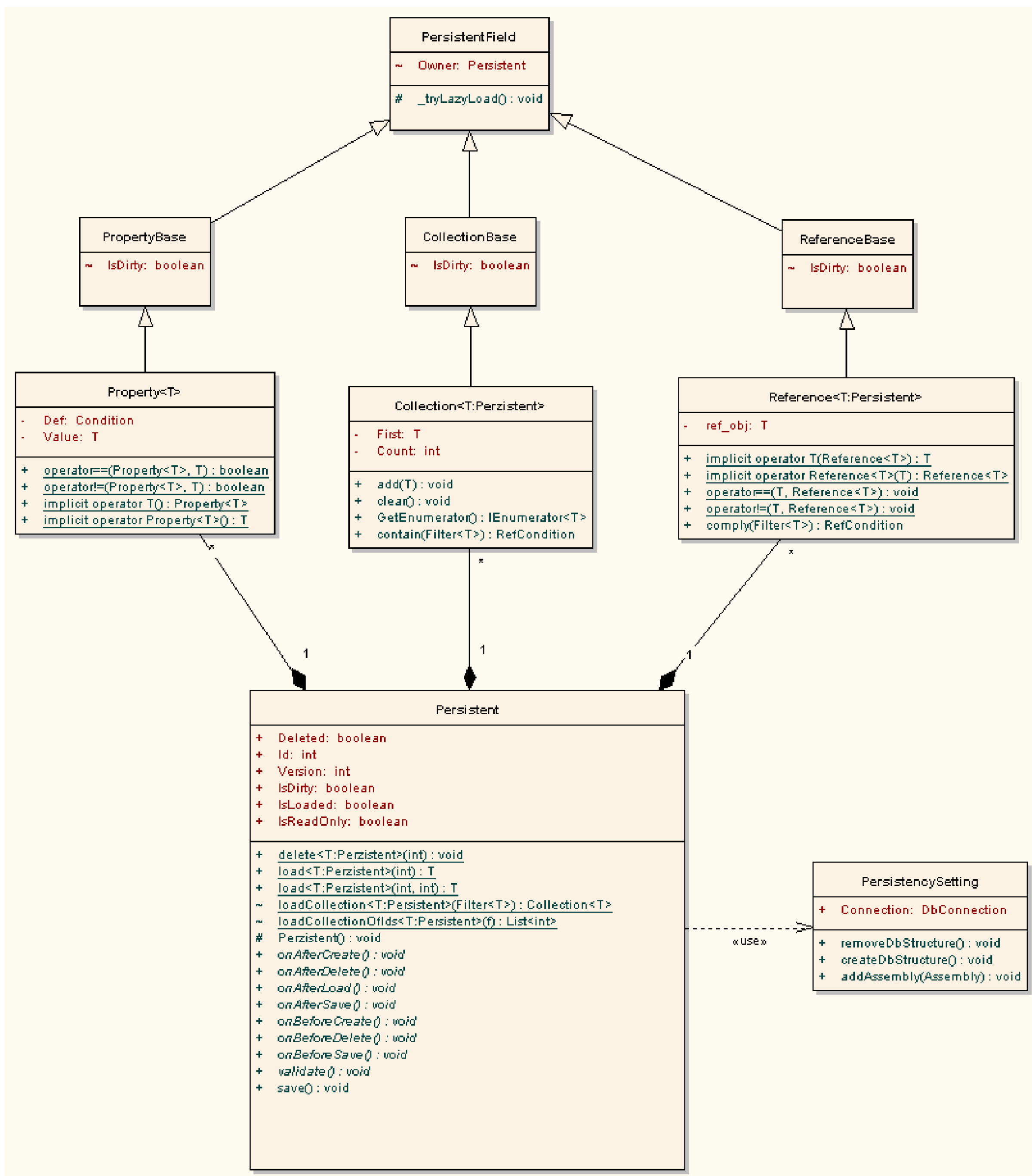
Tento modul poskytuje hlavní perzistenční služby a je jako jediný přístupný uživateli knihovny (vyjma typu výjimky z modulu "Common"). Lze jej rozdělit na dvě logické části:

5.8.4.1 Perzistentní objekt

Tato část definuje předka všech perzistentních objektů a atributy, které mu lze přidat. Obsahuje třídy:

- "PerzistenceSetting" – v instanci aplikace existuje jediný objekt této třídy – je to singleton
    - Umožňuje nastavení připojení k databázi
    - Umožňuje vytvoření a zrušení tabulek v databázi
    - Umožňuje přidávat "assembly" kde se mají hledat perzistentní třídy
    - Tento objekt je potřeba nastavit před započetím práce s perzistentními objekty
- "Persistent" – toto je třída, od které musí dědit všechny perzistentní objekty
    - Statické metody
        - Smazání objektu daného typem a identifikátorem
        - Načtení objektu daného typem a identifikátorem
            - Načítá objekty daného typu a podtypů (zaručí návrat správného typu)
        - Načtení objektu daného typem, identifikátorem a verzí
            - Načítá objekty daného typu a podtypů (zaručí návrat správného typu)
    - Metody
        - Uložení objektu
        - Obsluhy událostí a validace objektu
    - Vlastnosti
        - Identifikátor a verze
        - Příznak smazaného objektu
        - Příznak identifikující zda už je objekt uložen, nebo zda je nový
        - Příznak modifikace
        - Příznak modifikovatelnosti
- "Property<T>" – třída pomocí které jsou definovány perzistentní atributy
    - Typ do šablony lze použít libovolný, pouze je pro něj potřeba odvodit z typu "PropertyDef" novou třídu, která pro něj definuje způsob uložení v databázi
    - Definuje implicitní operátory z a na typ T
    - Definuje porovnávací operátory mezi T a Property<T>
        - user.Name=="Jarek"
    - Vlastnosti
        - "Def" umožňuje vytvářet podmínky do filtrů
        - "Value" je možno použít místo přetypování k získání vnitřní hodnoty
- "PropertyBase" – definuje základní, typově nezávislé operace pro typ "Property<T>"
    - Přidává kontrolu podmínek za běhu – mohou být vytvořeny jen pomocí definice z filtru, ne jinak
    - Umožňuje zjistit, zda byla vlastnost změněna

- "Collection<T:Perzistent>" – objekty této třídy jsou použity jednak jako atributy typu kolekce perzistentních objektů, ale také jako výsledky objektových dotazů
    - Umožňuje iteraci pomocí "foreach"
    - Obsahuje základní operace pro práci s kolekcí
    - Pomocí metody "contain" je možné filtrovat objekty podle toho, zda v kolekci obsahují hodnotu danou filtrem
- "CollectionBase" – obsahuje základní netypované metody a atributy pro kolekce
    - Obsahuje test změněnosti
- "Reference<T:Perzistent>" – třída pro realizaci vazby 1:1 mezi objekty
    - Přetěžuje porovnávací operátory == a !=, které porovnávají id a čísla verzí, nikoliv reference na objekty
    - Obsahuje implicitní konverze na typ T
    - Obsahuje metody pro filtraci podle atributů referovaných objektů "comply"
- "ReferenceBase" -
- "PerzistentField" - základní třída všech perzistentních atributů
    - Obsahuje referenci na vlastnící objekt
    - Umožňuje načíst objekt s odloženým načtením při prvním přístupu

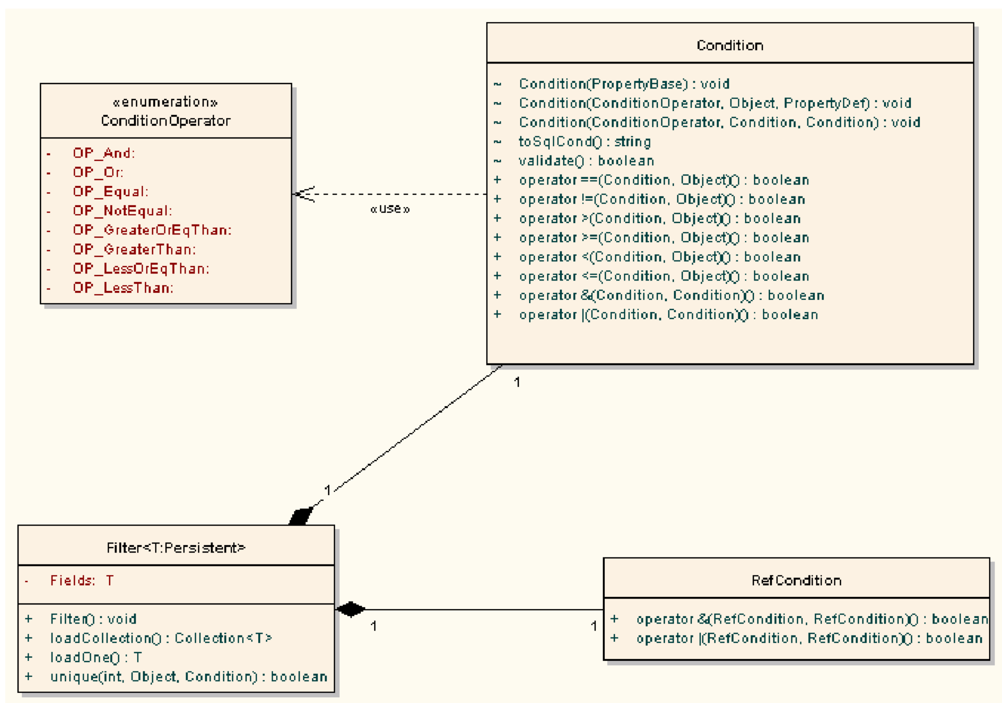


Obr. 5-6 Modul "Persistence"

5.8.4.2 Filtrování objektů

- "ConditionOperator" – výčet podporovaných operátorů v podmínkách
  - Lze snadno rozšířit na další operace
- "Condition" – podmínka založená na atributu třídy
  - Pro porovnávání hodnot atributů jsou přetíženy operátory >, <, >=, <=, ==, !=
  - Pro spojování podmínek jsou přetíženy operátory && a ||
  - Kontroluje se, zda vybrané atributy patří filtru pro který se podmínka definuje

- "RefCondition" – podmínka založená na filtru pro referovaný objekt
    - Získá se pomocí metody "comply" u reference nebo "contain" u kolekce
    - Přetěžuje operátory && a || pro skládání podmínek
        - Filtrování se provádí pro každou referenční podmínku zvlášť, protože se může obecně jednat o reference na různé typy
- "Filter<T:Perzistent>" – tato třída představuje jediný způsob jak načíst objekty jinak, než se znalostí id
    - Umožňuje načíst kolekci objektů vyhovujících podmínce
    - Umožňuje načíst jediný objekt vyhovující podmínce (0 nebo více než jeden nalezený objekt znamená chybu)
    - Dále obsahuje metodu na zjištění unikátnosti hodnoty atributu mezi objekty



Obr. 5-7 Filtrování objektů

6 Implementace

6.1 Prostředí, přenositelnost

Pro implementaci bylo použito Microsoft Visual Studio 2005 a jazyk C# ve verzi 2. Alternativně může být použito volně šiřitelné prostředí Mono, ovšem až po implementaci jazyka C# verze 2, na které se pracuje. Na použití v prostření Mono je knihovna připravena následujícím způsobem:

- Jsou používány pouze funkce ze jmenného prostoru "System"
- Nejsou použita volání Win32Api
- Konektory pro MySQL a SQLite databázi jsou multiplatformní
    - Pokud nejsou v cestě, kde se hledají knihovny, je potřeba opravit reference na ně v "GuiTest" projektu – dll knihovny jsou přiloženy
        - Pro reference bohužel nelze specifikovat relativní cestu

Z výše uvedených faktů lze předpokládat, že po dodělání prostředí Mono bude knihovna fungovat v něm a to i pod různými platformami (Linux). Tím se například při spojení se zabudovanou databází (třeba SQLLite) dá velmi dobře použít pro ukládání nastavení aplikací nezávislé na platformě.

6.2 Použitý databázový systém

Pro testovací účely jsem si vybral databázi MySQL a to ve verzi 5.0.3.7. Důvody pro tuto volbu byly ryze praktické.

- Nabízí .NET konektor
- Není náročná na hardware
- Není příliš komplexní ve smyslu nabízených funkcí – nesváděla mě k použití vyšších prostředků, které by na jiném databázovém stroji nebyly dostupné

Jedním z cílů, které jsem si vytyčil byla nezávislost na typu databáze. Tu zaručuje jednak připojení přes ADO.NET, což je nezávislost daná rozhraním společným širokému spektru databází. Další nezávislost – na jazyku SQL – je dána použitím co nejjednodušších konstrukcí, které navíc není problém upravit. Podle mých zkušeností je nejvíce rozdílů mezi databázemi v:

- DDL (data definition language)
    - Jiné názvy typů
    - Jinak pojmenované konstanty (true/false versus 1/0)
    - Jiný způsob definice primárních a cizích klíčů, případně jejich úplná absence

    - "Autoincrement" nastavení pro klíče
- Funkce které nejsou v SQL standardu
    - Zjištění posledního generovaného ID

Protože jsem chtěl vyzkoušet přizpůsobení na jinou databázovou platformu, rozhodl jsem se knihovnu upravit (po dokončení) také pro podporu vestavěné databáze SQLite. Nakonec jsem se rozhodl, že SQLite použiji jako výchozí databázi pro testovací aplikaci, protože zbaví uživatele nutnosti instalovat vlastní instanci MySQL databáze. Do třídy "CommonSetting" jsem přidal metody, které dovolují určit typ databáze a tomu přizpůsobit dotazy. Znamená to tedy, že knihovna je připravena pracovat s databázemi SQLite a MySQL. Typ použité databáze určuje pouze dodané připojení k ní. V knihovně bylo zapotřebí upravit dotazy asi na 15 místech. Celá úprava zabrala asi hodinu času.

7 Ukázka použití

Pro testování a demonstraci schopností knihovny jsem se rozhodl vytvořit okenní aplikaci. Jako téma aplikace jsem zvolil systém evidence zaměstnanců, projektů a jejich přiřazení. Několik následujících podkapitol poodhalí schopnosti knihovny.

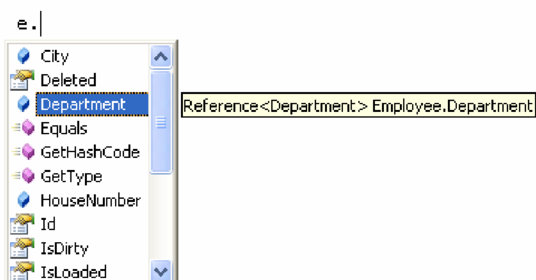
Protože by se mohl čtenáři zdát rozsah ukázek kódu v této kapitole nepatřičně velký, musím před jejím uvedením objasnit důvody proč tomu tak je. Domnívám se, že při psaní jakékoliv knihovny, kterou bude někdo používat je třeba mít na mysli dva aspekty:

- Funkcionalitu – knihovna musí naplňovat funkcionální podmínky uživatele a obsahovat minimum chyb
- Styl použití – použitím knihovny je uživatel donucen vnést styl jejího použití do svého kódu
    - Dobrým příkladem je čtení XML dokumentu – představme si rozdíl v použití knihoven na bázi SAX a DOM
    - Přitom tyto knihovny odvedou v zásadě tutéž práci

Výše zmíněné aspekty lze nejlépe demonstrovat příklady. Pozorný čtenář se při nich zamyslí a zkusí si představit, kolik by podle jeho zkušeností stálo práce, vytvořit je ručně, případně s použitím jemu známých prostředků. A také jak moc lahodí jeho smyslu pro elegantní zápis kódu (a to je něco, co se nedá popsat slovy – je potřeba to vidět v praxi). Budu se v nich snažit o maximální stručnost. Ta ovšem půjde zejména na vrub kontroly chyb, která v reálném projektu nechybí.

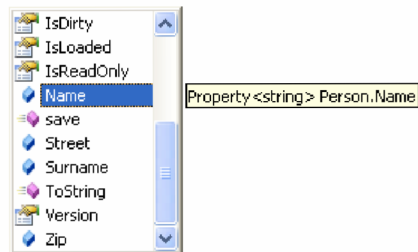
7.1 Práce s knihovnou ve Visual Studiu

Po definici perzistentních objektů má uživatel okamžitě přístup k jejich atributům pomocí funkce "IntelliSense". Nemusí si tedy názvy pamatovat.



Obr. 7-1 Vlastnosti objektu

Na tom ovšem není nic až tak zajímavého. Zajímavější je, že velmi podobným způsobem se dají psát objektové podmínky (přístup přes vlastnost "Fields"):

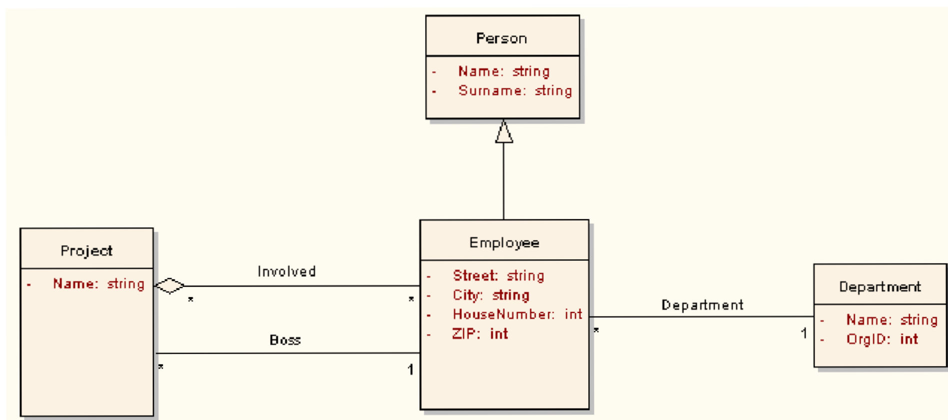


Obr. 7-2   Tvorba objektového dotazu

7.2 Definice perzistentních typů

Pro ukázku použití jsem si vybral velmi jednoduchý model tříd, jež bych chtěl učinit perzistentním. Obsahuje ale všechny požadované vlastnosti:

- Dědičnost
- Kolekce a reference
- Atributy



Obr. 7-3   Diagram tříd ukázky

Třída "Department":

```
class Department : Persistent
{
   public Property<string> Name;
   public Property<int> OrgId;
}
```

Třída "Person":

```
class Person : Persistent
{
    public Property<string> Name;
    public Property<string> Surname;
}
```

Třída "Employee":

```
class Employee : Person
{
    public Property<string> City;
    public Property<string> Street;
    public Property<int> HouseNumber;
    public Property<int> Zip;

    public Reference<Department> Department;
}
```

Třída "Project":

```
class Project : Persistent
{
    public Property<String> Name;

    public Reference<Employee> Boss;

    public Collection<Employee> Involved;
}
```

Z ukázek je vidět, že pokud by už objekty existovaly jako neperzistentní, bylo by potřeba učinit dvě drobné úpravy:

- Zdědit je z třídy "Perzistent"
- Všechny atributy, reference a kolekce zapouzdřit do vhodných šablon
    - Reference<T>, Collection<T>, Property<T>

7.3 Příprava použití knihovny

Před použitím knihovny je potřeba provést její inicializaci. Prvním krokem je předání spojení k databázi. Jak bylo výše zmíněno, vyhoví jakékoliv spojení s rozhraním ADO.NET.

```
SQLiteConnectionStringBuilder csb = new SQLiteConnectionStringBuilder();
csb.DataSource = "test.mdbf";
SQLiteConnection c = new SQLiteConnection(csb.ConnectionString);

/*
MySqlConnectionStringBuilder csb = new MySqlConnectionStringBuilder();
csb.Server = "localhost";
csb.Database = "test1";
csb.UserID = "root";
csb.Password = "jarek";
MySqlConnection c = new MySqlConnection(csb.ConnectionString);
*/

PersistenceSettings.Connection = c;
```

Druhý krok spočívá v uvedení "assemblies" ve kterých má knihovna hledat perzistentní třídy při vytváření ontologického popisu.

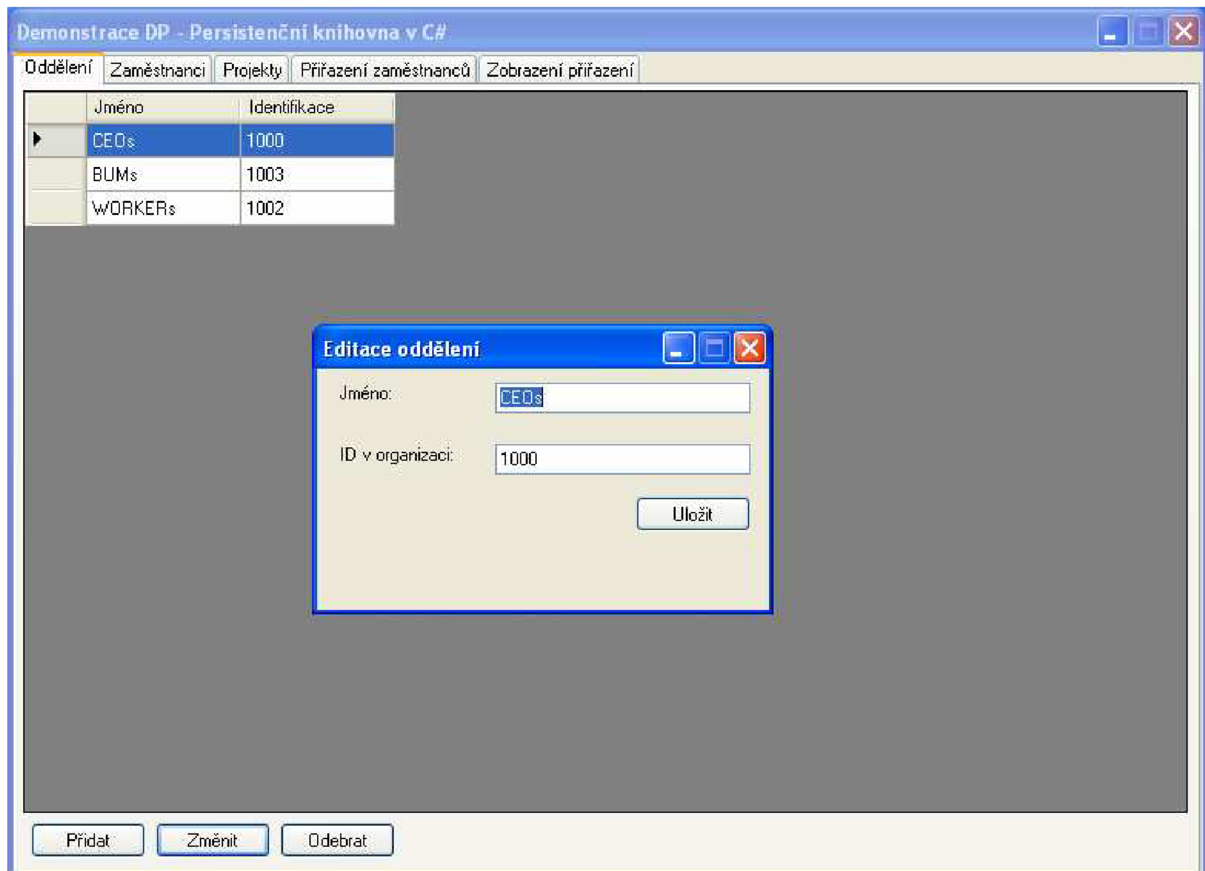
```
PersistencySettings.addAssembly(Assembly.GetExecutingAssembly());
```

Posledním krokem je vytvoření (nebo i zrušení) databázového schématu. Před jeho provedením už musí být nastaveno připojení k databázi a musí být specifikovány všechny používané "assemblies".

```
PersistenceSettings.removeDbStructure();
PersistenceSettings.updateDbStructure();
```

7.4 Záložka "Oddělení"

Na této záložce je odzkoušeno vytváření, měnění a mazaní oddělení. Třída reprezentující oddělení je primitivní třída, zděděná přímo z "Perzistent" a navíc nereferuje objekty žádné jiné třídy. Jedná se tedy o nejprostší možnou perzistenci.



Obr. 7-4 Ukázka editace oddělení

Z našeho hlediska zajímavé jsou dva kódy. Jednak načítání kolekce už uložených oddělení a jejich zobrazení. Tím se ukáže práce s atributy perzistentních objektů a základní objektové dotazy.

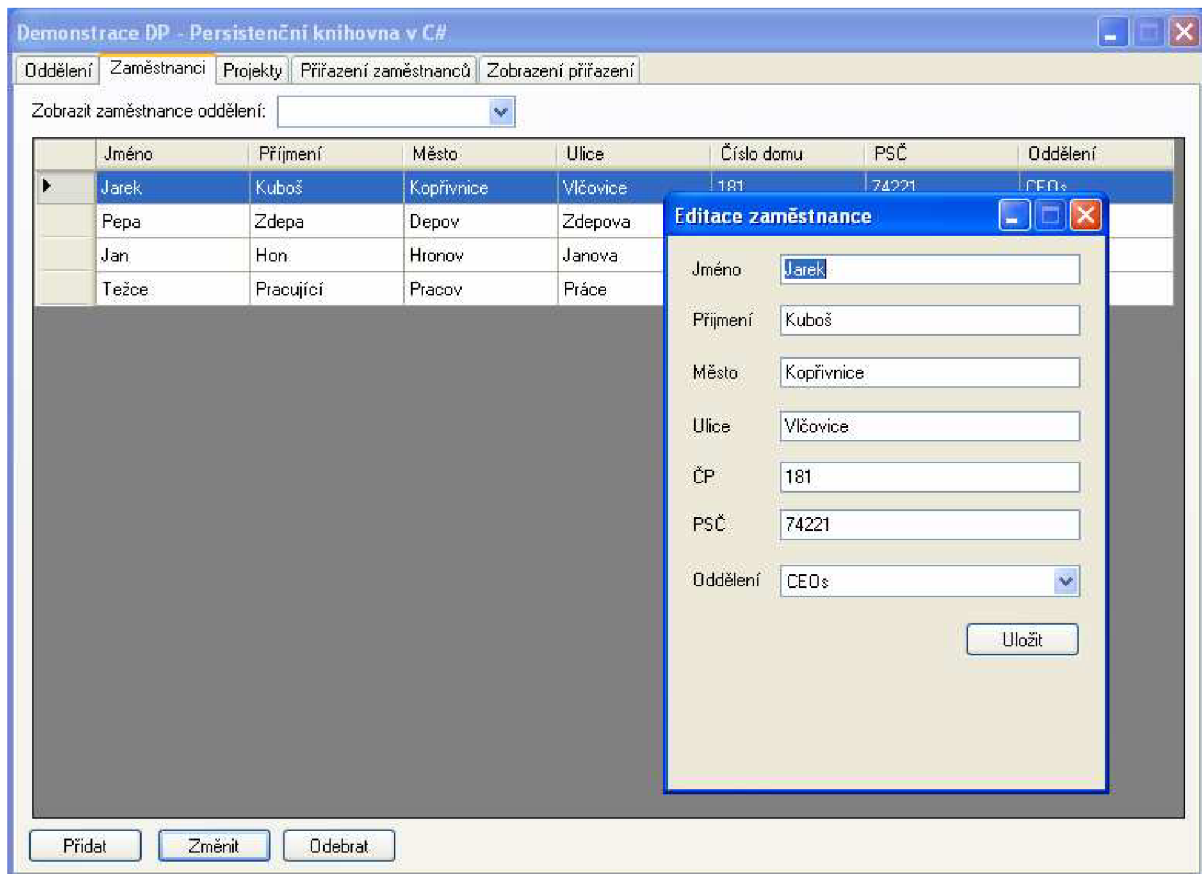
```
Filter<Department> f = new Filter<Department>();
foreach (Department d in f.loadCollection())
{
   Object[] data = new Object[]{d.Name, d.OrgId};
   departmentsTable.Rows.Add(data);
}
```

Dalším požadavkem je ukládání perzistentního objektu. Funkcionalita je ukryta ve formuláři, který funguje ve dvou režimech. Jednak vytváření nového oddělení ale také editace stávajícího:

```
if (_dep == null)              //pokud needituje stávající
{
    _dep = new Department();    //vytvoří nový
}

_dep.Name = depNameBox.Text;
_dep.OrgId = Int32.Parse(depIdBox.Text);

_dep.save();
```

7.5 Záložka "Zaměstnanci"

V této záložce je předveden způsob práce se složitějším objektem. Jednak je třída "Employee" zděděna z třídy "Person", ale navíc obsahuje referenci na oddělení. Dalším demonstrovaným rysem je možnost omezení výpisu zaměstnanců na zaměstnance přiřazené vybranému oddělení. Jedná se vlastně o podmínku na referovaný objekt. Jinými slovy: „Vyber všechny zaměstnance, kteří referují oddělení s názvem XYZ.“

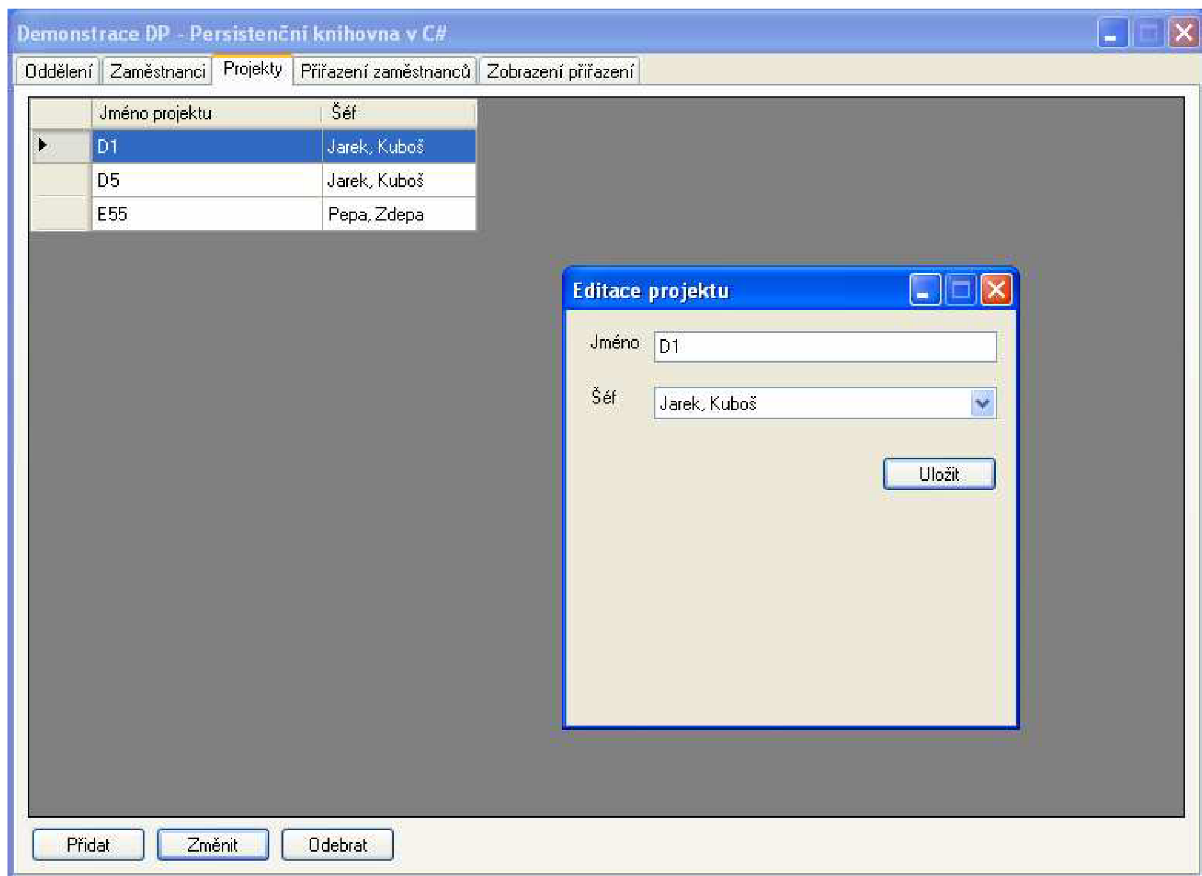


Obr. 7-5 Ukázka editace zaměstnance

```
Filter<Department> fd = new Filter<Department>();
fd.Condition = (fd.Fields.Name.Def==empDepCombo.Text);

Filter<Employee> fe = new Filter<Employee>();
fe.RefCondition = fe.Fields.Department.comply(fd);

foreach (Employee e in fe.loadCollection())
{
   //list
}
```

7.6 Záložka "Projekty"

V záložce projekty je demonstrována práce s třídou která obsahuje referenci i kolekci a to dokonce na shodnou referovanou třídu. Jedná se o třídu "Project", která referuje jednak svého vedoucího, ale také obsahuje kolekci zapojených pracovníků. V této záložce se pouze nastavuje vedoucí projektu, zapojení pracovníci se definují na další záložce.



Obr. 7-6 Ukázka editace projektu

Díky některým problémům popsaným výše, je potřeba pro přímý přístup k referovanému objektu používat vlastnost "obj_ref" reference.

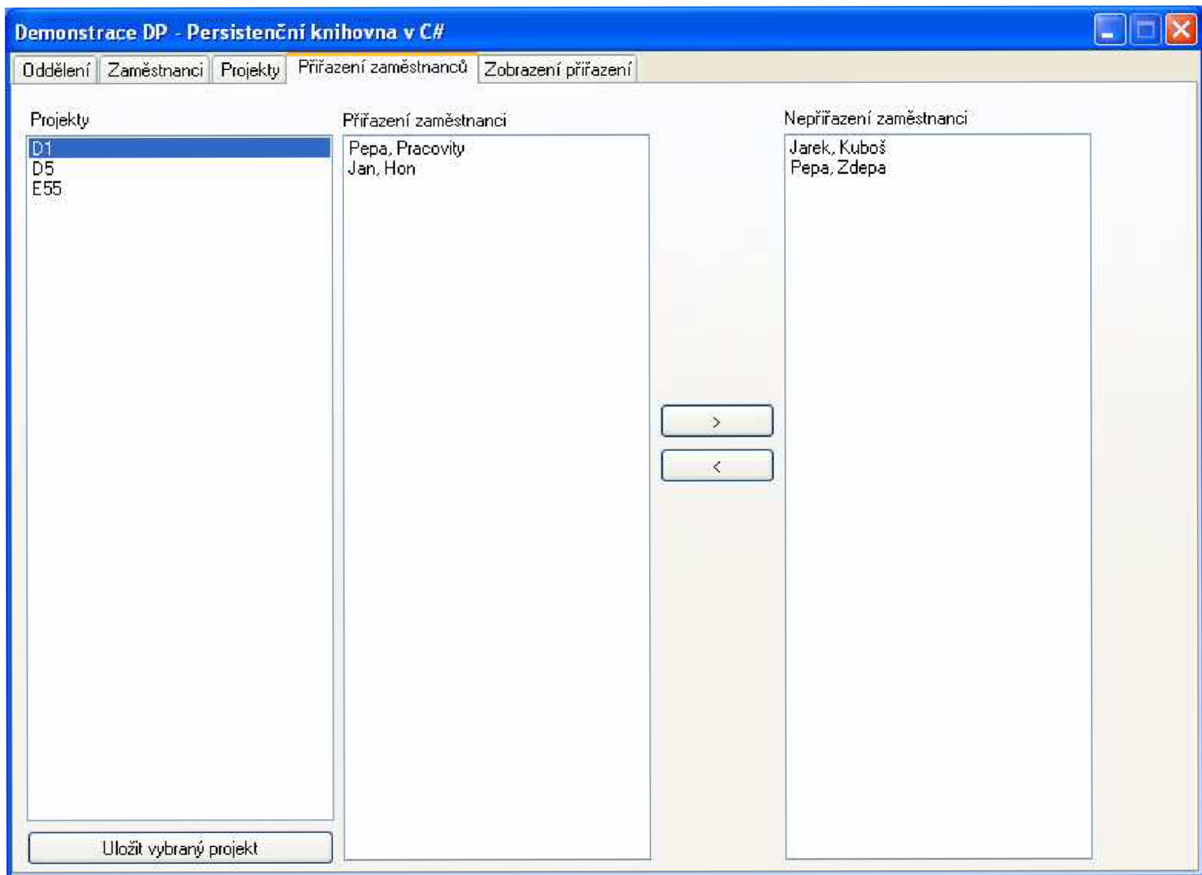
```
Filter<Department> f = new Filter<Department>();


foreach (Department d in f.loadCollection())
{
   Department s = _emp.Department;
   if (s.Id==d.Id)
   //nebo alternativně:
   //if (_emp.Department.ref_obj.Id==d.Id)
   {
      comboBox1.SelectedItem = d;
   }
}
```

Následující ukázka demonstruje uložení objektu obsahujícího referenci. Za zmínku stojí použití implicitního přetypování při nastavování hodnoty reference.

```
if (_emp==null)
{
   _emp = new Employee();
}

Department d = (Department)comboBox1.SelectedItem;

_emp.Department = d;

_emp.save();
```

7.7 Záložka "Přiřazení zaměstnanců"

Tato záložka demonstruje práci s kolekcemi. Vypisuje zapojené i nezapojené zaměstnance a umožňuje jejich přesun mezi těmito skupinami. Nastavení dokáže posléze uložit.



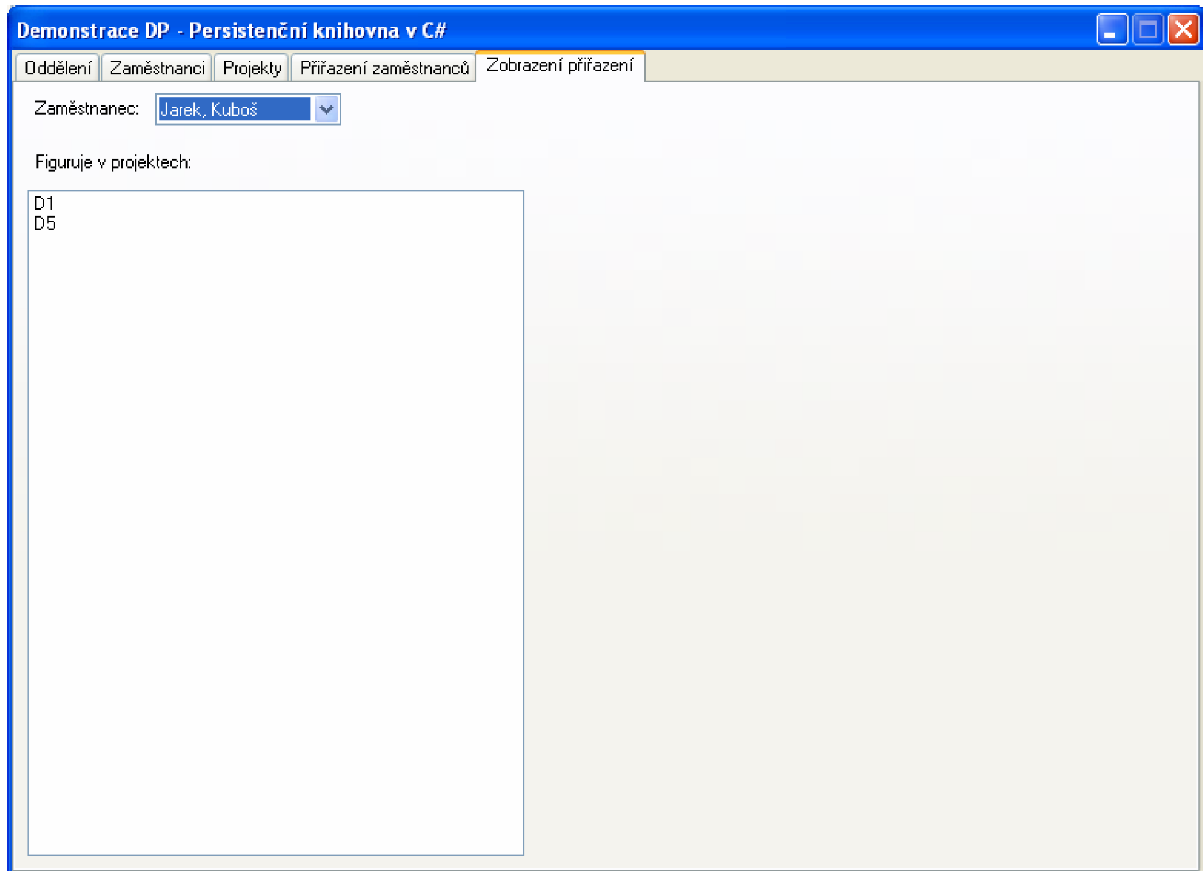
Obr. 7-7 Ukázka přiřazování zaměstnanců

Funkcionalita této záložky předvádí práci s kolekcemi. Jedná se o vyčítání objektů z kolekce, ale také její plnění a ukládání. Ukázka ukládání:

```
Project p = (Project)projectsList.SelectedItem;

p.Involved.clear();

foreach(object o in selectedEmpList.Items)
{
   Employee em = (Employee)o;
   p.Involved.add(em);
}

p.save();
```

7.8 Záložka "Zobrazení přiřazení"

V této záložce je použito dotazů na existenci objektu s popsanými atributy v kolekci. Tedy například: „Dej mi všechny projekty, ve kterých je zapojen zaměstnanec jména XYZ.“



Obr. 7-8 Ukázka zjišťování projektů přiřazených zaměstnanci

V této záložce je vidět použití dotazu na přítomnost prvku popsaného atributy v kolekci. Zároveň je zde použito řetězení podmínek pomocí logických operátorů.

```
//vybraný zaměstnanec
Employee em = cbInvEmploy.SelectedItem as Employee;

if ( em!=null )
{
    //filtr na zaměstnance podle jména a přijmení
    Filter<Employee> fEmp = new Filter<Employee>();
    fEmp.Condition = (fEmp.Fields.Name.Def == em.Name.Value) && (fEmp.Fields.Surname.Def ==
em.Surname.Value);
```

```
    //filtr na projekty, které obsahují vybraného zaměstnance nebo je vybraný zaměstnanec vede
    Filter<Project> fProj = new Filter<Project>();
    fProj.RefCondition = fProj.Fields.Involved.contain(fEmp) || fProj.Fields.Boss.comply(fEmp);

    //výpis výsledků
    foreach (Project p in fProj.loadCollection())
    {
        projFiltList.Items.Add(p);
    }
}
```

8 Závěr

Mým úkolem bylo navrhnou a vytvořit knihovnu pro tvorbu databázové vrstvy aplikací psaných v jazyce C#. Návrhu jsem věnoval velké množství času přestože už mám s perzistencí objektů poměrně bohaté zkušenosti. Hlavní práce totiž spočívala v identifikaci požadavků na dobrou perzistenční vrstvu, ale také v návrhu práce s ní. V návrhu práce s perzistentními objekty jsem nakonec našel těžiště své práce. Požadavky na perzistenční vrstvu jsou identifikovány poměrně dobře ostatními knihovnami.

Samotná implementace byla poměrně přímočarou záležitostí. Vzhledem k tomu, že jsem si před zadáním práce vytvořil prototyp demonstrující možnosti reflexe jazyka C#, nedočkal jsem se žádných nepříjemných překvapení. Kód jsem rozdělil do 3 modulů podle významu. Pokud to bylo vhodné, rozděloval jsem dlouhé třídy do více souborů díky možnosti v C# verze 2 – částečně definované třídy. Snažil jsem se psát maximálně univerzální a komentovaný kód.

Hlavním cílem práce bylo eliminovat všechny výskyty literálů v kódu jako identifikátorů (třídy, atributu, kolekce, reference) při použití knihovny. K atributům objektu se přistupuje pomocí "proxy" šablonového objektu, ke kterému se přistupuje pomocí operátoru ".", takže existenci operátoru ověří už překladač. Při psaní podmínek pro objektové dotazy nejsou literály pro identifikaci použity rovněž. Ve schopnosti psát objektové dotazy bez nutnosti používat literály pro identifikaci typů a atributů vidím hlavní přínos své práce. Tento způsob zápisu, jsem nenašel v žádné knihovně pro překládaný jazyk.

Hlavní nedokonalosti knihovny jdou na vrub nemožnosti přetypovat operátor ".". Toto je jediná část knihovny, jejíž použití není transparentní s běžným C# objektem. Proto při jeho použití musí uživatel zpozornět a udělat trochu zbytečné práce. Tento nedostatek bohužel nedokážu odstranit. Abych to dokázal, musel by být jazyk C# mírně upraven. V ostatních ohledech je knihovna snadno použitelná a bezpečná. Mnoho kontrol provádí už překladač.

Dokončením mé diplomové práce se knihovna dostala do použitelného stavu. Funguje základní perzistence, ale i pokročilejší objektové dotazy. Ačkoliv velká část jednotvárné manuální práce na vývoji perzistenční vrstvy cílové aplikace je odstraněna, pořád se najdou činnosti, které by bylo možno automatizovat. Napadají mne čtyři možné způsoby rozšíření knihovny:

- Pokročilejší objektové dotazy
    - Podpora více typů operátorů a funkcí ("LIKE", "STRPOS", …)
    - Optimalizace výkonu dotazů
    - Více možností dotazů nad kolekcí
        - Zatím jen test, zda obsahuje objekt s danými atributy
- Podpora dalších typů úložišť
    - Další SQL databáze by neměly být problém

- Zajímavější by byla podpora XML databází
- GUI komponenty
  - Tabulka pro výpis objektů daného typu (třeba i s doplňující podmínkou)
    - „Ukaž všechny zaměstnance z oddělení CEOs nebo WORKERs“
  - Automatický formulář
    - Stačilo by předat objekt daného typu a formulář by sám vytvořil editační pole
- Verzování metadat
  - Současná verze knihovny umožňuje verzování dat
  - Verzováním metadat by šlo splnit úkoly jako:
    - „Atribut X třídy Y existuje až od verze 5 třídy X“
    - „Třída R byla ve verzi 4 zrušena“
  - S verzováním metadat souvisí také migrace dat mezi verzemi
    - Přidání, modifikace a zrušení atributu
    - Přidání a zrušení třídy

Literatura

[1] Robinson, S., aj.: C# Programujeme profesionálně. Computer Press, 2003, ISBN 8025100855

[2] Open Source Persistence Frameworks in C#. CSharp-Source.Net, 2006. Dokument dostupný na URL http://csharp-source.net/open-source/persistence (duben 2007).

[3] Microsoft Development Library. Microsoft Corporation, 2007. Dokumenty dostupné na URL http://msdn.microsoft.com (duben 2007).

[4] Manuál knihovny NHibernate. Dokumenty dostupné na URL http://www.hibernate.org/22.html (duben 2007).

[5] MySQL Reference Manual. MySQL AB, 2007. Dokumenty dostupné na URL http://dev.mysql.com/doc/ (duben 2007).

[6] Manuály k databázi SQLite. Dokumenty dostupné na URL http://www.sqlite.org/docs.html (duben 2007).

[7] Kuboš, Jaroslav: Generátor tabulek a webových formulářů [Bakalářská práce]. Brno, FIT VUT, 2005

Seznam příloh

Příloha 1. CD obsahující zdrojové kódy knihovny, ukázkového příkladu a tento dokument