VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

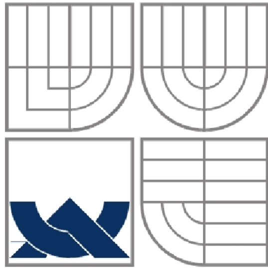
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH DATABÁZOVĚ NEUTRÁLNÍ OBJEKTOVĚ-RELAČNÍ VRSTVY

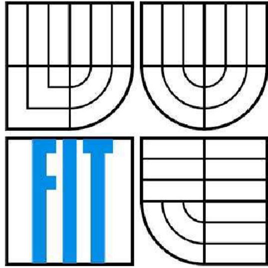
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PAVEL JEŽA
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

NÁVRH DATABÁZOVĚ NEUTRÁLNÍ OBJEKTOVĚ-RELAČNÍ VRSTVY

DESIGN OF A DATABASE NEUTRAL OR MAPPER IN C++

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. PAVEL JEŽA
AUTHOR

VEDOUCÍ PRÁCE Ing. ONDŘEJ RYŠAVÝ, Ph.D.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Ježa Pavel, Bc.**
Obor: Informační systémy
Téma: **Návrh databázově neutrální objektově-relační vrstvy**
Kategorie: Databáze

Pokyny:

1. Seznamte se s softwarovou technologií objektově relačního mapování.
2. Zaměřte se na rozdíly v syntaxi ve vybraných databázích (Firebird, MSSQL, Oracle), které budou v tomto projektu podporovány. Identifikujte potřebné technologie a prostředky.
3. Navrhněte a realizujte knihovnu pro objektově-relační mapovaní v prostředí Microsoft Visual C++.
4. Demonstrujte použití navržené knihovny na ukázkovém projektu a zhodnoťte dosažené výsledky a přínos práce pro vývoj intranetových aplikací.

Literatura:

- Gavin King and Christian Bauer: Hibernate in Action: Practical Object/Relational Mapping. Manning, 2004.
- Michael Vorburger, Heiko Bobzin, Keiron McCammon, Sameer Tyagi: Core Java Data Objects. Prentice Hall PTR, 2003.
- Scott Ambler: Agile Database Techniques: Effective Strategies for the Agile Software Developer. John Wiley & Sons, 2003.
- Michael Stonebraker, Dorothy Moore, and Paul Brown: Object Relational Dbms. Morgan Kaufmann Publishers; 2nd edition, 1999.
- Robert Vermeulen: Upgrading Relational Databases Using Objects. Cambridge University Press, 1997.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1-2.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Ryšavý Ondřej, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 28. února 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Pavel Ježa**
Id studenta: 22289
Bytem: Jízdárenská 574, 664 62 Hrušovany u Brna
Narozen: 03. 06. 1982, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   diplomová práce

Název VŠKP: Návrh databázově neutrální objektově-relační vrstvy
Vedoucí/školitel VŠKP: Ryšavý Ondřej, Ing., Ph.D.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....

....	....
Nabyvatel	Autor

Abstrakt

Diplomová práce se zabývá návrhem a implementací databázově neutrální objektově-relační vrstvy v prostředí jazyka C++ nad zděděnou databázi. Cílem je vytvoření vrstvy která odstíní (zapouzdří) přístup k databázi od aplikační vrstvy programu. Navrhovaná vrstva bude vycházet z technologie objektově relačního mapování, kterého je velké množství pro objektově programovací jazyky jako je C#, Java čí Visual Basic.

První část práce se zaměřuje na objasnění technologie objektově relačního mapování, konkrétně na různé možnosti a úrovně implementace. Další část se týká jednotlivých databází které budou v této práci uvažovány pro prezentaci databázové neutrálnosti. Zbylé části pojednávají o návrhu a implementaci této vrstvy včetně zhodnocení dosažených výsledků.

Klíčová slova

Objektově-relační mapování, ORM, relační databáze, mapování objektů, databáze, business objekty, DAO, Data Access Objects, Persistence Framework, C++.

Abstract

This diploma work deals with design and implementation of the database neutral object-relational (OR) layer in C++ language over inherited database. The goal is to create the layer to encase the access to database from the application layer. Suggested layer will stem from the object-relation mapping technology, which is currently available for many object-programming language, such as C#, Java or Visual Basic.

The work consists of three main parts. The forepart is focused on clearing object-relation mapping technology. It briefly overviews differences in capabilities and levels of implementation of various approaches. The next part describes significant properties of databases considered as back-ends for data storage in the project. The aim of this part is to present enough information to support database neutral design of the OR layer. The rest of the document deals with design and implementation of OR layer for the considered environment followed by the summarization of results and overall evaluational.

Keywords

Object-relational mapping, ORM, object-relational mapper, relational database, mapping objects, database, business objects, DAO, Data Access Objects, Persistence Framework, C++.

Citace

Pavel Ježa: Návrh databázově neutrální objektově-relační vrstvy, Brno, FIT VUT v Brně, 2007

Návrh databázově neutrální objektově-relační vrstvy

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Ondřeje Ryšavého, Ph.D. z ústavu informačních systému na fakultě informačních technologií VUT.
Další informace mi poskytli Ing. Erik Pomykal ze společnosti Grisoft a kolegové z oddělení „Enterprise řešení" téže společnosti.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jméno Příjmení
Datum

Poděkování

Zde bych rád poděkoval Ing. Ondřeji Ryšavému, Ph.D. za konzultace při tvorbě této diplomové práce. Dále chci poděkovat Ing. Eriku Pomykalovi ze společnosti Grisoft za poskytnutí témata diplomové práce a stejně tak i kolegům z oddělení „Enterprise řešení" téže společnosti za konzultace při analýze a návrhu objektově relační vrstvy.

© Pavel Ježa, 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 3
2 Přehled vlastností ORM 4
2.1 Metody zapouzdření databáze 4
2.1.1 Brute Force 4
2.1.2 Data Access Objects 5
2.1.3 Persistence Framework 6
2.1.4 Services 8
2.2 Mapování objektů do relačních databází 10
2.2.1 Mapování dědičnosti do tabulek 11
2.2.2 Typy vztahů 12
2.2.3 Implementace vztahů mezi objekty 13
2.2.4 Implementace vztahů v relační databázi 14
2.2.5 Mapování vztahů mezi objekty 14
2.2.6 Implementace výsledných objektů 16
2.2.7 Úrovně implementace ORM 17
2.2.8 Požadavky na perzistentní vrstvu (ORM) 18
2.3 Přehled objektově relačních mapovačů (ORM) 21
2.4 Zhodnocení vlastností uvedených ORM 22
3 Řízení přístupu a vícenásobný přístup 23
3.1 Řízení přístupu 23
3.1.1 Autentizace 23
3.1.2 Autorizace 24
3.1.3 Granularita přístupu 24
3.1.4 Omezení přístupu na úrovni databáze 24
3.1.5 Omezení přístupu na úrovni objektů 25
3.2 Vícenásobný přístup 26
3.2.1 Kolize 26
3.2.2 Typy zamykání 27
3.2.3 Řešení kolizí 28
4 Analýza rozdílů mezi uvažovanými databázemi 30
4.1 Stored (uložené) procedury 30
4.1.1 Jazyk pro psaní procedur 31
4.1.2 Základní konstrukce uložené procedury 31

4.1.3 Shrnutí základních rozdílů ... 33
4.2 Práva a uživatelé ... 33
4.2.1 Firebird ... 33
4.2.2 Oracle ... 33
4.2.3 SQL Server ... 34
4.3 Komunikace s databází ... 34
5 Návrh ... 35
5.1 Požadavky ... 35
5.2 Návrh nového řešení ... 38
5.2.1 XML popis tabulek ... 41
5.2.2 XML popis tabulek ... 44
5.2.3 Bázová perzistentní třída a kolekce ... 48
5.2.4 Perzistentní třída ... 50
5.2.5 DB Manager ... 52
5.2.6 Systém ohlašování chyb ... 56
6 Implementace ... 58
6.1 Popis použitých technologií v programu ... 58
6.1.1 Datové typy ... 58
6.1.2 Datové struktury ... 59
6.2 Řešení problémů spojených s implementací ... 59
6.3 Možnosti dalšího rozšíření objektově relační vrstvy ... 60
7 Závěr ... 61
Literatura ... 63
Seznam příloh ... 64
Příloha 1. DTD pro vstupní XML soubor ... 65
Příloha 2. Příklad použití ORL ... 66
Příloha 3. CD/DVD ... 67

1 Úvod

V dnešní době jsou hodně rozšířené relační databáze, kde business vrstva aplikace pracuje s jednotlivými řádky tabulky v databázi. Ale v reálném světě, jsou všechno objekty. Každý objekt má různě atributy a metody jak s nimi manipulovat. V počítačovém světě již proto také existuje spousta programovacích jazyků, které podporují nebo jsou přímo založené na objektech. Pro tyto typy programovacích jazyků a nejen pro ně, jsou již vytvořené také objektové databáze, které umožňují ukládat do databáze celé objekty. Vzhledem k tomu že mnoho současných aplikací používá pro ukládání dat tolik rozšířené relační databáze a okamžitý přechod na objektovou databázi není z nejrůznějších důvodů možný, je nutné vytvářet nové aplikace nad tzv zděděnými databázemi. Pojmem zděděná databáze se rozumí databáze, kterou musí nová aplikace využívat neboť nelze převést obsah stávající databáze do nové z důvodu existence starších aplikací, které stále tuto databázi využívají nebo z důvodu, kdy administrátoři nechtějí přejít na objektové databáze. Řešením tohoto problému je použití vrstvy, která bude mezi business vrstvou aplikace a databází. Její funkce bude spočívat v simulaci objektové databáze pro vyšší vrstvy aplikace a zajišťování perzistence objektů vyšší vrstvy v relační databázi.

Pro uložení dat existuje v současné době mnoho databázových systému od různých výrobců. Je možné vybírat z volně dostupných databázových systému nebo z komerčních databázových systému, které se vyznačují možností uložení velkého množství dat a rychlého dotazovaní, oproti volně dostupným databázovým systémům, které jsou určeny pro uložení relativně menšího množství dat. Každý výrobce databázového systému stejně jako jiní výrobci se snaží o to aby ten jeho produkt byl co nejlepší a zároveň originální, aby někdo nemohl říct že vyrábí to stejné jako konkurence. Z toho důvodu má každý databázový systém své specifické funkce, příkazy a způsob přístupu.

Proto jsem se rozhodl vytvořit databázově neutrální objektově relační vrstvu, která bude převádět data z relační databáze na objekty a také bude zpět ukládat objekty do relační databáze. Vytvořená vrstva bude umožňovat jednotný přístup k různým databázovým systémům a také bude umožňovat jednotný přístup k datům z pohledu aplikačního programátora. Z pohledu databázového programátora, bude potřeba provádět drobné korekce dotazů v závislosti na databázovém systému s kterým se bude pracovat. Další vlastností této vrstvy bude možnost vícenásobného přístupu k datům z více programů a řízení přístupu podle typu uživatele který bude k datům přistupovat.

V jednotlivých kapitolách se postupně seznámíme s technologií objektově relačního mapování od jednoduchých až po velmi propracované Persistence Frameworks. Dále si něco povíme o způsobech mapování objektů do databáze a o vícenásobném přístupu k datům, analyzujeme si rozdíly mezi jednotlivými databázovými systémy a seznámíme se s návrhem řešení této práce a s její implementací. Na závěr zmíním možná rozšíření této práce a zakončím ji zhodnocením.

2 Přehled vlastností ORM

V této kapitole se seznámíme s jednotlivými typy objektově relačního mapování a uvedeme si několik existujících zástupců technologie mapování objektů do relačních databází.

2.1 Metody zapouzdření databáze

Object Relation Mapper (ORM) je jeden ze způsobů nadstavby nad samotnou databází. Ale dříve než se zaměříme na jeden konkrétní způsob, uvedeme si přehled možných způsobů, jak zapouzdřit databázi. Mezi základní čtyři způsoby můžeme považovat [1]:

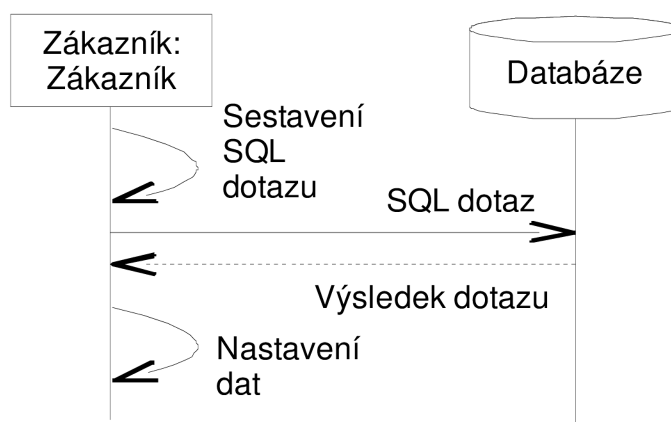
- Brute Force,
- Data Access Objects,
- Persistence Frameworks,
- Services.

2.1.1 Brute Force

Přístup „Hrubou silou" je základní a velmi jednoduchá metoda nadstavby nad databází, protože tento model neobsahuje vrstvu která by zapouzdřovala databázi, ale je vhodné ho zde uvést. Jedná se totiž také o platnou metodu pro přístup k databázi. Kromě toho je přístup hrubou silou po dlouhou dobu nejběžnější přístup, protože je jednoduchý a programátorům poskytuje plnou kontrolu nad daty u business objektů a databází. Tento přístup je vhodný pro jednoduché projekty, kde požadavky na přístup k databázi nejsou natolik náročnou záležitostí, jako například u jiných rozsáhlejších projektů. Základní přístup na pozadí přístupu hrubou silou je ten, kdy business objekty přistupují k úložišti dat přímo, typicky jsou podřízeny SQL nebo OQL (Object Query Language) kódu v databázi.

Následující UML sekvenční diagram názorně zobrazuje základní logiku čtení jednoho objektu z databáze. Po přijetí požadavku na načtení objektu, vytvoří business objekt SQL SELECT dotaz a předá jej do databáze. Databáze zpracuje dotaz a vrátí výslednou množinu dat. Objekt si pak provede úpravu vlastních dat, smaže stará data a nahradí je daty z výsledku dotazu a nastaví si atribut signalizující aktuálnost dat.

Přístup hrubou silou může být rozumně jednoduchý, ačkoliv časově náročný. Možná bude potřeba rádce v oblasti aplikačního programování, znalost základů dotazovacích jazyků jako je SQL a OQL, API funkcí pro přístup k databázi stejně jako JDBC a ODBC a také znalost řešení chyb. To všechno je potřeba znát, protože u přístupu hrubou silou píše všechen kód pro přístup k databázi aplikační programátor.



Obrázek 1: Čtení objektu metodou Brute Force.

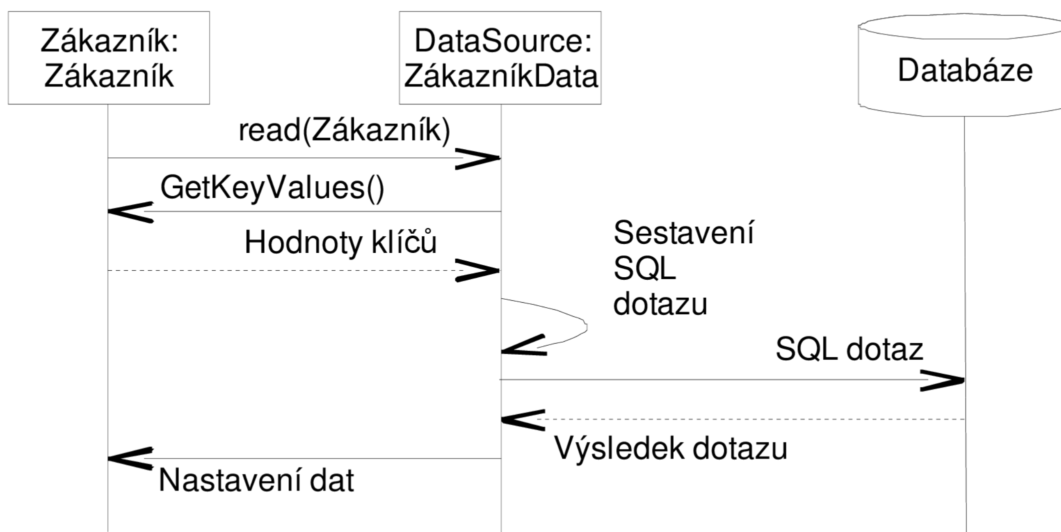
2.1.2 Data Access Objects

Data Access Objects (dále jen DAO) zapouzdřují logiku přístupu k databázi vyžadovanou business objekty. Pro tento model je typickým příkladem jeden datový objekt pro všechny business objekty. Pro konkrétní příklad, mějme třídu *Zákazník* která bude mít třídu *Zákazník_Data*. Třída *Zákazník_Data* implementuje SQL nebo OQL a další kód vyžadovaný pro přístup k databázi obdobně jako u přístupu hrubou silou. Hlavní výhodou DAO oproti přístupu hrubou silou je krátký čas, po který je business objekt přímo připojen k databázi, místo toho jsou k ní připojeny DAO objekty, tím je logika přístupu k databázi centralizována na jednom místě. Jedná se o zcela běžný postup pro jednoduché vyvíjení vlastních DAO, i když je možné využít již existující standardní přístupy jako je *Java Data Object* (JDO) a *ActiveX Data Object* (ADO).

> ***JDO neboli Java Data Objects je standardizované aplikační rozhraní pro persistenci objektů v Javě, vyvinuté přímo firmou Sun Microsystems, která je i původcem Javy jako takové. JDO umožňuje implementovat libovolný způsob ukládání objektů - může sloužit jako aplikační rozhraní objektové databáze, ale stejně tak může provádět mapování objektů na relační tabulky a ukládat data do relační databáze. Podstatné ovšem je, že aplikační rozhraní je zcela nezávislé na vlastní implementaci persistence, a proto aplikace využívající JDO mohou bez větších změn ukládat data do různých relačních i objektových databází.[2]***

> ***Microsoft ActiveX Data Objects (ADO) je množina COM(Component Object Model) objektů pro přístup k datovým zdrojům. Zprostředkovává vrstvu mezi programovacími jazyky a OLE DB. Umožňuje psát vývojářům programu s přístupem k databázi bez toho, aby znali jak je databáze implementována. Stačí jen vědět ke které databázi se chceme připojit. Není potřeba znát SQL pro přístup k databázi, když se používá ADO.[3]***

Opět si na následujícím obrázku ukážeme logiku pro načtení objektu *Zákazník* z databáze. Jak je vidět, tak objekt *Zákazník_Data* skrývá detaily přístupu k databázi před business objektem. V prvním kroku předá business objekt sám sebe jako parametr do DAO. V dalším kroku DAO získá primární klíč z databáze, pokud není znám. DAO potřebuje získat dostačující informaci z business objektu pro identifikaci objektu v databázi. Třetí krok pak spočívá v sestavení dotazu, který se použije v databázi a data z výsledku dotazu se použijí pro aktualizaci business objektu *Zákazník*.



Obrázek 2: Čtení objektu metodou Access Data Objects.

2.1.3 Persistence Framework

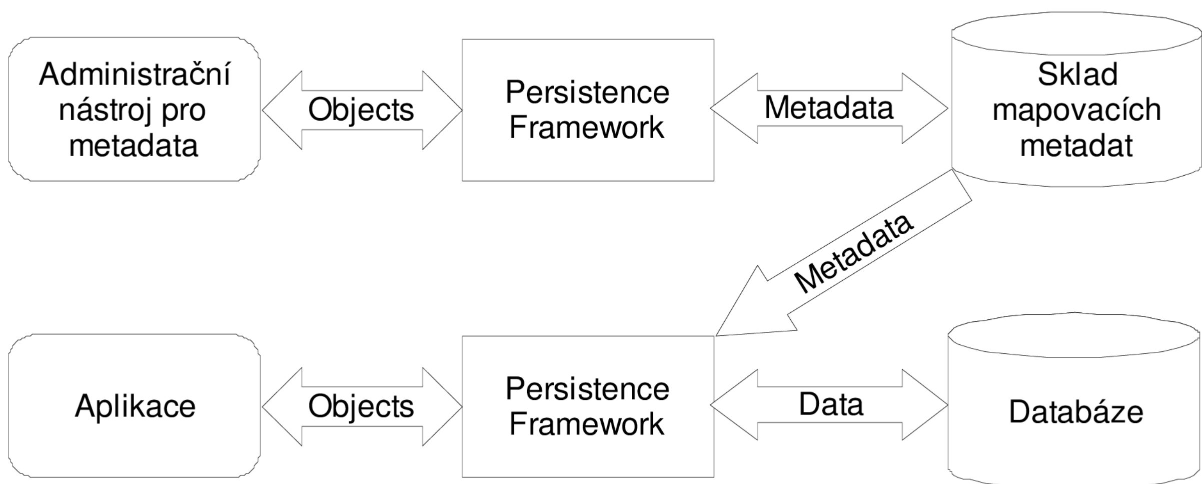
Persistence framework bývá často chápán jako perzistentní vrstva, plně zapouzdřující přístup k databázi z pohledu business objektů. Místo implementace logiky potřebné pro přístup k databázi se spíše definují metadata reprezentující mapování mezi business objekty a tabulkami databáze. Příkladem těchto metadat může být popis, který popisuje mapování třídy *Zákazník* do tabulky *T_Zákazník*. Metadata nereprezentují jen vztahy mezi business objekty a tabulkami v databázi, ale také vztahy mezi business objekty pokud existují. Metadata jsou důležitá pro zajištění perzistence business objektů a jsou generována vrstvou *Persistence Framework*. Tento kód může být generován jak dynamicky za běhu, nebo generován staticky v podobě *data access objects*(DAO) při kompilaci aplikace. První přístup poskytuje větší flexibilitu zatímco druhý přístup poskytuje větší výkon.

Persistence Framework může mít mnoho vlastností. Mezi základní patří podpora pro vytváření, čtení, aktualizaci a mazaní objektů (CRUD funkcionalita) stejně jako řízení transakcí a souběžnosti. Mezi pokročilé vlastnosti pak patří silné řešení chybových stavů, udržování spojení s databází, podpora XML, schopnost generování schémat, mapování a podpora standardních technologií jako je EJB.

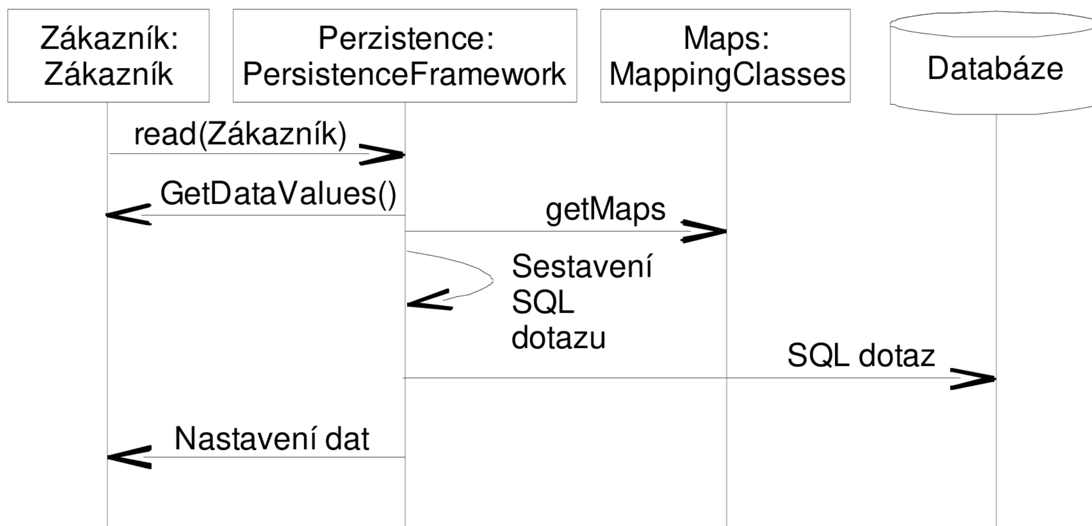
Technologie Enterprise JavaBeans (EJB) je komponentní architektura na straně serveru pro platformu Java, Enterprise Edici (Java EE). Technologie EJB umožňuje rychlý a jednoduchý vývoj a distribuovaných, transakčních, bezpečných a přenosných aplikací založených na technologii Java.[4]

Každý *Persistence Framework* by měl obsahovat administrátorský nástroj, který umožňuje spravování metadat pro mapování business objektu do databáze. Tento administrátorský nástroj většinou sám v sobě obsahuje *Persistence Framework* a při instalaci aplikace si zapíše metadata vyžadovaná pro popis mapování mezi úložištěm metadat a administrátorským nástrojem. Skladiště mapovacích metadat je typicky v relační databázi nebo v XML souboru, dokonce může být i v textovém souboru.

Důležitou otázkou při návrhu *Persistence Framework* je, zdali bude perzistence implicitní nebo explicitní. S implicitním přístupem *framework* automaticky zajišťuje perzistencí business objektů bez našeho vědomí, to znamená, že se nemusíme starat o ukládání a čtení business objektu. Dokonalým příkladem je koncept perzistentních kontejnerů u *Enperprise JavaBean* (EJB). S explicitním přístupem, který je dalece nejběžnější, potřebují objekty signalizovat, kdy by se měli uložit. Příklad explicitního a zároveň dynamického přístupu je znázorněn na ilustraci 4. *Zákazník* požaduje po *Persistence Framework* načtení sebe sama. V dalším kroku pak *Persistence Framework* spolupracuje s kolekcí namapovaných objektů pro účely vygenerování SQL dotazu, který je proveden na databázi.



Obrázek 3: Architektura Persistence Framework.



Obrázek 4: Čtení objektu metodou Persistence Frameworks.

Důležitá myšlenka, která stojí za povšimnutí je fakt, že business objekt potřebuje komunikovat pouze s *frameworkem*, mnoho z nich tedy obsahuje typicky třídy nazvané *Database*, *PersistenceFramework* nebo *PersistenceBroker*, které implementují veřejné rozhraní.

2.1.4 Services

Services (služby) poskytují možnost jedné počítačové entitě využít operace které poskytuje jiná entita. V současné době jsou webové služby jsou jednou z nejpopulárnějších architektur, neboť internet „vládne světu“ a webové služby jsou stále žádány a zdokonalovány. Webové služby jsou pouze malou části služeb, které lze použít. Služby jsou typicky používány pro zapouzdření přístupu k zděděným datům a funkcím. Zde si uvedeme několik architektur, které využívají předností architektury webových služeb, pro vytváření nových aplikací, které usnadňují systémovou integraci.

2.1.4.1 CORBA (Common Object Request Broker Architecture)

CORBA tvoří ucelené prostředí pro tvorbu objektově orientovaných distribuovaných aplikací. Jako norma byla CORBA vytvořena v roce 1991 skupinou OMG (Object Management Group) aby umožnila distribuci objektů v heterogenním prostředí ne-Microsoftí platformy. Ve stejném roce tato skupina také definovala jazyk IDL (Interface Definition Language) a aplikační programové rozhraní, které umožňuje klientu či serveru interakci se specifickou implementací ORB (Object Request Broker).

2.1.4.2 CICS Transakce (Customer Information Control systém)

Byl to transakční procesor, který díky firmě IBM získal oblibu v sedmdesátých letech dvacátého století. V dnešní době je to stále populární technologie pro kritické operace v business aplikacích.

2.1.4.3 DCOM (Distributed Component Object Model)

DCOM si získal oblibu na začátku devadesátých let dvacátého století jako preferovaný přístup pro implementaci distribuovaných komponent v prostředí Microsoft. DCOM je protokol který umožňuje programovým komponentám komunikovat přímo přes síť spolehlivým, bezpečným a efektivním způsobem. DCOM je stále důležitou částí platformy Microsoft.

2.1.4.4 EDI (Electronic Data Interchange)

Jedná se o standardizovanou výměnu dokumentů mezi organizacemi automatizovaným způsobem a přímo z jedné počítačové aplikace do druhé. EDI byl rozvíjen v polovině osmdesátých let dvacátého století mnoha výrobci. V dnešní době lze nalézt základy této technologie v mnoha aplikacích řešících kritické operace. Tato technologie je pomalu nahrazována technologií webových služeb.

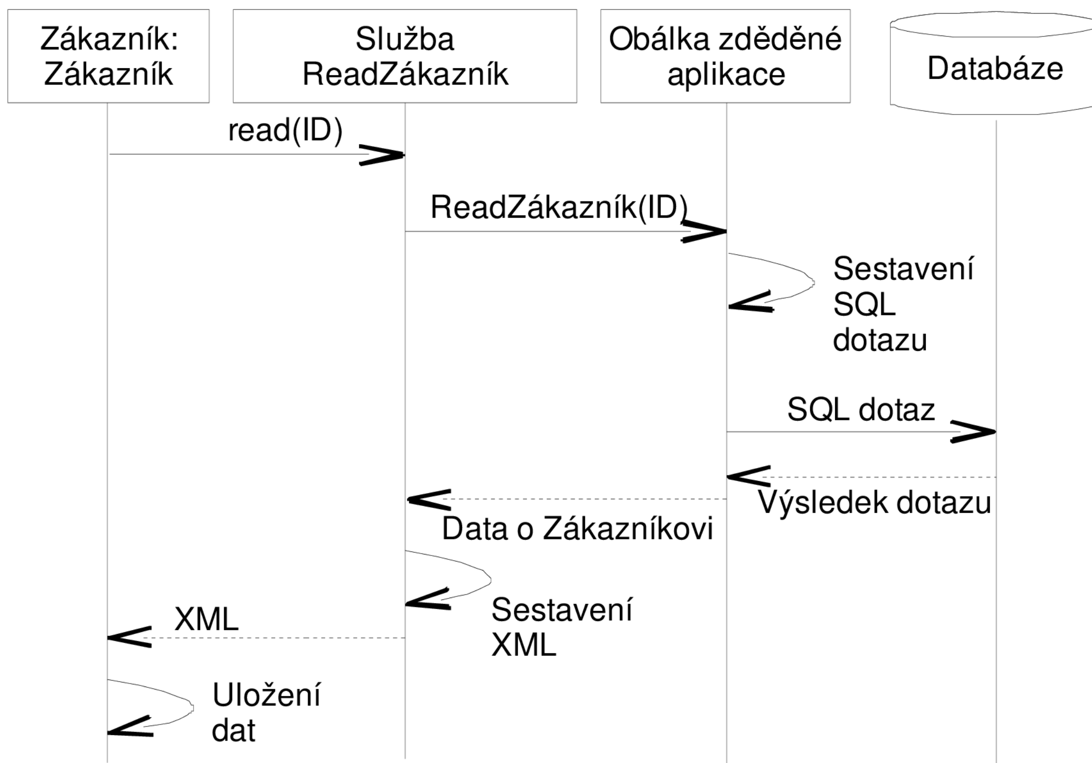
2.1.4.5 Stored (uložené) procedury

Stored procedury jsou podprogramy které lze volat z aplikace, která má přístup k relačnímu databázovému systému. Procedury se nazývají stored (uložené) proto, že jsou fyzicky uloženy v databázi společně s daty. Jejich rozšíření bylo zaznamenáno v polovině osmdesátých let dvacátého století a stále jsou platnou implementační technologií dneška.

2.1.4.6 Webové služby

Webové služby jsou business funkce psané podle striktní specifikace, které pracují přes Internet a používají XML. Příkladem obecné platformy pro webové služby je technologie „.NET“ od společnosti Microsoft a technologie „Sun ONE“ od stejnojmenné společnosti Sun.

Jako u předchozích přístupu pro zapouzdření databáze i zde si uvedeme UML sekvenční diagram který znázorňuje čtení objektu z databáze. V prvním kroku objekt *Zákazník* zavolá službu *ReadZákazník* a předá mu ID zákazníka, kterého chce načíst. Služba použije toto ID při požadavku na načtení zákazníka od „obálky“ která zapouzdřuje zděděnou aplikaci. Zděděná aplikace pak provede dotaz nad databází a výsledná data předá zpět přes „obálku“ zpět službě *ReadZákazník*. V závěru pak služba předá data business objektu, který si data uložení.



Obrázek 5: Čtení objektu přes službu.

2.2 Mapování objektů do relačních databází

Když se bavíme o mapovaní objektů do relační databáze, tak nejlepším místem kde začít jsou atributy tříd (objektů). Atributy se mapují na sloupce tabulky v relační databázi. Ale ne všechny atributy tříd je třeba ukládat, některé atributy lze z uložených atributů dopočítat. Protože některé atributy objektu mohou obsahovat další objekty (např.: Objekt *Osoba* bude pravděpodobně obsahovat objekt *Adresa* jako svůj atribut), jedná se zde o asociaci mezi dvěma třídami která bude muset být namapována stejně jako bude muset být namapována třída *Adresa*. Důležitou věcí je zde rekurzívnost definice, kterou bude potřeba zohlednit.

Dalším důležitou věcí která stojí za povšimnutí jsou stínové informace. Jedná se o určitá data která jsou potřeba pro zajištění perzistence objektů a tvoří doplňková data k samotným datům jednotlivých objektů. Typický jsou to primární klíče, obzvláště na místech, kde primární klíče tvoří náhradní klíče které nemají obchodní význam. Dalším příkladem stínové informace je časové razítko, nebo zvyšující se čítač při řízení vícenásobného přístupu. Mezi důležitou informaci kterou by měl obsahovat každý objekt, je atribut booleovského typu určující, zda je objekt již v databázi uložen, či nikoliv. Pokud by tento atribut třída/objekt neobsahoval(a), tak by nastal problém při ukládání objektu do relační databáze, protože by nebylo jasné zda se má pro uložení použít SQL update konstrukce na existující objekt, nebo SQL insert konstrukce na objekt, který ještě v databázi není uložen. Běžnou

praxí je implementovat pro každou třídu atribut *isPersistent*, který má hodnotu *true* když je objekt načten z databáze a hodnotu *false* pokud je objekt nově vytvořen.

2.2.1 Mapování dědičnosti do tabulek

Relační databáze nativně nepodporují dědičnost, proto si zde uvedeme 3 hlavní řešení pro mapovaní dědičnosti v relační databázi a jednu doplňující techniku jdoucí nad rámec mapování dědičnosti.

2.2.1.1 Mapování celé struktury tříd do tabulky

Mapování spočívá v ukládají všech atributů všech tříd do jedné tabulky. Metoda dobrého pojmenování tabulek spočívá v pojmenování tabulek podle kořene hierarchie tříd, což je velice jednoduchý způsob. Do tabulky jsou ještě přidány vedle business atributů 2 sloupce POID a *Typ*. Prvním přidaným sloupcem je primární klíč pro tabulku, kde zkratka označuje *Persistent Object Identifier*, který je také nazýván jako OID (Object identifier). Druhý přidaný sloupec slouží k určení konkrétního typu objektu v hierarchii (jedná se o listy stromové struktury dědičnosti). Výhodu tohoto přístupu je jednoduché přidání třídy, které spočívá pouze v přidaní nových sloupců pro nová data. Podporuje polymorfizmus jednoduchou změnou typu řádku, přístup k datům je rychlý, neboť data jsou v jedné tabulce. Nevýhodou je rychlé zvětšování tabulky pokud se jedná o složitou hierarchii tříd.

2.2.1.2 Mapování každé konkrétní třídy do vlastní tabulky

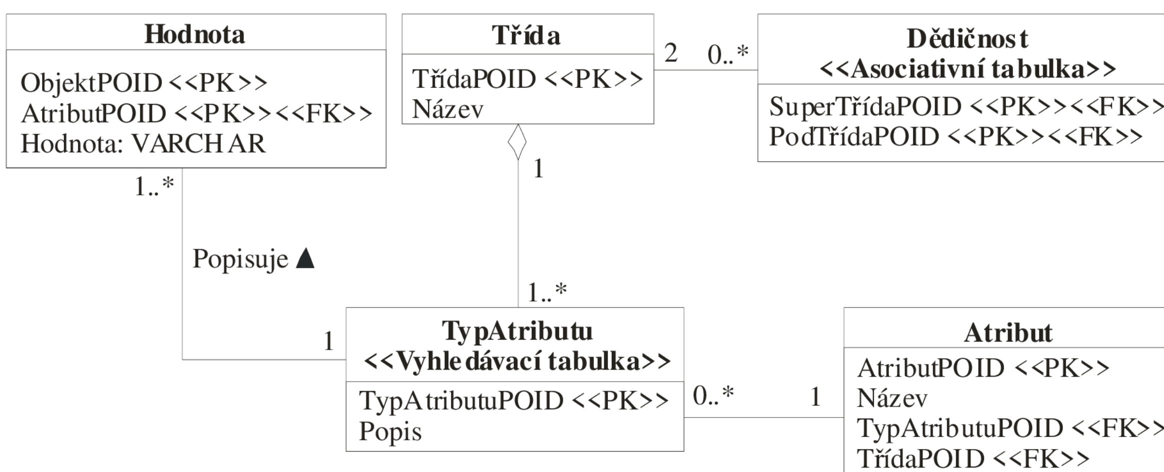
V tomto přístupu je pro každou konkrétní třídu vytvořena tabulka. Každá tabulka obsahuje jak vlastní atributy třídy, tak i všechny ty atributy, které získala dědičností. Do každé tabulky je pak přidám další sloupec POID identifikující objekt. Sloupec *Typ* již v tomto přístupu není potřeba. Dále je potřeba přidat další tabulku která obsahuje potřebné informace o vazbě konkrétních tříd. Výhodou je dobrý výkon, pokud se přistupuje pouze k datům z jednoho objektu. Nevýhodou je složitější úprava tabulek v databázi pokud se jejich bázový objekt rozšíří o další atributy.

2.2.1.3 Mapování každé třídy do vlastní tabulky

Cílem této strategie je vytvoření tabulky pro každou třídu, kde každý sloupec reprezentuje jeden business atribut třídy a nezbytné informace pro identifikaci (stejně jako ostatní sloupce pro řízení vícenásobného přístupu a verzování). Zde vzniká problém při načítání třídy která je odvozena z jiné třídy. V takovém případě je potřeba provést spojení minimálně dvou tabulek, nebo provést čtení dat pro každou tabulku zvlášť a tyto hodnoty pak spojit a uložit do jednoho objektu. Výhodou této metody je jednoduché vracení požadovaných objektů, protože se jedná o mapování *jedna-ku-jedné*. Mezi další výhody patří podpora polymorfizmu, který podporuje velmi dobře, velmi jednoduchá je i změna rodičovských tříd. Nevýhodou této metody je mnoho tabulek v databázi, potencionálně zdlouhavé načítaní a zapisování dat, protože se tyto operace provádějí nad několika tabulkami.

2.2.1.4 Mapování tříd do obecné struktury

Čtvrtým způsobem jak mapovat strukturu dědičnosti do relační databáze je volba obecného způsobu, někdy nazývaná metadaty řízený přístup mapovaní tříd. Tento přístup není specifický pro strukturu dědičnosti a podporuje všechny formy mapování. Než by jsme zde dlouze popisovali způsob implementace tohoto způsobu ukládání objektů ukážeme si obecné datové schéma na následujícím obrázku, z kterého je vše jasně vidět, takže nepotřebuje popis. Mezi výhody tohoto způsobu mapování patří velmi dobrý přístup k datům v databázi pokud je přístup k ní zapouzdřen do silné perzistentní vrstvy, umožňuje mapovaní i vztahů mezi objekty. Díky vyspělejší technice mapování muže být ze začátku těžké tento přístup implementovat. Mezi další výhody můžeme zahrnout podporu malého množství objektů, z důvodu potřeby komunikovat s více tabulkami pro načtení jednoho objektu a při velkém množství objektů by neúnosně narůstala režie spojená z čtením tabulek.



Obrázek 6: Obecné datové schéma pro uložení objektů.

Žádná z těchto metod není stoprocentně ideální pro všechny situace, ale je možné kombinovat první tři přístupy mapovaní dědičnosti do tabulek relační databáze, tedy konkrétně mapovaní hierarchie tříd na jednu tabulku, jedné konkrétní třídy na tabulku a jedné třídy na tabulku.

2.2.2 Typy vztahů

Doplňkem k mapování dědičnosti je technika mapování vztahů. Jsou zde tři typy vztahů mezi objekty, které je potřeba mapovat: asociace, agregace a kompozice. Vztahy můžeme rozdělit do dvou kategorií, na vztahy založené na násobnosti a vztahy založené na orientaci vztahu.

2.2.2.1 Vztah one-to-one

Toto je vztah, kde je u každého objektu násobnost maximálně jedna. Jako příklad uveďme Zaměstnance a pozici ve firmě, Zaměstnanec je zaměstnán právě na jedné pozici a na dané pozici ve firmě je právě jeden zaměstnanec.

2.2.2.2 Vztah one-to-many

Také známí jako vztah many-to-one, tento vztah se vyskytuje tam, kde na jedné straně je maximální násobnost vztahu rovna jedné a na druhé straně je maximální násobnost větší než jedna. Jako příklad uveďme opět příklad z praxe a práce, vztah zaměstnance a oddělení. Zaměstnanec pracuje na právě jednom oddělení, ale na jednom oddělení pracuje více jak jeden zaměstnanec.

2.2.2.3 Vztah many-to-many

V tomto vztahu je maximální násobnost na obou stranách větší než jedna. Jako příklad uveďme zaměstnance a úkoly, kde jeden zaměstnanec plní více jak jeden úkol a zároveň jeden úkol plní více jak jeden zaměstnanec.

Kategorie vztahů založena na orientaci obsahuje dva typy

- jednosměrný vztah a
- obousměrný vztah.

2.2.2.4 Jednosměrný vztah

Jednosměrný vztah se vyskytuje tam, kde jeden objekt ví o druhém objektu s kterým je ve vztahu, ale druhý objekt neví o prvním. Tento vztah je jednodušší na implementaci než vztah obousměrný. Jako příklad můžeme uvést zaměstnance a pozici, zaměstnanec ví na jaké pozici pracuje, ale pozice neví kdo na ní je.

2.2.2.5 Obousměrný vztah

Obousměrný vztah existuje, když oba objekty ve vztahu o sobě vzájemně vědí. Jako příklad můžeme uvést zaměstnance a oddělení. Zaměstnanec zná oddělení kde pracuje a objekt oddělení zná pracovníky kteří na něm pracují.

2.2.3 Implementace vztahů mezi objekty

Vztahy v objektovém schématu jsou implementovány jako kombinace referencí na objekty a operací. Když je násobnost rovna jedné (0..1 nebo 1), tak je vztah implementován referencí na objekt s *getter* a *setter* operacemi. Například objekt *Zaměstnanec* bude obsahovat atribut oddělení a metody *getOddělení* (která vrátí hodnotu oddělení) a *setOddělení* (která nastaví hodnotu oddělení).

U násobnosti větší jak jedna (N, 0..*, 1..*) je vztah implementován přes atribut kolekce, jako je například pole a operace s polem. Pro příklad uveďme třídu *Oddělení* která bude mít atribut *zaměstnanci* a metody *getZaměstnanci*, *setZaměstnanci*, *addZaměstnanec*, *removeZaměstnanec*.

Pokud se jedná o jednostrannou vazbu, tak je kód implementován pouze u jednoho objektu a to u toho, který ví o vazbě na druhý objekt, u obousměrné vazby je třeba vazbu implementovat u obou objektů.

2.2.4 Implementace vztahů v relační databázi

Vztahy jsou v relačních databázích spravovány skrze používání cizích klíčů. Pro vztahy 1:1 je nutné cizí klíč implementovat jednou z tabulek. K implementaci vztahu 1:N je implementován cizí klíč z jedné tabulky do mnoha tabulek. Dalším řešením je přepracování databázového schématu a vytvoření vztahu one-to-many přes asociační tabulku, což má efektivní využití u mapování vztahu many-to-many.

2.2.5 Mapování vztahů mezi objekty

2.2.5.1 Mapování vztahu one-to-one

Uvažujme vztah one-to-one mezi objekty *Zaměstnanec* a *Pozice*. Nyní předpokládejme, že kdykoliv se načte objekt *Pozice* nebo *Zaměstnanec* do paměti, tak aplikace automaticky zohlední vztah mezi objekty a automaticky načte odpovídající objekt. Tento přístup okamžitého načítání objektů ve vztahu se nazývá *eager* (netrpělivé) čtení. Jinou možností by bylo manuální zohlednění vztahu přímo v kódu, metodou *lazy* (líné) čtení, kde druhý objekt je načten až ve chvíli kdy jej aplikace vyžaduje.

A nyní budeme uvažovat jak by jsme objekt uložili do databáze. Jelikož vazba mezi objekty je automaticky vytvořena, je udržována referenční integrita a je vytvořena transakce, tak nezbývá než v dalším kroku přidat do transakce příkaz update pro oba objekty. Každý příkaz update obsahuje klíčové hodnoty pro mapování, jelikož se jedná o úpravu objektů, vztah mezi objekty bude efektivně uložen. Zde je jedna nepříjemnost ve způsobu držení vazby která má být mapována v databázi. Ačkoliv je směr této vazby z objektu Zaměstnanec do objektu Pozice v objektovém schématu, tak je v databázi implementována z *Pozice* do *Zaměstnance*, což je nepříjemné. V datovém schématu je možné pro vztahy 1:1 implementovat cizí klíče pro každou tabulku a to bez rozdílů, ale z výsledného pohledu, kdy všechny tabulky budou mít stejné cizí klíče, je jako hod mincí, neboť nebudeme mít informaci, které tabulky se tento cizí klíč týká. Zde mohou být potencionální požadavky pro držení vztahu použitelné ve vztahu one-to-many. Malá změna v návrhu může způsobit, že by se cizí klíče daly využít i v návrhu tohoto vztahu.

2.2.5.2 Mapování vztahu one-to-many

Nyní uvažujme pracovní vztah mezi Zaměstnancem a Oddělením, kde Zaměstnanec pracuje na jednom Oddělení a na jednom Oddělení pracuje několik Zaměstnanců. Zajímavou myšlenkou tohoto vztahu by mohlo být automatické navázání ze *Zaměstnance* do *Oddělení*, někdy nazývané kaskádní čtení, ale ne v opačném směru. Kaskádní ukládání a kaskádní mazání je také možné.

Když je *Zaměstnanec* načten do paměti, je automaticky navázán vztah a načten objekt *Oddělení* v kterém Zaměstnanec pracuje. Jelikož nechceme mít několik stejných kopií objektu *Oddělení* (například pokud máme 10 Zaměstnanců pracujících na stejném oddělení, tak budeme chtít aby se odkazovaly na stejný objekt *Oddělení* v paměti). V důsledku toho budeme potřebovat implementovat mechanizmus pro řešení tohoto chování. Jednou z možností je implementovat *cache* která bude zajišťovat že v paměti bude pouze jedna kopie objektu, nebo jednoduše bude mít třída *Oddělení* implementovanou vlastní kolekci instancí v paměti (efektivní *minicache*). Ukládání vztahu zde pracuje obdobným způsobem jako u vztahu one-to-one. Když objekt bude ukládán, tak jejích primární a cizí klíče budou automaticky ukládat jejích vztah.

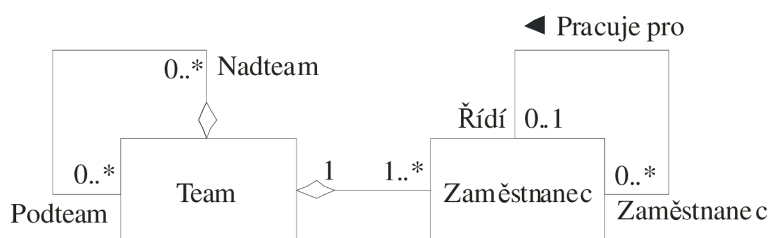
2.2.5.3 Mapování vztahu many-to-many

Pro implementaci vztahu many-to-many budeme potřebovat asociativní tabulku, jejíž jediným účelem bude správa vztahu mezi dvěma a více tabulkami v relační databázi. Jako příklad předpokládejme objekt *Zaměstnanec* který je načten v paměti a nyní potřebujeme načíst seznam všech úkolů, ke kterým byl zaměstnanec přiřazen. Načtení úkolů provedeme v několika krocích, prvním krokem bude spojení asociační tabulky s tabulkou úkolů podle POID *Zaměstnance*. Výsledkem tohoto dotazu bude množina záznamů, které uložíme do kolekce objektů typu *Úkoly*, zároveň proběhne ověření jestli objekt náhodou již není v paměti. Pokud objekt již v paměti je, provede se jeho obnova s nastavením nových hodnot. V posledním kroku se využije operace *Zaměstnanec.přidatÚkol()* pro všechny objekty *Úkoly* pro naplnění kolekce. Vztah many-to-many je zajímavý v tom, že využívá další, asociační tabulku. Dvě business třídy budou mapovány do tří tabulek pro zachování vztahu mezi nimi, takže ve výsledku je to práce navíc, kterou bude potřeba udělat.

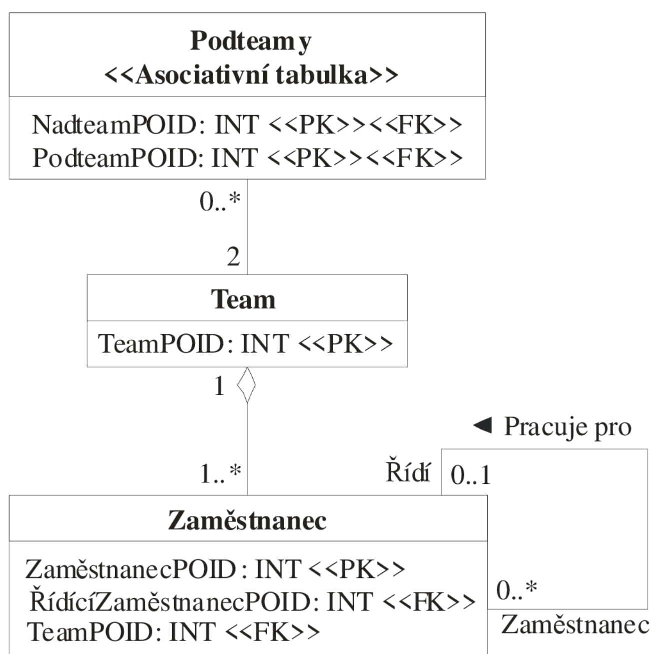
2.2.5.4 Mapování rekurzivních vztahů

Rekurzivní vztah, nazývaný také jako reflexivní vztah je vztah, kde jedna entita (třída, tabulka) má vztah sama se sebou, tedy vztah začíná i končí ve stejné entitě. Jako příklad uveďme Zaměstnance a vztah řídí, kdy jeden zaměstnanec může řídit jednoho či více dalších zaměstnanců. Následující obrázek znázorňuje model tříd který obsahuje dva rekurzivní vztahy a výsledný datový model, který toto mapuje. Kvůli jednoduchosti datového modelu budeme uvažovat jen tyto třídy a jejich vztahy a jako jednotlivé entity nebudou obsahovat datové atributy, pouze klíče. Rekurzivní agregace many-to-many je mapována do asociační tabulky *Podteamy*, stejnou cestou budeme mapovat i normální vztah many-to-many, rozdílem bude jen to, že tabulka bude obsahovat dva cizí klíče z jedné tabulky. Podobně, asociace one-to-many je mapována stejným způsobem jako normální vztah one-to-many.

[Model tříd]



[Fyzický datový model]



Obrázek 7: Mapování rekurzivních vztahů.

2.2.6 Implementace výsledných objektů

Rozdíly mezi objektovou technologií a relační technologií se promítají do vytvářeného objektového schématu a datového schématu. Pro implementaci tohoto mapování je potřeba přidat programový kód do business objektů, tento kód ovlivní výsledné aplikace nad těmito objekty. Tento programový kód by měl obsahovat:

Řízení stínových informací. Stínové informace nejsou důležité pro uživatele, ale pro programové mapování objektů, obsahují například informace o verzích a stavech jednotlivých objektů a nejlépe objekty samy by si měly tyto hodnoty spravovat.

Práci se zděděnými daty. Při práci se zděděnými daty je důležitá kvalita dat, návrhu a problémy spojené s architekturou. V důsledku toho často potřebujeme mapovat objekty do zděděných databází a objekty musí zajišťovat integritu a čištění dat.

Zapouzdření přístupu k databázi. Vytvoření vrstvy která zapouzdřuje přístup k databázi a zajišťuje mapování. Objekty by měly ovlivňovat výběr strategie a všude tam kde se vyskytuje SQL kód by měl být tento kód nahrazen rozhraním které poskytuje *Persistence Framework*.

Implementaci konkurenčního přístupu. Protože mnoho aplikací je víceuživatelských a databázi využívá mnoho jiných aplikací, tak je třeba ošetřit stavy, kdy se dva rozdílné procesy snaží modifikovat současně jednu datovou hodnotu. Tuto technologii si probereme později.

Nalezení objektu v relační databázi. Protože většinou se bude pracovat s kolekcí objektů místo práce s jedním objektem, je potřeba implementovat postup, jak vyhledávat jednotlivé objekty.

Implementaci referenční integrity. Je potřeba implementovat strategii pro zajištění referenční integrity mezi objekty a v databázi. Jelikož referenční integrita je obchodní problém, je třeba ji implementovat do business objektů.

Implementaci řízení bezpečnosti. Různí lidé mají různý přístup k informacím. Výsledkem toho je implementace bezpečnostní logiky do objektů a do databáze.

Implementaci skladiště (cache) pro objekty. Toto je důležité pro zajištění výkonnosti aplikace tím, že objekty v paměti budou unikátní.

2.2.7 Úrovně implementace ORM

Ne všechny objektově relační mapovače jsou schopny podporovat kompletní model a schopnosti klient-server architektury. Některé mapovače musí dělat ústupky a volit neúplný přístup, kdy nevyužívají na všechno objekty. O míře dokonalostí (úplnosti) mapovače rozhodují požadavky na funkcionalitu této vrstvy, implementační možnosti, tedy jakou funkcionalitu je možné naimplementovat a také rozhodujícím faktorem je cena a čas vývoje. Následujících pět bodů shrnuje úrovně objektově relačního mapování podle složitosti od jednoduchých, využívající relační model až po implementaci plně objektového mapování [5].

2.2.7.1 Čistě relační

Používá relační model jak na straně serveru, tak na straně klienta. Klientské uživatelské rozhraní je organizováno ve stylu řádků a tabulek a používá standardní relační operace nad tabulkou a jejími řádky.

2.2.7.2 Jednoduchý objektový mapovač

Používá určitou množinu objektů na straně klienta a snaží se izolovat většinu kódu aplikace který má něco společného s SQL. Převání řádky tabulek z databáze na straně serveru do jednoduchých objektů na straně klienta. Zapouzdřuje složité volání SQL kódu do určitých tříd a metod, tudíž velká část aplikace se nemusí zabývat psaním SQL kódu pro jednotlivé třídy.

2.2.7.3 Středně složitý objektový mapovač

Na straně klienta se primárně využívají objekty a celé chování aplikace se píše nad těmito objekty. Pružná konverze řádků tabulky z databáze do vhodných typu objektů a proměnných. Zahrnuje vyhledávání dotazů pro konkrétní objekty a zajišťuje konverzi potřebnou pro volání SQL příkazu. Také poskytuje mechanizmus pro jednoduchou identifikaci dotazů pro opakující se dotazy, kdy si ukládá odpovědi k těmto dotazům, aby pak rychleji mohla mapovací vrstva odpovídat na dotazy objektů i bez kontaktování databáze.

2.2.7.4 Úplný objektový mapovač

Naprosté využívání objektů v kódu aplikace. Je podobný předchozí úrovni, ale poskytuje mnoho obecných tříd pro uložení dat, většinou pro dědičnost a skládání. Řídí redundanci a rozdíly ve stavu serveru, umožňuje provádění lokálních dotazů kdykoliv je to možné, kombinuje mnoho jednoduchých dotazů na objekty do jednoho velkého dotazu pro databázi.

2.2.7.5 Objektový server

Server sám o sobě používá objekty pro organizaci informaci i chovaní, takže získávání a úprava objektů může být plně specifikována v podmínkách objektového modelu a muže tak být prováděna buď na straně serveru nebo na straně klienta. Zde se pak řeší kompromis mezi rychlostí zpracování operací a přesuny dat.

Výběr technické úrovně, která bude použita pro vytvoření ORM má dopad na výkonnost, cenu, škálovatelnost, chování aplikace, čas vývoje a údržbu. Žádná z těchto technik není nejlepší pro to, aby vyhověla všem těmto kritériím. Hodně sofistikované techniky nejsou vhodné pro všechny aplikace a mnoho aplikací zrovna tak dobře pracuje i s čistě relačním modelem. Zase jiné aplikace mohou narůstat na velikosti a složitosti, že je jednodušší pro správu použít objektový model. Bude tedy v tomto projektu důležité se rozhodnout, která technika bude pro tento projekt nejvhodnější.

2.2.8 Požadavky na perzistentní vrstvu (ORM)

Perzistentní vrstva zapouzdřuje chování potřebné pro zajištění perzistence objektů, jinými slovy jde o zajištění operací čtení, uložení a smazaní objektu z/do databáze nebo do jiného úložiště dat. Robustní perzistentní vrstva by měla, podle zkušeného vývojáře v této oblasti, podporovat [6]:

2.2.8.1 Podpora několika typu mechanizmů zajišťujících perzistenci

Systém perzistence je technologie která umožňuje uložení objektů po pozdější modifikaci, vyhledávání nebo smazání. Možný perzistentní mechanizmus zahrnuje soubory, relační databáze, objektově orientované databáze, hierarchické databáze a síťové databáze.

2.2.8.2 Plné zapouzdření mechanismu perzistence

V ideálním případě by mělo stačit pouhé zaslání zprávy ulož, načti nebo smaž aby se objekt uložil, načetl nebo smazal. Vše ostatní by už měla zařídit perzistentní vrstva. Mimoto by měla zařídit korektní ošetření chybových stavů. Programátor by neměl mít potřebu psát dodatečný kód mimo perzistentní vrstvu, který by zajišťoval perzistenci dat respektive objektů.

2.2.8.3 Práce s více objekty současně

Jelikož je běžné, že se načítá několik různých objektů najednou, například pro reporty nebo jako výsledek speciálního vyhledávání, musí silná perzistentní vrstva podporovat současné vrácení více než jednoho objektu. To stejné by mělo platit i pro mazání objektů.

2.2.8.4 Transakce

Tento požadavek souvisí s předchozím požadavkem, jedná se o transakce, kolekce akcí nad několika objekty současně. Transakce muže být tvořena kombinací čtení, ukládání nebo mazaní objektů. Transakce by mely být jednotné, s přístupem „všechno nebo nic“, kde všechny akce musí být uloženy (operace *commit*), nebo všechny zrušeny(operace *rollback*). Transakce může byt také tvořena z několika pod-transakcí, kde úspěch celé transakce je podmíněn úspěchem všech pod-transakcí. Transakce také mohou být s krátkou životností, probíhající několik tisícin vteřiny, nebo také s dlouhou životností, kde transakce trvá hodiny, dny, týdny nebo dokonce i měsíce než je dokončena.

2.2.8.5 Rozšiřitelnost

Měla by být možnost přidání nových tříd do objektových aplikací a také by měla být možnost jednoduché změny perzistentního mechanizmu(pro případ, kdy se rozhodneme po čase vylepšit stávající perzistentní mechanizmus nebo použít jiný od jiného výrobce). Jinými slovy perzistentní vrstva musí být natolik flexibilní, že se vzájemně svojí prací nebude ovlivňovat práce aplikačního programátora a administrátora starajícího se o perzistentní mechanizmus.

2.2.8.6 Identifikace objektů

Identifikace objektů nebo zkráceně také OID je atribut, typicky číslo, které slouží pro unikátní identifikaci objektů. OID je objektově orientovaný ekvivalent ke klíčí z relačních databázi, kde jeden sloupec obsahuje unikátní identifikaci každého řádku v tabulce.

2.2.8.7 Kurzory

Perzistentní vrstva umožňuje na jeden příkaz (dotaz) vrátit několik objektů. Zde ale vyvstává otázka, zda je efektivní vracet velké množství objektů najednou? Samozřejmě že ne. Proto už i v relačních databázích byl zaveden pojem kurzor, který se objevuje i u objektových databází. Jedná se o mechanizmus, kdy výsledkem dotazu je velké množství objektů, které zůstává v perzistentní vrstvě a

je možné je procházet po jednotlivých objektech pomocí kurzorů. Tento způsob je efektivnější neboť uživatel většinou nepotřebuje hned všechny objekty.

2.2.8.8 Proxy

Proxy je objekt, který reprezentuje jiný objekt, ale nepotřebuje tolik režie jako reprezentovaný objekt. Proxy obsahuje dostatek informací pro oba počítače a pro uživatele aby mohl objekt identifikovat. Proxy jsou běžně používané tam, kde výsledek dotazu je zobrazen jako seznam z kterého uživatel vybere jen jeden nebo dva objekty. Když uživatel vybere *proxy* objekt ze seznamu, je skutečný objekt (který je mnohem větší než *proxy*) automaticky načten z vrstvy zajišťující perzistenci.

2.2.8.9 Záznam

V dnešní době ohromná většina reportovacích nástrojů požaduje na vstupu kolekci databázových záznamů, nikoli kolekci objektů. Proto by perzistentní vrstva u objektově orientovaných aplikací měla podporovat možnost jednoduchého vrácení záznamů jako výsledek dotazu, aby se vyhnula režii při převádění databázových záznamu na objekty a zpět na záznamy.

2.2.8.10 Vícevrstvá architektura

Jako organizace přechází od centralizovaných sálových počítačů přes dvouvrstvé architektury klient/server a přes n-vrstvé architektury k distribuovaným objektům, tak by perzistentní vrstva měla být schopna podporovat tyto různě přístupy.

2.2.8.11 Různé verze databáze od různých výrobců

Perzistentní vrstva by měla podporovat možnost jednoduché změny vrstvy perzistence dat aniž by se to nějak dotklo z pohledu aplikace přistupující k datům. Proto by měla perzistentní vrstva podporovat široký rozsah verzí databázových systému, jakožto i databáze různých výrobců.

2.2.8.12 Vícenásobné připojení

Mnoho organizací používá více než jednu vrstvu perzistence dat, často od různých výrobců a požadují přístup pomocí jedné objektové aplikace. Důsledkem toho je, že perzistentní vrstva by měla být schopna podporovat vícenásobné, současné připojení pro každou příslušnou vrstvu perzistence dat. Dokonce i něco jednoduchého jako je kopírování objektů z jedné vrstvy perzistence dat do druhé, nebo třeba z centrální relační databáze do lokální relační databáze, kde je potřeba nejméně dvou současných připojení, jedno pro každou databázi.

2.2.8.13 Nativní a ne-nativní ovladače

Zde je několik rozdílných strategií pro přístup k relačním databázím a dobrá perzistentní vrstva by měla podporovat nejobecnější z nich. Strategie připojování zahrnuje použití ODBC (Open Database

Connectivity), JDBC (Java Database Connectivity) a nativních ovladačů dodávaných výrobci databázi nebo výrobci třetích stran.

2.2.8.14 SQL dotazy

Psaní SQL dotazů v objektově orientovaném kódu je do očí bijící porušení zapouzdření. Občas je to ale potřeba z důvodu výkonnosti porušit. Natvrdo psané SQL příkazy by měly být v aplikaci opravdu jen výjimečně a tyto výjimky by měly být dostatečně zdůvodněné než k nim dojde. V každém případě by perzistentní vrstva měla podporovat možnost přímého provedení SQL dotazu do relační databáze.

Perzistentní vrstva by měla dovolit aplikačním vývojářům soustředit se na to co umí nejlépe, tedy vytvářet aplikace, bez potřeby starat se o to, jak budou jejich objekty uloženy. Kromě toho by perzistentní vrstva měla také dovolit databázovým administrátorům dělat jen to co umí nejlépe, spravovat databáze bez nutnosti starat se o náhodné chyby v aplikaci. S dobře vytvořenou perzistentní vrstvou by mělo být možné přesouvat tabulky, přejmenovávat tabulky, přejmenovávat sloupce a reorganizovat tabulky bez ovlivnění aplikace, která k ní přistupuje.

2.3 Přehled objektově relačních mapovačů (ORM)

V předchozí kapitole jsme si uvedli základní způsoby jak zapouzdřit relační databázi. Objektově relační mapovač je vlastně také vrstva aplikace která zapouzdřuje relační databázi tak, aby se vyšší vrstva (business objekty) nemusela starat o způsob uložení business objektů a nabývala dojmu, že pracuje s objektovou databází místo relační. Konkrétně ORM je vlastně přístup k zapouzdření databáze metodou *Persistence Framework*, kterou jsme si zde již popsali. V současné době existuje mnoho objektově relačních mapovačů, a v následující tabulce si uvedeme jen některé v abecedním uspořádání.

Produkt	Platforma	Popis	Adresa
Castor	Java	Open Source persistence framework pro jazyk Java. Je to nejkratší cesta mezi Java objekty, XML a relačními tabulkami. Poskytuje vazbu mezi XML a Javou, perzistenci mezi Javou a SQL...	http://www.castor.org/
CocoBase Enterprise O/R	Java	Objektově relační mapper enterprise úrovně s CocoBase PURE POJO	http://www.thoughtinc.com/

Tabulka 1: Přehled několika objektově relačních mapovačů - první část.

Produkt	Platforma	Popis	Adresa
Deklarit	.NET	Modelem řízený nástroj který umožňuje generovat udržovatelné a konkurence schopné databázové schémata.	http://www.deklarit.com/
Hibernate	Java, .NET	Vysoce výkonná objektově relační perzistentní a dotazovací služba.	http://www.hibernate.org/
JC Persistent Framework	VB6,C#	Open Source objektově relační persistentní framework zahrnující objekty a rozhraní pro perzistenci.	http://jcframework.sourceforge.net/
Osage Persistence Plus XML	Java	Objektově relační mapper založený na JDBC, generující SQL pro práci s objekty.	http://osage.sourceforge.net/

Tabulka 2: Přehled několika objektově relačních mapovačů - druhá část.

2.4 Zhodnocení vlastností uvedených ORM

V předchozí podkapitole jsme si uvedly několik objektově relačních mapovačů, které jsou dostupné na trhu jak ve formě komerčních nástrojů tak jako open source. Tyto nástroje jsou neustále zdokonalovány a upravovány tak, aby vývojářům usnadnili práci tím, že za ně víceméně všechno zařídí a vývojáři se mohou starat o funkčnost jejich aplikace kterou nad databázi vytváří. Všechny uvedené objektově relační mapovače můžeme označit za úplně objektové mapovače, které také splňují všechny požadavky uvedené v předchozí podkapitole. Jak jsme si ale mohly všimnout, všechny tyto ORM jsou dostupně jen pro objektové programovací jazyky. Pro jazyk C++ lze těžko najít objektově relační mapovač. Za zmínku stojí jen DTL (Database Template Library) který vytváří recordsety podobné STL kontejnerům, nebo POST++(http://www.garret.ru/~knizhnik/post.html), IOPC (Implementation of object Persistency in C++) (http://iopc.sourceforge.net/) který generuje perzistentní objekty přímo z C++ popisu tříd, a jako poslední zde zmíním LiteSQL C++ Object Persistence Framework (http://sourceforge.net/projects/litesql/).

3 Řízení přístupu a vícenásobný přístup

3.1 Řízení přístupu

Důležitým aspektem každého systému je, zda řízení přístupu bude jednoduché nebo bezpečné. Řízení bezpečnostního přístupu znamená, zajistit autentizovanému uživateli přístup jen k těm datům, ke kterým má autorizovaný přístup a nikam jinam. Bohužel bezpečnost je zřídka kdy na prvním místě v lidském žebříčku, ale na druhou stranu se pro ně rychle staly zajímavými pojmy jako důvěrnost, citlivost a vlastnictví. Dobrou zprávou je existence velkého množství technik, které mohou být použity pro zajištění bezpečného přístupu k systémům. Vzhledem k tomu že člověk je nejslabším článkem v bezpečnosti systému je důležité aby lidský i technický faktor šel společnou cestou k dosažení bezpečnosti.

3.1.1 Autentizace

Autentizace je proces ověření identity. - cs.wikipedia.org

Autentizace je proces, při kterém se ověřuje, zda je uživatel nebo entita opravdu ten, za koho se vydává. V prvé řadě se jedná o ověření uživatele, který se pokouší pracovat se systémem, zdali je uživatel nebo systém oprávněn k této činnosti. Druhým cílem autentizace je shromáždění informací o činnostech, které mají být uživateli přístupné. Existuje několik způsobů, jak mohou být klienti identifikováni

3.1.1.1 ID uživatele a heslo

Toto je nejběžnější a typicky nejjednodušší řešení jak někoho identifikovat, protože je to kompletně softwarová záležitost. Tento typ autentizace se řídí tím co uživatel zná.

3.1.1.2 Fyzické bezpečnostní zařízení

Fyzické zařízení jako je kreditní karta, smart karta nebo počítačový čip, který se používá pro identifikaci osoby. Někdy je také vyžadováno heslo nebo PIN (Personal Identification Number) pro ověření, že se jedná o pravého uživatele. Zde se autentizace řídí tím co uživatel má.

3.1.1.3 Biometrická identifikace

Biometrie je věda zabývající se identifikací jedinců pomocí fyzických charakteristických a unikátních rysů. Tato technologie zahrnuje identifikaci podle hlasu, sítnice, dlaně ruky, otisku prstu, atd. U tohoto typu se autentizace řídí tím čím uživatel je.

3.1.2 Autorizace

Autorizace je postup, který omezuje přístup k datům, funkcím a dalším entitám. Přístup mají pouze oprávnění uživatelé. Proces autorizace vyžaduje autentizaci uživatele a vyhledání v seznamu oprávněných uživatelů, jejich rolí a práv.

3.1.3 Granularita přístupu

Granularita určuje úroveň kontroly přístupu (v databázi je to například tabulka, řádek tabulky, sloupec tabulky, atd), je třeba volit mezi režií kontroly a dostatečně jemným rozlišením.

Atribut(sloupec) – Například vedoucí personálního oddělení může měnit plat u jednotlivých zaměstnanců a personalista se může dívat na platy zaměstnanců, ale zaměstnanec by neměl mít k platu zaměstnanců přístup.

Objekt (řádek) – Někdo může vyzvednout peníze z bankovního účtu a někdo ne.

Třída (tabulka) – Databázový administrátor má plný přístup k modifikaci systémových tabulek v databázi a aplikační programátor by o těchto tabulkách neměl ani vědět.

Aplikace – Senior manažeři z určité organizace mají přístup do výkonného informačního systému, který poskytuje velmi důležité informace o jejich odděleních, které spravují. Tento systém by však neměl být dostupný zaměstnancům, kteří nejsou manažery.

Databáze – Například účetní by měla mít přístup pouze k databázi týkající se platů a neměla by mít přístup třeba do databáze výroby.

Hostitel – Uživatel může do systému přistupovat z počítače který má v práci, ale z počítače který má například doma už nikoliv. Toto je často nazýváno jako hostitelské oprávnění nebo geografické oprávnění.

3.1.4 Omezení přístupu na úrovni databáze

Pro začátek se seznámíme s pojmem role a bezpečnostní kontext. Role je pojmenovaná kolekce oprávnění, která mohou být přiřazena uživateli. Takže vlastně spravují autorizační práva zvlášť pro každého uživatele. Každá role má svůj specifický přístup. Role jsou obecným přístupem, který je implementován široké oblasti techniky, nejen v databázích, zjednodušují totiž administraci.

Bezpečnostní kontext je kolekce rolí, která může být uživateli přidělena. Bezpečnostní kontext je často definován jako část procesu autentizace. Je závislý na použité technologii, je spravován systémem a musí platit v celém systému.

Autorizace může být vynucena v databázi výběrem prostředků, které mohou být kombinovány. Mezi tyto technické prostředky patří:

3.1.4.1 Oprávnění

Oprávnění je výsada, nebo autorizační právo, pro uživatele nebo role týkající se elementu (jako je sloupec, tabulka nebo samotná databáze). Oprávnění definuje typ přístupu, který je oprávněn, jako je například úprava tabulky, nebo spouštění uložených procedur. V SQL je oprávnění přidělováno pomocí příkazu GRANT a odebíráno přes příkaz REVOKE. Když uživatel začne pracovat s databází je jeho oprávnění ověřeno, pokud uživatel není autorizován k požadované činnosti, tak je jeho práce ukončena a je vráceno chybové hlášení.

3.1.4.2 Pohledy

Jedná se o velmi pěknou oblast omezení přístupu, kdy uživatel má přístup k datům pouze přes pohledy. Jedná se o proces vyžadující dva kroky. V prvním kroku jsou definovány pohledy, které omezují pohled na tabulky, sloupce a řádky z tabulky podle rolí. Druhý krok spočívá v definování oprávnění pro pohledy.

3.1.4.3 Stored (uložené) procedury

Kód v uložených procedurách může být implementován tak, aby programově ověřoval oprávněnost přístupu.

3.1.4.4 Patentovaný přístup

Novou možností bývá použití patentovaného bezpečnostního nástroje dodávaného výrobcem databáze. Pro příklad uveďme *Oracle Label Security* (www.oracle.com), jedná se o modul, který umožňuje definování a vynucení oprávnění na úrovni řádků tabulky.

3.1.5 Omezení přístupu na úrovni objektů

Jelikož objekty zapouzdřují jak data, tak chování, je třeba implementovat objektově orientované autorizační postupy pro zajištění bezpečnosti obojího. Toto může být problémem, protože objektově orientované programovací jazyky jako je C++, C#, Java nebo Visual Basic nemají nativní podporu pro zajištění bezpečnosti. Takže je potřeba zvolit jednu, nebo i kombinaci z následujících technik.
Autorizace v objektech může být implementována následující kolekcí technik:

3.1.5.1 Hrubá síla

Aplikace která vyžaduje autorizaci, musí sama implementovat všechnu bezpečnostní logiku.

3.1.5.2 Nástroj pro obchodní pravidla

Autorizační logika je přenechána speciálním nástrojům jako je *Blaze*, *Versata Transaction Logic Engine*, *QuickRules*. Každá operace která vyžaduje autorizaci, jednoduše potřebuje uplatnit vhodná pravidla z obchodního nástroje a podle toho jednat.

3.1.5.3 Oprávnění

Toto je stejná strategie jako je oprávnění v databázi, jen s tím rozdílem, ze oprávnění aplikované na operace třídy jsou spíše aplikovány na prvky databáze. Tento přístup je používán v EJB (Enterprise JavaBean servers), kde EJB kontejnery automaticky porovnávají přístupová práva u operací s kterými uživatel přichází do styku. Když oprávnění není nastaveno, kontejner stále ověřuje implicitní oprávnění.

3.1.5.4 Bezpečnostní framework/komponenta

Autorizační funkcionalita je zapouzdřena do bezpečnostního frameworku. Jako příklad komerčního nástroje je *.NET framework*, který obsahuje bezpečnostní aspekt. Dalším příkladem nechť je JAAS (Java Authentication and Authorization Service). Je možné si také vytvořit vlastní bezpečnostní framework pro jiné prostředí. Bezpečnostní framework/komponenta může být umístěna u klienta, na aplikačním serveru nebo dokonce na databázovém serveru.

3.1.5.5 Bezpečnostní server

Jedná se o speciální externí server(y) implementující bezpečný přístup řízený pravidly, které jsou použity v požadavcích. Mezi komerční nástroje patří *Cisco Secure Access Control Server* a *RSA Cleartrust Authorization Server*.

3.2 Vícenásobný přístup

Uvažujme situaci, kdy několik uživatelů pracuje s daty jedné tabulky, konkrétně s jedním stejným záznamem a současně se dva uživatelé rozhodnou zapsat svoji novou verzi záznamu do databáze. Čí změna bude zapsána do databáze? Prvního uživatele? Druhého uživatele? Ani jednoho? Nebo snad kombinace obou? Na tyto otázky se zaměříme v následující podkapitole nazvané kolize.

3.2.1 Kolize

Kolize se říká stavu, kdy dvě činnosti které mohou a nemusí být v transakci požadují změnu položky ve stejném záznamu. Existují tři základní způsoby jak dvě aktivity mohou do sebe zasahovat.

3.2.1.1 Špinavé čtení (dirty read)

Činnosti 1 (A1) přečte prvek ze záznamu a potom upraví tento záznam ale neprovede uložení změn (například proto, že provádí ještě nějaké další operace). *Činnost 2* (A2) přečte prvek, nevědomky si vytvoří kopii neuložené verze. A1 provede *rollback* změn a obnoví všechny data do stavy, který byl před započetím činnosti 1. A2 má nyní verzi prvku, který nikdy nebyl uložen a tedy ani neodpovídá aktuálně existující verzi prvku.

3.2.1.2 Neopakovatelné čtení

A1 přečte prvek ze záznamu a vytvoří si kopii. A2 smaže prvek ze záznamu. A1 nyní má kopii prvku, který oficiálně neexistuje.

3.2.1.3 Čtení přízraků (phantom read)

A1 načte kolekci prvků podle určitých kritérií. A2 vytvoří nový prvek který shodou náhod vyhovuje kritériím, které použila A1 při načítaní prvků. A2 uloží tento nový prvek do databáze. A1 provede opětovné načtení kolekce podle stejných kritérií, ale dostane jiný výsledek.

Řešením těchto problémů jsou zámky. V následujícím textu si popíšeme rozdílné typy zamykání.

3.2.2 Typy zamykání

3.2.2.1 Pesimistické zamykání

Pomocí tohoto přístupu se zamykají položky v databázi pro celý čas, kdy je v aplikační paměti (často ve formě objektů). Zámek nemůže nikdo jiný otevřít a chrání tak položky v databázi aby s nimi nemohl nikdo pracovat. Existují dva druhy zámků, zámek pro čtení a pro zápis.

Zapisovací zámek – tento zámek zabraňuje změnění položky, tedy nedovolí čtení, změnu ani smazání zamčené položky.

Čtecí zámek – tento typ zámku povoluje čtení zamčeného prvku ostatními, ale nepovoluje změnu nebo smazání prvku. Rozsah tohoto zámku může být na celé databázi, na tabulce, na kolekci řádků nebo na řádek.

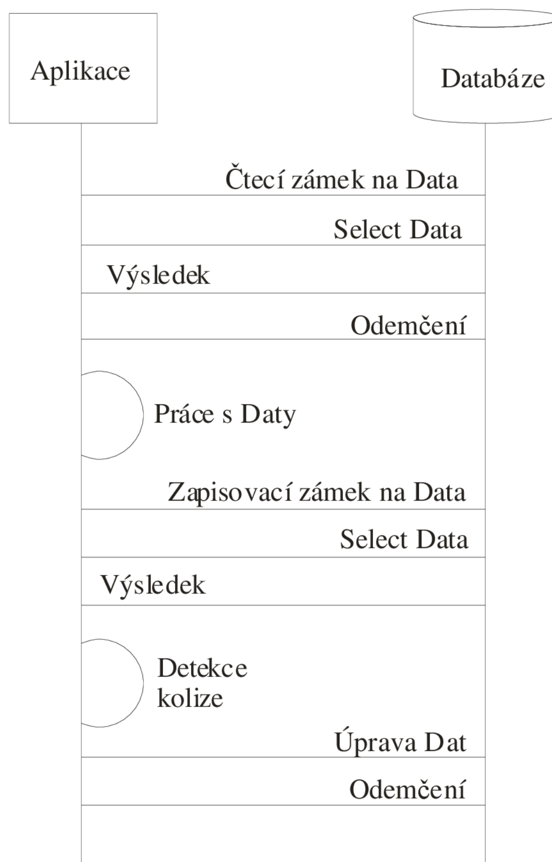
3.2.2.2 Optimistické zamykání

Tento způsob si ukážeme na příkladu (doplněného o následující ilustraci) úpravy objektu s použitím optimistického zamykání. Aplikace načte objekt do paměti, čtení provede za použití čtecího zámku, po načtení všech dat pro objekt provede uvolnění zámku. V tuto chvíli mohou být námi načtené řádky označeny za možné místo vzniku kolize. Aplikace pracuje s objektem a po čase se rozhodne uložit změny v objektu, aplikace použije zapisovací zámek na data a načte ze zdroje znovu ten stejný objekt a porovná oba objekty, jestli náhodou nedošlo ke kolizi. Zde nastanou dva případy:

- kolize nenastala a objekt může být uložen do databáze nebo,
- nastala kolize a je potřeba ji vyřešit.

3.2.2.3 Příliš optimistické zamykání

Tento postup odpovídá jednouživatelskému systému, kde systém záznamů kde je zaručen přístup pouze jednoho uživatele nebo procesu systému v jednom okamžiku. Tento systém je vzácný, ale může se objevit. Je důležité pochopit, že tento přístup je celý nevhodný pro víceuživatelské systémy.



Obrázek 8: Úprava objektu s použitím optimistického zamykání.

3.2.3 Řešení kolizí

3.2.3.1 Vzdát se řešení problému

Toto je nejjednodušší řešení. Spočívá v mechanismu výjimek, když nastane situace, kdy je detekována kolize, tak je vyvolána výjimka s informací o chybě, která je předána volající aplikaci a všechny změny jsou ztraceny.

3.2.3.2 Zobrazit problém a nechat rozhodnout uživatele

Při výskytu kolize je problém nahlášen uživateli aby jej vyřešil. Minimálně je uživateli zobrazena chyba s popisem jako „Jiný uživatel změnil data, chcete přesto zapsat nová data...“ a možností volby jak se rozhodnout, jestli pokračovat v úpravě nebo zrušit operaci.

3.2.3.3 Sloučení změn

V případě kolize zkusit najít které atributy konkrétně jsou v kolizi a pokusit se o sloučení těchto dvou změn dohromady. Pokud jsme si uložili kopii originálních dat, není problém určit které konkrétní atributy byly změněny. Pak můžeme uživatele informovat o rozdílech v datech a nechat uživatele rozhodnout o dalším kroku. Riskantní způsob je nechat systém aby dělal sám správná rozhodnutí.

3.2.3.4 Zapisování kolizí do logu a rozhodovat se později

Tato metoda je podobná metodě zobrazování problému uživateli, rozdíl spočívá ve volaní aplikací, kdy aplikace běží bez dozoru a není možné aby uživatel okamžitě řešil kolize. Tento přístup vyžaduje administrátorskou aplikaci, která umí pracovat s těmito „kolizními" logy. Aby nedocházelo k dvojímu přepisu kolize, tak jedním z řešení je použití čtecího zámku na kolizní data do doby než je problém vyřešen nebo nevyprší určitý čas určený pro opravu.

3.2.3.5 Ignorování kolizí a přepisování

Tento přístup je jednoduše způsoben režií optimistického zamykání, ve skutečnosti ještě mluvíme o přístupu příliš optimistického zamykání. Tato strategie není doporučovaná.

4 Analýza rozdílů mezi uvažovanými databázemi

V databázovém světě existuje mnoho databázových systému od různých výrobců a to jak komerční, tak open-source. Jako příklad si uveďme *Oracle*, *SQL Server*, *DB2*, *Sybase*, *Informix*, *InterBase/Firebird*, *SQLBase*, *MySQL*, *PostgreSQL*. V této práci budeme uvažovat pouze databáze *Firebird*, *Oracle*, *SQL Server* a podporu pro budoucí rozšíření o další typy databází.

Jak jsme si popsali v předchozích kapitolách, databáze slouží pro zajištění perzistence business objektů z aplikační vrstvy. V našem případě se jedná o relační databáze nad kterými bude cílem této práce vytvořit objektově-relační vrstvu. Pro zapouzdření SQL dotazů budeme používat *stored procedury*, které jsou také vhodným způsobem pro zajištění bezpečnosti přístupu k datům, kde nebude možné získání dat s databáze jen prostřednictvím těchto uložených procedur. V této kapitole si tedy povíme jak vypadají a jak se implementují *stored procedury* jednotlivě ve vybraných databázích. V další podkapitole se zaměříme na podporu práv a uživatelů v kontextu granularity omezení přístupu k datům v jednotlivých databázích a závěrem této kapitoly si řekneme něco o možnostech komunikace s databázemi.

4.1 Stored (uložené) procedury

Uložené procedury jsou databázové objekty (uložené v databázi), které neobsahují data, ale část kódu programu, který se vykonává nad daty v databázi. Uložená procedura je procedura, která je funkčně oddělena od svého okolí, má rozhraní s případnými parametry pro komunikaci s jinými moduly programu a může mít vlastní lokální proměnné. Pro psaní uložených procedur je používán specifický jazyk, který se liší podle databáze, respektive podle výrobce databáze a je rozšířením dotazovacího jazyka. Výhodou uložených procedur je jednotné rozhraní a skrytí datových operací. Například pokud budeme chtít uložit nového zaměstnance do databáze a naši databázi bude využívat několik typu programů které umějí vkládat nové zaměstnance, tak bez uložených procedur budeme muset v každém programu implementovat kód pro uložení zaměstnance do databáze. S použitím uložených procedur stačí napsat tento kód jednou a ze všech těchto programů jen zavolat metodu pro přidání zaměstnance a jako parametry předat proceduře informace o novém zaměstnanci. Výhodou skrytých datových operací je, že se nemusíme v klientském programu zabývat způsobem uložení dat do konkrétních tabulek, tedy jen zadáme název procedury s případnými parametry a necháme proceduru ať se postará o všechno ostatní.

4.1.1 Jazyk pro psaní procedur

4.1.1.1 Firebird

Firebird databáze používá pro psaní procedur jazyk pro psaní procedur a triggerů (zkratku tohoto jazyka jsem nenašel, v anglických dokumentech je tento jazyk označován jako „procedure and trigger language"). Tento jazyk zahrnuje SQL příkazy pro manipulaci z daty a několik užitečných rozšíření jako jsou příkazy pro řízení toku (Flow Control) programu (IF...THEN...ELSE, WHILE...DO, …), výjimky a ošetření chybových stavů.

4.1.1.2 Oracle

Databáze Oracle nabízí pro psaní uložených procedur jazyk PL/SQL. PL/SQL je procedurální rozšíření jazyka SQL firmy Oracle, jedná se o rozšířený 4GL jazyk. Stejně jako u databáze Firebird, tak i zde tento jazyk umožňuje manipulovat s daty pomocí SQL příkazů a řídit tok programu(procedury) příkazy pro větvení, nebo opakování části procedury. Na stránce *http://www.oracle.com/technology/tech/pl_sql/* je možné najít jaké operace je s tímto jazykem možné provádět.

4.1.1.3 SQL Server

Firma Microsoft ve svém databázovém systému SQL Server používá jazyk T-SQL (Transact-SQL), který poskytuje široké možnosti použití v databázi obdobně jako jazyky tohoto typu u ostatních databázi, které jsme si zde uvedli.

4.1.2 Základní konstrukce uložené procedury

4.1.2.1 Firebird

Uložená procedura se definuje příkazem CREATE PROCEDURE, uložené procedury nelze definovat pomocí vestavěného (embedded) SQL. Procedura se skládá ze dvou částí, hlavičky a těla. Hlavička obsahuje jméno procedury, které musí být unikátní v rámci procedur, pohledů a jmen tabulek v databázi. Volitelně může mít procedura vstupní parametry, které jsou předávány z volaného programu. Tělo pak může obsahovat lokální proměnné následované blokem příkazů jazyka pro procedury a triggery, tento blok je ohraničen klíčovými slovy BEGIN a END a jeden blok může obsahovat další bloky. Pro úplnost si zde ještě uvedeme příklad vytvoření *store procedury* GET_EMP_PROJ, která má vstupní parametr EMP_NO a vrací výsledek v parametru PROJ_ID.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
FOR SELECT PROJ_ID
FROM EMPLOYEE_PROJECT
WHERE EMP_NO = :EMP_NO
INTO :PROJ_ID
DO
SUSPEND;
END
```

Kód 1: Vytvoření uložené procedury v databázi Firebird.

4.1.2.2 Oracle

U tohoto typu databáze také pro vytvoření uložené procedury slouží příkaz CREATE PROCEDURE., který má skoro podobnou syntaxi jako Firebird, až na nějaké výjimky. Jako příklad si uveďme vytvoření procedury, která vymaže z tabulky *employees* záznam podle zadaného *employee_id* [10].

```
CREATE PROCEDURE remove_emp (employee_id IN
NUMBER) IS
tot_emps NUMBER;
BEGIN
DELETE FROM employees
WHERE employees.employee_id =
remove_emp.employee_id;
tot_emps := tot_emps - 1;
END;
/
```

Kód 2: Vytvoření uložené procedury v databázi Oracle.

4.1.2.3 SQL Server

SQL Server nevybočuje z řady a také pro vytvoření uložené procedury používá příkaz CREATE PROCEDURE, který má také částečně odlišnou syntaxi od ostatních databázi. I zde si uvedeme příklad vytvoření procedury, která provede smazáni záznamu z tabulky *"Emlp"* podle zadaného idetifikátoru, který je zadán jako parametr *id* [13].

```
CREATE PROCEDURE DeleteEmplProc (
     @id INTEGER)
AS
BEGIN
     DELETE FROM "Empl"
     WHERE id = @id;
END;
GO
```

Kód 3: Vytvoření uložené procedury v databázi MS SQL Server.

4.1.3 Shrnutí základních rozdílů

Seznam rozdílů mezi těmito databázemi by vystačil na další práci (minimálně tohoto rozsahu), neboť rozdílu je velmi mnoho, i když se většinou nejedná o velké rozdíly v syntaxi, tak rozdílnost jazyků pro implementaci procedur je dostačující. Standardu jazyka SQL vyhovují všechny databáze, pouze se jednotlivé databáze liší speciálními funkcemi které jsou charakteristické pro určité databáze. Například SQL server podporuje klíčové slovo IDENTITY, které například Firebird databáze nepodporuje. Z tohoto důvodu například pro automatickou inkrementaci identifikace jednotlivých řádků tabulek se v tomto projektu bude ve všech typech databází využívat triggerů, které budou zajišťovat tuto činnost. Pro podrobnější porovnáni rozdílů mohu doporučit nahlédnutí do dokumentů na internetu zabývajících se přecházením z jednoho typu databáze na druhý, kde jsou přehledně popsány všechny rozdíly [8].

4.2 Práva a uživatelé

4.2.1 Firebird

Stejně jako všechny databázové systémy, tak i nekomerční databáze jakou Firebird je, poskytuje mechanizmus pro správu uživatelů a přidělování oprávnění na přístup k databázi. Dále pak stejně jako ostatní databázové systémy poskytuje přidělování práv k přístupu k tabulkám, pohledům a uloženým procedurám pomocí příkazu GRANT a REVOKE.

4.2.2 Oracle

Databáze Oracle poskytuje nepřeberné množství nastavování co se týče uživatelských účtů a práv pro přístup k datům, všechny informace je možné nastavovat přes aplikaci s webovým rozhraním zvanou *Enterprise Security Manager*[11].

4.2.3 SQL Server

Tato databáze disponuje pokročilou obsluhou uživatelů a jejich přístupových práv. Správu lze provádět pomocí jazyka Transact-SQL (T-SQL) nebo přes grafické rozhraní programu *SQL Enterprise Manager*. SQL Server může pracovat ve dvou režimech autentizace, *Integrated Windows Authentication* nebo *SQL Server Authentication*. Druhý způsob je vhodný pro webové aplikace, kde v tomto režimu drží vlastní databázi uživatelů, která je nezávislá na uživatelích operačního systému.

4.3 Komunikace s databází

Dnešní databázové systémy umožňují přístup k uloženým datům pomocí rozhraní ODBC, které nabízí jednotný přístup k různým databázím od různých výrobců, neboť každý výrobce databáze poskytuje ODBC drivery pro svoji databázi.

Na druhou stranu, všechny databázové systémy mají vlastní rozhraní pro přístup k uloženým datům, poskytují vývojářům seznamy API funkcí a knihovny pro rychlejší práci z daty. Bohužel každý výrobce databáze používá jiné knihovny a jinou množinu API funkcí. Například Oracle používá OCI (Oracle Call Interface) a SQL Server používá DB-Library, OLE DB.

Řešením tohoto problému by bylo použití knihovny, která by umožňovala univerzální přístup k různým databázím a přitom nevyužívala „pomalé“ ODBC, ale „rychlé“ volání API funkcí. Řešením pro tento projekt bude využití knihovny SQLAPI++ [12], která poskytuje přesně tento přístup k databázím.

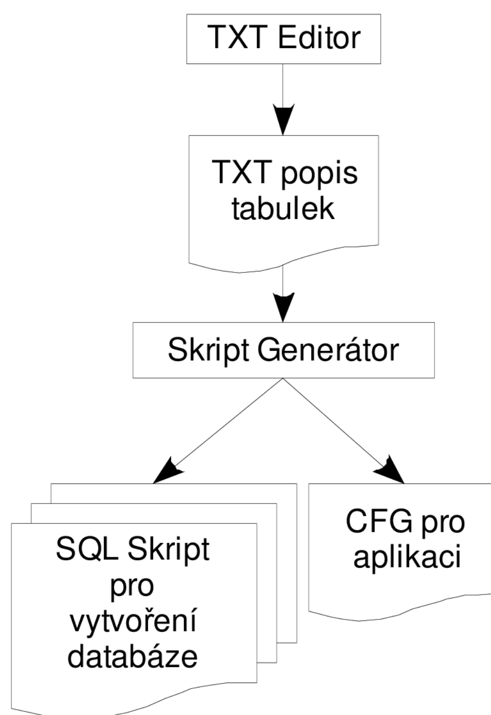
5 Návrh

Po tom co jsme se seznámili s problematikou objektově relačních vrstev, můžeme přistoupit k návrhu vlastní vrstvy. Při návrhu aplikace bude potřeba zohlednit všechny požadavku, které jsou na výsledný produkt kladeny. Právě požadavky zásadně určují jakým směrem se bude návrh a implementace ubírat a jaký bude výsledný produkt.

5.1 Požadavky

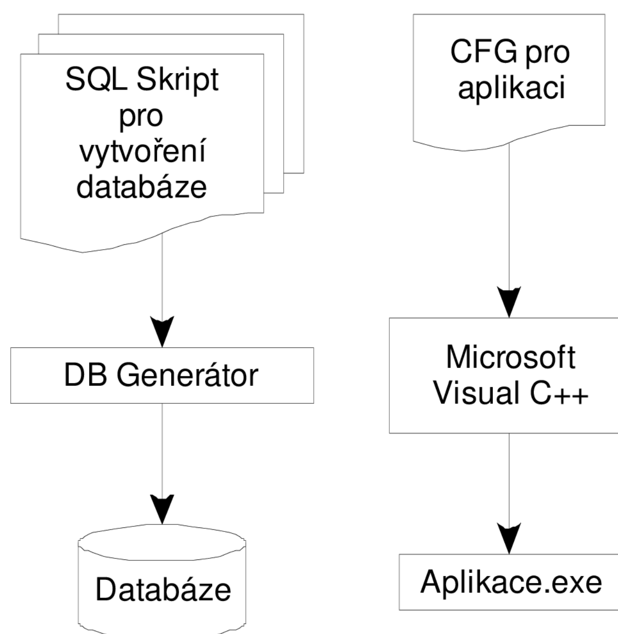
Cílem této práce je vytvoření jednoduché objektově relační vrstvy pro konkrétní nasazení ve firemním produktu, který často přistupuje k databázi, kde jsou uložena data s kterými pracuje. V současné době se využívá přímá komunikace s databázi a jednoduchá třída pro práci s výsledky dotazů nad databází. Firemní produkt je psán v jazyce C/C++, tudíž i objektově relační vrstva, která bude zapouzdřovat přímou komunikaci s databázi, bude psána v jazyce C/C++. Na internetu je možné najít několik zajímavých objektově relačních mapovačů, ale žádný nevyhovoval požadavkům firmy. Většinou se jednalo o klasické *Persistence Frameworks*, kde se procházely ručně psané speciální třídy z kterých se vytvářely objekty v databázi.

V praxi se jen velmi těžko může radikálně přejít z jedné technologie na jinou, kvůli zpětné kompatibilitě produktů a u vrstvy která poskytuje data pro celou aplikaci to platí dvojnásob. Při návrhu objektově relační vrstvy bylo nutné zohlednit současný způsob vytváření databáze. Ve stávajícím řešení je databáze vytvářena z popisu jednotlivých tabulek, který je uložen v textovém formátu a má určitou strukturu. Vývojář tento soubor edituje obyčejným textovým editorem a definuje v něm názvy tabulek, názvy a typy jednotlivých sloupců a případné další podmínky pro obsah sloupců jako je například maximální hodnota nebo délka řetězce. Pomocí speciálního programu pak z tohoto popisu vytvoří skripty pro vytvoření databáze. Z popisu tabulek se vytváří i konfigurační soubor pro aplikační vrstvu (nazvěme jej *CFG pro aplikaci*), aby bylo možné kontrolovat typ a hodnotu proměnné ještě předtím než je hodnota uložena do databáze, tedy hned na úrovni uživatelského rozhraní. Popsaný postup vytváření databázových skriptů si pro lepší pochopení znázorníme graficky.



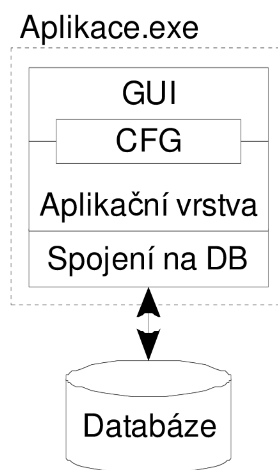
Obrázek 9: Stávající způsob vytváření databázových skriptů.

Po vytvoření skriptů a konfiguračního souboru pro aplikaci začíná práce na aplikaci, kde vývojář zpracovává data z databáze a vytváří z nich výstup pro uživatele. Pro vytváření SQL dotazů a příkazů nad databází využívá obsah SQL skriptů, kde je uložena konečná podoba tabulek v databázi včetně všech režijních (stínových) sloupců jako jsou například unikátní identifikátory. Pro typovou kontrolu vstupních dat zadaných uživatelem přes uživatelské rozhraní zakomponovává vývojář obsah souboru *CFG pro aplikaci* do uživatelského rozhraní. V *CFG pro aplikaci* jsou uloženy informace o typech a velikostech jednotlivých sloupců v databázi, aby mohl být uživatel okamžitě informován například na překročení povolené délky textového pole, nebo o zadání příliš velké číselné hodnoty. Po vytvoření aplikace která využívá databázi popsanou ve skriptu pro vytvoření databáze použije vývojář další nástroj, který ze skriptu vytvoří skutečnou databázi a pomocí MS Visual C++ vytvoří ze zdrojových souborů aplikace spustitelný soubor, který po spuštění bude pracovat s vytvořenou databází.



Obrázek 10: Dokončení postupu vytváření programu.

Vytvořená aplikace obsahuje několik vrstev, kde na nejspodnější vrstvě se nachází kód pro přístup k datům v databázi, nad kterou je vytvořena aplikační vrstva, která zajišťuje manipulaci s daty. Dále už jen následuje vrstva která pracuje z *CFG pro aplikaci* a kontroluje vstupní data od uživatele a nejvyšší vrstvu tvoří uživatelské rozhraní.



Obrázek 11: Detail rozvrstvení stávající aplikace.

Mezi další zásadní požadavek patří rozšíření podpory na více databázových systémů. V současném řešení se používá pro přístup k databázi vlastní rozhraní, které podporuje databázové systémy Firebird a Microsoft SQL Server. Perzistentní vrstva by měla také podporovat předchozí databázové systémy a navíc by měla být možnost snadného rozšíření na další databázové systémy

jako je například databáze Oracle. Vzhledem k faktu, že objektově relační vrstva bude postavena na zděděné databázi, bude nutné perzistentní objekty generovat z tabulek v databázi, nebo z jejich textového popisu a nikoliv opačně, neboť by bylo velice náročné vytvořit perzistentní objekty v takovém tvaru, aby odpovídaly současnému stavu databáze. S ohledem na tento fakt odpadá nutnost zohlednění dědičnosti mezi objekty a budou pouze zohledněny vztahy mezi tabulkami v databázi, později mezi perzistentními objekty. Tabulky v současné databázi nemají žádný vztah mezi sebou, nebo maximálně vztah 1:1 nebo 1:N, vztah M:N není v databázi použit, ale objektově relační vrstva by mela být alespoň připravena na připadnou modifikaci, která by zajistila i podporu vztahu M:N.

Firma si dále od tohoto projektu slibuje větší typovou kontrolu dat a to částečně již při kompilaci produktu a následně pak i při běhu programu. Mezi implicitní požadavky, které se očekávají od objektově relační vrstvy patří zapouzdření SQL příkazů, objektový přístup při vývoji aplikace a také řešení kolizí při vícenásobném zápisu dat do databáze.

V předchozích odstavcích jsme si popsali požadavky na objektově relační vrstvu a nyní si je všechny pro přehlednost shrneme v heslovitých bodech. Objektově relační vrstva by tedy měla splňovat:

- **kód v jazyce C/C++,**
- **zachování současné struktury databáze (práce nad zděděnou databází),**
- **zachování postupů při vývoji produktu,**
- **zapouzdření SQL příkazů na aplikační úrovni,**
- **lepší typovaná kontrola dat při překladu a běhu programu,**
- **objektový přístup,**
- **řešení konfliktů při vícenásobném zápisu dat,**
- **vztahy mezi objekty (1:1, 1:N),**
- **generování tříd z databáze nebo popisu databáze.**

5.2 Návrh nového řešení

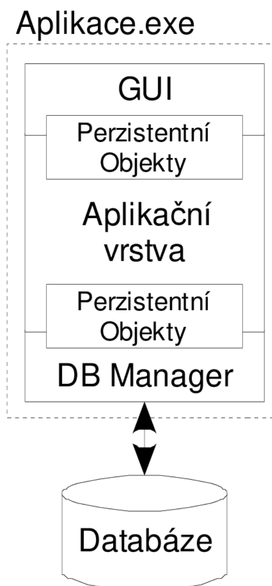
Pro vývoj takovéhoto typu produktu, který staví na statickém vytvoření části programu z předchozí fáze vývoje, tedy na postupu, kdy je napřed vytvořena jedna vrstva aplikace a již nedochází ke změnám v této vrstvě (viz. Postup vytváření aplikace popsaný v předchozí podkapitole), není možné použít klasické objektově relační mapování typu *Persistence Framework*, které dynamický mění všechny úrovně aplikace týkající se objektově relačního mapování, tzn Při změně objektu na aplikační úrovni dojde k automatické a nekontrolovatelné změně reprezentace objektu v databázi. Zvolil jsem tedy typ nazývaný jako DAO (Data Access Object), což je množina objektů, které se

vytvářejí při překladu aplikace nebo již jsou předem vytvořeny a v průběhu vývoje a běhu aplikace nedochází k jejím úpravám. Detailní popis tohoto typu objektově relační vrstvy jsme si uvedli již na začátku této práce. Pro tento typ objektů se používá název statické objekty. Klasická *Persistence Frameworks* vytvářejí perzistentní objekty na základě popisu tříd, nebo v některých případech z popisu databázového schématu. Pří návrhu objektově relační vrstvy jsem se nechal inspirovat původním řešením, které generovalo tabulky do databáze z textového strukturovaného popisu neboť tento přístup umožňuje práci více vývojářů na aplikační úrovni využívající databázi a nedochází k jejich ovlivňování v důsledku změn objektů na aplikační úrovni. Nové řešení tedy bude opět založeno na popisu tabulek, ale v novém řešení se nebude jednat o stávající textovou podobu, ale o XML formát pro který bude vytvořen speciální editor, který bude umožňovat převod ze starého textového formátu do XML a editaci nového XML formátu. Další kroky vytváření programu se jíž budou lišit od původního řešení, neboť se bude vytvářet objektově relační vrstva, která je cílem této práce a která má nahradit stávající řešení přímého volání SQL příkazů z aplikační úrovně. Vytvářená objektově relační vrstva se bude skládat z několika částí:

- bázové perzistentní třídy,
- metadata v databázi,
- vrstva v aplikaci zapouzdřující přístup do databáze.

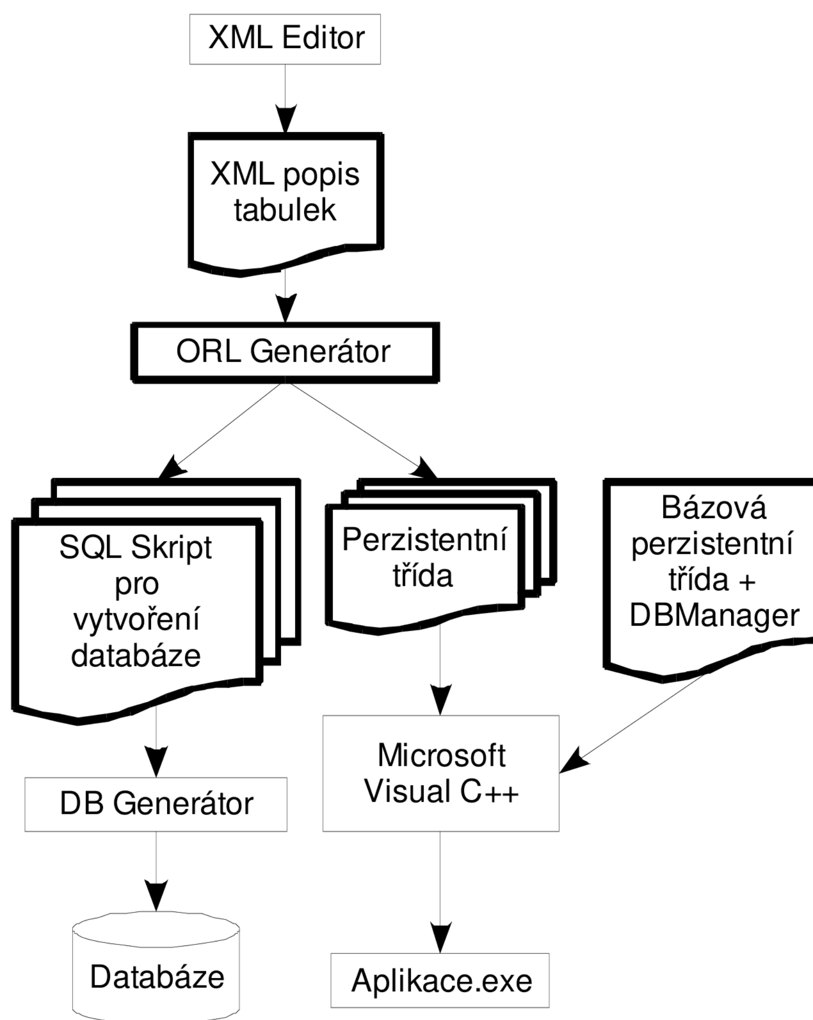
Pro vytváření konkrétních perzistentních tříd a metadat v databázi bude vytvořen ORL generátor, který bude vytvářet databázové skripty a generovat třídy pro konkrétní tabulky založené na bázové perzistentní třídě z XML popisu tabulek. Pří vytváření aplikace v prostředí Microsoft Visual C++ bude do projektu zahrnut kód pro zajištění přístupu do databáze, který bude reprezentován třídou *DBManager*. Tato třída bude mít implementovanou podporu pro práci s bázovou perzistentní třídou. Nad bázovou perzistentní třídou budou vytvořený vygenerovány konkrétní perzistentní třídy, které jsou popsány v XML popisu tabulek.

Výsledná aplikace využívající perzistentních objektů bude mít odlišnou strukturu oproti současné. Hlavním záměrným rozdílem mezi stávajícím řešením a nově vytvářeným bude zapouzdření SQL příkazů a přístupu k databázi do objektově relační vrstvy, která odstíní databázovou vrstvu od aplikační a přístup k datům v databázi bude možný jen přes vygenerované perzistentní objekty a *DBManager*. Typová kontrola která je v současné době zajišťována pomoci *CFG* v době běhu programu bude nahrazena silně typovou kontrolou již při překladu, neboť hodnoty se budou ukládat přes typově závislé metody perzistentních tříd, což bude mít přínos na zrychlení manipulace s daty, kdy se nebude provádět typová kontrola explicitně přes popis datových typů v *CFG*. Návrh nového rozvrstvení aplikace si pro lepší pochopení znázorníme na následujícím obrázku.



Obrázek 12: Detail rozvrstvení nového návrhu aplikace.

Dříve než si podrobněji popíšeme konkrétních části objektově relačního mapování, znázorníme si jednotlivé části v celkovém schématu, který ilustruje vývoj programu využívajícího generované perzistentní objekty. Schéma vychází ze současného způsoby vytváření aplikace, kde jsem pouze nahradil nebo upravil některé fáze. Hned na začátku je vidět použití XML editoru, který již svým názvem odhaluje první hlavní změnu oproti stávajícímu řešení, tou je použití XML formátu pro popis databázových tabulek oproti současné podobě popisu tabulek, která má podobu strukturovaného textu. Další změnou prošel i generátor SQL skriptů, který byl nahrazen ORL generátorem, který krom stávající funkce generátoru spočívající v generování databázových tabulek z popisu do databáze, generuje i konkrétní perzistentní třídy odpovídající tabulkám v databázi. Ze SQL skriptů se pak již generují databáze pomocí stávajícího firemního nástroje. Pro vývoj aplikace v prostředí Microsoft Visual C++ je třeba vložit již připravený projekt, který obsahuje bázovou perzistentní třídu, která je nutná pro kompilaci vygenerovaných perzistentních tříd a třídu *DBManager*, která bude sloužit jako databázová vrstva pro perzistentní objekty.



Obrázek 13: Návrh vývoj programu s perzistentními objekty.

5.2.1 XML popis tabulek

Stávající textový formát pro popis databázových tabulek je značně nevyhovující, neboť od dob návrhu tohoto formátu došlo k řadě úprav a stávající stav formátu je do budoucna nepoužitelný. Z tohoto důvodu jsem hledal nový formát pro tyto data. Textový formát má výhodu ve snadné editovatelnosti, bez nutnosti speciálního programu, ale po čase by se nový textový formát dostal do stejného stavu jako je ten současný. Při výběru formátu byla další volbou binární forma popisu tabulek, která by se editovala pouze pomocí speciálního programu, který by udržoval formát souboru v udržovatelném stavu a nastavil by mu jasný řád. Ale i binární formát má své proti, které se skrývá v nemožnosti rychlého editování a nahlížení na popis tabulek pomocí běžných textových editorů nebo Visual Studia. Proto jsem se rozhodl použít XML formát, který v souboru udržuje jasnou formu dat a data je možné editovat pomocí textového editoru, ale i pomocí speciálního programu, který bude ve firmě vyvíjen.

Při návrhu struktury XML souboru který bude popisovat tabulky, jenž budou v databázi, jsem čerpal z původního textového formátu, který zde nemohl byt uveden stejně jako ostatní firemní dokumenty, zdrojové kódy a programy, které byly použity při vývoji této práce, nebo které tato práce pro svoji činnosti využívá. Strukturu XML souboru by měly tvořit značky pro:

- identifikaci celého rozsahu značek určených pro popis tabulek,
- uvození sekce obsahující popisy všech tabulek,
- specifikaci začátku a konce jednotlivých tabulek,
- definování jmen a typů jednotlivých sloupců,
- možné alternativní významy hodnot ve sloupci a
- značky pro definování vztahu mezi jednotlivými tabulkami.

Z tohoto popisu by se již dala vytvořit základní verze struktury XML souboru, ale chybí zde ještě několik informací jako je například název tabulky, podporované datové typy, jaká omezení se budou aplikovat na sloupce tabulky a jakým způsobem se bude definovat vztah mezi jednotlivými tabulkami.

Pro identifikaci celého rozsahu značek určených pro popis tabulek jsem zvolil značku nazvanou *ObjectDatabase*. Někdo by se mohl ptát, proč *ObjectDatabase* když soubor obsahuje popis databázových tabulek a nikoliv databázových objektů? Odpověď je jednoduchá, z pohledu aplikačního programátora se bude jednat o objektovou databázi, neboť bude pracovat s objekty a jestli opravdu pracuje s objektovou databázi nebo pracuje jen s objektově relační vrstvou a relační databází jej už nemusí zajímat, proto název *ObjectDatabase*.

Pro možnost snadného rozšíření XML souboru o další konfigurační parametry, které se nemusí týkat popisu tabulek v databázi jsem použil další obecnou značku, která nese název *ObjectDBTables* a bude uvozovat veškerou definici tabulek, z pohledu aplikace, v objektové databázi.

Značce, která bude uvozovat sekci obsahující popisy všech tabulek jsem dal jméno *Tables*, jedná o jasný a výstižný název sekce, která bude následovat a tou je definice jednotlivých tabulek, respektive objektů na aplikační úrovni.

Specifikace jedné tabulky bude začínat levou párovou značkou s názvem *Table*, kde se poprvé objeví u značky i atribut, který bude mít název *name* a jeho hodnota bude řetězec specifikující název tabulky.

Mezi párovými značkami *Table* bude uložena definice jednotlivých sloupců tabulky, kde pro každý sloupec bude použita jedna párová značka nazvaná *Column*. I tato značka bude obsahovat atribut, spíše několik atributů, neboť bude potřeba definovat název sloupce a typ sloupce, který bude pro současná data nabývat jen dvou možných hodnot a to buď typu číslo nebo řetězec. U značky *Column* ještě chvíli zůstaneme, neboť je potřeba dodefinovat omezení pro jednotlivé sloupce a není

lepší místo než k tomu použít další atributy u značky *Column*. Jako první omezující atribut pro sloupec, který je důležitý pro vytvoření sloupce v databází je maximální délka řetězce, která se uvádí v kulatých závorkách za klíčovým slovem VARCHAR při definování sloupce u databázové tabulky. Mezi další omezující atributy, které bude možné u sloupce nastavit, bude minimální a maximální hodnota ve sloupci pro číselné hodnoty a pro sloupce jakéhokoliv typu pak atributy *visible* a *editable* určující, zda budou pro sloupec u perzistentního objektu vygenerovány metody pro manipulaci s tímto sloupcem či nikoliv a zda bude možné nastavovat hodnotu přes perzistentní objekt, když bude povolené generování metod pro sloupec (atribut z pohledu aplikačního programátora).

Mezi párovými značkami *Column* může být volitelně vložena značka pro specifikaci alternativních významů číselných hodnot ukládaných v databázi. Tato značka bude mít název *Option* a bude obsahovat dva atributy, kde první bude bude označen jako *label* a bude obsahovat text, který bude v databázi reprezentován číselnou hodnotou v atributu *value*. Volitelná značka *Option* bude využívána pro výčtové položky, kde typ sloupce nastavím na číselný typ a pomocí této značky nadefinuji všechny výčtové hodnoty.

Popis tabulek bude oproti stávajícímu formátu obsahovat ještě definici vztahů mezi tabulkami. Objektově relační vrstva bude obsahovat pouze jednosměrné vztahy typu 1:1 nebo 1:N a každá tabulka (objekt) bude moct mít jednosměrný vztah z více než jednou tabulkou (objektem). Pro splnění těchto podmínek jsem navrhl značku *Relationships* která bude obsahovat značky *Relationship*. Každá značka *Relationship* bude obsahovat dva atributy které budou identifikovat vztah, respektive bude obsahovat koncové uzly (tabulky) tohoto vztahu. Atribut *parentTblName* bude určovat tabulku, z které bude vycházet jednosměrný vztah do tabulky jejíž název bude uveden v atributu *childTblName*. Předpony *parent* a *child* u názvů atributů se normálně používají spíše pro znázornění dědičnosti, ale zde přesně vystihují směr vazby mezi objekty a z názvů je možné vytušit, který objekt bude obsahovat metody pro přístup k druhému objektu a který objekt bude ten závislý. Vztah mezi tabulkami se automatický promítne do definice perzistentních tříd specifikovaných jako *parent* a to v podobě metod pro přístup k *child* objektům. Navržená struktura XML souboru by měla mít následující formát (DTD k tomuto formátu je v příloze):

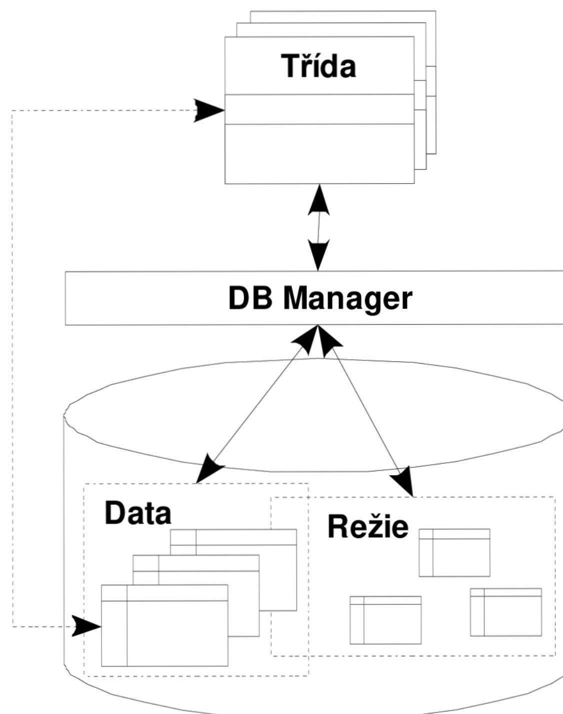
```
<ObjectDatabase>
  <ObjectDBTables>
    <Tables>
      <Table name="">
        <Columns>
          <Column name="" type="" maxLength=""></Column>
          <Column name="" type="" visible="1" editable="1">
            <Option label="" value="0"></Option>
          </Column>
        </Columns>
      </Table>
      <Relationships>
        <Relationship parentTblName="" childTblName=""/>
      </Relationships>
    </Tables>
  </ObjectDBTables>
</ObjectDatabase>
```

Kód 4: XML struktura popisující definici databázových tabulek.

5.2.2 XML popis tabulek

Při vytváření SQL skriptu je důležité mít informace o tom, co bude v databázi uložené, tedy jakou bude mít strukturu. Obsah databáze by se dal rozdělit na dvě hlavní části, na:

- datovou část a
- režijní část.



Obrázek 14: Schéma databáze s vazbou na zbytek programu.

Datová část bude obsahovat definice tabulek z větší části korespondující z XML popisem tabulek, do kterých se budou ukládat data z perzistentních objektů. Definice tabulek v databázi nebude plně korespondovat z XML popisem, neboť do databázových tabulek budou automaticky přidávány stínové informace (sloupce), které se budou týkat režijní části databáze.

Pod pojmem režijní část databáze nemám namysli systémovou část databáze, která je vytvářena výrobcem databázového systému, ale část databáze, která bude sloužit pro činnost objektově relační vrstvy a pro správu dat v datové části databáze. Mezi režijní informace v databázi budou patřit již zmiňované stínové informace, které bude obsahovat každá tabulka, která bude sloužit pro uložení dat z perzistentních objektů. Stínové informace jsou data, která nemají užitečný charakter pro uživatele, ale jsou důležitá z hlediska manipulace s daty v objektech, tedy pro objektově relační vrstvu. Mezi stínové informace, které budou uloženy u každého objektu na aplikační úrovni a u každého řádku databázové tabulky bude patřit:

- jednoznačná identifikace záznamu (objektu) v rámci jedné tabulky,
- určení verze záznamu (objektu),
- specifikace oprávnění na změnu záznamu (objektu) v databázi,
- určení tabulky s kterou je tento záznam (objekt) ve vztahu,
- jednoznačné určení záznamu (objektu) který využívá tuto hodnotu.

Poslední dva body budou dohromady tvořit obdobu cizích klíčů. Zatímco v relační databázi se vytváří jeden sloupec, který je označen jako cizí klíč do určité tabulky a tedy tento sloupec nelze využít pro uložení cizího klíče z jiné tabulky, mnou navržení dvojice toto umožňuje. Neboť první hodnota identifikuje tabulku a druhá hodnota označuje cizí klíč z této tabulky. Výhoda která se tímto řešením ztratila, je možnost kontroly hodnot cizích klíčů na úrovni databáze. Tato kontrola se musí zajišťovat na úrovni objektově relačního mapování.

Zajišťování perzistence objektů a manipulace s perzistentními objekty také vyžaduje určitá režijní data, aby mohlo být s objekty řádně manipulováno. Pro ukládání perzistentních objektů do databáze bude potřeba znát:

- název tabulky, do které se data z objektu mají uložit,
- řádek tabulky na který se mají data uložit,
- zda se jedná o již uložená (perzistentní) data nebo nově vytvořena neperzistentní data,
- do jakých sloupců databázové tabulky se mají uložit jednotlivé atributy objektu,
- zda nedošlo ke změně perzistentních dat mezi načtením a aktuálním uložením nové hodnoty a
- zda má nový objekt práva změnit stávající hodnotu v databázi.

Při načítání perzistentního objektu z databáze bude potřeba mít několik informaci které jsou vyžadovány při ukládání objektů do databáze, ale také nějaké další informace. Výčet potřebných informací je následující:

- název tabulky, z které se mají data do objektu načíst,
- řádky tabulky, respektive objekty které se mají načíst,
- schéma namapování jednotlivých sloupců tabulky na atributy třídy (objektu) a
- informace o vztahu načítaného objektu s jiným objektem.

Informací, které bude potřeba pro každý perzistentní objekt (záznam) znát není málo a bude je potřeba někde ukládat. Všechny tyto vypsané informace by se mohly ukládat v perzistentních objektech jako stínové informace, ale v kolekcích takovýchto objektů by vznikala značná redundance těchto informací. K redundanci by také docházelo, kdybych pro tyto informace vytvořil sloupce u každé tabulky, která by sloužila pro uložení perzistentních objektů.

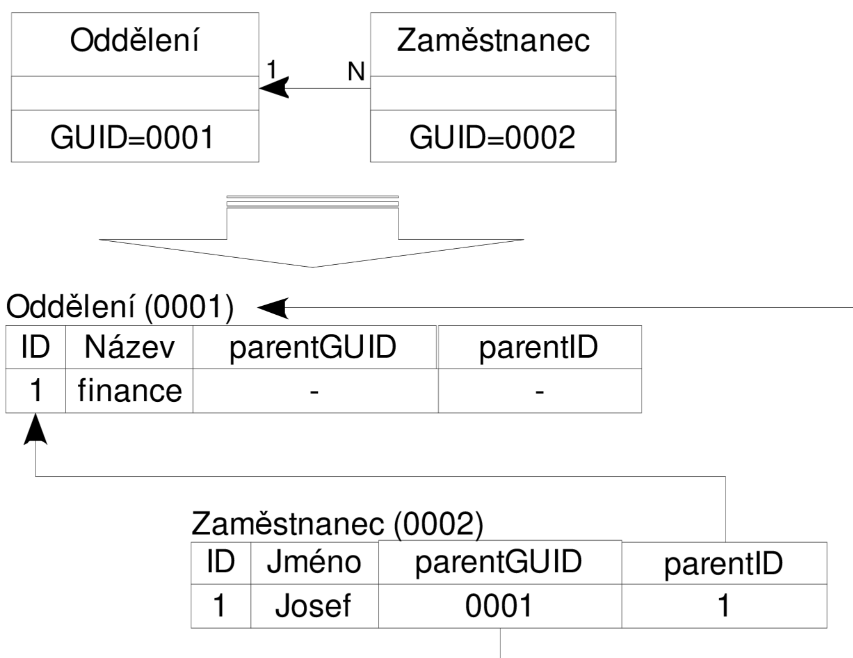
Proto jsem se rozhodl zavést v databázi skupinu tabulek, které budou obsahovat režijní data pro popis mapování objektů na tabulky v databázi. Jelikož se bude jednat o popisy mapování, nazval jsem tyto režijní tabulky jako metadata. Pro objektově relační mapovač (v mém řešení nazvaný jako DB Manager) je důležité mít přehled o všech perzistentních objektech a názvech tabulek, do kterých budou objekty ukládány. K tomu bude sloužit tabulka nazvaná *TblListMetadata*, obsahující název tabulky a globální identifikátor který bude shodný s globálním identifikátorem perzistentního objektu. Mezi další důležité informace pro *DB Managera* patří informace o jednotlivých sloupcích každé tabulky, aby mapovač mohl provádět typovou kontrolu dat a zároveň měl přehled o všech atributech u objektů. Pro tyto informace jsem vytvořil tabulku *TblColMetadata*. Kromě informací o jednotlivých sloupcích bude potřeba aby měl objektově relační mapovač k dispozici sadu příkazů pro

- vložení nového objektu,
- úpravu uloženého objektu a
- smazání uloženého objektu do databáze.

Tabulku obsahující tyto informace jsem příhodně nazval *TblSQLCmdMetadata* neboť bude obsahovat SQL příkazy zajišťující výše uvedené operace.

Doposud jsme se bavili pouze popisech, které se týkaly jen jedné databázové tabulky a jednoho objektu respektive jedné třídy. Nyní je čas si něco říct o řešení vztahů mezi objekty. Mějme dvě třídy, *Zaměstnanec* a *Oddělení* a jednosměrný vztah mezi těmito třídami, kdy na jednom oddělení bude pracovat několik zaměstnanců a jeden zaměstnanec bude moct pracovat pouze na jednom odděleních. Obě třídy mají definovány atributy pro jednoznačnou identifikaci a každý objekt těchto

tříd má rovněž jednoznačnou identifikaci, dále pak obě třídy obsahují vlastní implementaci cizích klíčů, které se skládají z identifikace třídy a identifikace objektu. Vztah mezi objekty a potažmo i mezi tabulkami v databázi si znázorníme na následujícím obrázku.



Obrázek 15: Vztah dvou objektů a záznamů v databázi.

Po vytvoření vztahu mezi objekty budeme nyní od objektově relační vrstvy požadovat aby při načtení objektu Oddělení bylo možné okamžitě přistupovat k jednotlivým zaměstnancům z tohoto oddělení. Vazba mezi zaměstnancem a oddělením je již v databázi vytvořena, ale objektově relační mapovač při načtení objektu Oddělení nemá nikde uloženu informaci o objektech, které jsou ve vztahu s tímto objektem, neboť informace o vztahu je uložena v objektu zaměstnance. Vytvořil jsem proto další režijní tabulku, která bude obsahovat informace o vztazích mezi jednotlivými třídami (tabulkami v databázi). Tabulku jsem nazval *TblRelMetadata*, což je zkratka *Table Relationships Metadata* tedy popis vztahů mezi tabulkami. Tabulka bude obsahovat GUID tabulky, na kterou se vztah odkazuje (sloupec *parentTblGUID*) a GUID tabulky z které vztah vychází (sloupec *childTblGUID*). Nyní již bude mapovač mít všechny informace o vztahu mezi objekty a při načtení objektu Oddělení načte z tabulky *TblRelMetadata* identifikace tabulek(tříd), které jsou ve vztahu s načítaným objektem a těchto tabulkách pak podle sloupců *parentGUID* a *parentID* vyhledá správné objekty, které jsou pro daný vztah relevantní.

SQL příkazy pro ukládání a mazání perzistentních objektů z/do databáze budou tvořit uložené (stored) procedury vytvářené pro každou tabulku, která bude obsahovat data z perzistentních objektů. Procedury jsem zvolil pro ušetření času a přenosu dat při operacích nad daty, které nevyžadují účast dalších částí programu. Těla uložených procedur pro příkazy SQL UPDATE a DELETE budou obsahovat kontrolu pro verzi uložených dat a kontrolu pro možné využití oprávnění k úpravě dat.

Nyní jsme si popsaly nejnižší vrstvu objektově relační vrstvy, kterou je databáze a nyní se můžeme pustit do návrhu vyšších vrstev.

5.2.3 Bázová perzistentní třída a kolekce

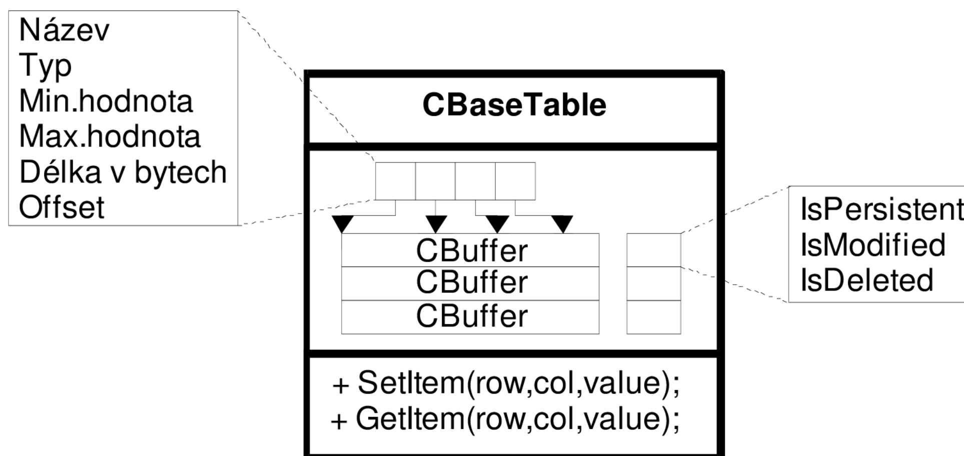
Při návrhu perzistentních tříd byl kladen důraz na snadnou úpravu těchto tříd a eliminaci úprav v kódu programu. Standardní perzistentní třídy jsou vytvářeny jako klasické třídy, které mají pevně nadefinované atributy a metody pro práci s těmito objekty a jakákoliv změna v popisu třídy znamená implicitní přegenerování tříd a objektů v databázi. Implementace se tedy ubírala směrem k vývoji obecné třídy, která by neměla pevně stanovené atributy a dala se lehce modifikovat bez nutností přegenerovat databázi a celou třídu. Byla tedy uvažována třída, která by obsahovala *buffer* pro uložení jednoho záznamu z tabulky (objektu) a kolekci struktur pro popis jednotlivých sloupců tabulky (atributů objektu), kde by každá struktura obsahovala:

- název sloupce,
- typ hodnoty ve sloupci,
- minimální hodnota ve sloupci,
- maximální hodnota ve sloupci,
- *offset* do *bufferu*, kde je hodnota uložena a
- maximální délka sloupce v bytech.

Z takto vytvořené třídy by se stejně jako ve standardní implementaci objektově relační vrstvy vytvářely kolekce, které by tak reprezentovaly celou tabulku z databáze. V současné aplikaci, pro kterou je tato objektově relační vrstva vytvářena, načítá ve většině případů více jak jeden záznam z jedné tabulky v databázi. Z pohledu objektů to znamenalo, že se ve většině případů bude pracovat s kolekcemi tříd a s jednotlivými objekty v ni uložených. Pro tyto případy by navržená třída zbytečně zabírala paměť pro uložení v kolekci redundantních informací o jednotlivých atributech (sloupcích) třídy (tabulky).

Proto jsem tedy zvolil odlišnou implementaci oproti klasickým objektově relačním vrstvám a objektovou vrstvu nezaložil na třídách, ale na kolekcích tříd. Tento krok sebou přinesl snížení paměťových nároků na uložení více objektů v kolekci. Vytvořil jsem tedy třídu která obsahuje jednu kolekci struktur, popisující jednotlivé sloupce a kolekci *bufferů*, kde jeden *buffer* reprezentuje jeden záznam z databázové tabulky (tedy jeden objekt). Ke každému *bufferu* z kolekce je přidána stavová proměnná, která určuje, zda se jedna o nový objekt, již uložený objekt nebo zda byl objekt modifikován či smazán. Takto vytvořený objekt svou strukturou hodně připomíná tabulku, kde kolekce popisující jednotlivé sloupce, respektive atributy tříd tvoří záhlaví tabulky a *buffer* pro uložení dat tvoří datový obsah tabulky. Z tohoto důvodu jsem se rozhodl třídu pojmenovat jako

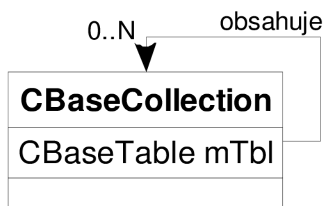
CBaseTable i přesto, že už se spíše jedná o objekt s univerzálními metodami pro přístup k atributům třídy, které jsou oproti klasickým zvyklostem s tříd uloženy v připraveném *bufferu*.



Obrázek 16: Struktura základní třídy pro uložení tabulky z databáze.

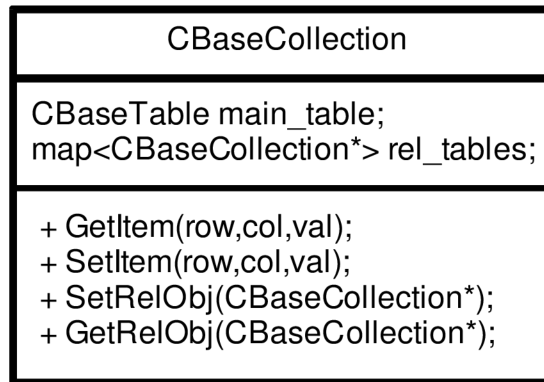
Takto vytvořený objekt je možné použít pro uložení jedné databázové tabulky. K jednotlivým řádkům a sloupcům tabulky se přistupuje pomocí obecných metod *SetItem* a *GetItem*, kde se specifikuje název sloupce nebo index sloupce a index řádku s kterým chceme manipulovat. Mohlo by se tedy jednat o zjednodušený perzistentní objekt, respektive kolekci perzistentních objektů. Aby se jednalo o použitelnou perzistentní kolekci podle požadavků, je potřeba implementovat do třídy podporu pro vztahy mezi objekty.

Pro vztahy mezi objekty jsem vytvořil třídu *CBaseCollection*, která bude obsahovat atribut typu *CBaseTable* pro uložení dat z databáze, tykajících se konkrétního objektu a kolekci ukazatelů na sebe sama. I přesto, že nejmenší element není objekt ale kolekce, bude se jednat o rekurzivní definici objektu v kolekci, která zajistí přístup k dalším kolekcím a objektům, které mají vztah s daným objektem v kolekci.



Obrázek 17: Rekurzivní definice bázové kolekce.

Třída *CBaseCollection* bude sloužit jako bázová třída pro vytvářené perzistentní objekty. Metody *SetItem* a *GetItem* budou přetížené metody pro jednotlivé datové typy podporované objektově relační vrstvou, tudíž bude zajištěna i typová kontrola dat v objektu.



Obrázek 18: Základní perzistentní kolekce pro ukládání dat z databáze.

Metody *SetRelObj* a *GetRelObj* budou sloužit pro nastavení nebo získání kolekce objektů které budou ve vztahu s daným objektem.

5.2.4 Perzistentní třída

Perzistentní třídou budeme nazývat konkrétní třídy generované pomocí *ORL Generátoru*. Konkrétní třída bude taková třída, která bude mít svůj specifický název a specifické metody pro práci s atributy této třídy. Konkrétní třídy budou vytvářeny z XML popisu tabulek (objektů) o kterém jsme se již dříve zmínily za pomoci *ORL Generátoru*. Tento generátor také vytváří databázové skripty, pomocí kterých budou vytvářeny databáze pro uložení těchto konkrétních perzistentních tříd. Aby objektově relační mapovač mohl přesně určit, který objekt má být uložen do které tabulky, použil jsem globální identifikátory pro označení dat popisujících konkrétní třídu jak na straně databáze, tak u jednotlivých tříd. V důsledku toho bude moct mapovač ke každému konkrétnímu objektu snadno dohledat v databázi potřebné informace pro uložení, či načtení dat.

Jako příklad konkrétní třídy uveďme třídu *Osoba*. O této třídě podle názvu můžeme říct, že objekty této třídy budou obsahovat informace o člověku, jako například:

- jméno,
- pohlaví,
- věk,
- atd.

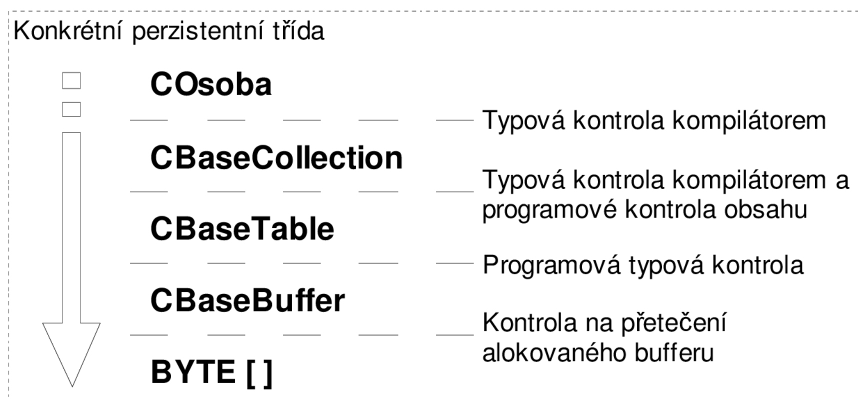
Informace o člověku budou v třídě *Osoba* uložený v atributech této třídy a pro manipulaci s těmito atributy budou sloužit konkrétní metody. Pro získání hodnoty atributu budou sloužit metody s prefixem *Get* a pro změnu hodnoty budou sloužit metody z prefixem *Set*. Tělo názvu metody pak bude shodné s názvem atributu, který bude konkrétní třída obsahovat. U atributů, které budou nabývat předem známých hodnot (např. atribut pohlaví) bude zajisté v XML popisu této třídy (tabulky)

uvedeno několik značek *Option*, které budou tvořit výčet těchto známých hodnot. Například pro atribut pohlaví to budou položky žena a muž. Aby se nemusely tuto položky v databázi ukládat jako řetězce, bude v databázi uložena jen jejich číselná reprezentace a překlad na definované názvy bude zajišťovat vygenerovaná konkrétní třída. Pro tyto atributy budou vygenerovány *Get* metody, které budou obsahovat v suffixu názvu metody slovo *Label*. Takže pro náš příklad třídy Osoba, to budou metody:

- GetJmeno,
- SetJmeno,
- GetPohlavi,
- GetPohlaviLabel,
- SetPohlavi,
- atd.

Konkrétní perzistentní objekty nebudou obsahovat atributy v pravém slova smyslu jak jsme u tříd (objektů) zvyklí, ale budou ukládány v obecných datových strukturách konkrétně ve vytvořeném bufferu *CBuffer* , který je součástí obecné třídy *CBaseTable*, která tvoří část základní perzistentní třídy *CBaseCollection* tvořící základ konkrétním perzistentním objektům. Vzhledem k tomu, že *CBaseCollection* není určena pro ukládání jednoho objektu, ale je určena pro ukládání celých kolekcí objektů, nebudeme ve výsledku pracovat pouze s jednotlivými konkrétními perzistentními třídami, nýbrž s konkrétními perzistentními kolekcemi.

Typová kontrola dat bude zajištěna i u konkrétních tříd (kolekcí tříd) a to stejně jako u bázové perzistentní kolekce již při překladu. Každá metoda u konkrétní třídy bude vracet nebo přijímat datové hodnoty jen v tom datovém typu, jaký byl pro atribut (sloupec) definován v XML popisu. Krom základní třídy *CBaseBuffer*, která bude sloužit pro uložení obecných dat jsou jednotlivé vyšší vrstvy opatřeny typovou kontrolou dat. Ve třídě *CBaseTable* je již prováděna typová kontrola dat při ukládání hodnot do *CBaseBufferů*, které *CBaseTable* obsahuje. Při ukládání objektů bude kontrolován datový typ, který je použit při popisu tabulek v XML. Tyto kontroly budou prováděny u typovaných přetížených metod *GetItem* a *SetItem*. Na vyšších vrstvách bude prováděna typová kontrola pouze při překladu a typovou kontrolu dat při běhu programu bude zajišťovat již zmíněná třída *CBaseTable*. Dále bude zajištěna i kontrola povolených vstupních hodnot jako je například minimální a maximální hodnota pro data typu číslo a pro řetězce to bude maximální povolená délka řetězce. Přehled kontrol prováděných na jednotlivých úrovních perzistentního objektu pro přehlednost a lepší pochopení znázorním na následujícím obrázku.



Obrázek 19: Přehled kontrolování dat na jednotlivých úrovních třídy.

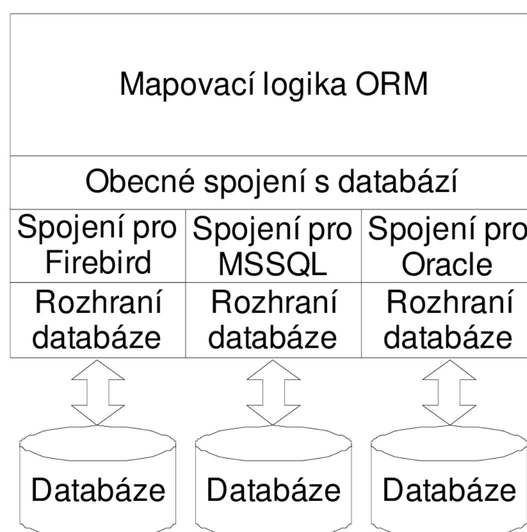
Konkrétní perzistentní kolekce tříd bude tvořit jen jednoduchou obálku třídě *CBaseCollection* která obsahuje veškerou funkcionalitu perzistentního objektu. Konkrétní kolekce tříd bude pouze definovat jednoznačný identifikátor konkrétní kolekce tříd pro objektově relační mapovač nazvaný jako *DB Manager*. Dále pak bude definovat konkrétní metody pro manipulaci s objekty této třídy, které budou využívat aplikační programátoři při zpracování dat v obchodní vrstvě aplikace nebo při zobrazování dat na uživatelském rozhraní. Vzhledem k pozdějšímu rozšíření funkcionality perzistentní vrstvy o možnost přenášení objektů mezi programy v architektuře klient-server, nebyla do tříd / objektů přidána funkcionalita pro ukládání a načítání dat do objektů. Proto u konkrétních perzistentních tříd nenajdete metody pro načtení dat nazvané *Load()* či *Retrieve()* a pro uložení metody *Save()* nebo *Persist()*. Důvodem tohoto řešení je absence objektově relačního mapovače a možnost přístupu k databázi v klientské částí systému, neboť přístup do databáze bude mít pouze serverová část systému která bude vyřizovat požadavky na data od jednotlivých klientů. Pro načítání a ukládání dat do objektů budou sloužit zmiňované metody *Retrieve()* a *Persist()* Implementované u vrstvy objektově relačního mapování nazvaného *DB Manager*, kterému se budeme věnovat v následující kapitole. Pro přenos objektů po síti bude později navržena třída, využívající vlastní komunikační protokol pro přenášení nejrůznějších informací mezi klienty a serverem a bude také implementovat funkcionalitu pro přenášení perzistentních objektů.

5.2.5 DB Manager

DB Manager bude jednou z částí objektově relační vrstvy nacházející se mezi perzistentními objekty na aplikační úrovni a tabulkami v relační databázi. Bude se tedy jednat o vlastní objektově relační mapovač zajišťující mapování objektů z aplikační vrstvy do tabulek relační databáze a naopak.

Jelikož *DB Manager* bude stále pracovat s databázi, bude mít všechny informace popisující perzistentní třídy uloženy v databázi ve speciálních tabulkách, které jsem již dříve nazval jako režijní část databáze a o jejíž složení zde také napsaly.

Architekturu *DB Manageru* jsem rozdělil do několika úrovní. Na nejvyšší úrovní bude implementována mapovací logika pro všechny perzistentní objekty a s databází bude komunikovat pomocí obecných příkazů. Pod touto úrovní bude následovat vrstva implementující napojení a komunikaci s konkrétními typy databázových systému a vyšší vrstvě bude poskytovat zmiňované obecné příkazy v podobě metod tříd. Pro zajištění shodnosti metod jednotlivých implementací přístupu k databázi bude použita virtuální třída popisující jednotlivé metody, které budou muset být implementovány. Pak nebude problém rozšířit *DB Manager* o podporu dalšího databázového systému aniž by muselo dojít k zásadním úpravám celého objektově relačního mapovače. Nejnižší úroveň *DB Manageru* bude tvořit již samotné rozhraní konkrétního databázového systému a databáze, kde budou data ukládána.



Obrázek 20: Obecná struktura navrženého objektově-relačního mapovače.

DB Manager bude pracovat s bázovými perzistentními kolekcemi objektů, které budou obsahovat všechny potřebné informace pro uložení či načtení dat z těchto objektů. Jak jsme si již popsaly v předchozí kapitole, konkrétní kolekce perzistentních objektů budou založený na bázových perzistentních kolekcích, které pouze doplní o konkrétní metody pro přístup k uloženým datům a budou specifikovat použití bázové perzistentní kolekce nastavením globálního identifikátoru pro kolekci tříd. Podle této identifikace bude mít *DB Manager* již přesné specifikované instrukce jak manipulovat s danou kolekcí objektů, respektive s daným objektem z kolekce.

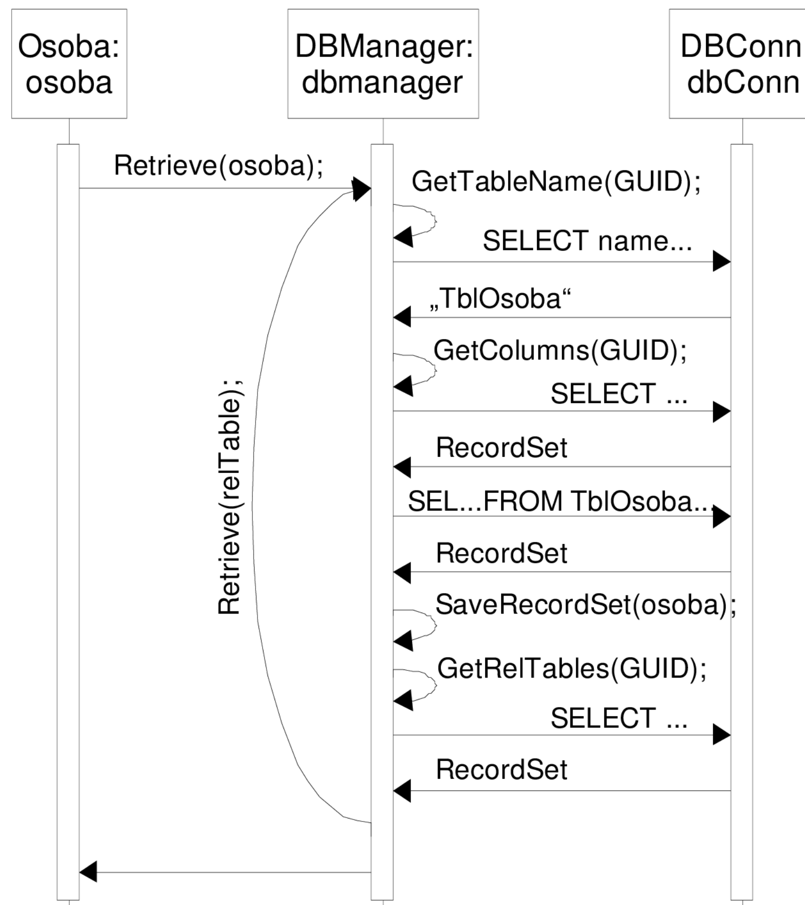
5.2.5.1 Načtení dat do objektu

Pro načtení dat z databázové tabulky do kolekce objektů bude vytvořena metoda nazvaná *Retrieve()*, která bude mít shodné chování s metodami shodných nebo podobných názvů u jiných objektově relačních mapovačů. Metoda bude mít dva parametry obsahující:

- referenci na konkrétní kolekci objektů a
- volitelnou část *where*,

která bude doplněna do SQL příkazu SELECT pro omezení získaných dat do zadané kolekce tříd. Vzhledem k požadavku na realizaci vztahů mezi objekty, bude muset metoda *Retrieve()* zajistit i načtení objektu z kterými budou jednotlivé objekty ve vztahu. Za tímto účelem se bude využívat rekurzivního volání metody *Retrieve()*.

Mějme například perzistentní kolekci objektů třídy *Osoba*. Pro načtení objektů se použije metoda *Retrieve()* z objektu *DB Managera*. Ten následně začne komunikovat z databází aby si zajistil potřebné informace pro načtení dat, jako je název tabulky z které se mají data načíst a popis jednotlivých sloupců tabulky respektive objektu třídy. Po načtení informací mapovač sestaví SQL SELECT dotaz, který provede nad vybranou tabulkou. Vrácený výsledek dotazu uloží do kolekce objektů. Nyní mapovač provede kontrolu, zda není objekt třídy osoba ve vztahu s jinými objekty. Pokud není, je ukončena metoda *Retrieve()* a aplikační metoda má k dispozici perzistentní objekty třídy *Osoba*. Pokud by třída Osoba byla ve vztahu s jinou třídou, například třídou *Adresa*, provede se uvnitř metody *Retrieve()* rekurzivní volání této metody, kde se místo kolekce objektů třídy *Osoba*, předá kolekce objektů třídy *Adresa*. Po návratu z rekurzivního volání se objekty z kolekce *Adres* uloží do tabulky vztahů objektu třídy *Osoba*. A metoda *Retrieve()* se ukončí.



Obrázek 21: Načtení dat do objektu Osoba.

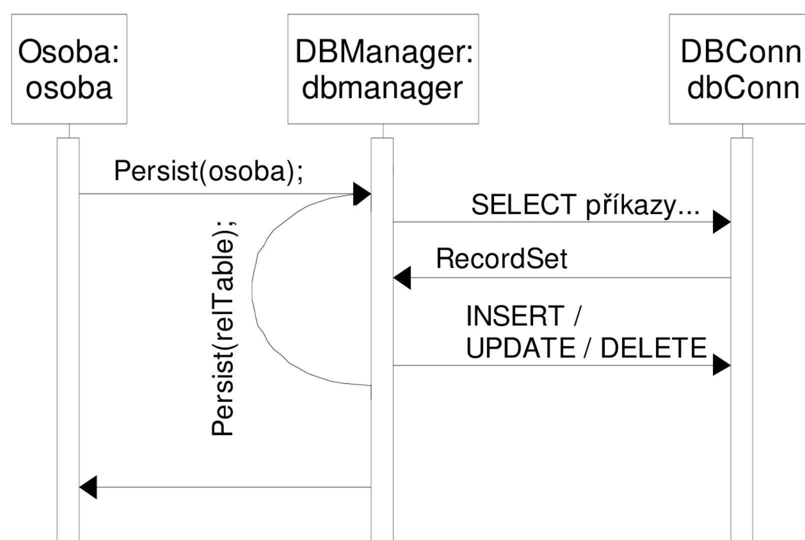
5.2.5.2 Zápis dat z objektu

Pro ukládání objektů bude opět po vzoru existujících objektově relačních mapovačů vytvořena metoda s názvem *Persist()*. Oproti metodě *Retrieve()* bude tato metoda obsahovat pouze jediný parametr, kterým bude odkaz na kolekci objektů.

Metoda *Persist()* si po zavolání bude z databáze načítat všechny potřebné příkazy pro uložení nebo smazání jednotlivých objektů z databáze. Po získání těchto příkazů zahájí procházení předané kolekce objektů a aplikovat operace na jednotlivé objekty. O způsobu manipulace s jednotlivými objekty bude *DB Manager* informován pomocí stavových informací, které bude každý objekt respektive každý buffer pro uložení objektu obsahovat. Stavové informace budou nabývat hodnot:

- nový,
- změněný,
- smazaný nebo
- již uložený objekt.

Pro objekty označené jako nové se použije příkaz SQL INSERT, který vloží nový objekt do databáze. U objektů se stavovou hodnotou nastavenou na změněný objekt, dojde na straně databáze k ověření zda ukládaná hodnota objektu může být uložena nebo zda hodnota v databázi je aktuálnější než poslední načtená verze objektu. Dále se pak provede kontrola na volitelně nastavitelnou hodnotu úroveň přístupu (Access Level) s tím, že zapsat hodnotu může jen objekt se shodnout hodnotou úrovně přístupu. Po úspěšném ověření těchto hodnot dojde k samotnému volání příkazu SQL UPDATE. Pro objekt, který je již v databázi uložen a zároveň je označen ke smazání bude provedena kontrola jako u změněných objektů a po úspěšném kontrole dojde k volání příkazu SQL DELETE na tento objekt, respektive na odpovídající záznam v databázové tabulce příslušného objektu. Pro objekty označené jako již uložené se neprovede žádná akce, neboť objekt odpovídá uložené variantě objektu, tedy objekt je již v perzistentním stavu.



Obrázek 22: Uložení dat z objektu Osoba.

5.2.6 Systém ohlašování chyb

Bylo by hezké, kdyby programy vždy pracovaly bez chyb a stabilně celou dobu životnosti, bohužel v reálném světě tomu tak není. I u době napsaných programů se mohou vyskytnout neočekávané chyby, nebo výjimečné události vyvolající chybu programu a není nic horšího, pokud se jedná o chyby neočekávané a kritické. Ve většině případů, když se uživatele zeptáte na popis akcí, které učinil těsně před pádem programu, nikdo vám přesně neodpoví. Místo toho dostanete mnoho odpovědí typu „...normálně jsem s programem pracoval a najednou z ničeho nic zahlásil nějakou chybovou zprávu, ale už si nepamatuji co bylo jejím obsahem...“ nebo vám uživatel bude tvrdit, že program občas dělá podivné věci, ale je to tak zřídka, že je těžké odhadnout, kdy se zase projeví. S tímto popisem chyb se moc dělat nedá, natož něco opravovat.

Z tohoto důvodu je vhodné zavést do programu systém, který bude všechny nestandardní chování programu zapisovat do logovacího souboru, z kterého v případě chyby bude možné vyčíst spoustu důležitých informací o chování programu nejen před chybou, kde se pak dá lokalizovat místo vzniku chyby, ale i chování programu na klientském počítači, neboť i když budete mít ve firmě farmu testovacích počítačů a program bude pracovat správně, najde se uživatel u kterého program vykazovat nestandardní chování.

Navrhl jsem tedy logovací systém, který bude zapisovat všechny relevantní informace do logovacího souboru. Ze samotných zpráv by se dalo ledacos vyčíst, ale pro rychlé odhalení místa chyby ve složitějším systému by bylo nemožné. Proto bude navrhovaný logovací systém zapisovat do souboru nejen samotnou zprávu, ale i místo ve zdrojovém souboru, kde k zápisu chyby došlo, v ideálním případě to bude dvojice zdrojový soubor a číslo řádku. Pro možnost odhalení stáří zapsaných zpráv a postupného procházení sledu událostí bude kromě všech uvedených informaci v logovacím

souboru uveden i čas a datum kdy k zápisu zprávy do souboru došlo. Výsledná zpráva v logovacím souboru by pak měla mít následující tvar:

```
<časový údaj><zdrojový soubor><číslo řádku><vlastní text zprávy>
```

Kód 5: Struktura záznamu v losovacím souboru.

Samotný logovací systém nebude sloužit jen pro zápis informací, ale bude využit i pro zjišťování úspěchu volání metod u jednotlivých objektů objektově relační vrstvy. Logovací třída by tedy mela tvořit bázovou třídu pro všechny vytvořené třídy. Logovací třída by měla obsahovat metody pro

- nastavení logovacího souboru,
- zjištění nastaveného logovacího souboru,
- nastavení zobrazovaného zdrojového souboru čí objektu v logovacím souboru,
- zápis zprávy s možností vložení čísla řádku v kódu pro opravu programu.

Dále pak metody pro běžný chod programu, týkající se zjišťování úspěchu provedení metod programu, tuto činnost budou v logovací třídě zajišťovat metody pro

- zjištění aktuálního chybového stavu a
- vynulování chybového stavu.

Při volání metody pro zápis zprávy do souboru se automaticky nastaví chybový stav na aktivní a v kódu, který volal metodu třídy založenou na logovací třídě, muže otestovat její úspěch nebo neúspěch pomocí metody pro zjištění chybového stavu. Tato metoda má typicky název *Failed()* a vrací hodnotu *true* pokud došlo k zápisu do logovacího souboru, nebo *false* pokud metoda proběhla úspěšně.

6 Implementace

6.1 Popis použitých technologií v programu

Při implementaci programu byl použit jazyk C/C++, který je také použit při vývoji ostatních programů ve firmě, kde vytvářený program bude jejich součástí.

Použitý programovací jazyk již vymezuje skupinu technologií které je možné použít. Vzhledem k možným budoucím požadavkům na přenositelnost objektově relační vrstvy na platformu Unix/Linux nebylo možné při implementaci využít speciální technologii pro určitou platformu. Vycházel jsem tedy pouze z možností, které nabízí programovací jazyk C/C++. Tedy jedinou technologii, kterou jsem mohl využít byla standardní knihovna šablon, neboli STL. Mnoho specifikací standardní knihovny C++ je založeno na STL od Silicon Graphics (SGI). Při implementaci jsem používal prostření Microsoft Visual C++ 2003 a jimi specifikovanou knihovnu STL, jestli se jedná o specifikaci založenou na STL od SGI se mi nepodařilo zjistit. Knihovna STL jak sám název napovídá obsahuje mnoho šablonových tříd jako jsou kontejnery a asociativní pole a algoritmy. Tyto třídy mohou být využívány jak s vestavěnými datovými typy jazyka C++, tak s uživatelsky definovanými datovými typy.

Jedinou výjimkou při vývoji maximálně přenositelné objektově relační vrstvy byla implementace *ORL Generátoru* který pro čtení XML popisu dat využívá volně dostupnou třídu *CXmlNodeWrapper*[14]. Jedná se o třídu zapouzdřující práci s MSXML, který slouží pro práci s XML. MSXML od společnosti Microsoft je implementací modelu COM (Component Object Model) vycházejícího z modelu W3C DOM (Document Object Model). Implementace *CXmlNodeWrapperu* je tedy založena na MFC a není možné ji přenést na Unix/Linux platformu. Nejedná se však o závažné omezení, neboť ORL Generátor je zapotřebí pouze při vytváření perzistentních objektů, které je možné provést na platformě Windows a vygenerované soubory přenést na platformu Unix/Linux, kde může dále pokračovat vývoj programu využívající objektově relační vrstvu.

Pro univerzální přístup k databázi byla využita knihovna SQLAPI++, která byla nasazena i v komerčním řešení této práce. Tato knihovna podporuje velké množství databázových systému a pro komunikaci s nimi využívá volání nativních API funkcí RDBMS (Relational database management systém) konkrétní databáze. Další výhodou knihovny je podpora platformy Windows i Unix/Linux.

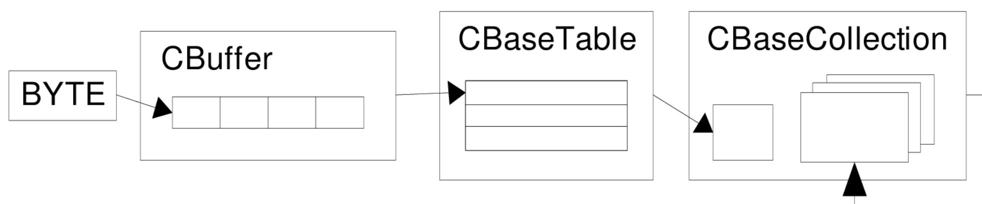
6.1.1 Datové typy

Při implementaci jsem na nejnižší úrovni používal datový typ BYTE, který odpovídá datovému typu *char*. Pro ukládání řetězců jsem použil pole znaků *char[]* a také hojně využil STL řetězců *std::string*.

Veškeré kolekce které se ve třídách vyskytují jsou implementovány pomocí STL kontejnerů *std::vector* a asociativního pole *std::map*.

6.1.2 Datové struktury

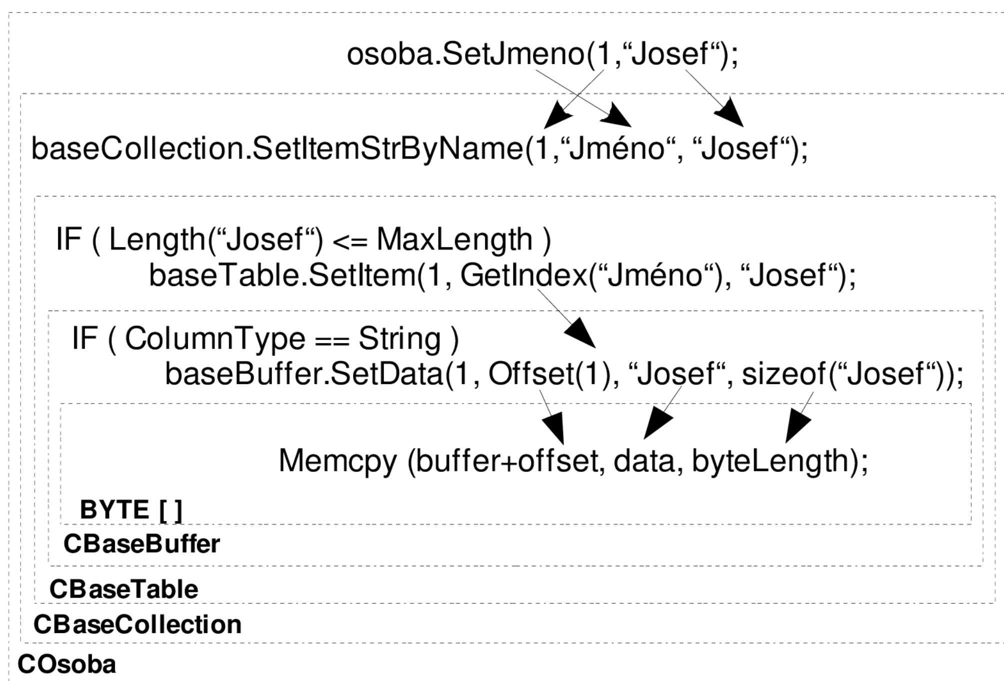
Pomocí datového typu BYTE jsem vytvořil úložiště pro ukládání dat perzistentních objektů a nazval jej *CBaseBuffer*. *CBaseBuffer* se pak stal základem třídy *CBaseTable*, která tvoří základ pro *CBaseCollection*, což je základní perzistentní kolekce tříd, neboť slouží pro ukládání perzistentních objektů. Třída *CBaseCollection* je pak použita jako základ pro vygenerované konkrétní perzistentní kolekce tříd.



Obrázek 23: Rozkreslení struktury datových typů pro perzistentní objekty.

6.2 Řešení problémů spojených s implementací

Při implementaci perzistentních objektů bylo důležité zajistit typovou kontrolu dat, díky niž by se odstranila řada chyb způsobených chybnými typovými převody za běhu aplikace, které by se v některých případech velmi často těžce dohledávaly. Díky typové kontrole a kontrole obsahu některých dat již nebude potřeba provádět dodatečnou kontrolu dat, která se v současném stavu musí provádět explicitně. Typová kontrola se tedy zajišťovala na všech úrovních perzistentního objektu kde bylo potřeba zajistit kontrolu ukládaných či načítaných dat.



Obrázek 24: Způsob provádění typové kontroly na jednotlivých úrovních objektu.

6.3 Možnosti dalšího rozšíření objektově relační vrstvy

V průběhu vývoje objektově relační vrstvy jsem se snažil vytvářet každou její část s ohledem na možné budoucí rozšíření. Mezi hlavními možnostmi rozšíření bych viděl rozšíření použití vytvořené vrstvy nad rámec platformy Windows a uplatnění na Unix/Linux systémech. Dále pak rozšíření vztahů o obousměrné vztahy a přidání funkcionality pro vztahy many-to-many. Perzistentní objekty byly vytvářeny jako jednoduché objekty, které slouží pouze pro uložení a manipulaci s daty. Funkcionalita pro načítání a ukládání objektu z/do databáze byla přesunuta do objektově relačního mapovače. Díky jednoduchosti těchto objektů je možné práci dále rozšířit o možnost přenášení objektů mezi počítači případně o možnost zálohování objektu do souboru. Technologie objektově relačního mapování je rozsáhlá oblast poskytující široké spektrum možností pro další rozšiřování jak na straně databázové, tak na straně aplikační. Na straně databáze by se mohlo pro další verze implementace uvažovat o přechodu pouze na objektový přístup a zrušení zpětné kompatibility struktury databáze a výběr lepšího řešení pro ukládání objektů a větší využití dotazování na objekty na straně databázového systému. Na Aplikační úrovni by se pak mohlo jednat o rozšíření objektů o podporu přímého napojení na prvky uživatelského rozhraní, nebo možnost automatického převodů objektů na report.

7 Závěr

V úvodu práce jsme se seznámili se způsoby jakými objektově relační mapovače zapouzdřují relační databáze a jaké požadavky jsou kladeny na tyto mapovače. V dalších částech jsme se pak seznámili se způsoby ověřování právoplatného přístupu k uloženým datům a o problémech vícenásobného přístupu. V závěru analýzy jsme se pak lehce seznámili s databázemi, které budeme v této práci uvažovat.

Cílem návrhu bylo vytvoření struktury databázově neutrální objektově relační vrstvy, která bude použita jako mezivrstva mezi aplikaci, která v současné době přímo přistupuje do databáze s možností výběru databáze. Vytvořená vrstva zapouzdřuje relační databázi a přímé volání SQL příkazů, komunikace s databází je pouze pomocí perzistentních objektů na aplikační úrovni, společně s veřejným rozhraním vytvořené objektově relační vrstvy.

Podle metod zapouzdření, které jsme si popsali se jedná o statickou verzi *Persistence Framework*, tedy tato vrstva bude připomínat metodu *Data Access Object*. Vytvořená vrstva zajišťuje přístup k jednotlivým databázím pomocí metody *connect* v které se specifikuje typ databáze. Jelikož tato vrstva je vytvořena nad stávající databázi, budeme hovořit o zděděné databázi. Objekty nad touto databází, respektive tabulky v databázi jsou mapovány na třídy aplikační vrstvy a řádky tabulky pak na jednotlivé objekty.

V této práci jsem se nezabýval vytvářením čistého objektově relačního mapovače (ORM), ale spíše jen využíval těchto vlastností a v porovnání s ORM se bude jednat o jednoduchý objektově relační mapovač. Tedy v této práci jsem se zaměřil na řešení mapování objektů do databáze a zapouzdření SQL a relační databáze do objektové vrstvy. Ve snaze omezit paměťové nároky spojené s ukládání m jednotlivých objektů v paměti jsem nepoužil klasické mapování jednotlivých objektů na řádky tabulky, které je známé z technologie objektově relačního mapování. Místo toho jsem zvolil obecnější přístup a vytvořil mapování celých kolekcí objektů na tabulku. Tyto kolekce objektů jsem navíc vytvořil jako databázově nezávislé objekty, tedy objekty (kolekce objektů) mohou pracovat s jakoukoliv databází a navíc nejsou závislé na konkrétním objektově relačním mapovači, neboť samotné objekty nemají implementovanou podporu pro ukládání objektů. Díky tomu je možné vytvořit i jinou mapovací vrstvu, nebo například vrstvu pro přenášení objektů mezi programy či počítači. Další výhodou separace podpory pro ukládání objektů je další snížení paměťových nároků na uložení jednotlivých objektů. Při implementaci vztahů mezi objekty jsem za účelem sjednotit přístup k objektům ve vztahu s načteným objektem použil místo dvou typů vztahů 1:1 a 1:N pouze vztahy 1:N, kde se pro modelování vztahu 1:1 bude využívat vztah 1:N kde hodnota „N“ bude jedna. Implementovat vztah 1:1 nebylo výhodné z důvodu snížení paměťových nároků a snížení složitosti objektově relačního mapování. Navíc tato vrstva bude nasazena v programech, kde se vztahy téměř nevyskytují nebo vyskytují pouze vtahy 1:N. Komerční verze této práce bude dále rozvíjena a

optimalizována pro použití na aplikační úrovni jako bude například podpora pro přímé napojení objektů na prvky uživatelského rozhraní.

Vytvořená objektově relační vrstva by mohla najít uplatnění v nejrůznějších aplikacích jako jsou například různé druhy informačních systémů, které ukládají a načítají data z databáze a pro snadnější modelování objektů z reálného světa by mohly byt použity perzistentní objekty. Vytvořená vrstva nemusí primárně sloužit pouze pro usnadnění práce s daty v databázi, může byt využita i pro ukládání konfigurace programu do databáze, kde budeme ukládat rozměry a pozice dialogových oken programu. Třídy ve vytvořené objektově relační vrstvě jsou vytvořeny jako nezávislé na vrstvě objektově relačního mapování, tedy po doimplementování vrstvy která by podporovala bázovou kolekci je možné třídy využít například pro zasílání objektů mezi programy a počítači.

Literatura

[1] Ambler, S. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, John Wiley & Sons, 2003.

[2] Burget, R. *JDO - Java Data Objects*, interval.cz, 6.1.2005, http://interval.cz/clanky/jdo-java-data-objects/

[3] *ActiveX Data Objects*, Wikipedia, 2006, http://en.wikipedia.org/wiki/ActiveX_Data_Objects

[4] *Enterprise JavaBeans Technology*, Sun Microsystems, Inc., 2006, http://java.sun.com/

[5] Fussell M. L., *Foundations of Object Relational Mapping*, 1997, http://www.chimu.com/publications/objectRelational/index.html

[6] Amber W., *The Design of a Robust Persistence Layer For Relational Databases*, 21.6.2005, http://www.ambysoft.com/downloads/persistenceLayer.pdf

[7] *InterBase 6 Data Definition Guide*, Ibphoenix, 2006, http://www.ibphoenix.com/downloads/60All.zip

[8] Ruiz M. L., *Migration from MS-SQL to Firebird*, 2005, http://firebird.sourceforge.net/devel/doc/manual/pdf/MSSQL-to-Firebird.pdf

[9] *Transact-SQL Reference*, Microsoft MSDN, 2006, http://msdn.microsoft.com

[10] *Oracle® Database SQL Reference 10g Release 2 (10.2)*, Oracle, B14200-02, 2005, http://www.oracle.com/

[11] Oracle® Enterprise User Administrator's Guide 10g Release 2 (10.2), Oracle, B14269-01, 2005, http://www.oracle.com/

[12] *C++ library for accessing SQL databases*, SQLAPI++, 2006, http://www.sqlapi.com/

[13] Henderson K., SQL Server Stored Procedure Basics, 2002, http://www.awprofessional.com/

[14] Hazanov A., *CXmlNodeWrapper, C++ XML the easy way!*, CodeProject.com, 10.6.2005, http://www.codeproject.com/cpp/C___XML_wrapper.asp

Seznam příloh

Příloha 1. DTD pro vstupní XML soubor

Příloha 2. Příklad použití ORL

Příloha 3. CD/DVD

Příloha 1. DTD pro vstupní XML soubor

```
<!DOCTYPE ObjectDatabase [

      <!--
      DTD pro XML soubor popisujici tabulky a objekty pro ORL.
      ========================================================-
      Autor: Pavel Jeza
      Mail: pavel_jeza@centrum.cz
      Vytvoreno 12.4.2007
      -->

  <!-- Hlavni element v souboru -->
  <!ELEMENT ObjectDatabase (ObjectDBTables)>
  <!-- Zacatek popisu jednotlivych tabulek(objektu) pro ORL -->
  <!ELEMENT ObjectDBTables (Tables)>
  <!-- Zde muze existovat nekolik tabulek a volitelne i vztahy -->
  <!ELEMENT Tables (Table+,Relationships?)>
  <!-- Kazda tabulka ma definovane sloupce -->
  <!ELEMENT Table (Columns)>
    <!-- Kazda tabulka musi mit svuj nazev -->
    <!ATTLIST Table name CDATA #REQUIRED>
  <!-- Tabulka obsahuje minimalne jeden sloupec -->
  <!ELEMENT Columns (Column+)>
  <!-- u sloupce mohou byt uvedeny alternativni reprezentace hodnot -->
  <!ELEMENT Column (Option*)>
    <!--
      nazev a typ sloupce jsou povinne,
      ostatni volitelne podle potreby
    -->
    <!ATTLIST Column name CDATA #REQUIRED>
    <!ATTLIST Column type (STRING|DWORD) #REQUIRED>
    <!ATTLIST Column visible (1|0) #IMPLIED>
    <!ATTLIST Column editable (1|0) #IMPLIED>
    <!ATTLIST Column maxLength CDATA #IMPLIED>
    <!ATTLIST Column minValue CDATA #IMPLIED>
    <!ATTLIST Column maxValue CDATA #IMPLIED>
  <!ELEMENT Option EMPTY>
    <!-- alternativni text pro hodnotu value ulozenou v databazi -->
    <!ATTLIST Option label CDATA #REQUIRED>
    <!ATTLIST Option value CDATA #REQUIRED>
  <!ELEMENT Relationships (Relationship*)>
  <!ELEMENT Relationship EMPTY>
    <!-- u parent tabulky se budou vytvaret metody pro pristup k child -->
    <!ATTLIST Relationship parentTblName CDATA #REQUIRED>
    <!ATTLIST Relationship childTblName CDATA #REQUIRED>

  <!-- konec popisu -->
]>
```

Příloha 2. Příklad použití ORL

Připojení k databázi

```
CDBManager dbm;
dbm.Connect("C:\\obj_db.fdb", "SYSDBA", "masterkey", eDBFirebird);
if (dbm.IsConnected()==false) return;
```

Vytvoření kolekce

```
CZamestnanec zam;
```

Inicializace kolekce před použitím

```
dbm.InitObject(&zam);
```

nebo rovnou načtení kolekce objektů

```
dbm.Retrieve(&zam);
```

Vytvoření objektu

```
long zamIdx = zam.CreateObject();
```

Nastavení hodnoty

```
zam.SetJmeno("Koholík Al"), zamIdx);
```

Přiřazení objektu

```
COddeleni odd;
dbm.InitObject(&odd);
DWORD oddIdx = odd.CreateObject();
odd.SetNazev("Finance"), oddIdx);
zam.SetOddeleni(zamIdx, odd);
```

Výpis objektů

```
DWORD nObj = zam.GetObjectsCount();
for (DWORD i=0; i < nObj; i++) {
  std::cout << "Jmeno='" << zam.GetJmeno(i) << "'\n";
  COddeleni oddeleni;
  dbm.InitObject(&oddeleni);
  if (SUCCEEDED(zam.GetOddeleni(i, oddeleni))) {
    for (DWORD idx = 0; idx < oddeleni.GetObjectsCount(); idx++) {
      std::cout << "Oddeleni.Nazev=" << oddeleni.GetNazev(idx) << "\n";
    }
  }
}
```

Smazaní objektu z kolekce

```
zam.DeleteObject(i);
```

Uložení změn v kolekci

```
dbm.Persist(&zam);
```

Odpojení od databáze

```
dbm.Disconnect();
```

Příloha 3. CD/DVD

Obsah:

- Technická zpráva v elektronické podobě.
- Programová dokumentace.
- Zdrojové texty programu.
- Ukázka použití ORL.
- Další pomocné programy
    - Databáze
        - Firebird 2.0
        - Microsoft SQL Server Express
        - Oracle XE
    - Knihovna SQLAPI++