TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky a mezioborových inženýrských studií

Studijní program: B2612 – Elektrotechnika a informatika
Studijní obor: 2612R011 - Elektronické informační a řídicí systémy

Přehled možností perzistentního ukládání objektů

Overview of possibilities of the persistent storage of objects

Bakalářská práce

Autor: **Pavel Votoček**
Vedoucí práce: Ing. Miroslav Holubec
Konzultant: Ing. Zdeněk Motl

V Liberci 17. 5. 2006

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky a mezioborových inženýrských studií

Katedra: Katedra aplikované informatiky Akademický rok: 2005/2006

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jméno a příjmení: Pavel Votoček

studijní program: B 2612 – Elektrotechnika a informatika

obor: 2612R011 – Elektronické informační a řídicí systémy

Vedoucí katedry Vám ve smyslu zákona o vysokých školách č.111/1998 Sb. určuje tuto bakalářskou práci:

Název tématu:

Přehled možností perzistentního ukládání objektů

Zásady pro vypracování:

1. Stručně popište koncept objektově orientovaného programování a standardní mechanizmus ukládání objektů do relačních databází
2. Zdomumentujte co nejširší přehled aplikačních rámců, řešících tuto problematiku, včetně jejich stručné historie
3. Pro každý rámec zrealizujte ukázkové řešení uložení a získání objektu, v rámci možností i vyhledání dle určitých kritérií
4. Proveďte porovnání těchto aplikačních rámců, vyzdvihněte jejich přednosti a konstatujte vhodnost použití

Rozsah grafických prací: dle potřeby dokumentace

Rozsah průvodní zprávy: cca 40 stran

Seznam odborné literatury:

[1] Sarang P.G. a kol.: Professional EJB, 1200p., Wrox Press (July, 2001)

[2] Bauer Ch., King G.: Hibernate in Action, 408 p., Manning Publicatios (August 1, 2004)

[3] Jordan D., Russell C.: Java Data Objects, 380 p., O'Reilly Media, Inc.; 1 edition (April 22, 2003)

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

Vedoucí bakalářské práce: ing. Miroslav Holubec

Konzultant: ing. Zdeněk Motl

Zadání bakalářské práce: **25.10.2005**

Termín odevzdání bakalářské práce: **19. 5. 2006**

....

Vedoucí katedry Děkan

V Liberci dne 25.10.2005

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum:19.5.2005

Podpis:Pavel Votoček

Poděkování

V úvodu své práce bych rád poděkoval svému vedoucímu inženýru Holubcovi za jeho cenné rady a náměty, které vedly k vzniku této práce.

Abstrakt

Tématem této bakalářské práce je přehled možností perzistentního ukládání objektů. Objekty je možné ukládat buď do souboru na disku nebo do databáze. I když je ukládání dat do databází pouze jiným způsobem ukládání dat na disku, tak z pohledu programátora jsou oba tyto přístupy dosti odlišné. Tato práce je zaměřena především na problematiku ukládání dat do databází, a to prostřednictvím objektově orientovaného jazyka Java. Je zde popsán standardní mechanizmus ukládání objektů do relačních databází prostřednictvím rozhraní JDBC (Java Database Conectivity). Dále jsou zde charakterizovány vybrané objektově relační rámce. Jsou zde popsány dva tyto rámce a to Hibernate a JDO (Java Data Objects). Práce je uzavřena praktickým vyzkoušením těchto rámců a srovnáním vhodnosti jejich použití.

Abstract

The aim of my bachelor work is an overview of possibilities of the persistent storage of objects. Objects can be saved either into the file on the disc or into the database. Even if saving the data in database seems to be just another method for data saving on the disc. For programmer are both of these approaches pretty different. This work is aimed at the problems of data saving into the databases namely through the object-oriented programming in Java. Here is described standard mechanism of object storing in relational databases by the JDBC (Java Database Conectivity). Next I´ve picked up and characterized object-relational frameworks here. I described two of these frameworks: Hibernate and JDO (Java Data Objects). My bachelor work is closed by practical testing of these frameworks and comparing with propriety of their usage.

Seznam zkratek

ADT	abstraktní datový typ
ALGOL	ALGOrithmic Language
ANSI	American National Standards Institute
API	Application Programming Interface
APL	Algorithmic Procedural Language
atd.	a tak dále
CLI	Call-Level Interface
COBOL	COmmon Business Orientd Language
CSF	Connected Services Framework
DB	DataBase
DB2	DataBase 2
DLL	Dynamic Link Library
IBM	International Business Machines
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JSP	JavaServer Pages
LISP	LISt Processing
mj.	mimo jiné
např.	na příklad
ODBC	Open DataBase Connectivity
ODBMS	Object Database Management Systém
OID	objektový identifikátor
OO	objektově orientovaný
OOP	objektově orientované programování
ORDBMS	Object Relational" Database Management Systém
RDBMS	Relational Database Management System
RDL	Report Definition Language
SEQUEL	Structured English Query Language
SIMULA	SIMple Universal LAnguage
SQL	Structured Query Language
SQL/DS	Structured Query Language/Data System

tzv. tak zvané

Seznam obrázků

Ochranné registrované známky

Úvod

Každý kdo se zabývá programováním se ve své praxi dříve či později setkal s problémem, jak ukládat data, s kterými vyvíjená aplikace pracuje? Rozhodneme–li se svou aplikaci vybavit možností ukládat data, musíme bezpodmínečně určit, kam svá data chceme ukládat. Z nabízených možností se nejčastěji používají dva způsoby uložení dat. Jsou to uložení dat do souboru na disku nebo do databáze. V praxi je ukládání dat do databáze jen jiným způsobem zápisu na disk. Databázové systémy totiž obvykle ukládají své informace na pevný disk. Z pohledu programátora i samotné koncepce se ovšem oba přístupy v mnohém liší. V této práci se zaměřím na popis možností perzistentního ukládání objektů do databází v objektově orientovaném jazyce Java, v kterém bude celá tato práce realizována.

V první části práce se budu věnovat popisu stručné historie vývoje programovacích jazyků, kde se zaměřím pouze na klíčové zástupce jednotlivých stupňů vývoje. Následně provedu porovnání procedurálního a objektového přístupu k programování a stručně charakterizuji koncept objektově orientovaného programování. Dále se zaměřím na možné způsoby zpracování dat a tvorby databází (relační, objektová a objektově-relační databáze). V závěru této části objasním standardní mechanismus ukládání objektů do relačních databází v programovacím jazyce Java. Tímto standardem je aplikační rozhraní JDBC (Java Database Coneectivity), které umožňuje přístup aplikace k datům z různých existujících specifikací standardů zdrojů dat.

Obsahem druhé části bude seznámení s vybranými objektově relačními aplikačními rámci. Zaměřím se především na dva z velkého množství existujících aplikačních rámců a to na Hibernate a JDO (Java Data Objects).

V poslední části se zaměřím na popis praktické realizace použití výše zmiňovaných aplikačních rámců.

1. Programovací jazyky a druhy databází

1.1.Stručný popis vývoje programovacích jazyků

První programy na prvních počítačích byly programovány ve strojovém kódu (elementární instrukce vyjádřené číselnými kódy a používání absolutních adres). Při programování ve strojovém kódu je nutné znát do podrobna architekturu počítače, pro který program píšeme.

Tento způsob programování je velice obtížný, proto se brzy objevily nástroje usnadňující programování. Jsou to jazyky symbolických adres, které jsou souhrně nazývany asemblery. V těchto jazycích jsou číselné kódy instrukcí nahrazeny mnemotechnickými zkratkami. Různá místa v programu lze označit návěštími, symbolickými adresami a na tato návěští se lze v instrukcích odvolávat. U programů napsaných v asembleru se pro převod do strojového kódu musí použít překladač.

Programátor, který používal asemblér musel stále velmi podrobně znát architekturu počítače i to byl jeden z důvodů, že programování bylo z počátku záležitostí úzce specializovaných odborníků. Skutečnou změnu způsobily až vyšší programovací jazyky. Za první použitelný vyšší programovací jazyk se zpravidla pokládá Fortran. Fortran vyvinul tým vedený Johnem Backusem u IBM pro počítač IBM 704. Vývoj trval od roku 1954 do roku 1957. Největší síla Fortranu spočívala v možnosti zapisovat složité matematické vzorce způsobem blízkým matematice. Na rozdíl od mnoha pozdějších programovacích jazyků umožňoval Fortran rozdělit program na samostatně překládané programy. Neobsahoval však konstrukce jako složený příkaz nebo úplný IF, kromě polí nepodporoval uživatelem definované typy, rekurzi při volání podprogramů atd., Fortran byl určen především pro vědecko technické výpočty.

Úspěch Fortranu způsobil zájem o vytváření dalších jazyků. Jazyky z té doby byly problémově orientované, byly určené k řešení poměrně úzké skupiny problémů. Následující programovací jazyky lze z hlediska vývoje programování považovat za klíčové. Algol 60 je jazyk, který pracoval s blokovou strukturou programu a využíval rekurzi. Poskytl také úplný příkaz IF, který ve Fortranu chyběl.V roce 1960 byl na objednávku Ministerstva obrany USA vytvořen jazyk Cobol 60, určený pro zpracování hromadných dat (předchůdce dnešních databází). Cobol seznámil programátory s konstrukcí záznam, která reprezentuje

jeden řádek v databázové tabulce. Zcela jiný přístup k programování nabídl LISP, publikovaný skupinou Artificial Inteligence Group vedenou prof. J. McCarthym. LISP je jazyk určený pro zpracovávání seznamů. LISP ukázal, že imperaktivní programovací jazyky, v nichž je program zapsán jako posloupnost příkazů, nepředstavují jedinou možnost. Podnítil zájem o dynamické datové struktury. Dalším zástupcem z počátku 60. let je jazyk APL , který považuje za základní jednotku pole a umožňuje s ním provádět nejrůznejší operace včetně změny tvaru v průběhu zpracování.

Dalším stupněm ve vývoji programovacích jazyků bylo vytvoření univerzálního programovacího jazyka. Prvním použitelným univerzálním programovacím jazykem byl PL/I (IBM, 1964) Šlo o spojení vlastností Fortranu, Algolu a Cobolu. Jako jeden z prvních umožnil ošetřovat vyjímečné situace za běhu programu, používat ukazatele a paralelně provádět vstupně výstupní operace. Jazyk Simula 67, vznikl jako nadstavba Algolu 60. Jedná se o plnohodnotný objektově orientovaný jazyk. Jsou v něm třídy, dědičnost, polymorfismus, automatická správa paměti a prostředky pro práci se seznamy. Výrazným zástupcem byl Algol 68. Tento jazyk přinesl velké množství nových možností, jako je paralelní programování, základní prostředky pro synchronizaci procesů, řízení alokace paměti, zacházení s funkcemi jako z datovými objekty a další.

Výrazným obratem ve vývoji programovacích jazyků byl vznik jazyku Pascal (N. Wirth 1971) Je zde kladen důraz na jednoduchost, přehlednost a dobrou strukturovatelnost. Tento jazyk zavedl výčtové typy. Podobné možnosti jako Pascal poskytoval jazyk C (D. Ritchie 1972), který má navíc možnost odděleného překladu a obsahuje prostředky pro nízkoúrovňové programování. SmallTalk (Alan Kay 1973) byl prvním čistě objektovým programovacím jazykem. Vše v něm představuje objekt. Jazyk C++ (B. Stroustrup 1985) byl nejpopulárnějším jazykem 90. let do programování vnesl mj. vícenásobnou dědičnost a prostory jmen. Na principech jazyku C a C++ vyvinula firma Sun programovací jazyk Java, který používá tzv. virtuální stroj. Tím se programy napsané v tomto jazyce stávají nezávislé na platformě. Na podobné myšlence jako Java jsou založeny i jazyky pro platformu .NET z nichž na prvním místě jmenujme C#.

1.2.Procedurální a objektově orientovaný přístup

Procedurální přístup, nazývaný také algoritmický, je přístup používaný například v jazyce Pascal a C. Procedurální program probíhá od začátku do konce, provádí příkaz po příkazu, přičemž příkazem může být i volání podprogramu. Procedurální přístup je úzce spojen se strukturovaným programováním. Pro srovnání s objektově orientovaným přístupem je však podstatné, že u procedurálního programování jsou samostatná data, která zpracováváme samostatnými funkcemi (procedurami). Tyto funkce mají jeden vstup a jeden výstup. V programu se funkce spojují s určitými daty, která jsou však zpravidla globální. Takovéto dělení na data a funkce je však z hlediska reálných objektů umělé, protože v běžném světě existují objekty, které vlastní data i funkce. Tento problém řeší objektově orientované programování (OOP), které je použito například v jazyce C++ a Java. V těchto jazycích se OOP používá k napodobení objektů skutečného světa definováním tříd. V třídě jsou data a funkce zabaleny do jednoho celku. Třída se skládá z datových členů (prvků třídy), k nímž patří atributy a metody. Atributy slouží k uložení dat, která charakterizují daný objekt. Metody popisují chování (funkce a procedury, které vykonávají nějakou činnost s daty) daného objektu.

1.2.1.Objektově orientovaný přístup

Základní charakteristiky objektově orientovaného přístupu

Mezi základní charakteristiky objektově orientovaného přístupu patří:

- používání abstrakce
- definování tříd objektů
- zapouzdření
- dědičnost
- polymorfismus
- znovupoužitelnost

1.2.2.Abstrakce

Abstrakce je proces vytváření jednoduché reprezentace složité reality. Objekty v programování reprezentují objekty, které se nacházejí v reálném světě. Je zřejmé, že je nemožné a nežádoucí udržovat veškeré informace o každém objektu vystupujícím v námi programovaném programu. Při návrhu objektu tedy musíme rozhodnout, které informace jsou důležité a které ne. Kromě toho musíme vybrat i nejlepší způsob jejich reprezentace. Tento proces představuje abstrakci. Abstrakce je náročný proces a je velmi závislý na člověku, který ho provádí. Nelze tedy říci, že existuje jediny správný způsob abstrakce daného objektu.

1.2.3.Třídy objektů

Abychom nemuseli každý objekt programovat znovu, je zaveden pojem třída objektů. Třída objektů je abstrakcí objektů se stejnými vlastnostmi (atributy), stejným chováním (metody) a stejnými vztahy k ostatním objektům. Při vytváření objektů stačí definovat atributy a naprogramovat metody jen jednou v definici třídy a mohou je zdílet všechny objekty této třídy. Třída je jakási šablona, podle které se objekty dané třídy vytvářejí. Pokud vytváříme nový objekt, stačí uvést, do jaké třídy patří, a tím jsou určeny jeho vlastnosti a chování.

1.2.4.Zapouzdření

V objektech jsou data a s nimi pracující procedůry a funkce zapouzdřeny do jednoho celku. Objekt vlastní všechna důležitá data a všechny metody, které realizují chování objektu. Takto navržené objekty lze snadno znovu použít v jiných programech. Zapouzdření je tedy technika, při které jsou data a metody s nimi pracující spojeny do jediné entity. Data objektu jsou skryta před ostatními objekty a lze k nim přistupovat pouze pomocí metod objektu. Zapouzdření je nejdůležitějším principem objektového přístupu.

Hlavním důvodem používání zapouzdření je zamezení neoprávněného přístupu k datům objektu.

1.2.5.Dědičnost

Dědičnost představuje znovupoužitelnost na úrovni deklarace třídy. Použijeme ji tam, kde chceme vytvořit novou třídu, která má podobné vlastnosti jako třída již existující. Třída, od které odvozujeme se nazývá bázová, rodičovská, nadtřída, třída předka. Třída odvozená se nazývá podtřída, dceřinná třída, třída potomka. Odvozená třída obsahuje všechny atributy a metody třídy předka. K těmto datovým členům můžeme v odvozené třídě přidat další atributy a metody. Dále můžeme v této třídě předefinovat některou z metod předka. Při dědění nemůžeme žádný atribut ani metodu třídy předka odstranit. Třída, která vznikla odvozením, může sloužit jako předek jiné třídy.

1.2.6.Polymorfizmus

Podobně jako v reálném světě existují objekty, které se chovají stejně, avšak tohoto chování docílíme různými postupy, je tomu i v objektově orientovaném programování, kdy různým objektům (třídám objektů) definujeme stejnou metodu, která má však jinou implementaci. Jedná se vlastně o určitý druh abstrakce, kde nejprve pohlížíme na daný objekt obecně a až následnou implementací metod s různými argumenty vytvoříme metody pro jednotlivé objekty. V OOP to znamená, že objekty různých tříd mají metodu se stejným jménem, přičemž její implementace se v jednotlivých třídách může lišit. Tato vlastnost se nazývá polymorfismus.

1.2.7.Znovupoužitelnost

Třída kterou vytvoříme, by měla představovat část kódu, kterou můžeme opakovaně použít. Nejjednodušší způsob znovupoužití kódu je vytvoření instance dané třídy. Dalším druhem znovupoužití je vložení objektu do definice jiné třídy. Dále také dědičnost představuje znovupoužití kódu třídy předka v třídě potomka.

Důvody zavádění znovupoužitelnosti:

- úspora práce při programování
- úspora práce při opravách a změnách
- zajištění přehlednosti a srozumitelnosti

1.3.Databáze

1.3.1.Popis možných způsobů zpracování dat a tvorby databází

Za posledních 25 let vývoje počítačů a programování aplikací můžeme vysledovat mohutný trend přechodu od strukturovaného k objektově orientovanému programování. Toto platí i v oblasti zpracování dat a databází. V osmdesátých létech způsobily revoluci relační databáze, v létech devadesátých s mohutným nástupem objektově orientovaného programování začaly vznikat i objektově orientované databáze, které si kladou za cíl ulehčit a zrychlit práci s daty. Ovšem situace není zdaleka tak jednoduchá, protože relační a objektový přístup je od základu rozdílný. Existuje jak mnoho výhod, tak i mnoho nevýhod pro relační i objektové databáze.

1.3.2. Hlavní rozdělení databází

V současné době existují tři základní typy databází – relační databáze (Relational Database Management System, RDBMS), objektově-relační ("Object Relational" Database Management System, ORDBMS) a objektové (Object Database Management System, ODBMS). V následujících odstavcích bude stručně charakterizován, každý

z těchto typů. Vzhledem k zaměření této práce zde budou popsány pouze základní rysy. Popis každého typu databáze bude rozdělen na tři části :

- Datový model
- Dotazovací jazyk
- Výpočetní model (tj. navigace a přístup k datům)

1.3.3.Relační databáze

V 70. letech 20. století probíhal ve firmě IBM výzkum relačních databází. Bylo nutné vytvořit sadu příkazů pro ovládání těchto databází. Vznikl tak jazyk SEQUEL (Structured English Query Language). Cílem bylo vytvořit jazyk, ve kterém by se příkazy tvořily syntakticky co nejblíže přirozenému jazyku (angličtině).

K vývoji jazyka se přidaly další firmy. V r. 1979 uvedla na trh firma Relational Software, Inc. (dnešní Oracle Corporation) svůj relační databázový systém Oracle. IBM uvedla v roce 1981 nový systém SQL/DS a v roce1983 systém DB2. Dalšími systémy byly např. Progress, Infomix a Sybase. Ve všech těchto systémech se používala varianta jazyka SEQUEL, který byl přejmenován na SQL.

Relační databáze byly stále významnější, a bylo nutné jejich jazyk standardizovat. Americký institut ANSI původně chtěl vydat jako standard zcela nový jazyk RDL. SQL se však prosadil jako de facto standard a ANSI založil nový standard na tomto jazyku. Tento standard bývá označován jako SQL-86 podle roku, kdy byl přijat.

V dalších letech se ukázalo, že SQL-86 obsahuje některé nedostatky a naopak v něm nejsou obsaženy některé důležité prvky týkající se hlavně integrity databaze. V roce 1992 byl proto přijat nový standard SQL-92 (někdy se uvádí jen SQL2). Zatím nejnovějším standardem je SQL3 (SQL-99), který reaguje na potřeby nejmodernějších databází s objektovými prvky.

Standardy podporuje prakticky každá relační databáze, ale obykle nejsou implementovány vždy všechny požadavky normy. A naopak, každá z nich obsahuje prvky a konstrukce, které nejsou ve standardech obsaženy. Přenositelnost SQL dotazů mezi jednotlivými databázemi je proto omezená.

Datový model

RDBMS uchovává data v databázi skládající se z řádků a sloupců. Řádek odpovídá záznamu (record, tuple); sloupce odpovídají atributům (polím v záznamu). Každý sloupec má určen datový typ. Datových typů je omezené množství, typicky 6 nebo víc (např. znak, řetězec, datum, číslo...). Každý atribut (pole) záznamu může uchovávat jedinou hodnotu. Vztahy nejsou explicitní, ale spíše plynou z hodnot ve speciálních polích, tzv. cizí klíče (foreign keys) v jedné tabulce, který se rovná hodnotám v jiné tabulce.

Dotazovací jazyk

Pohled (view) je podmnožina databáze, která je výsledkem vyhodnocení dotazu. V RDBMS je pohled tabulka. RDBMS využívá SQL pro definici dat, řízení dat a přístupu a získávání dat. Data jsou získávána na základě hodnoty v určitém poli záznamu.

Výpočetní model

Veškeré zpracováváni je založeno na hodnotách polí záznamů. Záznamy nemají jednotné identifikátory, které jsou neměnné po dobu existence záznamu. Neexistují žádné odkazy z jednoho záznamu na jiný. Vytvoření výsledku je prováděno pod kontrolou kurzoru, který umožňuje uživateli sekvenčně procházet výsledek po jednotlivých záznamech. Totéž platí pro update.

1.3.4.Objektová databáze

Pro objektové databáze neexistuje žádný oficiální standard. Důraz ODBMS je na přímou korespondenci mezi následujícími:

- Objekty a objektové vztahy v aplikaci napsané v OO jazycích
- Uchovávání těchto objektů v databázi

Datový model

Objektové databáze využívají datového modelu, který má objektově orientované aspekty jako třídy s atributy a metodami a integritními omezeními. Poskytují objektové identifikátory (OID) pro každou trvalou instanci třídy. Podporují zapouzdření, násobnou dědičnost a podporují abstraktní datové typy.

Objektové databáze kombinují prvky objektově orientovaného programování s databázovými schopnostmi. Rozšiřují funkčnost objektových programovacích jazyků (C++, Smalltalk, Java) a poskytují plnou schopnost programování databáze. Datový model aplikace a datový model databáze se ve výsledku hodně shodují a výsledný kód se dá mnohem efektivněji udržovat.

Dotazovací jazyk

Objektově orientovaný jazyk (C++, Java, Smalltalk) je jazykem jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty a v dotazech.

Výpočetní model

V RDBMS rozumíme dotazovacím jazykem vytváření, přístup a aktualizaci objektů, ale v ODBMS, ačkoliv je to stále možné, je toto prováděno přímo pomocí objektového jazyka (C++, Java, Smalltalk) využitím jeho vlastní syntaxe. Navíc každý objekt v systému automaticky obdrží identifikátor (OID), který je jednoznačný a neměnný během existence objektu. Objekt může mít buď vlastni OID, nebo může ukazovat na jiný objekt.

1.3.5.Objektově-relační databáze

"Rozšířená relační" a "objektově-relační" jsou synonyma pro databázové systémy, které se snaží sjednotit rysy jak relačních, tak objektových databází. ORDBMS je specifikována v rozšíření SQL standardu — SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

Datový model

ORDBMS využívají datový model tak, že "přidávají objektovost do tabulek". Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu, nazývanou abstraktní datové typy (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. ORDBMS jsou nadmnožinou RDBMS a pokud nevyužijeme žádné objektové rozšíření jsou ekvivalentní SQL2. Proto má omezenou podporu dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem.

Dotazovací jazyk

ORDBMS podporuje rozšířenou verzi SQL — SQL3. Důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod. ORDBMS je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření, pracuje právě s nimi.

Výpočetní model

Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s databází. Příma podpora objektových jazyků stále chybí, což nutí programátory k překladu mezi objekty a tabulkami.

1.3.6.Srovnání relační a objektové databáze z hlediska programování v OOP jazycích

Relační databáze a objektově orientované jazyky

Relační databázové systémy jsou dobré pro řízení velkého množství dat a objektově orientované programovací jazyky ve vyjadřování složitých vztahů mezi objekty. RDBMS jsou vhodné pro vyhledávání dat, ale poskytují nízkou podporu pro manipulaci s nimi. Objektově orientované programovací jazyky jsou výborné při manipulaci s daty, ale poskytují malou nebo žádnou podporu pro neměnnost a vyhledávání dat. Bohužel,

vzhledem k tomu, že tyto přístupy jsou protichůdné, je jejich skloubení poněkud komplikované.

Objektově orientovaný model tedy poskytuje základní vlastnosti objektů což je zapouzdření, dědičnost a polymorfismus. Navíc má každý objekt jednoznačnou identifikaci, která umožňuje používání referencí. Naproti tomu RDBMS poskytují vlastnosti, které OO programovací jazyky nemají, jako např. rychlé vyhledávání, sdílení objektů mezi programy, propracovaný systém oprav chyb pro databázové operace, trvalé uložení, atd.

Existuje několik různých obecných přístupů, jak skloubit relační databázový systém s objektovým programováním. Buď se pokusíme vymodelovat databázi v objektově orientovaném programu, nebo se můžeme pokusit vymodelovat aplikaci v databázi, nebo se můžeme pokusit zjednodušit přístup k databázi tak, že problémy už nebudou tak závažné.

První případ znamená vybudovat OO aplikaci kolem relačního modelu. Musíme ošetřit veškeré manipulace s daty a v každé třídě všechno zvlášť naprogramovat. Tato práce je většinou netriviální problém. Ve druhém případě se snažíme zobrazit objektový model do relační databáze, což je ještě složitější než první případ, protože relační databáze je velice omezená a spoustu věcí v ní nemůžeme provést přímo (např. dědičnost, ukazatele, polymorfismus). Pokud naše aplikace využívá pouze jednoduchých datových typů a nemáme hodně vztahů, můžeme využít třetí způsob, tedy pouze řádkově orientovaný přístup k databázi.

Objektově orientované databáze

Skutečná objektová databáze podporuje všechny vlastnosti nutné k práci s objekty, tedy:

- plná podpora objektů
- zapouzdření
- dědičnost
- polymorfismus
- jednoznačná identifikace objektu
- reference mezi objekty

S objekty se dá pracovat přímo v programovacím jazyku vytvářením a přístupem přes metody. Není tedy nutný žádný mezistupeň pro práci s daty, jako je například SQL.

1.4.JDBC

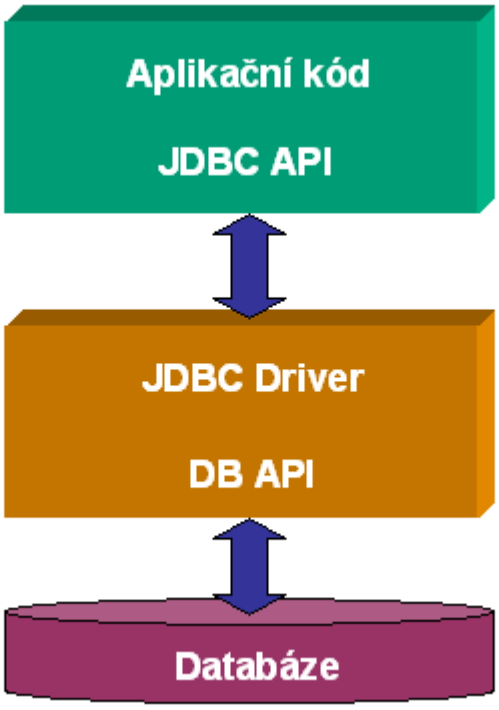
Jelikož jsem praktickou část této práce realizoval v objektově orientovaném jazyce JAVA, budu se v následujícím textu zaměřovat na řešenou problematiku s ohledem na použití tohoto jazyka.

JDBC API poskytuje základní rozhraní pro unifikovaný přístup k databázím. Základem konceptu JDBC je využití funkčnosti poskytované JDBC ovladačem, který je následně překládá do nativních volání dané databáze. Díky tomu je aplikační programátor odstíněn od specifického API databáze a může se naučit jednotné rozhraní JDBC, které pak použije pro přístup do libovolné databáze, která poskytuje JDBC ovladač. V dnešní době to jsou prakticky všechny hlavní systémy a ovladače jsou optimalizované a vyvíjené samotnými výrobci databázových strojů.

JDBC navíc není určeno pouze pro přístup k relačním databázím, ale k libovolnému formátu dat, ukládaného ve „sloupcové podobě“, což mohou být i datové soubory „spreadsheetů“, textové soubory atd.

Z pohledu historie bylo JDBC inspirováno ODBC standardem navrženým firmou Microsoft. ODBC jako takové bylo založené na X/Open CLI specifikaci a bylo primárně přístupné pouze přes C/C++ aplikace, případně přes různé „wrappery“ volání z různých vývojových prostředí (Visual Basic, Powerbuilder atd.). ODBC je tedy čisté C-čkové API, které nemá žádný objektový základ a je poměrně nepřehledné a nestrukturované. I proto firma Microsoft v podstatě tuto technologii opustila a v .NET je přístup k datům a databázím řešen sofistikovaně a nabízí minimálně stejně tak dobré možnosti, jako JDBC.

Kromě jiného vývojáři v Javě mohou funkcionalitu ODBC snadno využívat, protože Sun Microsystems standardně nabízí ve svých JDK (Java Development Kit) vlastní JDBC ovladač určený pro přístup k ODBC. Původně byl tento ovladač vyvinut firmou Intersolve a dodáván samostatně, ale později došlo k jeho začlenění do JDK



Obr. 1: Schéma JDBC z pohledu programátora

Přestože je schéma zjednodušené, naznačuje základní princip JDBC, jak jej vnímá aplikační programátor. Logika JDBC je ale složitější, v závislosti na vlastnostech JDBC ovladače.

1.4.1.Rozdělení JDBC

JDBC specifikace rozpoznává čtyři typy JDBC ovladačů, typ 1 až 4.

Typ 1

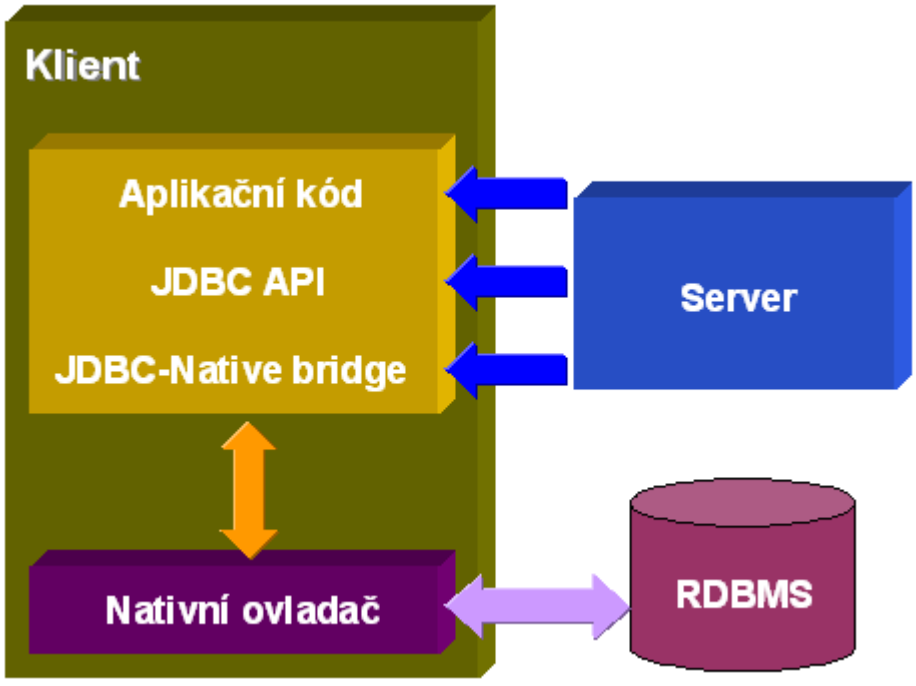
Tento typ ovladače využívá lokální ODBC ovladač a přistupuje k němu přes „JDBC-ODBC bridge“. Taková aplikace vyžaduje nainstalování a nastavení lokálního ODBC ovladače pro danou databázi společně s aplikací v Javě, která tento ODBC ovladač využívá.

ODBC ovladače jsou specifické pro každého výrobce databáze, a proto je práce s nimi složitá, je nutné instalovat lokální DLL knihovny, které musí být synchronizovány s ohledem na aktuální použitou databázi, jejich administrace je časově náročná díky nejednotnosti rozhraní, použití a nastavení atd. Z tohoto důvodu se aplikace využívající JDBC ovladač typ 1 hodí hlavně pro testování v prostředí Windows nebo na

internetové/intranetové aplikace využívající JSP/Servlety. Jejich použití v čistě klientsky orientovaných aplikacích je komplikované právě díky náročnosti na konfiguraci na klientském počítači, hodí se spíše na serverově orientované aplikace, kde konfigurace nemusí být až takový problém.

Typ 2

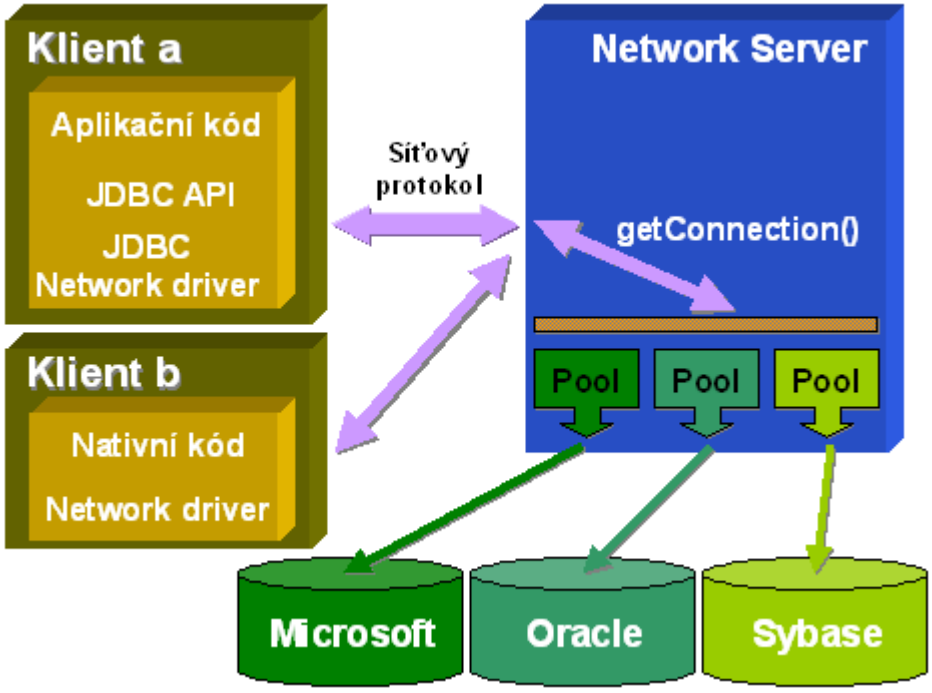
Ovladač typu 2 má za úkol překládat požadavky z JDBC do určitého specifického ovladače, který je v nativní podobě nainstalovaný na počítači a který je určen právě pro jeden typ databáze. Dalo by se říci, že „JDBC-ODBC bridge“ je podmnožinou tohoto typu ovladače, s tím, že je čistě vázán jen na ODBC. Pochopitelně se s tímto typem ovladačů pojí stejně výhod a nevýhod jako v případě JDBC-ODBC. Kromě toho ale mohou nastat ještě větší komplikace při administraci a při nasazení (opět záleží na umístění, zda se jedná o klienta nebo o serverový systém).



Obr. 2: Schéma JDBC typu 2

Typ 3

Tento typ již nepoužívá žádný nativní kód pro ovladač, ale je založen čistě na Javě a JDBC, které konvertuje svoji komunikaci do síťového protokolu, který se spojuje s centrálním serverem (Network Server), který poskytuje připojení k databázi (obvykle s „poolem“ připojení, viz dále). Tento server konvertuje síťový protokol, kterým komunikuje s klienty, do databázově specifického protokolu, jemuž již databáze rozumí. Takový model je vysoce efektivní, a to jak s ohledem na možnost „poolingu“ připojení a tím i zrychlení dotazování a práce s databází, tak i možnost připojení k sadě heterogenních databázových systémů.



Obr. 3: Schéma JDBC typu 3

Architektura tohoto typu se obvykle používá v případě rozsáhlých systémů, kde z historických či „politických“ důvodů mohou být rozdílné databázové produkty a network server v podstatě nabízí unifikované rozhraní. Ten se pak pro aplikační programátory tváří jako jedna databáze a je nutné jej pouze jednou nastavit administrátorem takového serveru a následně provozovat.

Kromě toho, k network serveru se mohou připojovat i jiní než Java klienti, díky tomu, že síťový protokol je platformově nezávislý. Tato architektura tedy poskytuje nejen

lepší možnosti k optimalizaci výkonu („pooling“), spojení heterogenních databází, ale i provázání heterogenních klientských platforem.

Typ 4

Ovladač typu 4 je napsán celý čistě v Javě s plnou podporou pro optimalizace vzhledem k dané databázi. Výhodou tohoto ovladače je, že klient nemusí být jakkoli konfigurován a nejsou nutné žádné lokální klientské instalace ovladačů.

2. Přehled objektově relačních aplikačních rámců

2.1.Jak ukládat data v Javě

V první řadě můžeme použít standardní rozhraní JDBC, které umožňuje přímo přistupovat k prakticky libovolné relační databázi pomocí jazyka SQL. Toto řešení je implementačně nenáročné, nevyžaduje žádné nadstavbové balíky, takže výsledná aplikace může být velmi malá a efektivní. Na druhou stranu se musíme smířit s tím, že si budeme muset sami navrhnout strukturu databáze a implementovat zvlášť ukládání, změnu a zpětné získávání každého kousku dat. V okamžiku, kdy je zapotřebí strukturu dat lehce změnit, je zapotřebí upravit strukturu příslušné tabulky a všechny SQL příkazy s ní pracující.

Java je ovšem jazyk objektově orientovaný a vnitřně jsou proto obvykle všechna data, se kterými pracujeme, reprezentována jako objekty. Z tohoto pohledu tedy vypadá rozumně neukládat jednotlivé datové položky (tedy jednotlivé vlastnosti daných objektů), ale donutit objekty, aby se ukládaly celé. Mějme například třídu Osoba definovanou takto:

```
public class Osoba
{
    protected String jmeno;
    protected String prijmeni;
    protected int vek;

    public Osoba(String jmeno, String prijmeni, int vek)
    {
        ...
    }
}
```

Každá osoba, o které uchováváme informace, je tedy reprezentována jako objekt třídy Osoba. Místo toho, abychom konstruovali SQL příkaz, který uloží jméno, příjmení a věk osoby do relační tabulky (INSERT) a následně obdobné příkazy pro změnu (UPDATE) a zpětné načtení (SELECT) těchto údajů, nabízí se myšlenka uchovat každý vytvořený objekt jako celek. Takové objekty, jejichž existence přetrvá vlastní běh aplikace, se nazývají persistentní objekty.

Toto řešení má mnoho výhod oproti prostému použití relační databáze. Předně objekty, na rozdíl od relací, mají vlastní identitu, odpadají proto starosti s volbou primárního klíče, který by v předchozím příkladě nejspíše musel být generován uměle. A dále, pokud během vývoje aplikace změníme strukturu objektu (například přidáme osobě rodinný stav), nemusíme revidovat všechny SQL příkazy.

Nejelegantnější řešení představuje přímo použití objektové databáze, která obsahuje Javovské aplikační rozhraní, jako například Caché. Naneštěstí objektové databáze dosud zdaleka nedosáhly tak masového rozšíření jako databáze relační a zatímco například velmi kvalitní relační MySQL je běžnou součástí linuxových distribucí a běží na kdejakém (i freehostingovém) serveru, objektovou databázi si obvykle můžeme dopřát pouze pokud máme kam si ji sami nainstalovat, nehledě k tomu, že se v naprosté většině jedná o nákladné komerční řešení. Mnohem reálnější řešení proto představuje použití některého nástroje pro persistenci objektů s využitím relační databáze.

V případě Javy je v tomto směru opět z čeho vybírat. Standardním řešením je použití J2EE a Entity beans, což ovšem vyžaduje podřídit styl psaní celé aplikace tomuto modelu a pro většinu použití je toto řešení příliš robustní. Mnohem jednodušší řešení

poskytují nestandardní nástroje pro persistenci objektů jako Hibernate nebo EJB, které zajišťují persistenci běžných objektů v Javě. Každý z těchto nástrojů má ovšem své vlastní aplikační rozhraní, takže případný přechod k jiné implementaci znamená přepsání větší části aplikace.

2.2.Hibernate

Nástroj Hibernate se používá k zajištění perzistence dat, což je vlastnost umožňující uchovávat data nebo informace i když program skončí. Perzistence dat má několik vlastností. Jednou z nich je současný přístup více uživatelů k datům. Další vlastností perzistence je transakční přístup. Právě transakce umožňují více uživatelům pracovat se stejnými daty nebo programy bez starostí souvisejících s porušením dat. Transakční zpracování se stará o konzistenci dat, která by mohla být narušena vícenásobným přístupem k jedněm datům. Třetí vlastností perzistence je přenositelnost a snadnost použití. Ve většině případů, když se zahájí implementace programu nebo aplikace, často se přenositelnost omezuje pouze na určitý typ specifického prostředí.

S relačními databázemi přichází i potřeba podpory objektově relačního mapování, což je proces překladu objektů na tabulkovou reprezentaci a překlad vazeb a vztahů mezi těmito objekty do dodatečných tabulek. Jedná se o netriviální problém, zvláště v případě použití složitějších objektových modelů.

Hibernate je vysoce výkonný objektově relační nástroj pro perzistenci objektů v aplikaci a správu databázových dotazů nad platformou jazyka Java. Hibernate umožňuje vývoj persistentních objektů v jazyce Java splňující základní vlastnosti jako jsou asociace, dědičnost, polymorfismus, kompozice a práce s kolekcemi objektů v jazyce Java.

Hibernate odstraňuje generování kódu v době překladu systému nebo zpracování bitového kódu. Místo toho jsou použity principy reflexe a generování bitového kódu za běhu programu, kdy generování SQL dotazu se provádí až v době startu systému. Tento přístup zajišťuje, že Hibernate neovlivňuje překlad nebo inkrementální kompilaci.

Hibernate obsahuje i vlastní typový systém, který je mapován na specifické datové typy databázových strojů. Přesto Hibernate není omezen pouze na své základní typy, ale jeho typy mohou být mapovány i na jednoduché typy jazyka Java (například String, Char nebo Float) zahrnující třídy z rámcového systému kolekcí jazyka Java. Tyto typy mohou

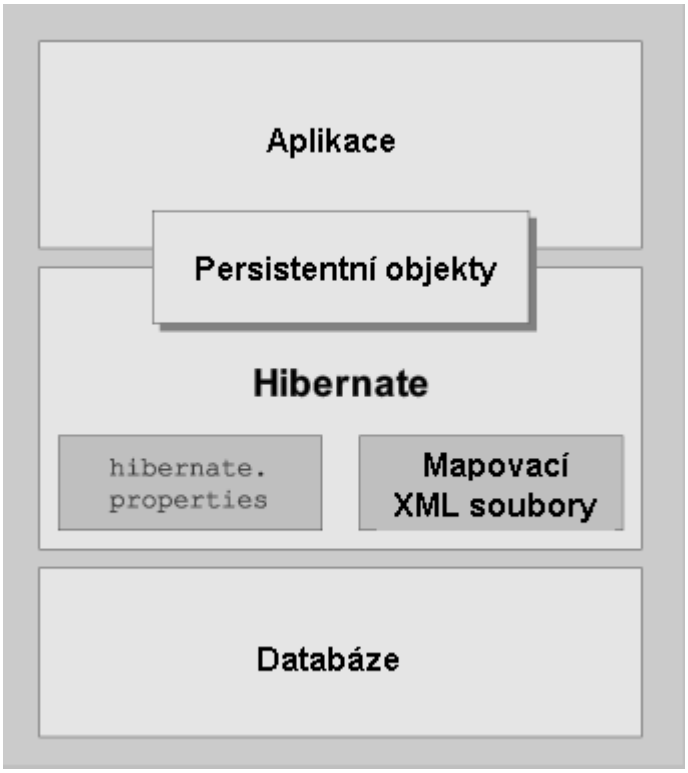
být mapovány jako hodnoty, kolekce hodnot nebo asociace k jiným entitám. Identifikační atribut persistentních objektu (ido) je zvláštním typem, který reprezentuje databázové identifikátory daných tříd. Tyto identifikátory odpovídají v databázovém prostředí primárním klíčům.

Pro zajištění persistence základních tříd nemusí být implementována žádná zvláštní rozhraní ani nemusí být použity speciální mechanismy dědění z kořenových persistentních tříd. Hibernate také nepoužívá žádné zpracování během procesu překladu a sestavování aplikace, plně využívá mechanismu reflexe jazyka Java. To znamená, že základní třídy mohou být mapovány na jednotlivé databázové tabulky bez nutnosti uvádění jakýchkoliv vazeb ve zdrojových kódech daných tříd.

Jediným omezením kladeným na implementaci tříd určených pro persistenci dat, je přítomnost identifikačního atributu. Tento atribut slouží později k odlišení jednotlivých persistentních objektů na základě databázové identity. Hibernate podporuje tento požadavek sadou zabudovaných generátorů identifikátorů pro různé scénáře použití, od implicitních identifikátorů pro databázové sekvence až po implementace různých algoritmů pro generování identifikátorů.

Hibernate, jakožto objektově relační nástroj, je řízen konfiguračními soubory ve formátu XML. V těchto souborech jsou uvedena mapování aplikovaná na jednotlivé tabulky daného databázového schématu na základní třídy. Na základe těchto mapovacích souborů je Hibernate schopen zajistit persistenci objektu. Mapovací soubory lze ale využít i při opačném postupu. Jsou-li nejdříve vytvořeny základní objekty a k nim odpovídající mapovací XML soubory, je možné z těchto souborů vygenerovat odpovídající databázová schémata.

Architektura Hibernate je rozdělena do několika nezávislých sekcí, které implementují některou z jeho funkčních částí. Nejobecnější pohled na architekturu Hibernate ilustruje následující obrázek, znázorňující základní rozdělení a návrh systému, který používá Hibernate pro implementaci persistentní vrstvy. Jak je patrné, aplikace sestává ze série objektů určených k persistenci. Tyto objekty tvoří pomyslné rozhraní mezi aplikační logikou a Hibernatem.



Obr. 4: Architektura Hibernate

Součástí persistentní vrstvy implementované pomocí Hibernate jsou konfigurační a mapovací soubory. Pomocí konfiguračních souborů se řídí přístup k persistenci, definují se zde umístění datových úložišť, přístupy k datovým zdrojům, úroveň izolace a jiné vlastnosti. Mapovací soubory slouží k definování samotné persistence objektu a k jejich provázání se daným databázovým schématem.

Dotazovací jazyk Hibernate (HQL - Hibernate Query Language) byl navržen jako minimální objektově orientované rozšíření základní obecné verze databázového dotazovacího jazyka SQL. HQL tím poskytuje elegantní přemostění mezi objekty a relačními architekturami databázových strojů. Přestože však HQL vychází ze syntaxe dotazovacího jazyka SQL, jedná se o plně objektově orientovaný jazyk, ve smyslu podpory dědění, polymorfismu nebo asociací.

2.3.Java Data Objects

Takzvané 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.

2.3.1.Implementace JDO

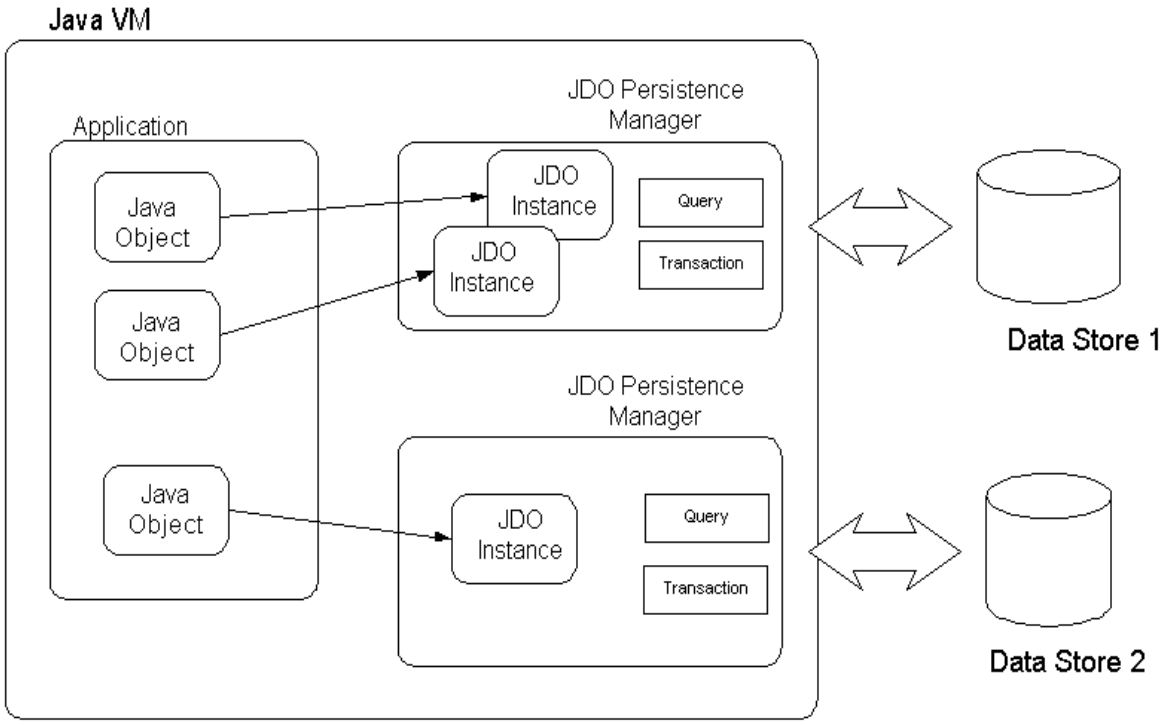
Standard JDO byl navržen za účasti mnoha stran v rámci Java Community Process (JCP). Na návrhu se podíleli mimo jiné výrobci objektových a relačních databází, aplikačních serverů, vývojových prostředí a další. Specifikace JDO se v současné době nachází ve verzi 1.0.1, verze 2.0 slibující mnohá rozšíření.

Firma Sun vydala takzvanou referenční implementaci JDO, která je volně dostupná jak v binární podobě, tak ve zdrojových kódech. Nicméně její funkčnost je značně omezená. Implementuje kompletní aplikační rozhraní JDO, avšak ukládání persistentních objektů je omezeno pouze na lokální soubory (žádná podpora jakéhokoli databázového serveru) a z dalších důležitých vlastností chybí například vyhledávání objektů pomocí dotazů. Od počátku vývoje JDO se totiž předpokládalo, že implementace kompletního JDO bude zcela v rukou třetích stran - ať již výrobců existujících databázových systémů, či ostatních, zcela nezávislých výrobců.

Implementace Sun JDO definuje sice transparentní rozhraní pro komunikaci mezi objekty aplikace a různými typy transakčních datových úložišť (relační x objektová DB), ale hlavní výrobci databází ovlivnili JDO specifikaci tak, že přenositelnost aplikací je výrazně omezena. Např. pro mapovací mechanismus (XML schémata) není definován závazný formát, a tak si jednotliví výrobci databází mohou definovat vlastní formáty. Sun JDO je vlastně standardem pro další JDO implementace ať již od nezávislých výrobců, kteří se

nezabývají databázemi (SolarMetric Kodo JDO, PrismTech OpenFusion JDO), tak výrobců objektových databází (GemStone Systems, IBM Informix, Versant Judo).

Na následujícím obrázku je znázorněna architektura Sun JDO, která aplikaci umožňuje pomocí jednotlivých instancí rozhraní PersistenceManager připojit se k více datovým úložištím.



Obr. 5: Architektura Sun JDO[1]

JDO Instance je instance Java objektu aplikace, které musí vždy implementovat rozhraní PersistenceCapable. Standard dále definuje First-class a Second-class objekty. First-class objekty jsou jako JDO instance jednoznačně identifikovatelné zatímco Second-class objekty jsou přiřazeny First-class objektům. Např. Objednávka představuje First-class objekt, přičemž položka objednávky Second-class objekt.

[1] JDO (Java Data Objects), [online], [cit. 2006 -5 - 1]. URL: <http://java.sun.com/products/jdo/>

Obrázek dále zachycuje tyto hlavní rozhraní specifikované podle Sun JDO :

- `PersistenceManagerFactory`
- `PersistenceManager`
- `PersistenceCapable`
- `Transaction`
- `Query`

`PersistenceManagerFactory` vytváří instance rozhraní `PersistenceManager` a dále umožňuje nastavit chování transakcí či připojení. `PersistenceManager` je rozhraním pro JDO instance a dále zajišťuje řízení transakcí nebo ukládání dočasných dat (cache management). V neposlední řadě vytvoří rozhraní `Query`. Jak již bylo uvedeno výše rozhraní `PersistenceCapable` je vyžadováno u všech tříd JDO instancí. Rozhraní `Transaction` poskytuje metody pro nastavení chování transakcí a u samostatně běžících aplikací zajišťuje úspěšné provedení transakce v databázi. Instance rozhraní `Query` jsou vytvářeny a spouštěny v kontextu instance PersistenceManager a pro dotazy do databáze používá Object Query Language (OQL) , který operuje s Java třídami a objekty místo tabulek (SQL). Tento objektový dotazovací jazyk používá výhradně Java syntaxi.

V současnosti existuje velký výběr dostupných implementací JDO. Ve většině případů se jedná o komerční produkty. Výhodou komerčních implementací je široký výběr nadstavbových nástrojů, rozsálá dokumentace a integrace do běžně používaných vývojových prostředí. Za všechny můžeme jmenovat například produkt JCredo, který je v základní verzi volně dostupný, nebo Kodo, který je také k dispozici pro bezplatné vyzkoušení. Z pohledu seznámení se s JDO jsou mnohem zajímavější volně šířené open-source implementace, kterých začíná být poměrně slušný výběr. Uveďme alespoň stručný přehled:

- Triactive JDO - rychlá a kompaktní implementace sestávající z jednoho JAR balíčku. Drobnou nevýhodou je chybějící podpora některých pokročilých vlastností JDO, pro většinu aplikací je však zcela postačující.
- Speedo - implementuje prakticky celý JDO standard včetně volitelných rozšíření. Na rozdíl od předchozí "odlehčené" implementace se však jedná o rozsáhlý kolos postavený na celé řadě různých knihoven, což způsobuje jistou těžkopádnost.

- JPOX - pravděpodobně nejkvalitnější open-source implementace, která je v současnosti k dispozici. Stabilní verze implementuje celý standard JDO 1.0.1, vývojové verze již implementují značnou část dosud neschváleného JDO 2.0. Vývoj je velice aktivní a vývojáři velmi rychle reagují na veškeré nalezené problémy.
- Jakarta OJB - příspěvek Apache Software Foundation k implementaci JDO. V současnosti existuje ve formě přídavného modulu k referenční implementaci SUNu a nepodporuje JDO v plné šíři.
- XORM - tato implementace poněkud vybočuje z řady. Aplikační rozhraní vychází z JDO, autoři ovšem zavádějí několik koncepčních změn, které způsobují určitou nekompatibilitu s původním JDO.
- Castor JDO - představuje vyspělé, levné neproprietální řešení. Ale omezuje se pouze na relační databáze. Základem Castor JDO je spojení objektového modelu Java aplikace s XML schématem, které poté mapuje data Java objektů do tabulek relační databáze. Castor umožňuje generovat Java třídy podle daného XML schématu a naopak takto vygenerované třídy lze opět převést do XML schématu (tzv. marshalling).

Prostřednictvím standardních JDBC konektorů umožňují všechny uvedené implementace používat k persistenci objektů všechny hlavní databázové servery jako Oracle, MySQL, Sybase, DB2, MS SQL, PostgreSQL a další.

3. Použití perzistentních rámců

V této kapitole se budu věnovat popisu praktické realizace použití perzistentních rámců pro uložení a získání objektu z databáze. Pro názornost uvedu i použití standardu rešícího tuto problematiku v programovacím jazyce Java, což je JDBC. Pro jednoduchost získání jsem použil relační databázi MySQL , která je volně dostupná na internetu.

V rámci této bakalářské práce byla zrealizována aplikace telefonní seznam ve které jsou použity všechny výše uvedené aplikační rámce včetně standardu JDBC. V aplikaci jsou definovány následující třídy: Osoba, Telefon a PrirazeníTelOsoba. Třída PrirazeníTelOsoba představuje vazbu Osoba Telefon, tedy relaci M:N.

V následujících ukázkách kódu bude demonstrováno užití těchto rámců při manipulaci s objektem Osoba.

3.1.JDBC

3.1.1.Ukládání objektu do databáze

```
  public static void ulozOsobuDoDB(Osoba osoba) throws SQLException
  {
    Connection conn = null;
    PreparedStatement ps = null;
    try
    {
      conn = DBPool.getConnection();
        ps = conn.prepareStatement(ulozOsobuDoDbSQL);
        ps.setInt(1, osoba.getIdOsoba());
        ps.setString(2, osoba.getJmeno());
        ps.setString(3, osoba.getPrijmeni());
        ps.executeUpdate();

    }

finally
    {
    if (ps!=null)
      {
        ps.close();
      }
     if (conn!=null)
      {
        conn.close();
      }
    }
  }
```

Z kódu vidíme, že při použití JDBC nejprve musíme získat objekt `Connection`, jenž představuje spojení do databáze. Na tomto spojení je již možno provádět jednotlivé operace na databázi. Voláním metody `PreparedStatement` třídy `Statement`, je příkaz SQL ovladačem JDBC nejprve „kompilován" a teprve pak odeslán databázovému systému. Jako aegument metody `PreparedStatement` použiji konstantu `ulozOsobuDoDbSQL`, která představuje SQL příkaz `INSERT INTO osoba (id_osoba, jmeno, prijmeni) VALUES (1,2,3);` Dále se provede ručně mapování, nastavením atributů `id_osoba, jmeno, prijmeni`. Následným voláním metody `executeUpdate`() dojde k fyzickému uložení objektu do relační tabulky. Nakonec je potřeba ručně uzavřít všechny databázové objekty (`PreparedStatement, Connection`) metodou `close()`. Opomenutí tohoto uzavření bývá zdrojem častých chyb proto je třeba toto uzavření důsledně provádět.

3.1.2.Získávání objektů z databáze

```
public static List vyberOsobyZDB() throws SQLException
{
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    List seznamLidi = new ArrayList();
    try
    {
      conn = DBPool.getConnection();
      ps = conn.prepareStatement(vyberOsobyZDbSQL);
      rs = ps.executeQuery();
      while (rs.next())
      {
        Osoba osoba = new Osoba();
        osoba.setIdOsoba(rs.getInt(1));
        osoba.setJmeno(rs.getString(2));
        osoba.setPrijmeni(rs.getString(3));
```

```
            seznamLidi.add(osoba);
          }
        }
        finally
        {
          if (rs!=null)
          {
            rs.close();
          }
          if (ps!=null)
          {
            ps.close();
          }
          if (conn!=null)
          {
            conn.close();
          }
        }
        return seznamLidi;
    }
```

Z kódu je patrné, že opět musíme stejně jako u ukládání objektu nedříve získat objekt `Connection` a `PreparedStatement`. K těmto dvěma objektům je ještě potřeba získat objekt `ResultSet`. Instance typu `ResultSet` jsou výsledkem spuštění metody `executeQuery()`. Třída `ResultSet` je zapouzdřením dat vrácených výběrovým dotazem `SELECT * FROM osoba ORDER BY prijmeni, jmeno`, který je reprezentován konstantou `vyberOsobyZDbSQL`. Třída ResultSet umožňuje zpracovávat data po řádcích (záznamech). Předtím, než můžeme záznam zpracovat , musíme se na tento záznam přesunout metodou `next()`. Po vytvoření instance `ResultSet` je kurzor nastaven před první řádek. Dále zde zapisujeme do atributů objektu `osoba (idOsoba, jmeno, prijmeni)` data získaná z instance typu `ResultSet`. Takto získané objekty jsou ukládány do proměnné `seznamLidi`, která je proměnná typu `List` a je výstupní proměnnou metody `vyberOsobyZDB`(). Nakonec je potřeba ručně uzavřít všechny databázové objekty (`PreparedStatement, Connection, ResultSet`) metodou `close()`.

3.2.JDO

3.2.1.Mapování

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="cz.tul.kai.votocek.bakalarka.dto">
        <class name="Osoba" identity-type="datastore" >
            <inheritance strategy="new-table"/>
            <field name="jmeno" persistence-modifier="persistent">
                <column length="45" jdbc-type="VARCHAR"/>
            </field>
            <field name="prijmeni" persistence-modifier="persistent">
                <column length="45" jdbc-type="VARCHAR"/>
            </field>
            <field name="idOsoba" persistence-modifier="persistent"/>
        </class>

    </package>
</jdo>
```

Mapování javovských tříd na databázové tabulky v JDO je řešeno pomocí XML souboru. Nevýhodou je, že nelze provést mapování na již existující databázi nebo si vytvořit vlastní databázové schéma, neboť aplikační rámec JDO si databázové schéma generuje sám.

3.2.2.Ukládání objektů do databáze

```
public Osoba ulozOsobu(Osoba osoba)
  {
    Transaction tx = pm.currentTransaction();
    try
    {
      tx.begin();
      pm.makePersistent(osoba);

      tx.commit();
    }
```

```
    finally
    {
      if (tx.isActive())
      {
        tx.rollback();
      }
    }
    return osoba;
  }
```

Ukládání objektů pomocí JDO probíhá prostřednictvím rozhraní `Transaction`, které poskytuje metody pro nastavení chování transakcí a u samostatně běžících aplikací zajišťuje úspěšné provedení transakce v databázi. Proměnná `pm` představuje objekt `PersistenceManager`, který se bude starat o persistenci jednotlivých datových objektů. Tento objekt je vytvořen pomocí `PersistenceManagerFactory` na základě předchozí konfigurace. Transakce začíná voláním metody `begin()` a během jejího trvání je možno vytvářet persistentní objekty a měnit jejich vlastnosti. V okamžiku ukončení transakce voláním `commit()` se aktuální stav persistentních objektů uloží do databáze. V našem případě v rámci transakce pouze učiníme z objektu osoba persistentní objekt pomocí volání metody `makePersistent()`, čímž zajistíme, že v okamžiku volání `commit()` se aktuální stav objektu uloží do databáze. Pokud by během transakce došlo k nějaké chybě dojde pomocí metody `rollback()` k navrácení všech objektů do stavu před začátkem transakce.

3.2.3.Získávání objektů z databáze

```
public List vyberOsobyZDB()
  {
    Transaction tx=pm.currentTransaction();
    List list = new ArrayList();
    try
    {
      tx.begin();

      Extent e = pm.getExtent(Osoba.class,true);
      Query q = pm.newQuery(e);
      q.setOrdering("prijmeni ascending");
```

```
      Collection c = (Collection)q.execute();
      Iterator iter = c.iterator();
      while (iter.hasNext())
      {
        Osoba osoba = (Osoba)iter.next();
        list.add(osoba);
      }

      tx.commit();
    }
    finally
    {
      if (tx.isActive())
      {
        tx.rollback();
      }
    }
    return list;
  }
```

Jako ukládání, tak i čtení objektů z databáze pomocí JDO probíhá prostřednictvím rozhraní `Transaction`. Význam metod `begin ()`, `commit ()`, `rollback ()` a proměnné `pm` je stejný jako v předchozí ukázce kódu pro ukládání objektů. Objekt `e` je tzv. extent třídy, což je množina všech instancí dané třídy uložených v databázi. Metoda `getExtent ()` má dva parametry z nichž první je objekt typu Class reprezentující nějakou perzistentní třídu a druhý parametr typu boolean určuje, zda se do extentu zahrnou i objekty odvozených tříd či nikoli. Do objektu c se uloží kolekce objektů, jenž splňují určitá kritéria, která jsou definována dotazovacím jazykem JDOQL (JDO Query Language). V našem případě dojde k výběru všech objektů třídy Osoba a od ní odvozených tříd. Tyto objekty budou seřazeny abecedně. Takto získané objekty jsou následně postupně uloženy do objektu `list`, který je typu `List`, což je seznam objektů. Objekt `list` je výstupní proměnnou metody `vyberOsobyZDB()`. Výhodou JDO je, že po volání metody commit() a rollback() dojde k automatickému uzavření objektů `PreparedStatement, Connection, ResultSet.`

3.3.Hibernate

3.3.1.Mapování

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="cz.tul.kai.votocek.bakalarka.dto">
  <class name="Osoba" table="osoba">
    <id name="idOsoba" column="id_osoba" type="integer">
       <generator class="increment"/>
    </id>
    <property name="jmeno" type="string" />
    <property name="prijmeni" type="string" />
  </class>
</hibernate-mapping>
```

Mapování javových tříd na databázové tabulky se provádí pomocí XML souboru umístěného v adresáři s mapovanými třídami. Výhodou Hibernate je, že se můžeme namapovat na již existující relační tabulku. Těchto XML souborů může být i více a jsou zaznamenány v konfiguračním souboru `hibernate.cfg.xml`. Tyto soubory můžeme psát ručně, nebo použít některý z nástrojů, které je generují automaticky. Jména těchto souborů musí být tvaru `jmenoTridy.hbm.xml`. V našem případě soubor `Osoba.hbm.xml` bude umístěn v podadresáři, ke kterému máme nastavenu CLASSPATH. V každé mapované třídě musí být deklarovaný primární klíč elementem `<id name>`. Každá vlastnost této třídy musí být zachycena elementem `<tproperty>`, který ji mapuje na příslušný atribut databázové tabulky. Atribut type pak určuje typ této vlastnosti. Hibernate podporuje všechny základní javové datové typy, ale hodnotou tohoto atributu může být i javový objekt. Pokud není typ zadán, Hibernate se ho pokusí určit sám.

3.3.2.Ukládání objektů do databáze

```
public Osoba ulozOsobu(Osoba osoba)
{

  Session session = HibernateUtil.currentSession();
  Transaction tx = session.beginTransaction();
  session.save(osoba);
  tx.commit();
  HibernateUtil.closeSession();
  return osoba;
}
```

Použití Hibernate v naší aplikaci je velmi jednoduché. Nejdříve musíme vytvořit instanci `SessionFactory`, která bude konfigurována podle vlastností zanesených v `hibernate.cfg.xml`, tato operace se provádí při inicializaci třídy `HibernateUtil`. Následuje vytvoření `Session` jako pracovní jednotky Hibernate, vytvoření transakce `Transaction`, její spuštění metodou `beginTransaction()`. Pomocí metody `save()` k uložíme objekt do relační tabulky. Následuje potvrzení transakce metodou `comit()` a uzavření pracovní jednotky metodou `closeSession()`. Pokud nastane během transakce k nějaké vyjímce použijeme metodu `rollback ()` pro vrácení všech objektů do stavu před začátkem transakce.

3.3.3.Získávání objektů z databáze

```
public List vyberOsobyZDB()
{
       Session session = HibernateUtil.currentSession();
       Query query = session.createQuery(SELECT_OSOBY);
       return query.list();
}

SELECT_OSOBY = "from Osoba order by prijmeni";
```

Při získávání objektů z databáze používáme speciální odnož SQL nazvanou HQL, jenž se liší především v tom, že místo názvu tabulek se používají názvy DTO (Data Transfer Object) objektů. DTO jsou objekty které obsahují bezparametrový konstruktor a "getter/setter" metody pro každou jejich vlastnost. Hibernate automaticky ukládá získaná data do proměnné typu List. Stejně jako je tomu v JDO dochází v Hibernate po zavolání metody `comit()` a `rollback()` k automatickému uzavření objektů `Connection, ResultSet, PreparedStatement.`

Závěr

Účelem této práce bylo seznámit se s problematikou perzistentního ukládání objektů do relačních databází. Tato problematika je velice aktuální, jelikož stále více aplikací je nuceno spravovat stále větší množství dat. V dnešní době se při vývoji aplikací nejčastěji používá objektově orientovaný přístup, zatímco data jsou ukládána v relačních databázích. Mezi objektově a relačním přístupem však existují značné rozdíly. Z tohoto důvodu je snaha oddělit vlastní logiku aplikace od konkrétního způsobu uložení dat. Proto vznikají specializované frameworky a nástroje, které umožňují mapování dat mezi objektově orientovaným a relačním světem.

V první kapitole této práce jsem se snažil o stručné zmapování vývoje programovacích jazyků od používání strojového kódu k objektově orientovanému programování. Dále jsem charakterizoval hlavní rysy procedurálního a objektového přístupu k programování a provedl jejich vzájemné porovnaní. V další části této kapitoly jsem popsal stávající a nastupující trendy v oblasti zpracovávání dat databázemi. V současné době existují tři základní typy databází – relační, objektově-relační a objektové. Jelikož je tato práce orientovaná na problematiku ukládání objektů do relační databáze a praktickou část jsem realizoval v objektově orientovaném jazyce Java, jsou poslední odstavce této kapitoly věnovány stndardu rešícího tuto problematiku. Tímto standardem je JDBC.

V následující kapitole jsem se zaměřil na teoretické zvládnutí dvou objektově relačních aplikačních rámců, Hibernate a JDO. Hibernate je vysoce výkonný objektově relační nástroj pro perzistenci objektů v aplikaci a správu databázových dotazů nad platformou jazyka Java. JDO je standardem, který přináší abstraktní vrstvu mezi objekty Java aplikací a uložištěm persistentních dat, které je dnes ve valné většině představováno relační databází.

V třetí části jsem se zabýval praktickou realizací použití perzistentních rámců pro uložení a získání objektu z databáze. Pro názorné předvedení smyslu používání výše uvedených aplikačních rámců je zde použit i standard rešící tuto problematiku v programovacím jazyce Java, což je JDBC.

Z praktické implementace je patrné, že používání standardu JDBC v aplikaci není obtížné avšak je pracné, jelikož programátor musí v kódu sám provést mapování objektů na databázi. Též je nutné ručně ošetřovat otevírání a uzavírání databázových objektů sloužících k spojení a k přenosu dat mezi aplikací a databází. Při používání aplikačních

rámců JDO a Hibernate se provádí mapování pomocí XML souboru, který je uložen v adresáři spolu s mapovanými třídami. Spojení z databází u JDO je realizováno prostřednictvím rozhraní Transaction, které poskytuje metody pro nastavení chování transakcí a zajištění úspěšného provedení transakce v databázi. U aplikačního rámce Hibernate je toto řízeno rozhraním Session. Rozdíl mezi aplikačním rámcem Hibernate a JDO je v tom, že u Hibernate je možné provést mapování na již existující databázové schéma kdežto u JDO je databázové schéma generováno samo. S ohledem na pracnost použití a na skutečnost možnosti mapování na již existující databázové schéma bych ze tří mnou vyzkoušených rámců doporučil k užívaní aplikační rámec Hibernate.

Seznam použitých zdrojů:

[1] BAUER Ch., KING G.: *Hibernate in Action*, 408 p., Manning Publicatios (August 1, 2004)

[2] JORDAN D., RUSSELL C.: *Java Data Objects*, 380 p., O'Reilly Media, Inc., 1 edition (April 22, 2003)

[3] SPELL B.: *Java Programujeme profesionálně*, 1022 s.,Computer Press (2002)

[4] VIRIUS M.: *Java pro zelenáče*, 240 s., Neocortex s. r. o., (2001)

[5] KEOGH J., GIANNINI M.: *OOP bez předchozích znalostí Průvodce pro samouky*, 222 s., Computer Press a.s. (2006)

[6] CHAPMAN J. S.: *Začínáme programovat v jazyce JAVA*, 308 s., Computer Press (2003)

[7] ECKEL B.: *Myslíme v jazyku JAVA*, 469 s., Grada Publishing (2000)

[8] HEROUT P.: *Učebnice jazyka Java*, 349 s., Koop nakladatelství (2003)

[9] PECINOVSKÝ R., VIRIUS M.: *Objektové programování 1*, 228 s., Grada Publishing (1996)

[10] ČERNÝ V.: *Slovník počítačových zkratek*, 174 s., Koop nakladatelství (1998)

[11] VAUGHN R. W.: *Visual Basic pro SQL Server*, 962 s., Computer Press (1998)

[12] RIORDAN R. M.: *Vytváříme relační databázové aplikace*, 280 s., Computer Press (2000)

[13] GRUBER M.: *Mistrovství v SQL*, 494 s., Softpress (2004)

[14] http://www.skolaekonom.cz/informace/HR/vyt/cgi/ucebnice/ferda/TUTOR/History.htm

[15] http://www.linuxsoft.cz/article.php?id_article=244 - obecny popis Javy

[16] http://nb.vse.cz/~zelenyj/it380/eseje/xkroj12/JDBC.htm

[17] http://dev.mysql.com/doc/refman/5.0/en/connector-j.html

[18] http://www.mysql.com/products/connector/j/

[19] http://nb.vse.cz/~zelenyj/it380/eseje/xjirf01/xjirf01.htm - hibernate a orm

[20] http://linux456.vsb.cz/~las034/jni/index.htmlnativni interface

[21] http://www.jantichy.cz/diplomka/pozadavky/databaze - odbc

[22] http://www.skolaekonom.cz/informace/hr/vyt/cgi/ucebnice/ferda/TUTOR/lekce4.htm

[23] http://www.nwt.cz/medical/tech.pdf

[24] http://java-2.navajo.cz/

[25] http://databaze.navajo.cz/

[26] http://interval.cz/clanky/jdo-java-data-objects/

[27] http://nb.vse.cz/~zelenyj/it380/eseje/xklio04/jdo.htm
[28] http://nb.vse.cz/~zelenyj/it380/eseje/vse.htm