VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

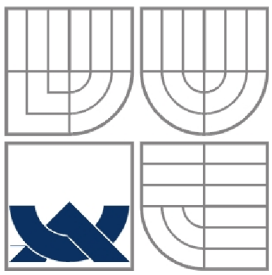
# PŘÍSTUP K OBJEKTOVÝM DATŮM DATABÁZE ORACLE 10G Z JAZYKA JAVA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

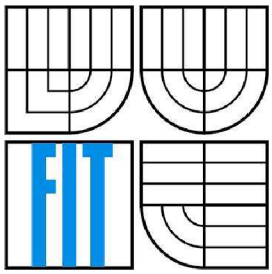
AUTOR PRÁCE
AUTHOR

Bc. MICHAL NOVÁK

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# PŘÍSTUP K OBJEKTOVÝM DATŮM DATABÁZE ORACLE 10G Z JAZYKA JAVA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL NOVÁK
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. JAROSLAV ZENDULKA, CSc.
SUPERVISOR

BRNO 2008

# Abstrakt

Práce pojednává o objektových rozšířeních databázového systému Oracle 10g a přístupu k nim z programového prostředí Javy.



# Klíčová slova

databázový systém, relační technologie, objektově-relační technologie, třída, objekt, objektová tabulka, objektový typ, odkaz, kolekce, Oracle, Java, SQL, SQLJ, JPublisher.



# Abstract

This diploma thesis deals with object extensions of Oracle database 10g system and describes access from Java environment.



# Keywords

database system, relational technology, object- relational technology, class, object, object table, object type, reference, collection, Oracle, Java, SQL, SQLJ, JPublisher.

# Přístup k objektovým datům databáze Oracle 10g z jazyka Java

## Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně pod vedením doc. Ing. Jaroslava Zendulky, CSc. Uvedl jsem literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Novák
14. 5. 2008

## Poděkování

Chtěl bych poděkovat vedoucímu mé práce Jaroslavu Zendulkovi za odbornou pomoc při řešení projektu.

# Obsah

# 1 Úvod

V relačních databázových systémech se čím dál více prosazuje integrace objektové technologie, jako alternativní, sofistikovanější způsob přistupovaní k datům. Výsledkem této evoluce jsou objektově-relační databázové systémy. Jedním z poskytovatelů takového systému je i společnost Oracle. Tím se dostávám k obsahu této práce. Ta si bere za cíl, analýzu objektově-relační technologie, kde se konkrétně soustředí na databázový systém Oracle 10g a jeho vlastnosti.

Na začátku se zabývám vlastnostmi a vývojem databázových modelů, které postupem času vyústily v objektově-relační model. Následně popisuji objektová rozšíření definovaná v SQL3 standardu. Poté se již zaměřují přímo na databázový server Oracle 10g. Popisuji jeho objektová rozšíření a možnosti. Zkoumám možné přístupy k této technologii z programového prostředí Javy. Zaměřuji se na integraci Javy v databázovém serveru. Dále zkoumám možná mapování objektových typů na aplikační třídy.

Poslední částí je návrh a implementace ukázkové aplikace, která má za cíl analyzované objektové vlastnosti prezentovat. Postupně popisuji specifikaci aplikace, databázový model, aplikační model a nakonec stručně prezentuji funkční moduly implementované aplikace. Specifikace obsahuje především diagram případů užití a jeho popis. Popis databázového modelu je jedna ze stěžejních věcí této práce. V rámci popisu uvádím diagram objektového schématu uloženého v databázi a postupně popisuji návrh a způsob implementace jednotlivých objektových technologií. Těmi jsou například odkazy nebo kolekce. Po databázovém modelu se zaměřuji na ten aplikační. Uvádím zvolené mapovací technologie mezi aplikačním a databázovým modelem. Uvádím aplikační třídy, spravující perzistentní objekty. Poté stručně popisuji grafické rozhraní a jednotlivé moduly aplikace. V samotném závěru popisuji možná alternativní řešení, která lze použít.

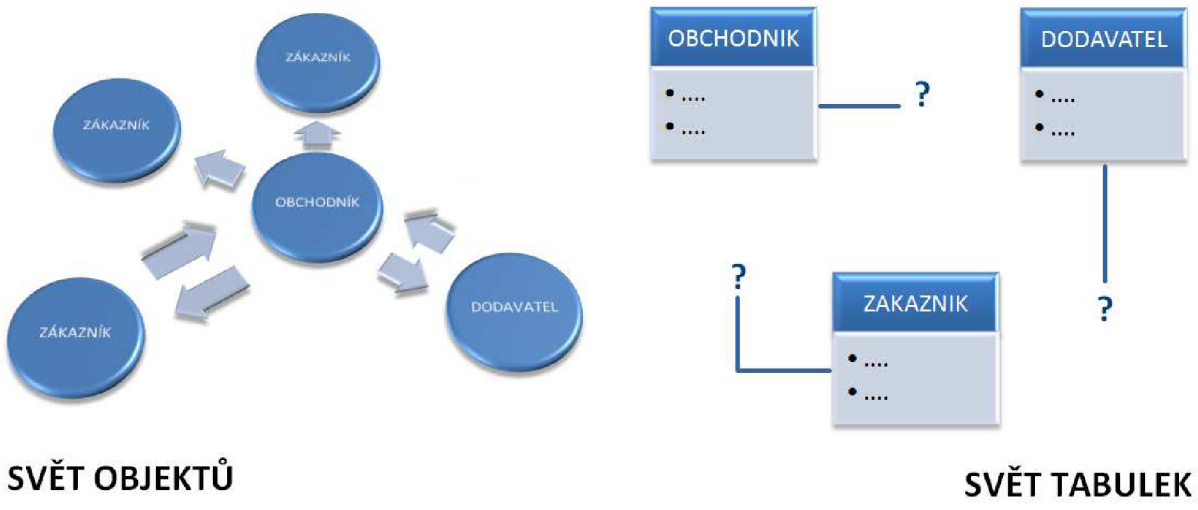
# 2 Databázové systémy

## 2.1 Úvod

Ve světě informačních technologií hrají důležitou roli databázové systémy. Ty se od sedmdesátých let vyvíjely různými směry, přičemž dnes jednoznačně převažují řešení založené na relační technologii. Tato technologie sebou ovšem časem přináší různá úskalí. Především v době, kdy je veškerý návrh business aplikací orientován objektově, kdy se snažíme dát datům smysl, tedy modelovat data jak je chápeme v reálném světě, se relační model vůbec nehodí. Data reálného objektu jsou v útržcích uložena v tabulkách bez implicitních vazeb. Takový přístup opravdu nemá s objektovým návrhem mnoho společného.

Tento neduh relační technologie se začal v devadesátých letech řešit. Přístupy byly dva. První, opravdu revoluční přístup hodlá vyměnit relační model za čistě objektový a vystavět na něm nové databáze. I vzhledem k dobré standardizaci, velké rozšířenosti a rozsahu investovaných financí do relační technologie se tento přístup neujal.

Druhý přístup se dá označit za evoluci. Vychází z poznatku, že relační technologie tu prostě je a do budoucna se s ní musí počítat. Může se ovšem rozšířit o objektové vlastnosti. Výsledkem je objektově-relační databázový systém. Toto řešení se dočkalo mnohem většího úspěchu než předchozí přístup a dnes všichni velcí hráči na poli databázových technologií tyto objektové prvky do svých relačních databází implementují.




**obr. 1 – objekty a tabulky**

## 2.2 Historický vývoj ve zpracování dat

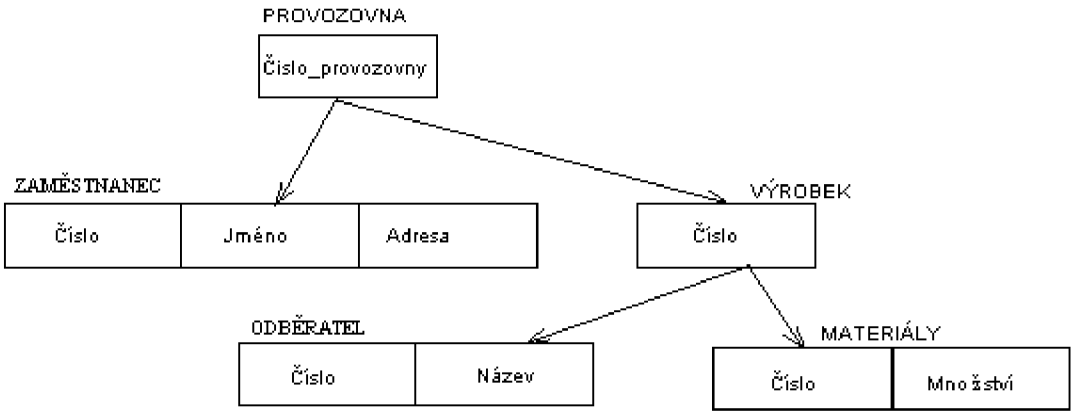
V padesátých letech bylo zpracování dat organizováno systémem typu vše v jednom. Aplikace pracovaly odděleně, každá se svými daty. Jediná aplikace v sobě zahrnovala komunikaci s uživatelem, popis dat, programovou část i samotná data.

V šedesátých letech byl vytvořen systém pro zpracování souborů. Dochází k přesunu samotných dat z aplikace do souborů. Data byla oddělena od uživatelské aplikace, avšak byla izolována v jednotlivých souborech. Tento fakt znamenal nebezpečí redundance a nekonzistence dat. Další nevýhodou byl problém přístupu k datům, kdy změna v systému vedla k přepracování aplikace (přistupovaly přímo k datům).

V sedmdesátých letech nastupuje databázová technologie (SŘBD). Základním přínosem této technologie bylo dosažení určité nezávislosti dat na uživatelských aplikacích a naopak. Data jsou udržována jednotně. Snižuje se redundance a zvyšuje konzistence dat.

### 2.2.1 Hierarchické systémy

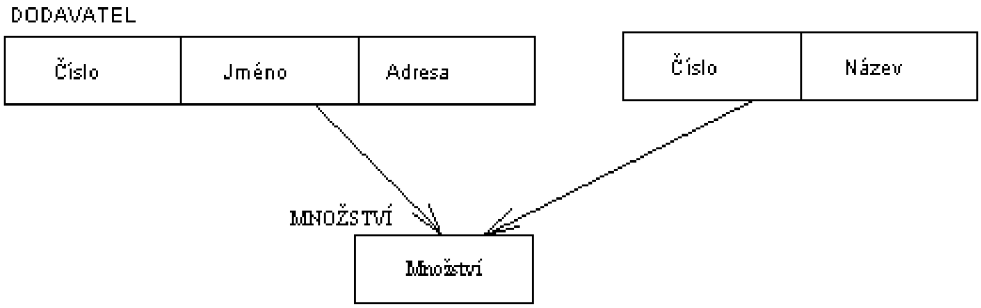
Jako první nastupují na scénu hierarchické systémy. Data jsou organizována jako stromová struktura, ve které každý záznam představuje uzel. Model obsahuje vzájemné vztahy typu rodič-potomek. Takový přístup je vhodný pro použití v doméně, která má sama o sobě hierarchickou strukturu. Vyhledávání dat vyžaduje navigaci přes záznamy směrem dolů (potomek), nahoru (rodič) a do strany (další potomek). Mezi nevýhody modelu patří nepřirozená organizace dat (tedy pokud nejsou sama data hierarchicky uspořádána). Obtížně lze znázornit vztah M:N. Složité operace vkládání a rušení záznamů. Pro hierarchický model nebyl ustanoven standard. Příkladem takového systému je IMS (Information Management System).[1]




**obr. 2 – hierarchický model**

### 2.2.2 Síťové systémy

Síťový model dat vychází z hierarchického modelu a rozšiřuje jej o mnohonásobné vztahy označované jako „C-množiny“ nebo „Sety“. Sety propojují záznamy a to i různého typu. Spojení může být realizováno pro jeden nebo více záznamů. Přístup k propojeným záznamům je přímý bez dalšího vyhledávání. Nevýhodou modelu je obtížná změna jeho struktury. Příkladem takového systému je IDMS (Integrated Database Management System).[1]




**obr. 3 - síťový model**

### 2.2.3 Relační systémy

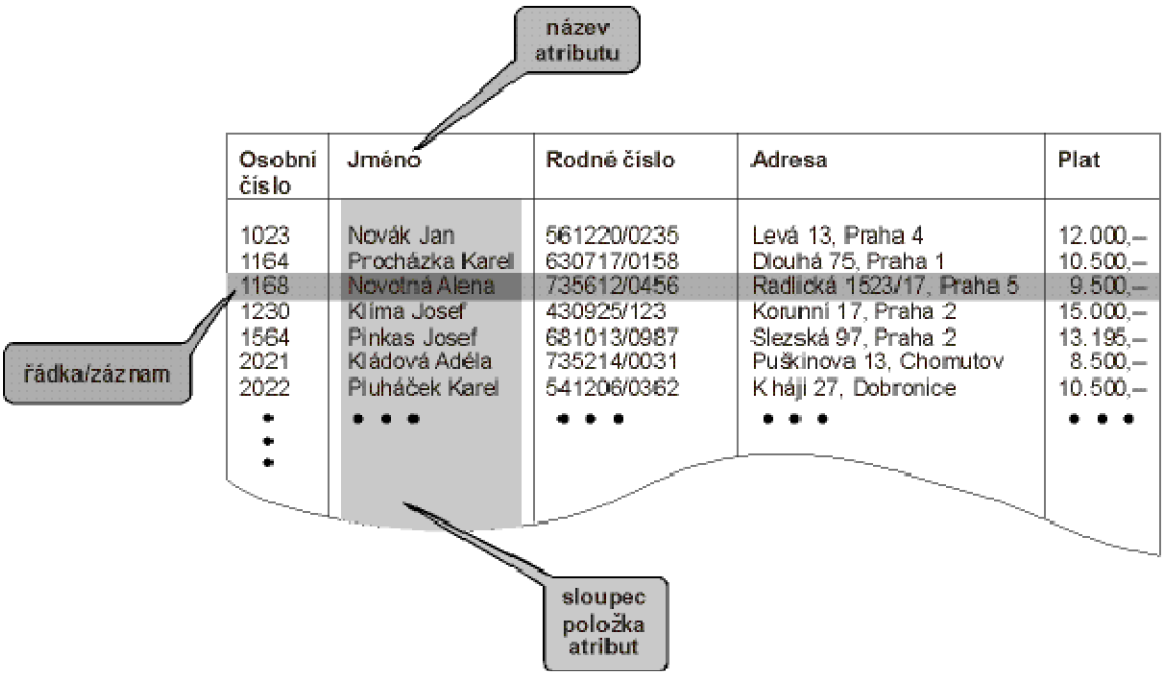
Relační model dat byl vyvinut panem E.F. Coddem v sedmdesátých letech. Roku 1970 byl relační model publikován v odborném časopise Communications of ACM. Šlo o přístup, který na jedné straně vrací data do izolovaných souborů na fyzickém disku, na druhé straně samotný pohled na data je povýšen na konceptuální (logickou) úroveň. Uživatel vidí data strukturovaná v tabulkách, jejichž sloupce jsou v určitém vztahu ke sloupcům jiných tabulek a má k dispozici silný nástroj pro manipulaci s nimi. Tyto úvahy byly na míle vzdáleny od tehdejšího hierarchického uspořádání souborového systému a zahájily revoluci ve tvorbě a používání databází. Postupně dochází k implementaci jazyka SQL. První relační systémy jsou vzhledem ke konkurenčním architekturám síťových a hierarchickým systémů z hlediska výkonu neefektivní. Postupem let se však relační databáze ke konkurenci dotahují.[2]

#### *Relační model dat*

Základním pojmem relačního modelu je relace. Relace je v databázi reprezentována tabulkou. Tabulka se skládá ze sloupců a řádků. Sloupce tabulky nazýváme atributy a jsou reprezentací vlastností modelované entity. Atributy plní funkci integritních omezení pro vkládané hodnoty, které tvoří řádky tabulky a jsou odrazem reality vnějšího světa.

Tedy tabulka je základním stavebním prvkem databáze, relace odpovídá tabulce a prvek relace je konkrétní řádek tabulky. Jeden řádek bývá nazýván databázovým záznamem. Soubor relací pak tvoří databázi (schéma databáze). [5]

Jeden ze sloupců se značí jako primární, takový sloupec obsahuje primární klíč, ten musí být unikátní v rámci tabulky a odkazujeme se jím na ostatní sloupce daného řádku. Databázové systémy umožňují definovat jako primární klíč i n-tici položek.




**obr. 4 - tabulka relační databáze**

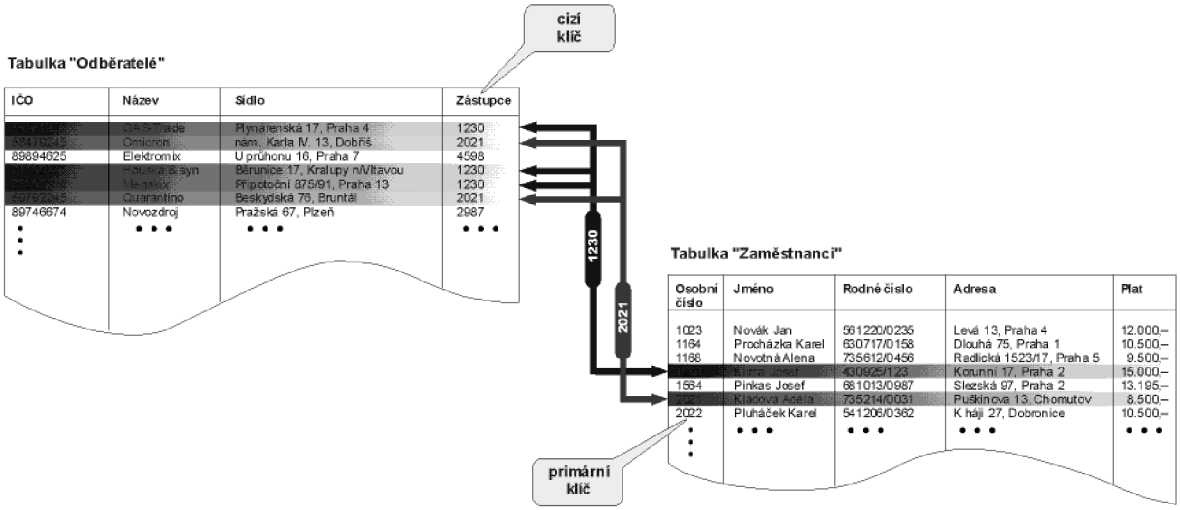
### Vztahy

V relační databázi rozlišujeme tři typy vztahů 1:1, 1:N a M:N.

**Vztah 1:1** je sdružení dvou tabulek, u něhož hodnota primárního klíče každého záznamu primární tabulky odpovídá hodnotě shodného pole nebo polí právě jednoho záznamu související tabulky.

**Vztah 1:N** je sdružení dvou tabulek, u něhož hodnota primárního klíče každého záznamu primární tabulky odpovídá hodnotě shodného pole nebo polí mnoha záznamů související tabulky.

**Vztahem M:N** nazýváme sdružení dvou tabulek, u něhož může každý záznam kterékoliv z obou tabulek souviset s mnoha záznamy zbývající tabulky. Tato relace může způsobovat problémy. Jedním z nich je možnost vzniku přebytečných dat, které porušují pravidla normalizace. Pro realizaci takového vztahu pouze dvě tabulky nestačí a musíme si pomoci tabulkou třetí. Vztah tak převedeme na dva vztahy typu 1:N, kde původní dvě tabulky plní roli primární tabulky k tabulce pomocné.




**obr. 5 - ukázka vztahu 1 : N**

### 2.2.4 Objektově-orientované systémy

V 80.tých letech se objevují nové aplikace náročné na zpracování dat, pro které tradiční SŘBD založené na relačním modelu dat nejsou adekvátní. Takovými nově vznikajícími systémy byly především designové a výrobní systémy (CAD), vědecké a medicínské databáze, geografické informační systémy a multimediální databáze. Tyto aplikace měly požadavky neslučující se s tradičními požadavky na relační databáze. Požadavky na vysoce strukturovaná data, multimediální data, aplikačně-specifické operace, dlouhé transakce apod.

V 90.tých letech pronikají mezi databázové technologie prvky objektově-orientovaných jazyků. Principem je ukládat objekty do databáze a současně využít užitečné prvky objektové technologie. Tyto Objektově-orientované databázové systémy (OOSŘBD) byly navrženy, aby vyhověly požadavkům výše zmíněných aplikací. Takové systémy jsou díky svému objektovému návrhu velmi pružné, jelikož nejsme limitováni datovými typy (obecně nestrukturovanými), ani dotazovacím jazykem (SQL), který je typický pro relační model. Tím základním a zásadním rysem OOSŘBD systémů je schopnost specifikace složitých objektů a tvorba vlastních operací pro manipulaci s těmito objekty. Vzniká tak ekvivalence mezi objekty, se kterými pracujeme v aplikačním programu a jejich reprezentací uloženou v databázi. OOSŘBD tedy do klasické databázové technologie, která má za hlavní úkol správu dat, integrují objektově-orientovaný přístup, který je známy z programovacích jazyků (C++).

Základním principem objektového přístupu v programovacích jazycích je uvažovat program jako kolekci nezávislých objektů, sdružovaných do tříd, které mezi sebou komunikují pomocí zpráv, které si zasílají. V programovacích jazycích objekty existují pouze v době běhu programu. V OOSŘBD, naproti tomu objekty mohou být vytvořeny jako perzistentní (a také se tomu standardně děje) a mohou být sdíleny více programy.[3]

### 2.2.5 Objektově-relační systémy

Objektově-relační databázové systémy (ORSŘBD) jsou zatím posledním vývojovým článkem na poli databázových systémů. Objektově-relační technologie je praktickým kompromisem mezi relační a objektovou databází. Bere si za cíl stavět na dosud nejvíce rozšířených a podporovaných relačních systémech a zakomponovat do nich výhody objektového přístupu. Poskytují alternativní přístup k používání objektů spolu s databázovým systémem. Zatímco pro OOSŘBD byl zvolen revoluční přístup, ve kterém se integrovaly objektově-orientované programovací jazyky do databázového systému, ORSŘBD zvolily cestu evoluce, kde integrovaly objekty do relačního modelu dat a rozšířily stávající relační systémy o objektově-orientované vlastnosti. Přidávají rozšiřitelnost, zapouzdření, polymorfismus či dědičnost.

Pro objektově-orientované modelování je charakteristická především bohatost typů. Za pomoci těchto typů, se snadno modelují a následně implementují objekty požívané v podnikových systémech. Takové objekty mývají složitou strukturu a složité vztahy. Relační databáze umožňují modelovat tyto objekty jednoduše, ovšem za cenu mnohdy neefektivního přístupu k odpovídajícím datům. Z tohoto hlediska mají objektově-relační systémy pro vývoj budoucích aplikací velký význam.

#### *Historické ohlédnutí*

První představa o objektově-relačních databázích se datuje do roku 1990, kdy byl vydán manifest „Third Generation Database System Manifesto“ jako reakce na manifest o objektově-orientovaných databázích. Základním požadavkem byla kompatibilita „třetí generace“ databázových systémů s čistě relačnímy systémy (označovány jako „druhá generace“). Ve stejném roce byl vydán objektově-relační SŘBD UniSQL, který používal databázový jazyk SQL/X, což bylo rozšíření jazyka SQL-92. V roce 1996 došlo k dohodě o podobě vlastností objektově-relačního modelu dat a následně se objevil standard SQL-3, který tyto vlastnosti specifikoval. Standard SQL-3 byl následně uvažován při implementaci všemi velkými dodavateli databázových systémů (Informix, Sybase, IBM, Oracle).

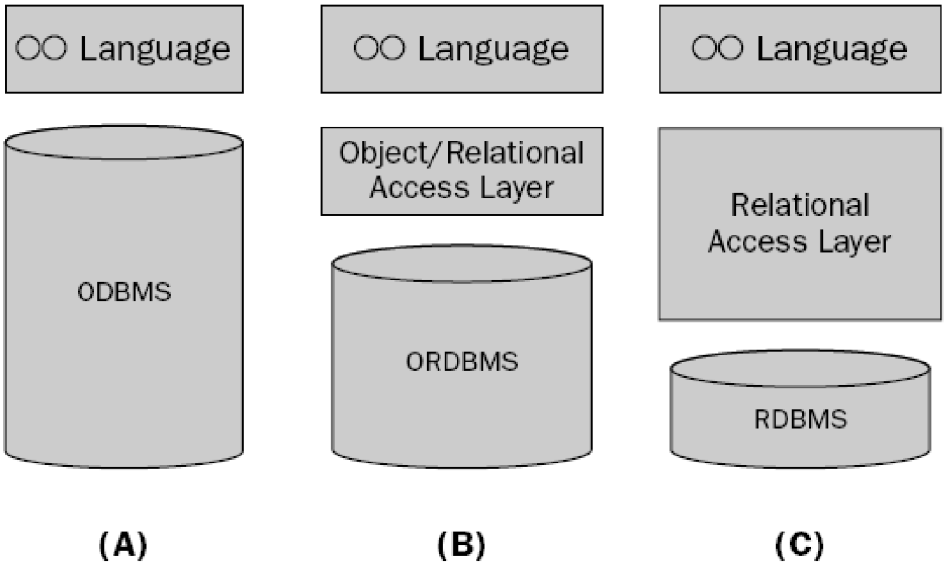
#### *Spojeni relační a objektové technologie*

Relační systémy nabízí možnost práce s jednoduchými daty uloženými v tabulkách a relativně silnými dotazovacími prostředky, které reprezentuje standard SQL-89(SQL1), SQL-92(SQL2) a SQL-99(SQL3). Relační technologie velice efektivně poskytuje souběžné zpracování požadavků v různých paralelních architekturách.

ORSŘBD přidávají možnost ukládat objekty do relační databáze. Jedná se o důležitý a hlavně potřebný krok, kdy výrobci relačních databází rozšířili možnost aplikace do oblastí požadujících integraci klasických tabulkových dat a objektů speciálních typů, např. časové řady, prostorová data a binární objekty, mezi které patří audio, video, obrázky nebo aplety. Zapouzdřením metod může u

datových struktur ORSŘBD volat složité operace nad těmito daty (transformace, prohledávání,..). V tom je velká výhoda oproti relačním SŘBD, které nabízejí pouze jednoduché typy.

Vlastnost rozšiřitelnost znamená dát možnost definování nových, vlastních strukturovaných datových typů, ale také funkcí pro efektivní vyhledávání dat v souladu s jejich vnitřní strukturou. Rozšiřitelnost je logickým důsledkem objektového přístupu. „Objektovost“ by měla u objektově-relační technologie zahrnovat z datového modelu minimálně hlavní rysy objektového standardu ODMG-93. Další nutností je odpovídající objektový jazyk vyšší úrovně. ODMG-93 nabízí jazyk OQL, který je podobný SQL-92. Jako vhodnější se však ukázal standard SQL3, který poskytuje vlastní řešení objektového rozšíření relační koncepce. Výsledkem jsou systémy, které podporují práci jak se strukturálně složitými objekty, tak s relačními datovými strukturami obsahující multimediální data.[2]

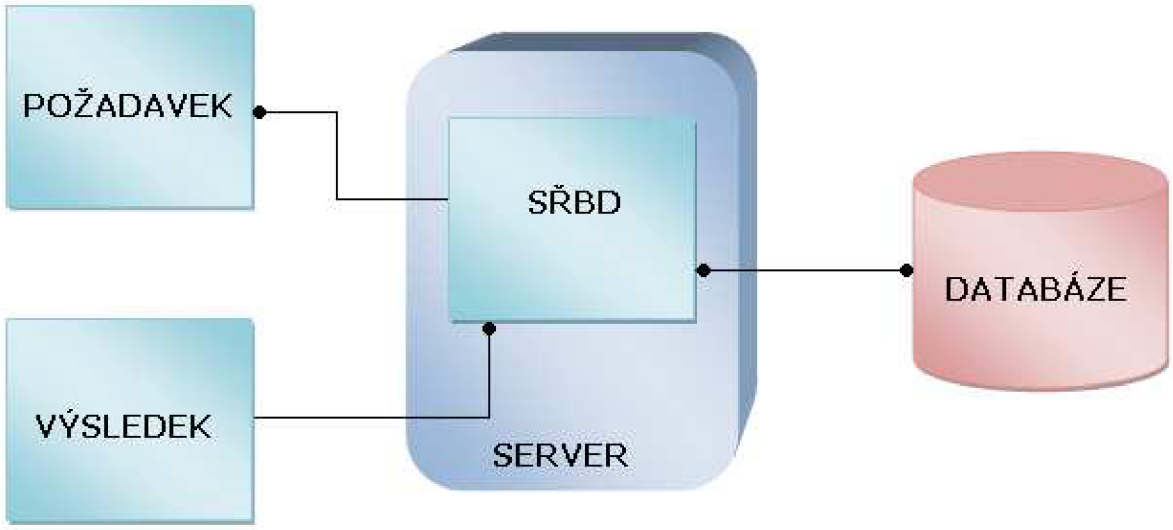



**obr. 6 – pohled na odlišný přístup, převzato z[5]**

# 3 SQL

## 3.1 Úvod

Jazyk SQL (Structured Query Language) je nástroj pro organizování, správu a získávání dat uložených v relační databázi. Pro získání dat z databáze vytvoříme dotaz v tomto jazyce, odešleme ho SŘBD kde je zpracován, proveden a je vrácen výsledek.




**obr. 7 - příklad průběhu dotazování**

Jazyk SQL ovšem představuje více než pouhý nástroj pro dotazování. Lze jej použít k řízení všech funkcí, které databázový systém uživateli poskytuje jako je definice dat, manipulace s daty, řízení přístupu (omezení práv) nebo řízení integrity dat. Samotný jazyk SQL je tedy integrovanou součástí relačního databázového systému, nástroj pro komunikaci s ním. Pokud bychom chtěli srovnávat SQL s tradičními programovacími jazyky jako je C, C++ nebo Java tak je nutné říci, že SQL není v pravdě úplným jazykem, chybí mu například příkazy pro řízení běhu programu (jako je cyklus FOR). Příkazy v SQL vypadají jako jednoduché anglické věty, a proto se jazyk snadno učí a chápe. Částečně je to dáno faktem, že příkazy popisují data, která se mají získat, ale nespecifikují, jak se mají získat. Jazyk SQL tedy říká, co chce, ale vůbec už neříká jak toho dosáhnout, to je práce pro samotné SŘBD.[1]

## 3.2 Historie

Jazyk SQL byl vyvinut pracovníky společnosti IBM v polovině sedmdesátých let. Postupně se stal standardem pro přístup k datům uložených v relačních databázích. První standard byl přijat v roce 1986, roku 1989 byl upraven (SQL-89 nebo také SQL1). Časem se však projevily v současném standardu nedostatky. Nová verze z roku 1992 (SQL-92/SQL2) je standardem v oblasti relačních databází dodnes. Roku 1999 byl vydán nový standard (SQL-99/SQL3), který předchozí standard rozšiřuje především o podporu objektově-relačních rysů.[1]

## 3.3 Podpora objektové technologie (SQL3)

Jak jsem zmínil v podkapitole historie, standard SQL3 definuje podporu pro objektově-orientované vlastnosti. Standard specifikuje nové příkazy, klauzule a výrazy pro tyto oblasti:[1]

- Uživatelsky definované datové typy
- Složené (abstraktní) datové typy
- Kolekce a odkazové typy
- Přetěžované (polymorfní) uložené procedury
- Řádkové a tabulkové konstruktory podporující abstraktní typy
- Podpora pro velké typy

### 3.3.1 Uživatelsky definované datové typy

Základní myšlenkou objektového rozšíření je umožnit uživateli definovat nové vlastní typy, které mohou být používány jako typy základní. Uživatelem definovaný abstraktní datový typ (ADT) zapouzdřuje atributy a operace do jedné entity. V SQL3 je ADT definován specifikací množiny atributů, které reprezentují hodnotu ADT. Dále operacemi, které definují rovnost, nebo vztah uspořádání ADT a operacemi, jež definují chování ADT. Operace jsou implementovány pomocí procedur zvaných rutiny. ADT může být definován jako podtyp jiného ADT. Takový typ dědí strukturu a chování svého rodiče (je podporována vícenásobná dědičnost). Instance ADT mohou být perzistentně uloženy pouze ve sloupcích tabulek v databázi.

#### *Řádkový typ*

Je posloupnost dvojic jméno položky/datový typ podobná definici tabulky. Řádkový typ umožňuje reprezentovat typ řádku v tabulkách, takže celý řádek může být přiřazen do proměnné, předán jako parametr nebo vrácen jako návratová hodnota. Pojmenovaný řádkový typ je uživatelem definovaný řádkový typ s přiřazeným jménem. Pomocí pojmenovaného řádkového typu lze definovat typ reference. Hodnota typu reference definované pro určitý řádkový typ je jedinečná hodnota a

identifikuje instanci typu v tabulce. Hodnota odkazu může být uložena v tabulce a použita jako ukazatel na specifický řádek v jiné tabulce. Stejný odkaz může být uložen v několika řádcích, takže je daný řádkový typ sdílen těmito řádky.

#### *Typ kolekce*

SQL3 definuje parametrizované typy kolekcí. Kolekce může být specifikována jako SET(<typ>), MULTISET(<typ>), nebo LIST(<typ>). Parametr <typ> může být předdefinovaným typem, ADT, řádkovým typem, nebo další kolekcí. Například SET(LIST(ADRESA)), kde adresa je ADT. Parametrem nemůže být odkazový typ, ale může jím být pojmenovaný řádkový typ, který obsahuje pole, jehož typ je odkazový typ. Kolekce může být v dotazech použita jako tabulka.

### 3.3.2 Specifikace chování objektů

Operace na objektech, které mohou být vyvolány v SQL, zahrnují předdefinované operace na tabulkách (SELECT, INSERT, UPDATE, DELETE), implicitně definované funkce pro atributy ADT a rutiny explicitně asociované s ADT nebo definované samostatně. Rutiny asociované s ADT jsou definice funkcí, které specifikují operace na ADT a vrací jedinou hodnotu definovaného datového typu. Funkce mohou být SQL funkce definované v rámci schématu nebo externí funkce definované ve standardních programovacích jazycích.

**Rutiny**

SQL rutiny jsou v podstatě podprogramy. Rutina může být buď funkce, nebo procedura, která čte nebo mění složky instance ADT. Rutina je určena svým jménem, svými parametry a pokud je funkce, tak i klauzulí RETURNS a svým tělem. Parametr se skládá ze jména, datového typu a specifikace IN, OUT, INOUT. SQL rutiny mají tělo napsané kompletně v SQL. Proto bylo nutno do SQL přidat několik výrazových typů, aby se dalo chování ADT specifikovat pouze pomocí SQL.

Rutiny (procedury a funkce) definují chování ADT. Mohou být zapouzdřeny v definici ADT (takové rutiny mají přístup k soukromým atributům ADT), nebo mohou být definovány mimo definici ADT. Některé rutiny mají předdefinovaná jména, příkladem je konstruktor, který je automaticky definován pro tvorbu nových instancí daného typu. Konstruktor má stejné jméno jako typ a nepřebírá žádné argumenty. Vrací instanci daného typu a atributy jsou nastaveny na implicitní hodnoty. Pro každý atribut jsou také generovány funkce pro čtení a změnu jeho hodnoty. K porovnání instancí ADT mohou být definovány typově specifické funkce EQUAL a LESS THAN. Uspořádání instance ADT specifikují funkce RELATIVE a HASH. Funkce CAST umožňuje konverzi mezi různými ADT.

**Stavy**

SQL podporuje stav ve formě hodnot různých datových typů. Stav instance ADT je posloupnost hodnot složek daného ADT. Stav řádku je posloupnost hodnot jeho sloupců apod.

**Události**

Trigger je pojmenovaná databázová konstrukce, která je implicitně aktivována kdykoli nastane daná událost. Pokud je trigger spuštěn, provede se daná akce při splnění podmínky.

```
CREATE TRIGGER update_balance
BEFORE INSER ON account_history   /*událost*/
REFERENCING NEW AS ta
FOR EACH ROW
WHEN (ta.TA_type = 'w')          /*podmínka*/
UPDATE accounts
SET balance = balance-ta.amount /*akce*/
WHERE account_# = ta.account_#
```

Triggery mohou být použity k mnoha účelům. Události pro triggery jsou vkládání, mazání a změna tabulek a sloupců. Trigger může být aktivován před (BEFORE) nebo po (AFTER) dané události. Podmínka i akce se mohou odkazovat na staré i nové hodnoty ovlivněných řádků. Mohou být provedeny pro každý ovlivněný řádek (FOR EACH ROW), nebo jednou pro celý výraz (FOR EACH STATEMENT).[4]

### 3.3.3 Zapouzdření

Každá vlastnost (atribut/funkce) ADT má úroveň zapouzdření PUBLIC, PRIVATE nebo PROTECTED. Veřejné komponenty vytváří rozhraní ADT a jsou viditelné všem autorizovaným uživatelům ADT. Prvky typu PROTECTED jsou částečně zapouzdřeny. Jsou viditelné jak v definici ADT, tak i v definici jeho podtypů.[4]

### 3.3.4 Polymorfismus

Různé rutiny mohou mít stejné jméno. Tato technika se nazývá přetížení (overloading) a umožňuje podtypu ADT předefinovat operace zděděné ze svého rodiče. SQL3 implementuje tzv. obecný objektový model, jenž pro rozhodnutí, kterou rutinu vyvolat, používá všechny typy argumentů rutiny. Pravidla pro toto rozhodnutí mohou být poměrně složitá. Instance rutiny, která je vybrána, se nejlépe shoduje v typech aktuálních parametrů při volání.[4]

### 3.3.5 Dědičnost a delegace

ADT může být definován jako podtyp jednoho nebo více ADT pomocí klauzule UNDER (je podporována i vícenásobná dědičnost). V takovém případě se ADT nazývá přímý podtyp svého rodiče a rodič je přímý nadtyp daného ADT. Typ může mít více než jeden podtyp nebo nadtyp. Podtypy dědí všechny atributy a chování svého nadtypu. Může také definovat své vlastní atributy a chování. Instance podtypu je považována za instanci všech svých nadtypů. Lze ji použít kdekoli, kde je vyžadována instance jakéhokoli jejího nadtypu.

Každá instance je sdružená s nejspecifičtějším typem, který koresponduje s nejnižším podtypem přiřazeným instanci. V jakémkoli okamžiku musí mít instance právě jeden nejspecifičtější typ. Tento typ však nemusí být listový v typové hierarchii.

Definice podtypu má přístup k reprezentaci všech svých přímých nadtypů, ale nemá přístup k reprezentaci svých potomků. Všechny reprezentace přímých nadtypů jsou zkopírovány se stejným jménem a datovým typem do reprezentace podtypu. Aby se podtyp vyhnul kolizi jmen, může přejmenovat vybrané části zděděné prezentace. Podtyp může definovat operace jako kterýkoli jiný ADT. Může také definovat operace, které mají stejné jméno jako operace jiných typů včetně nadtypu. Tabulka může být deklarována jako podtabulka, jedné nebo více nadtabulek, použitím klauzule UNDER v definici tabulky.

Princip tvorby podtabulek je úplně nezávislý na principu definice podtypů. Podtabulka dědí všechny sloupce ze své nadtabulky a může definovat i své vlastní. Tabulka, která není podtabulkou žádné jiné tabulky, se nazývá nejvyšší (kořenová) nadtabulka. Ta spolu se svými podtabulkami (přímými i nepřímými) tvoří tzv. rodinu podtabulek. Každá rodina musí mít právě jednu kořenovou nadtabulku. Řádek z podtabulky musí korespondovat právě s jedním řádkem v každé z jeho přímých nadtabulek. Řádek z nadtabulky odpovídá nejvýše jednomu řádku z přímých podtabulek. Pravidla pro SQL výrazy INSERT, DELETE a UPDATE jsou definovány tak, aby zajišťovala konzistenci řádků tabulek z jedné rodiny podtabulek podle výše zmíněných pravidel.

- Je-li vložen řádek do podtabulky T, potom je příslušný řádek (se stejným identifikátorem a stejnými hodnotami) vložen do každé nadtabulky tabulky T, což se opakuje směrem nahoru v hierarchii tabulek. Je-li T kořenovou nadtabulkou, vloží se řádek pouze do ní.
- Je-li řádek v nadtabulce změněn, potom jsou všechny zděděné řádky v přímých i nepřímých podtabulkách patřičně změněny.
- Je-li změněn řádek v podtabulce, je změněn každý korespondující řádek tak, aby hodnoty ve sloupcích odpovídaly aktualizovaným hodnotám.
- Je-li řádek smazán z tabulky v rodině podtabulek, jsou také smazány všechny korespondující řádky.[4][1]

### 3.3.6 Podpora pro velké typy

Pro podporu velkých objektů byly definovány typy BLOB (Binary Large Object) a CLOB (Charakter Large Object). Instance těchto typů jsou uloženy přímo v databázi místo toho, aby byly udržovány v externích souborech.[1]

# 4 Objektová rozšíření v Oracle 10g

## 4.1 Úvod

Oracle 10g je relační databáze, která podporuje objekty. Tato vlastnost má za cíl pomoci programátorům databáze v řešení nesourodosti při návrhu modelu aplikace a schématu databáze. Dát možnost modelování složitých objektů reálného světa i v prostředí samotné databáze. Objekty uvnitř databáze můžeme použít v rámci PL/SQL či Java uložených procedur. Jako mechanismus pro uložení dat v objektových tabulkách. Jako nadstavba (objektový pohled) nad existujícími relačními tabulkami.

Objekty lze vytvářet (resp. skládat) ze všech zabudovaných typů, které databáze dává k dispozici a dříve definovaných nových typů objektů, referencí a typů kolekcí. Metadata pro definované typy jsou uložena ve schéma, které je dostupné pro SQL, PL/SQL, Javu a ostatní vydaná rozhranní.

Objektové typy a kolekce poskytují možnosti jak na vysoké úrovni organizovat a zpřístupňovat data v databázi. Pod objektovou vrstvou jsou data stále uložená v tabulkách, ale máme možnost s těmito daty pracovat jako s objekty reálného světa. Tedy namísto toho, abychom pracovali s jednotlivými sloupci tabulek, pracujeme jednoduše s objektem jako celkem (např. zákazníkem). Můžeme se rozhodnout, vytvořit složitější entity jako nově vytvořené uživatelské datové typy a ukládat je ve sloupcích relačních tabulek, jako jsme to dělali doposud s jednoduchými typy. Můžeme vytvořit objektové pohledy existujících relačních tabulek a data prezentovat skrz objektový model. Můžeme přímo ukládat objektová data do objektových tabulek, kde každý řádek tabulky je objektem.

Objektový model je podobný mechanismu tříd jazyků C++ a Java. Podobně jako třídy, objektové typy dělají modelování reálného světa byznys logiky lehčím a srozumitelnějším. Opětovná využitelnost těchto objektových typů dělá vytváření databázových aplikací rychlejší a efektivnější. Nativní podporou objektů v databázi umožňuje Oracle vývojářům aplikací přímo přistupovat k datovým strukturám používaných v jejich aplikacích. Není vyžadována mapovací mezi-vrstva mezi objekty na straně klienta a daty v objektových tabulkách databáze. Objektová abstrakce a zapouzdření přispívají u aplikace k jejímu jednoduššímu pochopení a správě.[6]

**Seznam hlavních výhod, které objekty poskytují nad čistě relačním řešením:**

- Objekty mohou zapouzdřit operace spolu s daty – tabulky databáze mohou obsahovat pouze data. Objekty mohou přidat operace, které jsou potřebné pro práci s danými daty (např. operace, která nám vrátí historii objednávek zákazníka).

- Objekty jsou účelné – máme-li vytvořené objekty s metodami, pro které je předpoklad znovuvyužitelnosti, pak vývojáři nemusí znovu vytvářet podobné struktury, ale využít tyto a pouze je volat.
- Objekty mohou reprezentovat částečné vztahy (pistons) - k takové reprezentaci v relační databázi bychom potřebovali vytvořit násobné tabulky s primárními a cizími klíči. Objekt poskytuje lepší řešení, může mít jiné objekty jako své atributy a takovéto „objekty atributy" mohou mít zase jako atribut jiné objekty. Na tomto principu může být vystavěna hierarchie z takto propletených objektů.[6]

# 4.2 Objektové vlastnosti

## 4.2.1 Objektový typ

Objektový typ je uživatelem definovaný strukturovaný typ, který lze použít stejným způsobem jako základní datové typy jako je NUMBER. Hodnota objektového typu je jeho instancí – objektem. Následující příklad ukazuje deklaraci takového typu i s metodami.

```
CREATE TYPE person_typ AS OBJECT (
 idno          NUMBER,
 first_name    VARCHAR2(20),
 last_name     VARCHAR2(25),
 email         VARCHAR2(25),
 phone         VARCHAR2(20),
 MAP MEMBER FUNCTION get_idno RETURN NUMBER,
 MEMBER PROCEDURE display_details ( SELF IN OUT NOCOPY person_typ ));
```

Objekty jsou v Oracle podobně jako v klasických objektově-orientovaných programovacích jazycích tvořeny z částí, kterým říkáme atributy a metody. Atributy obsahují data o vlastnostech objektu. Metody jsou funkce vztažené k danému objektu, resp. jeho atributům a definují chování objektu. Další příklad ukazuje použití objektového typu jako atributu (sloupce) tabulky:

```
CREATE TABLE contacts (
 contact        person_typ,
 contact_date   DATE );
INSERT INTO contacts VALUES (
 person_typ (65, 'Verna', 'Mills', 'vmills@oracle.com', '1-800-555-4412'),  '24 Jun 2003' );
```
[6][5]

### *Řádkové a sloupcové objekty*

Objekty uložené v řádcích objektových tabulek nazýváme „řádkové objekty“. Objekty, které jsou uloženy jako sloupce „klasické“ relační tabulky, nebo jsou atributy pro jiné objekty, nazýváme „sloupcové objekty“.

### *Kompozitní objekty*

Za kompozitní objektový typ označme takový typ, který má za úkol reprezentovat modelovaný vztah agregace a kompozice. V rámci sama sebe tedy obsahuje jiný objekt, který v jeho rámci existuje, nebo kolekci.

### *Metody*

Metody jsou buď procedury, nebo funkce, které jsou asociovány s daným objektem. Dále objekt obsahuje konstruktor. Metody můžeme implementovat v prostředí PL/SQL, Java a ostatních podporovaných jazycích a zpřístupnit je z venčí databáze pomocí externích volání. S objektem mohou být asociovány 4 různé typy metod.

**Členské metody**

Jsou preferovanou cestou jak přistupovat k atributům objektu a manipulovat s nimi. Metody mají přístup k parametru „self“, který je obdobou „this“ v programovacím jazyku Java a zastupuje danou instanci. Příklad volání metody objektu:

*SELECT c.contact.get_idno() FROM contacts c;*

**Konstruktory**

Obdobně jako konstruktory objektově-orientovaných jazyků, vrací metoda novou nenulovou instanci objektového typu. Oracle poskytuje ke každému typu defaultní konstruktor, který obsahuje jako vstup jednotlivé atributy objektu. Následuje ukázka kódu v PL/SQL:

*DECLARE*
*newCustomer customer_objtyp := NULL;*
*BEGIN*
*newCustomer := customer_objtyp(42,‘Adams, Douglas’, Address_objtyp(‘123 Handy Road’, ‘Dandyville’, ‘CA’, 94999), phonelist_vartyp(‘+1-650-555-1212’, ‘+61-8-9555-1212’));*
*END;*
[6][5]

**Porovnávací metody**

Skalární data mají předefinované uspořádání, které se použije při pokusu porovnat nebo setřídit jejich instance. Co se týká objektů, pak Oracle poskytuje dva principy porovnání. Jsou jimi Mapovací metody a řadící metody. Objektový typ může definovat vždy pouze jednu z nich. Deklarací takové metody umožníme porovnávat v SQL příkazech instance daných objektů. Pokud není definována žádná z metod, pak lze objekt porovnávat pouze na ekvivalenci. Mapovací metody (Map metoda) je bez-parametrová metoda začínající klíčovým slovem „map“. Objekty budou podle takové metody porovnávány podle skalárního typu, který funkce vrací, nejobvykleji se jedná o primární klíč.
Příklad takové metody:

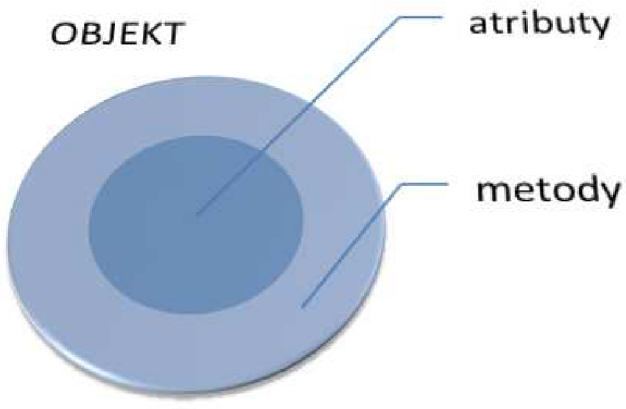
```
MAP MEMBER FUNCTION getPONo RETURN NUMBER IS
BEGIN
RETURN PONo;
END getPONo;
```

Druhá třídící metoda (Order metoda) je funkcí která přebírá právě jeden parametr jiného objektu stejného typu. Po porovnání vrací negativní číslo, pokud je daný objekt menší než porovnávaný objekt, nulu pokud jsou ekvivalentní a pozitivní číslo pokud je větší. Následuje příklad.

```
ORDER MEMBER FUNCTION compareCustOrders(x IN Customer_objtyp)
RETURN INTEGER IS
BEGIN
RETURN SELF.CustNo - x.CustNo;
END compareCustOrders;
```

**Statické metody**

Jak známo z objektově-orientovaných programovacích jazyků, na rozdíl od předchozích metod nepatří statická metoda přímo instanci objektu ale objektovému typu jako takovému. Definuje funkčnost aplikovatelnou na objektový typ jako celek.[6][5]




**obr. 8 – přístup k atributům pomocí metod**

### 4.2.2 Objektové tabulky

Objektová tabulka je speciálním druhem tabulky, kde každý řádek představuje objekt. Vytvořit objektovou tabulku je jednoduchá záležitost podobná klasickému zápisu vytváření relační tabulky, kde pouze místo těla tabulky uvedeme název objektového typu, který má uchovávat, viz. příklad.

*create table nazev_tabulky of uzivatelsky_typ*

Pokud chceme uchovávat kompozitní objekt jehož atributem je vložená tabulka, musíme specifikovat také název této vložené tabulky, která bude uchovávat daný atribut. To se provádí následovně:

*create table uzivatele of Uzivatel*
*nested table aktivity STORE AS uzivatel_aktivity*
*nested table kontakty STORE AS uzivatel_kontakty;*

Předchozí příklad vytváří objektovou tabulku uživatelů, kde každý uživatel obsahuje kolekci aktivit a kontaktů.

Na tabulku se můžeme dívat dvěma způsoby. Buď jako na jedno-sloupcovou tabulku, ve které každý řádek je objekt, který poskytuje objektově-orientované operace. Nebo jako více-sloupcovou tabulku, ve které má každý atribut objektu svůj sloupec, např. jméno a adresa, která poskytuje relační operace nad těmito daty.

Standardně, každý řádkový objekt v objektové tabulce, má asociován identifikátor objektu „OID“, který ho unikátně v objektové tabulce identifikuje. V distribuovaných prostředích nechá systémově-generovaný identifikátor objekty jednoznačně identifikovat. Je důležité si uvědomit, že objekt vytvořený jako sloupec klasické relační tabulky, nebo objekt vytvořený v rámci jiného objektu nebude mít OID![6]

### 4.2.3 Odkazový typ (Ref object type)

Odkazový typ REF je logický ukazatel na objekt či kolekci objektů. Takový odkazovaný objekt musí být uložený v objektové tabulce. Důvodem je fakt, že REF je vytvořen z OID odkazovaného objektu a OID je vytvářeno právě a pouze pro objekty v objektových tabulkách. REF je vestavěným datovým typem Oracle. Zde je dobré si z předchozího sdělení zapamatovat, že jeho hodnota je sice tvořena z OID, ale není to OID. Samotné odkazy a kolekce odkazů modelují vztahy mezi objekty, především 1:N, tím napomáhají redukovat potřebu cizích klíčů. Poskytují jednoduchý mechanismus pro navigaci mezi objekty. Odkaz lze změnit, že ukazuje na jiný objekt stejného typu nebo hodnotu „NULL“.

### Neplatný ukazatel

Pokud dojde k situaci, kdy je odkazovaný objekt smazán a REF není o tomto informován (explicitně změněn), dochází k situaci, že odkazovaný objekt je nedostupný a z odkazu se stává „dangling REF“. Oracle poskytuje pro tyto případy predikát „IS DANGLING“, který testuje, zda REF opravdu odkazuje nějaký platný objekt.

### Rámcový ukazatel

Při deklaraci typu sloupce, elementu kolekce, nebo atributu objektu jako odkazu, ho můžeme omezit, aby obsahoval pouze odkazy pro určenou objektovou tabulku. Takový odkaz nazýváme rámcový (scoped). Rámcový odkaz vyžaduje méně místa a umožňuje více efektivní přístup než obecný odkaz.[6]

### REF a DEREF

Na závěr uvedu základní situace, kdy a jak odkazy využíváme.

Pro vytvoření odkazu jako atributu objektu není třeba nic více, nežli před uvedený typ uvést klíčové slovo REF jako v následujícím příkladu.

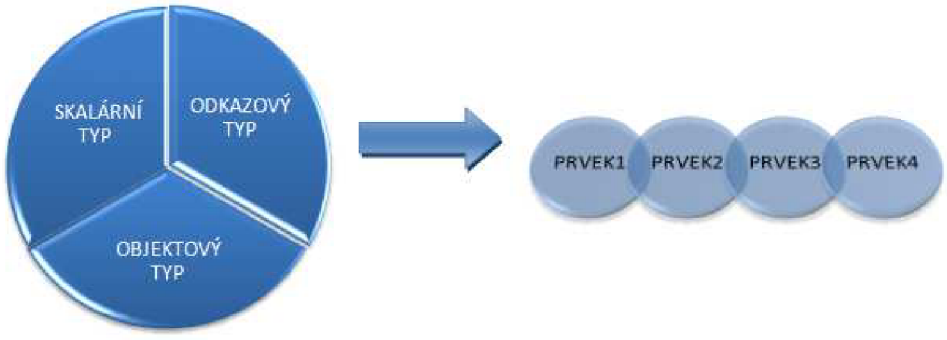
```
CREATE TABLE zamestnanci_obj
  ( jmeno   VARCHAR2(100),
    cislo NUMBER,
    pr_misto   REF adresa SCOPE IS adresa_obj_t );
```

Pro získání objektu z odkazu se používá klíčové slovo DEREF následovně (pozor, DEREF nefunguje v prostředí PL/SQL – navigace pomocí REF v tomto případě není možná ).

```
SELECT DEREF(adresa) FROM adresy;
```

V případě, kdy potřebujeme získat a vrátit referenci na místo objektu postupujeme následovně.

```
SELECT REF(e) FROM adresy e;
```




**obr. 9 – odkaz na kolekci**

### 4.2.4 Kolekce

Objekt může v rámci sama sebe obsahovat kolekci jiných objektů. Kolekce standardně používáme pro modelování asociací s kardinalitou větší než 1. Oracle podporuje dva různé způsoby konstrukce kolekce. Jsou jimi shora omezené pole proměnné délky „Varray“ (dále variabilní pole) a vložená tabulka „Nested table“. Kolekce mohou být standardně použity tam, kde ostatní datové typy.

#### *Variabilní pole (Varray)*

Variabilní pole je seřazená množina prvků. Všechny prvky dané množiny jsou stejného datového typu nebo jeho podtypu. Každý prvek má index, který je číslem korespondujícím s pozicí prvku v poli. Při definování pole, specifikujeme maximální počet prvků, které může obsahovat. Tuto hodnotu můžeme později změnit. Počet prvků je velikostí pole.

**Práce s polem v programovém prostředí**

Pole v databázi vytvoříme následujícím zápisem.

*create or replace type jmeno_kolekce as VARRAY (horni hranice poctu prvku) of datovy_typ;*

Následně s vytvořeným polem pracujeme podobně jako s objektem a tedy začínáme vytvořením pole pomocí konstruktoru. Po jeho vytvoření přistupujeme pomocí indexů k jednotlivým objektům, které pole nese. Práce s polem je podobná práci s poli jazyka Java. Následující případ vytvoří pole objektů, které bude reprezentovat informace o telefonních číslech a jako dotaz vrátí ty záznamy, které jsou mobilními telefony. Příklad při SQL dotazu používá klíčové slovo „table“, kterým říkáme aby pole bylo uvažováno jako tabulka.

```
Declare
telefony_varr  telefony_varr_type;
begin
        telefony_varr := telefony_varr_type ( telefon (111 111 111,  'mobil'),
                                              telefon (222 333 444,  'stabil'),
                                              telefon (555 555 666,  'mobil'));
        select p.cislo, p.typ from table(telefony_varr) p where p.typ =  'mobil' ;
end;
```

Nyní zkusme příklad, kdy pole není vytvořeno pouze samostatně, ale existuje jako úložiště záznamů v tabulce. Předpokládejme, že máme již vytvořenou a naplněnou tabulku kontaktů „kontakty“, která

sestává ze jména kontaktu a seznamu telefonních čísel. Ve výsledku obdobně vypíšeme mobilní telefony jistého kontaktu.

*select t.cislo from kontakty k, table( k. telefony_varr ) t  where k.jmeno = 'jmeno' and t.typ = 'mobil'*

**Bližší pohled na variabilní pole**

Podívejme se na toto pole reprezentující kolekci v databázi podrobněji. Pole je celé uloženo uvnitř struktury tabulky. To znamená, že jednotlivé prvky pole, ať už objekty nebo jednoduché typy, jsou uloženy v jediné struktuře v rámci daného řádku. Předchozí tvrzení bych podle informací Oracle upřesnil, kdy do 4000byte je pole uloženo v rámci datového typu raw(nebo varchar2) a pro větší velikosti je pak pole ukládáno jako LOB.

Pokud potřebujeme provést změnu v poli, což může být změna obsahu pole, přidání nového prvku do pole nebo odstranění prvku z pole, musíme provést nahrazení tohoto pole jako celku novým, takto upraveným polem. Uvedená fakta zřejmě snižují využitelnost tohoto řešení, musíme také počítat s tím, že pro úpravu takového pole budeme muset zavítat do procedurální nadstavby jazyka SQL. Dále je zde otázka ztráty výkonu při provádění samotných update dotazů voláním procedury (ve které pole upravujeme a znovu celé přiřazujeme) oproti nativnímu volání SQL kódu jednoduchého updatu.

## *Vložená tabulka (Nested table)*

Vložená tabulka je neseřazená množina prvků stejného typu. Při definování tabulky se oproti variabilnímu poli nespecifikuje maximum prvků, které může obsahovat. Při práci s vloženou tabulkou používáme příkazy SELECT, INSERT, DELETE a UPDATE stejně jako s klasickou tabulkou.

Prvky vložené tabulky jsou uloženy v oddělené tabulce, která obsahuje sloupec pro identifikaci řádku cílové tabulky nebo objekt, kterému každý prvek patří. Vložená tabulka má klasicky jeden sloupec. Pokud tento sloupec nese objektový typ, pak může být tabulka viděna jako více-sloupcová, se sloupcem pro každý atribut objektu. [6]

**Práce s vloženou tabulkou v programovém prostředí**

Vloženou tabulku vytvoříme následujícím způsobem.

*create or replace type  jmeno_kolekce  as table of datovy_typ;*

Příklad pro práci s vloženou tabulkou typu „select" v prostředí PL/SQL by byl obdobný variabilnímu poli. Jelikož výše popisované pole není vhodné pro update dotazy, podívejme se na ně pro vloženou tabulku. Pokračujme v naší tabulce kontakty se změnou, kdy pole náhradí vložená tabulka, kam vložíme nový telefon pro jistého uživatele. Jak jsem uvedl výše, vložená tabulka vytváří pro své záznamy externí tabulku. Vytvoření tabulky kontaktů s takovou kolekcí vypadá následovně.

```
create or replace type  telefony_n_tab  as table of Telefon_obj_typ;
create table kontakty
(
        jmeno varchar2(50),
        telefony telefony_n_tab
)
 nested table telefony store as telefony_nt;
```

A nyní již samotné vložení.

```
insert into table (select k.telefony from uzivatele k where k.jmeno = 'jmeno')  values
(Telefon('123 123 123', 'stabil'));
```

**Bližší pohled na vloženou tabulku**

O konstrukci vložené tabulky bych řekl (na rozdíl od variabilního pole, které pro ukládání spíše než kolekci reprezentuje pole „read-only"), že opravdu reprezentuje kolekci a umožňuje modelovat v rámci objektového typu v databázi ekvivalent kolekce vytvářené v rámci objektu v aplikaci. Při bližším zkoumání zjistíme, že vložená tabulka není vlastně tak úplně nic nového. V principu vlastně vytváříme pro data kolekce novou tabulku, jako bychom to dělali v klasickém přístupu a dotazovali se pomocí cizích klíčů. Vložená tabulka pak obsahuje sloupec s identifikátory na řádek objektu, ke kterému patří a to bych označil právě za cizí klíč, no a nakonec při spojování se využívá JOIN, přesně jako bychom to dělali my sami. Tedy dá se říci, že vložená tabulka před námi pouze skrývá klasický princip a poskytuje abstraktnější pohled.

## *Kdy použít variabilní pole a kdy vloženou tabulku*

Kdy použít variabilní pole?  Oracle uvádí, že při použití samotné konstrukce variabilního pole v programovém prostředí (procedury PL/SQL) podává pro dotazy mnohem lepší výkonnostní výsledky jak vložená tabulka, což je způsobeno charakterem jeho uložení (sbalen v rámci jedné struktury) a faktem, kdy na rozdíl od vložené tabulky nevyžaduje JOIN spojující dotazy při získávání dat.  Použití pole pro uchovávání dat v rámci tabulky je ovšem sporadické. Pokud máme v úmyslu časté update dotazy, tak pole rozhodně vhodné není, na druhou stranu pokud bychom takovou strukturu jednou vytvořili a především pouze četli, pak je to jistě alternativa, která umožní strukturování dat v databázi.

Kdy použít vloženou tabulku? Především, pokud potřebujeme reprezentovat kolekci uvnitř objektového typu.

### 4.2.5 Dědičnost

Dědičnost je velice užitečná vlastnost, která umožňuje vytvářet hierarchie typů, definováním úrovní, čím dál více specializovaných podtypů odvozených z obecného typu, který se pro odvozené typy nazývá supertyp. Odvozené podtypy dědí vlastnosti svých obecnějších typů (rodičů), ale navíc rozšiřují jejich definici. Mohou přidávat nové atributy a metody, nebo znovu definovat metody děděné od rodiče. Taková vytvořená hierarchie typů poskytuje vysokou úroveň abstrakce pro modelování složitosti aplikačního modelu v databázi.[6]

### 4.2.6 Evoluce

Příkazem ALTER TYPE lze typ modifikovat nebo rozvíjet současný typ, přičemž jsou možné tyto úpravy:

- Přidat nebo zrušit atribut
- Přidat nebo zrušit metodu
- Modifikovat numerický atribut, aby zvětšil svou délku, přesnost nebo stupnici
- Modifikovat délku znakového atributu (zvětšit jeho délku)
- Změnit typové vlastnosti FINAL a INSTANTIABLE

Metadata pro všechny tabulky používající pozměněné typy jsou aktualizovány na nový typ, takže data mohou být uložena v novém formátu. Existující data mohu být přetransformována na novou definici najednou nebo po částech, jak jsou modifikována. V takovém případě jsou data vždy reprezentována novým typem, i když jsou stále uložena ve starém.[6]

### 4.2.7 Objektové pohledy

Objektový pohled je cesta jak zpřístupnit relační data pomocí objektových vlastností. Pohledy dovolují vytvářet objektově-orientované aplikace bez nutnosti změny relačního schématu. Oracle umožňuje vytvářet objektovou abstrakci nad existujícími relačními daty skrze objektové pohledy. V principu zpřístupňujeme objekty patřící pohledu stejným způsobem jako řádkové objekty.

Objektové pohledy dovolují pracovat s polymorfismem. Můžeme vytvořit hierarchii pohledů, která odráží hierarchii typů. Tímto způsobem můžeme zpřístupnit data pouze v rámci určité úrovně specializace, která nás zajímá. Pokud se dotazujeme pohledu, který obsahuje pod-pohledy (specializované pohledy odvozené z dotazovaného pohledu), můžeme dostat zpět polymorfní data – řádky pro dotazovaný typ i pro jeho podtypy.[6]

# 5 Oracle a Programová prostředí

## 5.1 PL/SQL

### 5.1.1 Úvod

Samotný jazyk SQL je neprocedurálním jazykem. PL/SQL je procedurální nadstavba SQL z dílny Oracle. Plně podporuje procedury a větvení programu pomocí podmínek a cyklů. Umožňuje deklarovat konstanty, proměnné a kurzory. Jazyk PL/SQL je modulární a blokový. Základní entitou programu je blok.

**Typická struktura bloku:**

- Deklarační sekce
- Výkonná sekce
- Sekce výjimek

Deklarační sekce obsahuje deklarace proměnných, konstant, kurzorů apod. Výkonná sekce obsahuje funkční logiku. V sekci pro správu výjimek se ošetřují potenciální chybové stavy. Povinná je pouze výkonná sekce.

### 5.1.2 Podpora pro objekty

Přímo v PL/SQL nemůžeme objektové typy definovat. Můžeme je definovat v SQL a poté, co jsou zaneseny ve schéma databáze, je můžeme v PL/SQL bloku, subprogramu nebo balíčku používat.

Objektový typ můžeme použít tam, kde vestavěný typ, můžeme ho použít v těle bloku nebo jako vstupní/výstupní parametr. Pro zpřístupnění vlastností objektu používáme tečkovou notaci. Pro vytvoření nového objektu uvnitř PL/SQL kódu, nejprve typ deklarujeme a poté ve výkonné sekci inicializujeme pomocí konstruktoru.[6]

## 5.2 Java

### 5.2.1 Úvod

Za klíčovou vlastnost pro vývojáře stavějící na spolupráci s databázovými systémy bych označil podporu pro flexibilní, programový přístup k datům databáze z externí aplikace. Oracle podporuje řadu programovacích jazyků (COBOL, C, perl), jeden je však nutno vyzdvihnout a je jím Java. Java

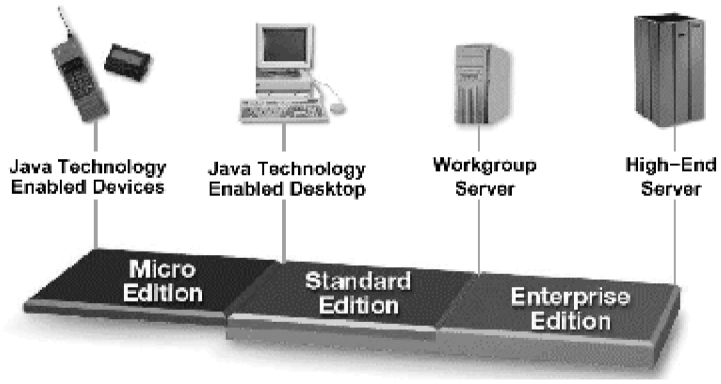
umožňuje vytvářet standardizované, přenositelné aplikace, které napíšeme jednou a následně nasadíme na různé databázové systémy či platformy. Java je více než „pouhým“ programovacím jazykem, konkrétně Java EE framework poskytuje architekturu znovu využitelných komponent.

Java ja platforma, která nabízí všechny potřebné součásti pro tvorbu kompletních a komplexních aplikací, od manipulace s daty, přes uživatelské rozhraní k aplikační logice, na straně serveru.

### 5.2.2 Java jako programovací jazyk i něco víc

Definice programovacího jazyka sama o sobě není tím úplně signifikantním faktorem pro budování efektivních, bezchybných, znovu-využitelných aplikací. Přesto volba vhodného typu jazyka velmi přispívá k úspěšnému splnění těchto cílů. Java je objektově-orientovaný jazyk, z toho vyplývá podpora pro zapouzdření, dědičnost a polymorfismus. Java staví na syntaxi jazyků C a C++, ale do značné míry oproti těmto jazykům redukuje složitost, a to zavedením silně typového systému, automatickou správou paměti (garbage collector) nebo redukcí ukazatelů.

Java sestává z mnoha verzí produktů, API atd. Abychom se ve všech jeho platformách vyznali, dělí se Java na několik specifických oblastí. Java poskytuje pro práci tři základní platformy, které jsou určeny pro specifická prostředí, od mobilních aplikací, přes desktopy, až po výkonné servery. Následující obrázek mapuje tyto platformy.




**obr. 6 - Java platformy**

Pro nás jsou zajímavé specifikace J2SE a Java EE, která předešlou variantu v sobě zahrnuje.

Java jako programovací jazyk sestává ze dvou hlavních částí. Z prostředí pro běh aplikace (runtime environment - JRE) a vývojového prostředí (development environment – J2SE). JRE je implementace JVM pro specifickou platformu. Je to minimum požadavků, aby mohl být spuštěn kód Javy. J2SE je softvérový vývojářský kit pro Javu. Sestává ze všech nástrojů pro vývoj a nasazení Java aplikací pro jednotlivé implementované platformy.

Pokud vytváříme netriviální podnikové systémy, které zahrnují spojení s databází, správu transakcí, rolí atd., je jistě možné vytvářet je zcela pomocí samotného programovacího jazyka Java,

avšak takové řešení bude zdlouhavé, bude mít limitovanou přenositelnost a znovu využitelnost kódu vně daného systému. Java 2 Enterprise Edition (Java EE) je založena na jazyku Java (obsahuje J2SE), ale je zároveň více než to. Definuje klíčové technologie pro tvorbu vícevrstvých byznys aplikací založených na komponentách. Obsahuje frameworky pro požadavky těchto aplikací, jako je spojení s databází, správa transakcí, rolí, bezpečnost, distribuovaná komunikace atd. V základu Java EE jsou specifikace API pro tvorbu platformně nezávislých, znovu využitelných komponent. Tyto komponenty jsou rozmístěny v Java EE kontejnerech. Kontejner pro komponenty poskytuje komplexní množinu služeb, kterými jsou řízení životního cyklu komponenty nebo také řízení souběžného přístupu ke komponentám. Tím, že za nás kontejner poskytuje tyto služby, se můžeme soustředit přímo na samotnou byznys logiku a specifické problémy na nižších úrovních vůbec neřešit. Java EE technologie se hojně využívají v kontextu s programováním databází. Ty hlavní si uvedeme.

**Servlety**

Servlety představují standardní cestu k výstavbě aplikací na straně serveru. Servlety mají přístup ke všem Java J2EE API a podporují veškerou Java logiku. Jsou nasazeny v J2EE servlet kontejnerech.

**Java Server Pages (JSP)**

JSP je cesta pro prezentační vrstvu (HTML, XML), provádí dynamické volání komponent skrze vestavěné skriptovací tagy. Tyto tagy oddělují uživatelské rozhraní od generování obsahu. JSP se po kompilaci stává servletem a běží v servlet kontejneru.

**Java Messaging Service (JMS)**

JMS je standardní API pro zasílání asynchronních zpráv mezi komponentami, aplikacemi, databázemi.

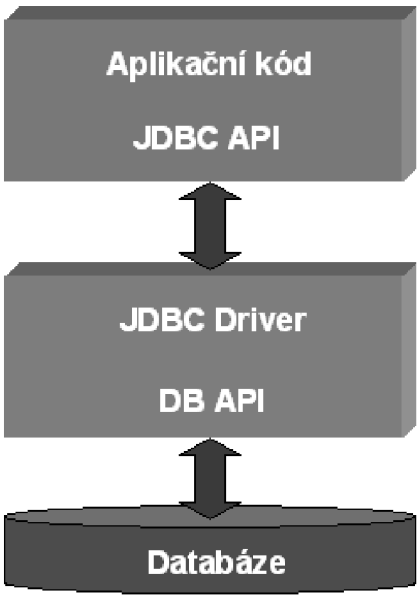
**Java Database Connectivity (JDBC)**

JDBC je standardní rozhraní pro propojení Java aplikací s datovými zdroji. Původně bylo určeno pro standardizaci spojení s relačními datovými zdroji. Standard umožňuje jednotlivým poskytovatelům implementovat API jako JDBC ovladač pro jednotlivé datové zdroje. Díky velké popularitě dnes existuje řada implementací JDBC pracující i s nerelačními datovými zdroji (například XML, C-ISAM). Vzhledem k všudypřítomnosti databázových aplikací je JDBC API podporováno i v J2SE.

### 5.2.3 JDBC

Firma Sun Microsystems vytvořila rozhraní JDBC, pro poskytování unifikovaného přístupu 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. Zjednodušený princip ukazuje následující obrázek.




**obr. 7 - zjednodušený princip komunikace pomocí JDBC**

### 5.2.4 Virtuální stroj a bytecode

Zdrojový kód Javy musí být před vykonáváním zkompilován. Kompilace převede kód z textové formy do binárního kódu (bytecode), což je sada instrukcí srozumitelná pro virtuální stroj Javy JVM (Java Virtual Machine). Bytecode je platformě přenositelný, díky tomu by nemělo být ani nutné překompilovat kód, pokud ho chceme nasadit na odlišném operačním systému. Virtuální stroj se dá nazvat srdcem Javy. Je to softvérově implementovaný abstraktní počítač, který vykonává bytecode.

### 5.2.5 Výkon

Efektivní správa paměti včetně automatického uvolňovaní (garbage collection) a platformě-nezávislá interpretace kódu jsou pozitivními prvky Javy. V prvních verzích JVM byl výkon pod úrovní klasických nativně kompilovaných jazyků. S dalšími verzemi dělala Java co do výkonu velké pokroky a dnes se na tyto jazyky již dotáhla. Kompilátor JIT (Just-In-Time), který provádí v reálném čase před samotným vykonáváním kódu jeho kompilaci do nativního kódu je dnes standardně v JVM implementován. Slouží ke kompilaci metod nebo tříd do nativního kódu pro rychlé zpracování. Java od verze 1.2 obsahuje technologii HotSpot, což je kombinace JIT a dynamického profilovače k identifikaci, kdy a co je nejvhodnější ke kompilaci. Výrobci jako Oracle dokonce poskytují pro své platformy nativní překladače Javy. Nativní překladač optimálně přeloží bytecode do strojového kódu daného systému (platformě závislé), výsledkem by měl být znatelný nárůst výkonu aplikace.

## 5.3 Java a PL/SQL uvnitř databáze Oracle

Oracle v současnosti podporuje přímo v databázi vykonávání jazyků PL/SQL a poměrně nově také jazyka Java. Je otázka, kdy a na co použít který jazyk. Každý z jazyků PL/SQL jako procedurální a Java jako objektově-orientovaný jistě najdou opodstatnění jejich užití v pro ně vhodných situacích. Co se týká PL/SQL, tak na rozdíl od Javy nemusí řešit nesourodost mezi sebou a relačním modelem díky sdílení systému typů s databází, kde Java musí provádět konverzi. PL/SQL dále nativně podporuje databázově-orientované konstrukce (kurzory, záznamy). Tyto vlastnosti znamenají, že PL/SQL dokáže poskytnout lepší výkonnost a produktivitu při aplikacích náročných na data. Java je naproti tomu vhodná při vytváření komplexní aplikační logiky.

## 5.4 Přístup do databáze z Javy

Existují dva primární modely pro tvorbu Java aplikací společně s využitím prostředí databáze Oracle. Prvním modelem uvažujeme pohled na aplikaci, jejíž logika je vykonávána vně databázového serveru. Takové aplikace využívají JDBC nebo jiné frameworky (Java EE, OracleAS TopLink), které JDBC využívají ke komunikaci s databází. Druhý model představuje pohled na aplikaci, která vykonává svou logiku uvnitř databáze, kde lze jako příklad uvést uložené procedury jazyka Java. Oba modely pak používají JDBC jako spojení s relačními daty.

### 5.4.1 Java vykonávaná vně databáze

Vzhledem k podpoře Oracle napříč všemi vrstvami architektury, zde existuje několik způsobů, jak komunikovat s databází, vhodná volba závisí na požadavcích kladených na aplikaci.

Společné pro celou skupinu těchto aplikací je to, že využívají JDBC API, ať už přímo nebo nepřímo pomocí perzistentních frameworků, k přístupu do databáze.

**Java applet s JDBC**

Jedná se o applet běžící v rámci webového prohlížeče, jehož VM provádí přímé JDBC dotazování do databáze. Tento způsob je vhodný při nízké frekvenci výměny dat mezi klientem a serverem, ale s požadavky na vyspělejší uživatelské rozhranní (v porovnání s HTML).

**Desktop Java aplikace s JDBC**

Je řešení založené na samotné desktop aplikaci provádějící přímé volání JDBC do databáze. Vhodné pro intranetové aplikace s požadavky na vyspělé uživatelské rozhraní (v porovnání s HTML).

**Desktop Java aplikace s OracleAS TopLink**

Podobné řešení jako předchozí, pouze namísto JDBC se používá OracleAS TopLink perzistentní framework (diskutovaný v kapitole 7).

**Webový prohlížeč a servlet**

Aplikace nasazená na aplikačním serveru. Používá JDBC, Entity Beans nebo OracleAS TopLink pro komunikaci s databází. O prezentaci obsahu se stará typicky HTML ve webovém prohlížeči. Jedná se o typickou architekturu pro Java EE internetové aplikace.

### 5.4.2 Java vykonávaná uvnitř databáze

Od verze Oracle 8i, je v databázovém serveru implementován Java virtuální stroj JVM (již dříve server obsahoval PL/SQL virtuální stroj). OracleJVM (dříve JServer) běží ve stejném programovém a adresovém prostoru jako jádro SŘBD. Každé databázové sezení se jeví, že má vlastní virtuální stroj pro vykonávání Java kódu. Ve skutečnosti jednotlivá sezení sdílí jeden JVM a v rámci něho má každé sezení vlastní izolovanou paměť. S podporou J2SE přímo v databázi přidává Oracle novou dimenzi ve vytváření standardizovaných aplikací. Nyní lze využít výhod uložených procedur bez nutnosti jejich implementace v databázově-závislém jazyku (PL/SQL, Transact-SQL). Oracle také podporuje nativní překlad pro uložené procedury psané v Javě, tím se dosahuje znatelného zvýšení výkonu v době běhu aplikace. S každou vzestupnou verzí Oracle přidává podporu pro nové verze J2SE. Oracle verze 10g podporuje specifikaci J2SE 1.4.

## 5.5 SQLJ

SQLJ je produktem vývoje standardní cesty, jak vložit SQL příkazy do programů Javy. SQLJ program je Java program, který obsahuje vložené SQL příkazy, které vyhovují standardu ISO SQLJ Language Reference syntax. Takový zdrojový kód obsahuje směs standardního kódu Javy, deklarací tříd SQLJ a SQLJ vykonavatelných příkazů s vloženými SQL operacemi. SQLJ se používá pro zjednodušení přístupu do databáze v kontrastu s klasickým přístupem pomocí JDBC.

Princip je takový, že programátor píše kód pro API vyšší úrovně a následně při kompilaci se překládá na standardní JDBC volání. SQLJ-překladač je program Javy, který překládá vložený SQL kód do standardního kódu Javy využívající JDBC, spolu s SQL operacemi ve formě volání do SQLJ běhového prostředí. Za běhu aplikace pak toto SQLJ běhové prostředí volá JDBC pro komunikaci s databází. SQLJ také umožňuje odchytávání chyb ve SQL dotazech již v době kompilace. SQLJ může být uvnitř databáze použito v rámci procedury, triggeru nebo metody.

SQLJ přístup poskytuje pro spojení aplikačních tříd s objekty databáze silné typové vazby. Obsahuje SQLJ iterátor, což je silně typová verze result setu v JDBC. Iterátory se používají pro

zpracování výsledků dotazů typu SELECT. Pro srovnání uvádím ukázku konkrétní komunikace pomocí JDBC a SQLJ.

**JDBC:**

```
// předpokládejme, že již máme vytvořené spojení conn
String name;
int id=37115;
float salary=20000;

// nastavíme příkaz
PreparedStatement pstmt = conn.prepareStatement
("SELECT ename FROM emp WHERE empno=? AND sal>?");
pstmt.setInt(1, id);
pstmt.setFloat(2, salary);

// vykonáme dotaz
ResultSet rs = pstmt.executeQuery();
while (rs.next())
{
  name=rs.getString(1);
  System.out.println("Name is: " + name);
}

// pozavíráme po sobě
rs.close()
pstmt.close();
```

**SQLJ:**

```
// předpokládejme, že již máme vytvořené spojení conn
String name;
int id=37115;
float salary=20000;
#sql {SELECT ename INTO :name FROM emp WHERE empno=:id AND sal>:salary};
System.out.println("Name is: " + name);
```

Aby bylo možné v databázi zdrojové soubory Javy využívat, musíme je do ní načíst jako objekty určitého schématu, kde schéma odpovídá konkrétnímu uživatelskému účtu v databázi. K tomu slouží utilita „loadjava“. Abychom mohli volat kód Javy z klienta, či prostředí SQL musíme ho nejprve zveřejnit. Zveřejnit lze jak celou třídu, tak individuální metodu.

V databázi lze také vytvořit typy mapující existující Java aplikační třídy. Objektům těchto tříd lze poskytnout persistentní úložiště využitím SQLJ objektových typů, kde jsou veškeré metody implementovány v rámci těchto tříd.

## 5.6 Uložené metody

### 5.6.1 Volání metod uvnitř databáze

V databázi můžeme spouštět uložené procedury napsané v Javě. Na to potřebujeme vytvořit třídu, u níž implementujeme statickou metodu, kterou budeme poté v databázi volat.
Tedy definujme takovou třídu:

```
public class Hello
{
  public static String world()
  {
    return "Hello world";
  }
}
```

Následně třídu načteme do databáze:
*loadjava -user scott/tiger Hello.class*

V posledním kroku ji zavoláme. Pokud chceme volat metodu v rámci SQL volání, musíme ji zveřejnit spolu se specifikací volání. Ta definuje argumenty, které metoda přebírá a SQL typy, které vrací.
Tedy vytvoříme funkci:

```
CREATE OR REPLACE FUNCTION helloworld RETURN VARCHAR2 AS
 LANGUAGE JAVA NAME 'Hello.world () return java.lang.String';
```

a zavoláme ji:

```
SQL> VARIABLE myString VARCHAR2(20);
SQL> CALL helloworld() INTO :myString;
Call completed.
SQL> PRINT myString;
MYSTRING
----------------------------------------
Hello Word [7]
```

### 5.6.2 Volání uložených metod z Klienta

Pro dotazování informací z databáze můžeme použít rozhraní SQLJ nebo JDBC. Následuje ukázka kompletního SQLJ kódu.

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
```

```
#sql ITERATOR MyIter (String ename, int empno, float sal);

public class MyExample
{
  public static void main (String args[]) throws SQLException
  {
    Oracle.connect("jdbc:oracle:thin:@oow11:5521:sol2", "scott", "tiger");
    #sql { INSERT INTO emp (ename, empno, sal) VALUES ('SALMAN', 32, 20000) };
    MyIter iter;
    #sql iter={ SELECT ename, empno, sal FROM emp };
    while (iter.next())
    {
      System.out.println(iter.ename()+" "+iter.empno()+" "+iter.sal());
    }
  }
}
```

A klientská metoda, která poběží na serveru:

*CREATE OR REPLACE PROCEDURE sqlj_myexample AS LANGUAGE JAVA NAME `MyExample.main(java.lang.String[])';* [7]

## 5.7 Interakce databáze Oracle a Javy

Zopakujme, že přímo v databázi je integrována platforma J2SE. Virtuální stroj (OracleJVM) uvnitř databáze Oracle je plně kompatibilní se standardním JVM, přesto se vzhledem k prostředí, ve kterém je nasazen, v určitých věcech liší. V klasickém J2SE prostředí přistupujeme ke zdrojovým souborům Javy na souborovém systému pomocí proměnné CLASSPATH. Když vykonáváme program, je automaticky volána metoda main jako proces operačního systému. V prostředí OracleJVM, pokud chceme zavolat třídu Javy, musíme ji nejdříve načíst v rámci nějakého schéma do databáze (databázové struktury obecně jsou uspořádány do schémat příslušející jednotlivým uživatelům). Vstupní bod, který je klasicky funkce main, může být nahrazen jakoukoli statickou metodou uvnitř načtené třídy.

Tedy metodika je načíst zdrojové soubory Javy do pro ně určeného schématu, následně určit schémata, která se budou používat (obsahují třídy, se kterými chceme pracovat, například jsou na ně odkazy z naší hlavní třídy), dále se musí provést „zveřejnění“. Zveřejnění je proces, který mapuje jména Java metod a typů parametrů na jejích SQL protějšky (dochází k zanesení v datovém slovníku). Po zveřejnění jsou tyto metody dostupné pomocí SQL a PL/SQL volání. Posledním krokem je samotné volání Java metod z prostředí SQL.

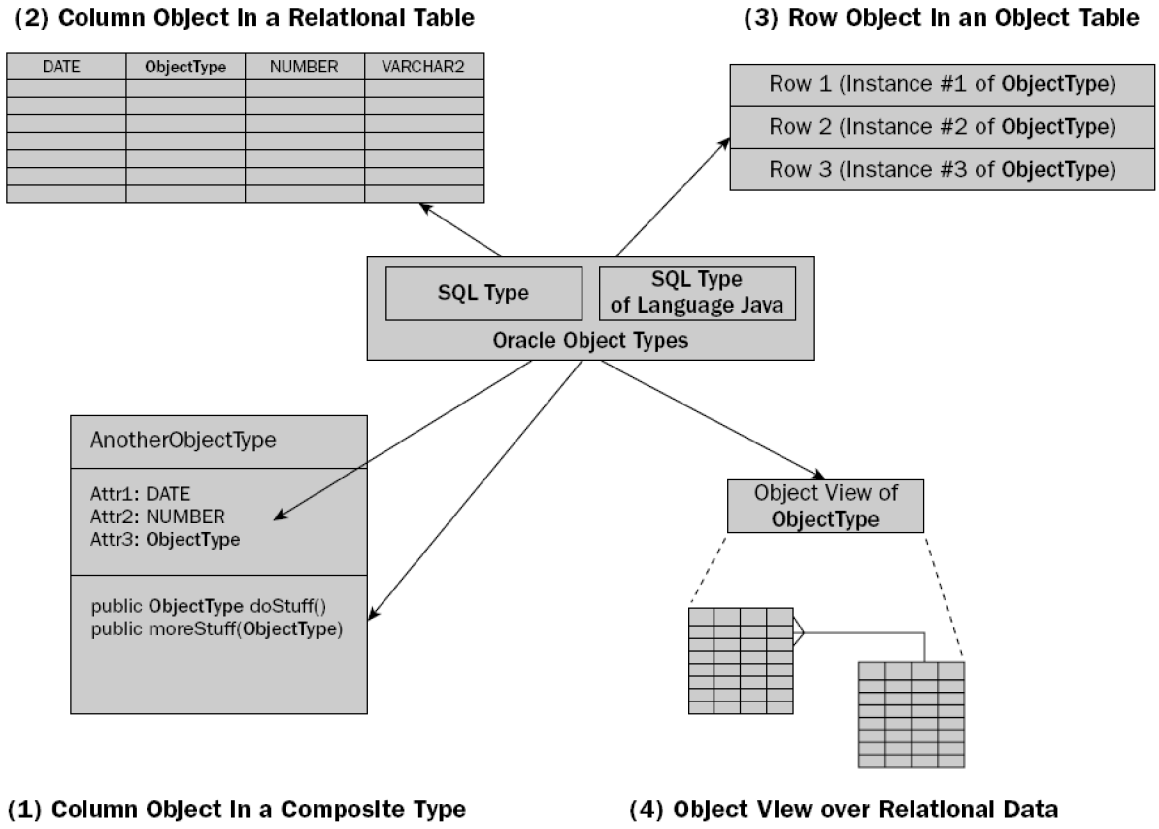
Toto byla směrem dovnitř, kdy chceme například část aplikační logiky přenést na server. Oracle také poskytuje programový přístup k objektům databáze z Aplikace. Touto problematikou, ať už směrem ven z databáze nebo dovnitř, se budeme zabývat ve zbytku páté kapitoly.

### 5.7.1 Použití objektových typů Oracle

Za základní myšlenku považujme řešit nesourodost mezi objektově-orientovaným jazykem a relační databází jako perzistentního úložiště aplikačních dat. Tato nesourodost vzniká při proceduře mapování, která je dekompozicí bohatých objektově-orientovaných vlastností na relační model dat podporující ve své podstatě dvou-dimensionální strukturu (tabulku) skalárních datových typů. Objektově-relační podpora dává flexibilní možnosti pro vybudování vhodných struktur v databázi pro potřeby aplikace.

Typický scénář tvoří využití existujících Java tříd k definici perzistentních objektových typů, nebo naopak. Podpora pro SQLJ objektové typy byla přidána ve verzi Oracle 9i. Je to způsob definice vlastního typu v databázi podle existující aplikační třídy v Javě. Aby to bylo možné, je nutné, aby třída Javy implementovala jedno ze dvou rozhraní. Těmi jsou buď standardní JDBC rozhraní (implementuje SQLData) nebo rozhraní, které poskytuje přímo Oracle (implementuje ORAData).

Definice objektového či SQLJ objektového typu specifikuje strukturu a přidružené chování, ale nespecifikuje způsob uložení nebo vytváření těchto objektů v databázi. Cest, jak realizovat tyto objekty v databázi, je více, viz následující obrázek.




**obr. 12 – objektové typy, převzato z[5]**

(1)Objekt může být použit jako součást jiného objektu. Pokud to upřesníme, může být použit jako atribut, parametr metody nebo návratová hodnota funkce jiných objektů. (2)Může být použit jako

sloupec relační tabulky, pak ho nazýváme sloupcovým objektem. (3)Další přístup používá objektovou tabulku, která obsahuje objekty právě jednoho specifického typu, kde každý řádek představuje daný objekt, a takové objekty nazýváme řádkové. Na rozdíl od těch sloupcových mají řádkové unikátní identifikátor OID, který je jednoznačně identifikuje a umožňuje použití reference. (4)Další možností jsou objektové pohledy. Objektové pohledy projektují relační tabulky do virtuálních objektových tabulek specifického datového typu. Každý řádek v pohledu je instancí objektu s atributy, metodami a OID. Je to alternativní cesta těžící z objektově-orientovaných vlastností bez nutnosti konverze existujících relačních tabulek na tabulky objektové, tedy relační tabulky zůstávají nedotčeny. Pomocí pohledů navíc můžeme používat tečkovou notaci namísto vytváření složitých spojovacích (JOIN) dotazů.

### 5.7.2 Slabé a silné mapování objektových typů na třídy Javy

Objektově-relační podpora byla představena v JDBC verzi 2.0. API poskytuje dvě rozhraní pro reprezentaci objektových typů databáze jako objektů Javy. Rozhraní java.sql.Struct pro slabě typový přístup a java.sql.SQLData pro silně typový přístup a oba přístupy jsou databází Oracle podporovány. Nadto Oracle poskytuje vlastní rozhraní oracle.sql.STRUCT a oracle.sql.ORAData.

První struktura „Struct“ je Java reprezentací objektového typu, která obsahuje hodnotu pro každý atribut toho typu. Struktura poskytuje „volné“ mapování. To znamená, že každý atribut objektového typu je reprezentován jako instance java.lang.Object, což je obecný typ, ze kterého se odvozují ostatní typy Javy. Je tedy na programátorovi vědět jakého datového typu je atribut a na něj ho přetypovat. Dané řešení se používá v případech, kdy manipulujeme s objekty přímo v databázi a nepotřebujeme je převést a prezentovat v uživatelském rozhraní aplikace.

Druhá cesta je poskytnout uživatelskou třídu, která implementuje rozhraní SQLData. V tomto případě JDBC zajistí přenos hodnot z databáze, vytvoření instance dané třídy a naplnění této třídy hodnotami z databáze. Toto řešení se za nás stará o veškeré detaily mezi reprezentací objektu v databázi a aplikaci. Při použití této varianty musíme zavést mapu typů, což je seznam objektových typů databáze s třídou Javy implementující rozhraní SQLData. Mapa typů slouží ke správnému výběru objektu na základě výsledku dotazu, který se bude vytvářet. Shrneme-li dosavadní poznatky, pak rozšíření pro podporu SQLData či ORAData velmi zjednodušuje přístup k objektovým typům, poskytnutím tříd a metod usnadňujících práci jak s prostými objekty, tak ukazateli a kolekcemi.[5]

## 5.8 Oracle JPublisher

JPublisher je utilita (napsaná v Javě) generující kód pro zpřístupnění JVM a objektově-relačních možností databáze Oracle vývojářům aplikací. Konkrétně provádí zveřejnění Java, PL/SQL a SQL na straně serveru pro zpřístupnění z venčí databáze, například aplikačního serveru nebo klienta. Od verze 10g také podporuje generování kódu pro podporu volání webových služeb z databáze.

**JPublisher umožňuje vytvářet třídy Javy pro následující typy:**

- Uživatelsky definované SQL objekty
- Odkazy na objekty
- Uživatelsky definované kolekce
- PL/SQL balíčky
- Server-side třídy Javy
- SQL dotazy a DML příkazy

Generuje přístupové metody pro každý atribut objektového typu. Pokud objektový typ obsahuje uložené procedury, pak JPublisher generuje obslužné metody pro jejich volání. Obslužná metoda je metoda, která zavolá uloženou proceduru běžící uvnitř databáze. JPublisher podobně generuje třídy a jejich přístupové metody pro PL/SQL balíčky.

Samozřejmě se lze obejít i bez JPublisheru a psát veškerý kód manuálně, to je však časově náročné a náchylné k chybám. Přesto nemusí řešení poskytované JPublisherem vždy dostačovat. V takovém případě můžeme nechat vygenerovat inicializační verze podtříd, které manuálně rozšíříme o požadovanou funkcionalitu.

V databázi Oracle 10g, JPublisher generuje SQLJ kód a implicitně ho překládá na kód Javy. Takto generovaný Java kód(.java) dále kompiluje na Java třídní soubory(.class). Oracle SQLJ překladač a běhové knihovny jsou dodávány spolu s JPublisherem. Ten je využívá pro generování mezikódu. Například pro vytváření metod volající operace v databázi.

### 5.8.1 Mapování

Provádí se mapování z uživatelsky definovaných SQL typů a PL/SQL typů na třídy Javy. Pro mapování uživatelských SQL typů databáze, jako jsou objekty, kolekce a reference na objekty můžeme použít Oracle specifickou implementaci, standardní implementaci nebo generickou implementaci.

Tedy můžeme generovat třídy implementující rozhraní ORAData, nebo použít obecné rozhraní SQLData poskytované specifikací JDBC (pak reference a kolekce jsou reprezentovány geneticky). Dále, pomocí rozhraní „oracle.sql.*“ můžeme veškeré tyto typy, ať už objektové typy, odkazy nebo kolekce, reprezentovat genericky (například pro objektové typy rozhraní STRUCT a pro reference rozhraní REF). Na rozdíl od prvních dvou řešení, není toto poslední silně typové.

Existují dvě základní kategorie mapování z SQL typu do Javy. Můžeme mapovat pomocí hodnoty „jdbc“ nebo hodnoty „oracle“. Numerické typy mají navíc možnost „objectjdbc“ a „bigdecimal“.

### *JDBC mapování*

Většinu numerických typů je mapována jako primitivní typy Javy(int, float,...). SQL typy DECIMAL a NUMBER jsou mapovány na java.math.BigDecimal. LOB(Large Object) typy a ostatní ne-numerické vestavěné typy jsou mapovány na standardní DBC typy(Blob, TimeStamp, String, ...).
Pro objektové typy jsou generovány třídy implementující rozhraní SQLData. Ne všechny vestavěné typy Oracle mají v JDBC mapovaní svůj ekvivalent, v takovém případě je použit slabě typový princip. Hodnota NULL tu není podporována. Není zde kontrola přetečení hodnoty.

### *Object JDBC mapování*

Na rozdíl od klasického JDBC mapování, kdy byly numerické typy mapovány na primitivní typy Javy, zde jsou mapovány na objekty (java.lang.Integer, java.math.BigDecimal,…). Umožňuje práci s nulovými hodnotami.

### *Oracle mapování*

Numerické, LOB, a jiné vestavěné typy jsou mapovány na třídy balíku oracle.sql (oracle.sql.DATE, oracle.sql.NUMBER,….). Pro objekty, kolekce a reference JPublisher generuje třídy implementující rozhraní ORAData. Jelikož „oracle“ mapování nepoužívá primitivní typy, může u všech atributů pro reprezentaci hodnoty NULL používat Java null.

Generované třídy obsahují kód nezbytný ke čtení a zápisu objektu v databázi. Při používání třídy generované JPublisherem, připojuje JDBC ovladač nezbytné běhové knihovny. Pro specifikaci jaké typy může JPublisher zveřejnit slouží vstupní soubor „input file“. Pro nastavení vlastností slouží zase „properties file“.

**Pro objektové typy za použití rozhraní ORAData se generuje:**

Třída reprezentující objektový typ v Javě (typ.java). Volitelně základní podtřída. Tu následně můžeme rozšířit o požadovanou funkcionalitu, kterou JPublisher v hlavní třídě neposkytuje. Volitelně rozhraní pro generovanou třídu. Příslušnou referenční třídu pro odkazování na objekt (typREF.java).
Třídy, pro všechny objekty a kolekce, vložené jako atributy objektu. To je samozřejmě logické, pokud chceme materializovat objekt, musí jít materializovat veškeré jeho součásti.

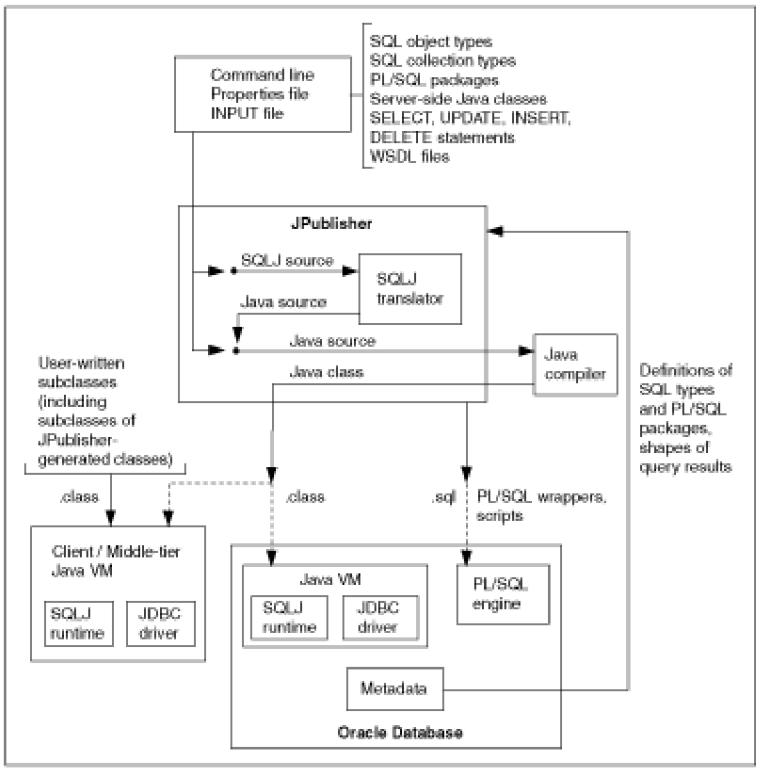
**Pro SQL dotazy a DML příkazy se generuje:**

Třída implementující metody, které spouští SQL příkaz. Volitelně základní podtřídu pro rozšíření funkcionality. Volitelně rozhraní pro implementaci dané třídy.[5][6]

## 5.8.2 Zveřejnění

### *Přehled procesu zveřejnění:*

1. Spustit JPublisher s parametry na vstupu.
2. JPublisher zpřístupní databázi, ke které je připojen a získá definice SQL či PL/SQL entity, které jsme zadali na vstupu.
3. JPublisher generuje .java nebo .sqlj zdrojové soubory (.sqlj v případě obslužné metody pro uložené procedury).
4. Standardně při překladu .sqlj, JPublisher volá SQLJ translátor, pro překlad .sqlj souborů na .java soubory.
5. JPublisher volá Java kompilátor, který překládá .java soubory na .class soubory.




**obr. 13 – schéma vstupů, výstupů a překladu pro JPublisher**

### *Zveřejnění SQL a DML příkazů*

Volba JPublisheru –sqlstatement zveřejní SQL dotazy (select) nebo DML příkazy (insert, update, delete) jako metody třídy Javy. Takový přístup se nabízí pro využití SQL ve webových službách. Také je to rychlá cesta pro přístup k SQL a DML bez psaní dodatečného JDBC kódů, což může být přesně, co chceme. Uvažujme jistou tabulku v databázi, jejíž data chceme upravovat. Klasickou

cestou na úrovni samotného JDBC jsme zatížení dodatečnou komunikací, kterou představuje kód pro načtení SQL dotazu, odeslání, přijetí odpovědi a její zpracování do vhodné podoby atd. Pomocí JPublisheru se od veškeré dodatečné komunikace abstrahujeme. Potřebné SQL dotazy zveřejníme jako metody jisté třídy a v Javě, pomocí vytvořené instance této třídy, přímo komunikujeme s danou tabulkou v databázi.

### *Objektové typy*

Pro komunikaci s objektovými typy jsem již popisoval rozhraní SQLData, případně ORAData. JPublisher umožňuje generování třídy Javy, které implementují jedno z těchto rozhraní. Procedura spočívá ve vytvoření vstupního souboru, který specifikuje objektové typy určené ke zveřejnění. Zároveň s každým typem uvádíme jméno třídy, která ho bude v Javě reprezentovat (př. *SQL TelefonCis_typKolekce AS SeznamTelCisel*).[6][5]

## 5.9 OCI (Oracle Call Interface)

OCI je množina C knihovních funkcí, které může aplikace využít pro manipulaci dat a schémat databáze Oracle. OCI podporuje tradiční i objektově-orientovaný přístup k databázi. Důležitou komponentou je množina volání pro správu objektové cache. Objektová cache je blok paměti na straně klienta, která dovoluje programům ukládat objekty a navigovat mezi nimi, bez dodatečné komunikace se serverem.

Oracle poskytuje také rozšíření OCCI, což je vrstva vystavěná nad OCI, která přidává objektově orientované paradigma.

**OCI poskytuje:**

- Přístup k objektům databáze
- Konverzi typů (datum, string, číselné) databáze na typy jazyka C
- Manipulaci s objekty v objektové cache a správu velikosti této paměti[6]

# 6 Návrh a implementace

## 6.1 Specifikace požadavků

Cílem praktické části projektu je návrh a implementace vhodné ukázkové aplikace, jejímž cílem není kompletní funkcionalita, nýbrž využití objektových rozšíření databáze Oracle 10g a demonstrace přístupu k nim.

Za tímto účelem byl navržen ukázkový podnikový systém pro podporu řízení vztahů se zákazníky, řízení aktivit, řízení času. Systém umožní správu zákazníků. Správou zákazníku se konkrétně myslí vést záznamy o samotném zákazníkovi, aktivitách uživatele jako jsou například obchodní schůzky nebo telefonní hovory, úkolech uživatele, dále povede záznam o vedení obchodních případů zákazníka. Umožní správu kontaktů uživatelů. Záznamy uživatele se buď povedou jako osobní (přístup má pouze majitel) nebo jako veřejné a k takovému záznamu mají přístup ostatní uživatelé. V systému se budou vyskytovat dva primární druhy uživatelů, a to manažer, který využívá systém pro vedení zákazníků a správce systému, který se stará o samotné účty uživatelů v systému nastavení systému (práv, rolí, číselníků,…).

### *Funkční požadavky:*

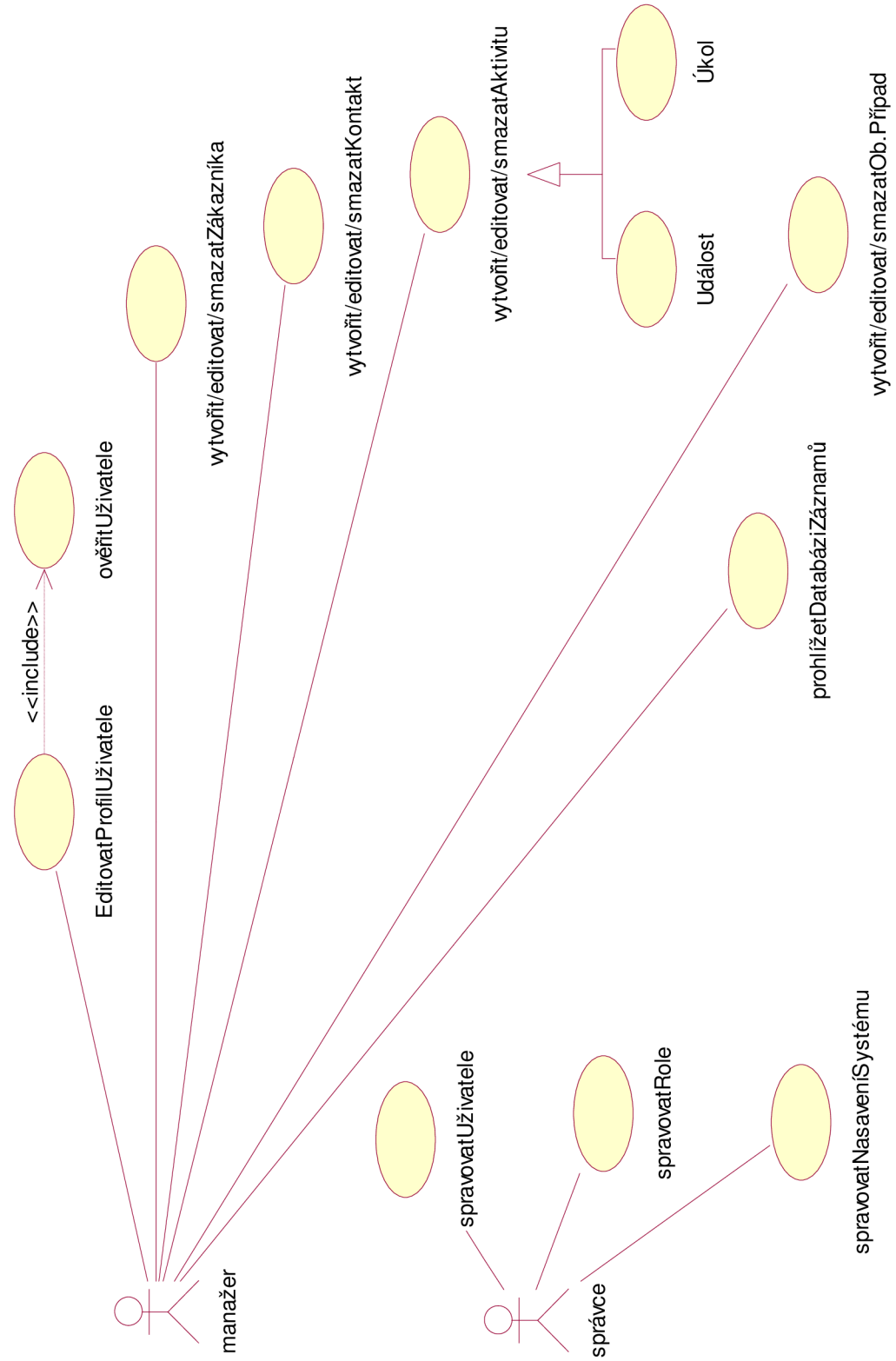
1. Systém bude vést databázi zákazníků.
2. Systém bude vést aktivity uživatele systému a jeho časový rozvrh.
3. Systém bude vést obchodní případy vázané k zákazníkům.
4. Systém bude vést kontakty uživatele.
5. Systém bude rozlišovat dva typy uživatelů z hlediska přístupu k systému (manažer, správce).
6. Systém bude uživateli umožňovat sdílet své záznamy s ostatními uživateli.

### *Nefunkční požadavky:*

1. Systém bude implementován v jazyce Java.
2. Pro perzistenci bude využit databázový systém Oracle 10g se zaměřením na jeho objektové vlastnosti.

## 6.2 Diagram a specifikace případů použití

### 6.2.1 Diagram




**obr. 14 – diagram případů použití**

### 6.2.2 Slovní popis

#### *Aktéři*

**Manažer**

Využívá systém. Spravuje a využívá záznamy o zákaznících, kontaktech, obchodních případech nebo aktivitách.

**Správce**

Spravuje nastavení systému. Spravuje uživatele. Spravuje oprávnění, které uživatelé v systému mají.

#### *Případy užití (spravovat = vytvářet/editovat/mazat)*

**Prohlížet databázi záznamů**

Systém umožní procházet záznamy databáze a to jak pomocí vyhledávání, tak poskytnutím hierarchického pohledu (navigace podle kategorií např. zobrazit záznamy - pouze obchodní případy - pouze aktuálně probíhající).

**Editovat Profil**

Systém nabídne změnu údajů o daném uživateli a zanese změny do databáze. Manažer smí editovat pouze svůj profil.

**Spravovat zákazníka**

Vytváří nového zákazníka, edituje údaje již existujícího zákazníka, odstraňuje existujícího zákazníka.

**Spravovat Kontakt**

Vytváří nový kontakt, edituje údaje již existujícího kontaktu, odstraňuje existující kontakt.

**Spravovat obchodní případ**

Vytváří nový případ v rámci zákazníka, edituje údaje již existujícího případu, odstraňuje existující případ.

**Spravovat Aktivitu**

Vytváří novou aktivitu, edituje údaje již existující aktivity, odstraňuje existující aktivitu.

**Ověřit uživatele**

Provádí identifikaci uživatele podle loginu a hesla.

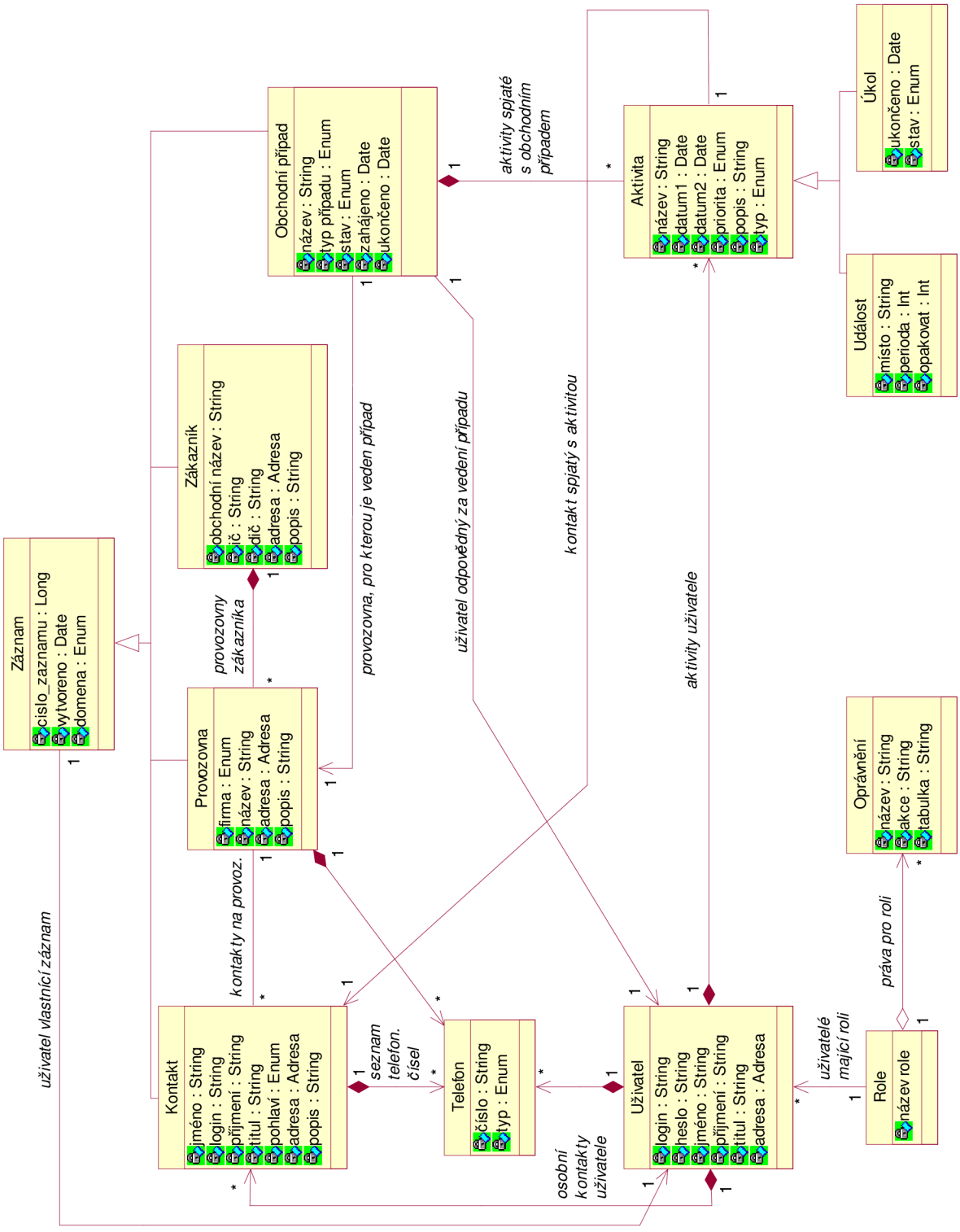
**Spravovat uživatele**

Vytváří účet nového uživatele, edituje existující účet uživatele, odstraňuje existující účet uživatele.

**Spravovat role**

Vytváří nové role, kterým přiděluje práva a následně tyto role přiřazuje uživatelům, edituje a odstraňuje existující role.

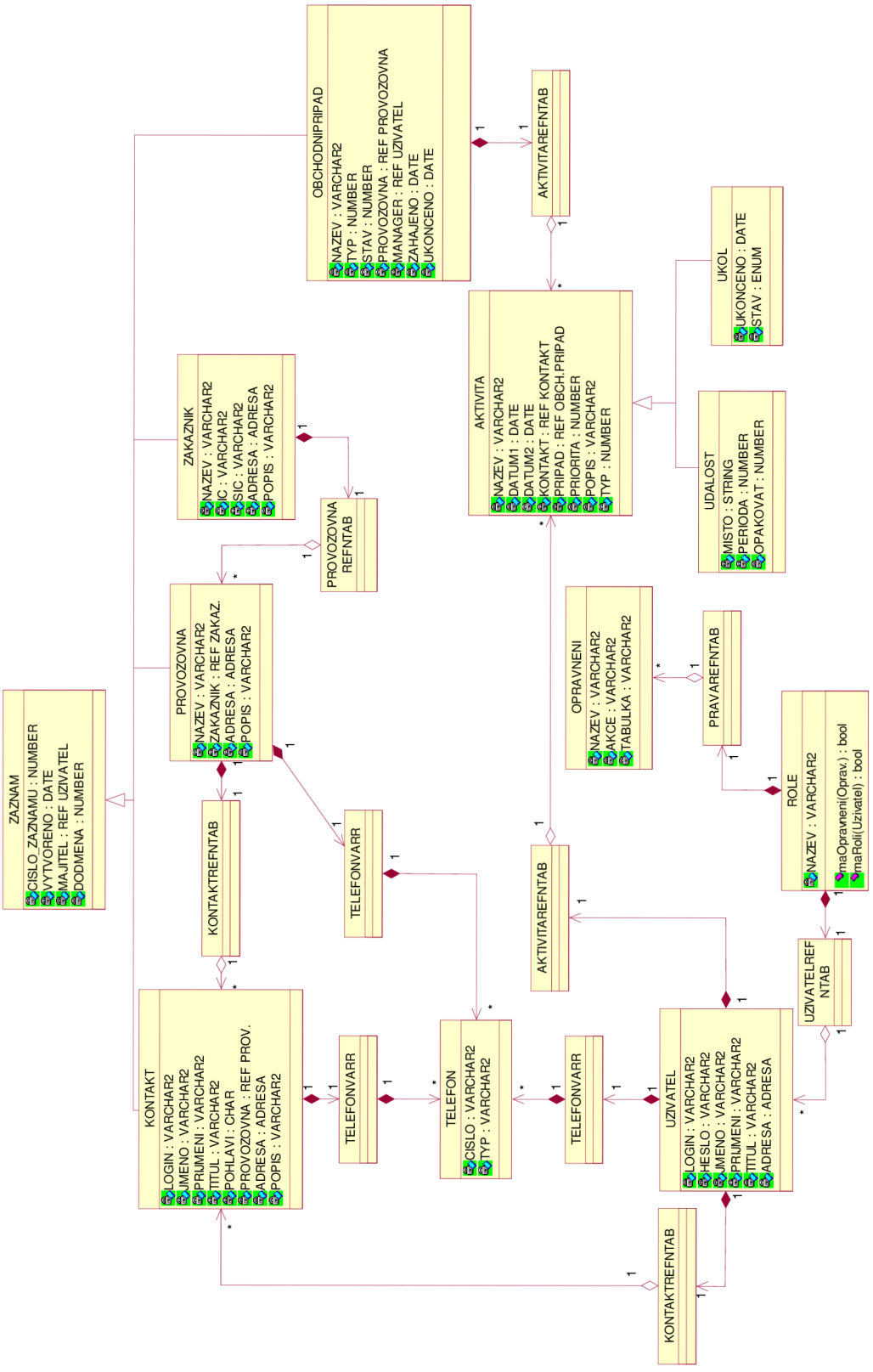
# 6.3 Konceptuální diagram




**obr. 15 – konceptuální diagram**

## 6.4 Databázový model

### 6.4.1 Diagram objektového schéma databáze




**obr. 16 – diagram objektového schéma databáze**

### 6.4.2 Změny oproti relačnímu návrhu

Při návrhu modelu objektově-relační databáze, oproti klasickému relačnímu návrhu, vyvstávají jisté odlišnosti. První takovou změnou, zvolíme-li práci s odkazy v databázi, bude fakt, že v rámci modelované entity začne být dříve takřka nepostradatelný sloupec „ID“ sloužící jako primární klíč nevyužitý, a to především ve smyslu identifikace dané entity. Tuto funkci přebírá OID objektu a je tedy nutné zvážit jakou funkci daný dříve nepostradatelný atribut plní (stále ho lze využít například pro indexování tabulky).

Další změnou, a to výraznou, bude způsob modelování vztahu 1:N či M:N. Takový vztah je synonymem pro kolekci. Uvažujme relační model. Pro vztah 1:N máme dvě tabulky, kde tabulka celku obsahuje jeden řádek s unikátním klíčem a tabulka součástí obsahuje N řádku s cizími klíči do tabulky celku. V případě vztahu M:N navíc vytváříme pomocnou třetí tabulku.

Objektový návrh, oproti předchozímu řešení, modeluje a zapouzdřuje kolekci jako atribut vytvořeného uživatelského typu. O problematice návrhu kolekce pomocí objektových rozšíření přijde řeč později.

Co se týká dalších změn při tvorbě objektového-relačního modelu, zmiňme nový prvek REF. Pomocí toho klíčového slova říkáme, že místo samotného objektu budeme uchovávat pouze jeho odkaz. Tímto principem odkazů, na řádkové objektové typy, dochází k nahrazení relačních cizích klíčů.

### 6.4.3 Tabulky a objektové typy

#### *Vytváříme objekty*

Definici objektu můžeme vytvořit dvěma způsoby. Můžeme sami přímo v databázi vytvořit definici objektu, resp. jeho atributů a metod, pak vytváříme klasický objektový typ v databázi. Druhým způsobem je vytvoření třídy, v jazyce Java, která implementuje rozhraní SQLData nebo pro Oracle specifické ORAData. Tuto třídu nahrát do databáze a v databázi specifikovat mapování mezi ní a objektovým typem, který ji bude v databázi reprezentovat. Definice metod takového objektového typu se již neuvádějí, ale vytvoří se z dané třídy. Výhodou druhého řešení je možnost specifikovat složitější aplikační logiku přímo v jazyce Java, která se bude vykonávat v databázi. Vytváříme tedy buď klasický objektový typ, nebo SQLJ objektový typ. Názorně obě možnosti uvádím.

Příklad ukazuje definici Uživatele jako objektového typu:

```
create or replace type Uzivatel as object
(
 login VARCHAR2(30),
 heslo VARCHAR2(50),
 jmeno VARCHAR2(30),
 prijmeni VARCHAR2(50),
 titul VARCHAR2(20),
 adresa crm.Adresa,
 telefon crm.TelefonVarr,
 aktivity crm.AktivitaRefNTab,
 kontakty crm.KontaktRefNTab
);
```

Příklad definice SQLJ objektového typu:

```
create or replace type aktivita as object
external name 'crm.Aktivita' language java using sqldata (
 nazev VARCHAR2(30) external name 'nazev',
 datum DATE external name 'datum',
 pripad LONG external name 'pripad',
 ........);
```

V projektu byly implementovány obě varianty.

### *Ukládáme objekty*

Objektový typ v databázi můžeme uložit dvěma způsoby. První možností je uložit ho jako jeden z atributů klasické relační tabulky. Takový přístup nám k objektu neukládá OID a tedy takový objekt není ani odkazovatelný. Tento princip se hodí, pokud chceme pouze strukturovat určitá data v rámci tabulky, kde příkladem může být struktura adresy. Druhou možností je vytvořit samostatný objekt v objektové tabulce, který je identifikovatelný podle svého OID a samozřejmě tedy i odkazovatelný pomocí referencí. Takový přístup se hodí, chceme-li zastoupit aplikační objekt v databázi a případně na něho i navázat určitou funkcionalitu pomocí přidružených členským metod (metody může samozřejmě objekt volat v obou případech, jako element objektové tabulky i jako atribut v řádku klasické relační tabulky, nicméně již nemůže použít odkazů).

V projektu byl, vzhledem k orientaci na identifikaci pomocí odkazů, jednotně použit princip uložení v objektových tabulkách. Jinými slovy, veškeré objektové typy jsou implementovány jako řádkové objektové typy.

### *Popis schéma databáze*

S odkazem na objektové typy v databázovém schéma, byly implementovány objektové tabulky uchovávající tyto objekty. Konkrétně objektové tabulku uživatelé, kontakty, zákazníci, provozovny, případy, aktivity, role a oprávnění. Díky principu dědičnosti není vytvářena tabulka pro super-typ Záznam, který v sobě generalizuje základní data, společná pro většinu objektových typů. Dále nejsou vytvářeny samostatné tabulky pro podtypy Událost a Úkol, ale namísto nich existuje společná, generalizovaná tabulka aktivity.

Ve schématu databáze, jsou vztahy typu 1:1 již modelovány jako atributy objektových typů. Pro lepší přehled, těchto vztahů, se můžeme podívat do konceptuálního diagramu. Ostatní vztahy jsou reprezentovány kolekcemi. Touto problematikou se budeme zabývat v následující kapitole „Kolekce“. Na závěr uvádím příklad implementačního kódu, který vytváří objektovou tabulku uživatelů.

```
create table uzivatele of Uzivatel (login NOT NULL)
nested table aktivity STORE AS uzivatele_aktivity
nested table kontakty STORE AS uzivatele_kontakty
```

## 6.4.4 Kolekce

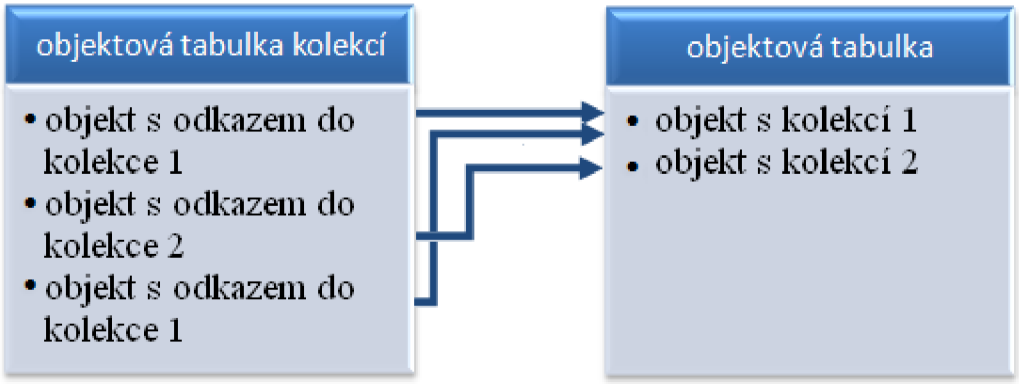
Nyní rozeberme problém, jak navrhnout a implementovat kolekci v databázi. Mějme objektově-orientovanou aplikaci v Javě. V rámci aplikace mějme objekt a v rámci objektu mějme kolekci objektů (neřešme, zda ve formě kompozice, nebo agregace). Řešme problém, jak takovou situaci modelovat v databázi.

Řekněme, že máme tři možnosti. Můžeme uložit prvky kolekce mimo objekt v jiné, samostatné tabulce, která by obsahovala cizí klíče identifikující objekt, kterému kolekce patří. Dále můžeme použít některou z dvou poskytovaných konstrukcí, jmenovitě variabilního pole a vložené tabulky.

### *Varianta s uložením do samostatné tabulky (bez vložené tabulky).*

Můžeme říci, že takováto varianta je typickým relačním přístupem, který obohatíme o využívání odkazů. Vlastně pouze vyměníme cizí klíč za odkaz na objekt a místo klasické tabulky použijme tu objektovou. Máme tedy dvě objektové tabulky. Jednu pro objekty vlastnící kolekci, říkejme jim celky, a pak druhou pro jejich kolekce objektů, říkejme jim součásti.

S takovým řešením nebudeme spokojeni. Sice zde využíváme odkazů, nicméně přesně opačným směrem, nežli by to mu mělo být. Každá součást má v tomto modelu uloženu informaci o tom, kterému celku patří. To není dobře, ne součásti o celku, nýbrž celek o součástech by měl mít informace a součást (atribut), pokud to není potřeba, by ani o svém celku vědět neměla.



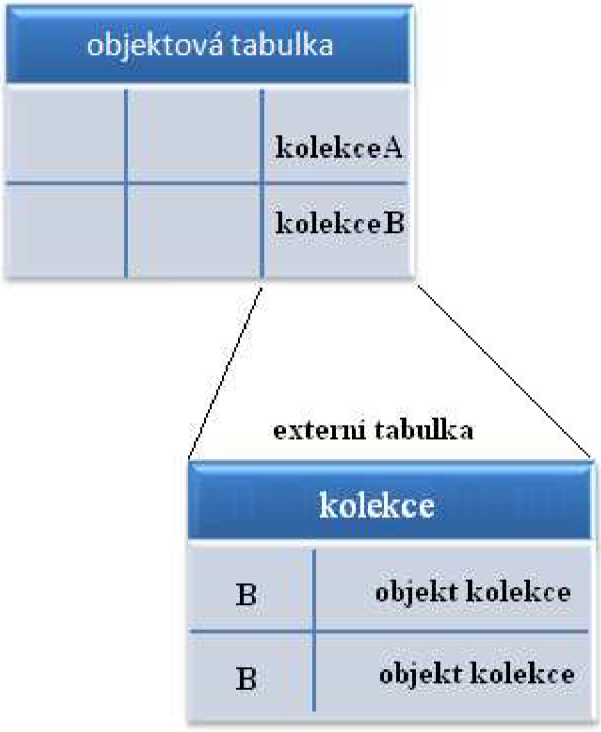

**obr. 17 – varianta s oddělenou tabulkou**

### *Varianta s variabilním polem*

Co se variabilního pole týče, tak spíše nežli kolekci představuje reprezentaci pole, které je určeno pro čtení jednou vytvořených záznamů a nikoliv časté změny. O variabilním poli píši v jedné z předcházejících kapitol popisující objektové rozšíření databáze Oracle. Tady jen dodám, že pole mi nepřišlo jako vhodný obecný přístup, jak modelovat kolekci. Lze ho použít pro vztah kompozice bez využití odkazů, kde se vyhneme častým změnám dat.

### *Úvod do vložené tabulky*

Vložená tabulka se tváří jako součást (konkrétně sloupec) jí nadřazené tabulky, ve které je uložen objekt, který vloženou tabulku vlastní jako svůj atribut. Podíváme-li se dovnitř, zjistíme, že se jedná o oddělenou tabulku mimo původní tabulku. Nejlépe bude, když se podíváme na obrázek.




**obr. 18 – varianta s vloženou tabulkou**

Jak můžeme vidět, taková vložená tabulka obsahuje identifikátor uložený v atributu vlastnického objektu a samotný objekt kolekce. Takové řešení se nápadně podobá klasickému principu rozložení dat do dvou tabulek s identifikací pomocí cizích klíčů.

### *Varianta s vloženou tabulkou přímo obsahující objekty*

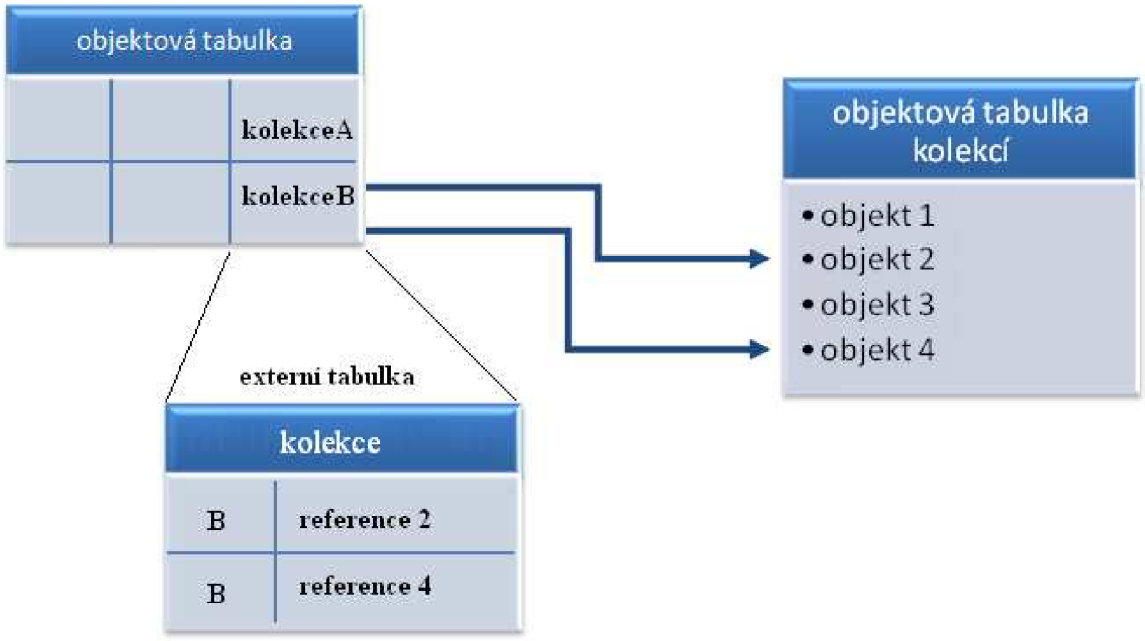
Co se týká vhodnosti reprezentace kolekce v databázi, pak vložená tabulka je favoritem ze všech diskutovaných řešení. Použijeme-li vloženou tabulku jako atribut objektu, získáme pohled podobný tomu na aplikační úrovni. Kolekce je zapouzdřena a přistupujeme k ní výhradně skrze vlastnický objekt. Pokud kolekce obsahuje přímo objekty, pak pro takový případ nelze modelovat agregaci, ale výhradně kompozici, a to bez identifikace prvků kolekce pomocí odkazů

Proč nelze použít odkazy? Každý objekt, uložený přímo v objektové tabulce (tedy ne jako pouhý atribut jiného objektu), je odkazovatelný pomocí svého jedinečného OID. V SQL, pokud potřebujeme získat jednoznačný identifikátor objektu v tabulce, použijeme klíčové slovo REF a získáme jednoznačnou identifikaci objektu na daném řádku tabulky. V takovém případě REF pracuje nad řádkovým objektem, přijímající jako argument alias tabulky. REF neumí pracovat nad sloupcovými objekty, ale také nad vloženými objekty, což je i příklad vložené tabulky, protože takové objekty nemají OID. Z toho vyplývá, chceme-li používat objekty přímo jako prvky vložené tabulky, musíme si najít jiný způsob identifikace (vrátit se k primárnímu klíči). Práce s takovými objekty se stává více komplikovanou.

### *Varianta s vloženou tabulkou obsahující odkazy*

Pokusme se skloubit výhod řešení vložené tabulky a samostatné objektové tabulky. Využijme faktu, že v rámci objektové tabulky fungují odkazy. Ukládejme tedy objekty kolekce do této tabulky. V rámci objektu vlastnícího kolekci, vytvořme vloženou tabulku uchovávající odkazy na objekty v kolekci. Nyní je problém s identifikací objektů vyřešen a modelování vztahu agregace je již jednoduchá záležitost. Pravdou sice je, že namísto dvou tabulek, máme nyní dokonce tři, ovšem lepší řešení, které by používalo vložené tabulky a odkazy, nemám.

V projektu byla implementována právě tato varianta, která se mi jevila nejvíce flexibilní a dle mého názoru nejlépe odrážela kolekci v aplikaci. A nakonec obrázek takového řešení.




**obr. 19 – varianta s vloženou tabulkou s odkazy**

### *Popis schéma databáze*

V diagramu databázového schématu si můžeme všimnout entit, které končí na „VARR“ nebo „NTAB“. Jedná se o vytvořené uživatelské typy, které reprezentují kolekci. V prvním případě se jedná o variabilní pole, v druhém pak o vložené tabulky.

Variabilní pole bylo použito pro implementaci kolekce telefonů. Tato kolekce je v objektu modelována jako atribut pomocí silné vazby celek-součást, tedy kompozicí, kdy jsou prvky kolekce uloženy v kolekci hodnotou.

Pro implementaci ostatních kolekcí byly použity vložené tabulky. Implementace těchto kolekcí byla založena na principu znázorněném na obrázku č.19. Prvky takové kolekce jsou v ní ukládány odkazem.

### 6.4.5 Uložené metody

K objektovému typu můžeme přiřadit logiku prostřednictvím definice členských metod. Možnosti, jak to provést, máme zhruba tři. Můžeme třídu s metody napsat přímo v jazyce Java a zveřejnit ji do databáze. Následně ze zveřejněné třídy vytvořit SQLJ objektový typ. Můžeme vytvořit samostatný objektový typ a v jazyce PL/SQL jeho metody naimplementovat. Můžeme tyto dva přístupy i kombinovat, kdy vytvoříme klasický objektový typ a přiřadíme mu nějaké, ze zveřejněných metod, které definujeme externě.

#### *Popis schéma databáze*

V diagramu databázového schématu si můžeme všimnout metod, které jsou přidruženy k objektovému typu Role. Metoda „maRoli“ přebírá jako argument odkaz na uživatele. Hledá shodu tohoto uživatele se svými registrovanými uživateli a vrací výsledek, zda tento uživatel takovou roli obsahuje. Metoda „maOpravneni“ provádí podobnou akci, kdy kontroluje, zda role obsahuje konkrétní oprávnění.

## 6.5 Úvod do problému mapování

### 6.5.1 Slabě typované struktury

Tento pojem v Javě odkazuje na objekty, které implementují JDBC standardní rozhraní java.sql.Struct (dále pouze Struct). Takové objekty reprezentují objektové typy v databázi v generické podobě jako kolekci atributů. Atributy objektu jsou uloženy v poli Objektů, které obsahuje individuální atributy jako Java objekty (každý je typu Object). Pořadí atributů v poli je stejné jako pořadí atributů objektového typu specifikovaných v době vytváření typu. S takovým přístupem potřebujeme vzít jeden atribut objektového typu po druhém a přetypovat ho na ekvivalentní typ v Javě. Takový přístup může být užitečný, pokud potřebujeme v aplikaci pracovat s objektem volného typu v generické podobě (jako kolekce jednotlivých atributů). Pokud potřebujeme pracovat a manipulovat v paměti s konkrétními objekty, je mnohem lepší cestou použít silně typované objekty.

Oracle implementuje standardní rozhraní Struct jako třídu oracle.sql.STRUCT a přidává specifické rozšiřující metody. Na rozdíl od rozhraní java.sql.Struct, které může být použito pouze pro získávání dat, rozšíření Oracle umožňuje i vkládání a změnu dat.

#### *Rozhraní java.Sql.Struct*

Standardní rozhraní definující mapování v Javě pro SQL objektové typy. Objekt Struct obsahuje hodnotu pro každý atribut objektového typu, který reprezentuje. Rozhraní má tři základní metody.

**public Object[] getAttributes(Map map) throws SQLException;**

Tato metoda získává hodnoty atributů, pomocí mapy typů. Mapa odkazuje na mapování mezi databázovým typem a příslušnou mapující třídou v Javě. Není-li vhodné mapování nalezeno, pak je objektový typ mapován jako typ Struct. Rozhraní dále poskytuje metodu stejného jména, jen nepřebírá žádný atribut, v takovém případě je použito přednastavené defaultní mapování.

**public String getSQLTypeName() throws SQLException;**

Tato metoda vrací plně kvalifikované jméno (schéma.jméno_typu) objektového typu v databázi, kterého daný objekt Struct reprezentuje.

Rozhraní neposkytuje žádné metody pro vytváření, vkládání či změny ve Struct objektech. Pro vkládání nebo změny musíme použít oracle.sql.Struct (nebo implementovat vlastní třídu). Třída Oracle.sql.STRUCT implementuje rozhraní Struct a poskytuje rozšiřující funkce, které umožňují vkládání a změnu dat.

### 6.5.2 Silně typovaná rozhraní

Každý objektový typ je reprezentován v aplikaci svou vlastní třídou. Pro vytvoření takové třídy pro reprezentaci objektového typu, zde musí být mechanismus, který by informoval JDBC ovladač, která třída mapuje který objektový typ a na jaký konkrétní typ bude daná třída mapovat jednotlivé atributy toho objektového typu. Ovladač musí být schopen číst a zapisovat tyto třídy. Každá třída, která mapuje v aplikaci objektový typ, musí implementovat rozhraní, které tyto věci splňuje. Pro oracle 10g máme dvě možná rozhraní zajišťující předešlé, z nichž jedno musíme implementovat. Jsou jimi java.sql.SQLData (dále jen SQLData) a oracle.sql.ORAData spolu s oracle.sql.ORADataFactory (dále jen ORAData a ORADataFactory).

#### *SQLData*

Rozhraní SQLData sestává z tří metod. Z metody *getSQLTypeName()*, která vrací plně kvalifikované jméno objektového typu, které objekt v aplikaci reprezentuje. Další metodou je *readSQL(SQLInput stream, String typeName)*. Tato metoda naplní objekt daty z databáze. Parametr SQLInput je vstupní proud dat, který obsahuje data reprezentující SQL objekt. Poslední metodou je *writeSQL(SQLOutput stream)*. Metoda zapisuje jednotlivé atributy objektu do výstupního SQL datového proudu. Konvertuje zpět jednotlivé atributy na SQL hodnoty.

#### *ORAData*

Namísto tříd implementující rozhraní SQLData můžeme využít třídy, které implementují Oracle specifické rozhraní ORAData a ORADataFactory. Rozhraní ORAData poskytuje lepší flexibilitu nežli předchozí řešení. Obsahuje metody *toDatum(OracleConnection connection)*, která konvertuje aplikační data na *oracle.sql.** typy reprezentované v databázi a *create (Datum datum, int sqlTypeCode)*, která vrátí instanci třídy implementující ORAData.

#### *SQLData vs. ORAData a ORADataFactory*

Výhodou SQLData je fakt, že se jedná o JDBC standard a jeho implementace dělá kód přenositelnější napříč databázemi, než by tomu bylo u ORAData. Na druhou stranu i ORAData má své výhody. ORAData je více flexibilnějším rozhraním, které dovolí mapování mezi objekty Javy a SQL typy podporovanými v oracle.sql balíku. Toho se dá využít, především pokud chceme mapovat SQL nestandardní typy (např. jako je BFILE), kdy to pomocí SQLData mapování není možné (umí pouze standardní SQL typy). ORAData nevyžaduje mapování z objektových typů na třídy Javy (provádí to za nás automaticky).

## 6.6 Mapování objektových typů na třídy Javy

Pro vytvoření aplikačních tříd, z objektového modelu databáze, jsem použil generující utilitu JPublisher. Byl vytvořen konfigurační soubor pro generování rozhraní ORAData. Pro účely projektu bylo potřebovat generovat tři třídy od každého objektového typu. Byla generována standardní perzistentní třída mapující objektový typ, dále její pomocná třída pro rozšíření vlastností databázového objektu o aplikační kód a jako poslední odkazová třída pro reprezentaci REF typu v aplikaci. Automaticky se vygenerují i třídy pro vložené tabulky a variabilní pole, pokud byly použity v generovaných třídách.

Jak sem zmínil výše, pro mapování bylo vybráno silně typované rozhraní ORAData. Důvodem je jednak fakt, že pro práci s databází Oracle bychom mohli využívat jím poskytnuté rozhraní, které umožňuje mapovat veškeré (i nestandardní) typy. Dalším důvodem je praktická zkušenost s mapováním kolekcí. Kolekce se z databáze vrací jako java.sql.Array a jako taková, lze pouze číst. ORAData implementují vlastní třídu pro danou kolekci (na rozdíl od SQLData, která to neumožňuje), která implementuje manipulaci s kolekcí jako je změna či přidávání prvků.

### *Použitý vstupní mapovací soubor pro JPublisher*

- SQL crm.Adresa generate crm.Adresa_p AS crm.Adresa
- SQL crm.Aktivita generate crm.Aktivita_p AS crm.Aktivita
- SQL crm.Kontakt generate crm.Kontakt_p AS crm.Kontakt
- SQL crm.ObchodniPripad generate crm.ObchodniPripad_p AS crm.ObchodniPripad
- SQL crm.Provozovna generate crm.Provozovna_p AS crm.Provozovna
- SQL crm.Telefon generate crm.Telefon_p AS crm.Telefon
- SQL crm.Udalost generate crm.Udalost_p AS crm.Udalost
- SQL crm.Ukol generate crm.Ukol_p AS crm.Ukol
- SQL crm.Uzivatel generate crm.Uzivatel_p AS crm.Uzivatel
- SQL crm.Zakaznik generate crm.Zakaznik_p AS crm.Zakaznik
- SQL crm.Zaznam generate crm.Zaznam_p AS crm.Zaznam
- SQL crm.AktivitaNTab AS crm.AktivitaNTab
- SQL crm.AktivitaRefNTab AS crm.AktivitaRefNTab
- SQL crm.KontaktNTab AS crm.KontaktNTab
- SQL crm.ProvozovnaRefNTab AS crm.ProvozovnaRefNTab
- SQL crm.KontaktRefNTab AS crm.KontaktRefNTab
- SQL crm.TelefonVarr AS crm.TelefonVarr

Za klíčovým slovem SQL následuje název objektového typu. Za slovem generace následuje název persistentní aplikační třídy a za slovem AS následuje název odvezené třídy, ve které můžeme rozšiřovat funkčnost původní třídy.

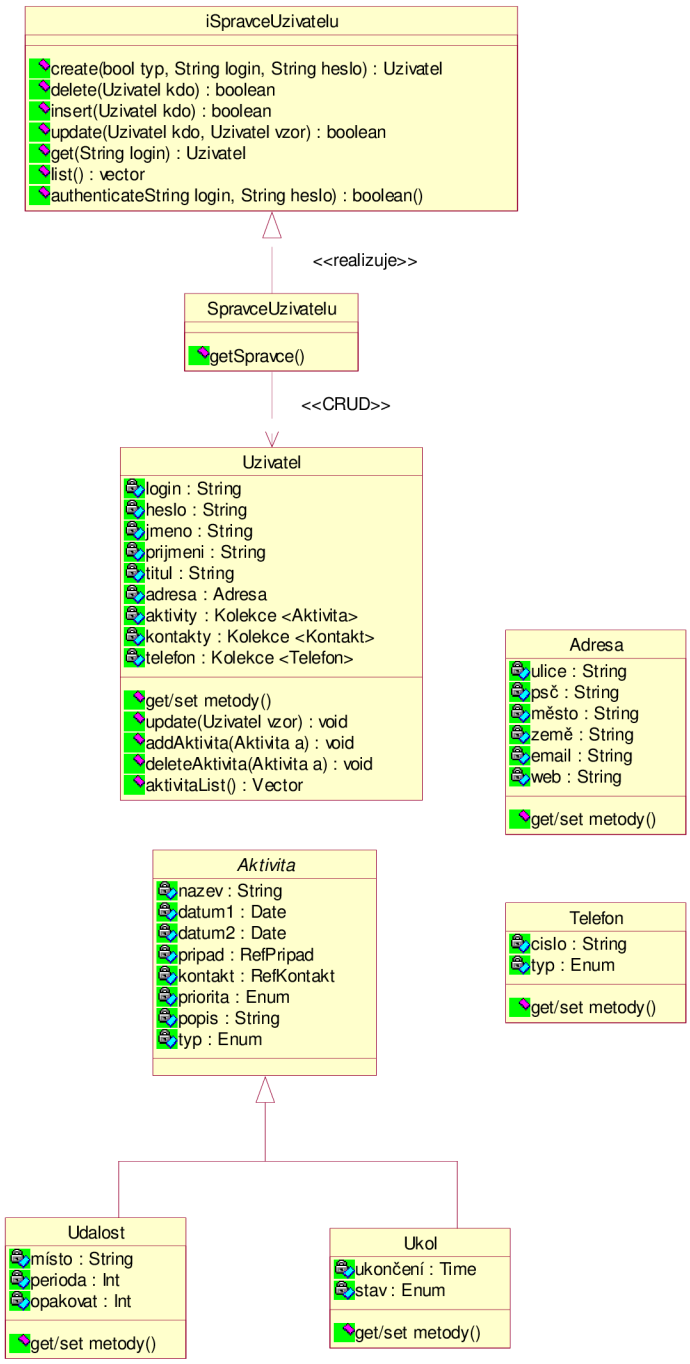
### *Použitá vstupního nastavení pro generování tříd JPublisherem*

- jpub.user = uzivatel/heslo
- jpub.url = jdbc:oracle:thin:@localhost:1521:orcl
- jpub.case = mixed
- jpub.input = vstupni_mapovaci_soubor
- jpub.methods = all
- jpub.builtintypes = jdbc
- jpub.numbertypes = objectjdbc
- jpub.usertypes = oracle

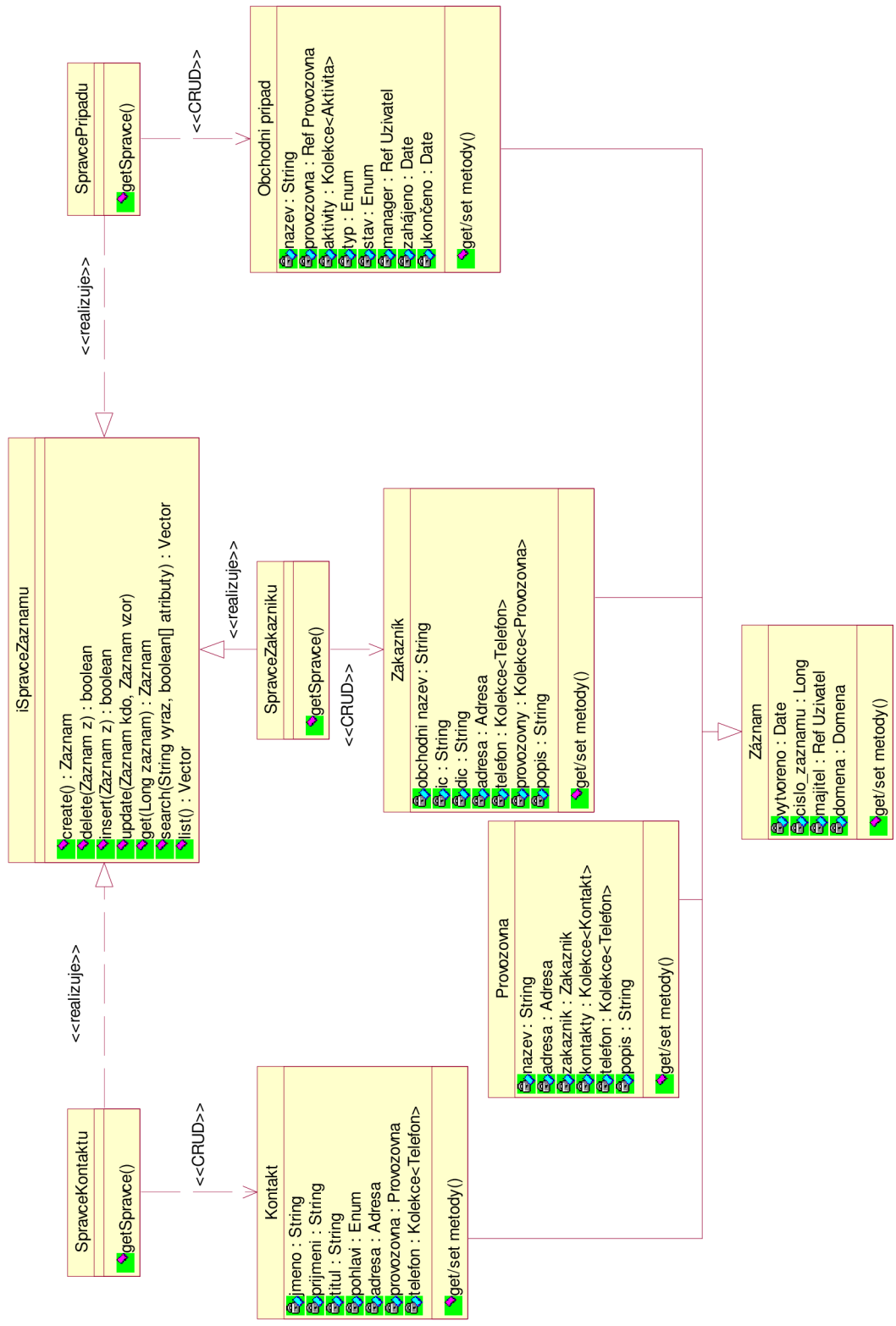
První dva řádky udávají uživatele (jeho jméno je zároveň název schématu, ve kterém je uložen model databáze) a spojení na databázi. Parametr *input* přebírá seznam objektových typů a jejich mapujících tříd. Parametr *method* je nastaven na generování všech metod (přístupových metod k atributům a vlastních implementovaných členských metod). Parametr *builtintypes* rozhoduje o způsobu mapování vestavěných nenumerických typů (char, varchar, …). Parametr *numbertypes* rozhoduje o způsobu mapování numerických typů. Parametr *usertypes* rozhoduje o variantě mapování uživatelsky definovaných typů, tedy zda mapovat pomocí SQLData nebo ORAData.

## 6.7 Aplikační model

### 6.7.1 Diagram návrhových tříd




**obr. 20 – diagram návrhových tříd 1/2**




**obr. 21 – diagram návrhových tříd 2/2**

### 6.7.2 Správci

O perzistentní třídy a operace s nimi v aplikaci se starají správci. Správci jsou odpovědní za vytváření, rušení, změny a persistenci objektů v aplikaci. Každá významná perzistentní třída má svého správce. Správci nejsou implementováni jako standardní Java třídy, nýbrž jako SQLJ třídy. Tato vlastnost jim umožňuje přímo vkládat SQL nebo PL-SQL kód do metod těchto tříd a kombinovat ho s kódem Javy. Správci při komunikaci s databází nevyužívají přímo JDBC rozhraní, ale vyšší SQLJ API, kde pro získávání dat používají takzvané iterátory (iterátory byly popsány v sekci o SQLJ).

Pokud chceme v aplikaci pracovat s persistentním objektem, čí kolekcí takových objektů, vždy k těmto objektům přistupujeme přes relevantního správce. Správci jsou v rámci aplikace globálně dostupné. Třída každého správce obsahuje statickou metodu *getSpravce()*, která vrací instanci konkrétního správce.

#### *Příklad kódu pro správce Uživatelů*

Jako příklad uvádím kus kódu třídy správce pro perzistentní třídu Uživatel. Vybraný kód bude obsahovat metodu pro získání správce, metodu pro získání kolekce objektů z databáze (nazvanou o*bjectList*) a metodu pro perzistenci objektu (nazvanou *insert*). Pro získání objektů z databáze se využívá iterátor *IterUzivatel*, kde jeho parametry jsou požadované typy na výstupu dotazu.

```
public class SpravceUzivatelu
{
    #sql public static iterator IterUzivatel (Uzivatel uzivatel, Ref ref);
    private static SpravceUzivatelu spravceUzivatelu = null;

   public static SpravceUzivatelu getSpravce() {
       If (spravceUzivatelu == null)
           spravceUzivatelu = new SpravceUzivatelu();
       return spravceUzivatelu;
    }

    public ArrayList objectList()  throws SQLException  {
      IterUzivatel it = null;
      #sql it = { select value(u) as uzivatel, ref(u) as ref  from uzivatele u };
      ArrayList uzivatele = new ArrayList();
```

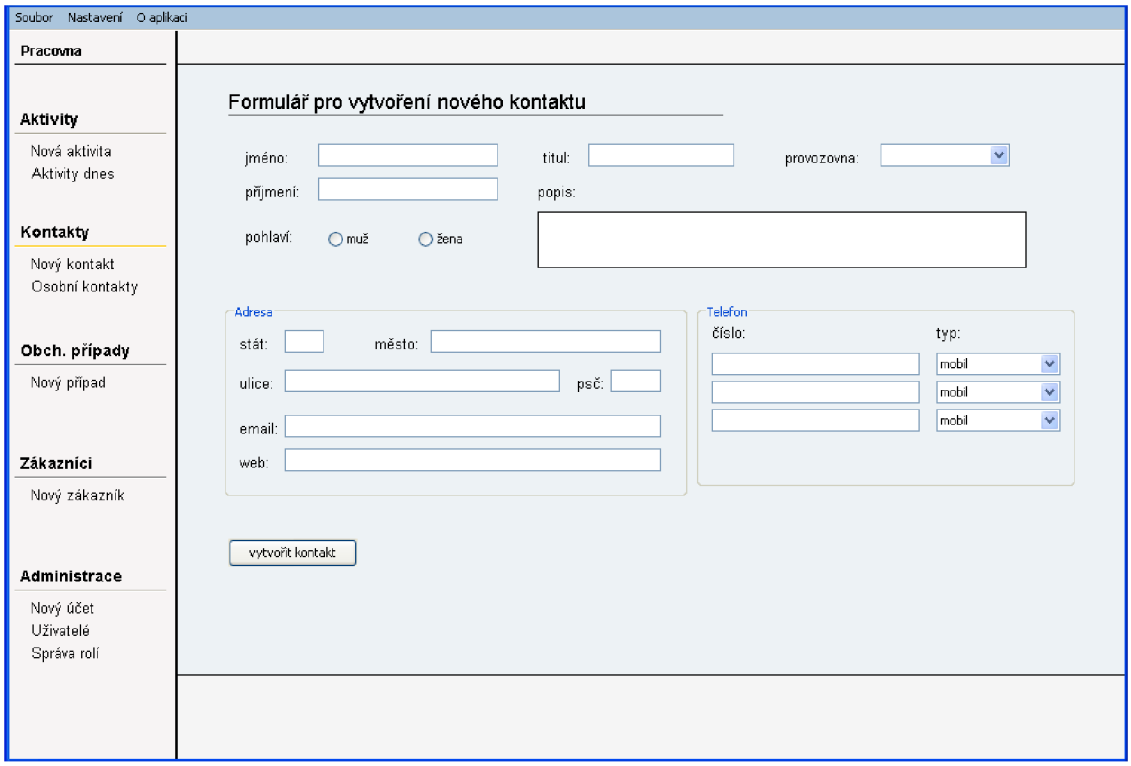
```
    while (it.next()  {
      uzivatele.add((Uzivatel) it.uzivatel());
      ((Uzivatel) uzivatele.get(uzivatele.size()-1)).setRef(it.ref());
    }
   return uzivatele;
  }

  public void insert(Uzivatel kdo)  throws SQLExceptino {
   #sql { insert into uzivatele values(:kdo)  };
  }
}
```

### 6.7.3 Uživatelské rozhraní

Aplikace byla implementována jako desktopová, využívající technologii J2SE. Grafické uživatelské rozhraní je postaveno na knihovně Swing. Aplikace se skládá z menu, horní lišty, levého panelu, dolního panelu a hlavního panelu.

Levý panel představuje navigační panel. Obsahuje tlačítka pro ovládání aplikace. Hlavní panel slouží pro zobrazování obsahu a lokálních funkcí. Dolní panel slouží pro zobrazování informací. Nejlépe vše ukážeme na obrázku.




**obr. 22 – aplikace, vytváření kontaktu**

## 6.8 Aplikační moduly

### 6.8.1 Pracovna

Pracovna je základním modulem systému. Je domovskou stránkou uživatele. Zobrazuje pohled na aktuální věci. Zobrazuje blízké aktivity. Zobrazuje probíhající obchodní případy, které uživatel vede. Obsahuje kalendář.

### 6.8.2 Zákazníci

Funkcí modulu je spravovat informace o zákaznících. Zákazníkem nejčastěji rozumějme nějakou firmu. Pro firmu se vedou její provozovny, kterých může mít několik. Samozřejmě má smysl vést pouze provozovny, se kterými se spolupracuje. Zákazník je vlastně učet, pro který se od chvíle jeho založení vedou veškeré události a případy týkající se tohoto účtu. S modulem Zákazníci jsou spojené moduly Kontakty a Obchodní případy.

### 6.8.3 Kontakty

Adresář kontaktů je základní nástroj pro vedení kontaktních osob, které zastupují zákazníky.

Přes kontakty obvykle komunikujeme se zákazníkem, takový kontakt je nejčastěji zákazníkovi přidružen, ale můžeme vytvářet i ryze osobní kontakty, které jsou skryty ostatním uživatelům.

### 6.8.4 Aktivity

Aktivity jsou záznamy, které jak název napovídá, znamenají nějakou událost nebo činnost. Aktivita se vede pro uživatele systému nebo obchodní případ. Aktivity jsou dvojího druhu.

Prvním typem aktivity jsou události. Události jsou aktivity typu schůze, telefonní hovor, nákup a podobně. Jde tedy o jednorázové činnosti, u kterých nás zajímá jejich začátek. Podle toho jsou události koncipovány. Události se mohou periodicky opakovat.

Druhým typem jsou úkoly. Úkol je činnost, která může trvat delší dobu, a může, ale nemusí být omezena termínem, do kterého má být úkol splněn.

#### *Kalendář*

Kalendář je nástroj, který obsahuje veškeré aktivity uživatele. Dá se říci, že je kontejnerem aktivit uživatele.

Kalendář poskytuje uživateli přednastavené pohledy. Například aktuální pohled na blízké aktivity, nebo například týdenní pohled zobrazuje souhrnně právě probíhající týden. Shrňme, že jde o vizuální komponentu, strukturující aktivity.

### 6.8.5 Obchodní případy

Modul, který se stará o obchodní aktivity zákazníků, pokud existuje nějaká možnost, vést se zákazníkem obchodní aktivitu, založí se zde pro ni záznam. Tento záznam je nazýván obchodním případem zákazníka. Můžeme založit i případ, který bude veden pro zatím pouze potencionálního zákazníka. Takový obchodní případ většinou označujeme za „příležitost“. Pro případ jsou vedeny, po celý jeho životní cyklus, události k němu vztažené.

#### *Obchodní případ*

Obchodním případem je myšlen záznam, který je veden pro každou obchodní aktivitu se zákazníkem, a to od jejího začátku až po konec. Obchodní případ se v čase mění podle toho, jak se zákazníkem v obchodu pokračujeme. Obchodní případ je tedy veden od stavu příležitosti, až po úspěšné ukončení, nebo ztrátu zakázky. Pro obchodní případ je určen manažer, který za něho odpovídá.

### 6.8.6 Administrace

Modul administrace slouží ke správě aplikace. Zahrnuje především správu rolí a správu uživatelů. Dále pak další nastavení systému. V systému vystupují dvě základní role. Jsou jimi manažer a administrátor. Pouze administrátor má právo využívat modul Administrace.

#### *Správa uživatelů*

Správa uživatelů složí k vytvoření nového uživatele, jeho editaci i smazání. Nově vytvořený uživatel je zanesený do databáze, ale aby mohl využívat systém, musíme mu přiřadit role s právy, které bude mít.

#### *Správa rolí*

Správa rolí je důležitá z hlediska přístupu jednotlivých uživatelů k systému. Můžeme říci, že role jsou množinami obsahující práva. Prázdná množina nevlastní žádná práva a uživatel vlastnící takovou roli nemůže v systému prakticky nic.

Přidělování práv je velmi dynamické a jednoduché. Nejdříve se vytvoří role, které chceme v systému využívat. Následně rolím přiřadíme pravá. Nakonec s danou rolí spojujeme uživatele.

# 7 Alternativní řešení

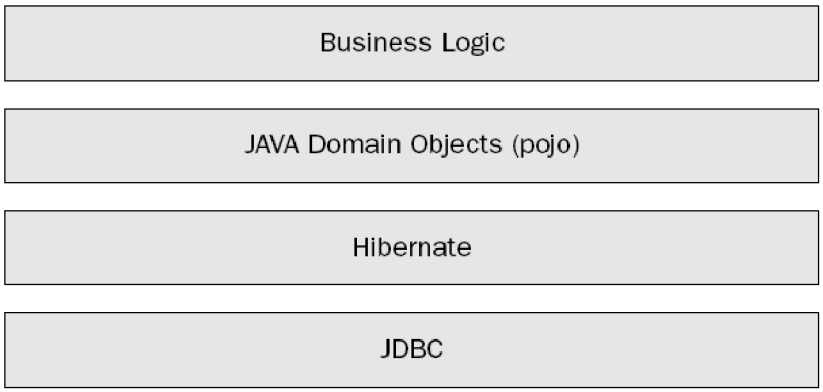
## 7.1.1 Objektové pohledy

Objektový pohled nám dovolí zpřístupňovat a manipulovat s relačními daty, jako by byla uložena v tabulce obsahující objekty. Takové řešení dává výhodu použít objektový přístup volitelně v situaci, kdy ho potřebujeme, tedy nejčastěji když chceme z databáze získat data přímo jako aplikační objekt. Jelikož samotný model dat je relační, tak jsem se tím to přístupem dále nezabýval.

## 7.1.2 Perzistentní frameworky

### *Hibernate*

Hibernate je výkonný open-source nástroj poskytující služby, zajišťující perzistenci pro objekty aplikační domény. Je to perzistentní framework pro objektově-relační mapování mezi aplikací a relační databází. Hibernate umožňuje vyjádřit databázové dotazy pomocí vlastního přenositelného rozšíření SQL, které se jmenuje HQL, ale i přímo pomocí nativního SQL. Další možností je využít objektově-orientované rozhraní Criteria. Tím, že před námi Hibernate neschovává SQL kód, dovoluje plně využít možností relačního databázového systému. Pomocí Hibernate můžeme izolovat kód pro přístup k datům od zbytku návrhu aplikace. Aplikační logika komunikuje s modelem domény. Model domény je množina Java objektů, které reprezentují abstrakci naší aplikace. Model domény poté komunikuje s Hibernate API pro ukládání či získávání částí domény.




**obr. 23 – hibernate model**

**Hibernate pro svou činnost používá následující komponenty:**

**Perzistentní Java objekt**

Je třída Javy, která reprezentuje abstrakci domény. Každý objekt bude mapovat jeden nebo více řádků tabulky v relační databázi. POJO (Plain Old Java Object) přístup poskytuje řešení, kde aplikace není pevně svázána s frameworkem.

**Konfigurační soubor**

Hibernate.cfg.xml je soubor pro konfiguraci globálních parametrů hibernate frameworku, pro objektově-relační mapování. Obsahuje informace o konkrétní databázi a informace ovlivňující perzistentní třídy.

**Mapovací soubor**

Každá třída, která má být perzistentní, potřebuje asociovat mapovací soubor, který popisuje, jakým způsobem se budou převádět vlastnosti dané třídy na relační data.

**Hibernate API**

Je rozhraní pro vykonávání přístupu k datům, hibernate překládá toto API volání do korektních příkazů SELECT, INSERT atd.

Hibernate API obsahuje veškeré zdroje potřebné pro podporu perzistence. Implementuje mapování, podporu pro transakce a dva robustní dotazovací modely. Jako hlavní z těchto rozhraní uvádím SessionFactory, Session, Transaction, Query a Criteria. Popis těchto rozhraní by zabral větší část obsahu, a jelikož tuto technologii pouze zmiňuji jako alternativu, tak se do něho nebudu pouštět a zájemce odkážu na stránky *www.hibernate.org*.

## *Persistentní řešení od společnosti Oracle*

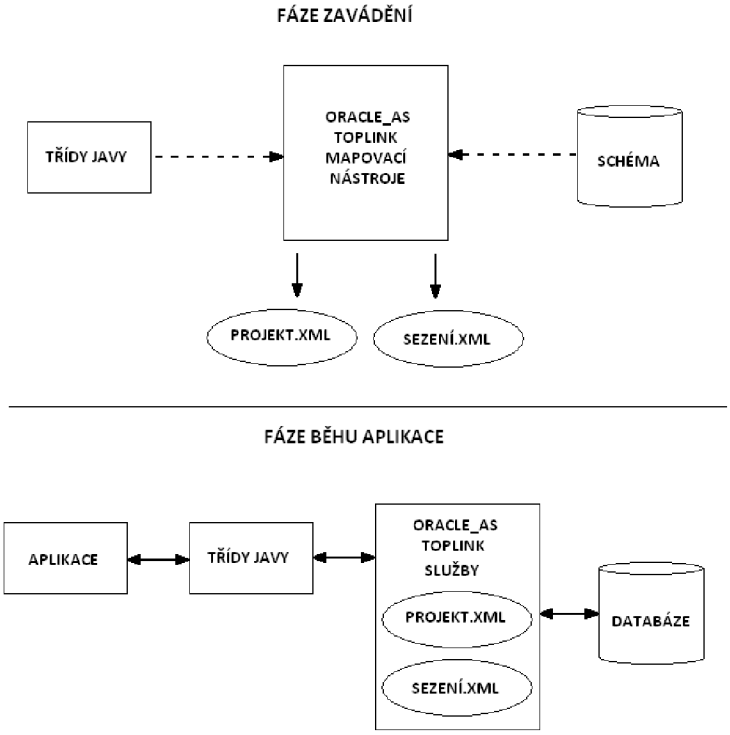
Oracle nabízí dvě možná řešení. Prvním jsou *Oracle kontejnery pro J2EE (OC4J)*, což je množina kontejnerů (servlet, EJB) a služeb (JNDI), kde kontejnery podporují jak prezentační, tak aplikační logiku na straně serveru. Tím druhým je pak *OracleAS TopLink*, který realizuje Java perzistence API (JPA). Je to databázový perzistentní framework provádějící objektově-relační mapování.

**OracleAS TopLink**

OracleAS TopLink je založený na JDBC API a umožňuje vývojářům docílit persistence Java tříd v relační databázi. Poskytuje alternativu ke spojení s databází bez psaní JDBC kódu. Používá grafický nástroj ke čtení Java tříd jednoho čí více relačních schémat a na ně umožňuje aplikovat množinu mapovacích pravidel, které generují mapovací deskriptory.

Při vytváření perzistence pro Java objekty nejdříve vytvoříme samotné mapování třídy na tabulku v databázi, následně vytvoříme informaci o sezení. Sezení reprezentuje spojení mezi aplikací

a databází, přičemž OracleAS TopLink poskytuje různé třídy sezení, optimalizované pro různé vrstvy architektury.




**obr. 24 – princip komunikace**

Za běhu aplikace pak OracleAS TopLink poskytuje služby pro realizaci perzistence. Tedy generuje JDBC kód pro přístup k objektům databáze. Výhoda tohoto přístupu, kdy využíváme perzistentní framework, je ve vztahu tabulka – třída, kdy jsou jedna od druhé abstrahovány skrze mapovací deskriptory. Volbou, mít je od sebe oddělené, také děláme jednodušší zanášení změn, ať už do schématu databáze nebo byznys objektu. Například při nutnosti změny schéma databáze pouze změníme mapování v deskriptoru a třída zůstává beze změny.

# 8 Závěr

Diplomová práce se zabývá objektovými rozšířeními v relační databázi a přístupem k nim z programového prostředí. Je zaměřena na databázový systém Oracle 10g a programové prostředí Java.

Po přečtení práce bude mít čtenář přehled o objektově-relační technologii. Měl by mít přehled o standardu SQL podporující objektová rozšíření. Získá přehled o objektové technologii, kterou poskytuje databázový server Oracle 10g. Získá přehled o přístupu k těmto technologiím z prostředí Javy a integraci Javy v samotné databázi. Z části popisující návrh a implementaci získá poznatky o specifikách takového modelu, jeho odlišností oproti relačnímu modelu a sám by měl zhodnotit, zda by pro něho taková cesta byla přínosem.

V rámci diplomové práce byla navržena a implementována aplikace, jejímž cílem je prezentovat objektový přístup a jeho přínos vůči standardnímu relačnímu přístupu. Konkrétně došlo k implementaci systému pro správu zákazníků. V průběhu implementace došlo ke kurióznímu problému, kdy byla prvotně pro projekt zvolena databázi Oracle 10g express edice. Ta ovšem, jak se později ukázalo, postrádala klíčové vlastnosti, především integraci prostředí Java do databáze. Problém se vyřešil instalací nové a bohatší verze tohoto systému.

# Literatura


[1] Groff, J., R., Weinberg, P., N. *SQL kompletní průvodce*. CP Books a.s., 2005.

[2] Stonebraker, M., Brown, P. *Objektově-relační SŘBD*. BEN, 2000.

[3] Chaudhri, A., B., Zicari, R. *Succeeding with Object Databases*. WILEY, 2000.

[4] Object Model Features. dostupné online: *http://www.objs.com/x3h7/sql3.htm*

[5] Greenwald, R., Professional Oracle Programming. WILEY, 2005.

[6] Oracle Database Application Developer's Guide - Object-Relational Features 10g Release 2 (10.2). Oracle documentation, 2005. Dostupné online: *http://download.uk.oracle.com/docs/cd/B19306_01/appdev.102/b14260.pdf*

[7] Oracle Database Java Developer's Guide 10g Release 2 (10.2) Dostupné online: *http://download.oracle.com/docs/cd/B19306_01/java.102/b14187.pdf*