Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Problematika mapování objektů z relačních databází

Petr Hotovec

Bakalářská práce

2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Petr Hotovec**
Osobní číslo:	**I16090**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Problematika mapování objektů z relačních databází**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem práce je zhodnocení objektově relačního mapování a vytvoření minimálně dvou testovacích aplikací, na kterých bude možné zjištěné výsledky analýzy ověřit. Práce bude popisovat základní teoretický rámec relačních databází s důrazem na jednotlivé komponenty databázového systému, které jsou nutné pro interakci s aplikační vrstvou. Další část práce přednese problematiku ORM a následně dojde k zhodnocení nativního přístupu k datům a při ORM přístupu. Na základě těchto teoretických výsledků bude provedeno testování pomocí dvou výstupních aplikací, jejichž výsledek přinese výhody a nevýhody pro jednotlivé přístupy. Tyto budou v závěru praktických výsledků obohaceny o navrhovaná řešení.

Rozsah pracovní zprávy:	**30-40 normostran**
Rozsah grafických prací:	**10**
Forma zpracování bakalářské práce:	**tištěná/elektronická**

Seznam doporučené literatury:

BAUER, Christian, Gavin KING a Christian BAUER. Java persistence with Hibernate. Rev. ed. London: Pearson Education [distributor], 2007. ISBN 1932394885.
LACKO, Ľuboslav. Oracle: správa, programování a použití databázového systému. 2., dopl. vyd. Brno: Computer Press, 2007. ISBN 978-802-5114-902.
OTTINGER, Joseph B., Dave MINTER a Jeff LINWOOD. Beginning Hibernate. Third edition. New York: Apress, [2014]. ISBN 978-143-0265-177.

Vedoucí bakalářské práce:	**Ing. Monika Borkovcová, Ph.D.** Katedra informačních technologií
Datum zadání bakalářské práce:	**18. září 2019**
Termín odevzdání bakalářské práce:	**13. prosince 2019**

L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

Ing. Lukáš Čegan, Ph.D.
pověřený vedením katedry

V Pardubicích dne 18. září 2019

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 12. 12. 2019

Petr Hotovec

PODĚKOVÁNÍ

Tímto bych rád poděkoval vedoucí této práce, Ing. Monice Borkovcové Ph.D., za její užitečné rady, materiály a trpělivé vedení při návrhu a zhotovování této práce.

ANOTACE

Bakalářská práce se zabývá problematikou objektově relačního mapování. Teoretická část vysvětluje problémy týkající se ORM, popisuje základní rámec databázových systémů s důrazem na části týkající se databázového připojení a popisuje použití knihoven JDBC a frameworku Hibernate. Praktická část se pak soustředí na porovnávání výkonu přístupu, při využití ORM a nativního přístupu. V rámci této části jsou implementovány dvě aplikace pro měření výkonu jednotlivých přístupů.

KLÍČOVÁ SLOVA

Objektově relační mapování, Java, nativní přístup, Hibernate, JDBC, testování

TITLE

The issue of Object-relational mapping.

ANNOTATION

The bachelor thesis deals with the issue of object-relational mapping. The theoretical part explains problems related to ORM, contains a basics of database systems with emphasis on the parts of the database connection and usage of JDBC library and Hibernate framework. The practical part focuses on comparing the performance of using ORM and native access. Within these parts, two applications are implemented to measure the performance of each approach.

KEYWORDS

Object-relational mapping, Java, native access, Hibernate, JDBC, testing

OBSAH

Seznam obrázků **10**

Seznam tabulek **11**

Seznam zdrojových kódů **12**

Seznam zkratek **13**

Úvod **14**

1 Databázové systémy **16**

1.1 Architektury databázových systémů 16

1.2 Databáze 17

1.3 Relační databáze 18

1.3.1 Relační model 18

1.3.2 Výhody relačních databází 19

1.3.3 SQL 19

1.3.4 Datová integrita 21

1.3.5 Integritní omezení 22

1.3.6 Referenční integrita 22

1.3.7 DML Triggery 23

1.3.8 Indexy 23

1.4 Zpracovávání volání SQL v databázi Oracle 24

1.4.1 Parser 24

1.4.2 Optimalizer 25

1.5 Přístup k databázi 28

1.5.1 Uživatelské rozhraní 28

1.5.2 Programové rozhraní (API) 28

2 Perzistence **29**

2.1 Data v objektově orientovaných aplikacích 29

2.2 Řešení perzistence dat v objektově orientovaných aplikacích 31

2.2.1 Serializace objektů 32

2.2.2 Využití relačních databází 32

2.3 Impedanční neshoda 33

2.3.1 Granularita ... 33
2.3.2 Podtypy (dědičnost) ... 34
2.3.3 Identita ... 35
2.3.4 Asociace ... 35
2.3.5 Datová navigace ... 36
2.4 Objektově relační mapování ... 37
2.4.1 Výhody ORM ... 37
2.4.2 Nevýhody ORM ... 38
2.4.3 Příklady ORM knihoven ... 38
3 JDBC ... 39
3.1 Rozhraní, třídy a komponenty JDBC ... 39
3.2 Ovladač JDBC ... 41
3.2.1 Typ 1 – JDBC–ODBC most ... 41
3.2.2 Typ 2 – Nativní API ... 42
3.2.3 Typ 3 – Síťový protokol ... 42
3.2.4 Typ 4 – Thin driver ... 42
3.3 Práce s JDBC ... 42
3.4 Zhodnocení přístupu ... 44
4 JPA ... 45
4.1 Hibernate ... 45
4.2 Perzistence dat v JPA ... 45
4.3 Konfigurace JPA ... 47
4.3.1 Primární klíč ... 47
4.3.2 CRUD operace ... 48
4.4 Vztahy mezi entitami v JPA ... 48
4.5 Získávání dat ... 49
4.6 Zhodnocení přístupu ... 50
5 Návrh a vývoj Programů ... 51
5.1 Vývoj ... 51
5.2 Technologie využité v aplikaci ... 51

5.2.1 Model MVC 52
5.2.2 JavaFX 53
5.2.3 Logback 53
5.3 Architektura aplikace 53
5.4 Struktura projektu 54
5.4.1 Soubor config.properties 55
5.4.2 Modul API 56
5.4.3 Program APP 58
5.4.4 Program JDBC 61
5.4.5 Program JPA 61
5.4.6 Testovací databáze 61
5.5 Problémy při implementaci 62
6 Testování 63
6.1 Metriky měření testů 64
6.2 Příprava testů 66
6.3 Test 1 67
6.3.1 Nativní přístup 67
6.3.2 Přístup ORM 69
6.3.3 Vyhodnocení testu 1 a optimalizace 70
6.4 Test 2 71
6.4.1 Nativní přístup 71
6.4.2 Přístup ORM 73
6.4.3 Vyhodnocení testu 2 a optimalizace 74
6.5 Zhodnocení testů 75
Závěr 77
Použitá literatura 78
Přílohy 81

SEZNAM OBRÁZKŮ

Obrázek 1: Diagram parsování *SQL* dotazu. 25

Obrázek 2: Příklad zapouzdření objektu *Kočka*. 30

Obrázek 3: Příklad dědění. 31

Obrázek 4: Znázornění objektů v paměti a serialozovaných objektů v souboru. 32

Obrázek 5: Diagram rozdílu mezi daty v aplikaci a v relační databázi. 33

Obrázek 6: Diagram porovnání granularity. 34

Obrázek 7: Diagram porovnání dědičnosti v různých modelech. 34

Obrázek 8: Možnosti ověření identity v jednotlivých modelech. 35

Obrázek 9: Diagram porovnání asociace. 36

Obrázek 10: Příklad porovnání datové navigace. 37

Obrázek 11: Diagram *JDBC* 39

Obrázek 12: Diagram ovladače *JDBC*. 41

Obrázek 13: Diagram připojení *JDBC*. 44

Obrázek 14: Vztahy komponent modelu *MVC*. 52

Obrázek 15: Architektura aplikace. 54

Obrázek 16: Diagram závislostí jednotlivých programů/aplikací. 55

Obrázek 17: Okno řídící aplikace. 59

Obrázek 18: Příklad grafu výsledků časů pro proběhlý test. 60

Obrázek 19: Příklad grafu výsledku paměti pro proběhlý test. 60

Obrázek 20: Relační databázový model testovací databáze. 62

Obrázek 21: Diagram scénáře jednotlivých testů. 65

SEZNAM TABULEK

Tabulka 1: Popis tříd *JDBC* 40
Tabulka 2: Popis rozhraní *Test*...57
Tabulka 3: Průměrné hodnoty výsledků testu 1. 70
Tabulka 4: Průměrné naměřené hodnoty testu 2. 74
Tabulka 5: Souhrnné časové výsledky z obou testů...76
Tabulka 6: Souhrnné paměťové výsledky testů. 76

SEZNAM ZDROJOVÝCH KÓDŮ

Zdrojový kód 1: Příklad vytvoření tabulky pro databázi *Oracle*. 20
Zdrojový kód 2: Příklad serializace objektu do souboru. 32
Zdrojový kód 3: Registrace ovladače *JDBC*. 43
Zdrojový kód 4: Registrace ovladače *JDBC* 2. 43
Zdrojový kód 5: Vytvoření připojení k databázi. 43
Zdrojový kód 6: Vytvoření objektů pro dotazy. 43
Zdrojový kód 7: Přiřazování parametrů dotazu. 43
Zdrojový kód 8: Spouštění dotazu. 44
Zdrojový kód 9: Uzavření připojení. 44
Zdrojový kód 10: Příklad *JPA* entity. 46
Zdrojový kód 11: Příklad souboru persistence.xml 47
Zdrojový kód 12: Příklad generování identity v entitě. 48
Zdrojový kód 13: Příklad jazyka *HQL* a jeho využití 48
Zdrojový kód 14: Příklad propojení tabulek v entitě *JPA* 49
Zdrojový kód 15: Příklad uživatelského nastavení programu *APP*. 56
Zdrojový kód 16: Anotace datové třídy *Objednavka*. 58
Zdrojový kód 17: Vytvoření připojení k databázi 66
Zdrojový kód 18: Příprava dotazu pro test 2. 66
Zdrojový kód 19: Příprava dotazu pro test 2 67
Zdrojový kód 20: Vytvoření EntityManageru pro *JPA* 67
Zdrojový kód 21: *SQL* dotaz testu 1. 68
Zdrojový kód 22: Zdrojový kód získávající data z databáze pro *JDBC*. 68
Zdrojový kód 23: Spouštění dotazu pro *PreparedStatement*. 68
Zdrojový kód 24: Kód pro mapování na objekty datové vrstvy pro nativní přístup 69
Zdrojový kód 25: Zdrojový kód získávající data z databáze pro *JPA*. 69
Zdrojový kód 26: Část metadat entity *Objednavka* pro test 1 70
Zdrojový kód 27: Databázové volání testu 2 pro třídu *Statement*. 71
Zdrojový kód 28: Databázové volání testu 2 pro třídu *PreparedStatement* 72
Zdrojový kód 29: Algoritmus pro mapování struktury do objektů datové vrstvy. 72
Zdrojový kód 30: Kód pro získávání dat z databáze pro test 2 s technologii *Hibernate* 73
Zdrojový kód 31: JPA anotace atributu *nadrizenyzamestnanec* třídy *Zamestnanec*............73

SEZNAM ZKRATEK

SQL	Structured Query Language
DDL	Data Definition Language
DML	Data Manipulation Language
DCL	Data Control Language
ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
I/O	Input/Output
CF	Clustering Factory
CBO	Cost Based Optimizer
RBO	Rule Based Optimizer
PL/SQL	Procedural Language for SQL
OS	Operační systém
OOP	Objektově Orientované Programování
ORM	Objektově Relační Mapování
HQL	Hibernate Query Language
MVC	Model View Controller
JDBC	Java Database Connectivity
JPA	Java Persistence API
ODBC	Open Database Connectivity
XML	Extensible Markup Language
CRUD	Create, Read, Update, Delete
JVM	Java Virtual Machine
JIT	Just In Time
SŘBD	Systém Řízení Báze Dat

ÚVOD

S rozvojem výpočetní techniky se objevila potřeba uchovávat data digitálně. Vznikla tak různá média a způsoby jak a kde tyto data uchovávat, velkého rozmachu se tak dostalo databázovým systémům, které se postupem času vyvinuly v síťové databázové systémy, na kterých se začaly provozovat relační databáze. Tyto databáze jsou poměrně výkonné, avšak objevil se další problém. Uložená data je potřeba dále zpracovávat a způsob jakým jsou ukládána, podle modelů objektově orientovaného programování, se často liší od způsobů v relačních modelech. V mnoha internetových fórech probíhá nekonečná diskuze, zdali je vhodnější k přístupu k datům v relační databázi využívat nativního přístupu, nebo objektově relačního mapování. Velmi často se lze dočíst, že nativní přístup je výkonnější a ORM v podstatě nemá žádné výhody. Oba dva přístupy mají své zastánce i odpůrce. Vzhledem k různým názorům a celkové rozdílné praxi, není jednoduché se v dané problematice řádně orientovat.

Cílem této práce je přiblížit možnosti, výhody a nevýhody jednotlivých přístupů začínajícím vývojářům a názorně ukázat výkonost jednotlivých řešení při různých situacích. Data pro porovnání budou vytvořeny sadou aplikací, které budou získány pomoc heuristických metod. Zhodnocením jednotlivých statistických dat, budou navrhnuty optimalizace, které se pokusí o zvýšení výkonu u jednotlivých přístupů. Sadu aplikací lze provozovat s jakoukoliv relační databázi, jen s minimální změnou konfigurace, bez zásahu do zdrojových kódů aplikace.

Práce je členěna do několika kapitol, první popisuje teoretický rámec databázových systémů, popisuje jejich členění, druhy a následně se věnuje relačním databázím a jejich části zodpovědné za zpracování dotazů.

Druhá kapitola teoreticky popisuje ukládání dat v objektově orientovaných aplikacích a jejich převodu do mezi aplikací a relační databází. Dále se soustřeďuje na objektově relační mapování a na problémy impedanční neshody.

Následující kapitola, přednáší popis nativního přístupu v jazyce *Java*, základní možnosti práce s knihovnami *JDBC* a popis databázových ovladačů.

Ve čtvrté kapitole, je popsán přístup objektově relačního mapování v jazyce *Java*, se specifikací *JPA*. Jsou předvedeny základní implementace, dále jejich konfigurace a základní práci s nimi.

Předposlední kapitola nabízí pohled na technologie využité v sadě testovacích aplikací, jednotlivé aplikace, jejich architekturu a problémy týkající se jejich vývoje.

V závěrečné kapitole jsou vytvořeny testy, které jsou následně spuštěny a naměřeny hodnoty pro každý přístup zvlášť. Tyto hodnoty jsou následně zhodnoceny a na základě porovnání těchto hodnot jsou navrhnuty kroky k možné optimalizaci.

1 DATABÁZOVÉ SYSTÉMY

1.1 Architektury databázových systémů

Databázové systémy se od svého vzniku v 60. letech značně vyvinuly. Od prvních hierarchických a síťových databázových systémů po objektově orientované databáze a dnes hojně využívané relační databázové systémy, NoSQL databáze až po současný trend v podobě cloudových databází. [1]

Centralizované databázové systémy

Centralizované databázové systémy představují systémy, které jsou provozovány na jednom fyzickém úložišti. Takové systémy jsou často ukládány jako jeden soubor a je vhodné je používat převážně pro aplikace malého rozsahu. Výhodou sice je, že je lze přenášet spolu s aplikací, ale velkou nevýhodou je skutečnost, že není možné do nich přistupovat z jiných míst v síti. [1]

Distribuované databázové systémy

Distribuované databázové systémy představují takovou architekturu, kdy jsou jednotlivé části uloženy na více fyzických místech a ve kterém je zpracování rozptýleno nebo replikováno mezi různými uzly v síti. Distribuované databáze mohou být homogenní nebo heterogenní. Všechna fyzická umístění v homogenním distribuovaném databázovém systému mají stejný hardware a provozují stejné operační systémy a databázové aplikace. Hardware, operační systémy nebo databázové aplikace v heterogenní distribuované databázi se mohou v jednotlivých lokalitách lišit. [1]

Cloudové databáze

Trendové cloudové databáze jsou optimalizované nebo vytvořené pro virtualizované prostředí, buď v tzv. hybridním cloudu, veřejném cloudu nebo soukromém cloudu. Cloudové databáze poskytují výhody, jako je schopnost platit za úložnou kapacitu a šířku pásma na základě použití a poskytují škálovatelnost na vyžádání spolu s vysokou dostupností. Cloudová databáze také dává podnikům příležitost podporovat obchodní aplikace při nasazení softwaru jako služba. Nevýhoda takových databází je potřeba stálého připojení k serveru přes internet a uložení dat mimo vlastní síťovou infrastrukturu. [1]

1.2 Databáze

Databáze je složena z různě uspořádaných systémů souborů dle architektury a dle typu řídícího systému. Obsahuje bázi dat, kterou lze také označit při přidání sémantiky jako soubor informací, který je uspořádán tak, aby k němu bylo možné snadno přistupovat, spravovat ho a aktualizovat. Databáze obsahují data s různou granularitou, dle svého účelu obsahuje databáze buď atomická data, nebo agregované záznamy, případně soubory. Například v případě hojně využívaných relačních databází, které jsou založeny na transakčním zpracování dat, obsahují také informace o transakcích nebo interakcemi s konkrétními protějšky. Tyto databázové systémy nabízejí tak vlastnosti ACID (atomicita, konzistence, izolace a trvanlivost), které zaručují, že data jsou konzistentní a že transakce jsou dokončené. [2]

Vzhledem k rozsáhlosti dané problematiky jsou relační databáze popsány v samostatné podkapitole 1.3.

Databáze NoSQL

Nejprve je třeba zmínit pojem *NoSQL (Not only SQL)*, jedná se o koncept databází, který nevyužívá převládající model relačních databázových systémů, místo toho jsou data ukládána jinými způsoby. Tyto způsoby však ještě nejsou standardizovány, a tak se zažil právě pojem *NoSQL*. Není však pravidlem, že takové databáze nepoužívají jazyk *SQL* (více popsáno v 1.3.3), obvykle mají vlastní jazyk, ale najdou se i výjimky, jako např.: *Apache Cassandra*, které využívají velmi podobný jazyk, např. CQL. Databáze *NoSQL* jsou užitečné pro velké sady distribuovaných dat. Jsou vhodné k nasazení v okamžiku, kdy vstupní podmínky hovoří o budoucích problémech týkajících se výkonu s velkým množstvím dat při úvaze o relačních databází. [3]

Základní kategorie *NoSQL* databází jsou:

- **Klíč-hodnota**: Data jsou většinou ukládána pouze do RAM, lze je získávat pomocí klíče, obvykle se jedná o malé, avšak velmi rychle poskytované informace, zástupcem takové databáze je například Redis. [3]
- **Sloupcové**: Záznamy jsou uchovávány ve sloupcích, oproti řádkům v relačních databázích. Apache Cassandra patří právě mezi sloupcové NoSQL databáze. [3]
- **Dokumentové**: Data jsou ukládána do dokumentů v různých formátech, nejčastěji JSON, typickým příkladem takové NoSQL databáze je MongoDB. [3]

- **Grafové**: Data jsou reprezentována jako graf objektů s relacemi mezi nimi, nejrozšířenějším zástupcem je Neo4j. [3]

Objektové databáze

Objektově orientovaná databáze je organizována spíše kolem objektů, akcí a dat než logiky. Například multimediální záznam v relační databázi může být definovatelným datovým objektem. Tyto databáze jsou vhodné pro přímé ukládání objektů vytvořených v objektově orientovaných aplikacích. [3]

1.3 Relační databáze

Relační databáze je sada formálně popsaných tabulek, ze kterých lze přistupovat nebo znovu sestavovat data mnoha různými způsoby, aniž by bylo nutné reorganizovat databázové tabulky. Pro komunikaci z databází se využívá jazyka *SQL*. Příkazy *SQL* se používají jak pro interaktivní dotazy na získání informací ze systému relační databáze, tak nastavení databázového přístupu definice dat a mnohé další. [2] [4]

1.3.1 Relační model

Každá tabulka, se v relačním modelu nazývá *relace*, obsahuje několik typů dat ve *sloupcích* nebo *atributech*. Každý řádek, také nazývaný *záznam* nebo *n-tice*, obsahuje jedinečnou instanci dat nebo klíče pro kategorii definovanou sloupci. Každá tabulka má jedinečný *primární klíč*, který identifikuje konkrétní záznam v tabulce. Vztah mezi tabulkami je řešen pomocí *cizích klíčů* ve sloupcích v tabulce, které odkazují na *primární klíč* jiné tabulky. [2] [4]

Například typická databáze záznamů o zákaznících by měla obsahovat *tabulku*, která uchovává zákazníka se *sloupci* pro jméno, adresu, telefonní číslo atd. Další *tabulka* uchovává objednávku, která se skládá například ze sloupců: produkt, zákazník, datum, prodejní cena atd. Uživatel relační databáze pak může získat pohled na databázi podle svých potřeb. Například vedoucí pobočky by mohl mít zájem o zobrazení nebo hlášení o všech zákaznících, kteří zakoupili produkty po určitém datu. Správce finančních služeb ve stejné společnosti by mohl ze stejných tabulek získat zprávu o účtech, které je třeba zaplatit. [2] [4]

Při vytváření databázového modelu mohou být definovány typy možných hodnot ve sloupcích a další omezení, která se mohou vztahovat na tyto datové hodnoty. Definují se dvě omezení týkající se integrity dat primárního a cizího klíče:

Integrita entity zajišťuje, že primární klíč v tabulce je jedinečný a hodnota není nastavena na prázdnou hodnotu (*null*). Hodnota *null*, je takové pole ve sloupci, které neobsahuje žádná data. [2] [4]

Referenční integrita vyžaduje, aby každá hodnota ve sloupci povinného cizího klíče byla nalezena v odkazující tabulce s primárními klíči tabulky, ze které pochází. [2] [4]

Příklady populárních relačních databází jsou následující:

- Oracle Database
- PostgreSQL
- MS SQL Server
- MySQL

1.3.2 Výhody relačních databází

Hlavní výhoda relačních databází spočívá v tom, že uživatelům umožňují snadno třídit a ukládat data, která mohou být později dotazována a filtrována. Relační databáze se také snadno rozšiřují. Po vytvoření databáze lze změnit, přidat nebo odebrat části databázového modelu, aniž by došlo ke změně ve většině existujících aplikací. Nutno podotknout, že některé části databázových systémů jsou proprietární (závislé na platformě) a některé jsou standardem podporující veškerého relační databáze. [5]

Výhody relačních databází jsou následující:

- **Přesnost**: data jsou uložena pouze jednou, což eliminuje deduplikaci dat.
- **Flexibilita**: komplexní dotazy jsou pro uživatele snadné.
- **Spolupráce**: více uživatelů má přístup ke stejné databázi.
- **Důvěra**: relační databázové modely jsou jasně definované a dobře srozumitelné.
- **Zabezpečení**: data v tabulkách relačního databázového systému mohou být omezena, aby umožňovala přístup pouze určité skupině uživatelů nebo jednotlivců. [5]

1.3.3 SQL

SQL je strukturovaný dotazovací jazyk, který se používá pro komunikaci s databází. Pro lepší porozumění *SQL*, je třeba přesně pochopit, k čemu se používá. *SQL* umožňuje vytvářet, spravovat a manipulovat s daty obsaženými v databázi podle potřeb a požadavků.

Definice databázových objektů

Příkazy *DDL* (*Data Definition Language*), umožňují vytvářet celou řadu databázových objektů jako jsou tabulky, schémata, uložené procedury, indexy, domény, znakové sady nebo dokonce nové databáze. Kromě vytváření (příklad vytvoření tabulky: zdrojový kód č. 1), *SQL DDL* má také schopnost odstranit existující komponenty, upravit vlastnosti databázového objektu, přejmenovat databázovou tabulku a odstranit objekty. Všechny tyto příkazy jsou součástí *SQL* kategorie DDL. [6] Do této kategorie se řadí příkazy jako CREATE, ALTER, DROP, RENAME, TRUNCATE a COMMENT.

```
CREATE TABLE 'zakaznici' (
  'id_zakaznika' INT(11) NOT NULL,
  'jmeno' VARCHAR(45) NOT NULL,
  'prijmeni' VARCHAR(45) NOT NULL,
  'datum_narozeni' DATE NULL DEFAULT NULL,
  'email' VARCHAR(45) NOT NULL,
  'adresa_id' INT(11) NOT NULL,
  PRIMARY KEY ('id_zakaznika'),
  INDEX 'fk_zakaznici_adresa_idx' ('adresa_id' ASC),
  INDEX 'prijmeni' (),
  INDEX 'jmeno' (),
  CONSTRAINT 'fk_zakaznici_adresa'
    FOREIGN KEY ('adresa_id')
    REFERENCES 'ca'.'adresa' ('id_adresa'));
```

Zdrojový kód 1: Příklad vytvoření tabulky pro databázi *Oracle*.[1]

Manipulace s daty

Jakmile jsou definovány databázové objekty, je nutné spravovat data. K tomu slouží kategorie jazyka *SQL* zvaná *Data Manipulation Language* (*DML)*. S pomocí prvků syntaxe jazyka datové manipulace (*DML*) může *SQL* vybírat, vkládat, mazat a aktualizovat datové záznamy z tabulky v jakékoli relační databázi. Na rozdíl od *DDL*, *DML* může měnit pouze data uložená v databázi, nikoli vlastnosti a parametry objektů nebo schémat v databázi. [6] Do této kategorie patří příkazy INSERT, UPDATE, DELETE, MERGE, CALL, EXPLAIN PLAN, LOCK TABLE.

[1] Vytvořeno autorem.

Řízení dat

S cílem zvýšit zabezpečení, zabránit zneužití dat a zabránit neoprávněnému přístupu k důležitým datům, *SQL* umožňuje efektivně řídit přístup k těmto datům v databázích pomocí jedné z jeho součástí, a to jazyka pro kontrolu dat *Data Control Language* (*DCL*). Syntaxí *DCL* lze snadno udělit nebo zrušit oprávnění konkrétním uživatelům pro přístup k databázi nebo její úpravu. [4] Do této kategorie jsou začleněny příkazy pracující s oprávněními, a to GRANT a REVOKE.

Datové dotazování

Pro výpis dat slouží DQL, Data Query Language. Jedná se o kategorii jazyka SQL, která umožňuje získání vztahu mezi relacemi, na základě zadaného dotazu pro výpis dat, představitelem této kategorie je příkaz SELECT. [7]

Transakční zpracování

Jelikož jsou relační databáze založeny na transakcích, je k tomuto přiřazena další kategorie jazyka SQL, a to *Transaction Control Language (TCL).* Pro řízení jednotlivých transakcí lze tak pomocí příkazů COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION potvrdit transakci a provést její úspěšné provedení, odvolat transakci, získat návratový identifikovatelný bod, který lze uvést během zpracování transakce nebo nastavit transakci s různými parametry.

1.3.4 Datová integrita

Integrita dat je zárukou přesnosti a konzistence dat v průběhu životního cyklu dat tedy od okamžiku, kdy jsou data do databáze vložena až do jejich odstranění. Integrita dat tedy znamená, že data jsou zaznamenaná tak, jak bylo zamýšleno, a že se v průběhu jejich životního cyklu nezáměrně nezměnili. Koncept je jednoduchý, praxe ale tak snadná není. Integrita dat je rozhodující součást při vytváření nebo navrhování jakéhokoli softwarového systému, který bude ukládat nebo jinak pracovat s uloženými daty. [8]

Integrita databáze znamená, že databáze vyhovuje zadaným pravidlům – integritním omezením. Nejčastějšími důvody vzniku neintegrity jsou například nedostatečná aktualizace dat, vkládání nesprávných hodnot, kde data neodpovídají skutečné realitě nebo nedodržování referenční integrity, kdy v jedné tabulce mohou zůstat informace, které se vztahovali již ke zrušenému řádku v jiné tabulce. Dobrou praxí je využití omezení (*constraints*), existuje pouze málo výjimek, kdy je nevyužívat, může se tak jednat například o dávkový import dat nebo jiné nestandartní situace, které nejsou běžné pro provoz databázového systému.

Datovou integritu lze dodržet řádným využíváním správných datových typů, tedy například číslo by mělo být deklarováno jako číslo, a ne jako řetězec, používáním integritních omezení a využíváním *triggerů*. [9]

1.3.5 Integritní omezení

Ve výchozím nastavení může sloupec obsahovat hodnoty *NULL*. Omezení *NOT NULL* vynucuje, aby sloupec nepovoloval prázdné pole. Vynucuje, aby pole vždy obsahovalo hodnotu, což znamená, že nelze vložit nový záznam bez přidání hodnoty do tohoto pole. [9]

Mezi hlavní omezení se řadí, *primární klíč*, který jednoznačně identifikuje každý řádek v tabulce. Primární klíče musí obsahovat hodnoty *UNIQUE* a nesmí obsahovat hodnoty *NULL*. Primární klíče také využívají index, který se vytváří i pro *složené primární klíče* z více sloupců. Každá tabulka může obsahovat pouze jeden primární klíč, a v tabulce tento primární klíč může být tvořen z jednoho nebo více sloupců.

UNIQUE zajišťuje, že všechny hodnoty ve sloupci jsou odlišné. Omezení *UNIQUE* a primární klíč poskytují záruku jedinečnosti sloupce nebo sady sloupců. Omezení primární klíč má automaticky *UNIQUE* omezení. Je možné však mít mnoho *UNIQUE* omezení v jedné tabulce. Toto omezení lze aplikovat na jednotlivé sloupce nebo celé skupiny (*COMPOSITE UNIQUE*). [9]

CHECK constraint se používá k omezení rozsahu hodnot, které lze umístit do sloupce. Pokud se definuje *CHECK* omezení v tabulce, lze omezit hodnoty v určitých sloupcích na základě hodnot v jiných sloupcích v řádku. [9]

Omezení lze definovat při vytváření databázových tabulek, nebo při jejich modifikaci pomocí příkazů *CREATE* a *ALTER TABLE*. [9]

Při návrhu databáze je vhodné si také uvědomit, že každé integritní omezení může negativně ovlivňovat výkon celého databázového systému na práci s jednotlivými řádky. Nejméně náročné omezení je *NOT NULL* poté *UNIQUE*, primární klíče, cizí klíče, *CHECK* a nejnáročnější na provoz jsou spouště neboli *triggery*. [9]

1.3.6 Referenční integrita

Referenční integrita je databázový koncept, který se používá k vytváření a udržování logických vztahů mezi tabulkami, aby nedošlo k znehodnocení dat. Jedná se o velmi užitečnou a důležitou součást *SŘBD*. Referenční integrita se obvykle skládá z kombinace primárního klíče a cizího klíče. Hlavním konceptem referenční integrity je to, že neumožňuje přidávat žádný záznam

do tabulky, která obsahuje cizí klíč, pokud referencovaná tabulka neobsahuje odpovídající primární klíč, tedy referenci. Pokud je odstraněn jakýkoli záznam v odkazované tabulce (tj. tabulka, která obsahuje primární klíč), budou pro referenční integritu odstraněny všechny odpovídající záznamy v referencované tabulce. Každá reference mezi tabulkami podléhá určitým pravidlům. Úkolem referenční integrity je tyto pravidla dodržovat a udržovat, lze toho docílit pomocí referenčních omezení *reference integrity constraints*. [9]

Nejběžnější referenční omezení je *cizí klíč* používaný k propojení dvou tabulek dohromady. Cizí klíč je sloupec (nebo kolekce sloupců) v jedné tabulce, které odkazuje na primární klíč v jiné tabulce. Tabulka obsahující cizí klíč se nazývá závislá tabulka a tabulka obsahující kandidátní klíč se nazývá referencovaná tabulka. [9]

1.3.7 DML Triggery

Spoušť (*trigger*) je speciální procedura uložená v databázovém systému, která se spustí, když se v databázi vyskytne určitá událost. Většina *triggerů* je definována ke spuštění, když se provedou změny v datech tabulky. Spouště lze definovat tak, aby se spouštěly namísto nebo po akcích *DML*, jako je *INSERT*, *UPDATE* a *DELETE*. Spouště pomáhají zajistit, aby byla zachována integrita dat a poskytují možnost vytvořit vlastní scénáře těchto akcí.

1.3.8 Indexy

Databázový index je datová struktura, která zpravidla zvyšuje rychlost operací v tabulce. Pokaždé, když aplikace spustí databázový dotaz, prohledá databáze všechny řádky v tabulce a najde ty, které odpovídají zadanému požadavku. S růstem databázových tabulek je třeba pokaždé zkontrolovat rostoucí počet řádků, což může zpomalit celkový výkon databáze, a tedy i aplikace. Indexy tento problém řeší tím, že vezmou data ze sloupce v tabulce a obvykle je uloží abecedně na samostatném místě zvaném index, jinak záleží na typu indexu. Lze tak číst data i bez přístupu do tabulky, což výrazně snižuje počet *I/O operací* a z povahy jejich struktury efektivně vynucuje omezení *UNIQUE*.

Indexy se vytváří podle předem daných schémat, jedno z nejvíce využívaných je schéma *B-tree*. Uvnitř něho se nachází vždy *root block, branch block* a *leaf block. Root block* je základní blok indexu, *branch block* ukládá klíč a pointer na první *leaf block* a *leaf block* ukládá klíč, identifikátor řádku v tabulce a ukazatel na další a předchozí *leaf block*. Indexy také mohou být setříděny podle vzdálenosti od *root blocku* nebo stylem zleva doprava.

Indexy lze statisticky hodnotit, jednou ze zásadních vlastností každého indexu je *B-level*, označuje výšku indexu mezi *root block* a *leaf*, definuje minimální počet „*getů*“ potřebných k projití indexu. Hloubka indexu vždy činí *B-level+1*. Další statistika hýbající s cenou přístupu je *clustering factory* (*CF*), představuje, jak dobře jsou setříděny řádky v tabulce vůči *indexu*. Tato statistika používá *CBO*, *cost base optimiser*, který je pro svou obsáhlost popsán v kapitole 1.4.2, pro určení ceny, s tím se ale také podotýká problém s náročností výpočtu.

K indexům lze přistupovat pomocí základních metod, v první řadě se jedná o *unique scan*, ten je jeden z nejefektivnějších přístupů k datům, jelikož využívá, jak již název napovídá omezení *UNIQUE* a může tedy vracet pouze jeden řádek. Další metoda se nazývá *range scan*, jedná se o běžnou operaci pro přístup k výběru hodnot, taková data poté vrací vzestupně setříděna. Pokud jsou zastoupeny řádky s identickými hodnotami, tak jsou dále seřazeny dle *rowid*. V případě, kdy nechceme index nijak omezovat, vzniká *full scan*, ten se chová identicky jako *range scan*, avšak je bez jakéhokoliv omezení. Pokud jsou všechny sloupce dotazu součástí indexu, lze využít vylepšenou variantu fast full *scan*, u kterého není vůbec potřeba přístup do databázové tabulky, avšak výsledná data nejsou nijak seřazena. [10]

Náročnost čtení a zápisu je u indexů důležitou vlastností, jelikož se jedná o *I/O operace*. V případě využití *CBO* je tedy více než vhodné mít naprosto přesné statistiky. Nejnáročnější je samozřejmě zápis, může se tak jednat o dotazy *INSERT, DELETE* a *UPDATE*. [10]

Mezi nejčastější chyby patří typicky příliš velké množství indexů, kdy je sice velmi rychlé získat z databáze data, avšak modifikace stávajících je velmi náročná. Často tak jsou indexovány položky, které indexovány být nemusí. Další problém se týká špatného pořadí jednotlivých sloupců a zbytečného využívání operátoru *OR*, optimalizer poté musí použít špatnou metodu přístupu. Většina problémů s indexy vychází ze špatných plánů, kdy jsou používány zbytečně *full scany*. [10]

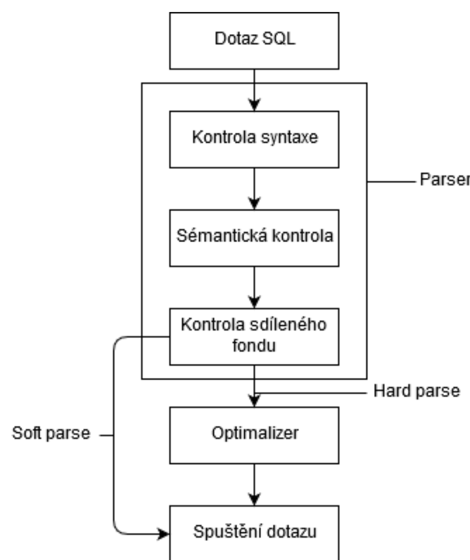
1.4 Zpracovávání volání SQL v databázi Oracle

1.4.1 Parser

Během parsového volání provede databáze následující kontroly – Kontrola syntaxe, Sémantická kontrola a Kontrola sdíleného fondu po převodu dotazu do relační algebry.

1. **Kontrola syntaxe –** kontroluje, zdali jsou jednotlivá klíčová slova zapsána správně a zdali pořadí jednotlivých slov je správné, součástí kontroly syntaxe je i lexikální analýza.
2. **Sémantická kontrola –** kontroluje, zda je výrok smysluplný nebo ne. Příklad: dotaz obsahuje název tabulky, který neexistuje.
3. **Kontrola sdíleného fondu –** Každý dotaz má hash kód během jeho provádění. Tato kontrola tedy určuje existenci zapsaného hashového kódu ve sdílené knihovně, pokud existuje kód ve sdíleném fondu, pak databáze neprovede další kroky pro optimalizaci a provedení.

Pokud existuje nový dotaz a jeho hash kód neexistuje v *library cache*, pak tento dotaz musí projít z dalších kroků známých jako *hard parse*, pokud existuje hash kód, pak dotaz neprochází dalšími kroky. Jednoduše přejde přímo ke spuštění (viz podrobný diagram obr. č. 1). Toto je známé jako *soft parse*. [10]



Obrázek 1: Diagram parsování *SQL* dotazu.[2]

1.4.2 Optimalizer

Během fáze optimalizace musí databáze provést *hard parse* alespoň jednoho jedinečného příkazu *DML* a během této analýzy provést optimalizaci. Relační databáze nikdy

---

[2] Vytvořenou autorem.

neoptimalizuje DDL, pokud neobsahuje komponentu *DML*, jako je poddotaz, který vyžaduje optimalizaci. [10]

V běžně využívaných databázích, jako je *Oracle*, jsou obvykle dva optimalizátory. Oba vytvářejí plány na provádění volání *SQL*. Optimalizátor založený na pravidlech (*role based optimiser – RBO*) je původní optimalizátor pro databázi Oracle. Postupem času, jak rostla velikost databáze, však vznikl optimalizátor založený na nákladech (*CBO – cost based optimiser*), který se stal více populární. [10]

- *RBO* se řídí řadou pravidel založených většinou na indexech a typech indexů.
- *CBO* používá statistiku a matematiku k tomu, aby odhadl při nejnižší cenu na vykonání dotazu. *CBO* zpracovává několik iterací plánů (nazývaných *permutace*), poté vybere ten s nejnižšími celkovými náklady.[11]

CBO

Databáze předává *CBO* volání *SQL* a následně získává zpět čísla nákladů a *execution plan*. *CBO* zamíchá kód a vyzkouší různé kombinace tabulek a predikátů klauzulí ve snaze najít nejnižší celkové náklady. *SQL* s nejnižšími náklady je považováno za nejrychlejší provádění *SQL*. Výsledné ceny jednotlivých iterací se ukládají do sdíleného prostoru *library cache*. [11]

Hard parse pomocí *CBO* trvá déle, takže je dobrým zvykem používat vazebné proměnné a jiné kódovací techniky, aby databázový systém mohl najít *SQL* z předchozích spuštění nežli provádět hard parse pro každý odeslaný dotaz zvlášť. Číslo ceny příkazu je celková cena zpracovávaného *SQL*. Toto číslo celkových nákladů se porovnává s čísly nákladů z předchozího permutací. Pokud jsou nové celkové náklady nižší, předchozí permutace se zahodí a současná permutace se stane „*nejnižší cenou*“ *SQL*. Tento proces se opakuje, dokud není databázový systém přesvědčen, že vyzkoušel všechny kombinace tabulek, kde jsou dostupné predikáty klauzule včetně vyzkoušení všech typů spojení. [11]

Mezi typy transformací, které *CBO* provádí, patří:

- Sloučení komplexních pohledů (převádí pohledy na spojení)
- Testování poddotazu (převádí poddotazy na vložená zobrazení)
- Vytvoření poddotazů z klauzule *WHERE*

CBO využívá ke svému běhu podkladová data, čím přesnější jsou podkladová data, tím přesnější bude určení optimálního plánu. Říkají tak například, kolik řádků obsahuje tabulka, zdali tabulka využívá indexy apod. Podkladová data lze souhrnně označovat jako statistiky. Pro každý sloupec tabulky se zjišťuje největší a nejnižší hodnota, hustota nebo počet hodnot *NULL*. Krom podkladových dat, se využívají i systémové statistiky, které zahrnují údaje

o hardwaru a operačním systému, rychlosti čtení jednotlivých bloků, hospodaření s operační pamětí apod. Jeden ze základních typů statistik je frekvenční histogram, určuje rozložení dat ve sloupci a může být pro *CBO* velice prospěšný při stanovení ceny. [11]

Execution plan

Každý krok *execution plane* vytváří sadu výsledků, která je předána do dalšího kroku plánu. Tyto sady výsledků jsou dočasné tabulky, které nejsou viditelné a nejsou ani indexovány. Sada výsledků, která dorazí na příkaz id 0, je konečným výsledkem provádění *SQL* dotazu a řádky zde jsou pak předány do oblasti kurzorů v globální oblasti programu uživatele. Jejich aplikace je upozorněna na to, že *SQL* je prováděno a existují řádky, které je třeba řešit. [12]

- **Estimated Execution Plan** je seznam operací, které je potřeba vykonat. Tento plán je elý postaven na odhadech z dostupných metadat tabulek. Udávané počty řádků jsou pouze vypočteny na základě dostupných statistik. Plán fakticky není proveden, proto se počty odhadnutých řádků mohou lišit od počtů skutečných řádků, které dotaz vrátí při skutečném spuštění. [12]
- **Actual Execution Plan** je naopak plán, který byl získán při spuštění dotazu. Počty řádků nejsou vypočteny, ale změřeny při provádění operací. [12]

Library cache

Library cache se někdy označuje jako „*sdílená oblast SQL*“. Jak název napovídá, sdílená oblast SQL se používá k uchovávání a zpracování příkazů *SQL* a kódu *PL / SQL*. [13]

Do *library cache* jsou zahrnuty všechny sdílené struktury. To zahrnuje následující:

- Zdrojové příkazy *SQL* nebo *PL / SQL* (*SQL*, uložené procedury, balíčky)
- Strom analýzy pro příkazy *SQL*
- Kurzory pro příkazy *SQL*
- Analyzovat stromy pro příkazy *SQL*
- *Execution plan* každého příkazu *SQL*

Objekty v *library cache* fungují stejně jako všechny ostatní vyrovnávací paměti a používají algoritmus naposledy použitých SQL dotazů. Objekty stárnou v *library cache* stejným způsobem, jakým data blokují mezipamět datových vyrovnávacích pamětí. *SQL* se znovu používá v *library cache* deklarováním soukromého kurzoru pro každou úlohu. Tímto způsobem může mnoho úkolů provádět stejný dotaz *SQL*, ale s různými hostitelskými proměnnými a různými výsledky. [13]

1.5 Přístup k databázi

Tato kapitola popisuje možnosti přístupu správce, programátora nebo aplikace k databázovému systému.

1.5.1 Uživatelské rozhraní

Uživatelské rozhraní databázového systému, je určeno převážně pro zjednodušení práce správce databázového systému, poskytuje mu nástroje, kterými rychle a přehledně může takový systém spravovat. Dokáže generovat jednotlivé *SQL DDL* dotazy z grafického návrháře databázového modelu, automaticky jej nahrát do systému, upravovat data uložená v systému, přidělovat oprávnění jednotlivým uživatelům a mnoho dalších správcovských funkcí, potřebných k provozu systému. Příklady uživatelských rozhraní:

- phpMyAdmin – MySQL
- Oracle SQL Developer – Oracle Database
- SQL Server Management Studio – MS SQL Server

1.5.2 Programové rozhraní (API)

Toto rozhraní se označuje také jako ovladač, obvykle žádné grafické rozhraní nemá. Obsahuje knihovnu nízkoúrovňových volání, které může programátor využívat a zprostředkovává tak připojení k databázi. Nutno podotknout, že i uživatelská rozhraní využívají na pozadí programové. *SQL* příkazy, může programátor ručně napsat, anebo se mohou za běhu programu sami generovat. Využívá se také pro svazování argumentů s parametry dotazů, spouštění dotazů a procházení jejich výsledků. Příklady ovladačů pro různé jazyky jsou následující: [14]

- Java
  - OJDBC – Oracle Database (popsáno v kapitole 3)
  - MySQL Connector – MySQL
- C#
  - OracleConnection – Oracle Database
  - SQL Client – MS SQL Server

2 PERZISTENCE

Perzistence je schopnost objektu přežít životnost procesu *OS*, ve kterém se nachází. Perzistence je relevantní pro objekty s vnitřním stavem. Stav musí být zachován mezi zničením objektu a opětovném vytvoření. Následující kapitola je zaměřena na popis perzistence v různých typech uložišť a problémech při jejich vzájemném zpracovávání. [15]

2.1 Data v objektově orientovaných aplikacích

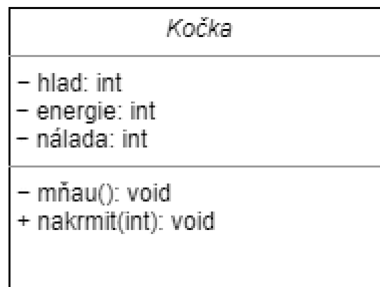
Objektově orientované programování (*OOP*) je model programovacího jazyka, ve kterém jsou programy uspořádány spíše podle dat nebo objektů než funkcí a logiky. Objekt lze definovat jako datové pole, které má jedinečné atributy a chování. Příklady objektů se mohou pohybovat od fyzických entit, jako je člověk, který je popsán vlastnostmi, jako je jméno a adresa, až po malé počítačové programy. [15]

Jakmile je objekt znám, je zobecněn jako třída objektů, která definuje typ dat a všechny logické sekvence, které s ním mohou manipulovat. Každá odlišná logická posloupnost je známa jako metoda a objekty mohou komunikovat s dobře definovanými rozhraními nazývanými zprávy. Mezi výhody *OOP* patří opakovaná použitelnost kódu, škálovatelnost a efektivita. Objektově orientované programování je založeno na následujících principech. [15]

Zapouzdření

Zapouzdření je dosaženo, když každý objekt udržuje své soukromí v rámci třídy. Jiné objekty nemají přímý přístup ke stavu daného objektu. Místo toho mohou volat pouze seznam veřejných metod. Objekt tedy spravuje svůj vlastní stav pomocí metod a žádný jiný objekt nemůže měnit jeho stav, pokud to není výslovně dovoleno. Pokud se má s objektem komunikovat, měli by se použít poskytnuté metody. [16]

Příkladem může být objekt *kočka*, ta se zapouzdří tak že veškerá logika kočky bude zapouzdřena ve třídě stejného názvu, příklad na obrázku č. 2. [17]



Obrázek 2: Příklad zapouzdření objektu *Kočka*.[3]

Kočku lze nakrmit. Ale nelze přímo změnit, jak hladová kočka je. Zde jsou stavem kočky soukromé proměnné *nálada*, *hlad* a *energie*. Má také soukromou metodu *mňau*(). Objekt kočky může tuto metodu zavolat kdykoliv chce, ostatní objekty však nemohou kočce říct, kdy má mňoukat. [17]

Abstrakce

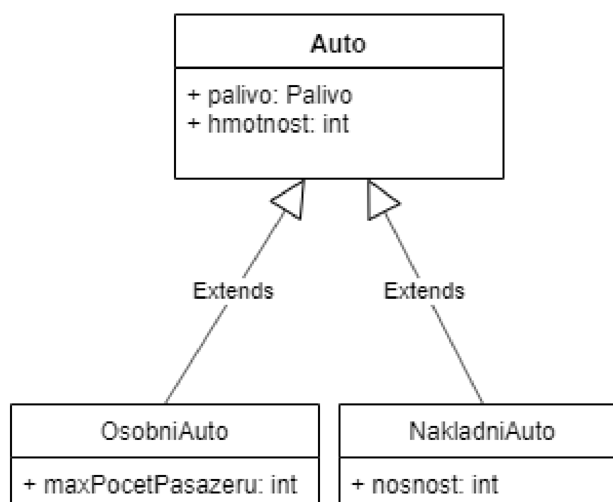
Abstrakci lze považovat za přirozené rozšíření zapouzdření. V objektově orientovaném designu jsou programy často extrémně velké. A samostatné objekty spolu často komunikují. Spravovat takový kód je poté velmi obtížné.

Abstrakce je koncept, jehož cílem je tento problém zmírnit. Použití abstrakce znamená, že každý objekt by měl nabízet pouze mechanismus na vysoké úrovni. Tento mechanismus by měl skrývat vnitřní implementační podrobnosti. Měla by se odhalit pouze operace relevantní pro ostatní objekty. Příkladem může být telefon, kde uživatel používá pouze tlačítka, ale neřeší, jak uvnitř jednotlivé obvody fungují. [17]

Dědičnost

Objekty jsou často velmi podobné. Sdílejí společnou logiku. Ale nejsou úplně stejné. Je vhodné tedy znovu použít společnou logiku a extrahovat jedinečnou logiku do samostatné třídy. Jedním ze způsobů, jak toho dosáhnout, je dědičnost. To znamená, že se vytvoří (podřízená) třída odvozením z jiné (nadřazené) třídy. Tímto způsobem se vytváří hierarchie. Podřízená třída znovu používá všechny vlastnosti a metody nadřazené třídy (společná část) a může implementovat svou vlastní (jedinečná část). Příklad třídy *Auto* je na obrázku č. 3. [17]

[3] Vytvořeno autorem.



Obrázek 3: Příklad dědění.[4]

Objekt typu *OsobniAuto*, získává veškeré vlastnosti z objektu *Auto*, avšak nesdílí své jedinečné vlastnosti s objektem *NakladniAuto*.

Polymorfismus

Polymorfismus je schopnost objektu nabrat mnoho podob. Nejběžnější použití polymorfismu v *OOP* nastává, když je odkaz na nadřazenou třídu použit k označení objektu podřízené třídy. Dovoluje pracovat s potomkem objektu jako jeho rodiče, tedy pro příklad z obrázku č. 3 by mohlo platit, že je deklarována proměnná typu *Auto*, avšak ve skutečnosti by se uvnitř nacházel objekt *OsobniAuto*. Tento způsob dovoluje generalizovat veškeré poděděné objekty a vytvořit z nich například seznam jednoho datového typu. Bez této vlastnosti by byl programátor nucen vytvořit dvě datové struktury, pro každou poděděnou třídu zvlášť. [17]

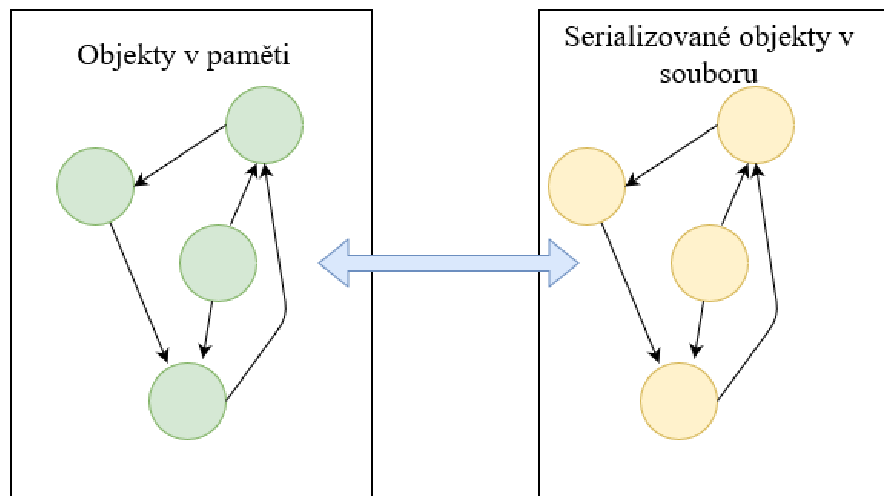
2.2 Řešení perzistence dat v objektově orientovaných aplikacích

Jedním z nejdůležitějších úkolů, které aplikace musí provádět, je ukládání a obnova dat. Aplikace musí rozhodnout, které objekty lze učinit perzistentními. Volba metody perzistence je důležitou součástí návrhu aplikace. Následující kapitola se bude této problematice věnovat.

[4] Vytvořeno autorem.

2.2.1 Serializace objektů

Jednoduchá metoda persistence, která umožňuje programu číst nebo zapisovat celý objekt do proudu bajtů. Umožňuje kódování objektů do proudu vhodného pro streamování do souboru na disku nebo přes síť. Každá *serializovatelná* třída musí implementovat speciální rozhraní (např.: *java.io.Serializable*), které nedeklaruje žádné metody, ale má modifikátory přístupu a mutátory pro své atributy. Příklad *serializace* objektů do souboru v jazyce Java je uveden ve Zdrojový kód 2. Podoba jednotlivých objekt v paměti a v souboru je znázorněna na následujícím obrázku č. 4. [3]



Obrázek 4: Znázornění objektů v paměti a serialozovaných objektů v souboru. [5]

```
File file = new File("teams_serialize.ser");
String fullPath = file.getAbsolutePath();
fos = new FileOutputStream(fullPath);
out = new ObjectOutputStream(fos);
out.writeObject(t1);
```

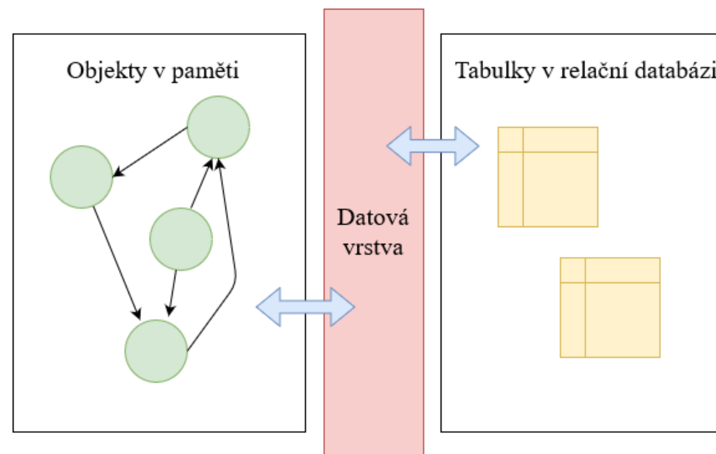
Zdrojový kód 2: Příklad serializace objektu do souboru.

2.2.2 Využití relačních databází

Většina aplikací typu *klient-server* používá relační databázové systémy jako svoje datové úložiště. Při vývoji pomocí objektově orientovaného programovacího jazyka, musí být objekty

[5] Vytvořeno autorem.

mapovány do tabulek v databázi a naopak. Aplikace tedy ve většině případů vyžadují použití příkazů *SQL* vložených do jiného programovacího jazyka. V tuto chvíli vzniká problém nazývaný impedanční neshoda. Nesoulad obou technologií je vidět na následujícím obrázku č. 5. [5]



Obrázek 5: Diagram rozdílu mezi daty v aplikaci a v relační databázi.[6]

2.3 Impedanční neshoda

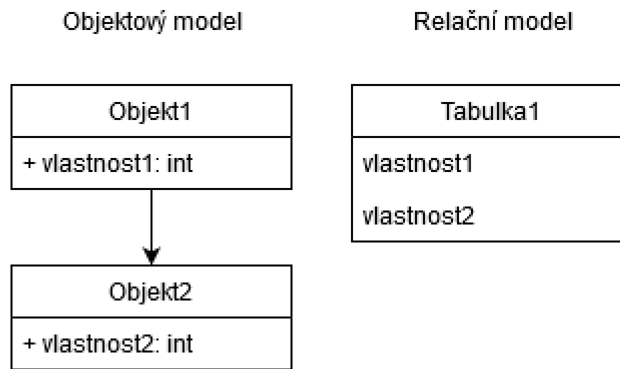
Relační databázové systémy představují data v tabulkovém formátu, zatímco objektově orientované jazyky, jako je Java, je představují jako propojený graf objektů. Mapování z objektů do tabulek, a naopak tedy může být obtížné, zejména pokud model obsahuje složité struktury tříd, velké nestrukturované objekty nebo dědičnost objektů. Výsledné tabulky mohou ukládat taková data neefektivně nebo přístup k nim může být neefektivní. Načítání a ukládání grafů objektů pomocí tabulkové relační databáze nás vystavuje 5 neshodám. [5]

2.3.1 Granularita

Objektový model, který má více tříd, než počet odpovídajících tabulek v databázovém modelu tedy stav, kdy objektový model je granulárnější než relační model. Dochází k problému, kdy nelze mapovat záznamy jedné databázové tabulky do jedné entity, jelikož jsou data rozprostřena do více entit. Například objekt *Objekt1* dle následujícího diagramu: [18]

---

[6] Vytvořeno autorem.

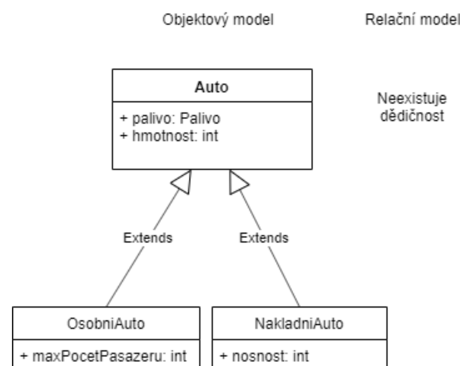


Obrázek 6: Diagram porovnání granularity. [7]

Zde je vidět problém granularity, v relačním modelu je v praxi běžné mít jednu tabulku odpovídající entitě *Objekt1*, avšak v objektovém modelu, je vhodnější rozložit části týkající se adresy do vlastních entit.

2.3.2 Podtypy (dědičnost)

Dědičnost je, jak již bylo řečeno přirozené paradigma v objektově orientovaných programovacích jazycích. Jedná se o možnost vytvoření potomků jednotlivých objektů. Relační databázové systémy však nedefinují nic podobného. Příkladem může být entita *auto* na obr. č. 7. [18]



Obrázek 7: Diagram porovnání dědičnosti v různých modelech. [8]

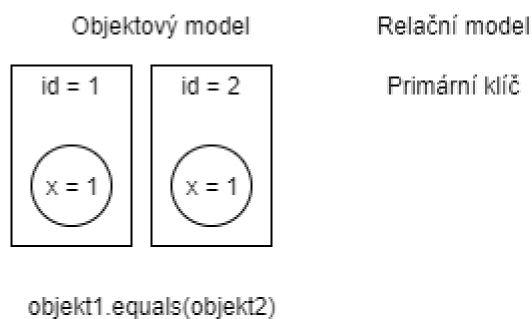
V objektovém modelu lze definovat dědičnost. V relačním modelu však tento problém řešit lze jen obtížně, a to buď vytvořením dvou tabulek pro každou entitu (*osobní auto*, *nákladní*

[7] Vytvořeno autorem.
[8] Vytvořeno autorem.

auto), v takovém případě hovoříme o pojmu specializace. Nebo vytvoření jedné tabulky, která obsahuje všechny vlastnosti všech entit, kde dochází k dědění, v tuto chvíli se hovoří o generalizaci.

2.3.3 Identita

Relační databázové systémy přesně definují jeden pojem „stejnost" a to primárním klíčem. Objektově orientované jazyk však definují jak identitu objektů pomocí referencí (a==b) tak rovnost objektů založenou na základě atributů objektu (metoda *equals*). Problém je znázorněn na následujícím diagramu obr. č. 8. [18]



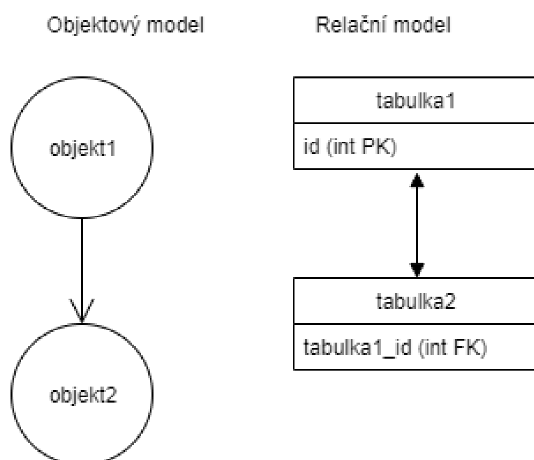
Obrázek 8: Možnosti ověření identity v jednotlivých modelech. [9]

Objekty v objektovém modelu lze jednoznačně identifikovat podle vlastnosti *x*, ta je však odlišná od *id*, naproti tomu v relačním modelu by se tyto objekty identifikovali pouze pomocí sloupce s primárním klíčem, tedy id. V každém modelu tedy vzniká naprosto odlišný výsledek.

2.3.4 Asociace

Asociace jsou v objektově orientovaných jazycích zastoupena jako jednosměrné odkazy, zatímco relační databázové systémy používají cizí klíče. Pokud jsou v objektově orientovaném jazyce potřeba obousměrné vztahy, musí být asociace definována dvakrát. Příklad asociace objektů je na obr. č. 9. [18]

[9] Vytvořeno autorem.



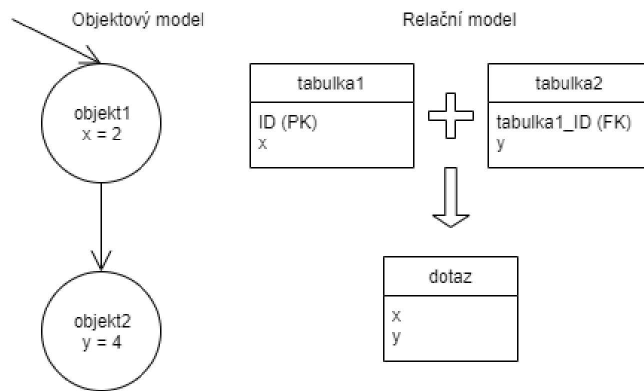
Obrázek 9: Diagram porovnání asociace.[10]

V objektovém modelu, je asociace vždy jednosměrná, *OOP* principy neumožňují zpětně dohledat, který objekt získal na *objekt2* referenci. Zatímco u tabulek v relační databázi, lze pomocí cizího klíče dohledat záznam v *tabulce1* s cizím klíčem z *tabulky2*, stejně tak lze v tabulce2 najít záznam s primárním klíčem *tabulky1*.

2.3.5 Datová navigace

Způsob, jakým se přistupuje k datům v objektově orientovaném jazyce, se zásadně liší od způsobu, jakým se to dělá v relační databázi. V objektově orientovaném jazyce se přechází z jedné asociace do druhého. Tento způsob získávání dat z relační databáze není efektivní. Obvykle je vhodné minimalizovat počet dotazů *SQL*, a tak načíst několik entit pomocí *JOIN* a vybrat cílové entity než procházet graf objektů. Na následujícím příkladu (obr. č. 10) je zapotřebí získat data x a y, v objektovém modelu, se musí sekvenčně získávat data, zatímco v relačním modelu je lze získat jedním dotazem naráz. [18]

[10] Vytvořeno autorem.



Obrázek 10: Příklad porovnání datové navigace.[11]

2.4 Objektově relační mapování

Objektově relační mapování (*ORM*) je technika, která umožňuje dotazovat a manipulovat s daty z databáze pomocí objektově orientovaného paradigmatu. Když je zmíněn *ORM,* většina lidí odkazuje na knihovnu, která implementuje techniku mapování objektových vztahů, tedy frázi „*ORM*". Knihovna *ORM* je zcela běžná knihovna napsaná ve zvoleném jazyce, která zapouzdřuje kód potřebný pro manipulaci s daty, takže už není používán jazyk *SQL*, ale dochází ke komunikaci přímo s objektem ve stejném jazyce, jaký je použit. O mechanickou část se stará automaticky knihovna *ORM*. Vrstva ORM umožňuje vývojářům objektově orientovaných aplikací vytvářet software, který udrží data, aniž by byl nucen opustit objektově orientované paradigma. [18] [19]

2.4.1 Výhody ORM

Použití ORM šetří spoustu času, jelikož datový model je napsán pouze v jednom místě a je tedy snazší jej aktualizovat, udržovat a znovu použít. Děje se tak, protože mnoho věcí se provádí automaticky, od zpracování databáze po lokalizaci. Většina frameworků nutí vývojáře psát kód *MVC*, což nakonec kód trochu vyčistí a zpřehlední. S *ORM* není nutné psát špatně tvarované *SQL* dotazy a míchat tak vlastně jazyky do sebe. Použití předpřipravených příkazů nebo transakcí je stejně snadné jako volání metod. [19]

[11] Vytvořeno autorem.

Používání knihovny *ORM* je flexibilnější, protože zapadá do přirozeného způsobu kódování a abstrahuje databázový model, takže může být kdykoli změněn bez větších úprav na straně aplikace. Model je slabě svázán se zbytkem aplikace, lze jej změnit nebo použít kdekoli jinde. Umožňuje snadno používat vlastnosti *OOP*, jako je například dědičnost. [19]

2.4.2 Nevýhody ORM

Pro správné použití knihovny *ORM* je potřeba jej správně pochopit a naučit se s ní pracovat a často i jejich speciální jazyky (např.: *HQL*), které však nepatří mezi nejjednodušší nástroje. Dalším problémem je výkon, ten je vysoký u obvyklých dotazů, ale databázový server bude mít vždy lepší výkon s vlastními dotazy *SQL*, převážně je tento fakt významně znát u velkých databázových aplikací. Nemalý problém může *ORM* znamenat pro začátečníky, jelikož abstrakce databázového modelu se může stát velkým problémem nových programátorů, kteří často píší neoptimalizované programy. Bohužel u *ORM* knihoven nejsou vždy všechny funkce pod kontrolou, nelze tak naprosto dokonale vyladit výkon aplikací. [19]

2.4.3 Příklady ORM knihoven

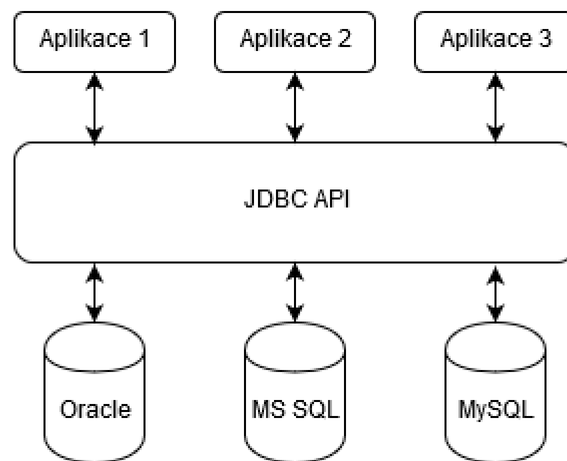
Zástupci knihoven *ORM* používají stejné zásady. Nejznámějšími knihovnami jsou:

- Hibernace – implementace pro jazyk Java
- Doctrine – implementace pro jazyk PHP
- Django ORM – implementace pro jazyk Python
- Entity Framework – implementace pro jazyk C#

3 JDBC

Následující kapitola se zabývá problematikou implementace přímého přístupu k databázovým systémům pomocí *SQL* dotazů v programovacím jazyku Java, který je klasickým příkladem hojně využívaného objektově orientovaného jazyka, a jsou v něm často využívány oba přístupy.

Java Database Connectivity (*JDBC*) je aplikační programové rozhraní (*API*), součást Java SE, které umožňuje standardizovat (viz obr. 11) a zjednodušit proces připojení aplikací psaných v jazyce Java k databázovým systémům. V zásadě platí, že aplikace napsané v jazyce Java provádějí logiku. Java poskytuje prostředky pro provádění logiky a objektově orientovanou analýzu pomocí tříd a rozhraní. Java aplikace však data trvale neukládají. Perzistence dat je obvykle delegována na databáze *NoSQL*, jako je *MongoDB* a *Cassandra*, nebo na relační databáze, jako je *Oracle database*, *Microsoft SQL Server* nebo populární otevřená zdrojová databáze *MySQL*. [20]



Obrázek 11: Diagram *JDBC*[12]

3.1 Rozhraní, třídy a komponenty JDBC

JDBC API se skládá z několika rozhraní a tříd, které představují připojení k databázi, poskytují komponenty pro odesílání dotazů *SQL* do databáze a pomáhají vývojáři zpracovat výsledky interakcí relačních databází. Třídy *JDBC* jsou obsaženy v balíčcích *java.sql* a *javax.sql*. Popis základních tříd JDBC je v tabulce č. 1. [21]

[12] Vytvořeno autorem,

Tabulka 1: Popis tříd *JDBC*

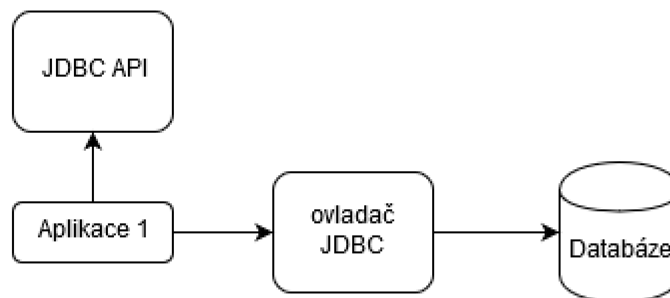
Rozhraní/třída	Popis
DriverManager	Tato třída řídí ovladače *JDBC*. Poskytuje metody jako *registerDriver()* a *getConnection()*.
Driver	Toto rozhraní je základní rozhraní pro každou třídu ovladačů, tj. Pokud si chceme vytvořit vlastní ovladač *JDBC*, musíme toto rozhraní implementovat. Pokud načteme třídu *Driver* (implementace tohoto rozhraní), vytvoří se vlastní instance a zaregistruje se ve správci ovladačů (*DriverManager*).
Statement	Toto rozhraní představuje příkaz *SQL*. Pomocí objektu *Statement* a jeho metod lze provést *SQL* dotaz a získat jeho výsledky. Poskytuje metody *execute()*, *executeBatch()*, *executeUpdate()*.
PreparedStatement	Toto rozhraní představuje předkompilovaný dotaz SQL. Dotaz je zkompilován a odeslán do databáze, později ho lze provést vícekrát. Objekt tohoto rozhraní lze získat pomocí metody objektu připojení s názvem *preparStatement()*. Rozhraní poskytuje metody jako *executeQuery()*, *executeUpdate()* a *execute()* a metody *getInt()*, *setInt()*, atd., kterými lze nastavit a získat hodnoty proměnných připraveného příkazu.
CallableStatement	Pomocí objektu tohoto rozhraní lze spouštět uložené procedury a funkce. Vrací jeden nebo více výsledků. Přijímá také vstupní parametry. *CallableStatement* lze vytvořit pomocí metody *preparCall()* objektu připojení. Poskytuje také metody *getInt()*, *setInt()*, atd., k předání vstupních parametrů a získání výstupních parametrů.
Connection	Toto rozhraní představuje spojení s konkrétní databází. Příkazy SQL jsou prováděny v souvislosti s připojením. Poskytuje metody *close(), commit(), rollback(), createStatement(), preparCall(), preparStatement(), setAutoCommit(), setSavepoint()*.

ResultSet	Toto rozhraní představuje sadu výsledků databáze, tabulku, která je generována spouštěním příkazů. Toto rozhraní poskytuje metody pro získání a aktualizace dat získaných z databáze.
ResultSetMetaData	Toto rozhraní se používá k získání informací o sadě výsledků, například o počtu sloupců, názvu sloupce, datovém typu sloupce, schématu sady výsledků, názvu tabulky atd. Poskytuje metody *getColumnCount(), getColumnName(), getColumnType(), getTableName(), getSchemaName()*.

[16]

3.2 Ovladač JDBC

Ovladače *JDBC* jsou adaptéry na straně klienta (nainstalované v klientském počítači, nikoli na serveru), které převádějí požadavky z programů na protokol, kterému rozumí databázový systém. Implementují jednotlivá rozhraní *JDBC API* (viz. obr. 12) a většinou jsou naprogramovány pro konkrétní typ databáze. Existují 4 typy ovladačů *JDBC*. [22]



Obrázek 12: Diagram ovladače *JDBC*.[13]

3.2.1 Typ 1 – JDBC–ODBC most

Ovladač typu 1 nebo ovladač mostu *JDBC-ODBC* používá ovladač *ODBC* k připojení k databázi. Převádí volání metod z *JDBC* na volání funkcí *ODBC*. Ovladač typu 1 se také nazývá univerzální ovladač, protože jej lze použít k připojení k libovolné databázi. Jelikož se používá pro interakci s různými databázemi, data přenesená prostřednictvím tohoto ovladače

[13] Vytvořeno autorem,

nejsou tak zabezpečena. Pro správný běh je třeba na klientských zařízeních nutné nainstalovat ovladač *ODBC*. [22]

3.2.2 Typ 2 – Nativní API

Ovladač typu 2 používá knihovny databáze na straně klienta. Tento ovladač převádí volání metod JDBC na nativní volání databázového *API*. K provozu s různými typy databází je nutné na straně klienta dodatečná knihovna odpovídající ke konkrétní databázi, z tohoto důvodu je přenos dat mnohem bezpečnější než ovladač typu 1. [22]

3.2.3 Typ 3 – Síťový protokol

Ovladač síťového protokolu používá *middleware* (aplikační server), který převádí volání *JDBC* přímo nebo nepřímo na databázový protokol specifický pro databázi. Zde jsou všechny ovladače pro připojení k databázi přítomny na jednom serveru, a proto není třeba jednotlivá instalace na straně klienta. Ovladače typu 3 jsou plně psány v Javě, proto jsou přenosnými ovladači. Žádná knihovna na straně klienta není nutná kvůli aplikačnímu serveru, který může provádět mnoho úkolů, jako je auditování, vyrovnávání zatížení, protokolování atd. Údržba ovladače síťového protokolu je nákladná, protože vyžaduje, aby se ve střední vrstvě provádělo kódování specifické pro databázi. [22]

3.2.4 Typ 4 – Thin driver

Ovladač typu 4 se také nazývá nativní ovladač. Tento ovladač komunikuje přímo s databází. Nevyžaduje žádnou nativní databázovou knihovnu, proto se také nazývá *Thin Driver*. Jeho specifikace je plně popsána přímo v jazyce Java. [22]

3.3 Práce s JDBC

Nejprve je nutné načíst ovladač nebo jej zaregistrovat před použitím v programu. Registrace se provádí jednou. Ovladač lze zaregistrovat jedním ze dvou způsobů, prvním je načtení třídy pomocí reflexe. Tento způsob je vhodný pro případ, kdy v době kompilace není jisté, jaký ovladač bude použit. Příklad registrace ovladače pro databázi *Oracle* je vidět na následujícím příkladu:[23]

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Zdrojový kód 3: Registrace ovladače *JDBC*.

Druhým a nejspíše snazším způsobem je registrace ovladače přímo do manageru, u tohoto způsobu, musí být v době kompilace ovladač přesně definován. Následný zdrojový kód ukazuje registrace ovladače pro databázi *Oracle* za pomoci třídy *DriverManager*.

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

Zdrojový kód 4: Registrace ovladače *JDBC* 2.

V další fázi je nezbytně nutné navázat připojení s databází. K tomu slouží rozhraní *Connection* a objekt implementující toto rozhraní, vytvořený *DriverManagerem*. V tuto fázi je nutné znát adresu databáze, port, typ databáze, ovladač a přihlašovací údaje. Ty lze předat následujícím kódem:

```
String url = "jdbc:oracle:thin:@localhost:1521:xe";
Connection
con = DriverManager.getConnection(url,user,password);
```

Zdrojový kód 5: Vytvoření připojení k databázi.

Po navázání spojení lze s databází komunikovat a vytvářet objekty určené pro komunikaci, pomocí následujících objektů:

```
Statement st = con.createStatement();
PreparedStatement st = con.prepareStatement(dotaz);
CallableStatement st = con.prepareCall(dotaz);
```

Zdrojový kód 6: Vytvoření objektů pro dotazy.

Těmto objektům lze snadno nastavit jednotlivé parametry pomocí metod *setString*, *setInt*, apod. Příklad přiřazení parametru typu *String* a *int* je následující:

```
st.setString(1, lastName);
st.setInt(2, id);
```

Zdrojový kód 7: Přiřazování parametrů dotazu.

A metodami *execute* provést dotaz na databázi následně:

```
ResultSet rs = st.executeQuery();
```

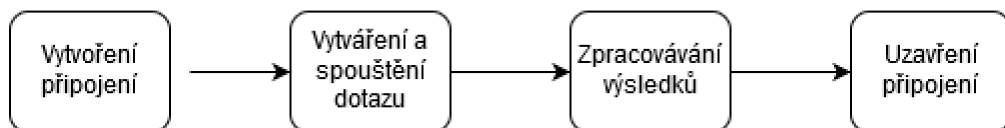
Zdrojový kód 8: Spouštění dotazu.

V navráceném objektu implementující *ResultSet* se ukrývají požadovaná data, získaná z databáze. Pro jejich načtení je vhodné využít cyklus. Mezi jednotlivým dotazováním je také vhodné, uzavřít připojení, jelikož by při velkém množství těchto připojení, databázový systém nebyl schopen otevřít další. K uzavření slouží metoda *close*: [23]

```
con.close();
```

Zdrojový kód 9: Uzavření připojení.

Celý proces připojení a dotazování se dá popsat takto:



Obrázek 13: Diagram připojení *JDBC*.[14]

3.4 Zhodnocení přístupu

Hlavní výhodou využívání čistě *JDBC*, je rychlost, pokud jsou vybírány správně typy *statementů*, lze docílit k bezkonkurenčním výkonům. Stejně tak výhodou je snadnost použití, každý, kdo umí *SQL,* se velmi rychle naučí, jak celé rozhraní funguje a postupně dokáže přesně odladit výkon. Pro provoz stačí pouze ovladač, není třeba žádná další knihovna, která by se musela dodatečně stahovat, jelikož specifikace je součástí jazyka Java.

Na druhou stranu programátor musí vytvářet mnoho opakujícího se kódu, který znepřehlední celý projekt a prodlužuje čas vývoje. Kód znepřehlední také *SQL*, které se vkládá do Javy a vytváří se ne příliš úhledná konstrukce.

JDBC je vhodné využívat ve velkých projektech, kde záleží na vysokém výkonu. [21]

---

[14] Vytvořeno autorem.

4 JPA

Java Persistence API je specifikace, která se týká perzistence, a volí jakýkoli mechanismus, díky kterému Java objekty přežívají proces aplikace, jenž je vytvořil. Specifikace *JPA* umožňuje definovat, které objekty by měly přetrvávat, a jak by tyto objekty měly být zachovány v *Java* aplikacích.

Samotné *JPA* není nástrojem ani frameworkem, spíše definuje sadu konceptů, které lze implementovat jakýmkoli nástrojem nebo frameworkem. Zatímco *JPA* model objektově relačních mapování (*ORM*) byl původně založen na frameworku *Hibernate*, postupně vznikli další různé implementace. Stejně tak, zatímco *JPA* byl původně určen pro použití s relačními databázemi, některé implementace *JPA* byly rozšířeny pro použití s datovými úložišti *NoSQL*. Další populární framework kromě *Hibernate*, je *EclipseLink* ten podporuje *JPA* s *NoSQL*.

4.1 Hibernate

Díky vzájemně propojené historii jsou *Hibernate* a *JPA* často spojovány. Stejně jako specifikace *Java Servlet*, *JPA* vytvořila mnoho kompatibilních nástrojů a frameworků, *Hibernace* je jen jedním z nich, který vyvinul Gavin King a byl vydán na začátku roku 2002. King vyvinul *Hibernate* jako alternativu k nástroji *Entity Bean*. Framework byl tak populární a v té době tak potřebný, že mnoho jeho nápadů bylo přijato a modifikováno v první specifikaci *JPA*. Dnes je Hibernate ORM jednou z nejvyspělejších implementací *JPA* a stále populární volbou pro *ORM* v *Javě*. Kromě toho se rodina nástrojů *Hibernate* rozšířila o populární *Hibernate Search*, *Hibernate Validator* a *Hibernate OGM*, které podporují perzistenci doménových modelů pro *NoSQL*. [24]

4.2 Perzistence dat v JPA

Z pohledu programování je vrstva *ORM* vrstvou adaptéru, přizpůsobuje graf objektů jazyku *SQL* a relačním tabulkám. Když se používá *JPA*, vytvoří se mapa z datového úložiště k objektům modelu aplikace. Namísto definování způsobu ukládání a načítání objektů se definuje mapování mezi objekty a databází a poté se volá *JPA*. Pokud se používá relační databáze, většinu skutečného spojení mezi kódem aplikace a databází řeší *JDBC*. [19]

Specifikací *JPA* jsou poskytovány anotace metadat, pomocí kterých lze definovat mapování mezi objekty a databázovým modelem. Každá implementace *JPA* poskytuje svoje

vlastní třídy pro anotace *JPA*. Specifikace *JPA* také poskytuje *PersistanceManager* a *EntityManager*, což jsou klíčové třídy komunikace se systémem JPA (kde obchodní kód říká systému, co dělat s mapovanými objekty). [19]

```
public class Musician {
  private Long id;
  private String name;
  private Instrument mainInstrument;
  private ArrayList performances = new ArrayList<Performance>();

  public Musician( Long id, String name){ }

  public void setName(String name){
    this.name = name;
  }
  public String getName(){
    return this.name;
  }
  public void setMainInstrument(Instrument instr){
    this.instrument = instr;
  }
  public Instrument getMainInstrument(){
    return this.instrument;
  }
}
```

Zdrojový kód 10: Příklad *JPA* entity.[15]

Persistence.xml

Jedná se o soubor, který ukládá konfiguraci pro *JPA*. Existují i jiné způsoby, jak poskytnout tyto informace systému, například programově nebo pomocí vlastního *xml souboru* každého frameworku. Vhodné je však použít soubor persistence.xml, protože ukládání závislostí tímto způsobem velmi usnadňuje aktualizaci aplikace bez úpravy kódu. [24]

```
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
        <persistence-unit name="MyUnit" transaction-type="RESOURCE_LOCAL">
                <properties>
                        <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/foo_bar"/>
                        <property name="javax.persistence.jdbc.user"
value=""/>
                        <property name="javax.persistence.jdbc.password"
value=""/>
                        <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver"/>
                </properties>
```

[15] Vytvořeno autorem,

```
    </persistence-unit>
</persistence>
```

Zdrojový kód 11: Příklad souboru persistence.xml

Spring a JPA

Použití frameworku *Spring* výrazně usnadní integraci *JPA* do naší aplikace. Příklad: umístění anotace `@SpringBootApplication` do záhlaví aplikace nařídí *Springu*, aby automaticky prohledával třídy a podle potřeby vkládal *EntityManager* na základě zadané konfigurace. [19]

4.3 Konfigurace JPA

Konfiguraci mapování jednotlivých tabulek lze nastavit dvojím způsobem, prvním je využiti xml konfiguračního souboru a druhým je využití anotací. U obou způsobů lze nastavovat stejné vlastnosti. [20]

Stejně jako většina moderních frameworků, *JPA* zahrnuje kódování konvencí, ve kterém framework poskytuje výchozí konfiguraci založenou na nejlepších postupech. Pokud například je vytvořena třída se stejným názvem jako databázová tabulka, framework si sám dokáže namapovat tuto tabulku. Konvenční konfigurace šetří čas a v mnoha případech funguje dobře. Je také možné přizpůsobit konfiguraci *JPA*. Jako příklad lze použít anotaci `@Tabulka` *JPA* k určení tabulky, do které má být uložena daná třída. [24]

4.3.1 Primární klíč

V JPA je primární klíč pole používané k jedinečné identifikaci každého objektu v databázi. Primární klíč je užitečný pro odkazování a přiřazování objektů k jiným entitám. Označuje se anotací `@Id.` *JPA* podporuje další strategie pro generování primárního klíče objektu. Obsahuje také anotace pro změnu názvů jednotlivých polí. Obecně je *JPA* dostatečně flexibilní, aby se přizpůsobil případnému mapování perzistence. [19]

```
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id;

    @Version
```

```
    @Column(name = "version")
    private int version;

    @Column
    private String firstName;

    @Column
    private String lastName;

    @ManyToMany(mappedBy="authors")
    private Set<Book> books = new HashSet<Book>();

}
```

Zdrojový kód 12: Příklad generování identity v entitě.[16]

4.3.2 CRUD operace

Jakmile je namapována třída na tabulku databáze a lze vytvořit její primární klíč, lze objekt každé třídy v databázi vytvořit, načíst, odstranit a aktualizovat. Volání `session.save()` vytvoří nebo aktualizuje zadanou třídu v závislosti na tom, zda je pole primárního klíče nulové nebo zda se vztahuje na existující entitu. Volání `entityManager.remove()` odstraní zadanou třídu. [19]

Většina implementací *JPA*, má také svůj specifický jazyk, např.: pro *Hibernate* se jedná o jazyk *HQL*. *HQL* je objektově orientovaný dotazovací jazyk podobný *SQL*, ale namísto práce s tabulkami a sloupcí, *HQL* pracuje s perzistentními objekty a jejich vlastnostmi. Dotazy *HQL* jsou pomocí *Hibernate* přeloženy do konvenčních dotazů *SQL*, které zase provádějí na databázi.[19]

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

Zdrojový kód 13: Příklad jazyka *HQL* a jeho využití[17]

4.4 Vztahy mezi entitami v JPA

JPA má také schopnost spravovat entity ve vztahu k sobě navzájem. V tabulkách i objektech jsou možné čtyři typy vztahů entit:

[16] Vytvořeno autorem.
[17] Vytvořeno autorem.

- One-to-many
- Many-to-one
- Many-to-many
- One-to-one

Každý typ vztahu popisuje, jak se entita vztahuje k jiným entitám. Například jedna entita by mohla mít vztah s jinou entitou představovanou kolekcí více entit, jako je List nebo Set. [19]

```
public class Musician {

  @OneToMany
  @JoinColumn(name="musicianId")
  private List<Performance> performances = new ArrayList<Performance>();
  //...
}
```

Zdrojový kód 14: Příklad propojení tabulek v entitě *JPA*[18]

4.5 Získávání dat

Kromě toho, že *JPA* ví, kam umístit související entity do databáze, potřebuje vědět, jak je chceme načíst. Strategie načítání říká *JPA*, jak načíst související entity. Při načítání a ukládání objektů musí framework poskytnout možnost dokončit způsob zpracování grafů objektů.

JPA má schopnost tzv „*lazy fetch*“, kdy jsou data načtena z databáze postupně, až když jsou v aplikaci opravdu potřebná, opakem je „*eager fetch*“, kdy jsou data načtena již předem. Obě tyto možnosti mohou mít jak kladný, tak záporný důsledek na běh aplikace a databáze. Pomocí anotací můžete přizpůsobit své strategie načítání, ale výchozí konfigurace *JPA* často nefunguje beze změn:

- One-to-many: Lazy
- Many-to-one: Eager
- Many-to-many: Lazy
- One-to-one: Eager

---

[18] Vytvořeno autorem.

4.6 Zhodnocení přístupu

Pravděpodobně nejčastěji uváděnou výhodou JPA a jakékoli její implementace je produktivita při vývoji. Hlavním důvodem je to, že musíme definovat mapování mezi databázovými tabulkami a modelem pouze jednou. Kromě toho získáváme řadu dalších funkcí, které bychom jinak museli implementovat sami, jako je generování primárních klíčů, synchronizace vláken a různé optimalizace výkonu. Další výhodou jen nezávislost k databázi, jelikož nás *JPA* nenutí psát přímo SQL, nemusíme řešit dialekt jazyka pro každý databázový systém, framework jej dokáže sám identifikovat a správně s ním pracovat. Používání *JPA* je také bezpečné, jelikož jsou veškeré parametry *bindovany* a nehrozí tak riziko *SQL injection*. Implementace jako *Hibernate* také poskytují *cachování dat*, což je schopnost, která může mít kladný i záporný dopad, avšak při správném využívání by nemělo docházet k problémům.

Největším problémem *JPA* a celkově *ORM*, je výkon, tyto frameworky nebudou mít nikdy tak vysoký výkon jako v případě využívání čistě *JDBC*. Bohužel u těchto frameworků není nikdy absolutní kontrola toho, co vykonává kód, příkladem může být *lazy fetch*, kdy jsou data automaticky stahována, avšak není možno definovat za jakých podmínek apod. Pro velkou část programátorů je také problém specifický jazyk, ten se totiž musí naučit, a to je časově poněkud náročné.

JPA je spíše vhodné pro využití v menších projektech, ve kterých není nezbytně nutný vysoký výkon a není problém využívat další knihovny. [24] [19]

5 NÁVRH A VÝVOJ PROGRAMŮ

5.1 Vývoj

K vývoji výstupních aplikací byl využit, již zmíněný jazyk *Java EE*, konkrétně ve verzi 8. Ten je plně objektový a hojně rozšířený mezi širokou veřejnost vývojářů. Jelikož poskytuje velké množství knihoven je v něm velmi rychlý vývoj. Samotný jazyk je multiplatformní, může tak být nasazen jak na Windows, tak na systémech založených na principech *UNIX*. Je toho docíleno pomocí *bytecodu*, ten je nejprve zkompilován a posléze ho lze spustit na běhovém prostředí *JVM* (*Java Virtual Machine*). *JVM* je vytvořeno pro každou platformu zvlášť a využívá mnoho optimalizačních technik, např.: *JIT* (*Just In Time*), takže dosahuje vysokého výkonu, ačkoli se jedná o interpretovaný jazyk.

Pro samotné psaní kódů bylo využito vývojové prostředí *IntelliJ IDEA ultimate*, ve verzi 2019, které sice není šířeno zdarma, avšak poskytuje funkce, které zásadně zrychlují vývoj, např.: *refactoring*, integrace *verzovacích systémů*, mnoho *pluginů*, snadná možnost *debuggingu*, správa databáze nebo nástroje pro správu a řízení sestavování projektů jako je *Apache Maven*.

Apache Maven je nástroj pro automatizaci sestavení projektů v jazyce Java. Jedná se o open source projekt vyvíjený nadací *Apache* od roku 2003. Vzhledem k jeho silné podpoře a obrovské popularitě je *Maven* velmi stabilní a bohatý na funkce, poskytuje mnoho pluginů, které mohou dělat skoro cokoli, od generování dokumentace v *PDF* ke generování seznamu posledních změn z *verzovacího systému*. K přidání této funkce stačí pouze jednoduché *XML* nebo zvláštní parametr příkazového řádku. *Maven* se také dokáže připojit ke vzdáleným repositářům (nebo si lze nastavit vlastní *lokální repositáře*) a automaticky stahovat všechny *závislosti* potřebné k sestavení projektu.

Používání *Mavenu* je velmi snadné. Každý projekt obsahuje soubor nazvaný *POM* (*Project Object Model*), což je pouze soubor *XML* obsahující podrobnosti o projektu. Tyto podrobnosti mohou zahrnovat název projektu, *verzi*, *typ balíčku*, *závislosti*, *pluginy Maven* atd.

5.2 Technologie využité v aplikaci

Krom již zmíněných technologií (*JDBC*, *Hibernate* a *specifikace JPA*), jsou v aplikaci využity následující kategorie, popsané v této kapitole.

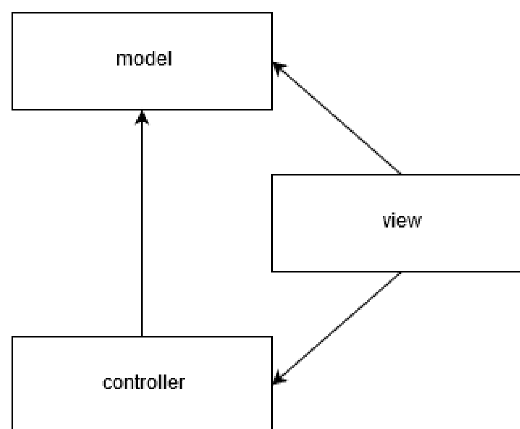
5.2.1 Model MVC

Model View Controller (*MVC*) je návrhový vzor, který rozděluje aplikaci na tři hlavní logické komponenty: *model*, *pohled* a *ovladač*. Každá z těchto komponent je navržena tak, aby zvládla specifické aspekty vývoje aplikace. *MVC* je jedním z nejčastěji používaných průmyslových standardů vývoje webových aplikací pro vytváření škálovatelných a rozšiřitelných projektů. [26]

Komponenta *Model* odpovídá veškeré logice související s daty, se kterou uživatel pracuje. To může představovat buď data, která jsou přenášena mezi komponenty *pohledu* a *ovladače*, nebo jakákoli jiná data související s obchodní logikou. Například objekt *Zákazník* načte informace o *zákaznících* z databáze, s nimi manipuluje a aktualizuje je zpět do. [26]

Komponenta *View* (*pohled*) se používá pro veškerou logiku uživatelského rozhraní aplikace. Například pohled zákazníka bude zahrnovat textová pole, rozevírací seznamy atd., Se kterými koncový uživatel pracuje. [26]

Ovladače fungují jako rozhraní mezi komponentami *Model* a *View* pro zpracování všech obchodních logik a příchozích požadavků, manipulaci s daty pomocí komponenty *model* a interakci s pohledy za účelem vykreslení konečného výstupu. *Ovladač zákazníka* bude například zpracovávat všechny interakce a vstupy z pohledu zákazníka a aktualizovat databázi pomocí modelu zákazníka. Jednotlivé komponenty mají mezi sebou vztahy dle obr. č. 14. [26]



Obrázek 14: Vztahy komponent modelu *MVC*. [19]

[19] Vytvořeno autorem.

5.2.2 JavaFX

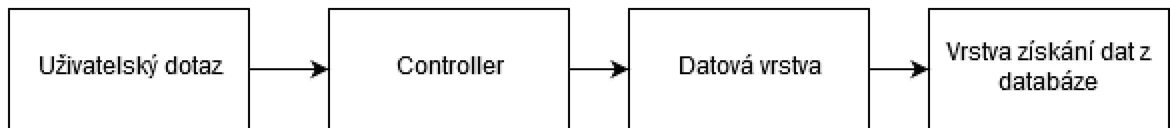
JavaFX je sada Java nástrojů pro vytváření *GUI* aplikací, které běží na *Windows*, *MacOS* nebo *Linuxu*. Je to nástupce nástrojů *Swing* a *AWT*. Nabízí tak například dialogy, textová pole, knihovny pro zobrazování grafů, tlačítka nebo nástroje pro vykreslování 3D grafických objektů. Od vydání *Java* 11 v roce 2018 je *JavaFX* součástí projektu *OpenJDK*, a je tedy šířena open-source a je vyvíjen mnoha lidmi z mnoha různých společností.

5.2.3 Logback

Logback je logovací framework pro aplikace psané v Javě, je vytvořen jako nástupce populárního projektu log4j. Ve skutečnosti byly oba tyto rámce vytvořeny stejným vývojářem. Vzhledem k tomu, že protokolování je klíčovou součástí jakékoli aplikace pro účely ladění i auditu, výběr vhodné logovací knihovny je základním rozhodnutím každého projektu. Oproti logování pomocí standartních knihoven Javy, nabízí *logback* vytvoření vlastních „*Appenders*“ těmi lze logovat nejen do standartnímu výstupu ale například do souboru nebo přímo do aplikace. Lze také nastavit mírů závažnosti hlášení apod.

5.3 Architektura aplikace

Celá aplikace je navržena tak aby simulovala co možná nejčastější využití, data získaná z databáze jsou většinou zpracována do datových objektů tak, aby bylo možná co nejsnazší s nimi pracovat ze zbytku aplikace. Proto byla navržena datová vrstva, která se o převod dat stará, dle následujícího diagramu:



Obrázek 15: Architektura aplikace. [20]

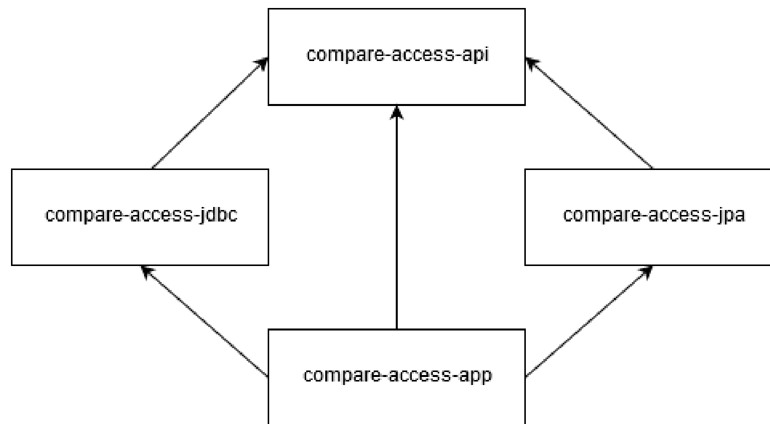
5.4 Struktura projektu

Celý projekt je tvořen více různými programy, ty jsou poté využívány jako samostatné knihovny. Vzhledem k četnému množství souborů, byly jednotlivé programy vytvořeny jako moduly Maven a následně označeny.

- **compare-access-agg**: Agregátor projektu, zahrnuje všechny moduly.
  - **compare-access-api**: Společné API pro ostatní programy.
  - **compare-access-app**: Řídící aplikace.
  - **compare-access-jdbc**: Testovací program pro nativní přístup.
  - **compare-access-jpa**: Testovací program pro ORM přístup.
  - **db**: Skripty pro vytvoření testovací databáze.
  - **lib**: Knihovny nutné pro běh jednotlivých programů (vytváří se automaticky).
  - **config.properties**: Uživatelský konfigurační soubor aplikace.
  - **pom.xml**: Konfigurační soubor projektu pro Maven.

Jednotlivé programy mají závislosti dle následujícího diagramu:

[20] Vytvořeno autorem.



Obrázek 16: Diagram závislostí jednotlivých programů/aplikací.[21]

5.4.1 Soubor config.properties

Příklad uživatelského nastavení souboru pro použití s databází Oracle může být následující.

```
#################################################################
#konfigurační soubor pro testovací program compare-access-app
#@author Petr Hotovec
#################################################################
#----------------------------------------------------------------
#--------------------konfigurace k připojení k DB-----------------
#----------------------------------------------------------------
#lokace knihovny s ovladačem DB
driverFile=lib/ojdbc8.jar
#classpath hlavní třídy ovladače
driverClasspath= oracle.jdbc.OracleDriver
#dialekt
dialect=org.hibernate.dialect.OracleDialect
#url databáze
url=jdbc:oracle:thin:@localhost:1521:test
#název uživatele
user=root
#heslo uživatele
password=root
```

[21] Vytvořeno autorem.

```
#skript který připravý DB na testy
scriptFile=db/scheme.sql

numberTestObjects=50
```

Zdrojový kód 15: Příklad uživatelského nastavení programu *APP*.[22]

Vlastnosti *driverFile*, *driverClasspath*, určuji již zmíněný ovladač databáze, který je načítán *controllerem*. Hodnoty *url*, *user* a *password* jsou údaje poskytované *controllerem* jednotlivým testovacím programům a znázorňují připojení k databázovému systému. Vlastnost *numberTestObjects* udává, testovacím programům, kolikrát má být testovací scénář proveden. Nastavená *dialect* je určeno pouze pro program JPA a slouží k nastavení *dialectu* databáze pro *JPA* nástroj. Poslední naprosto specifická vlastno je *scriptFile*, jedná se o cestu k souboru, který by měl ukrývat *SQL* pro vytvoření databáze, před samotným testováním, tuto činnost vykonává samostná knihovna *controlleru* IO, určená výhradně pro komunikaci mimo systém.

5.4.2 Modul API

Program pro definování programových rozhraní a objektů datové vrstvy. Je programem, který ukrývá základní testovací rozhraní, pro testovací programy. Celý program má za cíl udržet stejné podmínky pro testování v různých programech napříč projektem. Modul je tvořen zejména následujícími třídami:

- **obj**: Balíček datových tříd (entit) datové vrstvy, přímo odpovídají databázovým tabulkám.
- **Result**: Třída pro sběr statistických informací o testech.
- **Results**: Kontejner pro objekty *Result*.
- **Test**: Rozhraní definující testovací třídu testovacích programů, popsáno tabulkou níže.

[22] Vytvořeno autorem.

Tabulka 2: Popis rozhraní *Test*.[23]

Název metody	Parametr	Popis
basicTest	Consumer<Result[]> taskAfterComplete	Předpis pro test 1, parametr slouží pro asynchronní získání dat po dokončení testu
nodeTest	Consumer<Result[]> taskAfterComplete	Předpis pro test 2, parametr slouží pro asynchronní získání dat po dokončení testu

Objekty balíčku *obj* obsahují také metadata tvořená anotacemi JPA, například třída *Objednavka* je popsána takto:

```
@Entity
@Table(name = "objednavka")public class Objednavka {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "idobjednavky") private int idobjednavky;
  @Column(name = "casobjednavky") private java.sql.Timestamp casobjednavky;
  @Column(name = "casodeslani") private java.sql.Timestamp casodeslani;
  @Column(name = "adresa") private String adresa;
  @Column(name = "mesto") private String mesto;
  @Column(name = "psc") private String psc;
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "idzamestnance") private Zamestnanec zamestnanec;
  @Column(name = "jmenoprijemce") private String jmenoprijemce;
  @Column(name = "prijmeniprijemce") private String prijmeniprijemce;
  @Column(name = "telefonprijemce") private String telefonprijemce;
  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "idprepravy") private Prepravnimoznost preprava;
  @ManyToOne(fetch = FetchType.EAGER)
  @JoinColumn(name = "idzakaznika") private Zakaznik zakaznik;
```

[23] Vytvořeno autorem.

Zdrojový kód 16: Anotace datové třídy *Objednavka*.[24]

5.4.3 Program APP

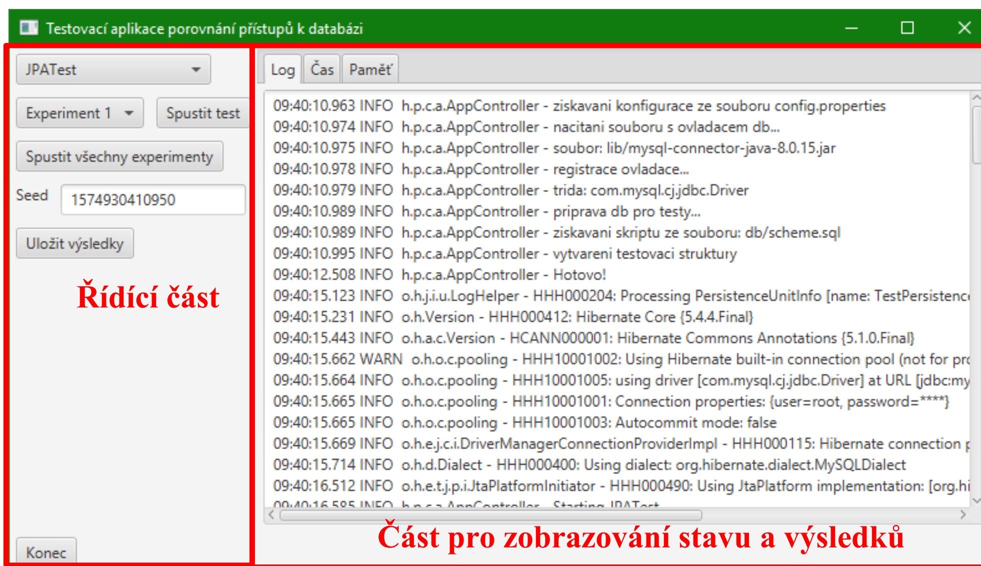
Hlavní řídící aplikace, poskytující controller, který spouští testovací programy a předává jím potřebná data, získana z konfiguračního souboru a uživatelské rozhraní. Struktura programu je následující:

- **AppController**: Hlavní controller pro ovládání jednotlivých programů, předává jím potřebné parametry z konfiguračního soboru a uživatelského rozhraní. Při vytváření tvoří testovací strukturu databáze.
- **GraphicsLib**: Knihovna poskytující grafické prvky, např.: dialogy.
- **GUI**: Controller uživatelského rozhraní, samotné rozhraní je vytvořeno v souboru gui.fxml, zprostředkovává informace mezi uživatelem a *AppControllerem.*
- **IOLib**: Knihovna vstupů a výstupů, ukládá výsledky testů a vytváří testovací databázovou strukturu.
- **LogOutputStreamAppender**: Appender frameworku logback pro vkládání logů do uživatelského rozhraní.
- **RefreshCallback**: Třída pro znovu vykreslení aplikace po dokončení test.

[24] Vytvořeno autorem.

Uživatelské rozhraní

Okno aplikace je tvořeno ze dvou hlavních částí dle následujícího obrázku:

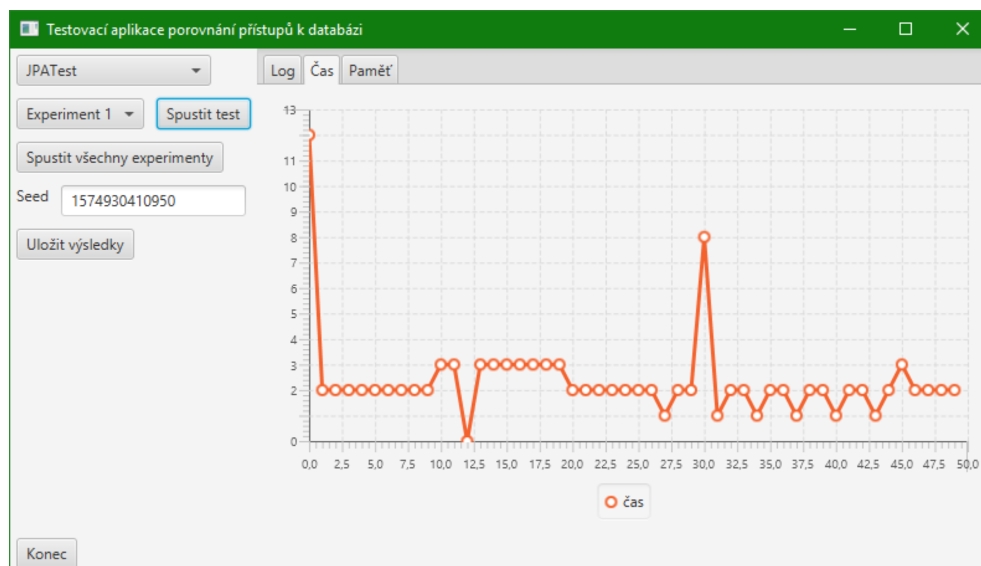


Obrázek 17: Okno řídící aplikace.[25]

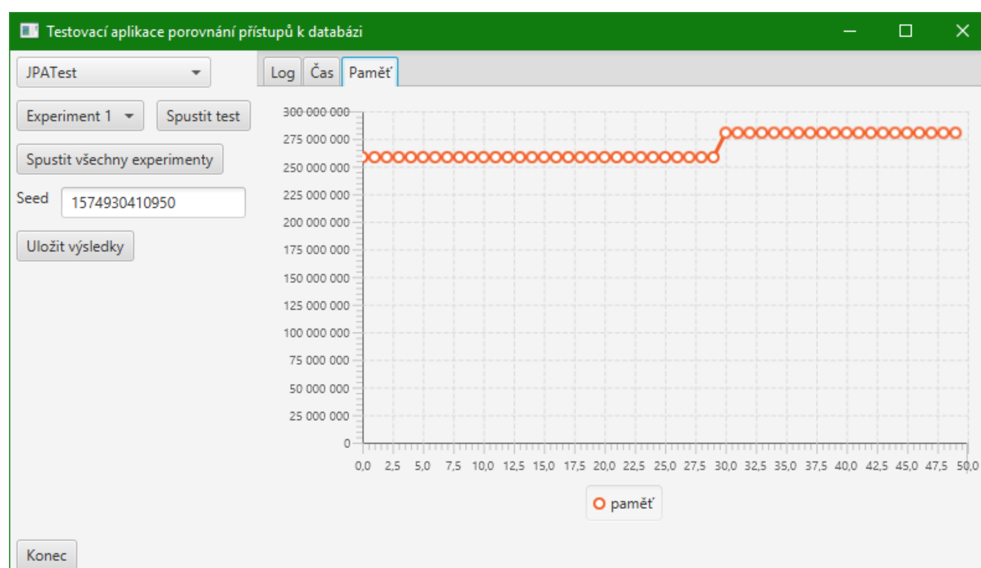
V řídící části má uživatel možnost nastavit, který test se má spouštět a který test, má také možnost spustit všechny testy pro daný program najednou. Další možnost je zadání násady generátoru náhodných čísel, uložení všech výsledků již proběhlého testu a ukončení programu.

Druhá část je rozdělena do třech podčástí, první zobrazuje logy, které zadaný test generuje, ostatní 2 jsou určeny k zobrazování grafů výsledků jednotlivých testů, dle obrázku níže.

[25] Vytvořeno autorem.



Obrázek 18: Příklad grafu výsledků časů pro proběhlý test.[26]



Obrázek 19: Příklad grafu výsledku paměti pro proběhlý test.[27]

[26] Vytvořeno autorem.
[27] Vytvořeno autorem.

5.4.4 Program JDBC

Jedná se o jednu ze dvou implementací rozhraní *Test,* tato implementace znázorňuje testovací program pro spouštění scénářů za pomoci nativního přístupu, jen za pomoci SQL. Jednotlivé třídy jsou znázorněny následovně:

- **Preparedstatement**: balíček pro testování za pomocí *PreparedStatement*.
  - **PreparedStatementTest**: hlavní testovací třída (implementuje rozhraní *Test*).
  - **PreparedStatementTestDao**: třída zapouzdřující jednotlivá volání.
- **Statement**: balíček pro testování za pomocí *Statement*.
  - **StatementTest**: hlavní testovací třída (implementuje rozhraní *Test*).
  - **StatementTestDao**: třída zapouzdřující jednotlivá volání.

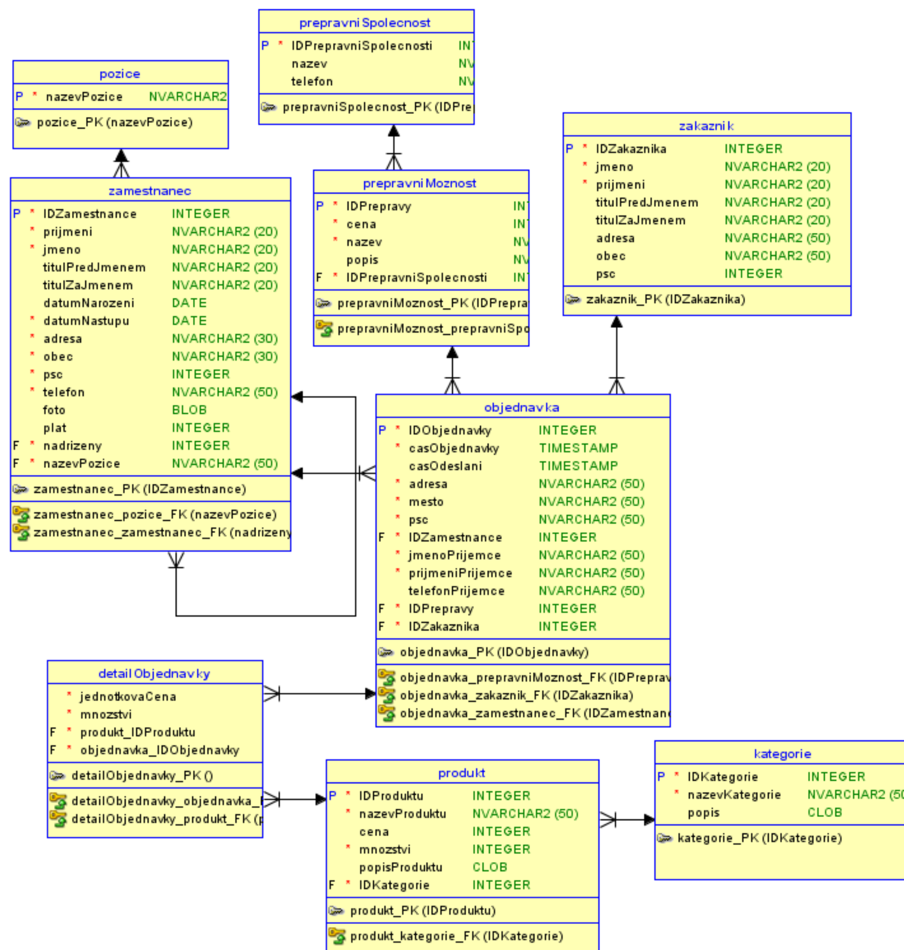
Po spuštění testu, hlavní třída vždy na vlastním vlákně vyhodnocuje výsledky daných scénářů a poskytuje spuštění dodaného consumera, předaného parametrem. Vytvoření hlavních třídy se provádí konstruktorem, který vyžaduje konfigurace Properties, z konfiguračního souboru, a objekt třídy Random, vyžadující scénáře.

5.4.5 Program JPA

Program implementující rozhraní *Test*, poskytuje metody pro spuštění scénářů a následné vyhodnocení náročnosti jeho běhu. Tato implementace poskytuje přístup k databázovému systému pomocí *ORM*. Pro vytvoření hlavní třídy programu, je nutné předat konstruktorem údaje *Properties* převzaté z konfiguračního souboru *config.properties* a objekt třídy *Random*, která je nutná pro běh scénářů. Při vytvoření testu je také vytvořena konfigurace persistence *JPA*, ta se vytváří programově na základě převzaté konfigurace. Samotný běh scénáře probíhá asynchronně na vlastním vlákně, po dokončení, je spuštěn *Consumer* předaný parametrem metody daného testu.

5.4.6 Testovací databáze

Pro spouštění testů, byl vytvořen databázový model. Tento model se při spuštění řídící aplikace automaticky nahrává do databáze. Je znázorněn relačním modelem na obr. č. 20.



Obrázek 20: Relační databázový model testovací databáze. [28]

5.5 Problémy při implementaci

Největší problém bylo spuštění samotného testu, jelikož je třeba načíst větší množství dat z databáze, tak tato operace zabírá relativně hodně času. Bylo tak třeba nutné využití dalších vláken. Avšak tímto řešením se dostavili další problémy, bylo třeba tyto vlákna synchronizovat a získávání statistických dat nebylo tak snadné jako v případě jednovláknové aplikace. Řešením bylo předávání již zmíněné třídy *RefreshCallback*, která je implementací rozhraní *Consumer*, tudíš obsahuje pouze jedinou metodu, která má za úkol aktualizovat data v uživatelském prostředí. Zde se objevil další problém, data do uživatelského prostředí založeném na *JavaFX* mohou být vložena pouze z hlavního vlákna řídící aplikace, to bylo vyřešeno použitím metody *Platform.runLater()*, která spustí zadanou metodu na hlavním vlákně a ohlídá synchronizaci.

---

[28] Vytvořeno autorem.

6 TESTOVÁNÍ

Následující část se bude týkat testů, budou provedeny dva testy určené pro získávání dat z databáze za pomoci vytvořených dvou testovacích aplikací. Tyto aplikace budou v průběhu kontrolovat základní statistiky ovlivňující běh programu. Jedna ze základních hodnot, která bude zkoumána, je doba běhu, na začátku každého testu bude změřen čas a po skončení bude čas změřen znovu, rozdíl těchto dvou hodnot bude zaznamenán a využit k porovnání. Druhým faktorem bude využití paměti, zde bude po dokončení každého dotazu změřeno využití paměti využité *JVM*.

V rámci prvního testu, budou získána náhodná data z tabulky objednávky s jejich zákazníky. Tento test bude proveden jak pro nativní přístup, tak pro přístup s využitím *ORM*. Počet získaných dat bude závislý na konfiguraci testovacího programu. Výsledná data o běhu budou vzájemně porovnána.

Druhý test bude získávat veškerá data z tabulky zaměstnanci, jelikož jsou jednotlivé záznamy v této tabulce závislé na jiné záznamy této tabulky (nadřízený zaměstnanec), bude v rámci testu vytvořena mapa těchto výsledných objektů. I zde dojde k testům jak nativním přístupem, tak za použití *ORM*, avšak u nativního přístupu bude vytvořen jednoduchý algoritmus pro správné navázání jednotlivých objektů na sebe.

Aplikace pro testování jsou nastaveny tak, aby co možná nejvíce odpovídalo reálnému provozu. Do databázových tabulek byl vygenerován následující počet řádků dat:

- *Zamestnanec* – 1000 řádků
- *Zakaznik* – 2000 řádků
- *Produkt* – 2000 řádků
- *Prepravnispolecnost* – 3 řádky
- *Prepravnimoznost* – 4 řádky
- *Pozice* – 5 řádků
- *Objednávky* – 5000 řádků
- *Kategorie* – 20 řádků
- *Detailobjednavky* – 5000 řádků

Testování bylo prováděno s *JVM* ve verzi 1.8.0_192 64bit architektury, databází *Oracle Database 19c Standart Edition*, knihovnami *logback* verze 1.2.3, *javax.persistence-api* verze 2.2, *apache commons-io* verze 2.6 a ovladačem databáze *ojdbc8* verze 19.3.0.0. Program *JPA* navíc využívá knihovnu *hibernate-core* verze 5.4.4.Final pro implementaci *JPA*. Operačním systémem byl *Windows 10 Pro x64* verze 1809. Testovací počítačová sestava má následující hardware:

- CPU: Intel Core i5-8400
- RAM: Kingston DDR4-2400 8GB
- SSD: Kingston SA400S37240G

6.1 Metriky měření testů

V následujících testech jsou měřeny dvě již zmíněné hodnoty, jedná se o čas, za který je test dokončen a využitá operační paměť. U obou testovacích programů, jsou tyto hodnoty měřeny naprosto identicky. Každý test je prováděn několikrát dle zadané konfigurace, hodnoty jsou měřeny za každou iteraci (viz. obr. 22). Samotnému testování předchází několik kroků připravující potřebné objekty. Po dokončení testu jsou výsledky zapisovány do výsledné datové struktury, dle obr. 22.

Obrázek 21: Diagram scénáře jednotlivých testů.[29]

Čas

Před spuštěním každého testu je zjištěn čas a poté uložen do proměnné. Čas je zjišťován pomocí statické metody *System.currentTimeMillis()*. Následně je spuštěn cyklus, uvnitř kterého se provádí daný test. Po dokončení testu je čas změřen naprosto identickým způsobem a spočítán rozdíl časů před začátkem testu a po jeho dokončení. Tento rozdíl je poté uložen do pole výsledků za každou iteraci cyklu (každé dokončení testu). Po dokončení cyklu je pole výsledných časů zaznamenáno to datové struktury *Result*.

[29] Vytvořeno autorem.

Pamět

Nejprve je vytvořen objekt *Runtime runtime = Runtime.getRuntime()*, ten uchovává veškeré informace o běhu programu. Následuje spuštění cyklu s testem a na konci každého testu je zjištěno využití paměti JVM pomocí: *runtime.totalMemory() - runtime.freeMemory()*, tato hodnota je uložena do pole výsledků za každou iteraci cyklu. Po ukončení poslední iterace je pole výsledků uloženo do datové struktury *Result*.

6.2 Příprava testů

V prvním kroku jsou připravovány veškeré nutné objekty. Ty jsou odlišné podle aplikace a přístupu k datům. U Programu *JDBC* se jedná o vytvoření instance tříd *StatementTestDao* a *PreparedStatementTestDao*. V tomto kroku jsou předány informace o připojení k databázi a následně je vytvořeno připojení pro oba typy, pomocí kódu:

```
Connection con = DriverManager.getConnection(dbURL, user, password);
```

Zdrojový kód 17: Vytvoření připojení k databázi[30]

Při vytváření třídy *PreparedStatementTestDao* je navíc připravován dotaz pro databazi následujícím kódem pro test 1:

```
PreparedStatement stmt1 = con.prepareCall("SELECT * FROM OBJEDNAVKA JOIN
ZAKAZNIK ON OBJEDNAVKA.IDZAKAZNIKA = ZAKAZNIK.IDZAKAZNIKA WHERE
IDOBJEDNAVKY=?");
```

Zdrojový kód 18: Příprava dotazu pro test 2. [31]

Pro test 2:

```
PreparedStatement stmt2 = con.prepareCall("SELECT * FROM ZAMESTNANEC WHERE
IDZAMESTNANCE=?");
```

[30] Vytvořeno autorem.
[31] Vytvořeno autorem.

Zdrojový kód 19: Příprava dotazu pro test 2[32]

Pro test s využitím *JPA* je nutno vytvořit *EntityManager*, ten zpřístupní jednotlivé funkce potřebné ke spuštění samotného testu. Pro jeho instancování se vytváří struktura *Properties* odpovídající souboru *persistence.xml*, která se vkládá jako parametr do následujícího kódu:

```
EntityManagerFactory emf Persistence.createEntityManagerFactory(persistence
Properties);
EntityManager em = emf.createEntityManager();
```

Zdrojový kód 20: Vytvoření EntityManageru pro *JPA*[33]

V dalším kroku se získává objekt *Runtime* pro sledování využití paměti. Tím je příprava pro testy hotová.

6.3 Test 1

První test se snaží popsat, jakým způsobem se může lišit především přístupová doba k běžně přistupovaným datům v závislosti na technologii.

Vstupem tohoto testu je generátor náhodných čísel a počet opakování testu (50), výstupem musí být seznam objektů/entit datové vrstvy získaných z databáze s naměřenými hodnotami času získání a zpracování dat z databáze a momentální využití paměti JVM za každý test.

6.3.1 Nativní přístup

Po odměření počátečních hodnot (čas procesoru a využití paměti) je spuštěn cyklus s 50 opakováními, v každé iteraci je volán dotaz:

[32] Vytvořeno autorem.
[33] Vytvořeno autorem.

```
SELECT * FROM OBJEDNAVKA JOIN ZAKAZNIK WHERE IDOBJEDNAVKY=id;
```

Zdrojový kód 21: *SQL* dotaz testu 1. [34]

Hodnota *id* v dotazu je parametr, který je náhodně generován od hodnoty 1 do hodnoty 5000. V celém kódu toto vypadá následovně:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM OBJEDNAVKA JOIN  ZAKAZNIK ON
OBJEDNAVKA.IDZAKAZNIKA = ZAKAZNIK.IDZAKAZNIKA WHERE IDOBJEDNAVKY="+id);
```

Zdrojový kód 22: Zdrojový kód získávající data z databáze pro *JDBC*. [35]

Pro srovnání byl stejným způsobem testován přístup pomocí třídy *PreparedStatement*, oproti předchozímu případu, parametr id není přímo vložen v dotazu, ale je posílán dodatečně do databázového systému pomocí:

```
stmt.setInt(1,id);
ResultSet rs = stmt.executeQuery();
```

Zdrojový kód 23: Spouštění dotazu pro *PreparedStatement*. [36]

Získaná data (*ResultSet*) za každou iteraci cyklu byla mapována do objektu datové vrstvy *Objednávka* a uložena do datové struktury seznam, do objektu *Result* se zaznamenával rozdíl mezi časem poslední iterace nebo počátečním a aktuálním časem, stejným způsobem se ukládá využití paměti. Algoritmus pro mapování objektů nativního přístupu je následující:

```
if (rs.next()) {
            Objednavka objednavka = new Objednavka();
            objednavka.setIdobjednavky(rs.getInt(1));
            objednavka.setCasobjednavky(rs.getTimestamp(2));
            objednavka.setCasodeslani(rs.getTimestamp(3));
            objednavka.setAdresa(rs.getString(4));
            objednavka.setMesto(rs.getString(5));
            objednavka.setPsc(rs.getString(6));
            objednavka.setJmenoprijemce(rs.getString(8));
            objednavka.setPrijmeniprijemce(rs.getString(9));
            objednavka.setTelefonprijemce(rs.getString(10));
```

[34] Vytvořeno autorem.
[35] Vytvořeno autorem.
[36] Vytvořeno autorem.

```
    Zakaznik zakaznik = new Zakaznik();
    zakaznik.setIdzakaznika(rs.getInt(12));
    zakaznik.setJmeno(rs.getString(14));
    zakaznik.setPrijmeni(rs.getString(15));
    zakaznik.setTitulpredjmenem(rs.getString(16));
    zakaznik.setTitulzajmenem(rs.getString(17));
    zakaznik.setAdresa(rs.getString(18));
    zakaznik.setObec(rs.getString(19));
    zakaznik.setPsc(rs.getString(20));
    objednavka.setZakaznik(zakaznik);
    return objednavka;
  }
```

Zdrojový kód 24: Kód pro mapování na objekty datové vrstvy pro nativní přístup[37]

6.3.2 Přístup ORM

Test pro objektově relační přístup vypadá naprosto identicky jako při nativním přístupu, jediný rozdíl se týká dotazu. Pro generování náhodných čísel byl využit generátor se stejnou násadou jako tomu je u nativního přístupu. Stejný je i počet opakování a rozsah generátoru. Místo SQL dotazu se využívá následující kód:

```
Session session = em.unwrap(Session.class);
session.get(Objednavka.class,random.nextInt(5000));
```

Zdrojový kód 25: Zdrojový kód získávající data z databáze pro *JPA*.[38]

Navíc jsou v entitě *Objednavka* nastaveny následující *metadata*, tak aby výstup byl naprosto identický jako u nativního přístupu a záznamy i z jiných tabulek byli získány ve stejný čas:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "idzakaznika")
```

[37] Vytvořeno autorem.
[38] Vytvořeno autorem.

```
  private Zakaznik zakaznik;
```

Zdrojový kód 26: Část metadat entity *Objednavka* pro test 1. [39]

Mapování na objekty datové vrstvy je provedeno automaticky a není jej třeba dodatečně řešit.

6.3.3 Vyhodnocení testu 1 a optimalizace

Tabulka 3: Průměrné hodnoty výsledků testu 1. [40]

	Průměrný čas jednoho testu	Průměrné využití paměti programem během jednoho testu
Statement	1,58 ms	14469010 B
PreparedStatement	1,02 ms	16928097 B
Hibernate	1,94 ms	20779050 B

Z výsledků v tabulce 2, je zřejmé, že nejlepšího výkonu lze dosáhnout za pomoci JDBC a konkrétně třídy *PreparedStatement*, je tomu tak, jelikož databázový systém se připravil na dotaz, a při spuštění byl odeslán pouze parametr, ne celý dotaz. Při zasílání pokaždé celého dotazu, byli průměrné časové výsledky o 54 % času větší než u předešlého případu, avšak s benefitem nižší paměťové náročnosti. S přístupem *ORM* a frameworkem *Hibernate* je na zpracování informací průměrně potřeba o 90 % více času, nežli s využitím *JDBC* a třídy *PreparedStatement*. Nutno podotknout, že *Hibernate* v tomto testu neměl aktivovanou funkci cache, s ní by mohl dosáhnout lepších výkonů, avšak za cenu rizika méně aktuálních dat. Také potřebuje pro svůj běh více paměti než obě dvě předchozí varianty, důvod je prostý, sama knihovna pro svůj běh vyžaduje operační paměť a jelikož a mnohem rozsáhlejší než knihovny JDBC, potřebuje jí více.

[39] Vytvořeno autorem.
[40] Vytvořeno autorem.

Graf 1: Průběžné časové výsledky jednotlivých testů 1.[41]

6.4 Test 2

Druhý test má za úkol srovnat výkon obou technologií na získání hierarchicky složených dat z databáze a poukázat na problém zpracovávání záznamů do objektů datové vrstvy. Jedná se konkrétně o získávání dat z tabulky *zamestnanec*, kde zaměstnanec může mít svého nadřízeného.

6.4.1 Nativní přístup

Stejně jako v prvním testu, je i zde spuštěn cyklus, který v každé iteraci volá dotaz, avšak následujícím kódem:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM ZAMESTNANEC WHERE IDZAMESTNANCE=" + id);
```

Zdrojový kód 27: Databázové volání testu 2 pro třídu *Statement*.[42]

Pro *PreparedStatement*:

```
stmt2.setInt(1, id);

ResultSet rs = stmt2.executeQuery();
```

[41] Vytvořeno autorem.
[42] Vytvořeno autorem.

Zdrojový kód 28: Databázové volání testu 2 pro třídu *PreparedStatement*.[43]

Proměnná *id* je náhodně generována v rozsahu 0 až 1000. V tomto testu byli třeba k jednotlivým získaným datům, rekurzivně získávat závislá data, proto byl navrhnut následující algoritmus, který má za cíl také mapovat výsledek do objektu datové vrstvy *zamestnanec*:

```
HashMap<Integer, Zamestnanec> zamestnanci = new HashMap<>();
if (rs.next()) {
            Zamestnanec zamestnanec = new Zamestnanec();
            zamestnanec.setIdzamestnance(rs.getInt(1));
            zamestnanec.setPrijmeni(rs.getString(2));
            zamestnanec.setJmeno(rs.getString(3));
            zamestnanec.setTitulpredjmenem(rs.getString(4));
            zamestnanec.setTitulzajmenem(rs.getString(5));
            zamestnanec.setDatumnarozeni(rs.getDate(6));
            zamestnanec.setDatumnastupu(rs.getDate(7));
            zamestnanec.setAdresa(rs.getString(8));
            zamestnanec.setObec(rs.getString(9));
            zamestnanec.setPsc(rs.getString(10));
            zamestnanec.setTelefon(rs.getString(11));
            zamestnanec.setFoto(rs.getString(12));
            zamestnanec.setPlat(rs.getString(13));
            zamestnanec.setNazevpozice(rs.getString(15));
            Integer idNadrizeny = rs.getInt(14);
            if (idNadrizeny != null) {
                if (zamestnanci.containsKey(idNadrizeny))
                    zamestnanec.setNadrizenyzamestnanec(
                        zamestnanci.get(idNadrizeny));
                else {
                   Zamestnanec   zm   =   getZamestnanec(idNadrizeny);
                   zamestnanci.put(idNadrizeny,zm);
                   zamestnanec.setNadrizenyzamestnanec(zm);
                }
            }
            return zamestnanec;
        }
```

Zdrojový kód 29: Algoritmus pro mapování struktury do objektů datové vrstvy.[44]

---

[43] Vytvořeno autorem.
[44] Vytvořeno autorem.

Metoda *getZamestnanec(int),* je rekurzivní volání na sebe sama a vytváří nové databázové volání.

Během každé iterace cyklu byly ukládány naměřené hodnoty a po dokončení poslední iterace byly tyto hodnoty předány zpět řídícímu programu.

6.4.2 Přístup ORM

Pro ORM přístup konkrétně framework *Hibernate*, platí identické podmínky jako pro nativní přístup. V každé iteraci cyklu jsou data získávána pomocí:

```
Session session = em.unwrap(Session.class);
zamestnaneci.add(session.get(Zamestnanec.class,random.nextInt(1000)));
```

Zdrojový kód 30: Kód pro získávání dat z databáze pro test 2 s technologii *Hibernate*[45]

Navíc jsou ve třídě entity (*Zamestnanec*) přidána následující anotace, tak aby byli podmínky identické:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "nadrizenyzamestnanec")
 private Zamestnanec nadrizenyzamestnanec;
```

Zdrojový kód 31: JPA anotace atributu *nadrizenyzamestnanec* třídy *Zamestnanec*[46]

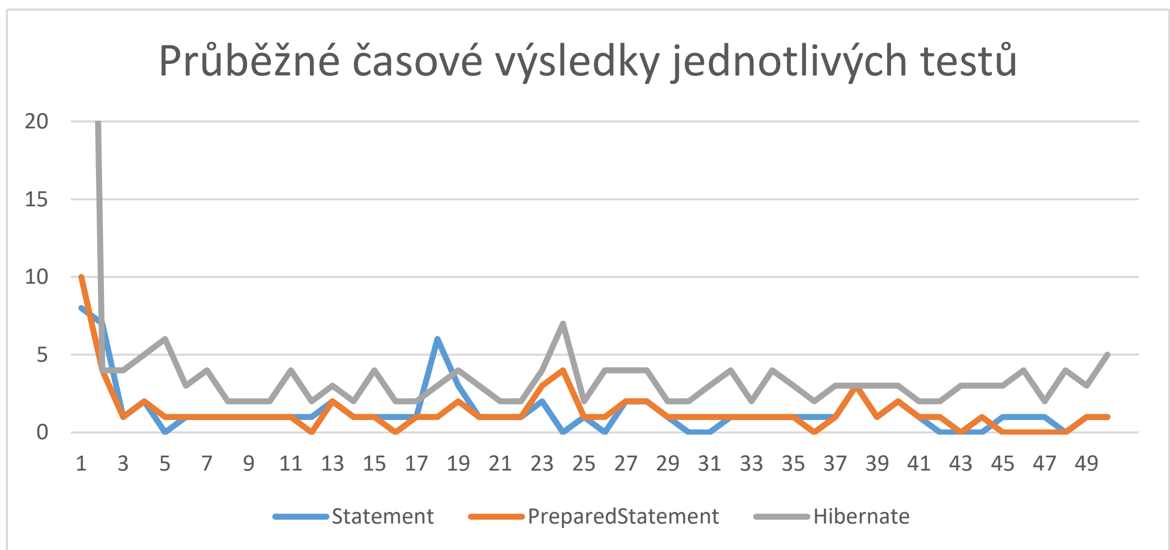
Oproti nativnímu přístup však není potřeba vytvářet žádný algoritmus na sestavení hierarchie objektů. Tuto funkci *Hibernate* vyřeší plně automaticky.

[45] Vytvořeno autorem.
[46] Vytvořeno autorem.

6.4.3 Vyhodnocení testu 2 a optimalizace

Tabulka 4: Průměrné naměřené hodnoty testu 2.[47]

	Průměrný čas jednoho testu	Průměrné využití paměti programem během jednoho testu
Statement	1,38 ms	18966004 B
PreparedStatement	1,34 ms	20051808 B
Hibernate	4,76 ms	18952880 B



Graf 2: Průběžné naměřené časové hodnoty testu 2. [48]

Dle naměřených hodnot v tabulce č. 5 je zřejmé, že co se týče času, nejlepších výsledků opět dosáhnula implementace třídy *PreparedStatement* v těsném závěsu implementace se třídou *Statement* a průměrně nejdéle trval test s využitím frameworku *Hibernate*. Ovšem co se týče paměťové náročnosti, *ORM* implementace dokázala dle naměřených hodnot dosáhnout nižšího vytížení. Je tomu tak z důvodu špatné optimalizace algoritmu zpracovávající data získaná nativním způsobem. Pro optimalizaci těchto problémů u nativního přístupu, by bylo vhodné vytvořit hashovací mapu, do které by se přidávali záznamy z každého spuštění testu. Takto

[47] Vytvořeno autorem.
[48] Vytvořeno autorem.

vzniká zásadní problém, kdy nativní přístup nemusí mít větší výkon než použití ORM frameworku.

Z grafu č.: 2, je také zřejmé, že existuje závislost počtu stejných nebo podobných dotazů na databázi s dobou získání a zpracování dat. U prvního dotazu, totiž databáze provádí *hard parse* a optimalizuje výkon na straně databázového systému. Při dalších spuštění má databázový systém dotaz již k dispozici ze sdíleného fondu.

6.5 Zhodnocení testů

Z obou testů vychází s nejlepšími časovými výsledky použití nativního přístupu s třídou *PreparedStatement* (viz. Tabulka 5), ta svůj dotaz nejprve připravuje na straně databáze a následně posílá pouze parametry. Tímto způsobem se databáze dokáže včas připravit a s malou odezvou odpovědět na dotaz. Se třídou *Statement*, bylo dosaženo jen minimálně nižších výsledků, a to právě kvůli tomu, že se dotaz vždy posílá celý najednou. S použitím frameworku *Hibernate*, bylo dosaženo časově nejhorších výsledků v obou testech, avšak jeho předností je jednoduchost použití, kde je potřeba minimální počet řádků kódu pro přípravu dotazu a zpracování přijatých dat. Další výhoda *Hibernate*, je paměťová optimalizace jednotlivých algoritmů. Pro převod dat získaných nativním přístupem do datové vrstvy je potřeba navrhnout optimalizovaný algoritmu, jinak může nastat situace kdy, *JPA* dosahuje nižších paměťových nároků než *JDBC* (viz. Tabulka 6). U obou přístupů byl potřeba dodatečný databázový ovladač, avšak u *Hibernate*, bylo třeba stáhnout další knihovny, což může být u některých projektů problém. Pro začátečníky může být také problém pochopení zápisu metadat a jednotlivých nastavení *Hibernate*, na druhou stranu, pokud programátor disponuje těmito znalostmi je vývoj několikanásobně rychlejší. U nativního přístupu bylo potřeba psát dlouhé zdrojové kódy obohacené o vnořené SQL dotazy, díky kterým se kód postupem času stává nečitelným, čímž se prodlužuje doba vývoje aplikace.

Tabulka 5: Souhrnné časové výsledky z obou testů. [49]

	Test 1	Test 2
Statement	1,58 ms	1,38 ms
PreparedStatement	1,02 ms	1,34 ms
Hibernate	1,94 ms	4,76 ms

Tabulka 6: Souhrnné paměťové výsledky testů.[50]

	Test 1	Test 2
Statement	14469010 B	18966004 B
PreparedStatement	16928097 B	20051808 B
Hibernate	20779050 B	18952880 B

[49] Vytvořeno autorem.
[50] Vytvořeno autorem.

ZÁVĚR

Byla vytvořena sada aplikací, odpovídající cílům bakalářské práce, která testuje výkonost nativního přístupu a přístupu s použitím objektově relačního mapování. Obě testovací aplikace, umožňují spustit opakovaně zadaný test a během tohoto testu měřit čas běhu a využití operační paměti během získávání dat z databáze a převádění jich do objektů datové vrstvy. Řídící aplikace nabízí možnost zobrazení grafu výsledků jednotlivých testů a uložení hodnot do textového souboru. Při vytváření aplikací bylo dbáno na striktní využívání principů objektově orientovaného programování.

Sada aplikací byla naprogramována v programovacím jazyku Java, který je mezi programátory široce oblíbený a ukázal se jako správná volba, jelikož je možno dohledat mnoho informací, které byli cenné při vývoji.

Z naměřených hodnot v praktické části práce, se potvrdilo, že nativní přístup je opravdu mnohonásobně výkonnější než použití *ORM* frameworku, avšak k těmto výsledkům je nutná vysoká optimalizace zdrojových kódů a záleží tedy na zkušenostech programátora. Naopak se nepotvrdilo, že by *ORM* přístup neměl žádné výhody, spíše naopak, například zkracuje dobu vývoje nebo zpřehledňuje zdrojový kód aplikace. Samozřejmě, že má i své nevýhody týkající se převážně výkonu, nicméně své uplatnění jednoznačně nalézá v menších projektech, kde dlouhé nepřehledné kódy nativního přístupu nabízí jen minimum výhod, například potřeba pouze databázového ovladače.

V teoretické části práce je zřejmé, že zpracování dotazu na straně relační databáze není vždy stejné, probíhá parsování a optimalizace, která může stejně jako použitý framework nebo knihovna ovlivnit výkon při získávání dat. Při vývoji je také nutné mít na paměti, že existují problémy impendační neshody, které není vždy snadné vyřešit.

Při dalším rozšiřování sady aplikací, by bylo vhodné přidat možnost využití jiných implementací *JPA* a porovnávat tak výkon mezi různými frameworky. Stejně tak by mohlo být zajímavé přidání dalších testů a metrik, díky kterým by bylo možné získat nová cenná data. Dalším zajímavým rozšířením by mohla být aplikace určená pro přístup k datům v *NoSQL* databázích a obdobným způsobem porovnat jednotlivé výsledné hodnoty.

POUŽITÁ LITERATURA

[1] *Databázové systémy – trocha teorie* [online]. , 7 [cit. 2019-12-05]. Dostupné z: http://www.ped.muni.cz/wtech/03_studium/cvt4/databaze.pdf

[2] ROUSE, Margaret. Relational database management system guide: RDBMS still on top. *SearchDataManagement* [online]. 2019 [cit. 2019-10-15]. Dostupné z: https://searchdatamanagement.techtarget.com/definition/relational-database

[3] RYCHLÝ, Marek. NoSQL databáze. *Oracle* [online]. Brno, 2013 [cit. 2019-12-05]. Dostupné z: http://www.fit.vutbr.cz/~rychly/public/docs/slides-nosql-databases/slides-nosql-databases.print.pdf

[4] HORVÁTH, Tomáš. Teoretický úvod do relačních databází. *Programujte.com* [online]. 2007 [cit. 2019-10-15]. Dostupné z: http://programujte.com/clanek/2007110801-teoreticky-uvod-do-relacnich-databazi/

[5] *Object Persistence* [online]. 2004, 56 [cit. 2019-11-17]. Dostupné z: http://www.odbms.org/wp-content/uploads/2013/11/004.02-Paterson-Object-Persistence-December-2004.pdf

[6] BYTESCOUT TEAM OF WRITERS. WHAT IS SQL AND WHAT IS IT USED FOR. *Logo* [online]. 2006 [cit. 2019-11-17]. Dostupné z: https://bytescout.com/blog/what-is-sql-and-what-is-it-used-for.html

[7] CÁNEPA, Gabriel. Querying Data with SQL. *Pluralsight* [online]. 2019 [cit. 2019-12-10]. Dostupné z: https://www.pluralsight.com

[8] GARRETT, Alley. What Is Data Integrity? *DZone* [online]. 2019 [cit. 2019-11-17]. Dostupné z: https://dzone.com/articles/what-is-data-integrity-alooma

[9] IAN. What is Referential Integrity? *DATABASE.GUIDE* [online]. 2016 [cit. 2019-11-17]. Dostupné z: https://database.guide/what-is-referential-integrity/

[10] PRIYANKAGUJRAL. SQL | Query Processing. *GeeksforGeeks.org* [online]. [cit. 2019-11-27]. Dostupné z: https://www.geeksforgeeks.org/sql-query-processing/

[11] HOTKA, Dan. Reading Oracle Explain Plans, Part 2: the RBO and the CBO. *LOGICALREAD* [online]. [cit. 2019-11-27]. Dostupné z:

https://logicalread.com/2012/11/26/reading-oracle-explain-plans-part-2-h01/#.XcJkeWaLpaR

[12] JURKE, Tomáš. Ladění dotazů: Execution Plan. *Systemák IT články, návody a jiné* [online]. 2017 [cit. 2019-11-27]. Dostupné z: http://www.systemak.cz/2017/03/09/ladeni-dotazu-execution-plan/

[13] K. BURLESON, Donald. Oracle and Expert Systems Technology. *BC* [online]. [cit. 2019-11-27]. Dostupné z: http://www.dba-oracle.com/t_oracle_net_library_cache.htm

[14] Database Interfaces. *GITTA* [online]. [cit. 2019-11-27]. Dostupné z: http://www.gitta.info/DBSysConcept/en/html/DBLanguages_learningObject2.html

[15] What is persistence in object-oriented programming? *QUORA* [online]. [cit. 2019-11-27]. Dostupné z: https://www.quora.com/What-is-persistence-in-object-oriented-programming

[16] ROUSE, Margaret. Object-oriented programming (OOP). *Essential Guide* [online]. [cit. 2019-11-27]. Dostupné z: https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP

[17] BAYER, Tomáš. Úvod do OOP. *Karlova Univerzita* [online]. Praha [cit. 2019-12-05]. Dostupné z: https://web.natur.cuni.cz/~bayertom/images/courses/Prog2/prog2_0.pdf

[18] Hibernate ORM: What is Object/Relational Mapping? *Hibernate* [online]. [cit. 2019-12-05]. Dostupné z: https://hibernate.org/orm/what-is-an-orm/

[19] MOHANAKRISHNAN. 15 reasons why we need to choose Hibernate over JDBC. *Habile* [online]. [cit. 2019-12-01]. Dostupné z: https://habiletechnologies.com/blog/reasons-to-choose-hibernate-over-jdbc/

[20] ROUSE, Margaret. Java Database Connectivity (JDBC). *TheServerSide* [online]. 2019 [cit. 2019-12-01]. Dostupné z: https://www.theserverside.com/definition/Java-Database-Connectivity-JDBC

[21] Why Use Jdbc. *Sitesbay* [online]. [cit. 2019-12-01]. Dostupné z: https://www.sitesbay.com/jdbc/jdbc-why-use-jdbc

[22] JDBC Drivers. *GeeksforGeeks* [online]. [cit. 2019-11-27]. Dostupné z: https://www.geeksforgeeks.org/jdbc-drivers/

[23] Establishing JDBC Connection in Java. *GeeksforGeeks* [online]. [cit. 2019-11-27]. Dostupné z: https://www.geeksforgeeks.org/establishing-jdbc-connection-in-java/

[24] TYSON, Matthew. What is JPA? Introduction to the Java Persistence API. *JAVAWORLD* [online]. 2019 [cit. 2019-12-01]. Dostupné z: https://www.javaworld.com/article/3379043/what-is-jpa-introduction-to-the-java-persistence-api.html

[25] THOMAS, Daniol. What are the main classes and interfaces of JDBC? *Tutorialspoint* [online]. 2019 [cit. 2019-11-27]. Dostupné z: https://www.tutorialspoint.com/what-are-the-main-classes-and-interfaces-of-jdbc

[26] BERNARD, Borek. Úvod do architektury MVC. *Zdrojak.cz* [online]. 2009 [cit. 2019-12-05]. Dostupné z: https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/

PŘÍLOHY

Příloha A – Testovací projekt 82

PŘÍLOHA A – TESTOVACÍ PROJEKT

pom.xml

logback.xml

config.properties

compare-access.iml

lib/antlr-2.7.7.jar

lib/byte-buddy-1.9.11.jar

lib/classmate-1.3.4.jar

lib/dom4j-2.1.1.jar

lib/FastInfoset-1.2.15.jar

lib/hibernate-commons-annotations-5.1.0.Final.jar

lib/hibernate-core-5.4.4.Final.jar

lib/istack-commons-runtime-3.0.7.jar

lib/jandex-2.0.5.Final.jar

lib/javassist-3.24.0-GA.jar

lib/javax.activation-api-1.2.0.jar

lib/javax.annotation.jar

lib/javax.ejb.jar

lib/javax.jms.jar

lib/javax.persistence.jar

lib/javax.persistence-api-2.2.jar

lib/javax.resource.jar

lib/javax.servlet.jar

lib/javax.servlet.jsp.jar

lib/javax.servlet.jsp.jstl.jar

lib/javax.transaction.jar

lib/jaxb-api-2.3.1.jar

lib/jaxb-runtime-2.3.1.jar

lib/jboss-logging-3.3.2.Final.jar

lib/jboss-transaction-api_1.2_spec-1.1.1.Final.jar

lib/jcommander-1.72.jar

lib/logback-classic-1.2.3.jar

lib/logback-core-1.2.3.jar

lib/ojdbc8.jar

lib/protobuf-java-3.6.1.jar

lib/slf4j-api-1.7.25.jar

lib/stax-ex-1.8.jar

lib/testng-7.0.0.jar

lib/txw2-2.3.1.jar

db/model.dmd

db/scheme.sql

compare-access-jpa/pom.xml

compare-access-jpa/src/main/java/hotovec/petr/compareaccess/jpa/JPAService.java

compare-access-jpa/src/main/java/hotovec/petr/compareaccess/jpa/JPATest.java

compare-access-jpa/src/resource/META-INF/persistence.xml

compare-access-jdbc/pom.xml

compare-access-jdbc/src/main/java/hotovec/petr/compareaccess/jdbc/preparedstatement/
PreparedStatementTest.java

compare-access-jdbc/src/main/java/hotovec/petr/compareaccess/jdbc/preparedstatement/
PreparedStatementTestDao.java

compare-access-jdbc/src/main/java/hotovec/petr/compareaccess/jdbc/statement/
StatementTest.java

compare-access-jdbc/src/main/java/hotovec/petr/compareaccess/jdbc/statement/
StatementTestDao.java

compare-access-app/pom.xml

compare-access-app/src/main/resources/gui.fxml

compare-access-app/src/main/resources/logback.fxml

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/AppController.java

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/GraphicsLib.java

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/GUI.java

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/IOLib.java

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/
LogOutputStreamAppender.java

compare-access-app/src/main/java/hotovec/petr/compareaccess/app/RefreshCallback.java

compare-access-api/pom.xml

compare-access-api/src/main/java/hotovec/petr/compareaccess/api/obj/Objednavka.java

compare-access-api/src/main/java/hotovec/petr/compareaccess/api/obj/Prepravnimoznost.java

compare-access-api/src/main/java/hotovec/petr/compareaccess/api/obj/
Prepravnispolecnost.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/obj/Zakaznik.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/obj/Zamestnanec.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/Result.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/Results.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/Test.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/TestException.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/Tests.java
compare-access-api/src/main/java/hotovec/petr/compareaccess/api/TestType.java
aplikace/compare-access-api-1.0.jar
aplikace/compare-access-jdbc-1.0.jar
aplikace/compare-access-jpa-1.0.jar
compare-access-app-1.0.jar