Objektově orientovaný přístup k perzistentní vrstvě v prostředí Java

Object Oriented Database Access in Java

Roman Janás

Bakalářská práce
2014

*** nascannované zadání str. 1 ***

*** nascannované zadání str. 2 ***

ABSTRAKT

Předmětem bakalářské práce *„Objektově orientovaný přístup k perzistentní vrstvě v prostředí Java“* je analýza dostupných ORM aplikačních rámců pro programovací jazyk Java. Cílem je zjistit, který aplikační rámec je v současnosti nejlépe použitelný. Obsahem práce je také srovnání s technologií JDBC - porovnání výkonnosti jednotlivých řešení a popis používaných návrhových vzorů.

Praktická část názorně ukazuje práci se zvolenými aplikačními rámci a výsledky výkonnostních testů.

Klíčová slova: ORM, Java, návrhové vzory, JDBC, Hibernate, Cayenne

ABSTRACT

The subject of the submitted thesis *“Object Oriented Database Access in Java”* is analysis of available ORM frameworks for programming language Java. The goal is to determine which framework is the most applicable. The content is also comparison with JDBC technology – comparison of efficiency of each solutions and description of used design patterns.

The practical part clearly shows work with chosen frameworks and results of efficiency tests.

Keywords: ORM, Java, design patterns, JDBC, Hibernate, Cayenne

Na tomto místě bych rád poděkoval Ing. Janu Šípkovi za cenné připomínky a odborné rady, kterými přispěl k vypracování této bakalářské práce. Dále děkuji svým kolegům z Invelect, s.r.o. za poskytnutou pomoc při zpracování praktické části.

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

OBSAH

ÚVOD 9
I TEORETICKÁ ČÁST 10
1 ÚVOD DO PROBLEMATIKY 11
1.1 Programovací jazyk Java 11
1.2 Technologie JDBC 12
1.2.1 Požadavky, vývoj JDBC 12
1.2.2 Struktura JDBC 13
1.2.3 Připojení k databázi 14
1.2.4 Základní přístup k databázi 16
1.2.5 Ukončení připojení k databázi 17
2 OBJECT RELATIONAL MAPPING 18
2.1 Podpora perzistence v Javě 19
2.1.1 Proprietární řešení 19
2.1.2 JDBC 20
2.1.3 Enterprise JavaBeans 20
2.1.4 Java Data Objects 21
2.1.5 Potřeba dalšího standardu 21
2.1.6 Java Persistence API 23
3 APLIKAČNÍ RÁMCE VHODNÉ PRO JAZYK JAVA 25
3.1 Analýza aplikačních rámců a jejich popis 25
3.2 Hibernate 26
3.2.1 Architektura 26
3.2.2 Základní rozhraní 28
3.3 Apache Cayenne 30
3.3.1 Cayenne Modeler 30
3.3.2 Základy práce s Cayenne 31
4 NÁVRHOVÉ VZORY PRO NAČÍTÁNÍ DAT 34
4.1 Table Data Gateway 34
4.2 Row Data Gateway 35
4.3 Active Record 35
4.4 Data Mapper 36
4.5 Data Transfer Object 37
4.6 Data Access Object 38
4.7 Lazy Load 38
5 GENEROVÁNÍ CRUD OPERACÍ 40
5.1 Možnosti generování CRUD vybranými aplikačními rámci 40
5.2 Shrnutí 42
II PRAKTICKÁ ČÁST 43
6 PŘÍPRAVA TESTOVACÍHO PROSTŘEDÍ 44
6.1 Testovací databáze 44
6.1.1 Program PrepareTestData 45

6.2 REÁLNÁ DATABÁZE 46
7 PŘÍKLADY VYUŽITÍ JDBC A ORM 48
7.1 POPIS ZKUŠEBNÍCH PROGRAMŮ 48
7.1.1 Třídy Configuration, Generator a Log 50
7.1.2 Popis JDBC implementace 52
7.1.3 Popis Hibernate implementace 53
7.1.4 Popis Cayenne implementace 54
7.2 POPIS FINÁLNÍCH PROGRAMŮ 55
7.2.1 Popis finální JDBC implementace 57
7.2.2 Popis finální Hibernate implementace 58
7.2.3 Popis finální Cayenne implementace 59
7.3 SROVNÁNÍ TECHNOLOGIÍ HIBERNATE, CAYENNE A JDBC 60
8 TESTOVÁNÍ RYCHLOSTI CRUD OPERACÍ 62
8.1 CRON – SPUŠTĚNÍ TESTŮ 62
8.2 OPERACE SELECT 62
8.2.1 Jednoduchý SELECT 63
8.2.2 SELECT pomocí generovaných metod 63
8.2.3 SELECT obsahující join 64
8.3 OPERACE INSERT 65
8.4 OPERACE UPDATE 65
8.5 OPERACE DELETE 66
8.6 SHRNUTÍ VÝSLEDKŮ 67
ZÁVĚR 68
ZÁVĚR V ANGLIČTINĚ 70
SEZNAM POUŽITÉ LITERATURY 72
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 74
SEZNAM OBRÁZKŮ 75
SEZNAM TABULEK 76
SEZNAM PŘÍLOH 77

ÚVOD

V současné době je nejpoužívanějším řešením pro přístup k databázi z programovacího jazyku Java technologie *Java Database Connectivity* (JDBC). Nevýhodou této technologie je, že vyžaduje vytváření dotazů pomocí *Structured Query Language* (SQL). To vylepšuje technologie *Object Relational Mapping* (ORM), kde k tabulkám můžeme přistupovat jako k objektům *Plain Old Java Object* (POJO), což je v prostředí Java přirozenější přístup.

Motivací pro vypracování této práce byla nejasnost a spory o to, který aplikační rámec je nejvhodnější pro dané aplikace využívající práci s perzistentní vrstvou na platformě Java. Existují skupiny vývojářů, kteří prosazují Hibernate nebo Apache Cayenne technologii, třeba jen z toho důvodu, že se ji náhodou naučili a poté o ní tvrdí, že je nejvhodnější bez jakékoliv další rozumné argumentace. V práci se budu zabývat právě důvody, proč zvolit který aplikační rámec a pro jaké typy problémů bude vhodný.

Teoretická část se bude zabývat popisem JDBC a ORM, návrhovými vzory, které se používají pro přístup k databázi, poskytne srovnání výhod a nevýhod JDBC a zvolených ORM aplikačních rámců a také možnosti generování mapovacích objektů. Cílem práce je podat ucelený pohled na možnosti práce s perzistentní vrstvou v Javě.

Praktická část práce představí programy pro měření rychlosti databázových operací, které byly použity k měření časové náročnosti základních databázových operací. Jako první budou představeny testovací programy s náhodně generovanými daty a dále bude následovat ucelenější ukázka využívající kopii reálné databáze – data set české Wikipedie. V závěru praktické části budou vyhodnoceny změřená data a prezentovány formou grafů a tabulek.

Cílem celé práce je ukázat současné možnosti při práci s perzistentní vrstvou a nejvhodnější řešení pro různé aplikace. Důkazy výsledků budou časová náročnost ekvivalentních operací a subjektivní názor na komfort práce programátora při tvorbě projektů pomocí zvolených řešení, škálovatelnosti a udržovatelnosti kódu.

I. TEORETICKÁ ČÁST

1 ÚVOD DO PROBLEMATIKY

Na následujících stránkách bude uveden popis programovacího jazyku Java a technologie JDBC. ORM jako obsáhlejšímu tématu hlavní části práce bude věnována následující podkapitola. Z důvodu logické návaznosti a vazby na ORM bude následovat kapitola o aplikačních rámcích s konkrétní implementací přístupu k perzistentní vrstvě. Bude vysvětleno, co tyto technologie znamenají, důvody jejich vzniku a detailní popis.

Následující podkapitola se bude zabývat návrhovými vzory používanými ve spojení s těmito technologiemi pro přístup k databázi. Nejdříve se soustředíme na novinky v nedávno vydané verzi Java 8. Následující podkapitola popíše technologii JDBC, z toho důvodu, že v praktické části budou představeny programy pro srovnání rychlosti operací nad databází a jeden z nich je napsaný právě pomocí technologie JDBC.

Problematika bude vysvětlována postupně dle vzniku každé technologie. Budou tak jednoduše pochopitelné motivace tvůrců jednotlivých technologií a bude zřetelné, z čeho vycházeli a co se snažili vylepšit.

Z technologií používaných v této práci je nejstarší Java, následuje JDBC a dále nejnovější technologie ORM a její implementace Hibernate a Apache Cayenne.

1.1 Programovací jazyk Java

Java patří mezi interpretované programovací jazyky. Kompiluje se do byte kódu, který překládá *Java Virtual Machine* (JVM). Tento systém zaručuje vysokou míru přenositelnosti kódu (oproti např. C nebo C++). V době vypracování této páce vyšla verze 8 (konkrétně verze 8 Update 5). Protože hlavní verze 8 vyšla teprve nedávno, následující seznam shrnuje, co přináší nového:

- přináší podporu lambda výrazů, které zkracují zápis častých operací (např. metoda *forEach*),
- argument pro *switch* může být *String* (dříve pouze *int*),
- nově je možné používat anotace k typům (využití pro bezznaménkovou aritmetiku),
- nové, rozumnější API pro práci s datem a časem,
- pro řazení polí lze nyní využít efektivní paralelizaci,
- skriptovací možnosti rozšířilo přidání javascriptového enginu Nashorn.

1.2 Technologie JDBC

Databázové programování obecně bylo jako *"technologická Babylónská věž"*. Vývojář se potýká s desítkami dostupných databázových produktů a každý s aplikací komunikuje vlastním specifickým jazykem. Pokud vznikne potřeba komunikovat s novým databázovým enginem, musí se aplikace naučit (a také vývojář) nový jazyk. Java programátoři by se ale neměli zajímat o překladové problémy. Java má programátorům přinést možnost *„napište jednou, zkompilujte jednou, spusťte kdekoliv,"* takže by to měla umožnit také v programování databází.

SQL byl první klíčový krok ke zjednodušení přístupu k databázi. JDBC API v Javě staví na tomto základu a poskytuje vývojáři sdílený databázový jazyk, kterým aplikace může komunikovat s různými databázovými enginy. Následováním tradice jiných multiplatformních API, jako například AWT, JDBC poskytuje sadu rozhraní, které vytváří společný bod, kde se mohou setkat databázové aplikace a databázové enginy. [1],[15]

1.2.1 Požadavky, vývoj JDBC

Ve spolupráci s předními odborníky na poli databázi vyvinula společnost Sun jednotné API pro databázový přístup – JDBC. Jako součást tohoto procesu neustále mysleli na tyto cíle:

- JDBC by mělo být API na úrovni SQL,
- JDBC by mělo využívat zkušeností z již existujících databázových API,
- JDBC by mělo být jednoduché.

API na úrovni SQL znamená, že JDBC vývojáři dovolí vytvořit SQL příkazy a vložit je do volání Java API. Ve zkratce, v podstatě používá SQL. JDBC ale nechá programátora plynule překládat mezi světem databází a Java aplikací. Např. výsledky z databáze jsou vráceny jako Java objekty a problémy s přístupem jsou vyhazovány jako výjimky.

I přes zmatek způsobený šířením proprietárních přístupů k databázovému API, myšlenka univerzálního přístupu k databázi nebyla nová. Ve skutečnosti Sun čerpal z úspěšných aspektů jednoho takového API, Open Database Connectivity. ODBC bylo vyvinuto, aby vytvořilo jednotný standard pro přístup k databázi v prostředí Windows. Přestože výrobní odvětví přijalo ODBC jako hlavní prostředek pro komunikaci s databází, neznamená to, že i Java. Za prvé, ODBC je C API, které vyžaduje zprostředkující API pro další jazyky. Ale i pro vývojáře v jazyku C, ODBC trpí příliš složitou konstrukcí, která jeho přenos mimo platformu Windows odsoudila k nezdaru. Složitost ODBC vychází z faktu, že složité,

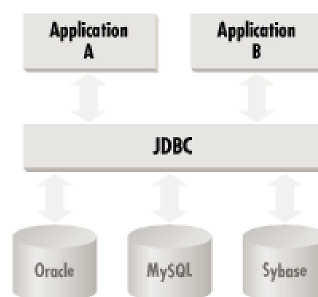
neobvyklé úkoly jsou obaleny v API s jednodušší a mnohem běžnější funkcionalitou. Jinými slovy, pokud chcete trochu porozumět ODBC, bude muset pochopit hodně.

Navíc oproti ODBC, JDBC je silně ovlivněno existujícími API pro databázové programování jako je X/OPEN SQL Call Level Interface. Sun chtěl znovu použít klíčové abstrakce z tohoto API, což by ulehčilo přijetí databázovými dodavateli a využilo by existujících znalostí hlavních ODBC a SQL CLI vývojářů. Sun si také uvědomil, že odvození API z již existujících může poskytnout rychlý vývoj řešení pro databázové enginy, které podporují staré protokoly. Konkrétně Sun pracoval společně se společností Intersolv na vytvoření ODBC mostu, který mapuje JDBC volání na ODBC volání a dává tak Java aplikacím přístup k jakémukoliv Systému Řízení Báze Dat, který ODBC podporuje.

JDBC se snaží zůstat co nejjednodušší a zároveň se snaží poskytnout vývojářům maximální flexibilitu. Rozhodujícím kritériem pro Sun je jednoduchá otázka, které aplikace s databázovým přístupem jsou nejpoužívanější. Jednoduché a běžné úkoly používají jednoduché rozhraní, zatímco splnění neobvyklých úkolů je povoleno skrze specializované rozhraní. Například tři rozhraní zvládnou převážnou většinu přístupů k databázi, JDBC přesto nabízí několik dalších rozhraní pro práci se složitějšími a neobvyklými úkoly. [1],[14]

1.2.2 Struktura JDBC

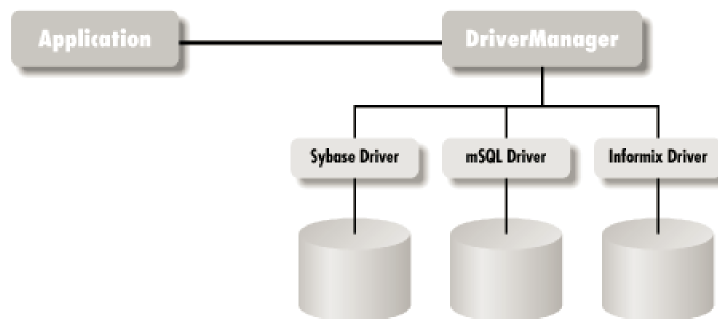
JDBC dosahuje svých cílů prostřednictvím sady Java rozhraní, kde je každé implementováno odlišně jednotlivými dodavateli. Sada tříd, které implementují rozhraní JDBC pro daný databázový engine se nazývá ovladač JDBC. Při vytváření databázové aplikace tak není potřeba vůbec přemýšlet o implementaci těchto podkladových tříd; cílem JDBC je skrýt specifikace každé databáze a nechá vývojáře přemýšlet pouze o vlastní aplikaci. [1]



Obrázek 1: Architektura JDBC [1]

1.2.3 Připojení k databázi

Nyní budou ukázány detaily o JDBC voláních a o tom, jak se používají. *Obrázek 2* ukazuje, jak aplikace používá JDBC ke komunikaci s jednou nebo více databázemi bez toho, aniž by znala podrobnosti týkající se toho, jak je ovladač implementován. Aplikace používá JDBC jako rozhraní, přes které předává všechny požadavky na databázi. [1] [14]



Obrázek 2: JDBC odděluje aplikaci od specifikací jednotlivých implementací databází [1]

Jediná informace která je v aplikaci specifická pro každý ovladač a kterou JDBC potřebuje, je URL databáze. Použitím URL databáze a dalších parametrů vyžadovaných ovladačem (obvykle uživatelské jméno a heslo) si aplikace jako první vyžádá implementaci *java.sql.Connection* od *DriverManageru*. *DriverManager* pak prohledá všechny známé *java.sql.Driver* implementace a najde tu, která odpovídá zadané URL. Pokud vyčerpá všechny implementace a nenajde shodu, tak vyhodí výjimku zpět do aplikace. [1]

Driver po rozeznání URL vytvoří připojení k databázi za použití poskytnutých údajů (přihlašovací jméno a heslo). Toto připojení poté poskytne *DriverManageru* s implementací *java.sql.Connection* reprezentující připojení k databázi. *DriverManager* pak poskytne připojení naší aplikaci. Popsaný proces v kódu vyvolá tento řádek:

```
Connection conn =     DriverManager.getConnection(url,
                      username, password);
```

Ještě je ale potřeba pro *DriverManager* registrovat implementaci ovladače (který pak bude hledat podle poskytnuté URL databáze). To lze udělat několika způsoby.

- Je možné explicitně volat *new*, aby se implementace ovladače načetla.
    - Toto je nejnevhodnější použití, protože pokaždé, když se změní databáze nebo ovladač databáze, je potřeba provést úpravu kódu a rekompilaci.
- Je možné použít vlastnost *jdbc.drivers*.

    - *DriverManager* načte všechny ovladače z tohoto seznamu automaticky, což funguje dobře u aplikací spouštěných z příkazové řádky, kde lze tuto vlastnost specifikovat, nejde to tak ale udělat například u Java appletů.
- Je možné načíst ovladač použitím

```
class.forName(„DrvImplClass�much").newInstance();
```

    - Tento zápis slouží pro dynamické vytváření instancí dané třídy, pokud má programátor k dispozici řetězec reprezentující název požadované třídy. (*newInstance* je potřeba, některé JVM nevolají statický inicializátor, pokud použijeme pouze *class.forName*.)

JDBC tedy používá jednu třídu (*java.sql.DriverManager*) a dvě rozhraní (*java.sql.Driver* a *java.sql.Connection*) pro vytvoření připojení:

- java.sql.Driver
    - *java.sql.Driver* slouží JDBC jako vstupní bod do databáze, tak že odpovídá na požadavek na připojení od *DriverManageru* a poskytuje mu informace o implementaci.
- java.sql.DriverManager,
    - *DriverManager*, na rozdíl od téměř všech ostatních částí JDBC je třída a ne rozhraní. Hlavní zodpovědnost má za udržování seznamu implementací ovladačů a prezentaci zvoleného (dle poskytnuté URL) ovladače aplikaci. *DriverManager* poskytuje metody *registerDriver()* a *deregisterDriver()*, které dovolují ovladačům, aby se samy registrovaly nebo odebraly ze seznamu *DriverManageru*. Výčet registrovaných ovladačů lze zjistit pomocí metody *getDrivers()*.
- java.sql.Connection
    - Třída *Connection* reprezentuje jedno logické databázové připojení. Jinými slovy, *Connection* se používá pro posílání SQL příkazů databázi a k řízení potvrzení nebo zrušení těchto příkazů.

Pokud se něco při vykonávání programu v komunikaci s databází nezdaří, JDBC vyhazuje výjimku *SQLException*. Navíc, oproti běžným výjimkám z Javy, *SQLException* obsahuje chybové informace specifické pro použitou databázi (např. *SQLState* a chybový kód od dodavatele). Pokud dojde současně k několika výjimkám, JDBC ovladač je „zřetězí“

za sebe, měli bychom tedy také kontrolovat, zda výjimek není více voláním metody *getNextException()*. [1],[15]

1.2.4 Základní přístup k databázi

Nejzákladnější druh přístupu k databázi zahrnuje případy, kdy je dopředu známo, jaké příkazy budou databázi odesílány – zda se jedná o aktualizace (*INSERT*, *UPDATE, DELETE*), nebo dotazy (*SELECT*).

Základní přístup k databází začíná objektem *Connection*. Při vytvoření je tento objekt jednoduše přímý odkaz na databázi. Používá se ke generování implementací *java.sql.Statement* svázaných se stejnou databázovou transakcí. Po použití jednoho nebo více *Statementů* generovaných pomocí *Connection*, můžeme být použit k potvrzení nebo odvolání objektů *Statement* asociovaných s *Connection*.

Před odesláním *Statementu* je ale důležité vědět, jaký typ SQL bude odeslán, protože JDBC používá odlišné metody pro posílání dotazů a pro posílání aktualizací. Klíčový rozdíl je skutečnost, že metoda pro dotazy vrací instanci třídy *java.sql.ResultSet*, zatímco metoda pro aktualizace vrací celé číslo (počet vložených, změněných, nebo smazaných záznamů). *ResultSet* poskytuje přístup k datům získaným dotazem.

Nejzákladnější třídy JDBC jsou *Connection*, *Statement* a *ResultSet*. Tyto třídy budou vývojáři používat pokaždé, když budeme používat JDBC. Třída *Connection* byla shrnuta již dříve, nyní budou stručně popsány *Statement* a *ResultSet*.

- Třída *Statement* je nejzákladnější ze tří JDBC tříd reprezentujících SQL statementy. Vykonává všechny základní SQL statementy, které byly výše zmíněny. Obecně, jednoduchá databázová transakce používá pouze jednu ze tří metod pro vykonání statementu. První z nich, *executeQuery()*, přebírá SQL řetězec jako argument a vrací objekt *ResultSet*. Tato metoda by měla být použita pro všechny SQL volání, které očekávají vrácení záznamů z databáze. Na druhou stranu, statementy pro aktualizace vykonává metoda *executeUpdate()*.
  Statement také poskytuje metodu *execute()* pro případy, u kterých nevíme, zda bude SQL, které má být vykonáno, dotaz nebo aktualizace. Většinou k tomu dochází, když SQL generujeme dynamicky. Pokud statement vrátí záznam z databáze, vrací metoda *true*, jinak *false*. Metoda *getResultSet()* pak slouží k získání záznamu.

- *ResultSet* je jeden nebo více záznamů vrácených dotazem v databázi. Třída jednoduše poskytuje několik metod k získání sloupců z výsledku databázového dotazu. Všechny metody pro získání sloupce mají tuto podobu:

```
    type get type(int | String)
```

  ve které argument značí buď pořadí sloupce, nebo jeho název. Vedlejší efekt tohoto návrhu je, že je možné hodnoty ukládat jako jeden typ a číst je jako jiný typ. Například, pokud je potřeba *Date* z databáze jako *String*, lze jej získat jako *String* použitím *resultSet.getString(1)* místo *resultSet.getDate(1)*.

  Protože *ResultSet* vždy pracuje pouze s jedním záznamem z databáze, třída poskytuje metodu *next()* pro vytvoření reference na následující záznam. Pokud *next()* vrátí *true*, znamená to, že *ResultSet* obsahuje další záznam ke zpracování a na tento záznam je možné opět použít *next()* pro získání další reference. Pokud v *ResultSetu* už nezbydou další záznamy, *next()* vrátí *false*. [1],[14],[15]

1.2.5 Ukončení připojení k databázi

Třídy *Conection*, *Statement* a *ResultSet* mají všechny metodu *close()*. Daná implementace JDBC může nebo nemusí požadovat zavření těchto objektů před dalším použitím. Některé to tedy mohou vyžadovat, proto je vždy správné instance těchto objektů zavírat po jejich použití.

Je dobré si pamatovat, že uzavření *Connection* automaticky uzavře i všechny *Statements*, které má k sobě přiřazené. Podobně zavření *Statementu* způsobí implicitní zavření všech přidružených *ResultSetů*. Zavření *Connection* před potvrzením může vést ke ztrátě nepotvrzených transakcí (pokud je vypnutý auto-commit). [1]

Autor knihy *„Database programming with JDBC“* , George Reese, upozorňuje na případy z praxe, kdy zavření samotného *Connection* nezavře přidružené *Statementy* a k nim přidružené *ResultSety*. Radí, abychom vždy zavírali zmíněné objekty explicitně a neohrozili tak přenositelnost kódu.

2 OBJECT RELATIONAL MAPPING

„Model doména má třídu. Databáze má tabulku. Vypadají docela podobně. Mělo by být jednoduché převést jeden na druhý automaticky.“ To je myšlenka, která někdy napadla asi každého vývojáře, nebo jsme o tom přemýšleli při psaní dalšího DAO abychom převedli JDBC na něco více objektově orientovaného. Doménový model vypadá docela podobně jako relační model databáze, takže se zdá, že by mělo být jednoduché najít způsob, jak spolu mohou tyto dva modely komunikovat.

Technika pro překonání propasti mezi objektovým a relačním modelem je známá jako objektově-relační mapování, často nazýváno jako O-R mapování, nebo jednoduše ORM. Tento termín vychází z představy, že je nějakým způsobem možné mapovat pojmy z jednoho modelu na jiný, s cílem vysvětlit prostředníkovi, jak řídit automatické transformace z jednoho stavu na druhý.

Před specifikací objektově-relačního mapování ještě bude definováno, jak by výsledné řešení *mělo* vypadat.

- **Objekty, ne tabulky.** Aplikace by měla být psána podle pravidel doménového modelu, neměla by být vázána na relační model. Musí existovat způsob, jak pracovat a provádět dotazy proti doménovému modelu, bez nutnosti vyjádřit to v relačním jazyku tabulek, sloupců a cizích klíčů.
- **Pohodlí, ne neznalost.** Mapovací nástroje by měly být použity někým, kdo je obeznámený s relační technologií. O-R mapování nemá za cíl vývojáře zachraňovat od pochopení mapovacích problémů, nebo je skrývat. Jsou míněny pro ty, kteří rozumí těmto problémům, a vědí, co chtějí, ale nechtějí psát tisíce řádků kódu, aby se vypořádali s problémem, který už někdo vyřešil.
- **Nenápadný, ne transparentní.** Je nerozumné očekávat, aby perzistentní vrstva byla transparentní, protože každá aplikace potřebuje, aby mít kontrolu nad objekty, které uchovává a být si vědomá životního cyklu entity. Řešení perzistence by nemělo zasahovat do modelu domény, ale zároveň nesmí být od doménových tříd vyžadováno, aby rozšiřovaly třídy nebo implementovaly rozhraní, aby byly perzistentní.
- **Stará data, nové objekty.** Je daleko pravděpodobnější, že aplikace se budou zaměřovat na existující relační databázová schémata, než že by vytvářely nové.

Podpora starých schémat je nejdůležitější případ použití, který vznikne, a je docela možné, že takové databáze nás všechny přežijí.

- **Dostatečně, ale ne příliš.** Podnikoví vývojáři mají problémy, které mají řešit, a potřebují funkce vhodné pro řešení těchto problémů. To co nemají rádi je, aby byli nuceni chápat těžkopádný perzistentní model, který představuje velké režie, protože řeší problémy, o kterých si někteří ani nemyslí, že by problémy byly.
- **Lokální, ale mobilní.** Perzistentní reprezentace dat nemusí být vytvářena jako plnohodnotný vzdálený objekt. Distribuce je něco, co existuje jako součást aplikace, ne jako část perzistentní vrstvy. Entity, které obsahují perzistentní stav, ale musí být schopny cestovat do kterékoliv vrstvy, která je potřebuje. [2]

2.1 Podpora perzistence v Javě

V začátcích platformy Java existovaly rozhraní, které poskytovaly brány do databáze a k abstrakci mnoha doménově specifických perzistentních požadavků podnikových aplikací. Dále budou probírány současné a minulé řešení perzistence v Javě a jejich role v podnikových aplikacích. [2],[16]

2.1.1 Proprietární řešení

Možná bude překvapivé zjištění, že řešení na základě objektově-relačního mapování bylo okolo již dlouho, déle než Java samotná. Produkty jako Oracle TopLink začínaly ve Smalltalku, než přešly do Javy. Obrovská ironie v historii perzistence v Javě je, že jedna z prvních implementací entity beans byla vlastně demonstrována přidáním vrstvy entity beanu nad objekty mapované TopLinkem.

Dvě z nejpopulárnějších proprietárních API byly TopLink v komerční sféře a Hibernate v open source komunitě. Komerční produkt jako TopLink byl dostupný v dřívějších dnech Java a byl úspěšný, ale jeho techniky nikdy nebyly standardizovány pro Java platformu. Bylo to později, kdy se open source objektově-relačně mapované řešení jako Hibernate stalo populární, že změny v perzistenci v Java platformě přešel, což vedlo k přiblížení objektově-relačního mapování jako upřednostňovaného řešení.

Tyto dva produkty a další mohly být lehce integrovány se všemi hlavními aplikačními servery a mohly poskytovat aplikacím všechny funkce perzistence, které potřebovaly. Vývojáři byli naprosto spokojení, když mohli používat tyto produkty třetích stran, hlavně

s ohledem na to, že nebylo dostupné žádné běžné a ekvivalentní standardy v dohlednu. [2],[16]

2.1.2 JDBC

Ironií JDBC je, že i přestože programová rozhraní jsou přenosná, jazyk SQL není. Navzdory mnoha pokusům o standardizaci, stále je vzácné napsat komplexnější SQL, které nezměněné poběží na dvou hlavních databázových platformách. I když jsou dialekty SQL podobné, každá databáze pracuje jinak v závislosti na struktuře dotazu a v mnoha případech potřebuje doladění specifické podle dodavatele.

Existují také problémy v úzké vazbě mezi kódem a SQL textem. Vývojáři jsou neustále pokoušeni kouzlem ready-to-run SQL dotazů místo dynamického konstruování za běhu. Toto je velmi atraktivní programovací model, ale jen do doby, než si uvědomíte, že aplikace musí podporovat nového databázového dodavatele, který nepodporuje SQL dialekt, který jste používali dosud. [2]

2.1.3 Enterprise JavaBeans

První vydání platformy Java 2 Enterprise Edition obsahovalo nové řešení perzistentní vrstvy v Javě formou entity beanů, což je část EJB komponent. Zamýšleny jako kompletní izolace vývojáře od vypořádávání se přímo s perzistencí, představily řešení postavené na rozhraních, kde konkrétní třída obsahující bean nebyla nikdy přímo použita klientským kódem. Na místo toho, specializovaný kompilátor beanů vytvářel implementaci rozhraní beanu, aby usnadnila např. perzistenci, bezpečnost a řízení transakcí, delegováním business logiky do implementace entity beanu. Entity beany byly konfigurovány použitím kombinace standardních a podle dodavatele specifických zaváděcích popisovačů, které se staly známé pro svou komplikovanos.

Pravděpodobně bude správné říct, že entity beany byly navrženy na řešení mnohem komplikovanějších problémů, než které zkoušeli skutečně řešit. Je ironií, že první vydání této technologie postrádalo mnohou funkčnost potřebnou k implementaci realistické aplikace.

Specifikace EJB 2.0 vyřešila mnoho problémů objevených ve dřívějších verzích. Byl představen pojem kontejnerově-řízené entity beany, kde se třídy obsahující beany staly abstraktními a server byl zodpovědný za generování pod-tříd pro řízení perzistentních dat.

Toto vydání také ukázalo EJB QL, dotazovací jazyk navržený, aby pracoval s entitami tak, že mohly být přenositelně kompilovány do jakéhokoliv SQL dialektu.

Navzdory vylepšením představeným v EJB 2.0, jeden hlavní problém stále přetrvával: přílišná složitost. Specifikace předpokládala, že vývojové nástroje izolují vývojáře od konfigurace a řízení mnoha artefaktů potřebných pro každou bean. Naneštěstí realizace takových nástrojů trvala dlouho a tak toto břemeno spadlo na bedra vývojářů. I tak ale velikost a rozsah EJB aplikací vzrůstal. Vývojáři se cítili opuštění v moři složitosti bez slibované infrastruktury, která by je udržela na hladině. [16]

2.1.4 Java Data Objects

Částečně v důsledku selhání perzistentního modelu EJB, částečně kvůli frustraci z toho, že neexistuje uspokojující standardizované perzistentní API, vznikl pokus o novou specifikaci pro perzistenci. Java Data Objects byly inspirovány a podporovány hlavně dodavateli objektově orientovaných databází a nikdy nebyly opravdu přijaty tradiční programátorskou komunitou. Od dodavatelů to vyžadovalo vylepšení byte kódu doménových objektů tak, aby produkovaly soubory tříd, které byly na binární úrovni kompatibilní napříč všemi dodavateli a každý kompatibilní produkt dodavatele musel být schopen tyto třídy produkovat a pracovat s nimi. JDO také mělo dotazovací jazyk, který byl orientovaný objektově, což se nelíbilo uživatelům relačních databází, kteří představovali drtivou většinu.

JDO dosáhlo stavu, kdy se z něj stalo rozšíření JDK, nikdy se ale nestalo skutečnou součástí platformy Java. Obsahovalo spoustu dobré funkcionality a bylo přijato malou komunitou oddaných uživatelů, kteří se ji snažili zoufale prosazovat. Bohužel, hlavní komerční dodavatelé neměli stejný názor na to, jakým způsobem má být implementován perzistentní aplikační rámec. Několik jich ale tuto specifikaci podporovalo, tak se o JDO mluvilo, ale používáno bylo spíše vzácně. [17]

2.1.5 Potřeba dalšího standardu

Softwaroví vývojáři věděli, co chtějí, ale mnoho jich to nemohlo najít v existujících standardech, tak se rozhodli poohlédnout jinde. To, co našli, byla škála proprietárních aplikačních rámců – komerčních i open source. Mnoho produktů, které implementovaly tyto technologie, přijalo za vlastní perzistentní model, který neměl zasahovat do doménového modelu. Pro tyto produkty existovala perzistentní vrstva, která nenarušovala business

objekty, tak jako entity beany, a nemusely si být vědomy technologie, která je uchovává. Nemusely implementovat žádná rozhraní nebo rozšiřovat speciální třídy. Vývojář s nimi mohl jednoduše zacházet, jako by to byly obyčejné Java objekty, pak je mapovat na perzistentní úložiště a použít perzistentní API k jejich uchování. Právě protože tyto objekty byli regulérní Java objekty, tento perzistentní model vešel ve známost jako Plain Old Java Object perzistence.

Jak se začaly Hibernate, TopLink a další perzistentní aplikační rámce usazovat v aplikacích a perfektně splňovaly jejich potřeby, byla často kladena otázka: „Proč se obtěžovat s aktualizací standardu EJB, tak aby odpovídal tomu, co tyto produkty již dělají? Proč jednoduše nepoužívat dál tyto produkty, jako už léta, nebo proč jednoduše nestandardizovat nějaký open source produkt, jako například Hibernate?“ Ve skutečnosti existuje mnoho důvodů, proč to takto nemůže být a proč by to byl špatný nápad, i kdyby to šlo.

Standard jde mnohem hlouběji než produkt a jeden produkt (ani produkt tak úspěšný jako Hibernate nebo TopLink) nemůže ztělesňovat specifikaci, i když ji může implementovat. V samotném jádře je záměr specifikace, aby byla implementována různými dodavateli, kteří by měli odlišné produkty nabízející standardní rozhraní a sémantiku předpokládanou aplikacemi bez nutnosti vázání aplikace na jeden produkt.

Vázání standardu na open source projekt jako Hibernate by bylo problematické pro samotný standard, ale pravděpodobně ještě horší pro Hibernate. Představme si specifikaci, která je založena na specifické verzi kódu open source projektu a jak matoucí by to bylo. Teď si představte open source software projekt, který by se nemohl změnit, nebo pouze v omezených verzích kontrolovaných speciálním výborem každé dva roky, na rozdíl od změn, které se rozhodne udělat sám projekt kdykoliv.

Přestože pro jednotlivce nebo malé vývojářské týmy se může zdát standardizace zbytečná, je velmi důležitá pro firemní použití. Softwarové technologie představují pro mnoho společností velké investice a musí se měřit možná rizika, když je v sázce takové množství peněz. Používání standardizované technologie snižuje riziko tím, že společnostem dovoluje změnit dodavatele, pokud původní dodavatel nedokáže dodat požadovaný produkt.

Nemluvě o přenositelnosti kódu, hodnota standardizování technologií se projevuje i v jiných oblastech. Vzdělání, návrhové vzory a průmyslová komunikace – to je pouze několik výhod, které přináší standardizace. [2],[16],[17]

2.1.6 Java Persistence API

Java Persistence API je jednoduchý aplikační rámec založený na POJO pro řešení perzistence v Javě. Ačkoliv objektově-relační mapování je hlavní součást API, nabízí také řešení pro integrování perzistence do podnikových aplikací.

Po letech stížností na složitost vytváření aplikací v jazyku Java bylo hlavním tématem vydání Java EE 5 hlavně „jednoduchost vývoje“. EJB 3.0 se chopilo vedení a objevilo způsob, jak udělat EJB jednodušší a produktivnější.

V případě session beanů a message-driven beanů bylo řešením pro zjednodušení odstranění několika obtížných implementačních požadavků a nechat komponenty vypadat více jako prosté Java objekty.

Entity beany ale byly zavrhnuty úplně a s návrhem se začalo od začátku. Java Persistence API se zrodilo z rozpoznání požadavků z praxe a ze současných proprietárních řešení, které se používaly pro řešení jejich problémů. Ignorovat tyto zkušenosti by bylo hloupé.

A tak vystoupili přední dodavatelé řešení založených na objektově-relačním mapování a standardizovali osvědčené způsoby reprezentované jejich produkty. První to byli Hibernate a TopLink s dodavateli EJB, později následováni dodavateli JDO.

Roky zkušeností v oboru spojené s posláním zjednodušit vývoj vedly k vytvoření specifikace, která mohla skutečně zahrnout programovací modely nabízené platformou Java SE 5. Použití anotací vyústilo v nový způsob používání perzistence, jaký zatím nebyl k nikde k vidění.

Výsledná specifikace EJB 3.0 skončila rozdělením na tři rozdílné části a ve třech různých dokumentech, z nichž třetí byl o Java Persistence API. Byla to samostatná specifikace popisující perzistentní model v prostředích Java SE i Java EE.

V době, kdy JPA začínalo, se už ORM vyvíjelo celou dekádu. Bohužel na vytvoření prvotní specifikace zbylo relativně velmi málo času (přibližně dva měsíce), takže ne každá funkcionalita, se kterou se počítalo, mohla být součástí vydání. I tak ale bylo specifikováno obrovské množství funkčností, ale s podmínkou, že něco zůstane pro menší vydání a něco zůstalne dodavatelům, aby to v mezičase implementovali proprietárně.

Následující vydání, JPA 2.0, obsahovalo velké množství funkčnosti, která nebyla představena v první verzi, zejména té, která byla nejvíce požadována uživateli. Poskytnutím

kompletnější sady perzistentních funkcí se docílilo toho, že se nyní mnohem méně aplikací bude muset uchýlit k rozšířením dodavatele.

Některé funkčnosti vytvořené ve verzi 2.0 jsou: přidání dalších možností mapování, flexibilní způsob sloužící k určení, jakým způsobem může poskytovatel přistupovat ke stavu entity, rozšíření JPQL a objektově orientované kritériové API pro tvorbu dynamických dotazů. [2],[17]

3 APLIKAČNÍ RÁMCE VHODNÉ PRO JAZYK JAVA

Kapitola bude popisovat aplikační rámce pro práci s ORM v prostředí Java. Podkapitoly budou obsahovat seznam současných dostupných aplikačních rámců, seznámení s nimi a zdůvodnění výběru aplikačních rámců pro testování rychlosti databázových operací ve srovnání s JDBC. Následovat budou konkrétní popisy výhod a nevýhod zvolených aplikačních rámců.

Aplikační rámec je druh software podporující vývoj a programování jiných softwarových projektů. Obsahuje sady podpůrných programů, API pro práci s nimi a podporuje návrhové vzory nebo doporučené postupy používané při vývoji.

Byly zvoleny tyto kritéria pro výběr aplikačních rámců, seřazena sestupně dle důležitosti: aktuálnost projektu, funkcionalita, podpora (např. aktivita komunity), snadnost použití.

3.1 Analýza aplikačních rámců a jejich popis

Následující seznam pokrývá několika větších projektů s krátkým popisem každého z nich a zdůvodněním pro zařazení (nebo odebrání) do testovaných aplikačních rámců v praktické části.

- **Hibernate** je řešením O-R mapování pro platformu Java. Byl vytvořen v roce 2001 Gavinem Kingem jako open source perzistentní aplikační rámec. Hibernate podporuje specifikaci JPA 2.1, má obrovskou komunitu a poslední stabilní verze vyšla 2. 4. 2014. Z těchto důvodů **byl zařazen** mezi testované aplikační rámce.
- **EclipseLink** vznikl z projektu TopLink od Oracle a podporuje specifikaci JPA 2.1. Uživatelská základna je ale menší než u Hibernate a také neposkytuje tak komplexní funkcionalitu jako Hibernate. Poslední stabilní verze vyšla 18. 9. 2013 a od začátku roku 2014 také značně klesla aktivita přispěvatelů do toho projektu. EclipseLink **nebyl zařazen** do testovacích aplikačních rámců, protože je podobný technologii Hibernate, ale má méně aktivních uživatelů a poslední stabilní verze byla vydána před relativně dlouhou dobou (přestože projekt je stále označován jako aktivní).
- **Cayenne** je produktem společnosti Apache. Nepodporuje specifikaci JPA, ale na druhou stranu z ní čerpá mnoho nápadů. Poslední stabilní verze je z roku 2011, nicméně tento aplikační rámec **byl zařazen** do testování, protože obsahuje Cayenne Modeler, nástroj oblíbený a používaný mezi vývojáři, ve kterém lze vytvořit

databázi, nebo již existující databázi exportovat a následně generovat potřebné perzistentní objekty.

- Dále do testování nebyly zařazeny **TopLink** – protože z něj vychází EclipseLink, **Java Data Objects**, protože v praktické části je použita relační databáze, **EJB** pro svou složitost a další aplikační rámce, např. neaktivní projekty, komerční projekty, případně další, které nejsou tolik známé nebo používané.

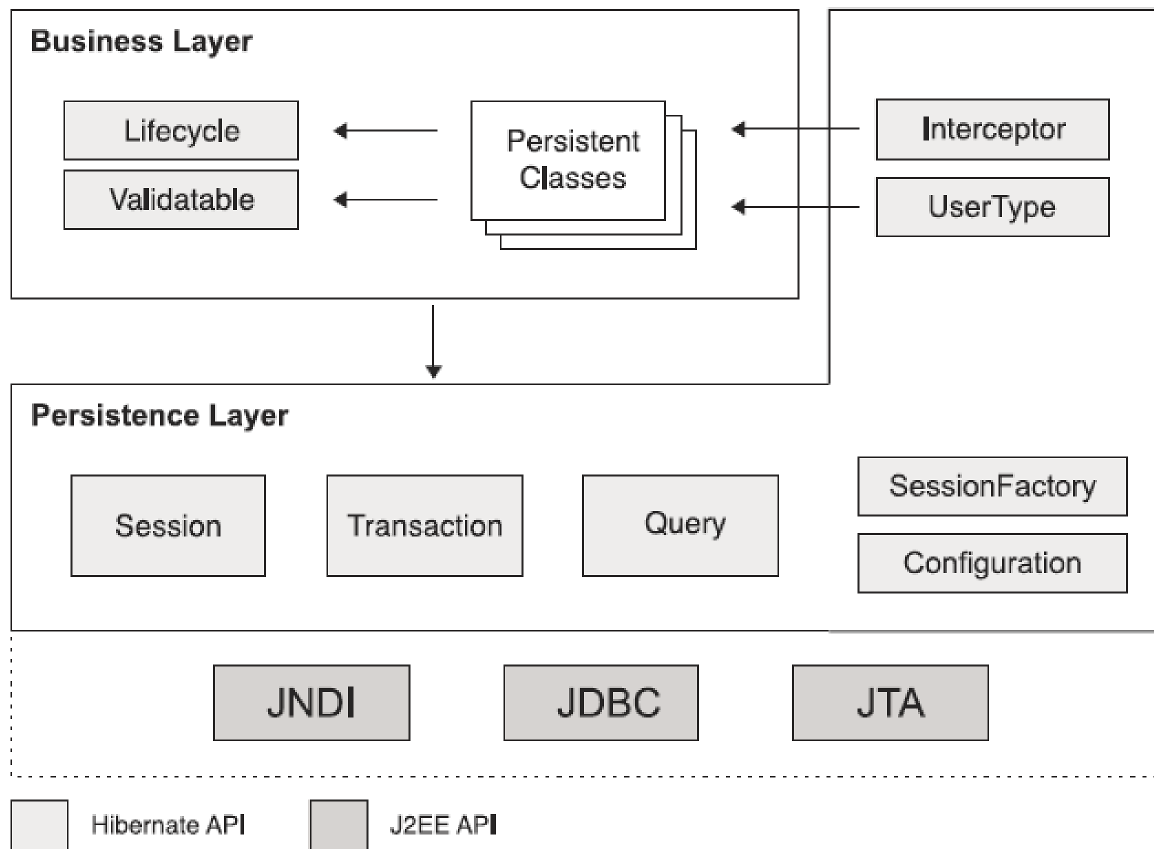
Do závěrečného testování se nakonec dostaly aplikační rámce Hibernate pro svou komplexnost a podporu JPA 2.1 a Cayenne, oblíbený pro svůj doplněk Cayenne Modeler. Tyto dva aplikační rámce budu srovnávat v praktické části s technologií JDBC v testech rychlosti ekvivalentních operací. [3],[6],[8],[9]

3.2 Hibernate

Popis implementace bude začínat popsáním architektury technologie Hibernate a vysvětlením jejich rozhraní. Srovnání s technologií JDBC bude v praktické části, kdy už budeme z hlediska teorie rozumět i aplikačnímu rámci Apache Cayenne a zhodnocení tedy bude objektivní v rámci těchto tří technologií.

3.2.1 Architektura

Programová rozhraní jsou první věcí, kterou je potřeba pochopit o Hibernate, aby jej bylo možné používat v perzistentní vrstvě aplikace. Hlavním úkolem návrhu API je držet rozhraní mezi jednotlivými komponenty aplikace odděleně, jak jen to bude možné. V praxi ale nejsou ORM API vyloženě malá. Nicméně nemusíme se obávat, nemusíme pochopit všechny rozhraní ihned. Na obrázku *Obrázek 3* vidíme roli nejdůležitějších rozhraní Hibernate v business a perzistentní vrstvě. Business vrstvu ukážeme nad perzistentní vrstvou, přestože v tradičních aplikacích se chová business třída jako klient perzistentní vrstvy.



Obrázek 3: Vysokoúrovňový přehled vrstvové architektury Hibernate API [3]

Obrázek 3 může být klasifikováno přibližně takto:

- rozhraní volaná aplikací vykonávající základní CRUD operace. Tyto rozhraní jsou hlavním bodem závislostí aplikační obchodní / řídicí logiky na Hibernate. Obsahují *Session*, *Transaction* a *Query*,
- rozhraní volaná infrastrukturou aplikace pro konfiguraci Hibernate, nejdůležitější je třída *Configuration*,
- *Callback* rozhraní které dovolují aplikaci reagovat na události objevující se uvnitř Hibernate, třídy *Interceptor*, *Lifecycle* a *Validatable*,
- rozhraní povolující rozšíření Hibernate pro jeho mapovací funkce, jako jsou *UserTpye*, *CompositeUserType* a *IdentifierGenerator*. Tato rozhraní jsou implementována kódem aplikační infrastruktury (pokud je to nezbytné).

Hibernate využívá existující Java API, včetně JDBC, JTA a JNDI. JDBC poskytuje základní úroveň abstrakce funkčnosti běžné pro relační databáze, což dovoluje Hibernate podporovat téměř jakoukoliv databázi s JDBC ovladačem. JDNI a JTA umožňují Hibernate integraci s J2EE aplikačními servery.

Dále se budeme věnovat hlavním rozhraním aplikačního rámce Hibernate. Nebude pokryta detailně sémantiku všech metod, pouze role těch důležitých. [3] [5]

3.2.2 Základní rozhraní

Těchto pět základních rozhraní bude téměř určitě používat jakákoliv aplikace využívající Hibernate.

- **Rozhraní Session**.

```
Session session = sessionFactory.openSession();
```

Jedná se o základní rozhraní používané aplikacemi s Hibernate. Instance Session jsou odlehčené a jejich vytvoření, nebo zrušení je nenákladné. Toto je důležité, protože reálná aplikace bude potřebovat vytvářet a rušit sezení po celý čas svého běhu, snad pro každý jednotlivý požadavek. Sezení v Hibernate nejsou bezpečná při použití více vlákny a měly by být navrženy tak, aby k nim mohlo v každý okamžik přistupovat pouze jedno vlákno.

V Hibernate je sezení něco mezi *Connection* a *Transaction*. Na sezení může být nahlíženo jako na cache, nebo kolekci načtených objektů vztahující se ke každé *unit of work*. Někdy *Session* nazýváme *persistence manager*, protože poskytuje rozhraní pro operace vztahující se k perzistenci – jako je ukládání a načítání objektů. Sezení v Hibernate nemají nic společného s *HttpSession* používanými při práci se sítí.

- **Rozhraní SessionFactory.**

```
SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
```

Aplikace získává Session od *SessionFactory*. V porovnání se *Session* je tento objekt poměrně nezajímavý.

SessionFactory určitě není žádná odlehčená třída. Je určena k tomu, aby byla sdílena mezi mnoha vlákny aplikace. Typicky existuje jedna *SessionFactory* pro celou aplikaci – vytvořena například v průběhu inicializace. Pokud ale aplikace používá více databází s využitím Hibernate, je potřeba mít *SessionFactory* pro každou databázi.

SessionFactory uchovává v cache paměti vygenerované SQL dotazy a další mapovací metadata, které Hibernate používá za běhu. Také udržuje data, která byla získána jednou *unit of work* a může být využita v budoucí *unit of work* (pouze pokud třída a kolekce pro mapování specifikují, že je tato *second-level cache* požadována).

- **Rozhraní Configuration.**

```
SessionFactory sessionFactory = new Configuration()
.configure("hibernate.cfg.xml")
.buildSessionFactory();
```

Objekt *Configuration* je používán k nastavení a zavedení Hibernate. Aplikace používá instanci *Configuration* pro nalezení mapovacích dokumentů a vlastností specifických pro Hibernate a poté vytváři *SessionFactory*.

I když rozhraní *Configuration* hraje relativně malou roli v celé Hibernate aplikaci, je to první objekt, se kterým se vývojář musí seznámit při vyvíjení aplikací s použitím aplikačního rámce Hibernate.

- **Rozhraní Transaction.**

```
session.beginTransaction();
session.getTransaction().commit();
```

Rozhraní *Transaction* je volitelné API. Vývojář se může rozhodnout toto rozhraní nepoužívat a namísto toho mapovat transakce za použití vlastního kódu. *Transaction* abstrahuje aplikační kód od implementací, což mohou být JDBC transakce, JTA *UserTransaction*, nebo i Common Object Request Broker Architecture (CORBA) transakce – dovolující aplikaci řídit transakce pomocí konzistentního API. To pomáhá udržovat Hibernate aplikace přenositelné mezi různými druhy prostředí.

- **Rozhraní Query a Criteria.**

```
Query query = session.createQuery(„query… id = :id‟);
query.setInteger(„id‟, id);
List res = query.list();
```

Rozhraní *Query* dovoluje vykonávat dotazy proti databázi a řídit, jak jsou tyto dotazy vykonávány. Dotazy jsou psány buď v HQL nebo nativním SQL dialektu databáze. Instance *Query* se používá pro přiřazování parametrů k dotazu, omezování počtu výsledků a konečně také k vykonání dotazu.

```
Criteria criteria =   session.createCriteria(
                      Text.class,      "t");
criteria.createAlias("t.oldId", "x");
criteria.setProjection(Projections.distinct(
      Projections.projectionList()
      .add(Projections.property("t.oldId"))
```

```
    ));
    List results = criteria.list();
```

Rozhraní *Criteria* je velmi podobné, dovoluje vytvořit a provést objektově orientované dotazy. [3]

3.3 Apache Cayenne

Pro popis a pochopení Cayenne bude použita oficiální dokumentace a materiály dostupné na webových stránkách této technologie, protože knihu popisující tento aplikační rámec neexistují. Nejprve bude popsán nástroj Cayenne Modeler, se kterým se bude muset vývojář používající Cayenne seznámit jako první a poté bude následovat seznámení se samotným Cayenne – budou vysvětleny základní třídy nutné pro práci s tímto aplikačním rámcem.

3.3.1 Cayenne Modeler

Cayenne Modeler já nástroj zjednodušující život vývojáře. Lze v něm vytvořit novou databázi pro použití v aplikaci, zvládá také ale reverzní inženýrství – tedy práci s již existující databází. Po vytvoření nového projektu bude programátor pracovat s objekty z následujícího seznamu.

- **DataNode.** *DataNode* slouží k popisu jednotlivých databází, ke kterým se bude databáze připojovat – mapovací projekt Cayenne (ten, který vytvoříme pomocí Cayenne Modeleru) dokáže totiž používat více různých databází. *DataNode* od programátora potřebuje specifikovat parametry pro připojení k databázi. Pro Derby databázi to vypadá například takto:
  JDBC Driver: *org.apache.derby.jdbc.EmbeddedDriver*
  URL: *jdbc:derby:memory:testdb;create=true*
  DataNode také umožňuje vybrat strategii pro aktualizaci schématu. Nejpoužívanější je *org.apache.cayenne.access.dbsync.CreateIfNoSchemaStrategy*, tedy ta, kdy Cayenne při startu aplikace vytvoří nové schéma v databázi založené na ORM mapování, pokud takové již neexistuje.
- **DataMap.** *DataMap* je objekt, který udržuje mapovací informace. *DataMap* se vytvoří automaticky v GUI po kliku na *Create DataMap* a je svázáno s dříve vytvořeným *DataNode*. Důležitá informace, kterou od vývojáře *DataMap* požaduje je, aby specifikoval výchozí Java balík, například

```
    org.exampe.cayenne.persistent
```

kam se uloží vygenerované třídy.

Objekt *DataMap* pak umožňuje přidávat *DbEntity* – což může v databázi znamenat pohled (*view*), nebo tabulku. Lze je tedy přidávat takto ručně, na základě čehož pak bude databáze vygenerována, ve velké většině případů ale budou vývojáři využívat reverzní inženýrství, kdy se databáze do projektu importuje automaticky. To je dostupné v menu *Tools -> Reengineer Database Schema*. Nakonec je potřeba vygenerovat výsledné třídy pomocí menu *Tools -> Generate Classes*, kde je potřeba nastavit složku pro uložení hotových tříd. [6]

3.3.2 Základy práce s Cayenne

Podkapitola obsahuje popis tříd, které by programátor měl znát a které bude využívat v aplikačním rámci Cayenne.

- **ServerRuntime.**

```
ServerRuntime cayenneRuntime = new ServerRuntime(
"cayenne-project.xml");
```

ServerRuntime slouží pro inicializaci celého aplikačního rámce. Načítá nastavení potřebné pro běh aplikace z dodaného XML souboru. *ServerRuntime* se v každém programu volá pouze jednou na začátku.

- **ObjectContext.**

```
ObjectContext context = cayenneRuntime.getContext();
```

ObjectContext je izolované „sezení“ v Cayenne, které poskytuje potřebné API pro práci s daty. Má metody pro vykonávání dotazů a řízení perzistentních objektů. Při vytvoření prvního *ObjectContextu* se teprve načítá nastavení z konfiguračního souboru specifikovaného v *ServerRuntime* a toto nastavení je pak sdíleno s dalšími vytvořenými *ObjectContexty*.

Užitečná vlastnost Cayenne je způsob generování mapovacích tříd. V balíku, který vývojář definuje, se vytvoří třídy, se kterými může pracovat a které zůstanou zachovány i po následném generování databázového schématu. Tento balík totiž obsahuje další balík, *auto*, který obsahuje třídy začínající podtržítkem. Úpravy provedené v těchto třídách tedy nebudou zachovány po opětovném generování např. kvůli změně schématu, a proto by vývojáři do těchto tříd neměli zasahovat.

Nakonec budou představeny běžné operace s databází v prostředí aplikačního rámce Cayenne.

- **Vytvoření objektu**. Nové objekty se vytvářejí pomocí metody *.newObject()* poskytované instancí třídy *ObjectContext*.

```
Table t = context.newObject(Table.class);
t.setName(„Name");
```

  Takto je objekt vytvořen v paměti. Aby se změny projevily v databázi, je nutné je odeslat pomocí konstrukce

```
context.commitChanges();
```

  Metoda *.commitChanges()* slouží pro uložení jakýchkoliv změn do databáze, tedy i pro operace *insert*, *update* nebo *delete*. Ukončení programu bez volání této metody může způsobit, že se změny v databázi neprojeví.

- **Získání objektu.** *SelectQuery* slouží pro přístup k datům z databáze. Může být vytvořen pomocí API Cayenne nebo už v Cayenne Modeleru. Získání všech záznamů z Table vypadá takto:

```
SelectQuery sq = new SelectQuery(Table.class);
List result = context.performQuery(sq);
```

  Pro omezení získaných záznamů se používá třída *Expression*. Pokud například chceme získat záznamy z *Table*, kde hodnota ve sloupci *PROPERTY* začíná písmeny *„ahoj"*, použijeme následující konstrukci:

```
Expression e = ExpressionFactory.likeIgnoreCaseExp(
     Table.PROPERTY, „ahoj%");
SelectQuery sq = new SelectQuery(Table.class, e);
List result = context.performQuery(sq);
```

  Pro optimalizaci dotazů slouží další třída, *SQLTemplate*. Použije se obdobně jako *SelectQuery*:

```
SQLTemplate templ = new SQLTempalate(„sql dotaz");
Templ.setFetchLimit(100);
List result = context.performQuery(templ);
```

- **Smazání objektu**. Pro smazání objektu se používá metoda dostupná v instanci *Context* s názvem *.deleteObject()*. Změny v databázi je opět potřeba potvrdit.

```
    Context.deleteObject(table);
    Context.commitCahnges();
```

- **Aktualizace objektu.** Nejprve je potřeba objekt načíst použitím některého z uvedených způsobů, poté bude dostupný jako objekt v Javě, který lze měnit pomocí vygenerovaných metod. Po příslušných úpravách se provede uložení voláním pouze metody *.commitChanges()* příslušné instance objektu *Context*.

Uvedené informace by měly vývojáři stačit pro začátky s aplikačním rámcem Cayenne. Spoustu dalších informací lze najít v různých diskuzních fórech, příklady pro složitější práci s Cayenne poskytuje i oficiální dokumentace. Byly zde vysvětleny základní pojmy s použitím informací dostupných z oficiální dokumentace a referenčního tutoriálu, nicméně nedostatek materiálů popisujících architekturu naznačuje, že společnost Apache preferuje učení se z vystavených ukázkových kódů, spíše než z obsáhlých textů, kde by bylo popsáno fungovaní aplikačního rámce Cayenne. [6]

4 NÁVRHOVÉ VZORY PRO NAČÍTÁNÍ DAT

Za návrhový vzor je pokládáno obecné řešení softwarových problémů, se kterými se vývojáři potýkají při návrhu programů. Návrhový vzor není součástí zdrojového kódu, je to součást návrhu – způsob, jakým je o problému a jeho řešení uvažováno. Návrhové vzory jsou obecné šablony pro řešení častých problémů v programování, nejsou to ale algoritmy, ty slouží pro řešení konkrétního problému. Právě při sestavování algoritmu by měl vývojář používat návrhové vzory jako standardní postup, což způsobí, že struktura kódu bude srozumitelná i ostatním vývojářům a také pomáhá přeskočit proces „objevování kola", protože rozsah návrhových vzorů je dnes tak široký, že pokrývá všechny běžné situace, které vývojář v praxi potřebuje řešit.

Kvůli omezenému rozsahu práce nebudou ukázány všechny návrhové vzory (jako např. *Unit of Work*, *Identity Map*, *Repository…*), ale budou představeny pouze ty nejpoužívanější, jako jsou *Table Data Gateway*, *Row Data Gateway*, *Active Record*, *Data Mapper*, *Data Transfer Object, Lazy Load* a ujasníme si výraz *Data Access Object*.

4.1 Table Data Gateway

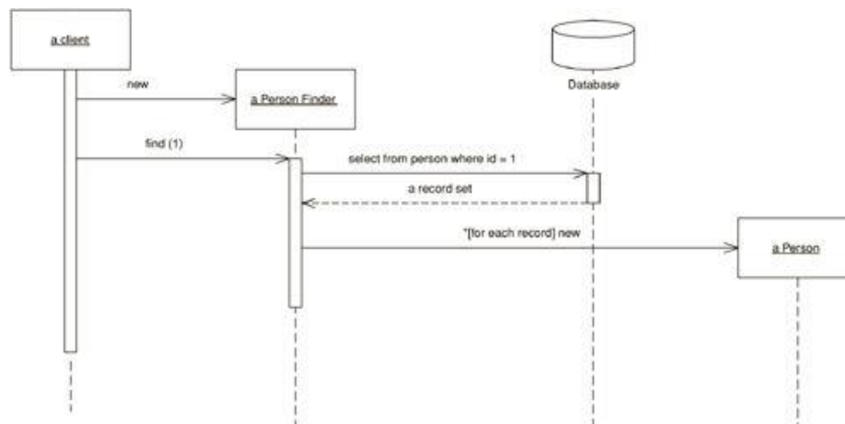
Jde o přístup k databázové tabulce. Komunikace s databází je obalena do jedné třídy – tato data získává a vrací je klientskému kódu. Obsahuje metody pro operace *insert*, *update*, *delete* a obvykle také funkce sloužící pro vyhledávání v kolekci dat, jako jsou *find*, příp. obdoby *findById*, *findByName*, …

Nejzáludnější na tomto návrhovém vzoru je, jak vrací informace na odeslaný dotaz. I jednoduchá funkce *find* totiž ve většině případů vrací více než jednu položku. Java tento problém řešit nemusí, protože umí pracovat s objektem *ResultSet*, který obsahuje veškeré informace obdržené z databáze. Přesto může být výhodné vracet výstup v podobě *Data Transfer Object*, který bude jednoduše použitelný klientskou částí kódu. Více o DTO v následující kapitole *Data Transfer Object*.

Table Data Gateway je asi nejjednodušší databázové rozhraní, které můžeme použít, protože je mapováno na tabulku jako celek, což je velmi přirozené chápání databázové struktury. Jedná se o logické zapouzdření přístupu ke zdroji dat. [4]

4.2 Row Data Gateway

V *Row Data Gateway* vždy existuje jedna instance pro jeden záznam. Přesně mapuje vlastnosti tabulky, tedy její jednotlivé sloupce. Tento návrhový vzor zvládá operace jako *insert*, *update* a *delete*. Pro získání tohoto objektu se používá externí třída *Finder* – příklad ukazuje *Obrázek 4*: Provedení operace find v návrhovém vzoru Row Data Gateway [4].



Obrázek 4: Provedení operace find v návrhovém vzoru Row Data Gateway [4]

Často je obtížné určit rozdíl mezi *Row Data Gateway* a *Active Record*. Záleží zejména na tom, zda je ve výsledku nějaké doménová logika – pokud ano, jedná se o *Active Record*. *Row Data Gateway* by měla obsahovat pouze logiku pro přístup do databáze a žádnou doménovou logiku. [4]

4.3 Active Record

Jak je zřejmé z předchozí kapitoly, tento návrhový vzor je velmi podobný vzoru *Row Data Gateway*, navíc ale přidává ještě doménovou logiku. *Active Record* obsahuje informace o datech i o jejich chování. Většina dat potřebuje být perzistentně uchována v databázi, *Active Record* tedy plní úplného prostředníka – doménovému modelu předává logiku přístupu k datům, takže data potom lze z databáze jednoduše číst i do ní zapisovat.

Struktura *Active Record* objektu by měla přesně odpovídat struktuře v databázi – jedna vlastnost třídy pro každý sloupec v databázi. Návrhový vzor obvykle obsahuje metody obsahující tuto logiku:

- vytvoření instance *Active Record* z SQL *ResultSetu*,
- vytvoření instance pro pozdější vložení do tabulky,
- statické metody hledání pro běžné SQL dotazy vracející objekty *Active Record*,

- aktualizace databáze a vložení dat z objektu *Active Record*,
- *gettery* a *settery* pro jednotlivé pole,
- implementace některých částí obchodní logiky aplikace.

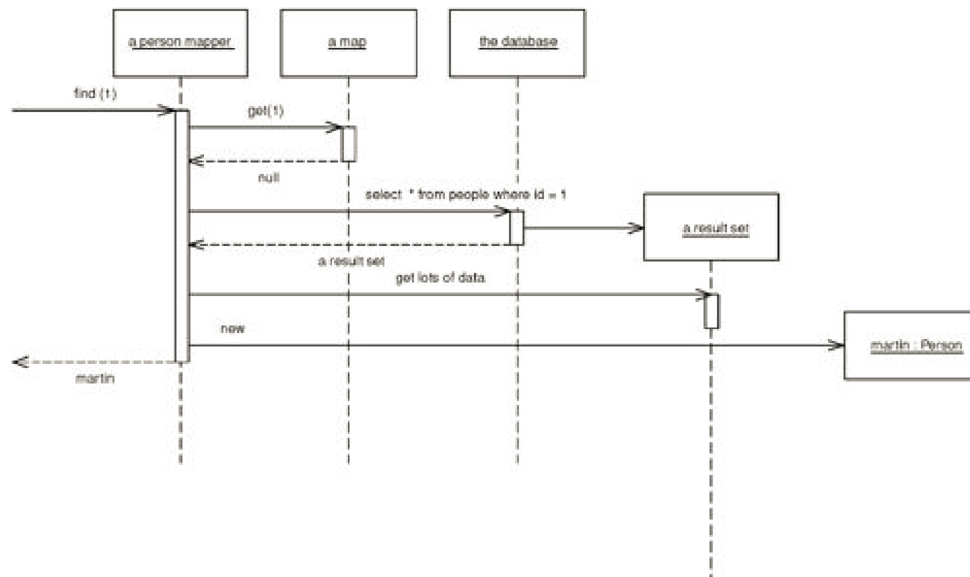
Active Record je vhodné použít pro jednoduchá databázová schémata. Je jednoduchý na pochopení i implementaci, zejména pokud objekty *Active Record* přímo korespondují s databázovými tabulkami. Pokud ne, je použití *Active Record* složitější a s pokračující náročností aplikace se bude programátor dostávat do čím dál větších problémů, pokud bude požadovat dědičnost, kolekce nebo vztahy mezi tabulkami. V tom případě bude rozumné použít *Data Mapper*. [4]

4.4 Data Mapper

Data Mapper je vrstva, která zajišťuje přenos dat mezi objekty a databází a zároveň izoluje objekty od databáze a naopak. Objekty a relační databáze používají odlišné mechanizmy. Mnoho součástí objektu, jako jsou kolekce nebo dědičnost, neexistují v relačních databázích. Je výhodné použít vzor *Data Mapper*, když vytvoříme model se spoustou business logiky, pro jednodušší organizaci dat a pro popis jejich chování.

Data Mapper je vrstva oddělující objekty v paměti od databáze. Je zodpovědná za přenos dat mezi těmto dvěma stavy a také má za úkol od sebe tyto způsoby uchování dat oddělit. Se vzorem *Data Mapper* nemusí objekty v paměti vůbec vědět, že existuje nějaká databáze, nepotřebují žádné SQL kódy v implementaci rozhraní a už vůbec nemusejí znát databázové schéma.

Obrázek 5: Získání dat z databáze [4] ukazuje schéma pro načtení dat z databáze a měl by přiblížit princip fungování tohoto návrhového vzoru. [4]

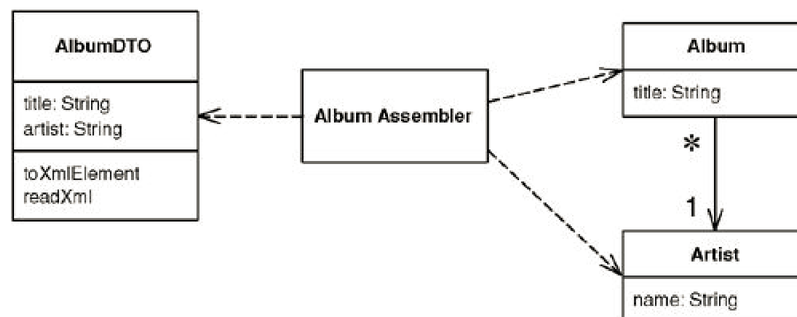


Obrázek 5: Získání dat z databáze [4]

4.5 Data Transfer Object

DTO slouží pro přenos dat mezi procesy tak, aby se redukoval počet volání různých metod.

Princip opět jednoduše znázorňuje *Obrázek 6*: Ukázka redukce volaných metod [4].



Obrázek 6: Ukázka redukce volaných metod [4]

Pokud pracujeme se vzdáleným rozhraním (jako je např. vzor *Remote Facade*), každé volání je nákladné. Potřebujeme snížit počet takovýchto volání, což znamená, že s každým voláním musíme přenášet více informací. Jeden ze způsobů je použít spoustu parametrů. To je ale velmi nepříjemné pro programy – často nemožné s jazyky jako je Java, které vrací pouze jednu hodnotu.

Řešením je vytvořit DTO, které bude obsahovat všechna potřebná data pro jednotlivá volání. Tyto DTO potřebují být serializovatelná, aby je bylo možné efektivně použít. Překlad dat mezi DTO a objekty většinou probíhá na straně serveru.

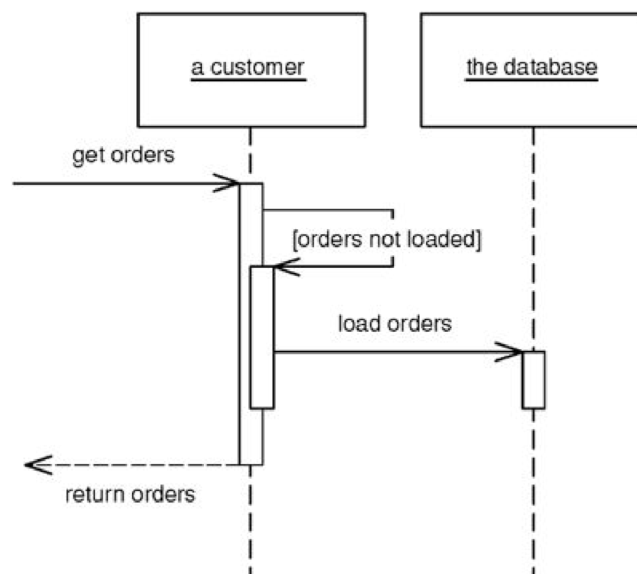
Mnoho lidí z komunity Sun nazývá tento vzor *Value Object*, což ale může být matoucí a někdy stejný termín může označovat jiný návrhový vzor. [4]

4.6 Data Access Object

Data Access Object je jiné označení pro vzor *Table Data Gateway* popsané v předchozí kapitole. Je ale předmětem diskuzí, zda se návrhový vzor označovaný *Data Access Object* vždy zakládal na tabulkách; diskuze naznačují, že to byl buď *Table Data Gateway*, nebo to mohl být i vzor *Row Data Gateway*. Použití jiného názvu má ten důvod, že *Data Access Object* a zkratka DAO mají svůj vlastní význam ve světě Microsoft. [4]

4.7 Lazy Load

Objekt neobsahuje všechny data, ale ví o způsobu, jak je získat.



Obrázek 7: Práce s daty v návrhovém vzoru Lazy Load

Pro načítání dat z databáze do paměti je praktické navrhnout to tak, aby se při načtení požadovaných dat načetla také související data. Tento způsob zjednodušuje práci vývojáři, který by jinak musel načítat všechny data explicitně.

Jednou se ale programátor dostane do bodu, kdy načtení jednoho objektu spustí načítání obrovského množství dalších objektů. Pokud je potřeba pouze několik objektů, je to pak zbytečná zátěž navíc.

Návrhový vzor *Lazy Load* zastavuje tento načítací proces a nechává vlastní značky ve struktuře, takže data mohou být načtena, až když jsou skutečně potřeba.

Existuje několik variant návrhového vzoru *Lazy Load*: *lazy initialization*, *virtual proxy*, *value holder* a *ghost*.

Lazy initialization je z těchto variant nejjednodušší. Před každým přístupem do databáze zkoumá, zda je objekt *null*. Pokud ano, tak se provede načtení z databáze, jinak se vrátí již načtený objekt. Problém nastává, když je v databázi skutečně hodnota *null*. V takovém případě je potřeba dopsat další logiku, která ověří, zda byl dotaz již načten nebo ne.

Dobrá věc na **Virtual proxy** je, že vypadá přesně jako objekt, ke kterému má přistupovat. Špatná věc je, že to není ten samý objekt, takže se velmi lehce může stát, že se vývojář ztratí v problémech ohledně identity objektu. Navíc pro jeden skutečný objekt může existovat více virtual proxy, což může vést k těžce objevitelným chybám.

S doménovými třídami lze tento problém obejít použitím **value holderu**. Koncept je takový, že objekt obaluje jiný objekt. Abychom se dostali k obalenému objektu, zeptáme se *value holderu* na jeho hodnotu, ten ji ale načte z databáze pouze při prvním dotazu. Nevýhoda *value holderu* je, že potřebuje vědět, zda je stále aktuální.

Ghost je částečně reálný objekt. Při načtení objektu z databáze obsahuje tento objekt pouze ID. Kdykoliv se pak snažíme přistoupit k některému sloupci, načte se celý záznam. Samozřejmě není nutné načítat všechny data v jednom kroku, ale je možné rozdělit je do logických celků běžně používaných spolu.

Použití vzoru *Lazy Load* je vhodné snad pouze v případě, že získávaná hodnota potřebuje další speciální volání databáze. Většinou je dobrý nápad získat všechna potřebná data najednou, protože z hlediska výkonu bývá náročnější provádět více jednoduchých dotazů, než jeden složitější. *Lazy Load* také způsobí, že aplikace bude o něco více složitá, takže před použitím je vhodné si rozmyslet, zda je to skutečně potřeba. [4]

5 GENEROVÁNÍ CRUD OPERACÍ

Kapitola pojednává o možnostech generování CRUD operací a snaží se porovnat současné nástroje k tomu určené. Předmětem zájmu bude zejména o generování složitějších metod pro dotazy, jakými jsou například *JOIN* v SQL, nebo pohledy a podobnými. Mnoho aplikačních rámců totiž nabízí pouze základní, omezené generování operací, obsahující zejména *gettery* a *settery* pro sloupce v každé jednotlivé tabulce, zvládají práci s automatickou inkrementací primárních klíčů, ale zbytek je potřeba dodefinovat ručně, případně to není možné vůbec.

Téměř všechny operační rámce nabízejí možnost generování kódu, aby ulehčili rutinní práci vývojářům. Úroveň generování je ale často pouze základní a nepokryje reálné použití databáze, programátoři jsou po použití těchto nástrojů nuceni dopsat operace, které v reálné aplikaci potřebují, ručně. Bude ujasněno, které nástroje jsou tedy nejlepší a kolik pohodlí při vývoji skutečně nabízejí.

5.1 Možnosti generování CRUD vybranými aplikačními rámci

Nejprve budou možnosti generování CRUD operací popsány v aplikačním rámci Hibernate. Existují nástroje **Hibernate Tools**, které dokáží generovat základní operace, ale i propojení tabulek na základě cizích klíčů. Nad Hibernate lze ještě umístit vrstvu *Data Access Object*, nebo *Data Transfer Object*, kde si vývojáři mohou doplnit vlastní metody a vznikne tím tedy další vrstva oddělující perzistentní vrstvu a aplikační logiku. V ideálním případě se pak v klientském kódu vůbec nebudou vyskytovat SQL nebo HQL dotazy a ani prvky Criteria API. To vše by mělo být od aplikační logiky v reálném projektu odděleno. Závěrem tedy lze říci, že komfort poskytovaný Hibernate Tools je průměrný v přínosu pro vývojáře. [3]

Další technologie pro generování CRUD operací je EclipseLink. Pro generování kódu je doporučován nástroj Oracle **JDeveloper**. Obsahuje průvodce pro reverzní inženýrství JPA entity z databázových tabulek a pro generování EJB 3.0 session beanů s injekcí do *EntityManageru*. Také obsahuje metody pro dotazování se JPA entit a pro testování generovaných entit. JDeveloper funguje tedy podobně jako Hibernate Tools, vytvoří základní operace pro všechny tabulky, včetně provázanosti cizími klíči, ale zbytek si pomocí vygenerovaných dotazovacích metod musí vývojář dopsat sám, tedy opět průměrný komfort, jako v případě první uvedené technologie. [10]

V testech popsaných v praktické části je také popsaný aplikační rámec Cayenne od společnosti Apache. Jeho největší předností a důvodem oblíbenosti mezi mnoha vývojáři je nástroj **Cayenne Modeler**. Kromě generování CRUD operací nabízí přívětivé funkce jako je reverzní inženýrství databázových tabulek, jako oba již uvedené aplikační rámce, navíc však nabízí možnost vytvořit si vlastní schéma a podle toho pak generovat tabulky do reálné databáze a také entity pro jazyk Java. Kromě těchto komunitou velmi kladně hodnocených funkcí tedy nabízí generování pouze základních metod pro přístup k jednotlivým tabulkám a také lze v Cayenne Modeleru definovat vztahy pomocí cizích klíčů a následně generovat správné entity. Třetí ze známějších aplikačních rámců tedy bude také z hlediska požadované funkcionality zařazen do průměru. [6]

Dalším nástrojem vhodným pro generování kódu je **RevGen** od Outsource Cafe, Inc. Tento nástroj je docela vhodný pro generování CRUD operací, nicméně potřebuje další konfiguraci, aby poskytoval výsledky srovnatelné s Hibernate Tools nebo EclipseLink. Generuje také unit testy pro ověření správnosti vygenerovaných operací a zvládne i Data Access Objects s čistou implementací pro Hibernate, Spring ORM nebo JPA. Protože ale pro podání srovnatelných výsledků jako ostatní nástroje potřebuje další speciální konfiguraci, musí být nástroj RevGen označen jako nevhodný pro generování kódu. [13]

Dali Java Persistence Tools je plugin pro prostředí Eclipse a obsahuje nástroje pro definici a úpravu JPA entit generovaných z relační databáze. Dali se zaměřuje na zjednodušení složitosti generování základních operací poskytnutím grafických průvodců pro mapování a bohatého uživatelského rozhraní pro konfiguraci generovaných jednotek. Podobnou funkcionalitu nabízí i integrované vývojové prostředí Netbeans. Oboje rozšíření ale nabízejí opět pouze základní funkcionalitu, což z nich činí průměrné nástroje, můžeme je však díky grafickým průvodcům doporučit pro nováčky v oblasti ORM. [11]

Nástroj **jpm2java** pracuje stejně jako předchozí uvedené, navíc ale umožňuje volitelně generovat soubor *persistence.xml*, případně hbm.xml pro Hibernate a speciální třídy, které pomáhají s perzistencí vygenerovaných entity tříd. Není ale rozšířený a poslední verze dle dostupných informací je starší než pět let. Přínos pro programátory tedy je podprůměrný. [12]

5.2 Shrnutí

Uvedené aplikační rámce tedy nabízejí v podstatě stejnou funkcionalitu a nelze říct, že by byla ohromující. V tomto směru existují ještě velké rezervy pro další zlepšování funkcionality. Na nejlepší cestě je dle mého názoru aplikační rámec Cayenne s grafickým nástrojem Cayenne Modeler. Umím si představit, že jej lze nepříliš složitě rozšířit alespoň o jednoduché funkce, např. obdobu *view*: mohlo by být možné vybrat si sloupce z různých tabulek (provázaných cizími klíči) a generovat funkce pro jejich zobrazení, ale například i aktualizaci (což právě databázová funkce *view* neumí). Případně by před generováním kódu mohlo jít vybrat, jaký návrhový vzor chceme používat a podle toho přizpůsobit generované třídy.

II. PRAKTICKÁ ČÁST

6 PŘÍPRAVA TESTOVACÍHO PROSTŘEDÍ

Před vypracováním finálních projektů byla vytvořena testovací databáze a jednoduché projekty pro seznámení s technologiemi JDBC, Hibernate a Cayenne. Bude následovat stručný popis databází a funkce jednotlivých programů, více pak budou rozvedeny až finální projekty.

Testy byly spouštěny na virtuálním stroji se systémem Debian. Aby bylo měření co nejpřesnější, na serveru byla nainstalována čistá instalace bez grafického prostředí a pouze nutné minimum programů potřebných pro běh testů.

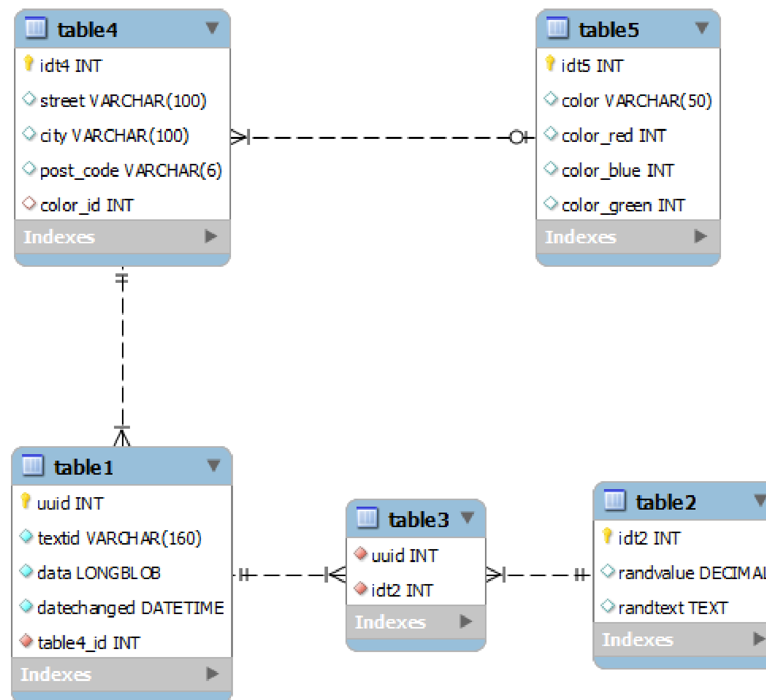
Testy jsou koncipovány tak, že přijímají argumenty z příkazové řádky. Testovací programy umožňují zvolit druh testované operace, počet měřených operací a samozřejmě soubor pro uložení výstupu. Tento přístup umožňuje spouštění pomocí *cronu*, což velmi zefektivnilo celé testování.

Použité databáze jsou v obou případech (testovací i finální Wikipedia) MySQL databáze.

6.1 Testovací databáze

Testovací databáze obsahuje 5 tabulek s náhodnými sloupci a vygenerovanými daty. Program pro generování záznamů (PrepareTestData.jar) opět umožňuje vstup z konzole a je tak možné zvolit tabulku, která má být naplněna daty, počet záznamů, které budou vloženy a hodnotu „modulo“ – určující, jak často se má provést informační výpis o postupu generování na konzoli.

Obrázek 8 ukazuje EER diagram databáze použité pro testování.



Obrázek 8: EER diagram testovací databáze

6.1.1 Program PrepareTestData

Použití může vypadat třeba takto:

```
java -jar PrepareTestData.jar -t1 1000 100
```

Tabulka 1: Argumenty pro PrepareTestData

	Tabulka	Počet záznamů	Hodnota modulo
Možné hodnoty	-t1	[int] – kolik celkem záznamů se má vložit.	[int] – po kolika záznamech se má provést informační výpis do konzole.
	-t2		
	-t3		
	-t4		
	-t5		

Program využívá technologii JDBC, protože s touto technologií byl autor nejvíce obeznámen před započetím práce na bakalářské práci.

Hlavní třída obsahuje hlavní komponentu – přepínač, který se dělí do pět větví, kde každá větev obstarává generování dat do jedné z pěti tabulek.

Uvnitř každé větve cyklu je pak příkaz pro cyklus *for*, který provede tolik vložení, kolik je zadáno jako argument při spuštění.

Na konci cyklu je prováděna kontrola, kolik již bylo vloženo záznamů a pokud je podmínka počítající modulo z aktuálního pořadí záznamu a hodnoty pro modulo udané argumentem rovna nule, provede se výpis na konzoli informující o postupu vkládání dat.

6.2 Reálná databáze

Jako reálnou databázi pro finální testování byla použita záloha databáze české Wikipedie, aby mohla být ukázána práce s reverzním inženýrstvím, což byl už v návrhu ORM předpokládaný nejběžnější příklad použití (namísto vytvoření nové databáze). Při importu se objevily problémy, protože záloha byla k dispozici pouze ve formátu XML, ne standardním SQL, nakonec se ale import podařil.

Použitou zálohu lze stáhnout ze stránek *dumps.wikimedia.org*. Schéma databáze bude uvedeno formou odkazu, protože je tak rozsáhlé, že by bylo v tomto formátu nečitelné. Schéma lze najít také na *mediawiki.org*.

Stažený soubor byl ve formátu *.xml.bz2*. Rozbalení z *.bz2* bylo provedeno standardně příkazem `bunzip2`. Přímý import tohoto XML souboru standardní SQL cestou ale nebyl možný, ten dokáže načíst pouze data z XML do jedné tabulky a nezvládne celou databázi, jak bylo potřeba.

Řešením bylo nainstalovat Apache Server a *stáhnout PHP skripty* pro provozování MediaWiki. Po rozbalení stačilo spustil dodaný skript (*importDump.php* ve složce *maintenance*) pro import a poté byla databáze konečně připravená.

Protože již byla na virtuálním serveru nainstalována a funkční MediaWiki, bylo už jen nutné zapnout logování databáze, abych byly zachyceny vykonávané dotazy. Cílem této snahy bylo použít relevantní SQL dotazy vyskytující se v praxi. Pouze bylo potřeba v souboru (výchozí umístění) */etc/mysql/my.cnf* přidat následující dva řádky:

```
general_log_file        = /var/log/mysql/mysql.log
general_log             = 1
```

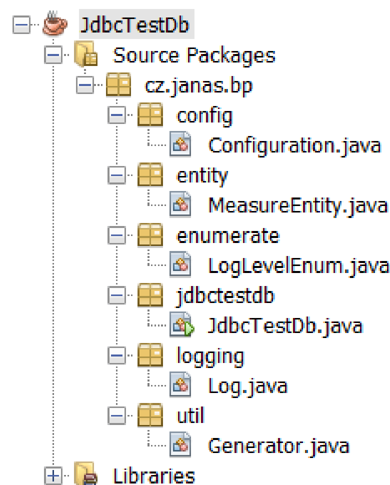
Po dokončení těchto kroků bylo hotové vše, co bylo potřeba pro vykonání programů, které budou měřit rychlost jednotlivých operací.

7 PŘÍKLADY VYUŽITÍ JDBC A ORM

Kapitoly budou probírat skladbu a fungování jednotlivých programů. Popis zkušebních programů bude zjednodušen – podrobnější popis bude uveden u finálních programů. Princip programů se nezměnil, pouze finální programy obsahují například lepší ošetření výjimek, rozumnější načítání vstupních argumentů a sekvenční zápis do výstupního souboru.

7.1 Popis zkušebních programů

Nejdříve, bez uvedení konkrétní implementované technologie bude popsána kostra programu, která je stejná pro následující tři implementace – JDBC, Hibernate i Cayenne. *Obrázek 9*: Struktura projektu JdbcTestDb ukazuje strukturu projektu JdbcTestDb, ta je podobná ve všech implementacích.



Obrázek 9: Struktura projektu JdbcTestDb

Na začátku funkce *main()* je vloženo velmi jednoduché načtení vstupních parametrů. Argumenty musí být načítány v přesně daném pořadí a musí mít správný formát. Protože to byly projekty sloužící pouze pro seznámení s novými technologiemi, neobsahuje program žádné ošetření záporných scénářů – předpokládá se, že vše bude probíhat správně.

Třeba JdbcTestDb lze spustit následovně:

```
java -jar JdbcTestDb.jar -select 1000 100
```

Možné hodnoty argumentů shrnuje *Tabulka 2*. Jsou použitelné pro projekty JdbcTestDb, HibernateTestDb i CayenneTestDb.

Tabulka 2: Hodnoty argumentů pro testovací programy

	Operace	Počet operací	Cesta k výstupu
Možné hodnoty	--select --insert --update --delete	[int] – počet operací, které se mají vykonat	[String] – absolutní cesta, kam se uloží vsledný CSV soubor

Jediná kontrola je ověření počtu argumentů

```
if (args.length > 2) { … }
```

Uvnitř této podmínky se porovnávají očekávané řetězce se vstupem a nastavuje se zvolená testovaná operace (select, insert, …)

```
if ("--select".equals(args[0])) {
      testType = 1;
}
```

Dále se vyparsuje požadovaný počet prováděných operací

```
numOfQueries = Long.parseLong(args[1]);
```

a cesta k souboru pro výstup

```
csvPath = args[2];
```

Tabulka 3 ukazuje formát výstupního souboru.

Tabulka 3: Formát výstupu testovacích programů

	Akce	Počáteční čas	Konečný čas
Možné hodnoty	select | insert | update | delete	[long]	[long]

Příklad pro testování operace update:

update;	1267700982582260;	1267700999041830;

Hlavní část programu obsluhuje přepínač

```
switch (testType) { … }
```

podle typu testu, který byl získán jako argument při spuštění.

Jednotlivé větve se liší podle konkrétní implementace, mají ale společné měření času, které vždy obaluje pouze danou elementární operaci. Příklad: Je požadována operace update, nejprve je ale potřeba získat náhodné ID záznamu, který bude upraven a uložen zpět do databáze. Bude tedy proveden potřebný dotaz *select* a měření času bude následovat až po tomto *selectu*. Součástí je také nutnost vygenerovat novou hodnotu, která bude uložena. Generování ale také není součástí měření. Pouze nezbytné minimum pro provedení operace *update*, tak aby bylo měření co nejpřesnější. Tento přístup je využíván ve všech programech pro měření rychlosti.

Každý jednotlivý čas se ukládá do entity *MeasureEntity*, která obsahuje dvě vlastnosti:

```
long startTime;
long endTime;
```

a příslušné operace *get* a *set*. V měření jsou tyto entity ukládány jako hodnoty do mapy

```
HashMap<Long, MeasureEntity> timeData = new HashMap<>();
```

a klíč je pořadí operace, ve kterém byla provedena.

Po provedení všech požadovaných operací se provede zápis do CSV souboru:

```
writer = new PrintWriter(csvPath, "UTF-8");
      timeData.entrySet().stream().forEach((me) -> {
            writer.append(
            action + ";" + me.getValue().getStartTime() + ";"
            + me.getValue().getEndTime() + "\n"
      );
});
```

A tím program končí.

7.1.1 Třídy Configuration, Generator a Log

Třídy, které jsem zatím nebyly uvedeny, jsou pouze pomocné: *Configuration*, *Log* a *Generator*.

Configuration je třída uchovávající nastavení pro každou implementaci. Společná vlastnost je

```
public static final int LOG_LEVEL = LogLevelEnum.WARN;
```

která určuje, které záznamy se budou vypisovat do konzole.

Tabulka 4 ukazuje možné úrovně pro výpis logovacích záznamů.

Tabulka 4: Úrovně logování v LogLevelEnum

Název	Popis
INFO	Informace například o načtení ovladače, úspěšném připojení k DB, apod.
WARN	Obsahuje varování, které ale neohrožují běh programu.
ERROR	Vážné chyby znemožňující vykonání programu, převážně výjimky.
MSG_FOR_USER	Zprávy pro uživatele, např. upozornění, že výstupní soubor je ve formátu CSV, pokud uživatel zadá jinou příponu.

Třída **Log** pracuje také s touto tabulkou a přepínač podle zvolené hodnoty úrovně výpisu logovacích záznamů informaci buď vypíše, nebo nevypíše. Třída *Log* pracuje pouze se standardním výstupem – pro rozsáhlejší testování je však jednoduše rozšiřitelná například pro výstup do databáze.

Třída obsahuje statické metody pro zápis požadovaných informací v následující podobě:

```
public static void logInfo(String msg) {
      log(msg, LogLevelEnum.INFO);
}
```

Metoda *log()* pak provede výpis na základě zvolené úrovně logování:

```
switch(level) {
case LogLevelEnum.INFO:
      if(logLevel <= LogLevelEnum.INFO) {
            System.out.println("INFO     ["
            + df.format(date) + "]: " + msg);
      }
      break;
…}
```

Kde *logLevel* se získá z *Configuration*

```
int logLevel = Configuration.LOG_LEVEL;
```

a také se vypisuje čas každého zápisu logu v tomto formátu

```
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd
                      hh:mm:ss");
```

Poslední třída, **Generator**, slouží pro operace *insert* a *update*. Poskytuje metody pro generování hodnot pro různé datové typy. Příkladem takové metody je funkce pro generování datového typu *double*:

```
public static double generateDouble() {
      return random.nextDouble();
}
```

Asi nejpoužívanější je metoda pro generování typu *String*, která pracuje takto:

```
StringBuilder sb = new StringBuilder(max);
for (int i = 0; i < random.nextInt(max - min) + min; i++) {
      sb.append(CHARACTERS.charAt(
            random.nextInt(CHARACTERS.length()))
      );
}
```

Kde proměnná *CHARACTERS* obsahuje znaky A – Z a a-z.

7.1.2 Popis JDBC implementace

Dříve uvedená třída *Configuration* navíc obsahuje hodnoty pro připojení k databázi:

```
public static final String DB_HOST = "localhost";
public static final String DB_PORT = "3306";
public static final String DB_USER = "testdbuser";
public static final String DB_PASS = "****";
public static final String DB = "testdb";
```

Před funkcí *main()* jsou uvedeny statické texty SQL dotazů, pro jsou pro svou obsáhlost uvedeny v příloze *Příloha P I*. Ekvivalentní dotazy jsou poté provedeny také v programech CayenneWiki.jar a HibernateWiki.jar.

Při použití technologie JDBC je nejprve potřeba získat *Connection* k databázi, které požaduje URL databáze a přihlašovací údaje:

```
Connection connection = (Connection)
DriverManager.getConnection("jdbc:mysql://host:port/
database‟, user, password);
```

Po úspěšném připojení lze instanci *Connection* využít pro *PreparedStatement*:

```
PreparedStatement colorsStmt =
connection.prepareStatement(SELECT_COLORS_QUERY);
```

Pokud dotaz vrací záznamy, získáme je v *ResultSetu*:

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

Jinak je získán počet ovlivněných řádků (pro operace aktualizace, vložení a smazání):

```
Int affectedRows = stms.executeUpdate();
```

7.1.3 Popis Hibernate implementace

Třída *Configuration* obsahuje pouze nastavení úrovně logování:

```
public static final int LOG_LEVEL = LogLevelEnum.INFO;
```

Pro práci s aplikačním rámcem Hibernate je vždy potřeba získat instanci *SessionFactory*:

```
SessionFactory sessionFactory =
HibernateUtil.getSessionFactory();
```

HibernateUtil je třída automaticky vygenerovaná v prostředí Netbeans a poskytuje právě potřebnou *SessionFactory*. Ta obsahuje nastavení ze souboru *hibernate.cfg.xml*.

Následně se pro každou transakci využívající metodu *.createQuery()* tvoří vlastní *Session*, kterou poskytne metoda *SessionFactory* - *openSession()*:

```
Session session = sessionFactory.openSession();
```

Po získání session začneme transakci:

```
session.beginTransaction();
```

Nyní můžeme vytvořit HQL nebo SQL dotaz voláním metody *.createQuery()*:

```
Query query = session.createQuery("select …“);
```

A výsledky získáme v seznamu takto:

```
List tmpResults = query.list();
```

Nakonec je potřeba transakci potvrdit a uložit tak provedené změny v databázi.

```
session.getTransaction().commit();
```

V Hibernate ale není třeba používat transakce pro dotazování, zejména pro jednoduché dotazy, které nám Hibernate Tools vygenerují. Můžeme použít následující konstrukci:

```
Table4 t4 = (Table4) session.load(Table4.class, 2);
```

Pokud potom budou provedeny změny v takto načteném objektu, stačí je uložit jednoduše prostřednictvím metody *.save()*:

```
session.save(t4);
```

7.1.4 Popis Cayenne implementace

Třída *Configuration* v projektu Cayenne také obsahuje pouze informaci o úrovni logování:

```
public static final int LOG_LEVEL = LogLevelEnum.INFO;
```

Použití Cayenne aplikačního rámce vyžaduje před začátkem samotné práce získání instance *ServerRuntime*, která požaduje pouze konfigurační soubor:

```
ServerRuntime runtime =    new ServerRuntime("cayenne-
                           project.xml");
```

Dále bude získán *ObjectContext*, který nám poskytne veškerou potřebnou funkcionalitu:

```
ObjectContext context = runtime.newContext();
```

Nyní už lze vytvářet např. *SQLTemplate* pro dotazy v jazyku SQL. *SQLTemplate* jako první parametr požaduje třídu entity, aby mohl správně přiřadit vrácené sloupce a jako druhý parametr samotný řetězec pro SQL dotaz:

```
SQLTemplate query = new SQLTemplate(Table4.class,"select…");
```

Seznam výsledných záznamů bude získán po vykonání dotazu metodou *performQuery()* instance `Context`:

```
List results = (Integer) context.performQuery(query);
```

Objekty lze ale získávat i pohodlněji, zejména pokud není potřeba optimalizovat dotazy pro rychlost, případně využívat speciální konstrukce, je možné použít třídu *Expression* takto:

```
Expression e = ExpressionFactory.matchExp("idt4", id4);
SelectQuery q = new SelectQuery(Table4.class, e);
List table4List = context.performQuery(q);
```

Výsledek je pak opět seznam, v tomto případě s jedním prvkem – *idt4* je primární klíč v databázi.

Pokud má být vytvořen nový objekt, který má být uchován v databázi, případně provedeme změny v načteném objektu, musíme je potvrdit:

```
context.commitChanges();
```

Smazání objektu se provede jednoduše

```
context.deleteObjects(t2);
```

a opět je potřeba změny potvrdit.

7.2 Popis finálních programů

Finální programy se od testovacích liší několika vylepšeními. Postupně bude ukázáno načítání parametrů, sekvenční zápis výsledků a rozeznávání chybných dotazů. Poté budou probrána specifika jednotlivých technologií a hlavní část programu starající se o měření délky trvání zvolených operací.

Změnou prošlo načítání vstupních parametrů, které je nyní více inteligentní. Je také v samostatné třídě *InputArgsUtil*. Vstupní parametry mohou být různě prohozené a není potřeba používat všechny – použijí se výchozí z třídy *Configuration*, která kromě zvolení úrovně logování obsahuje navíc tyto vlastnosti:

```
public static final int NUMBER_OF_QUERIES = 1000;
public static final int SAVE_INTERVAL = 100;
public static final String CSV_PATH = "test-result.csv";
public static final int TEST_TYPE = TestTypeEnum.SELECT;
```

Tyto hodnoty se použijí místo hodnot, které uživatel nezadá. Načítání parametrů probíhá pomocí dvou vnořených přepínačů v cyklu:

```
for (String arg : args) {
	switch (arg) {
	case "-a":
	case "--action":
		actionInt = 1;
		break;
	default:
		switch (actionInt) {
		case 1:
			setTestType(parseAction(arg));
			break;
		}
	}
}
```

Pro jednoduchost je v ukázaném kódu kontrola pouze jednoho parametru, obdobně by se ale přidaly další.

Tabulka 5: Argumenty pro finální programy ukazuje názvy a hodnoty argumentů ovlivňující běh programu.

Tabulka 5: Argumenty pro finální programy

Argument:	**-a, --action**	**-n, --queries-count**	**-i, --save-interval**	**-o, --output-csv**	**-h, --help**
Hodnoty:	s, select sg, select-generated sj, select-join i, insert u, update d, delete	[int] – počet dotazů, které se mají provést	[int] – interval, po kterém budou výsledky sekvenčně zapisovány do výstupního souboru	[String] – určuje cestu k souboru, ve kterém bude uložen výstup	-

Argumenty lze libovolně kombinovat, měnit pořadí, nebo neuvést. Spuštění 10 dotazů *select* v projektu HibernateWiki lze provést následovně:

```
java -jar HibernateWiki.jar -a select --output-csv
/home/user/results -i 3 -queries-count 10
```

Oproti testovacím projektům je ve finálních projektech implementován sekvenční zápis. Podle hodnoty argumentu –save-interval lze určit velikost dávky, která se bude zapisovat do souboru s výsledky.

```
if ((i % settings.getSaveInterval() == 0 && i != 0)) {
Log.logInfo("Added " + i + " lines of " +
        settings.getNumOfQueries() + " to output file.");
```

```
for (Map.Entry<Long, MeasureEntity> me :
timeData.entrySet()){
      writer.append(me.getValue().getDescriptionText() + ";"
            + me.getValue().getStartTime() + ";" +
            me.getValue().getEndTime() + "\n");

      timeData.remove(me.getKey());

}

}
```

Také došlo k rozšíření entity *MeasureDescEnum*, která kromě dvou hodnot pro časy obsahuje navíc informaci o tom, zda se dotaz podařil, nebo ne. Pokud se v některém cyklu některá operace nezdaří, zachytí je konstrukce *try…catch* a do výsledného souboru se zapíše, že se jedná o chybnou operaci.

Příloha P II obsahuje seznam SQL dotazů, které byly použity ve finálních programech pro testování. Uvedený dotaz pro *select* vybírá náhodný článek z data setu Wikipedie a je zachycen z reálného provozu. Další dotazy jsou uměle vytvořené, aby reflektovaly potřebu testovat různé typy dotazování (např. dotazy obsahující *join*).

7.2.1 Popis finální JDBC implementace

Všechny aspekty použité v kódu tohoto programu byly již představeny a zbývá pouze ukázat hlavní výkonnou část ve větvích hlavního přepínače.

Operace select.

Nejprve je proveden neměřený dotaz, který získá sloupce *page_title* a *page_namespace* potřebné pro měřený dotaz. Provedení měřeného dotazu vykonávají tyto řádky:

```
stmt = dbUtil.getConnection().prepareStatement(SELECT_QUERY);

stmt.setString(1, pageMetaInfo.get(0).getTitle());

stmt.setInt(2, pageMetaInfo.remove(0).getNamespace());

rs = stmt.executeQuery();
```

Kde *SELECT_QUERY* je proměnná obsahující text SQL dotazu a *pageMetaInfo* je seznam výsledků předchozího pomocného dotazu.

Operace *generated-select* a *join-select* fungují stejně, pouze je jim dodán jiný SQL dotaz a jsou jim nastaveny správné parametry.

Operace insert, update a delete.

Jedná se o operace aktualizací, které nevracejí výsledek jako *ResultSet*, ale jako celé číslo označující počet ovlivněných řádků. Kód pro všechny tři operace je tedy stejný, kromě

použitých SQL dotazů a příslušných parametrů a od dotazu *select* se liší metodou, která vykonání dotazu provádí.

```
stmt = dbUtil.getConnection().prepareStatement(INSERT_QUERY);
stmt.setString(1, randText);
stmt.setString(2, randFlags);
int results = stmt.executeUpdate();
```

7.2.2 Popis finální Hibernate implementace

Podobně jako v předchozí kapitole budou popsány pouze operace, o kterých se práce zatím nezmiňuje. Také se jedné pouze o kód ve větvích přepínače, který vybírá testovanou operaci.

Operace select.

Operaci *select* lze provést následujícím způsobem:

```
query = session.createQuery("SELECT … AND pageTitle =
            :pageTitle").setMaxResults(1);
query.setString("pageTitle", pageMetaInfo.getTitle());
List results = query.list();
```

Metoda *createQuery()* zpracuje HQL dotaz a vrátí seznam výsledků do proměnné *results*. Tento způsob se využívá zejména pro optimalizaci dotazů. Může být ale použit i další, jednodušší způsob:

```
Page page = (Page) session.get(Page.class, 1);
```

Page je třída vygenerovaná nástroji Hibernate a je možné získat ji podle *ID* (druhý parametr metody *get()*) tímto voláním.

Operace insert a update.

Operace *update* se skládá z načtení objektu z databáze, což bylo ukázáno dříve, a z uložení zpět do databáze – to je stejná operace, jakou používá operace *insert*. Uložení zajistí tento kód:

```
Text text = new Text();
text.setOldFlags(randFlags.getBytes());
text.setOldText(randText.getBytes());
session.save(text);
```

Objekt *Text* buď bude vytvořen nový, nebo bude získán z databáze, pomocí vygenerovaných metod mu lze nastavit požadované vlastnosti a uložit tak, že bude předán jako parametr metodě *.save()*.

Operace delete.

Smazání objektu z databáze je možné provést následovně:

```
Text text = (Text) session.load(Text.class, 1);
session.beginTransaction();
session.delete(text);
session.getTransaction().commit();
```

7.2.3 Popis finální Cayenne implementace

Projekt využívající aplikační rámec byl také zcela popsán, až na prováděné operace, jejichž doplnění nyní následuje.

Operace select.

Vykonání dotazu v databázi obstará instance třídy SQLTemplate. Dotazy select bude nejčastěji vykonávat právě tato třída, protože umožňuje optimalizaci dotazů.

```
query = new SQLTemplate(
      Page.class,
      "SELECT … WHERE page_title #bindEqual($pageTitle)");
Map<String, Object> params = new HashMap<>();
params.put("pageTitle", pageTitle);
query.setParameters(params);
query.setFetchingDataRows(true);
List results = context.performQuery(query);
```

Parametry dotazu je důležité předávat stejně, jak je uvedeno, pomocí konstrukce *#bind…()*. Pokud by byly parametry vložileny přímo do dotazu, Cayenne nezajistí jejich převod na bezpečné znaky.

Jako v případě Hibernate, i Cayenne umožňuje jednodušší zápis, pokud budou vygenerované třídy dostačující. Například třídu Page lze jednoduše získat takto:

```
Expression byId = ExpressionFactory.matchExp("pageId",
pageId);
SelectQuery select = new SelectQuery(Page.class, byId);
List results = context.performQuery(select);
```

Operace insert a update.

Některým z výše uvedených způsobů lze získat objekt, který chceme upravit, a uložení probíhá opět podobně jako samotná operace insert:

```
Text loadedText = (Text)
context.performQuery(selectQuery).get(0);
loadedText.setOldFlags(randFlags.getBytes());
loadedText.setOldText(randText.getBytes());
context.commitChanges();
```

Bude vytvořen objekt, budou mu nastaveny požadované parametry a poté budou změny potvrzeny v databázi metodou *commitChanges()*.

Operace delete.

Smazání objektu z databáze provede následující část kódu:

```
Text textToDelete = (Text)
context.performQuery(selectQuery).get(0);
context.deleteObjects(textToDelete);
context.commitChanges();
```

Nejdříve bude načten objekt, který má být smazán, následně se vykoná metoda *deleteObject()* a opět je třeba změny v databázi potvrdit.

7.3 Srovnání technologií Hibernate, Cayenne a JDBC

Nyní už byly prezentovány potřebné informace z oblasti teorie i praktické ukázky, které byly potřeba pro srovnání těchto technologií.

JDBC je technologie osvědčená, ale již překonaná. Protože využívá SQL dotazy, není zaručena přenositelnost aplikací, které JDBC používají. Také nenabízí nástroje pro generování dotazů vhodných pro danou databázi a tak není oblast, pro kterou by bylo vhodné tuto technologii doporučit.

Hibernate i Cayenne nabízejí nástroje, které generují základní dotazy do databáze, ale hlavně mapují databázové schéma na objekty. To je velmi užitečné u reálných projektů, kde se zadání mění téměř až do dne odevzdání projektu. V případě JDBC je potřeba pokaždé ručně přepisovat dotazy, které změny struktury ovlivnily. Při použití ORM aplikačních rámců ale stačí několik málo rychlých operací, které celé objekty vygenerují znovu a vývojář se tak může soustředit na důležitější práci, než mechanické přepisování dotazů.

Hibernate má, tak jako Cayenne, velkou uživatelskou základnu, takže v případě potíží lze požádat komunitu o podporu. Podporovaná funkcionalita je také podobná, nabízejí jednoduché dotazy s možností napsat si dotazy vlastní. V tomto ohledu je lepší Hibernate, než Cayenne, z toho důvodu, že Hibernate používá jazyk HQL, který je překládán

do potřebného SQL dialektu. Cayenne ale nemá podobný jazyk a dotazy se píší přímo v jazyku SQL.

Poslední rozdíl je, že Hibernate podporuje specifikaci JPA 2.1 a Cayenne ji nepodporuje, pouze z ní přebírá nápady. Z toho plyne i předpokládané použití – Cayenne lze kvůli výbornému nástroji Cayenne Modeler doporučit pro malé projekty jednotlivců, případně malých firem a Hibernate do velkých korporací, pro které jsou specifikace důležité.

8 TESTOVÁNÍ RYCHLOSTI CRUD OPERACÍ

Kapitola bude obsahovat výsledky testovaných operací s ohledem na různá specifika implementací a na reálné operace prováděné nad databází. Pro operaci *select* existují tři scénáře: obyčejný *select*, další je *select* provedený pomocí vygenerovaných metod ORM aplikačních rámců a poslední je *select* obsahující *join*, který simuluje běžně používané operace.

Součástí výstupu mělo být zatížení databáze sledované v průběhu testování linuxovým programem *mytop*. Bohužel jsem nedokázal napsat takovou *cron* tabulku, aby se provedlo spuštění.

8.1 CRON – spuštění testů

Pro spuštění testů byla použita *cron* tabulka. První operace byla restartování databáze před každým testem, aby byla zaručena co největší objektivita při provádění testů. Následně byl spuštěn vytvořený testovací program se zvolenou operací a následovaly tři spuštění utility *mytop* s rozestupem 15 minut.

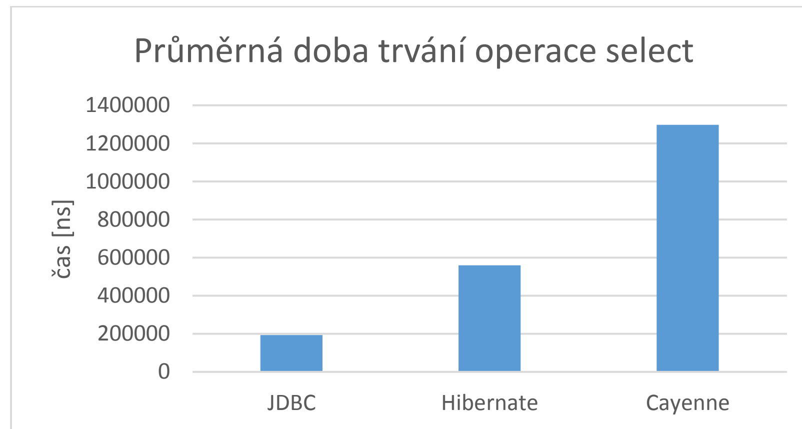
Jak bylo uvedeno v úvodu, utilita *mytop* se nespustila, byl špatně nastaven výstup této utility, nebo vznikl nějaký problém s oprávněním v prostředí Linux, kde byly testovací programy spouštěny.

Příloha P III obsahuje záznamy *cron*, které testování spouštěly.

8.2 Operace SELECT

Měření operace *select* obsahuje tři různé testy, protože se jedná o nejpoužívanější databázovou operaci. První test je obyčejný *select*, v aplikační rámcích Hibernate a Cayenne jsou použity metody pro přímé psaní dotazů buď v HQL nebo SQL. Druhý typ měření byl vykonán pomocí vygenerovaných tříd aplikačních rámců, kdy JDBC provedlo ekvivalentní operaci. Poslední test byl proveden s dotazem obsahujícím *join* – v tomto měření jsou očekávány největší výkonnostní rozdíly.

8.2.1 Jednoduchý SELECT



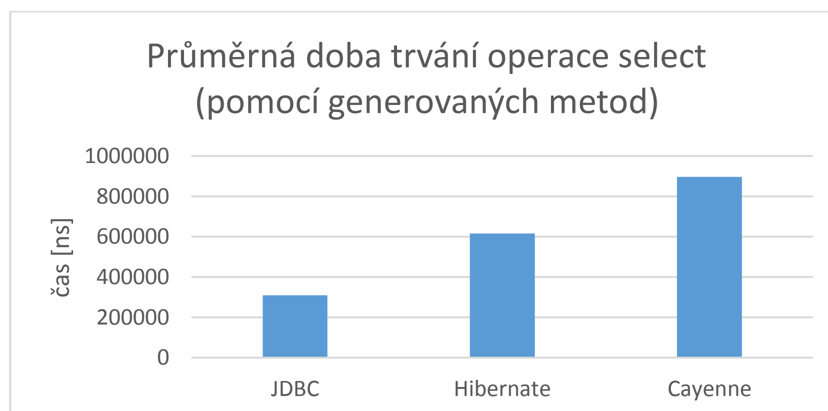
Graf 1: Operace select

Tabulka 6: Statistiky pro jednoduchý select

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	11817700	153730	172375	183410
Hibernate	10657820	14310	511470	588402,5
Cayenne	35242240	15580	1074260	1188980

Z testu vyplývá, že na jednoduché dotazy typu *select* je zdaleka výkonově nejhorší aplikační rámec Cayenne. Hibernate je následující a JDBC je nejvýkonnější.

8.2.2 SELECT pomocí generovaných metod



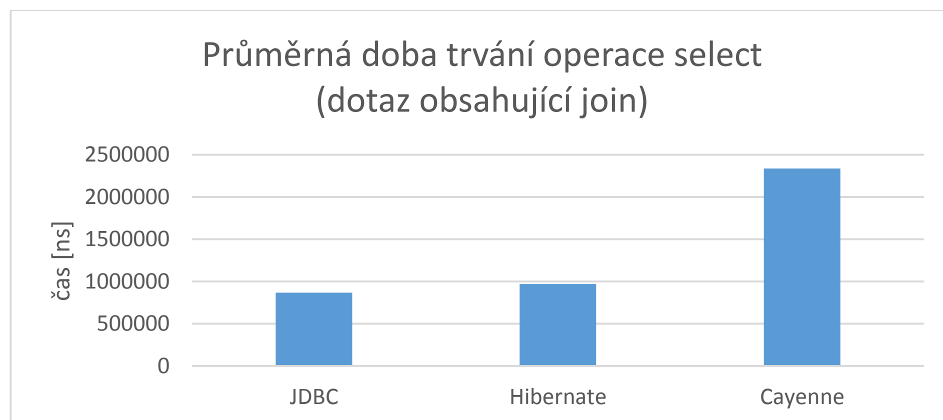
Graf 2: Operace select (pomocí generovaných metod)

Tabulka 7: Statistiky pro select pomocí generovaných metod

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	13941220	1140	277465	313645
Hibernate	21991860	11780	453910	512570
Cayenne	14100260	13710	828400	907095

Výsledek testu ukazuje, že získání dat pomocí generovaných testů je velký problém v aplikačním rámci Cayenne, dále Hibernate a nakonec JDBC je nejvýkonnější.

8.2.3 SELECT obsahující join



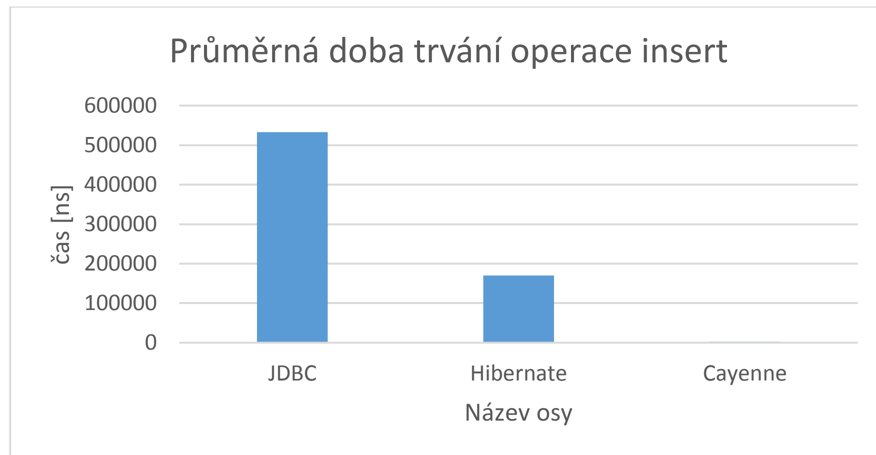
Graf 3: Operace select (obsahující join)

Tabulka 8: Statistiky pro select obsahující join

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	36374870	9160	825730	866285
Hibernate	35704120	5980	871435	929697
Cayenne	69951460	11540	1356485	1485093

Výsledek nejnáročnějšího testu ukazuje, že pro reálný případ použití je Cayenne nevhodný, Hibernate a JDBC jsou na tom ale velmi podobně – JDBC je ale podle statistik nejvýkonnější i v tomto testu.

8.3 Operace INSERT



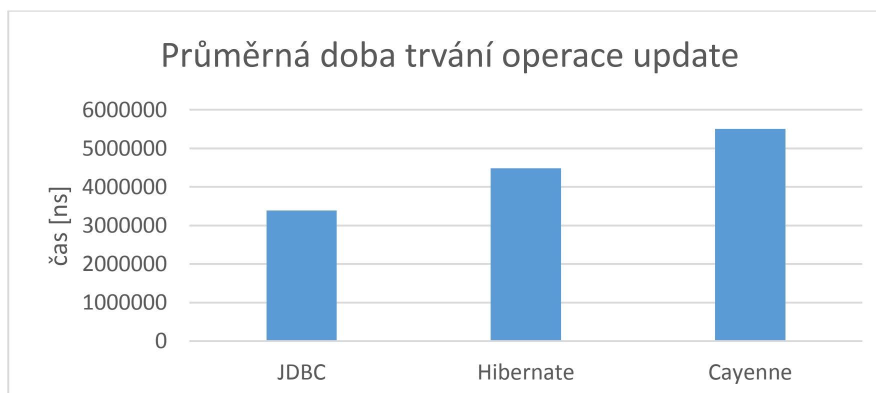
Graf 4: Operace insert

Tabulka 9: Statistiky pro insert

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	16442210	389190	439740	482720
Hibernate	9507460	73160	145060	166600
Cayenne	3903770	780	1240	1470

Testování operace *insert* jasně ukázalo, že nejvýkonnější je Cayenne, následován Hibernate a JDBC je na tom nejhůře.

8.4 Operace UPDATE



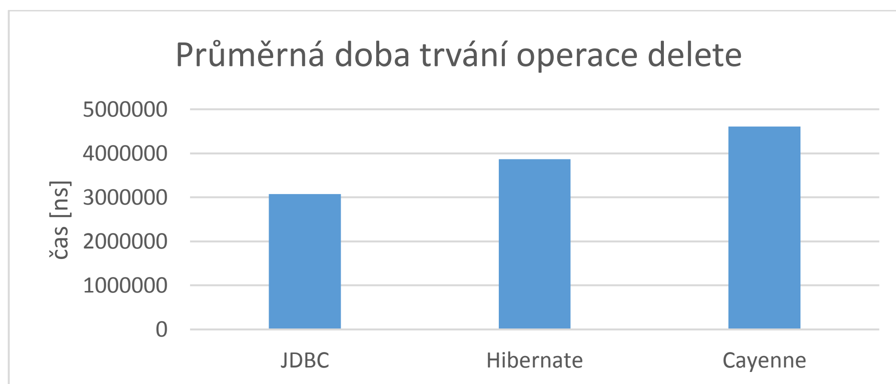
Graf 5: Operace update

Tabulka 10: Statistiky pro update

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	77435900	283410	2037730	2275570
Hibernate	85488230	745070	2514450	2996805
Cayenne	86038870	14260	3311905	4101290

Výsledky všech tří technologií v testu operace *update* jsou srovnatelné, avšak nejvýkonnější je JDBC, druhé nejvýkonnější řešení je aplikační rámec Hibernate a poslední je Cayenne.

8.5 Operace DELETE



Graf 6: Operace DELETE

Tabulka 11: Statistiky pro delete

	Max [ns]	Min [ns]	Medián [ns]	75% percentil [ns]
JDBC	99962430	96460	1872630	2140085
Hibernate	88108500	930570	2341005	2670335
Cayenne	87584780	47260	3193680	3576148

Poslední test, operace *delete*, opět ukazuje, jak si jednotlivá řešení konkurují, ale opět je vidět, že nejvýkonnější je JDBC.

8.6 Shrnutí výsledků

Nejjednodušší vyhodnocení výsledků bude pomocí udělování bodů jednotlivým řešením podle dílčích testů. Nejlepší řešení získá tři body, následující dva body a poslední jeden bod.

Tabulka 12: Výsledky testování rychlosti operací select, insert, update a delete

Řešení:	**JDBC**	**Hibernate**	**Cayenne**
1. SELECT	3	2	1
2. SELECT	3	2	1
3. SELECT	3	2	1
INSERT	1	2	3
UPDATE	3	2	1
DELETE	3	2	1
Výsledek:	**16**	**14**	**8**

Zvolené bodové hodnocení ukazuje, že nejvýkonnější řešení nabízí JDBC, jak se očekávalo před vykonáním testů.

ZÁVĚR

V teoretické i praktické části jsme si ukázali několik aspektů, podle kterých lze určit, které řešení bude vhodné pro jaký typ aplikace. Mezi tyto aspekty patří pohodlí při vývoji, podpora specifikací a rychlost jednotlivých operací.

Pohodlí při vývoji je subjektivně nejlepší při použití aplikačního rámce Hibernate. Nástroje pro práci s Hibernate jsou součástí vývojového prostředí Netbeans – generování konfiguračních souborů, nastavení pro reverzní inženýrství databáze i generování metod. Cayenne bohužel vyžaduje použití dalšího nástroje, který mnoho vývojářů uznává, ale já jsem si jej neoblíbil. JDBC pak vývojáři nenabízí nic pro zjednodušení rutinních činností a proto je na tom z hlediska komfortu nejhůře.

Dalším hlediskem je rychlost prováděných operací. Na začátku jsem předpokládal, že nejrychlejší bude technologie JDBC, což se v testech potvrdilo. Pro vývoj aplikací, které kladou vysoká nároky na efektivní práci s databází je jako počáteční bod nejlepší aplikační rámec Hibernate, ve kterém se následně mohou optimalizovat dotazy prostřednictvím poskytnutého API pro psaní HQL dotazů.

Automatické generátory popsané v teoretické části práce jsou na tom v dnešní době všechny obdobně. Nabízejí pouze základní funkcionalitu pro generování jednoduchých operací s databází. Tady vidím ještě velký prostor pro vylepšení této technologie. Zatím je na tom asi nejlépe nástroj Cayenne Modeler, ve kterém také vidím nejjednodušší cestu, jak generování vylepšit a zpříjemnit pro vývojáře.

Z teoretické části vyplývá i doporučené použití jednotlivých řešení. JDBC mohu doporučit pouze začínajícím programátorům, kteří se chtějí seznámit s prací s databází, než pokročí a budou používat libovolný aplikační rámec, který je o úroveň výše než JDBC. Cayenne je vhodné pro použití jednotlivci, nebo malými týmy lidí, protože nepodporuje specifikaci JPA a může se kdykoliv libovolně změnit. Přejít na jinou technologii, nebo se přizpůsobit, bude pro danou skupinu lidí rychlou a jednoduchou záležitostí. Hibernate bych z důvodu, že specifikaci podporuje, doporučil do velkých korporací, kde na aplikaci bude spolupracovat několik týmů lidí. Specifikace pak bude zárukou toho, že pokud ji Hibernate přestane podporovat, nebo se vedení rozhodne použít jiné řešení, přechod na novou technologii nebude problém.

Součástí práce byla i práce s operačním systémem Linux. Zjistil jsem, že v tomto ohledu mám do budoucna prostor pro zlepšování, což se ukázalo, když se mně nepodařilo spustit měření zatížení databáze pomocí plánovaných úloh.

ZÁVĚR V ANGLIČTINĚ

The theoretical and practical parts, we have shown several features in which you can identify which solutions will be appropriate for the certain type of application. These aspects includes the comfort of development, support of specifications and speed of individual operations.

The best comfort in the development subjectively offers framework Hibernate. Tools for working with Hibernate are part of the NetBeans IDE - generating configuration files, settings for database reverse engineering and methods generation. Cayenne unfortunately requires use of another tool that many developers appreciates, but I did not favorited it. JDBC technology then offers nothing to simplify routine tasks and therefore is doing in terms of comfort worst.

Another aspect is the speed of the operations. At the beginning I thought it would be the fastest JDBC technology, which proved to be right assumption. For the development of applications that place high demands on effective work with databases, I have to reccomand as a starting point application framework Hibernate, which can further optimize queries via the provided API for writing HQL queries.

Automatic generators described in the theoretical part nowadays are all similar. They offers only basic functionality for generating simple database operations. I see still lots of space for improvements of this technology. Now, there's probably the best tool Cayenne Modeler, in which I see the easiest way to improve automatic generation and make it more pleasant for developers.

The theoretical part also shows the recommended use of the various solutions. I can only recommend JDBC for novice programmers who wants to learn to work with the database and then going forward and use any application framework, which is about one level higher than JDBC. Cayenne is suitable for use by individuals or small teams of people, because it does not support JPA specification and can be changed at any time. Make use of another technology for given group of people would be without troubles. On the other hand, Hibernate supports specification JPA, therefor it would be recommended in large corporations, where on the same application will work several different teams of people. Specifications will guarantee that if you stop supporting Hibernate or management decides to use another solution, the transition to the new technology will not be a problem.

Part of this work was also working with the Linux operating system. I found that in this matter I have lots of space for future improvements, which was shown to me when I failed to start measuring the load on the database using scheduled tasks.

SEZNAM POUŽITÉ LITERATURY

[1] REESE, George. *Database programming with JDBC and Java. 2nd ed.* Cambridge, Mass.: O\'Reilly, c2000, xvii, 328 p. Java series (O\'Reilly. ISBN 15-659-2616-1)

[2] KEITH, Mike a Merrick SCHNICARIOL. *Pro JPA 2: mastering the Java Persistence API.* New York: Distributed to the Book trade worldwide by Springer-Verlag New York, c2009, xxv, 503 p. Expert\'s voice in Java technology. ISBN 14-302-1956-4.

[3] BAUER, Christian a Gavin KING. *Hibernate in action.* Greenwich: Manning Publications, 2005, xxiii, 408 s. ISBN 19-323-9415-X.

[4] FOWLER, Martin. *Patterns of enterprise application architecture.* Boston: Addison-Wesley, c2003, xxiv, 533 s. The Addison-Wesley Signature Series. ISBN 978-0-321-12742-6.

[5] WALLS, Craig. *Spring in action. 3rd ed.* Shelter Island: Manning, c2011, xxiii, 400 p. ISBN 19-351-8235-8.

[6] *Documentation for Cayenne 3.1*. APACHE SOFTWARE FOUNDATION. Apache Cayenne [online]. [cit. 2014-06-11]. Dostupné z: https://cayenne.apache.org/docs/3.1/

[7] BAUER, Christian a Gavin KING. *Java persistence with Hibernate*. rev. ed. Greenwich: Manning Publications, c2007, xxiii, 408 s. ISBN 19-323-9488-5.

[8] *Java Data Objects (JDO)*. ORACLE. [online]. [cit. 2014-06-11]. Dostupné z: http://www.oracle.com/technetwork/java/index-jsp-135919.html.

[9] *EclipseLink/Documentation Center*. THE ECLIPSE FOUNDATION. Eclipsepedia [online]. [cit. 2014-06-11]. Dostupné z: http://wiki.eclipse.org/EclipseLink/Documentation_Center

[10] *Oracle JDeveloper 12c (12.12)*. ORACLE. Tutorials [online]. [cit. 2014-06-11]. Dostupné z: http://docs.oracle.com/cd/E37547_01/tutorials/toc.htm

[11] *An Eclipse Web Tools Platform Sub-Project*. ECLIPSE FOUNDATION. [online]. [cit. 2014-06-11]. Dostupné z: http://www.eclipse.org/webtools/dali/

[12] *JPM2Java: Java Persistence API code generator*. HTTPS://JPM2JAVA.DEV.JAVA.NET. [online]. [cit. 2014-06-11]. Dostupné z: https://jpm2java.java.net/

[13] *RevGen Documentation*. OUTSOURCE CAFE, Inc. RevGen Introduction [online]. [cit. 2014-06-11]. Dostupné z: http://outsourcecafe.com/revgen/

[14] MAYDENE FISHER, Jon Ellis. *JDBC API tutorial and reference. 3rd ed*. Boston, MA: Addison-Wesley, 2004. ISBN 978-032-1173-843.

[15] SPEEGLE, Gregory D. *JDBC: practical guide for Java programmers. 2nd ed*. San Francisco: MK/Morgan Kaufmann Publishers, c2002, xiii, 113 p. Java series (O'Reilly. ISBN 15-586-0736-6.

[16] KOGENT SOLUTIONS, Inc. *Java server programming: Java EE5 (J2EE 1.5)*. Platinum ed. New Delhi, India: Dreamtech Press, 2010. ISBN 81-772-2835-8.

[17] KOGENT SOLUTIONS, Inc. *Java persistence with jpa 2.1: Java EE5 (J2EE 1.5)*. Platinum ed. S.l.: Outskirts Press, 2013. ISBN 14-787-0019-X.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

JDBC	Java Database Connectivity.
ORM	Object Relational Mapping
SQL	Structured Query Language
HQL	Hibernate Query Language
JPA	Java Persistence API
JTA	Java Transaction API
JNDI	Java Naming and Directory Interface
JPA QL	JPA Query Language
DTO	Data Transfer Object
DAO	Data Access Object (Table Row Gateway, viz kapitola *Data Access Object*)
JVM	Java Virtual Machine
AWT	Abstract Window Toolkit
ODBC	Open Database Connectivity
API	Application Interface
GUI	Graphics User Interface
PHP	Hypertext preprocesor
CRUD	Create, read, update, delete

SEZNAM OBRÁZKŮ

Obrázek 1: Architektura JDBC [1]
Obrázek 2: JDBC odděluje aplikaci od specifikací jednotlivých implementací databází [1]
Obrázek 3: Vysokoúrovňový přehled vrstvové architektury Hibernate API [3]
Obrázek 4: Provedení operace find v návrhovém vzoru Row Data Gateway [4]
Obrázek 5: Získání dat z databáze [4]
Obrázek 6: Ukázka redukce volaných metod [4]
Obrázek 7: Práce s daty v návrhovém vzoru Lazy Load
Obrázek 8: EER diagram testovací databáze
Obrázek 9: Struktura projektu JdbcTestDb

SEZNAM TABULEK

Tabulka 1: Argumenty pro PrepareTestData

Tabulka 2: Hodnoty argumentů pro testovací programy

Tabulka 3: Formát výstupu testovacích programů

Tabulka 4: Úrovně logování v LogLevelEnum

Tabulka 5: Argumenty pro finální programy

Tabulka 6: Statistiky pro jednoduchý select

Tabulka 7: Statistiky pro select pomocí generovaných metod

Tabulka 8: Statistiky pro select obsahující join

Tabulka 9: Statistiky pro insert

Tabulka 10: Statistiky pro update

Tabulka 11: Statistiky pro delete

Tabulka 12: Výsledky testování rychlosti operací select, insert, update a delete

SEZNAM PŘÍLOH

1 Příloha P I: SQL dotazy (Testovací databáze)

2 Příloha P II: SQL dotazy (databáze Wikipedie)

3 Příloha P III: Záznamy cron pro spouštění testů

PŘÍLOHA P I: SQL DOTAZY (TESTOVACÍ DATABÁZE)

Dotaz pro operaci select:

```
SELECT DISTINCT t1.textid, t5.color FROM table5 t5 INNER JOIN table4 t4 ON t5.idt5 = t4.color_id  INNER JOIN table1 t1 ON t1.table4_id = t4.idt4 WHERE t4.color_id = ?;
```

Insert:

```
INSERT INTO table2 (randvalue, randtext) VALUES (?, ?);
```

Update:

```
UPDATE table4 SET street = ?, city = ?, post_code = ? WHERE idt4 = ?;
```

Delete:

```
DELETE FROM table2 WHERE idt2 = ?;
```

PŘÍLOHA P II: SQL DOTAZY (DATABÁZE WIKIPEDIE)

Dotaz pro operaci select:

```
SELECT /* WikiPage::pageData  */
page_id,page_namespace,page_title,CONVERT(page_restrictions USING
utf8) as
page_restrictions,page_counter,page_is_redirect,page_is_new,page_r
andom,CONVERT(page_touched USING utf8)
page_touched,page_latest,page_len FROM `page`  WHERE
page_namespace = ? AND page_title = ?  LIMIT 1;
```

Select (pro testování automaticky generovaných operací aplikačních rámců Hibernate a Cayenne):

```
     SELECT * FROM `page`  WHERE page_id = ?;
```

Select (obsahující JOIN operaci):

```
     SELECT r.* FROM text t JOIN revision r ON r.rev_text_id =
t.old_id WHERE t.old_id = ?;
```

Insert:

```
     INSERT INTO `text` (old_text, old_flags) VALUES (CONVERT(?
USING binary), CONVERT(? USING binary));
```

Update:

```
     UPDATE text SET old_text = CONVERT(? USING BINARY), old_flags
= CONVERT(? USING BINARY) WHERE old_id = ?;
```

Delete:

```
     DELETE FROM text WHERE old_id = ?;
```

PŘÍLOHA P III: ZÁZNAMY CRON PRO SPOUŠTĚNÍ TESTŮ

```
# m h  dom mon dow command
#SELECT
0 0 31 * * service mysql restart
5 0 31 * * nohup java -jar /home/roman/dist/JdbcWiki.jar -a s -n 10000 -i 100 -o /home/roman/log/final/test01-jdbc-select-wiki.csv > /home/roman/log/final/test01-jdbc-select-wiki.log 2>$1$
20 0 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-select-mytop01.txt &
35 0 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-select-mytop02.txt &
50 0 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-select-mytop03.txt &
#
0 4 31 * * service mysql restart
5 4 31 * * nohup java -jar /home/roman/dist/HibernateWiki.jar -a s -n 10000 -i 100 -o /home/roman/log/final/test01-hibernate-select-wiki.csv > /home/roman/log/final/test01-hibernate-select-wiki.log 2>$1$
20 4 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-select-mytop01.txt &
35 4 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-select-mytop02.txt &
50 4 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-select-mytop03.txt &
#
0 8 31 * * service mysql restart
5 8 31 * * nohup java -jar /home/roman/dist/CayenneWiki.jar -a s -n 10000 -i 100 -o /home/roman/log/final/test01-cayenne-select-wiki.csv > /home/roman/log/final/test01-cayenne-select-wiki.log 2>$1$
20 8 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-select-mytop01.txt &
35 8 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-select-mytop02.txt &
50 8 31 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-select-mytop03.txt &
#
#INSERT
0 12 31 * * service mysql restart
5 12 31 * * nohup java -jar /home/roman/dist/JdbcWiki.jar -a i -n 10000 -i 100 -o /home/roman/log/final/test01-jdbc-insert-wiki.csv > /home/roman/log/final/test01-jdbc-insert-wiki.log 2>$1$
```

```
20 12 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-insert-mytop01.txt &

35 12 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-insert-mytop02.txt &

50 12 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-insert-mytop03.txt &

#

0 16 31 * * service mysql restart

5 16 31 * * nohup java -jar
/home/roman/dist/HibernateWiki.jar -a i -n 10000 -i 100 -o
/home/roman/log/final/test01-hibernate-insert-wiki.csv >
/home/roman/log/final/test01-hibernate-insert-wiki.log 2>$1$

20 16 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-hibernate-insert-mytop01.txt &

35 16 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-hibernate-insert-mytop02.txt &

50 16 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-hibernate-insert-mytop03.txt &

#

0 20 31 * * service mysql restart

5 20 31 * * nohup java -jar /home/roman/dist/CayenneWiki.jar
-a i -n 10000 -i 100 -o /home/roman/log/final/test01-cayenne-
insert-wiki.csv > /home/roman/log/final/test01-cayenne-insert-
wiki.log 2>$1$

20 20 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-cayenne-insert-mytop01.txt &

35 20 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-cayenne-insert-mytop02.txt &

50 20 31 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-cayenne-insert-mytop03.txt &

#

#UPDATE

0 0 1 * * service mysql restart

5 0 1 * * nohup java -jar /home/roman/dist/JdbcWiki.jar -a u
-n 10000 -i 100 -o /home/roman/log/final/test01-jdbc-update-
wiki.csv > /home/roman/log/final/test01-jdbc-update-wiki.log 2>$1$

20 0 1 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-update-mytop01.txt &

35 0 1 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-update-mytop02.txt &

50 0 1 * * mytop -u root -p Passw0rd >&
/home/roman/log/final/mytop/test01-jdbc-update-mytop03.txt &

#

0 4 1 * * service mysql restart
```

```
5 4 1 * * nohup java -jar /home/roman/dist/HibernateWiki.jar -a u -n 10000 -i 100 -o /home/roman/log/final/test01-hibernate-update-wiki.csv > /home/roman/log/final/test01-hibernate-update-wiki.log 2>$1$

20 4 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-update-mytop01.txt &

35 4 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-update-mytop02.txt &

50 4 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-update-mytop03.txt &

#

0 8 1 * * service mysql restart

5 8 1 * * nohup java -jar /home/roman/dist/CayenneWiki.jar -a u -n 10000 -i 100 -o /home/roman/log/final/test01-cayenne-update-wiki.csv > /home/roman/log/final/test01-cayenne-update-wiki.log 2>$1$

20 8 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-update-mytop01.txt &

35 8 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-update-mytop02.txt &

50 8 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-update-mytop03.txt &

#

#DELETE

0 12 1 * * service mysql restart

5 12 1 * * nohup java -jar /home/roman/dist/JdbcWiki.jar -a d -n 10000 -i 100 -o /home/roman/log/final/test01-jdbc-delete-wiki.csv > /home/roman/log/final/test01-jdbc-delete-wiki.log 2>$1$

20 12 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-delete-mytop01.txt &

35 12 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-delete-mytop02.txt &

50 12 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-jdbc-delete-mytop03.txt &

#

0 16 1 * * service mysql restart

5 16 1 * * nohup java -jar /home/roman/dist/HibernateWiki.jar -a d -n 10000 -i 100 -o /home/roman/log/final/test01-hibernate-delete-wiki.csv > /home/roman/log/final/test01-hibernate-delete-wiki.log 2>$1$

20 16 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-delete-mytop01.txt &

35 16 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-delete-mytop02.txt &
```

```
50 16 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-hibernate-delete-mytop03.txt &

#

0 20 1 * * service mysql restart

5 20 1 * * nohup java -jar /home/roman/dist/CayenneWiki.jar -a d -n 10000 -i 100 -o /home/roman/log/final/test01-cayenne-delete-wiki.csv > /home/roman/log/final/test01-cayenne-delete-wiki.log 2>$1$

20 20 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-delete-mytop01.txt &

35 20 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-delete-mytop02.txt &

50 20 1 * * mytop -u root -p Passw0rd >& /home/roman/log/final/mytop/test01-cayenne-delete-mytop03.txt &
```