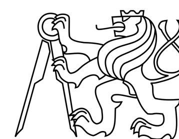
ESKÉ VYSOKÉ U   ENÍ TECHNICKÉ V PRAZE

FAKULTA INFORMA   NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ   SKÉ PRÁCE

Název: Android knihovna pro objektov -rela ní mapování

Student: Jan Bína

Vedoucí: Ing. Vladislav Skoumal

Studijní program: Informatika

Studijní obor: Teoretická informatika

Katedra: Katedra teoretické informatiky

Platnost zadání: Do konce zimního semestru 2018/19

Pokyny pro vypracování

Rozši te aplika ní rozhraní Android opensource knihovny Joogar mapující rela ní databázi na objektový model. Realizujte rešerši existujících knihoven tohoto typu a zmapujte jejich výhody a nedostatky. Prove te analýzu funk ních i systémových požadavk . Navrhn te nástroje a jiné produkty t etích stran pot ebné pro realizaci aplika ního rozhraní a pro návrh jeho architektury. Navrhn te zp sob testování a následného nasazení do ostrého provozu. Aplikujte výkonové testy. Zpracujte dokumentaci. Po dohod   s vedoucím práce realizujte prototyp aplika ního rozhraní a mobilní aplikaci, která jej bude využívat.

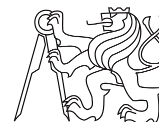
Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D. vedoucí katedry	prof. Ing. Pavel Tvrdík, CSc. d kan

V Praze dne 2. b ezna 2017

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Bakalářská práce

Android knihovna pro objektově-relační mapování

Jan Bína

Vedoucí práce: Ing. Vladislav Skoumal

10. května 2017

Poděkování

Děkuji vedoucímu práce, Ing. Vladislavu Skoumalovi, za vedení práce.
Děkuji svým rodičům za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2017 . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2017 Jan Bína. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Bína, Jan. *Android knihovna pro objektově-relační mapování*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato bakalářská práce se zabývá návrhem a implementací Android knihovny pro objektově-relační mapování. Popisuje a porovnává dva různé způsoby implementace. Na základě požadavků a poznatků z porovnání existujících řešení navrhuje aplikační rozhraní. Pro jeho následnou implementaci je využito generování kódu.

Klíčová slova Android knihovna, relační databáze, objektově-relační mapování, zpracování anotací, generování kódu, SQLite, Java, Android

Abstract

This bachelor thesis deals with the design and implementation of Android library for object-relational mapping. It describes and compares two different ways of implementation. Based on the requirements and analysis of existing solutions, it designs an application programming interface. Code generation is used for its subsequent implementation.

Keywords Android library, relational database, object-relational mapping, annotation processing, code generation, SQLite, Java, Android

Obsah

Úvod **1**

1 Úvod do problematiky **3**
- 1.1 SQLite . . . 3
- 1.2 Android API pro SQLite . . . 4
- 1.3 Objektově-relační mapování . . . 6

2 Knihovny pro objektově-relační mapování **9**
- 2.1 Rozdělení . . . 9
- 2.2 Výběr knihoven a způsob hodnocení . . . 10
- 2.3 ORMLite . . . 11
- 2.4 GreenDAO . . . 15
- 2.5 DBFlow . . . 19
- 2.6 Joogar . . . 23
- 2.7 Squeaky . . . 26

3 Návrh **27**
- 3.1 Požadavky . . . 27
- 3.2 Návrh API . . . 28

4 Realizace **31**
- 4.1 Nástroje potřebné pro realizaci . . . 31
- 4.2 Anotace . . . 31
- 4.3 Anotační procesor . . . 33
- 4.4 JavaPoet . . . 34
- 4.5 Implementace . . . 37

5 Testování **41**
- 5.1 Jednotkové a integrační testy . . . 41
- 5.2 Výkonové testy . . . 43

Závěr **47**
Možnosti rozšíření 47

Literatura **49**

A Seznam použitých zkratek **53**

B Obsah přiloženého média **55**

Seznam obrázků

5.1 Srovnání rychlostí zápisu 44
5.2 Srovnání rychlostí čtení 45

Seznam tabulek

2.1 Žebříček nejpoužívanějších databázových knihoven pro Android dle statistik serveru AppBrain [1] 11

Seznam ukázek kódu

1.1 Správa databáze pomocí třídy SQLiteOpenHelper 4
1.2 Předpisy vybraných metod pro úpravu a čtení dat z databáze . 6
1.3 Použití metody delete 6
1.4 Vytvoření a použití předpřipraveného příkazu 6
1.5 Použití ContentValues pro ukládání do databáze 7
1.6 Čtení dat z Cursoru 7
2.1 Nastavení hodnoty proměnné pomocí reflexe 10
2.2 Inicializace knihovny ORMLite 14
2.3 Třída nastavená pro použití s knihovnou ORMLite 14
2.4 Dotaz do databáze v knihovně ORMLite 15
2.5 Inicializace databáze v knihovně GreenDAO 18
2.6 Třída nastavená pro použití s knihovnou GreenDAO 18
2.7 Dotaz do databáze v knihovně GreenDAO 19
2.8 Inicializace knihovny DBFlow 21
2.9 Třída nastavená pro použití s knihovnou DBFlow 22
2.10 Dotaz do databáze v knihovně DBFlow 22
2.11 Inicializace knihovny Joogar 25
2.12 Třída nastavená pro použití s knihovnou Joogar 25
2.13 Dotaz do databáze v knihovně Joogar 26
3.1 Sestavení dotazu do databáze 30
3.2 Sestavení klauzule where pomocí vygenerované třídy 30
4.1 Příklad použití anotace s parametry 32
4.2 Příklad použití anotace s jediným paremetrem *value* 32
4.3 Definice anotace 33
4.4 Kostra anotačního procesoru 35
4.5 Použití knihovny JavaPoet 36
4.6 Kód vygenerovaný knihovnou JavaPoet z ukázky 4.5 36
4.7 Instrukce pro build systém gradle 37
4.8 Vytvoření a použití připraveného příkazu SQLiteStatement . . 38
4.9 Funkční rozhraní EntityBuilder 38

5.1 JUnit test, který poběží na zařízení s OS Android 43

Úvod

Databáze, jakožto prostředek pro ukládání dat, je nedílnou součástí většiny mobilních aplikací. Ačkoliv existují alternativy, nejvíce používané jsou stále databáze relační, v případě zařízení s OS Android konkrétně databáze SQLite. Problémem relačních databází je nutnost převodu dat mezi objekty programovacího jazyka a záznamy této databáze, což vyžaduje značné množství opakujícího se kódu. Existuje tak řada knihoven, které se snaží tento problém řešit. Využívají k tomu techniku zvanou objektově-relační mapování (ORM).

Tato práce v kapitole *Knihovny pro objektově-relační mapování* popisuje dva různé způsoby implementace takové knihovny a porovnává existující knihovny, které jsou jedním z těchto způsobů implementovány.

Cílem práce je upravit open-source ORM knihovnu *Joogar* [2] tak, aby fungovala na principu generování kódu a využít tohoto způsobu implementace k rozšíření jejího aplikačního rozhraní. Návrhem API se zabývá kapitola *Návrh*, implementací pak kapitola *Realizace*. Poslední kapitola *Testování* porovnává rychlosti zápisu a čtení do databáze pomocí několika ORM knihoven, včetně původní i upravené verze knihovny *Joogar*.

Kapitola 1

Úvod do problematiky

1.1 SQLite

Pro ukládání strukturovaných dat se v zařízeních s OS Android používá SQLite databáze. Je to jednoduchá relační databáze s otevřeným zdrojovým kódem. Její hlavní výhodou předurčující ji k použití na mobilních zařízeních je to, že na rozdíl od většiny ostatních SQL databází běží přímo v procesu aplikace a nevyžaduje zvláštní proces pro server. SQLite přímo čte a zapisuje do obyčejných souborů na disku, přičemž celá databáze včetně indexů a pohledů je uložena v jediném souboru.

SQLite databáze podporuje jen omezené množství datových typů, jsou to:

- `NULL` – hodnota null
- `INTEGER` – celé číslo, uloženo maximálně v 8 bytech
- `REAL` – desetinné číslo, uloženo v 8 bytech
- `TEXT` – textový řetězec
- `BLOB` – binární data

SQLite používá dynamické typování. Při vytváření databáze sice určujeme datové typy jednotlivých sloupečků, ty slouží ale jen jako jakési doporučení, sloupeček může obsahovat jakýkoliv datový typ. Jedinou výjimkou je sloupeček vytvořený jako `INTEGER PRIMARY KEY`, do kterého můžeme ukládat jen hodnoty typu integer. [3]

[4, 5]

1.2 Android API pro SQLite

V následujících podkapitolách si popíšeme nejdůležitější části Android API pro práci s SQLite databází. To se nachází v balíčku `android.database.SQLite`.

1.2.1 Vytvoření databáze

Pro vytváření a upravování databáze slouží třída `SQLiteOpenHelper` [6], kterou můžeme vidět v ukázce 1.1. Ta ve svém konstruktoru přijímá `Context`, jméno databáze, třídu implementující rozhraní `CursorFactory` pro konstrukci Cursoru (může být `null` pro výchozí chování) a verzi databáze (celé číslo větší než 0). Obsahuje také dvě abstraktní metody, které musíme implementovat. Metoda `onCreate` je volána po prvním vytvoření databáze a měli bychom v ní vytvořit požadované tabulky a případně je naplnit výchozími hodnotami. Metoda `onUpgrade` je volána, pokud je verze existující databáze menší než verze předaná konstruktoru třídy `SQLiteOpenHelper`. Slouží zejména k úpravě a odstranění existujících tabulek, případně k vytvoření tabulek nových.

```
1  public class DbHelper extends SQLiteOpenHelper {
2      public static final int DATABASE_VERSION = 1;
3      public static final String DATABASE_NAME = "Database.db";
4  
5      public DbHelper(Context context) {
6          super(context, DATABASE_NAME, null, DATABASE_VERSION);
7      }
8  
9      public void onCreate(SQLiteDatabase db) {...}
10 
11     public void onUpgrade(SQLiteDatabase db, int oldVersion, int
        ↪  newVersion) {...}
12 }
```

Ukázka kódu 1.1: Správa databáze pomocí třídy SQLiteOpenHelper

1.2.2 Upravování a čtení dat z databáze

K upravování a čtení dat z databáze slouží třída `SQLiteDatabase` [7], jejíž instanci získáme z instance třídy `SQLiteOpenHelper` voláním jedné z metod `getWriteableDatabase`, `getReadableDatabase`. Předpisy vybraných metod pro úpravu a čtení dat můžeme vidět v ukázce 1.2.

Metoda `execSQL` slouží k provedení libovolného SQL příkazu, nehodí se však

pro příkazy vracející nějaká data (SELECT, INSERT, UPDATE, DELETE). Vhodná je například pro příkazy vytvářející a upravující tabulku.

Metodu `insert` použijeme k vložení řádku do tabulky. Pro reprezentaci sloupečků a jejich hodnot se používá třída `ContentValues` (viz kapitola 1.2.3). Metoda vrací `id` vloženého řádku, případně hodnotu -1, pokud vložení neproběhlo.

Metoda `update` slouží k úpravě existujících řádků, které splňují podmínku danou parametry `whereClause` a `whereArgs`. Vrací počet upravených řádků.

Metoda `delete` smaže řádky vyhovující podmínkám a vrátí jejich počet.

Metody `rawQuery` a `query` v databázi najdou řádky odpovídající parametrům, které jim předáme a vrátí nám je v podobě `Cursoru` (viz kapitola 1.2.4). Zatímco metoda `rawQuery` přijímá rovnou celý SQL příkaz, metoda `query` přijímá odděleně jeho jednotlivé části a příkaz z nich sestaví za nás.

Parametr `whereArgs`, vyskytující se v některých metodách doplňuje parametr `whereClause` a umožňuje nám v dotazu použít otazníky namísto hodnot. Ty budou později nahrazeny hodnotami z parametru `whereArgs`, přičemž budou automaticky ošetřeny znaky, které mají v SQL příkazech zvláštní význam. Příklad takového použití můžeme vidět v ukázce 1.3.

Pro často používané příkazy se hodí třída `SQLiteStatement`. Ta reprezentuje předpřipravený příkaz, který stačí jen naplnit aktuálními hodnotami a spustit. Tento příkaz můžeme využít k ukládání, upravování a mazání dat z databáze, dále nám umožňuje přečíst z databáze jedinou hodnotu typu `Long` nebo `String`, případně vykonat libovolný SQL příkaz. Nemůžeme ho však využít ke čtení celých řádků z tabulky. Vytvoření a použití takového příkazu můžeme vidět v ukázce 1.4.

1.2.3 ContentValues

ContentValues [8] je jednoduchá třída, která drží dvojice klíč–hodnota, kde klíč je jméno sloupečku tabulky. Používá se pro vkládání a upravování řádků tabulek (viz ukázka 1.2). Příklad použití můžeme vidět v ukázce 1.5.

1.2.4 Cursor

Cursor [9] je objekt reprezentující výsledek dotazu do databáze. Ukazuje na jeden řádek výsledku. Čtení dat z databáze probíhá až ve chvíli, kdy zavoláme metodu Cursoru, která je vyžaduje, čímž se výrazně šetří paměť. Mezi řádky se pohybujeme pomocí metod `moveToFirst()`, `moveToNext()`,

`moveToPosition(int position)` a dalších. Ke zjištění počtu řádků a pozice Cursoru slouží mimo jiné metody `getCount()`, `getPosition()`, `isFirst()`, nebo `isLast()`. Hodnoty sloupečků pak z Cursoru čteme pomocí několika `get*(int position)` metod. Ty berou jako parametr pořadové číslo sloupečku. Pokud známe jen jméno sloupečku, můžeme jeho index zjistit pomocí metody `getColumnIndex(String name)`. Cursor by měl být po dokončení práce uzavřen metodou `close()`. Příklad čtení dat z Cursoru můžeme vidět v ukázce 1.6. [10]

```
1 void execSQL(String sql);
2 long insert(String table, String nullColumnHack, ContentValues
   ↪ values);
3 int  update(String table, ContentValues values, String whereClause,
   ↪ String[] whereArgs);
4 int  delete(String table, String whereClause, String[] whereArgs)
5
6 Cursor rawQuery(String sql, String[] selectionArgs);
7 Cursor query(String table, String[] columns, String selection,
   ↪ String[] selectionArgs, String groupBy, String having, String
   ↪ orderBy, String limit);
```

Ukázka kódu 1.2: Předpisy vybraných metod pro úpravu a čtení dat z databáze

```
1 db.delete("table", "user_name = ?", new String[]{username});
```

Ukázka kódu 1.3: Použití metody delete

```
1 SQLiteStatement statement = db.compileStatement(
2         "INSERT INTO users (name, birthYear) VALUES (?, ?)");
3
4 statement.bindString(1, "Joe");
5 statement.bindLong(2, 1990);
6 long id = statement.executeInsert();
```

Ukázka kódu 1.4: Vytvoření a použití předpřipraveného příkazu

1.3 Objektově-relační mapování

V objektově orientovaných jazycích, jakým je i Java, pracujeme s daty ve formě objektů. Ty mohou obsahovat proměnné nejrůznějších datových typů, zatímco

relační databáze je omezena na několik jednodušších, SQLite konkrétně na výše zmíněné čtyři. ORM je technika převodu dat mezi řádky relační databáze a objekty objektově orientovaného jazyka.

```
ContentValues values = new ContentValues();

values.put("username", "John");
values.put("birthYear", 1990);

long id = db.insert("users", null, values);
```

Ukázka kódu 1.5: Použití ContentValues pro ukládání do databáze

```
Cursor cursor = fetchData();

if (cursor.moveToFirst()) {
    int usernameIndex = cursor.getColumnIndex("username");
    do {
        String username = cursor.getString(usernameIndex);
        ...
    } while (cursor.moveToNext());
    cursor.close();
}
```

Ukázka kódu 1.6: Čtení dat z Cursoru

Kapitola 2

Knihovny pro objektově-relační mapování

2.1 Rozdělení

Knihovny pro objektově-relační mapování můžeme na základě principu jejich fungování rozdělit do dvou kategorií. Ty, které fungují na principu reflexe a ty, které generují zdrojový kód.

2.1.1 Knihovny fungující na principu reflexe

Reflexe je vlastnost programovacího jazyka, která umožňuje za běhu programu zjistit informace o objektu a dokonce ho i měnit. Java nabízí programátorům *Reflection API* [11], pomocí kterého můžeme z jakékoliv třídy získat všechny deklarované proměnné a metody. Z proměnných můžeme poté získat jejich hodnotu, nebo jim hodnotu nastavit, metody můžeme volat. V knihovně pro objektově-relační mapování se reflexe využívá k serializaci a deserializaci objektů.

Nevýhodou je, že přistupování k proměnným a metodám pomocí reflexe je pomalejší než přístup standardní cestou. Například podle článku Františka Kučery [12], který porovnával rychlost nastavení hodnoty proměnné pomocí reflexe a přímého přístupu, je reflexe v nejlepším případě třikrát pomalejší.

Příklad použití reflexe k nastavení hodnoty proměnné můžeme vidět v ukázce 2.1.

```
Object object = new MyObject();

Field field = object.getClass().getDeclaredField("fieldName");
field.setAccessible(true);
field.set(object, 37);
```

Ukázka kódu 2.1: Nastavení hodnoty proměnné pomocí reflexe

2.1.2 Knihovny generující zdrojový kód

Generování zdrojového kódu probíhá na rozdíl od reflexe před kompilací. Využívá se k tomu nejčastěji anotační procesor, který prostřednictvím anotací prochází námi napsaný zdrojový kód a může vytvářet soubory se zdrojovými kódy, které budou následně zkompilovány. Více o fungování anotačního procesoru v kapitole *Realizace*.

Výhoda tohoto přístupu spočívá v tom, že vygenerovaný kód nemusí proměnné číst a nastavovat pomocí reflexe. Díky znalosti kódu můžeme navíc generovat metody typu `findUserByName`, protože víme, že třída `User` obsahuje proměnnou `name`.

2.2 Výběr knihoven a způsob hodnocení

Při výběru knihoven jsme vycházeli ze statistik nejpoužívanějších databázových knihoven v Android aplikacích, které poskytuje server AppBrain [1]. Podoba této statistiky ke dni citace je zachycena v tabulce 2.1. V tabulce je u každé knihovny uveden její typ. Kromě knihoven fungujích na principu reflexe a knihoven generujích zdrojový kód zde máme také dvě knihovny, které neukládají data do relační databáze (typ NoSQL). V následujícím srovnání se budeme zabývat knihovnami ORMLite, greenDAO a DBFlow.

Při hodnocení knihoven se zaměříme na způsob inicializace, nároky kladené na model a zejména na API, které nám knihovna poskytuje pro čtení a zapisování do databáze.

Tabulka 2.1: Žebříček nejpoužívanějších databázových knihoven pro Android dle statistik serveru AppBrain [1]

Jméno	Typ knihovny	Procento instalací
Firebase	NoSQL	36,83 %
ORMLite	Reflexe	1,42 %
greenDAO	Generování kódu	0,94 %
ActiveAndroid	Reflexe	0,24 %
SugarORM	Reflexe	0,22 %
MongoDB Java Driver	NoSQL	0.12 %
DBFlow	Generování kódu	0.11 %

2.3 ORMLite

ORMLite [13] je dle zmiňované statistiky nejpoužívanější ORM knihovnou která používá jako úložiště relační databázi. Není určena přímo pro Android, ale obecně pro Javu. Funguje na principu reflexe, takže nepatří k nejrychlejším.

2.3.1 Inicializace

Pro inicializaci musíme vytvořit třídu rozšiřující `OrmLiteSqliteOpenHelper`, na jejímž pozadí se nachází `SQLiteOpenHelper` z Android API. Z její instance následně získáme objekt typu `ConnectionSource`, který potřebujeme pro vytvoření tabulek pomocí třídy `TableUtils`. Zjednodušený příklad inicializace knihovny je v ukázce 2.2.

2.3.2 Nároky na model

Knihovna využívá anotace. Třídy, které chceme ukládat do databáze musíme označit anotací `@DatabaseTable`. Primární klíč může být tvořen jen jedním sloupečkem, datový typ `long` nebo `String`. Každá proměnná, která má být ukládána, musí být označena anotací `@DatabaseField`, pomocí jejíchž parametrů můžeme dále specifikovat, jestli se jedná o primární klíč, název sloupečku databáze a další vlastnosti. Ve výchozím stavu jsou hodnoty čteny a nastavovány proměnným přímo, můžeme si ale vyžádat i použití getterů a setterů. Díky použití reflexe nejsou kladeny žádné nároky na přístupnout proměnných, mohou být i *private*. Vyžadován je bezparametrický konstruktor. Příklad takové třídy je v ukázce 2.3.

2.3.3 API pro čtení a zápis

Pro čtení a zapisování dat do databáze potřebujeme instanci objektu `Dao`. Tu získáme z naší třídy rozšiřující `OrmLiteSqliteOpenHelper`, jak je vidět v ukázce 2.2.

Pro ukládání jsou k dispozici následující metody:

```
int create(T data);
T createIfNotExists(T data);
CreateOrUpdateStatus createOrUpdate(T data);
```

Metoda `create` uloží objekt do databáze, přičemž nebere v úvahu jeho primární klíč – měl by tak vždy vzniknout nový řádek v databázi. Metoda `createIfNotExists` uloží objekt do databáze, ale jen pokud ta už neobsahuje řádek se stejným primárním klíčem. Podle dokumentace to dělá tak, že nejdříve provede vyhledávání v databázi podle primárního klíče objektu, a pokud nic nenajde, zavolá metodu `create`. Metoda `createOrUpdate` uloží objekt do databáze, pokud již v databázi existuje, upraví daný záznam. Opět nejdříve vyhledává v databázi podle primárního klíče a na základě výsledku volá buď metodu `create` nebo `update`.

Pro upravování jsou k dispozici následující metody:

```
int update(T data);
int updateId(T data, ID newId);
int update(PreparedUpdate<T> preparedUpdate);
```

Metoda `update` upraví řádek v tabulce podle odpovídajícího primárního klíče. Metoda `updateId` navíc změní hodnotu primárního klíče na hodnotu specifikovanou parametrem `newId`. Poslední metoda s parametrem `preparedUpdate` se hodí pro upravování více řádků tabulky najednou na základě podmínek specifikovaných pomocí třídy `Where` (o té více později).

Pro mazání jsou k dispozici následující metody:

```
int delete(T data);
int deleteById(ID id);
int delete(Collection<T> datas);
int deleteIds(Collection<ID> ids);
int delete(PreparedDelete<T> preparedDelete);
```

První čtyři nám umožňují mazat z databáze na základě primárního klíče, který si buď zjistí z předaného objektu, případně jim ho předáme parametrem přímo. Poslední metoda s parametrem `preparedDelete` nám opět umožňuje mazat na základě podmínek specifikovaných pomocí třídy `Where`.

Pro čtení dat jsou k dispozici následující metody:

```
long countOf();
long countOf(PreparedQuery<T> preparedQuery);
T queryForId(ID id);
T queryForSameId(T data);
T queryForFirst(PreparedQuery<T> preparedQuery);
List<T> query(PreparedQuery<T> preparedQuery);
List<T> queryForAll();
List<T> queryForEq(String fieldName, Object value);
List<T> queryForMatching(T matchObj);
List<T> queryForFieldValues(Map<String, Object> fieldValues);
```

Funkce většiny z nich by měla být zřejmá. Za zmínku stojí metoda `queryForEq`, která hledá záznamy jejichž sloupeček specifikovaný parametrem `fieldName` má stejnou hodnotu jako parametr `value`. Metoda `queryForMatching` hledá na základě hodnot předaného objektu, ale pouze těch, které mají jinou než výchozí hodnotu – pokud bude tedy objekt obsahovat například proměnnou typu `boolean` s hodnotou `false`, nebude tato použita.

Komplexní dotazy do databáze nám umožňuje třída `QueryBuilder`. Ta navíc může vracet výsledek dotazu v podobě třídy `ClosableIterator`, která konstruuje jednotlivé objekty až ve chvíli, kdy k nim chceme přistoupit. Její použití, stejně jako použití zmiňované třídy `Where` je vidět v ukázce 2.4.

2.3.4 Hodnocení

Výhody:

- Vyspělost a velké rozšíření knihovny – dá se očekávat, že neobsahuje žádné závažné chyby.

Nedostatky:

- Není specializovaná na Android, nemůže tak být zcela optimalizována pro použití na této platformě.
- Všechny metody pro vyhledávání s výjimkou dotazů zkonstruovaných pomocí třídy `QueryBuilder` vrací výsledek v podobě třídy `List`. To znamená, že všechny objekty musí být zkonstruovány před tím, než obdržíme výsledek. To je časově náročné a ne vždy je to potřeba.

- Použití reflexe – rychlostí nemůže konkurovat knihovnám, které generují zdrojový kód.
- Relativně složitá inicializace – nutnost vytvořit speciální třídu a každou tabulku vytvořit zvláštním příkazem.
- Při sestavování dotazu pomocí třídy `Where` musíme uvádět jména sloupečků, což je náchylné k chybám.
- Parametr porovnávací funkce třídy `Where` není omezen na datový typ porovnávaného sloupečku, ale může jím být jakýkoliv objekt.

```
1  public class DataBaseHelper extends OrmLiteSqliteOpenHelper {...}
2
3  // In onCreate of our Application class
4  DataBaseHelper.init(context);
5  ConnectionSource connectionSource =
    ↪  DatabaseHelper.getInstance().getConnectionSource();
6  TableUtils.createTableIfNotExists(connectionSource, User.class);
7
8  // When we need Dao to read/write to database
9  Dao<User, Long> userDao =
    ↪  DataBaseHelper.getInstance().getDao(User.class);
```

Ukázka kódu 2.2: Inicializace knihovny ORMLite

```
1  @DatabaseTable
2  public class User {
3      @DatabaseField(id = true)
4      private String name;
5      @DatabaseField
6      private int birthYear;
7      // Won't be persisted
8      private int tempUsageCount;
9
10     public User() {
11     }
12 }
```

Ukázka kódu 2.3: Třída nastavená pro použití s knihovnou ORMLite

```
1 QueryBuilder<User, Long> qb = userDao.queryBuilder();
2 Where<User, Long> w = qb.where();
3 w.and(w.eq("first_name", "Joe"),
4     w.or(w.gt("birth_year", 1970),
5         w.and(w.eq("birth_year", 1970), w.ge("birth_month", 10))));
6 qb.limit(20).offset(5);
7 ClosableIterator<User> iterator = qb.iterator();
```

Ukázka kódu 2.4: Dotaz do databáze v knihovně ORMLite

2.4 GreenDAO

GreenDAO [14] je dle statistik druhá nejrozšířenější knihovna, která ukládá data do relační databáze. Dle testů je navíc nejrychlejší (viz kapitola 5.2). Funguje na principu generování kódu, k čemuž ovšem nevyužívá anotační procesor, ale vlastní plugin pro build systém gradle.

2.4.1 Inicializace

Pro inicializaci není potřeba vytvářet žádné třídy, vše potřebné je vygenerováno knihovnou. Inicializace probíhá způsobem, který je vidět v ukázce 2.5, a bude se typicky nacházet v `onCreate` metodě naší třídy `Application`.

2.4.2 Nároky na model

Třídy, které chceme ukládat do databáze musíme označit anotací `@Entity`. Primární klíč může být tvořen jen jedním sloupečkem a musí být typu `long`, značí se anotací `@Id`. Jednotlivé proměnné nemusí být nijak označeny a budou do databáze ukládány automaticky. Tomu můžeme zabránit pomocí anotace `@Transient`. GreenDAO neumí k proměnným přistupovat přímo, nezávisle na jejich přístupnosti. Musíme pro ně tedy vytvořit gettery a settery, případně je knihovna vygeneruje za nás. GreenDAO nám také vygeneruje kostruktor se všemi proměnnými ukládanými do databáze, pokud to explicitně nezakážeme (potom ale musíme poskytnout konstruktor bezparametrický). Konstruktor i gettery a settery musí mít přístupnost alespoň *package-private*. Příklad takové třídy je v ukázce 2.6.

2.4.3 API pro čtení a zápis

Ke čtení a zapisování do databáze se využívají objekty typu `*Dao`, které knihovna vygeneruje pro každou tabulku. Získání instance třídy `UserDao` je vidět v ukázce 2.5.

Pro ukládání jsou k dispozici následující metody:

```
long insert(T entity);
long insertOrReplace(T entity);
void save(T entity);
```

Metoda `insert` uloží objekt do databáze, pokud by tím došlo k porušení integritního omezení, bude vyvolána výjimka. Metoda `insertOrReplace` naproti tomu při konfliktu nahradí data konfliktního řádku předaným objektem. Metoda `save` objekt vloží do databáze, pokud má proměnná reprezentující jeho primární klíč výchozí hodnotu (null nebo 0). V takovém případě zavolá metodu `insert`, v opečném případě metodu `update`. Výhodou je, že je to o něco rychlejší než použití metody `insertOrReplace`. Nevýhodou pak je, že pokud má objekt primární klíč sice nastavený, ale na hodnotu která v databázi ještě není uložená, neprovede se nic. Všechny uvedené metody jsou navíc k dispozici ve variantách pro uložení více objektů najednou, v takovém případě je to automaticky provedeno v transakci.

Pro upravování je k dispozici následující metoda:

```
void update(T entity);
```

Metoda `update` upraví řádek v tabulce podle odpovídajícího primárního klíče. Je k dispozici také varianta pro upravení více objektů najednou v transakci.

Pro mazání jsou k dispozici následující metody:

```
void deleteAll();
void delete(T entity);
void deleteByKey(K key);
```

Funkce všech tří metod je zřejmá. Metody `delete` a `deleteByKey` jsou opět k dispozici i ve variantách pro smazání více záznamů najednou, které probíhá v transakci. Mazat na základě specifického výběru nám umožňuje třída `QueryBuilder`, o té více později.

Pro čtení dat jsou k dispozici následující metody:

```
1 long count();
2 T load(K key);
3 List<T> loadAll();
4 List<T> queryRaw(String where, String... selectionArg);
```

Funkce všech metod je zřejmá.

Pro pokročilé dotazy slouží třída QueryBuilder. Její použití je vidět v ukázce 2.7. Výsledek takto vytvořeného dotazu můžeme získat ve více podobách. Jednak jako standardní List, ale také jako LazyList. To je vlastní třída knihovny, která z výsledku dotazu konstruuje objekty až v momentě, kdy k nim přistupujeme. Dotaz vytvořený pomocí třídy QueryBuilder můžeme také využít k získání počtu vybraných řádků, nebo k jejich smazání.

2.4.4 Hodnocení

Výhody:

- Rychlost (viz kapitola 5.2).
- Poměrně jednoduchá inicializace nevyžadující vytváření žádných tříd.

Nedostatky:

- Funkce update a delete nemají návratovou hodnotu a nemůžeme tedy zjistit výsledek operace. Tyto funkce by měly vracet počet upravených, respektive odstraněných řádků.
- Absence metody pro úpravu tabulky na základě určitých podmínek. Jediným způsobem jak tohoto můžeme docílit je načtení všech objektů z databáze, jejich požadovaná úprava a následné promítnutí změn zpět do databáze pomocí metody update.
- Metody loadAll a queryRaw vrací výsledek v podobě třídy List. To znamená, že všechny objekty musí být zkonstruovány před tím, než obdržíme výsledek. To je časově náročné a ne vždy je to potřeba.
- Způsob vytváření klauzule where při použití třídy QueryBuilder. Nemusíme sice na rozdíl od knihovny ORMLite uvádět jména sloupečků jako řetězce, zápis typu UserDao.Properties.FirstName je však celkem komplikovaný. Vnořené podmínky také nejsou příliš intuitivní.

- Parametr porovnávací funkce při vytváření klauzule where není omezen na datový typ porovnávaného sloupečku.
- Nemožnost přistupovat k proměnným přímo – nutné vytvoření getterů a setterů.

```
1  DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(this,
   ↪  "database");
2  SQLiteDatabase db = helper.getWritableDatabase();
3  DaoMaster daoMaster = new DaoMaster(db);
4  DaoSession daoSession = daoMaster.newSession();
5
6  UserDao userDao = daoSession.getUserDao();
```

Ukázka kódu 2.5: Inicializace databáze v knihovně GreenDAO

```
1  @Entity
2  public class User {
3      @Id
4      private Long id;
5      private String name;
6      @Transient
7      private int tempUsageCount;
8
9      @Generated(hash = 873297011)
10     public User(Long id, String name) {
11         this.id = id;
12         this.name = name;
13     }
14     @Generated(hash = 586692638)
15     public User() {
16     }
17
18     // Getters and setters
19 }
```

Ukázka kódu 2.6: Třída nastavená pro použití s knihovnou GreenDAO

```
1 QueryBuilder<User> qb = userDao.queryBuilder();
2 qb.where(UserDao.Properties.FirstName.eq("Joe"),
3     qb.or(UserDao.Properties.YearOfBirth.gt(1970),
4         qb.and(UserDao.Properties.YearOfBirth.eq(1970),
           ↪ UserDao.Properties.MonthOfBirth.ge(10))));
5 qb.limit(20).offset(5);
6 qb.orderAsc(UserDao.Properties.YearOfBirth);
7 List<User> users = qb.list();
```

Ukázka kódu 2.7: Dotaz do databáze v knihovně GreenDAO

2.5 DBFlow

DBFlow [15] funguje na principu generování kódu, k čemuž využívá anotační procesor.

2.5.1 Inicializace

Inicializaci provedeme zavoláním metody `FlowManager.init` v `onCreate` metodě naší třídy `Application`. Pokud chceme při inicializaci automaticky otevřít všechny databáze, můžeme k tomu využít `FlowConfig.Builder`. Oba způsoby jsou vidět v ukázce 2.8.

2.5.2 Nároky na model

Musíme vytvořit třídu reprezentující naši databázi, která bude označena anotací `@Database`, jejímiž parametry jsou jméno a verze databáze. Každá třída, kterou chceme do databáze ukládat musí být označena anotací `@Table`, jejímž parametrem je dříve vytvořená třída databáze. Tato třída může rozšiřovat třídu `BaseModel`, což nám později umožní volat metody jako `save` přímo na její instanci, není to ale nutné. DBFlow jako jediný umožňuje mít primární klíč tvořený více sloupečky, a to bez omezení datového typu. Pro označení sloupečku tvořícího primární klíče se používá anotace `@PrimaryKey`. Můžeme si určit, jestli mají být proměnné ve výchozím stavu ukládány, nebo ne. Knihovna umí k proměnným přistupovat jak přímo, tak pomocí getterů a setterů, jedno z toho musí mít přístupnost alespoň *package-private*. Vyžadován je bezparametrický konstruktor. Příklad takové třídy je v ukázce 2.9.

2.5.3 API pro čtení a zápis

Pro zapisování do databáze slouží objekty typu `ModelAdapter`. Některé metody také můžeme volat přímo na ukládaných objektech, pokud dědí ze třídy `BaseModel`. Ke čtení poté slouží statické metody třídy `SQLite`.

Pro ukládání jsou k dispozici následující metody:

```
long insert(TModel model);
boolean save(TModel model);
```

Metoda `insert` uloží objekt do databáze. Pokud ovšem jako primární klíč použijeme numerický datový typ, je jeho hodnota v objektu ignorována. Pokud tedy tuto metodu použijeme pro uložení objektu, který už má nastavený takový primární klíč, objekt bude uložen s úplně jiným primárním klíčem. Metoda `save` buď vloží nebo upraví existující záznam, podle hodnoty primárního klíče. Obě metody existují i ve variantě pro uložení více objektů zároveň, ne však v transakci.

Pro upravování je k dispozici následující metoda:

```
boolean update(TModel model);
```

Tato metoda upraví řádek v tabulce podle odpovídajícího primárního klíče. Je k dispozici také varianta pro upravení více objektů najednou, ne však v transakci.

Pro mazání je k dispozici následující metoda:

```
boolean delete(TModel model);
```

Metoda `delete` smaže řádek v tabulce podle odpovídajícího primárního klíče. Je k dispozici také varianta pro smazání více objektů najednou, ne však v transakci.

Pro čtení slouží třída `SQLite`. Její metody se snaží o podobnou syntaxy jakou má samotný jazyk SQL. Ačkoliv nabízí celkem velké možnosti, chybí ji například vnořování podmínek. Nejjednodušší způsob jak vytvořit podobný dotaz jako v předchozích dvou knihovnách bude tedy asi ten v ukázce 2.10. Výsledky můžeme získat jako standardní `List`, nebo jako `CursorResult`. Ten z výsledku dotazu konstruuje objekty až v momentě, kdy k nim přistupujeme. Dotaz vytvořený pomocí třídy `SQLite` můžeme také využít k získání počtu vybraných řádků, nebo k jejich smazání.

2.5.4 Hodnocení

Výhody:

- Jednoduchá inicializace.
- Primární klíč může být tvořen více sloupečky a není omezen na jeden datový typ.
- Nejjednodušší vytváření klauzule where ze všech porovnávaných knihoven.
- Parametr porovnávací funkce při vytváření klauzule where je omezen na datový typ porovnávaného sloupečku.

Nedostatky:

- Mezi knihovnami generující kód patří k nejpomalejším (viz kapitola 5.2).
- Absence metody pro smazání záznamu na základě primárního klíče. Pro tento účel musíme celý příkaz poměrně zdlouhavě vytvořit pomocí třídy `SQLite`.
- Nemožnost vnořovat podmínky v klauzuli where.
- Zbytečně složitý přístup k metodám pro zápis do databáze pokud naše třídy nedědí ze třídy `BaseModel`.

```
1   FlowManager.init(this);
2
3   FlowManager.init(
4       new FlowConfig.Builder(this)
5           .openDatabasesOnInit(true)
6           .build()
7   );
```

Ukázka kódu 2.8: Inicializace knihovny DBFlow

```java
@Database(name = "AppDatabase", version = 1)
public class AppDatabase {}

@Table(database = AppDatabase.class)
public class Automobile extends BaseModel {
  @PrimaryKey
  String vin;

  @Column
  String make;

  @Column
  String model;

  @Column
  int year;

}
```

Ukázka kódu 2.9: Třída nastavená pro použití s knihovnou DBFlow

```java
CursorResult<User> result = SQLite.select().from(User.class)
        .where(User_Table.name.eq("Joe"))
        .and(User_Table.birthYear.greaterThan(1970))
        .or(User_Table.name.eq("Joe"))
        .and(User_Table.birthYear.eq(1970))
        .and(User_Table.birthMonth.greaterThanOrEq(10))
        .limit(20)
        .offset(5)
        .queryResults();
```

Ukázka kódu 2.10: Dotaz do databáze v knihovně DBFlow

2.6 Joogar

Knihovna Joogar [2] vychází z knihovny SugarORM [16] a sloužila jako základ pro implementační část této práce. Funguje na principu reflexe.

2.6.1 Inicializace

Inicializace knihovny probíhá v `onCreate` metodě naší třídy `Application`. Zde pomocí třídy `JoogarDatabaseBuilder` vytvoříme požadované databáze. U každé databáze musíme určit verzi a třídy, které budou tvořit její tabulky. Pokud vytváříme více databází, je také nutné specifikovat jejich jména. Příklad inicializace můžeme vidět v ukázce 2.11.

2.6.2 Nároky na model

Knihovna využívá anotace. Třída, kterou chceme ukládat do databáze, musí být buď označena anotací `@Table`, nebo dědit ze třídy `JoogarRecord`. Při použití prvního způsobu musíme v naší třídě definovat proměnnou `id` typu `Long`, která bude sloužit jako primární klíč tabulky. Pokud se rozhodneme pro druhý způsob, proměnná `id` je již obsažena ve třídě `JoogarRecord`. Z toho plyne že primární klíč může být tvořen jen jedním sloupečkem, musí být typu `Long` a proměnná, která ho reprezentuje, se musí jmenovat `id`. Jednotlivé proměnné jsou do databáze ukládány automaticky, pokud některou do databáze ukládat nechceme, můžeme použít anotaci `@Ignore`. Hodnoty jsou čteny a nastavovány proměnným přímo, použití getterů a setterů není možné. Díky použití reflexe nejsou kladeny žádné nároky na přístupnout proměnných, mohou být i *private*. Vyžadován je bezparametrický konstruktor. Příklad takové třídy je v ukázce 2.12.

2.6.3 API pro čtení a zápis

Pro čtení a zápis dat do databáze slouží statické metody třídy `JoogarRecord`.

Pro ukládání je k dispozici jediná metoda:

```
1 long save(Object object);
```

Metoda uloží objekt do databáze. Pokud by vložením řádku došlo k porušení integritního omezení, bude starý záznam odtraněn a nahrazen novým. Metoda je také k dispozici ve variantě pro uložení více objektů najednou, nedělá to však v transakci.

Pro upravování není k dispozici žádná metoda. Uživatel tedy musí použít metodu `save`, případně metodu `executeQuery`, která umožňuje vykonat libovolný `SQL` příkaz.

Pro mazání jsou k dispozici následující metody:

```
1 boolean deleteById(Class<?> type, Long id);
2 boolean delete(Object object);
3 int deleteAll(Class<?> type);
4 int deleteAll(Class<?> type, String whereClause, String... whereArgs);
```

Všechny metody až na tu poslední odtraňují záznamy z databáze na základě rovnosti primárního klíče. Poslední metoda umožňuje specifikovat libovolnou podmínku.

Pro čtení dat jsou k dispozici následující metody:

```
1 long count(Class<?> type);
2 long count(Class<?> type, String whereClause, String[] whereArgs);
3 T findById(Class<T> type, Long id);
4 JoogarCursor<T> findById(Class<T> type, String[] ids);
5 JoogarCursor<T> findAll(Class<T> type);
6 JoogarCursor<T> find(Class<T> type, String whereClause, String[] whereArgs,
   ↪ String groupBy, String orderBy, String limit);
7 JoogarCursor<T> findWithQuery(Class<T> type, String query, String...
   ↪ arguments);
```

Funkce všech metod je zřejmá. Na rozdíl od všech předchozích knihoven neposkytuje Joogar API pro vytváření dotazavacího příkazu po částech, ani pro vytvoření klauzule where. Pro vytvoření podobného datazu jako u předchozích knihoven musíme použít metodu `find`, jak je vidět v ukázce 2.13. Všechny metody, které mohou vracet více objektů vrací třídu `JoogarCursor`, která z výsledku dotazu konstruuje objekty až v momentě, kdy k nim přistupujeme.

2.6.4 Hodnocení

Výhody:

- Jednoduchý přístup ke všem metodám pomocí třídy `JoogarRecord`.

Nedostatky:

- Knihovna je velmi pomalá (viz kapitola 5.2).
- Složitější inicializace: nutnost uvést všechny třídy které chceme ukládat.
- Nedostatečné možnosti ukládání: chybí metoda typu `insert`.
- Chybí jakákoliv metoda pro upravování.
- Metody pro vyhledávání nenabízí žádnou výhodu oproti standanrdnímu Android API, musíme sami napsat téměř celý SQL příkaz.

```
1  JoogarDatabaseBuilder builder = new JoogarDatabaseBuilder()
2          .setDomainClasses(User.class, Message.class)
3          .setName("Database.db")
4          .setVersion(1);
5
6  Joogar.initForAndroid(applicationContext).addDB(builder);
```

Ukázka kódu 2.11: Inicializace knihovny Joogar

```
1  @Table
2  public class User {
3      private Long id;
4      private int birthYear;
5      @Ignore
6      private int tempUsageCount;
7
8      public User() {
9      }
10 }
```

Ukázka kódu 2.12: Třída nastavená pro použití s knihovnou Joogar

```
1 JoogarCursor<User> result = JoogarRecord.find(
2         User.class,
3         "name = ? AND (birth_year > ? OR birth_year = ? AND
           ↪  birth_month >= ?)",
4         new String[]{"Joe", "1970", "1970", "10"},
5         null, null, "5, 10");
```

Ukázka kódu 2.13: Dotaz do databáze v knihovně Joogar

2.7 Squeaky

Krátce se zmíníme ještě o knihovně Squeaky [17]. Ta vychází z knihovny ORMLite, na rozdíl od ní ale využívá generování kódu. Rychlostí je tedy srovnatelná s knihovnami GreenDAO a DBFlow (viz kapitola 5.2), přičemž staví na pevných základech ORMLite. Bohužel generování kódu bylo využito pouze pro serializaci a deserializaci objektů. Pro čtení a ukládání dat do databáze používá stejné API jako ORMLite, v tomto ohledu tedy zmíněným knihovnám konkurovat nemůže.

Kapitola 3

Návrh

3.1 Požadavky

3.1.1 Funkční požadavky

- Knihovna bude podporovat všechny primitivní datové typy jazyka Java i jejich objektové varianty.
- Primární klíč bude moci být tvořen více sloupečky a nebude omezen datovým typem.
- Knihovna bude umět přistupovat k proměnným přímo i pomocí getterů a setterů.
- Při inicializaci nebude nutné specifikovat třídy, které mají tvořit tabulky databáze.
- Knihovna bude umožňovat ukládat objekty do databáze.
- Knihovna bude umožňovat mazat řádky z databáze na základě primárního klíče a podle podmínek klauzule where.
- Knihovna bude umožňovat číst celé řádky z databáze a automaticky je převádět do objektové podoby.
- Číst bude možné:
  - všechny záznamy tabulky,
  - záznam s konkrétním primárním klíčem,
  - záznamy splňující podmínky dané klauzulí where.

- Knihovna bude umožňovat sestavení klauzule where bez potřeby uvádět jména sloupečků. Jednotlivé podmínky bude možné spojovat operátory *and* a *or* a bude možné je i vnořovat.

3.1.2 Nefunkční požadavky

- Knihovna bude podporovat OS Android verze 3.0 (API level 11) a vyšší.
- Rychlost čtení a zápisu bude srovnatelná s ostatními knihovnami, které generují zdrojový kód.

3.2 Návrh API

3.2.1 Inicializace

Kód potřebný pro vytvoření databází a tabulek bude vygenerován. Inicializace tedy bude spočívat pouze v zavolání metody `Joogar.init`, které předáme aplikační *Context*.

3.2.2 Nároky na model

Bude využito stejných anotací, které obsahovala původní verze knihovny, ty budou jen mírně upraveny.

Třída ukládaná do databáze bude označena anotací `@Table`, která bude mít dva volitelné parametry – `name` a `database`. Parametr `name` bude určovat jméno tabulky, pokud nebude uveden, tabulka bude pojmenována stejně jako příslušná třída. Parametr `database` bude určovat jméno databáze, ve které bude tabulka vytvořena. Pokud nebude uveden, bude tabulka vytvořena ve výchozí databázi jménem *joogar*.

Každá instanční proměnná bude ve výchozím stavu tvořit sloupeček databáze. Pokud nebudeme chtít některou proměnnou ukládat, použijeme anotaci `@Ignore`. Proměnné, které budou označeny anotací `@PrimaryKey`, budou tvořit primární klíč příslušné tabulky. Každá tabulka musí obsahovat alespoň jednu takovou proměnnou. Proměnná může být dále označena anotacemi `@NonNull` a `@Unique`, které zařídí, že bude jejich sloupeček vytvořen s příslušným integritním omezením. K dispozici bude také anotace `@ColumnName`, pomocí jejíhož parametru bude možné specifikovat jméno sloupečku v databázi (jinak se bude sloupeček jmenovat stejně jako příslušná proměnná). Knihovna bude umět k proměnným přistupovat jak přímo, tak pomocí getterů a setterů. Přístup pomocí getterů a setterů bude upřednostňován. Proměnné ukládané do databáze, případně jejich gettery a settery, musí mít přístupnost alespoň *package-private*. Vyžadován bude bezparametrický konstruktor.

3.2.3 API pro čtení a zápis

Pro práci s databází budou stejně jako v předchozí verzi použity statické metody. Pro každou tabulku bude vygenerována třída `*Table`, která bude tyto metody obsahovat. Pro každou tabulku bude dále vygenerována třída `*Where`, která bude umožňovat sestavení klauzule where (viz kapitola Sestavení klauzule where).

3.2.3.1 Ukládání

Pro ukládání budou k dispozici dvě základní metody, `save` a `insert`. Jejich parametrem bude ukládaný objekt, vracet budou `id` vytvořeného řádku, případně hodnotu -1, pokud nebyl vytvořen nový řádek. Metody se liší způsobem řešení konfliktů. Pokud by vložením řádku došlo k porušení integritního omezení, metoda `save` konfliktní záznam odstraní, metoda `insert` vyvolá výjimku.

Pro úpravu bude k dispozici metoda `update`. Upravování bude probíhat na základě rovnosti primárních klíčů. Parametrem metody bude upravený objekt, návratová hodnota bude `true`, pokud v databázi existoval záznam s daným primárním klíčem a byl tedy upraven, v opačném případě `false`. Pro úpravu více řádků na základě nějaké podmínky bude možné využít metodu `update` s parametry `ContentValues` a `Where`.

Dále budou k dispozici metody `saveInTx`, `insertInTx` a `updateInTx`, pomocí kterých bude možné uložit, respektive upravit větší množství objektů v transakci.

3.2.3.2 Mazání

Pro mazání budou k dispozici základní metody `delete` a `deleteByPrimaryKey`. Obě budou mazat řádky z databáze na základě rovnosti primárního klíče. Parametrem první bude objekt, který chce uživatel odstranit. Parametrem druhé metody bude hodnota primárního klíče. Metoda `deleteByPrimaryKey` bude pojmenována podle toho, kolik sloupečků tvoří primární klíč – pokud bude primární klíč tvořen jediným sloupečkem, bude pojmenována podle tohoto sloupečku (například `deleteById`), pokud bude primární klíč tvořen více sloupečky, bude pojmenována `deleteByPrimaryKey`. Obě metody budou vracet hodnotu `true`, pokud v databázi existoval záznam s příslušným primárním klíčem a byl tedy odtraněn, v opačném případě `false`.

Dále bude k dispozici metoda `deleteInTx` pro smazání několika objektů současně v transakci, metoda `deleteAll` pro smazání všech záznamů z dané tabulky a metoda `delete(where)`, která smaže všechny záznamy vyhovující

podmínce dané parametrem `where`. Všechny tyto metody budou vracet počet smazaných řádků.

3.2.3.3 Čtení

Pro čtení z databáze budou k dispozici metody `countAll` a `count(Where)`, které vrací počet všech řádků v tabulce, respektive počet řádků, které vyhovují podmínce. Dále bude k dispozici metoda `findByPrimaryKey`, v případě primárního klíče tvořeného jediným sloupečkem pojmenovaná podle tohoto sloupečku. Ta vrací objekt s daným primárním klíčem, pokud se v databázi takový nachází, v opačném případě hodnotu `null`. Nejdůležitější je metoda `find`, která nám bude umožňovat sestavit dotaz po částech, jak je znázorněno v ukázce 3.1.

```
1 PersonTable.find()
2         .where(new PersonWhere().birthYear().eq(1990))
3         .orderBy(PersonTable.lastName, "ASC")
4         .orderBy(PersonTable.firstName, "ASC")
5         .limit(50).offset(10)
6         .find();
```

Ukázka kódu 3.1: Sestavení dotazu do databáze

3.2.3.4 Sestavení klauzule where

Pro sestavení klauzule where bude sloužit třída `*Where`, která bude vygenerována pro každou tabulku. Její použití bude vypadat tak, jak je znázorněno v ukázce 3.2. Metody logických operátorů budou přijímat parametry podle datového typu příslušného sloupečku. Jednotlivé podmínky bude možné kombinovat pomocí metod `and` a `or`, přičemž metoda `and` bude implicitní a nebude muset být uváděna. Vnořené podmínky budou umožněny metodami `and(Where)` a `or(Where)`.

```
1 PersonWhere where = new PersonWhere()
2         .name().in("Joe", "John", "Alice")
3         .and(new PersonWhere()
4                 .birthYear().between(1970, 1990)
5                 .or()
6                 .birthYear().eq(1969).birthMonth().gtEq(10));
```

Ukázka kódu 3.2: Sestavení klauzule where pomocí vygenerované třídy

Kapitola 4

Realizace

4.1 Nástroje potřebné pro realizaci

Pro realializaci aplikační rozhraní budeme potřebovat analyzovat zdrojový kód ještě před kompilací programu. K tomu použijeme anotace, které bude zpracovávat anotační procesor. Pomocí něj budeme také generovat nové zdrojové soubory, s čímž nám pomůže knihovna JavaPoet [18].

4.2 Anotace

Anotace nám umožňují přidávat metadata k elementům zdrojového kódu. Mají několik různých použití, například:

- Informace pro překladač – detekce chyb, potlačení varování.
- Zpracování v průběhu kompilace – generování kódu.
- Zpracování za běhu aplikace.

Příklad použití anotace můžeme vidět v ukázce 4.1. Anotace může obsahovat parametry, které se uvádí v závorkách za jejím jménem. Pokud anotace definuje jen jeden parametr pojmenovaný `value`, nemusíme jeho jméno uvádět, jak je vidět v ukázce 4.2.

[19, 20]

4.2.1 Vytváření vlastních anotací

Pro deklaraci anotace se používá klíčové slovo `@interface`. Jak může vypadat deklarace anotace je vidět v ukázce 4.3. Na deklaraci anotace můžeme rovněž

```
@Column(name = "jmeno", unique = true)
private String name;
```

Ukázka kódu 4.1: Příklad použití anotace s parametry

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

Ukázka kódu 4.2: Příklad použití anotace s jediným paremetrem *value*

aplikovat jiné anotace, kterým se říká metaanotace. Několik metaanotací je součástí Java API, pro nás budou důležité především následující dvě:

- `@Retention` – určuje, kde je daná anotace dostupná. Možné hodnoty jsou:
  - `RetentionPolicy.SOURCE` – dostupná pouze při kompilaci, kompilátor ji zahazuje
  - `RetentionPolicy.CLASS` – dostupná v bytekódu, za běhu aplikace být dostupná nemusí
  - `RetentionPolicy.RUNTIME` – dostupná i za běhu aplikace pomocí reflexe
- `@Target` – určuje, na které elementy můžeme anotaci aplikovat. Možné hodnoty jsou:
  - `ElementType.ANNOTATION_TYPE` – aplikujeme na anotace
  - `ElementType.CONSTRUCTOR` – aplikujeme na konstruktory
  - `ElementType.FIELD` – aplikujeme na třídní a instanční proměnné
  - `ElementType.LOCAL_VARIABLE` – aplikujeme na lokální proměnné
  - `ElementType.METHOD` – aplikujeme na metody
  - `ElementType.PACKAGE` – aplikujeme na balíčky
  - `ElementType.PARAMETER` – aplikujeme na parametry metod
  - `ElementType.TYPE` – aplikujeme na třídy, rozhraní, výčtové typy, anotace

Jak již bylo zmíňeno dříve, anotace může obsahovat parametry. Definice dvou parametrů `name` a `database` je vidět v ukázce 4.3. Každému parametru můžeme určit výchozí hodnotu pomocí klíčového slova `default`. Datový typ parametru nemůže být libovolný, je omezený na:

- primitivní datový typ,
- String,
- Class,
- výčtový typ,
- anotační typ,
- jednorozměrné pole kteréhokoliv z výše uvedených.

[21, 22, 23]

```
1 @Target(ElementType.TYPE)
2 @Retention(RetentionPolicy.SOURCE)
3 public @interface Table {
4     String name();
5     String database() default "joogar";
6 }
```

Ukázka kódu 4.3: Definice anotace

4.3 Anotační procesor

Anotační procesor je třída, která implementuje rozhraní `Processor` z balíčku `javax.annotation.processing`, případně dědí od třídy `AbstractProcessor` z téhož balíčku. Anotační procesor nám umožňuje zpracovávat anotace v době kompilace. Díky tomu můžeme generovat zdrojový kód, který bude následně zkompilován, případně analyzovat stávající kód a na jeho základě vytvářet chyby nebo varování.

Kostru anotačního procesoru můžeme vidět v ukázce 4.4. Každý anotační procesor musí mít veřejný konstruktor bez parametrů, který bude použit kompilátorem pro vytvoření instance procesoru. Musí také implementovat metody `init`, `getSupportedAnnotationTypes`, `getSupportedSourceVersion` a `process`, jak je vidět ve zmíněné ukázce.

Metoda `init` je volána po vytvoření instance anotačního procesoru a její parametr typu `ProcessingEnvironment` nám poskytuje metody pro vytváření

nových souborů, hlášení chybových zpráv a další užitečné funkce [24]. Metodou `getSupportedAnnotationTypes` určujeme, jaké anotace bude daný procesor zpracovávat. Metodou `getSupportedSourceVersion` určujeme nejnovější verzi jazyka Java, se kterou umí daný procesor pracovat. Pomocí metody `process` nám jsou předány elementy, které byly označeny některou z požadovaných anotací. [25]

Zpracování anotací probíhá v několika kolech. V každém z nich dostane procesor ke zpracování anotace nalezené v souborech vygenerovaných v kole předcházejícím. V prvním kole to budou anotace ze všech zdrojových souborů. Je garantováno, že pokud byl konkrétní procesor v některém kole zavolán, bude volán i ve všech následujích kolech, nehledě na to, jestli přibyly nové anotace, které by mohl zpracovat. [26]

4.4 JavaPoet

JavaPoet [18] je knihovna, která nám velmi zjednodušuje generování zdrojových souborů. Poskytuje API pro vytváření deklarací metod, třídních proměnných, tříd, anotací a dalších konstruktů programovacího jazyka. Automaticky za nás kód vhodně odřádkuje, zarovná, přidá středníky za příkazy a importuje všechny potřebné třídy. Použití knihovny je znázorněno v ukázce 4.5, vygenerovaný kód pak v ukázce 4.6.

```
1  @AutoService(Processor.class)
2  public class JoogarProcessor extends AbstractProcessor {
3
4      @Override
5      public synchronized void init(
6          ProcessingEnvironment processingEnv) {
7          super.init(processingEnv);
8          ...
9      }
10
11     @Override
12     public Set<String> getSupportedAnnotationTypes() {
13         return Collections.singleton(
14             Table.class.getCanonicalName()
15         );
16     }
17
18     @Override
19     public SourceVersion getSupportedSourceVersion() {
20         return SourceVersion.latestSupported();
21     }
22
23     @Override
24     public boolean process(
25         Set<? extends TypeElement> annotations,
26         RoundEnvironment roundEnv) {
27         ...
28     }
29 }
```

Ukázka kódu 4.4: Kostra anotačního procesoru

```java
ClassName androidLog = ClassName.get("android.util", "Log");

MethodSpec main = MethodSpec.methodBuilder("main")
        .addModifiers(Modifier.PUBLIC, Modifier.STATIC)
        .returns(void.class)
        .addParameter(String[].class, "args")
        .beginControlFlow("for ($T $L : $L)", String.class, "arg",
        ↪  "args")
        .addStatement("$T.d($S, $L)", androidLog, "HelloWorld",
        ↪  "arg")
        .endControlFlow()
        .build();

TypeSpec helloWorld = TypeSpec.classBuilder("HelloWorld")
        .addModifiers(Modifier.PUBLIC, Modifier.FINAL)
        .addMethod(main)
        .build();

JavaFile javaFile = JavaFile.builder("com.example.helloworld",
↪  helloWorld).build();
```

Ukázka kódu 4.5: Použití knihovny JavaPoet

```java
package com.example.helloworld;

import android.util.Log;
import java.lang.String;

public final class HelloWorld {
    public static void main(String[] args) {
        for (String arg : args) {
            Log.d("HelloWorld", arg);
        }
    }
}
```

Ukázka kódu 4.6: Kód vygenerovaný knihovnou JavaPoet z ukázky 4.5

4.5 Implementace

4.5.1 Rozdělení do modulů

Knihovna je rozdělena do tří modulů:

- **joogar** – vychází z původní verze knihovny a tvoří jakousi nadstavbu nad Android API
- **compiler** – obsahuje anotační procesor a jeho podpůrné třídy
- **annotations** – obsahuje vytvořené anotace

Rozdělení do několika modulů dává uživatelům knihovny možnost použít moduly *annotations* a *compiler* pouze při kompilaci a nezahrnovat je do zkompilované aplikace. Příklad, jak tohoto docílit pomocí build systému gradle, je v ukázce 4.7.

```
1  compile "net.skoumal.joogar2:joogar:version"
2  provided "net.skoumal.joogar2:joogar-annotations:version"
3  annotationProcessor "net.skoumal.joogar2:joogar-compiler:version"
```

Ukázka kódu 4.7: Instrukce pro build systém gradle

4.5.2 Serializace

Pro ukládání, upravování a mazání objektů z databáze používá naše knihovna připravené příkazy `SQLiteStatement` (viz kapitola 1.2.2). Objekty se pak serializují přímo do těchto příkazů způsobem, který můžeme vidět v ukázce 4.8.

4.5.3 Deserializace

Data přečtená z databáze mohou být v naší knihovně reprezentována dvěma třídami, `JoogarDatabaseResult` a `JoogarCursor`. První z nich obaluje standardní `Cursor` z Android API a poskytuje data v databázové podobě. Data z ní můžeme získat pomocí metod jako `getString(columnIndex)`, mezi jednotlivými výsledky (řádky tabulky) se posouváme pomocí metod `setPosition` a `next`. Naproti tomu třída `JoogarCursor` poskytuje data v podobě objektů, které jsou z databázových dat vytvářeny pomocí tříd implementujích funkční rozhraní `EntityBuilder`, jehož definici i implementaci můžeme vidět v ukázce 4.9. Pro každou tabulku takovou třídu automaticky vygenerujeme, uživatel si ale může vytvořit i své vlastní implementace.

```
statement = db.compileStatement(
        "INSERT INTO persons (name, birthYear) VALUES (?, ?)");

private void bindStatement(Person entity) {
    statement.clearBindings();
    if (entity.getName() != null) {
        statement.bindString(1, entity.getName());
    }
    statement.bindLong(2, entity.getBirthYear());
}
```

Ukázka kódu 4.8: Vytvoření a použití připraveného příkazu SQLiteStatement

```
public interface EntityBuilder<T> {
    T get(JoogarDatabaseResult result);
}

EntityBuilder<Person> personBuilder = new EntityBuilder<Person>() {
    @Override
    public Person get(JoogarDatabaseResult result) {
        Person person = new Person();
        person.setName(result.getString(0));
        person.setBirthYear(result.getInt(1));
        return person;
    }
};
```

Ukázka kódu 4.9: Funkční rozhraní EntityBuilder

4.5.4 Sestavení klauzule where

Pro sestavení klauzule where je důležitá abstraktní třída `WhereStatement` a třída `Comparator`. Třída `WhereStatement` obsahuje zejména metody `and`, `or`, `and(Where)` a `or(Where)`, které umožňují spojovat a vnořovat do sebe jednotlivé podmínky. Třída `Comparator` obsahuje metody reprezentující různé porovnávací operátory jazyka SQL. Třída je parametrizovaná datovým typem právě porovnávaného sloupečku, její metody tak přijmou pouze validní parametry. Konstruktor vyžaduje instanci třídy `WhereStatement`, kterou poté všechny metody vrací – díky tomu můžeme jednotlivé podmínky řetězit. Pro každou tabulku je dále vygenerována třída `*Where` (například `UserWhere`), která rozšiřuje třídu `WhereStatement`. Ta obsahuje metodu pro každý sloupe-

ček, jejímž voláním se zahajuje každá jednotlivá podmínka. Tyto metody vrací instanci třídy `Comparator`. Vytvoření klauzule where poté vypadá tak, jak už jsme viděli v ukázce 3.2 na straně 30 v kapitole Návrh. Jak je patrné, tímto způsobem je možné vytvořit i komplikované podmínky relativně jednoduchým způsobem.

Kapitola 5

Testování

Testování by mělo být součástí vývoje jakéhokoliv softwaru, tím spíše pokud se jedná o knihovnu. Správně napsané testy nám velmi usnadňují vývoj – po provedení změn stačí spustit testy a ty by měly odhalit většinu chyb, které jsme mohli změnou způsobit. V případě knihovny mohou být testy navíc dobrou ukázkou jejího použití.

5.1 Jednotkové a integrační testy

Jednotkové testy testují určitou komponentu, například funkci nebo třídu, v izolaci. V našem případě využijeme jednotkové testy například k otestování třídy `QueryBuilder`, která poskytuje funkce pro vytváření a validaci dotazů do databáze. Tato třída nezávisí na žádné jiné a jednoduše tak můžeme otestovat její správnou funkci voláním jejích metod se zvolenými argumenty a porovnáním návratové hodnoty s očekávaným výsledkem.

Integrační testy testují spolupráci komponent. V našem případě je využijeme zejména k otestování ukládání, mazání a čtení z databáze, vytváření databází a tabulek a nastavení integritních omezení.

5.1.1 Typy testů

Na androidu existují dva typy testů, *Local tests* a *Instrumented tests*.

Local tests jsou testy, které nepotřebují přístup k Android API. Jsou prováděny v lokální JVM a umisťují se do adresáře `app/src/test/java`.

Instrumented tests mají přístup k Android API. Jsou prováděny na fyzickém zařízení s OS Android, případně v emulátoru, a umisťují se do adresáře `app/src/androidTest/java`. [27]

5.1.2 Testovací framework

Pro jednotkové testy se na Androidu používá framework *JUnit* [28]. Android s ním spolupracuje pomocí knihovny *Testing Support Library* [29], která obsahuje mimo jiné třídu `AndroidJUnitRunner`. Ta nám umožňuje spouštět testy JUnit verze 3 a 4 na Android zařízeních. Zajišťuje nahrání testů do zařízení, jejich vykonání a zobrazení výsledků. Její součástí je také třída `InstrumentationRegistry`, která zajišťuje přístup k informacím o běhu testu. Můžeme z ní získat například `Context` naší aplikace. `AndroidJUnitRunner` nám dále umožňuje filtrovat testy na základě různých kritérií, k čemuž využívá následující anotace:

- `@RequiresDevice` – test se nespustí na emulátoru, jen na fyzickém zařízení
- `@SDKSupress(minSdkVersion = 18)` – test se nespustí na zařízení, které má API level menší než specifikuje parametr `minSdkVersion`
- `@SmallTest` – test by neměl trvat déle než 200 milisekund
- `@MediumTest` – test by neměl trvat déle než jednu sekundu
- `@LargeTest` – test trvající déle než jednu sekundu

Anotace `@SmallTest`, `@MediumTest` a `@LargeTest` však dobu běhu daného testu nevynucují, jak by se mohlo zdát. Slouží spíše k rozdělení testů do tří skupin, vývojář má následně možnost spustit jen některou z nich.

Samotný JUnit verze 4 využívá také řadu anotací, k nejdůležitějším patří:

- `@RunWith(AndroidJUnit4.class)` – určuje třídu, která bude testy vykonávat, v našem případě to bude `AndroidJUnit4.class` pro testy, které potřebují Android API a `JUnit4.class` pro ty ostatní
- `@Test` – touto anotací musí být označena každá metoda, která se má v rámci testování spouštět
- `@Before` – metoda označená touto anotací se provede před každou testovací metodou
- `@After` – metoda označená touto anotací se provede po každé testovací metodě

V ukázce 5.1 je příklad testovací třídy. Můžeme zde mimo jiné vidět použití metod označených anotacemi `@Before` a `@After`, které zajišťují vytvoření stejného počátečního prostředí pro každý test. Dále je zde použita třída `InstrumentationRegistry` pro získání *Contextu*, který potřebujeme k inicializaci naší knihovny.

```
1  @RunWith(AndroidJUnit4.class)
2  public abstract class TestClass {
3
4      @Before
5      public void before() {
6          Context context = InstrumentationRegistry.getTargetContext();
7          Joogar.init(context);
8      }
9
10     @Test
11     public void standardTest() {...}
12
13     @Test
14     @SdkSuppress(minSdkVersion = Build.VERSION_CODES.HONEYCOMB)
15     public void testWithFilter() {...}
16
17     @After
18     public void after() {
19         List<JoogarDatabase> databases =
           ↪  Joogar.getInstance().getDatabases();
20
21         for (JoogarDatabase database : databases) {
22             database.close();
23             database.getPath().delete();
24         }
25
26         Joogar.getInstance().close();
27     }
28 }
```

Ukázka kódu 5.1: JUnit test, který poběží na zařízení s OS Android

5.2 Výkonové testy

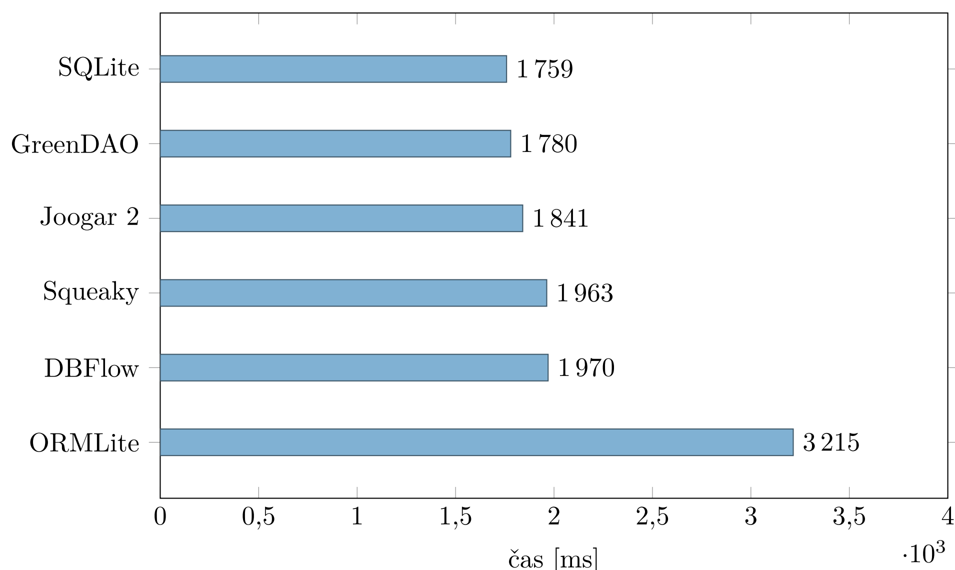
Pro porovnání výkonnosti jsme využili projekt *Android ORM benchmark updated* [30], který porovnává rychlosti čtení a zapisování několika ORM knihoven. K porovnávaným knihovnám navíc přidává maximálně optimalizované řešení

využívající pouze Android API. Námi byla přidána původní verze knihovny Joogar a verze vytvořená v této práci. Testy byly vykonávány na zařízení LG Nexus 5X s OS Android verze 7.1.2 (API level 25).

5.2.1 Test rychlosti zápisu

Zápis probíhal v jediné transakci, zapsáno bylo 2 000 objektů typu `User` a 20 000 objektů typu `Message`. Každý objekt typu `User` obsahoval primární klíč typu `long` a dvě proměnné typu `String`, které obsahovaly řetězce o délce deseti znaků. Objekty typu `Message` obsahovaly primární klíč typu `long`, jednu proměnnou typu `String` s řetězcem délky 100 znaků a dále čtyři proměnné typu `long`, jednu proměnnou typu `double` a jednu proměnnou typu `int`, které byly naplněny náhodnými hodnotami.

Test proběhl celkem třikrát, výsledná hodnota je průměrem těchto tří měření. Výsledky jsou zaznamenány v grafu 5.1. V grafu není uveden výsledek původní verze knihovny Joogar. Jeho hodnota byla 11 954 milisekund, což více než třikrát převyšuje jinak nejvyšší hodnotu v testu. Jejím uvedením by tak nebyl dostatečně patrný rozdíl mezi ostatními knihovnami.



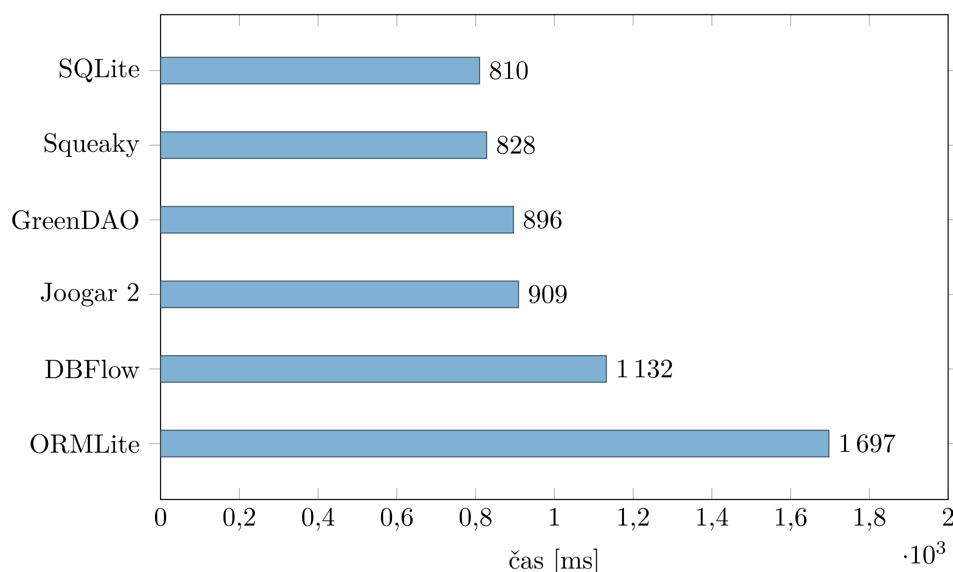
Obrázek 5.1: Srovnání rychlostí zápisu

5.2.2 Test rychlosti čtení

Čteno bylo všech 20 000 záznamů typu `Message` zapsaných v přechozím testu, započítaval se i čas konstrukce všech objektů.

Výsledná hodnota byla opět určena jako průměr tří měření. Výsledky jsou zaznamenány v grafu 5.2.

V grafu opět není uvedena původní verze knihovny Joogar, její výsledek v tomto testu byl 5 660 milisekund.



Obrázek 5.2: Srovnání rychlostí čtení

5.2.3 Závěr

Není překvapením, že v obou testech s mírným náskokem zvítězilo řešení, které přímo využívá Android API (v grafech uvedeno jako *SQLite*). Všechna ostatní řešení totiž musí ve výsledku použít stejný nebo velmi podobný způsob řešení, navíc však přidají nějakou režii.

Mezi knihovnami, které využívají generování kódu (GreenDAO, Joogar 2, Squeaky, DBFlow), jsou s výjimkou DBFlow jen nepatrné rozdíly. Knihovna DBFlow je z této skupiny knihoven v obou testech nejpomalejší.

Na posledních místech v obou testech skončily s výrazným odstupem knihovny fungující na principu reflexe. Předposlední místo patří knihovně ORMLite, poslední pak původní verzi knihovny Joogar.

Závěr

Cílem práce bylo rozšířit aplikační rozhraní Android open-source knihovny *Joogar* mapující relační databázi na objektový model.

Práce se nejprve v kapitole *Knihovny pro objektově-relační mapování* zabývala dvěma způsoby, kterými se dá taková knihovna implementovat (reflexe a generování kódu) a porovnávala existující knihovny, které jsou jedním z těchto způsobů implementovány. Na základě požadavků a analýzy existujících řešení bylo v kapitole *Návrh* navrhnuto aplikační rozhraní, které eliminuje většinu nedostatků existujících knihoven. To bylo následně implementováno s využitím anotačního procesoru pro generování zdrojového kódu, o čemž pojednává kapitola *Realizace*. Byly provedeny výkonové testy, které prokázaly, že výkon upravené knihovny *Joogar* je srovnatelný s nejrychlejšími existujícími knihovnami a je tak výrazně lepší než výkon původní verze. O způsobu testování výkonnosti a jeho výsledcích pojednává kapitola *Testování*.

Možnosti rozšíření

Do budoucna je v plánu rozšířit knihovnu o podporu více datových typů. Budoucí verze knihovny by také měly umožnit vytváření vztahů mezi jednotlivými tabulkami. Zajímavé by bylo přímo do knihovny zakomponovat možnost asynchronního zpracování s využitím knihovny *RxJava*.

Literatura

[1] AppTornado GmbH: Database libraries. [online], 2017, [cit. 2017-05-03]. Dostupné z: http://www.appbrain.com/stats/libraries/tag/database/database-libraries

[2] Skoumal, V.; Kiss, M.; Vaďura, D.: Joogar. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://github.com/skoumalcz/joogar

[3] Hipp, Wyrick & Company, Inc.: Datatypes In SQLite Version 3. [online], [cit. 2017-04-17]. Dostupné z: https://sqlite.org/datatype3.html

[4] Hipp, Wyrick & Company, Inc.: About SQLite. [online], [cit. 2017-04-17]. Dostupné z: https://sqlite.org/about.html

[5] Hipp, Wyrick & Company, Inc.: Distinctive Features Of SQLite. [online], [cit. 2017-04-17]. Dostupné z: http://www.sqlite.org/different.html

[6] Google Inc.: SQLiteOpenHelper Documentation. [online], [cit. 2017-04-17]. Dostupné z: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html

[7] Google Inc.: SQLiteDatabase Documentation. [online], [cit. 2017-04-17]. Dostupné z: https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase.html

[8] Google Inc.: ContentValues Documentation. [online], [cit. 2017-04-17]. Dostupné z: https://developer.android.com/reference/android/content/ContentValues.html

[9] Google Inc.: Cursor Documentation. [online], [cit. 2017-04-17]. Dostupné z: https://developer.android.com/reference/android/database/Cursor.html

[10] Vogel, L.: Android SQLite database and content provider - Tutorial: Cursor. [online], 2016, [cit. 2017-04-17]. Dostupné z: http://www.vogella.com/tutorials/AndroidSQLite/article.html#cursor

[11] Oracle Corporation: The Reflection API. [online], 2015, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/tutorial/reflect/

[12] Kučera, F.: Přepisujeme soukromé proměnné v Javě pomocí reflexe. [online], 2015, [cit. 2017-05-03]. Dostupné z: https://blog.frantovo.cz/c/348/P%C5%99episujeme%20soukrom%C3%A9%20prom%C4%9Bnn%C3%A9%20v%C2%A0Jav%C4%9B%20pomoc%C3%AD%20reflexe

[13] Watson, G.: OrmLite - Lightweight Object Relational Mapping (ORM) Java Package. [online], [cit. 2017-04-18]. Dostupné z: http://ormlite.com

[14] Greenrobot; aj.: GreenDAO. [online], 2017, [cit. 2017-04-10]. Dostupné z: http://greenrobot.org/greendao/

[15] Grosner, A.; aj.: DBFlow. [online], 2017, [cit. 2017-04-10]. Dostupné z: https://github.com/Raizlabs/DBFlow

[16] Narayan, S.: Sugar ORM. [online], [cit. 2017-04-18]. Dostupné z: http://satyan.github.io/sugar/index.html

[17] Galligan, K.: Squeaky - APT version of ORMLite. [online], [cit. 2017-04-18]. Dostupné z: https://github.com/touchlab/Squeaky

[18] Wharton, J.; Wilson, J.; aj.: Javapoet. [online], 2017, [cit. 2017-04-10]. Dostupné z: https://github.com/square/javapoet

[19] Oracle Corporation: Lesson: Annotations. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations/index.html

[20] Oracle Corporation: Annotations Basics. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations/basics.html

[21] Oracle Corporation: Declaring an Annotation Type. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations/declaring.html

[22] Oracle Corporation: Predefined Annotation Types. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html

[23] Oracle Corporation: Annotation Type Elements. [online], 2015, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html#jls-9.6.1

[24] Oracle Corporation: Interface ProcessingEnvironment documentation. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/ProcessingEnvironment.html

[25] Oracle Corporation: Interface Processor documentation. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html

[26] Oracle Corporation: Interface Processor documentation. [online], 2016, [cit. 2017-04-10]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/annotation/processing/Processor.html

[27] Google Inc.: Getting Started with Testing. [online], [cit. 2017-04-25]. Dostupné z: https://developer.android.com/training/testing/start/index.html

[28] Gamma, E.; Beck, K.; aj.: JUnit. [online], [cit. 2017-04-25]. Dostupné z: http://junit.org/junit4/

[29] Google Inc.: Testing Support Library. [online], [cit. 2017-04-25]. Dostupné z: https://developer.android.com/topic/libraries/testing-support-library/index.html

[30] Galligan, K.; aj.: Android ORM benchmark updated. [online], 2017, [cit. 2017-05-03]. Dostupné z: https://github.com/touchlab/android-orm-benchmark-updated

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface.

JVM Java Virtual Machine.

ORM objektově-relační mapování.

OS operační systém.

Příloha B

Obsah přiloženého média

```
/
├── apk
│   └── benchmark.apk ........................ spustitelná forma benchmarku
├── src
│   ├── android-orm-benchmark ................ zdrojové kódy benchmarku
│   ├── joogar ............................... zdrojové kódy implementace
│   └── thesis ............................... zdrojová forma práce ve formátu LaTeX
├── BP_Bina_Jan_2017.pdf ..................... text práce ve formátu PDF
└── readme.txt ............................... stručný popis obsahu média
```