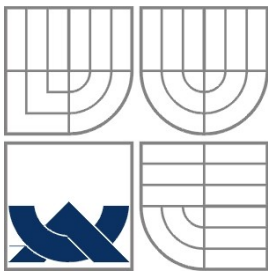


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV AUTOMATIZACE A INFORMATIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

VERZOVÁNÍ DATABÁZÍ
DATABASES VERSIONING

DIPLOMOVÁ PRÁCE
DIPLOMA THESIS

AUTOR PRÁCE **BC. PETR JINDRA**
AUTHOR

VEDOUCÍ PRÁCE **ING. JAN ROUPEC, PH.D.**
SUPERVISOR

BRNO 2012

ZADÁNÍ ZÁVĚREČNÉ PRÁCE

(na místo tohoto listu všijte originál a nebo kopii zadání Vaš práce)

ABSTRAKT

Tato diplomová práce se zabývá problematikou verzování databází. V prvních kapitolách jsou stručně popsány největší problémy, jež bude nutné vyřešit. Druhá část se týká popisu databázových systémů a úvodu do jazyka SQL. Poslední oddíl slouží jako dokumentace vývoje programu, který nastíněné problémy řeší.

ABSTRACT

This diploma thesis is about versioning of databases. Firsts chapters are shortly describing the biggest problems what will need to solve. Thr second part refers to choosen database systems and introcution to SQL language. The last unit describes the development of program solving sketched problems.

KLÍČOVÁ SLOVA

Databáze, verzování, C++, SQL, XML

KEYWORDS

Databases, versioning, C++, SQL, XML

PROHLÁŠENÍ O ORIGINALITĚ

Já, Petr Jindra, prohlašuji, že jsem diplomovou práci vypracoval samostatně a že jsem uvedl všechny použité prameny a literaturu.

V Brně dne 24. 5. 2013 .....

BIBLIOGRAFICKÁ CITACE

JINDRA, P. Verzování databází. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2013. 89 s. Vedoucí diplomové práce Ing. Jan Roupec, Ph.D..

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat zejména svému vedoucímu práce, Ing. Janu Roupcovi, Ph.D., který mi poskytl cenné rady při řešení problémů a své přítelkyni Hance za korekturu a trpělivost v období, kdy program nefungoval tak jak měl.

Obsah:

Zadání závěrečné práce....3
Abstrakt....5
Prohlášení o originalitě....7
Poděkování....9
1 Úvod....15
2 Rozbor problematiky....17
3 Stávající metody přenosu....19
3.1 Automatizované nástroje....20
3.2 Shrnutí....20
4 Jazyk SQL....21
4.1 Příkazy manipulující se strukturou....21
4.1.1 Tvorba databází a úprava....21
4.1.2 Tvorba tabulek a úprava....22
4.1.3 Tvorba pohledů a úprava....23
4.2 Příkazy pro manipulaci s daty....24
4.2.1 Vkládání dat do tabulek....24
4.2.2 Vyhledávání dat v tabulkách....24
4.2.3 Úprava záznamů....24
4.2.4 Mazání záznamů....25
4.3 Příkazy definující pomocné struktury....25
4.3.1 Funkce a procedury....25
4.3.2 Triggery....25
4.4 Ostatní příkazy....26
5 Přehled vybraných databázových systémů....27
5.1 MySQL....27
5.2 PostgreSQL....27
5.3 Oracle....27
5.4 Microsoft SQL Server....28
5.5 SQLite....28
6 Zpracování struktury databáze....29
6.1 Návrh modelu DKAZ....30
6.2 Graf....31
6.3 Struktura automatu a mechanismus přechodu....35
6.3.1 Pomocné třídy automatu....36
6.3.2 Mechanismus překlápění stavů a generování dat....38
6.4 Konstrukce automatu z externího zdroje....40
6.4.1 Prvek FA....40
6.4.2 Prvek MACHINE....40
6.4.3 Prvek STATE....41
6.4.4 Prvek PROCESSOR....41
6.4.5 Prvek PARAM....41
6.4.6 Prvek STATEAGREGATED....41
6.4.7 Prvek ASSEMBLER....41
6.4.8 Prvek TRANSITION....41
6.4.9 Prvek LOADER....41
6.4.10 Prvek FILTER....41
6.4.11 Prvek TEST....42
6.4.12 Zpracování souboru definic....42
7 Popis databáze uvnitř programu....43
7.1 Třídy popisu....43
7.1.1 Struktura PARAMETER....44
7.1.2 Třída SQLBase....44

7.1.3 Třída Database....45
7.1.4 Třída Table....45
7.1.5 Třída View....46
7.1.6 Třída Column....46
7.1.7 Třída Index....46
7.1.8 Třída Check....47
7.1.9 Třída Trigger....47
7.1.10 Třída Procedure....47
7.1.11 Struktura PROC_PARAM....48
7.1.12 Třída Function....48
7.1.13 Třída ForeignKey....48
7.1.14 Třída LoadedData....48
7.2 Implementace modelu....48
7.2.1 Reprezentace závislostí v paměti....49
7.2.2 Provázání grafu s popisem databáze....50
7.3 Reprezentace struktury v souborech....51
7.3.1 Prvek DATABASE....51
7.3.2 Prvek TABLE....52
7.3.3 Prvek VIEW....52
7.3.4 Prvek FOREIGNKEY....52
7.3.5 Prvek COLUMN....52
7.3.6 Prvek KEY....53
7.3.7 Prvek CHECK....53
7.3.8 Prvek PROCEDURE....53
7.3.9 Prvek FUNCTION....53
7.3.10 Prvek PROCPARAM....53
7.3.11 Prvek TRIGGER....53
7.3.12 Prvek TARCOLUMN....53
7.3.13 Prvek CODE....54
7.3.14 Prvek REFCOLUMN....54
8 Generování reprezentace databáze....55
8.1 Abstraktní adaptér....55
8.2 MySQL....56
8.3 XML Adaptér....58
9 Zjištění a uchování změn....61
9.1 Porovnání databází....61
9.2 Ukládání získaných dat....61
9.2.1 Konfigurace....62
9.2.2 Třída delta balíčku....62
9.2.3 Větvení změn....63
9.2.4 Formát přenosového souboru....64
9.2.5 Tvorba řetězu změn....64
9.3 Aplikace řetězu změn....66
9.3.1 Lokální adaptér....66
9.4 Slučování větví....67
9.4.1 Konflikty....69
10 Uživatelské rozhraní....75
10.1 Create conf....75
10.2 Create diff....76
10.3 Show conf....78
10.4 Export db....78
10.5 Merge....79
10.6 Help....81
11 Závěr....83

Seznam použité literatury.....85

1 ÚVOD

Při týmové spolupráci více programátorů na jednom projektu není v dnešní době problém udržovat zdrojové kódy u všech zúčastněných stále aktuální a jejich změny distribuovat mezi ostatní. K tomuto účelu je možné použít například systémy Git, CVS, SVN, které poskytují vývojářům širokou paletu funkcí využitelných při týmovém vývoji aplikace. Díky těmto aplikacím mohou být od sebe vzdálení stovky kilometrů a mít neustále k dispozici nejnovější změny, které provedli ostatní členové týmu.

Pokud projekt využívá databází, jsou možnosti přenosu změn mezi vývojáři mnohem omezenější. Soubory databází se zpravidla nacházejí v binárních souborech a verzovací systémy nejsou schopny v nich rozeznat změněná data.

Stávající metody, které lze použít na přenos databází, jsou často neefektivní a poskytují velký prostor k chybě. Současný stav na poli běžně dostupných nástrojů pro verzování a migraci je popsán v kapitole 3.

Situaci si klade za cíl změnit tato diplomová práce. Jejím výstupem by měl být program s možností pracovat nad více databázovými systémy. Měl by poskytovat funkce, které usnadní správu změn provedených ve struktuře databáze, a možnost jednoduše přenášet změny k ostatním členům týmu.

Vývojem aplikace se zabývá druhá část práce, začínající kapitolou 6.

Před vlastní tvorbou programu je nutné provést rozbor, který zjistí, v čem spočívá problematika automatického verzování, a najít možné komplikace, jež by mohly ohrozit vývoj projektu. Problematika je analyzována v následující kapitole.

Dále je vhodné zhodnotit stávající možnosti, které naplňují stejnou nebo podobnou funkci. Tento krok poskytne rozhled nad situací na trhu a inspiraci pro vývoj nového programu.

2 ROZBOR PROBLEMATIKY

Na rozdíl od verzování souborů není možné pro uchovávání změn v databázích použít klasické systémy jako například Git. Ty totiž porovnávají data ve změněných souborech a zapisují si jejich odlišnosti. Problém nastává, když je soubor binární. V tuto chvíli nelze vyhledat skutečně změněná data a soubor se zpravidla zapisuje do repozitáře celý znovu.

Protože ale drtivá většina databází používá pro zápis dat právě binární formát, je nutné přijít s jinou formou hledání změn.

Díky předběžnému zkoumání problematiky bylo zjištěno, že každý z prověřovaných databázových systémů poskytuje úplně jinou metodu pro zjištění struktury databáze. Systémy, které byly podrobeny průzkumu jsou:

- MySQL
- PostgreSQL
- Sqlite

Nejjednodušší kontrolu změn patrně umožňuje MySQL. Tento databázový systém poskytuje „read-only“ databázi `information_schema`, kde jsou ve virtuálních tabulkách uloženy všechny informace o strukturách, k nimž má daný uživatel přístup. Díky tomu se dá odhadnout, že implementovat spolupráci s MySQL bude s největší pravděpodobností nejjednodušší.

Naproti tomu PostgreSQL nic takového neposkytuje a přístup ke struktuře umožňuje přes utilitu `pg_dump` dodávanou společně s distribucí databázového software. Výstupem této utility je SQL kód, který provede vytvoření databázových struktur.

Knihovna Sqlite je kompromisem mezi oběma zmíněnými systémy. Obsahuje přímo příkazy pro zjišťování parametrů definovaných databázových struktur. Ty jsou ukládány jako SQL kód, kterým byly vytvořeny.

Jak je patrné z výše napsaného textu, prvním úskalím bude nutnost zpracovat textové příkazy v jazyce SQL. Načítání SQL kódu se věnuje kapitola 6. Dalším velkým problémem pak bude slučování vývojových větví a aplikace změn provedených kolegou. Při tomto procesu se může stát, že v obou větvích došlo ke změnám ve stejném prvků a bude potřeba navrhnout způsob, jak tyto situace řešit.

Další potíže mohou nastat při některých méně náročných situacích:

- porovnávání změn mezi uloženou verzí databáze a skutečnou situací na serveru
- uchování změn na disku
- správa změnových balíčků
- rekonstrukce struktury databáze z dříve uložených dat

Pokud se povede všechny výše uvedené problémy vyřešit, vznikne program použitelný jak pro výukové účely, tak i pro verzování databáze u skutečných projektů.

3 STÁVAJÍCÍ METODY PŘENOSU

Problematika verzování a distribuce (migrace) databázových modelů je v současnosti řešena zpravidla migračními nástroji, kde vývojáři ručně píší kód provádějící změny. U tohoto přístupu hrozí ovšem velká šance, že vznikne chyba, kterou autor změny přehlédne. Jedná se o různé překlepy ve jménu jednoho sloupce nebo tabulky, špatně definované indexy, cizí klíče a podobně. Tato nedopatření se mohou projevit okamžitě, například vyvolání chyby na straně serveru během provádění aktualizačních skriptů, nebo až za nějaký čas, když se začne daná část databáze používat. V případě chybně definovaného indexu se problém hledá velmi obtížně. Projeví se totiž až s používáním většího množství dat tím, že některé operace trvají déle než s indexem (vyhledávání) a některé naopak rychleji (vkládání a úprava uložených dat).

Stejně velkou, ne-li větší, nevýhodou je pak časová náročnost psaní takového kódu. Tomu se lze částečně vyhnout psaním změn přímo do migrací, ale tento přístup je značně nepřehledný, zvláště pokud dochází k neočekávaným změnám (nové požadavky zadavatele a podobně).

Příkladem takového nástroje je například Ruckusing (http://code.google.com/p/ruckusing/), který slouží k verzování databází psaných v PHP (jedná se o PHP port, kde původní utilita byla pro Ruby on Rails). Práce s touto aplikací probíhá následovně:

- založíme migraci s vytvořením tabulky `php generate.php create_posts_table`
- v souboru createPostsTable.php napíšeme těla metod `up` (povýšení verze) a `down` (revert)
- zavoláme migraci `php main.php db:migrate`

  Těla metod `up()` a `down()` mohou vypadat například jako v příkladu 1.

```
<?php
class CreatePostsTable extends Ruckusing_BaseMigration {
  public function up() {
    $t = $this->create_table("posts");
    $t->column("subject", "string");
    $t->column("body", "string");
    $t->column("created_at", "datetime", array('null' => false));
    $t->column("author_id", "integer");
    $t->finish();

    $this->add_index("posts", "author_id");
  }//up()

  public function down() {
    $this->drop_table("posts");
  }//down()
}
?>
```

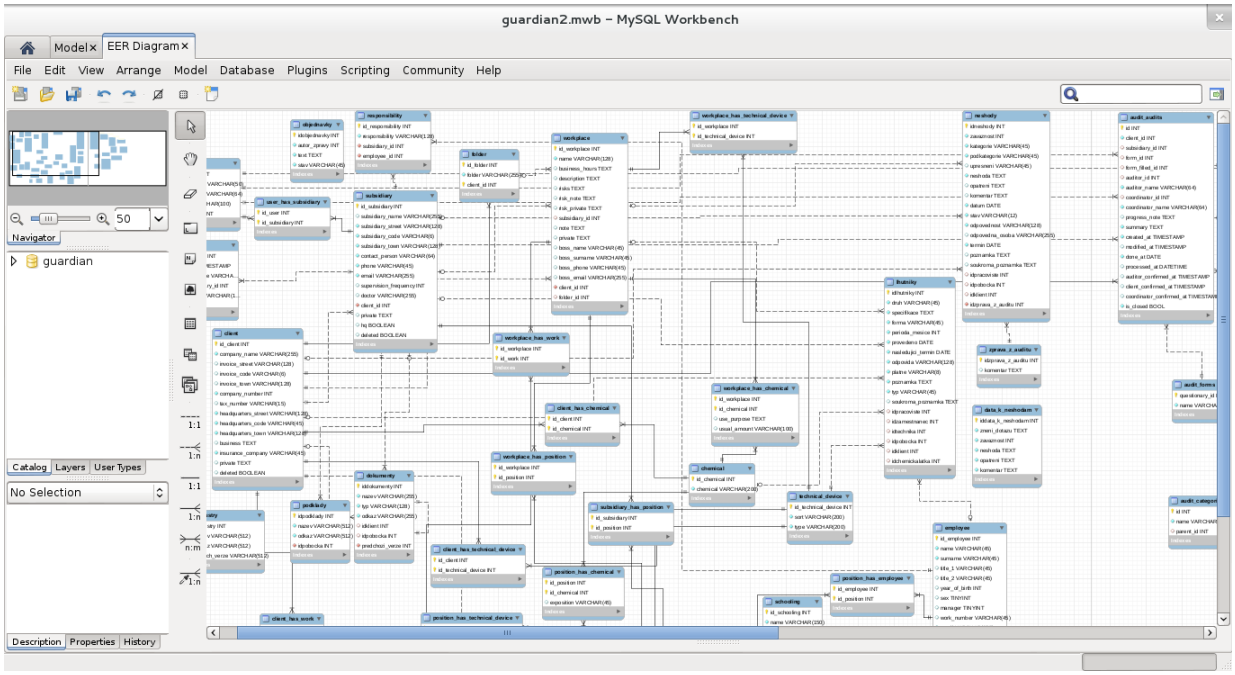
Příklad 1: Třída migrace databáze [1]

Jak je patrné z příkladu, při práci na větším projektu, který pracuje se stovkami tabulek, je tato metoda značně neefektivní, pracná a náchylná ke vzniku chyb. Asi jedinou výhodou tohoto přístupu je snadná správa verzí pomocí verzovacího systému (například Git, SVN atd.), protože každá akce je uložena ve vlastním souboru.

3.1 Automatizované nástroje

Kromě popsané manuální metody existují i nástroje schopné provést migraci automaticky. Většinou se ovšem jedná o programy spolupracující pouze s jedním databázovým systémem, a nelze je tedy použít univerzálně. Navíc jsou obvykle určeny pouze k migracím, a nikoliv ke sledování ukládání jednotlivých změn. Po provedení změny tedy není možný návrat k původnímu stavu schématu.

Klasickým příkladem takového programu je MySQL Workbench.



Obrázek 1: MySQL Workbench

Jedná se o grafický nástroj pro návrh tabulek a relací mezi nimi. Umožňuje přenést přenést celou strukturu po dokončení návrhu na MySQL server nebo jí získat z databáze pro budoucí úpravy. Program má ovšem několik značných nevýhod:

- nestabilita – při některých operacích program „spadne“ (zjištěno vlastní zkušeností)
- nemožnost ukládat si jednotlivé stavy schématu – po provedení změn tedy není možné se vrátit zpět přímo pomocí některé funkce tohoto programu
- nepředvídatelné generování SQL kódu pro provedení změn v tabulkách na serveru – při testování se několikrát stalo, že výsledný kód několikrát opakoval přidání a odebrání jednoho a toho samého sloupce nebo klíče v jediném SQL dotazu

Více možností než MySQL Workbench poskytuje aplikace Power Architect. Dokáže pracovat s nejrůznějšími typy databází, nabízí nástroje na profilování a podporuje práci s více databázovými servery i systémy v jednom projektu. Tento projekt je distribuován pod licencí GPL, a je tedy volně dostupný včetně zdrojových kódů.

3.2 Shrnutí

Tato diplomová práce se pokusí výše popsané problémy vyřešit, co nejvíce zautomatizovat proces verzování a migrací a zjednodušit výměnu změn mezi vývojáři. Cílem je dosažení časové úspory a zjednodušení práce vývojářů, kteří se již nebudou muset starat přímo o kód migrací, ale pouze o vlastní databáze.

Zamýšlená funkce verzování navíc vývojářům umožní předávat jednotlivé provedené změny místo celé databáze.

4 JAZYK SQL

Dotazovací jazyk SQL (Structured Query Language) je momentálně definován normou SQL:1999, na které spolupracovaly organizace ANSI a ISO [2]. Tento jazyk implementují s menšími či většími úpravami prakticky všechny relační databázové systémy pro komunikaci s okolím.

Příkazy tohoto jazyka lze rozdělit do několika hlavních kategorií:

- příkazy manipulující se strukturou
- příkazy manipulující s daty
- ostatní příkazy

Struktura příkazových vět je poměrně volná. Díky tomu je možné sestavit dva odlišné kódy, které provádějí naprosto totožný úkol. Jako ukázka může posloužit kód pro vytvoření indexu. Sekvence příkazů:

```
ALTER TABLE `tabulka` ADD INDEX `nejakyIndex` (`nejaky_sloupec`)
```

dělá stejnou věc jako:

```
CREATE INDEX `nejakyIndex` ON `tabulka` (`nejaky_sloupec`)
```

V tabulce `tabulka` se tedy vytvoří index `nejakyIndex` nad sloupcem `nejakySloupec`. Tato volnost vede k příjemnějšímu psaní kódu pro jednotlivé programátory (každý používá syntaxi, která mu vyhovuje), ale při spolupráci více lidí na jedné databázi může dojít vlivem špatné domluvy k různým nedorozuměním nebo k nesprávnému pochopení programu ostatními.

Kromě příkazů definují databáze také spoustu funkcí pro práci s daty, ty jsou ale zpravidla omezené na jednotlivé platformy, proto zde o nich nebude řeč. Tato kapitola tedy uvede jen hrubý přehled příkazů a jejich funkcí a pro podrobný popis odkáže na dokumentace jazyka SQL.

4.1 Příkazy manipulující se strukturou

Pro definici a úpravy tabulek a databází slouží příkazy pracující se strukturou. K tvorbě datových struktur je příkaz `CREATE`. Syntaxe příkazu je:

CREATE [NASTAVENI] CO_VYTVORIT jmeno [PARAMETRY]

Standard SQL99 definuje poměrně velké množství datových struktur. Jako nejdůležitější lze uvést:

- databáze (SCHEMA – někdy též DATABASE)
- tabulka (TABLE)
- pohled (VIEW)

Níže je stručně uveden význam vybraných struktur a základní syntaxe příkazu, jak je použít. Kompletní syntaxe není pro účel této práce potřeba.

4.1.1 Tvorba databází a úprava

Databázi je možné vytvořit pomocí příkazu `CREATE SCHEMA` s příslušnými parametry. Někdy je možné použít i `CREATE DATABASE`, ale tento zápis není v souladu se SQL99.

Jednoduchou databázi lze vytvořit příkazem:

```
CREATE SCHEMA `jmeno_databaze`
```

Příklad 2: Vytvoření databáze

Kód v příkladu 2 vytvoří prázdnou databázi se základní znakovou sadou, která je definována serverem. Pokud je vyžadována jiná znaková sada, je ji možné nastavit pomocí

doplňujících parametrů. Ve chvíli, kdy je databáze vytvořena, ji už dle standardu není možné měnit. Většina implementací ovšem podporuje změnu jména nebo výchozího kódování a porovnávání.

Smazání databáze se provede jednoduše pomocí příkazu `DROP SCHEMA`. Při aplikaci na příkladu 2 je to:

```
DROP SCHEMA `jmeno_databaze`
```

Příklad 3: Smazání databáze

Po odeslání tohoto příkazu se nenávratně smaže celá databáze včetně tabulek a dat v ní uložených, proto je vhodné předtím důkladně ověřit, jestli neobsahuje něco důležitého.

4.1.2 Tvorba tabulek a úprava

Po vytvoření prázdné databáze je možné přistoupit k tvorbě tabulek sloužících pro uchovávání dat, která budou v databázi uložena. Základní příkaz k vytvoření tabulky je CREATE TABLE. Na rozdíl od příkazu CREATE SCHEMA je nyní k dispozici mnohem víc možností definic a nastavení výsledné struktury.

Jako příklad pro vytvoření tabulky uživatelů je možné uvést:

```
CREATE TABLE `users` (
`id` int primary key,
`username` varchar(255) not null unique,
`country` varchar(40) not null default 'Czech republic'
)
```

Příklad 4: Vytvoření tabulky

Ukázka kódu v příkladu 4 vytvoří tabulku uživatelů. Ta obsahuje identifikační číslo, které je zároveň primárním klíčem, v dalším sloupci uživatelské jméno, jež musí vždy obsahovat nějakou hodnotu a také musí být v celé tabulce unikátní, a v posledním sloupci je země původu uživatele. Pokud není tento sloupec vyplněn, je automaticky nastavena Česká republika.

Tvorba tabulek poskytuje poměrně rozsáhlé možnosti omezení dat a kontroly integrity, kterým se však tato kapitola nebude více věnovat. Podrobně jsou rozvedeny v dokumentaci některého databázového systému nebo ve specifikaci SQL99.

Stejně jako tvorba tabulek je i jejich úprava, pomocí příkazu `ALTER TABLE`, poměrně komplikovaná a zde budou nastíněny jen základní operace:

- přejmenování tabulky
- přidání sloupce
- odebrání sloupce
- změna sloupce

Dále existují ještě množství operací se strukturou tabulky, jako je definice a úprava cizích klíčů a indexů, změna kódování a porovnávání dat a další

Nejjednodušší je patrně změna názvu tabulky. Jednoduchý příkaz v příkladu 5 přejmenuje tabulku se jménem `users` na `uzivatele`.

```
ALTER TABLE `users` RENAME TO `uzivatele`
```

Příklad 5: Přejmenování tabulky

Pokud je potřeba do tabulky přidat nový sloupec, použije se klauzule ADD COLUMN, za kterou se nachází jeho definice a případně pozice, na kterou se sloupec zařadí. Ukázka je v příkladu 6:

```
ALTER TABLE `uzivatele` ADD COLUMN `city` VARCHAR(255) AFTER
    `username`
```

Příklad 6: Přidání sloupce do existující tabulky

V této ukázce je přidán sloupec `city`, který je zařazen za `username`. Pokud by klíčové slovo `AFTER` nebylo použito, byl by sloupec zařazen jako poslední. Pro zařazení sloupce na první pozici se užívá místo `AFTER` slovo `FIRST`.

Úprava existujícího sloupce se provede příkazem `CHANGE COLUMN`. Za tímto výrazem potom následuje nová definice sloupce, podobně jako u vytváření nového.

```
ALTER TABLE `uzivatele` DROP COLUMN `city`
```

Příklad 7: Smazání sloupce z existující tabulky

Smazání sloupce se provede klíčovými slovy `DROP COLUMN`, za kterými se nachází

```
DROP TABLE `uzivatele`
```

Příklad 8: Smazání tabulky

jméno sloupce.

Celá tabulka se smaže, podobně jako sloupec, klíčovým slovem DROP. V tomto případě je ovšem použito v kontextu celé tabulky, jak ukazuje příklad 8.

4.1.3 Tvorba pohledů a úprava

Pokud jsou na tabulku (nebo více tabulek) často směřovány podobné dotazy, lišící se od sebe jen nepatrně, dá se vytvořit takzvaný pohled. Ten vznikne po odeslání příkazu typu CREATE VIEW na server.

Pohledy jsou tabulky, jejichž obsah je převzatý nebo odvozený z jiných tabulek. Ve skutečnosti jsou známy také jako virtuální tabulky nebo dokonce pohledové tabulky.

V praxi je možné při definici pohledu odkazovat i na pohled jiný. Potom záleží na implementaci, jestli bude kód z původního pohledu přejat do nového, nebo jestli se bude na původní pohled dotazovat [3].

Pro příklad definice pohledu je použita tabulka vytvořená v příkladu 4:

```
CREATE VIEW `from_cz` AS
SELECT * FROM `users` WHERE `country` LIKE 'Czech republic'
```

Příklad 9: Vytvoření pohledu

Pohled definovaný v příkladu 9 filtruje z tabulky `users` pouze ty lidi, kteří mají jako stát nastavenou Českou republiku. Pracujeme s ním dále jako s tabulkou, data tedy můžeme dále filtrovat například podle identifikačního čísla nebo emailové adresy.

Žádné manipulace s pohledem nejsou dle standardu SQL99 možné. Přesto některé systémy klauzuli ALTER VIEW implementují, ale její možnosti a chování jsou závislé na konkrétním systému [2].

```
DROP VIEW `from_cz`
```

Příklad 10: Mazání pohledu

Vytvořený pohled lze smazat pomocí příkazu DROP VIEW, za kterým následuje jméno pohledu ke smazání.

4.2 Příkazy pro manipulaci s daty

Po vytvoření datových struktur je do nich možné vkládat data, upravovat je a mazat. K těmto účelům slouží příkazy:

- `INSERT` – vkládání nových položek
- `SELECT` – vyhledávání v jedné nebo více tabulek/pohledů
- `UPDATE` – aktualizace dat
- `DELETE` – mazání záznamů

4.2.1 Vkládání dat do tabulek

Vkládání nových dat, které nejsou závislé na jiných tabulkách a syntaxi, se provádí pomocí příkazu `INSERT` se syntaxí:

`INSERT INTO `tabulka` VALUES (hodnota, hodnota, …), (hodnota, …),…`

Pokud je potřeba vložit hodnoty načítané z jiných tabulek:

`INSERT INTO `tabulka` SELECT …`

O příkazu `SELECT` je stručně pojednáno níže.

Když není jisté, zda primární klíče budou skutečně unikátní, je možné použít klauzuli `ON DUPLICATE KEY UPDATE`, která definuje, co se má při tomto konfliktu dělat. Příkaz `UPDATE` je stejně jako `SELECT` stručně popsán níže v textu.

4.2.2 Vyhledávání dat v tabulkách

Po vložení informací do databáze se dají konkrétní záznamy vyhledávat pomocí příkazu `SELECT`. Ten může být použit jak pro vyhledávání záznamů v jedné tabulce, tak i pro dotazy spojující záznamy z tabulek napříč celou databází.

Jednoduchá konstrukce příkazu `SELECT` může vypadat například takto:

```
SELECT `id`, `username`, `email` FROM `tabulka` WHERE
`is_admin` = 1
```

Příklad 11: Jednoduché vyhledání dat

Dotaz v příkladu 11 vyhledá identifikační čísla uživatelů z tabulky. Podmínkou pro vrácení řádku je hodnota 1 ve sloupci `is_admin`.

Pro vyhledání dat z více tabulek lze použít klauzule pro spojování záznamů. Touto problematikou se ovšem tato práce nezabývá, a proto nebude dále rozepisována.

4.2.3 Úprava záznamů

V případě potřeby upravovat již existující záznamy je možné použít klíčové slovo `UPDATE`. Za ním následuje seznam tabulek, kterých se úprava bude týkat, definice nových hodnot a nakonec vyhledávací kritérium.

```
UPDATE `users` SET `is_admin` = 0 WHERE `id` > 1
```

Příklad 12: Aktualizace dat

V příkladu 12 je uveden dotaz na databázi, který odebere administrátorská práva všem uživatelům, jejichž identifikační číslo je větší než 1.

4.2.4 Mazání záznamů

Smazání záznamu se provádí pomocí příkazu `DELETE`. Syntaxe je obdobná jako u příkazu `SELECT`, ovšem s tím rozdílem, že je zde vynechán výčet sloupců. Mazání uživatele s uživatelským jménem 'jannovak' je ukázáno v příkladu 13.

Mazat lze záznamy jednotlivě nebo po skupinách z jedné i z více tabulek spojených relací.

```
DELETE FROM `users` WHERE `login` LIKE 'jannovak'
```

Příklad 13: Mazání záznamu

4.3 Příkazy definující pomocné struktury

Pro zjednodušení práce s tabulkami a daty poskytuje standard SQL99 možnost vytvořit funkce, triggery a další pomocné struktury. Tyto struktury automatizují často prováděné operace přímo na úrovni databázového systému a není nutné pokaždé volat stejnou sekvenci příkazů ze strany klienta.

Vzhledem k omezení jazyka SQL umožňuje řada databázových systémů použít pro definici těla těchto struktur deriváty různých programovacích jazyků (například odvozeniny od Pythonu, Ruby, PHP, C/C++ a dalších) [4][5][6][7].

4.3.1 Funkce a procedury

Hlavička funkce, respektive procedura, se definuje pomocí posloupnosti klíčových slov `CREATE FUNCTION`, respektive `CREATE PROCEDURE`, které jsou následovány jménem funkce/procedury a jejími parametry [8][9]. Po hlavičce následuje klíčové slovo `BEGIN` obsahující vlastní kód a ten je ukončen příkazem `END`. V případě funkce je ještě nutné vracet výslednou hodnotu pomocí příkazu `RETURN`.

4.3.2 Triggery

Zásadní omezení funkcí a procedur spočívá v tom, že je nutné je spouštět ručně. Pro zaznamenaní všech změn v tabulce do logu by tedy bylo potřeba po každé jednotlivé změně zavolat příslušnou proceduru. Ke zautomatizovaní tohoto procesu slouží triggery.

Trigger se definuje příkazem `CREATE TRIGGER <jmeno>` a okamžikem spuštění procedury:

- `AFTER` – kód se spustí po proběhnutí události
- `BEFORE` – kód se spustí před událostí

Po specifikaci okamžiku spuštění následuje výčet událostí, pro které se má procedura spustit. Události mohou být `INSERT`, `UPDATE` nebo `DELETE`. Pro spuštění více událostí jsou jednotlivá jména oddělena příkazem `OR`. Po této sekvenci příkazů je specifikována tabulka `ON <jmeno_tabulky>`, a jestli se má procedura volat pro každý řádek (`FOR EACH ROW`), nebo pro celou ovlivněnou skupinu řádků (`FOR EACH STATEMENT`). Celý zápis je ukončen příkazy `EXECUTE PROCEDURE <jmeno_procedury>(parametry)`.

Ukázka definice triggeru je v příkladu 14:

```
CREATE TRIGGER `novy_trigger` AFTER INSERT OR UPDATE
ON `uzivatele` FOR EACH ROW `zapis_zmenu`(CURRENT_DATE())
```

Příklad 14: Definice triggeru

Tento kód vytvoří trigger, který po vložení nebo modifikaci řádku v tabulce uživatelů spustí proceduru `zapis_zmenu`, s parametrem aktuálního data (funkce z MySQL).

4.4 Ostatní příkazy

Kromě výše zmíněných příkazů pro tvorbu struktur, definici dat, funkcí, procedur a triggerů obsahuje SQL99 také příkazy pro práci s prostředím, pro dávkové zpracování dat a podobně.

Pro práci s prostředím se používají příkazy pracující s proměnnými. Jsou to:

- `SHOW VARIABLES` – vypíše seznam proměnných a jejich hodnot
- `DECLARE <jmeno> <typ>` – vytvoří novou proměnnou daného typu
- `SET <jmeno> = <hodnota>` – nastaví novou hodnotu proměnné
- `DEALLOCATE <jmeno>` – odstraní proměnnou

Pomocí těchto příkazů lze manipulovat s nastavením prostředí systému. Využití najde například při hromadném nahrávání nových hodnot, kdy je potřeba vypnout automatickou kontrolu integrity, nebo pro nastavení výchozího formátu tabulky (MySQL).

Dále se jedná například o kurzory, které umožňují dávkové zpracování dat. Svojí strukturou jsou velmi podobné pohledům (viz 4.1.3 Tvorba pohledů a úprava), ovšem práce s nimi je odlišná. Po inicializaci kurzoru je vrácena dávka dat, která může být zpracována podle požadavků klientské aplikace buď najednou, nebo postupně. Toho lze využít pro stránkování dat zobrazených uživateli, pro snížení zátěže serveru/klienta i jinde.

5 PŘEHLED VYBRANÝCH DATABÁZOVÝCH SYSTÉMŮ

V této kapitole je uveden stručný přehled vybraných databázových systémů. Zvoleny byly dva systémy open source, dva komerční systémy a jedna knihovna pro práci s daty za použití SQL.

5.1 MySQL

Databázový systém MySQL má velké zastoupení na českých webových serverech (subjektivní názor autora). Obsahuje několik typů databázových úložišť, které se liší funkcemi a metodou ukládání dat. Jedná se zejména o:

- MyISAM – data jsou komprimována, umožňuje fulltextové vyhledávání [4]
- InnoDB – podporuje transakce a automatickou kontrolu integrity dat mezi tabulkami (cizí klíče), nevýhodou je, že v tabulce tohoto typu není možné definovat fulltextový klíč [4]
- Memory – úložiště pro dočasné tabulky, data jsou uchovávána pouze v paměti a nedochází k zápisu na disk [4]
- CSV – pracuje s textovými soubory, kde jsou data oddělena čárkami, výhodou je snadná čitelnost dat v jakémkoliv textovém editoru, ale nepodporuje klíče ani automatickou inkrementaci [4]

Pro práci s databází je možné využít jak konzolové nástroje (například nástroj mysql v linuxu), tak i nástroje s GUI. Od vývojářů MySQL je k dispozici sada grafických nástrojů, která obsahuje program pro grafický návrh databáze (MySQL Workbench zmíněný v kapitole 3.1), správu struktury databáze (MySQL Administrator) a klient pro práci s daty (MySQL Query Browser).

Ke komunikaci programu s databází existují knihovny (konektory) pro různé programovací jazyky. V poslední době jsou nejvyužívanější zejména C, Perl, PHP a platforma .NET [4].

5.2 PostgreSQL

Databázový systém PostgreSQL vyvinul tým na Kalifornské univerzitě v Berkeley. Původní verze Posgres95 vznikala 1994-1995, na přelomu let 1996 a 1997 byla přejmenována na PostgreSQL [5]. Jedná se o objektově orientovanou databázi. To znamená, že tabulky mohou dědit z jiných tabulek a při výběru dat se dá odkazovat na předky a potomky. Při psaní funkcí je možné použít kromě SQL i celou řadu dalších (modifikovaných) jazyků jako PL/pgSQL, plPHP, PL/Lua, C/C++ a další [5]. Práce s daty je automaticky řešena pomocí transakcí. Díky tomuto přístupu je omezena pravděpodobnost vzniku chybných dat při konkurenčním přístupu více uživatelů ke stejným datům.

5.3 Oracle

Oracle je komerční systém vyvíjený firmou Oracle Corporation. Tato databáze se vyznačuje důrazem na paralelní zpracování dat. K tomu využívá takzvaného gridu (označeno písmenem g za číslem verze). Grid umožňuje jak škálování databází, tak i rozložení zátěže na méně využívané fyzické stroje. Dále je schopen jednotlivé databáze automaticky mezi těmito stroji přesouvat podle toho, je zrovna potřeba [6].

Pro rozšiřování systému a pro tvorbu funkcí využívá jazyk PL/SQL. Tento jazyk byl vyvinut přímo společností Oracle. Data jsou standardně vracena ve formátu XML. To umožňuje snadný přenos výstupních dat mezi vlastním databázovým serverem a cílovou aplikací, aniž by vznikalo omezení kvůli použití rozdílných platforem [6].

5.4 Microsoft SQL Server

Microsoft SQL Server má na rozdíl od výše popsaných systémů modulární architekturu a je určen zejména pro komerční použití ve firmách jako ucelený informační systém. Od toho se odvíjí také sada modulů dodávaných s programem.

Základem celého systému je databázový modul obsahující další podmoduly. Jeho jádrem je modul úložiště (Storage module), který se stará o vlastní práci s daty na fyzickém disku, dále obsahuje programovací rozhraní, zabezpečovací podmodul a podobně. Dalším modulem je Bussines Intelligence umožňující zprostředkování a analyzování dat z databáze běžným uživatelům [7].

Microsoft SQL Server je stejně jako Oracle natolik rozsáhlý program, že je nad rámec této práce popisovat všechny jeho komponenty.

5.5 SQLite

SQLite není přímo databázový systém. Je to knihovna napsaná původně v C a poté převedena do mnoha dalších jazyků (PHP, Ruby, Python, Java, atd.). Implementuje kompletní specifikaci jazyka SQL99. Data ukládá do souboru za použití hashovaných indexů [10].

Díky poměrně jednoduchému programovému rozhraní je možné tuto knihovnu použít všude tam, kde nejsou kladeny vysoké nároky na složité dotazy nebo velký objem dat. Je zpravidla využívána v PDA, mobilních telefonech, MP3 přehrávačích a podobných zařízeních [10].

6 ZPRACOVÁNÍ STRUKTURY DATABÁZE

Systém, který je výstupem této práce, umožňuje přenos tabulek, funkcí a procedur. Pro vlastní realizaci je nutné nejprve zjistit popis těchto struktur v databázi.

Naneštěstí není přístup k definicím databázových struktur sjednocený a liší se zpravidla systém od systému. Z tohoto důvodu je nutné pro každý typ databáze vytvořit speciální adaptér, který bude schopen pomocí svých metod získat informace o struktuře a sestavit z nich reprezentaci databáze v paměti.

Jako příklad zmiňované diverzity je možné uvést situaci, kdy potřebujeme vypsat SQL kód pro vytvoření tabulky jménem `moje_tabulka`. Tento úkon budeme provádět na databázových systémech MySQL, PostgreSQL a SQLite.

Nejjednodušším případem je MySQL, kde je k tomuto účelu určen zvláštní příkaz SHOW CREATE TABLE Chyba: zdroj odkazu nenalezen:

```
SHOW CREATE TABLE `moje_tabulka`
```

Příklad 15: Zobrazení SQL kódu vytvářejícího tabulku v MySQL

Tento příkaz vypíše zdrojový SQL kód, jehož provedením bude vytvořena `moje_tabulka`.

Poněkud obtížnější situace je u PostgreSQL, která žádnou podobnou funkci nemá v dokumentaci uvedenou Chyba: zdroj odkazu nenalezen. Funkci SHOW sice implementuje, ale slouží výhradně k účelu zobrazení hodnot nastavení běhového prostředí databáze [5]. Námi požadovanou funkci plní utilita jménem pg_dump, volaná s parametry –schema-onlya -t :

```
pg_dump –schema-only -t=moje_tabulka moje_database
```

Příklad 16: Export struktury databáze z PostgreSQL

Naopak SQLite poskytuje nejjednodušší přístup k těmto datům. Tato jednoduchost vyplývá přímo z povahy tohoto „databázového systému“, který je implementován pouze jako knihovna, a ne jako samostatný databázový systém. Celá databáze je uložena v jediném souboru, k němuž knihovna přistupuje Chyba: zdroj odkazu nenalezen. Pomocí API je načten soubor s databází a nad databází je zavolán dotaz na tabulku sqlite_master, která obsahuje definice všech struktur v souboru [14].

```
SELECT sql FROM sqlite_master WHERE type like 'table' and name
    like 'moje_tabulka'
```

Příklad 17: Zobrazení SQL kódu pro tvorbu tabulky v Sqlite

Výsledek všech těchto příkazů vypadá velmi podobně. Ve výpisu 15 je kód z MySQL, 16 je PostgreSQL a 17 obsahuje výpis z tabulky `sqlite_master` u SQLite:

```
CREATE TABLE `moje_tabulka` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `admin` tinyint(1) NOT NULL DEFAULT '1',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Příklad 18: Výstup z příkladu 15

```
CREATE TABLE moje_tabulka (id INTEGER PRIMARY KEY, name
    VARCHAR(255), admin BOOLEAN NOT NULL DEFAULT 1)
```

Příklad 19: Výstup z příkladu 17

Jak ukazují příklady a definice SQL-99, definice nové struktury vždy začíná klíčovým slovem CREATE, za kterým následuje popis toho, co bude vytvořeno. Je však nutné poznamenat, že MySQL umožňuje značně jednodušší přístup k definičním datům pomocí virtuální databáze information_schema, o které už byla zmínka dříve.

Pro převod kódu do formy zpracovatelné programem se jeví jako nejlepší řešení použít deterministický konečný automat se zásobníkem (dále jen DKAZ). Zásobník je nezbytný z toho důvodu, že se v kódu mohou vyskytovat závorky, které uzavírají vhnízděné struktury. Tento automat potom bude procházet zdrojový kód pro tvorbu dané struktury a podle obsahu vytvářet a nastavovat instance tříd, které odpovídají jejím součástem (například sloupec tabulky, zdrojový kód funkce atd.).

6.1 Návrh modelu DKAZ

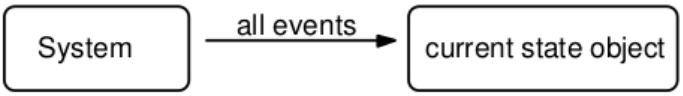
Díky tomu, že jazyk SQL je normalizován standardem SQL99, by se mohlo na první pohled zdát, že lze celý automat realizovat jedinou třídou. Ta by reprezentovala automat, její metody by reprezentovaly stavy a volání metod zase překlápění stavů. Výhodou tohoto řešení je „zabalení“ celého automatu do instance třídy. Velmi závažná nevýhoda je ale v pevném nastavení podoby stavů a jejich chování při zpracovávání výrazu. Z tohoto důvodu musí být popsané řešení okamžitě zavrženo, neboť takový automat by byl předurčen pouze ke zpracování čistě SQL99 kompatibilního kódu. Uzavřela by se tím také cesta k rozšířením poskytovaným jednotlivými implementacemi, ale i k možnosti potlačení částí, které dané implementace ignorují, nebo implementují v rozporu se standardem.

Mnohem flexibilnější je reprezentovat konečný automat pomocí grafu. Automat jako celek je reprezentován jako graf, stavy jako uzly a hrany jsou přechody mezi stavy. V tomto modelu se reprezentuje zásobník pomocí uzlu (stavu), který obsahuje vnořený graf (automat). Řešení pomocí grafu je výhodné pro možnost změnit strukturu automatu podle aktuálních potřeb (například kvůli rozdílům v jednotlivých databázových systémech) kdykoliv za běhu programu. Proto byl tento přístup zvolen pro výsledný program.

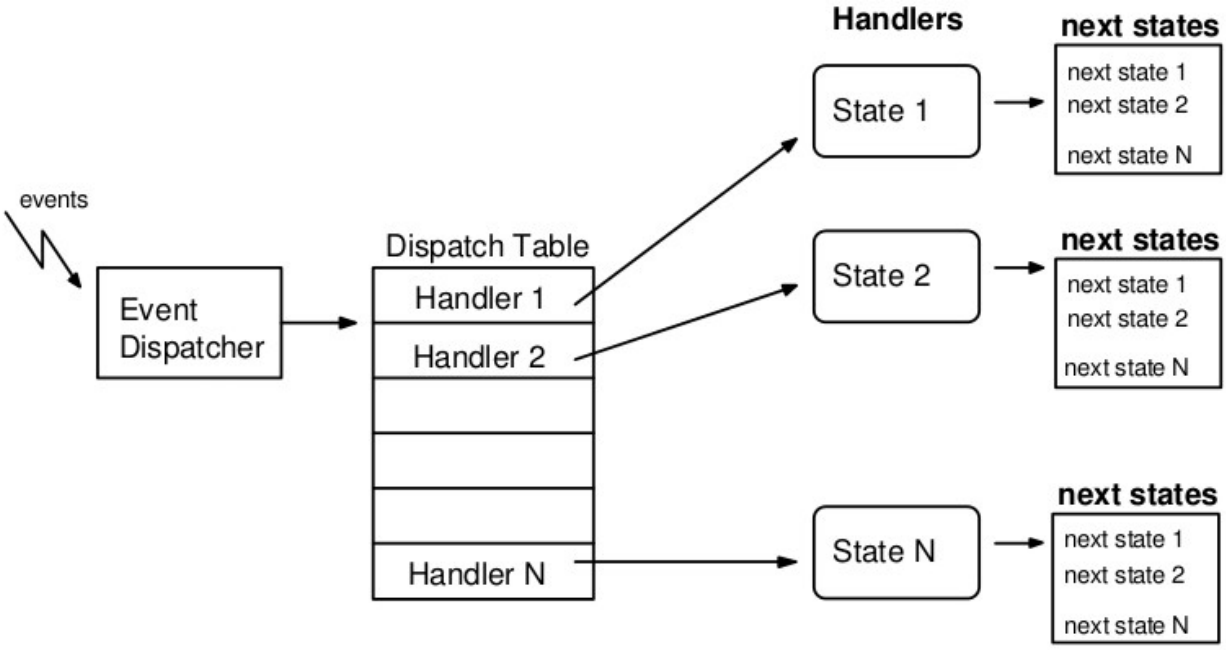
Dále je nutné určit metodu implementace DKAZ, jestli bude program tvořen funkcemi, nebo objekty. Pokud by byl automat tvořen funkcemi, musel by existovat seznam obsahující ukazatele na tyto funkce (v tomto přístupu reprezentace stavů). Ty by nejdříve vykonaly příslušné operace a poté přepsaly tabulku dalšími možnými stavy, do kterých se by se mohly překlopit [8]. Výběr správného stavu, který volat, má v tomto případě na starosti specializovaná funkce, takzvaný dispečer událostí (anglicky event dispatcher). Schéma možné

implementace je na obrázku 3. Tento nízkoúrovňový přístup se někdy také nazývá Tradiční přístup [8]. Podobná metoda je použita například u kompilátorů rodiny GCC, kde na rozdíl od výše uvedeného popisu neexistuje seznam stavů jako pole, ale další postup je vyhodnocován pomocí programové konstrukce `SWITCH-CASE`. Tato konstrukce nahrazuje seznam dalších stavů bez nutnosti použít pole ukazatelů na funkce.

Objektový přístup zase reprezentuje automat jako instanci třídy reagující na podněty z vnějšího světa (předávaný řetězec a podobně). Diagram možné implementace za použití objektů je na obrázku 2. V tomto případě je automat reprezentován přímo svým stavem, kterému jsou předávány události, na něž buď reaguje, nebo je ignoruje. Pokud nedojde k překlopení stavu, nemusí to znamenat, že stav na událost nereagoval. Mohlo dojít k vnitřní změně stavu, která se nijak neprojevila na jeho vnějším popisu [8].



Obrázek 2: Implementace automatu pomocí objektů [8]



Obrázek 3: Reprezentace DKAZ pomocí funkcí [8]

DKAZ použitý pro zpracování struktur využívá objektový přístup, kde aktuální stav zkoumá na základě možnosti přechodu po hraně do následujícího stavu.

6.2 Graf

Než bude popsán mechanismus přechodu mezi jednotlivými stavy, je potřeba popsat reprezentaci grafu, který je použit jak pro DKAZ, tak i pro reprezentaci databázových struktur popsaných v kapitole 7.2.

Pro reprezentaci grafu budou použity tři třídy:

- třída reprezentující celý graf (`Graph`)
- třída reprezentující uzel (`GraphNode`)

- třída reprezentující hranu (`GraphEdge`)

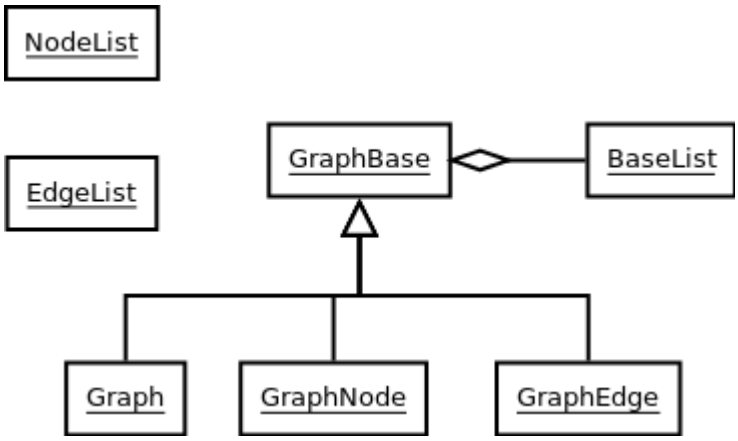
Třída grafu obsahuje metody pro selekci uzlů určitého typu (viz následující odstavec), filtrační metody pro tvorbu podgrafů, metody hledání izolovaných uzlů a samozřejmě metody pro přidání uzlů a hran. Třída také obsahuje celkový seznam uzlů (bez ohledu na typ), aby se filtrování, vyhledávání a sestavení podgrafu urychlilo. Kvůli optimalizaci jsou v seznamu tyto uzly seřazeny podle hodnoty příznaku typu.

Uzel grafu je základní stavební kámen jak pro automat, tak i pro reprezentaci databázových struktur. Jak bylo uvedeno výše, má důležitou vlastnost, a to příznak typu. Tento příznak je 32 bitové celé číslo, pomocí kterého lze identifikovat, co daný uzel reprezentuje, a slouží zejména pro reprezentaci databázových struktur. Více je o něm napsáno v kapitole 7.2.

Poslední stavebním kamenem grafu je hrana. Přestože tento graf je orientovaný, hrana uchovává informaci o obou svých koncích, tedy o výchozím i koncovém uzlu. Díky této vlastnosti je každá hrana programově prostupná z obou stran.

Kromě těchto tří základních tříd jsou definovány ještě třídy pomocné. Sem patří bázová třída (`GraphBase`), od které dědí všechny ostatní prvky grafu a kontejnerové třídy pro uchování seznamu:

- libovolných prvků grafu (kontejner bázových tříd)
- hran
- uzlů



Obrázek 4: Hierarchie tříd grafu

Seznam pro uchování libovolných prvků (seznam bázových tříd) je určen pro použití uvnitř tříd, a proto je zapouzdřený v bázové třídě. Naopak seznamy hran a uzlů slouží ke komunikaci se zbytkem programu, a jejich definice jsou tedy součástí veřejného rozhraní.

Bázová třída obsahuje kromě definice datového typu obecného seznamu (zmíněného výše) také pole s instancemi tohoto seznamu. Toto pole seznamů slouží k sjednocení práce s prvky grafu na úrovni společného předka, a tím pádem ke zjednodušení spolupráce mezi prvky grafu. Spoluprací mezí prvky grafu je myšlena reakce grafu a jeho prvků na vytvoření, nebo zánik některého svého prvku. Příklad takové spolupráce je ve výpisu 20:

```
1. Graph graph;
2. GraphNode *node1 = new GraphNode(&graph);
3. GraphNode *node2 = new GraphNode(&graph);
4. GraphEdge *edge = new GraphEdge(&graph, node1, node2);
5. delete node1;
```

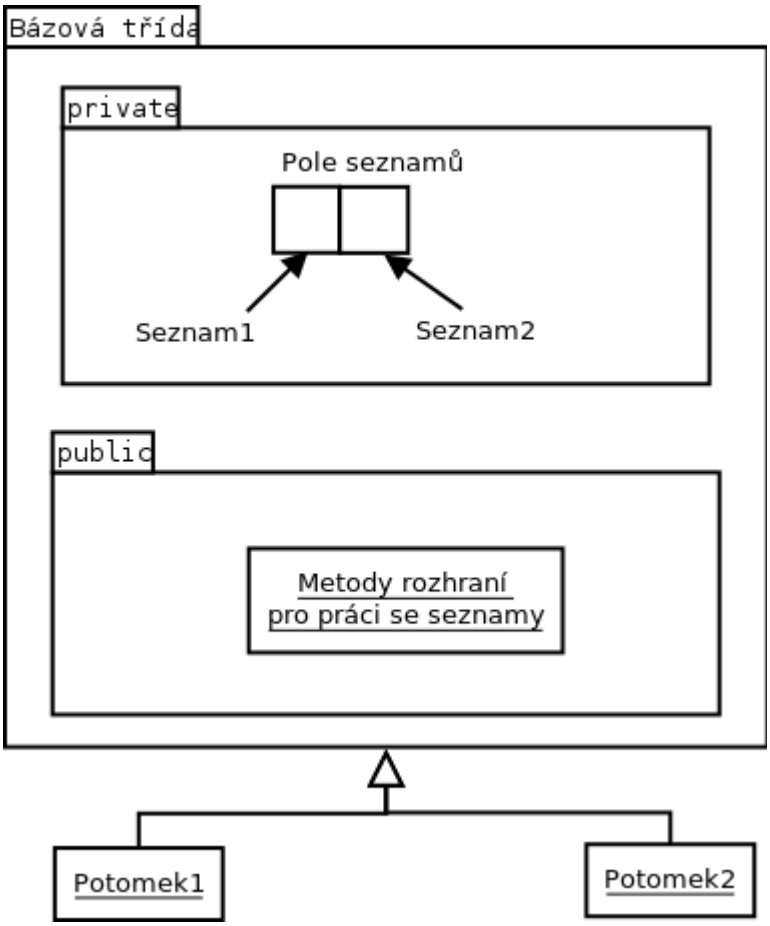
Příklad 20: Spolupráce prvků grafu

Na jednotlivých řádcích se děje následující:

1. Vytvoří se instance grafu (bázová třída vytvoří seznam pro uzly a seznam pro hrany).
2. Vytvoří se nový uzel. Do seznamu uzlů grafu se zapíše ukazatel na tuto instanci.
3. Vytvoří se druhý uzel.
4. Vytvoří se hrana mezi dvěma vytvořenými uzly. Hrana je zapsána do seznamu hran v grafu a také je zanesena do seznamu výstupních hran prvního uzlu a do seznamu vstupních hran druhého uzlu.
5. Je odstraněna instance prvního uzlu. Před vlastním smazáním je odstraněna vytvořená hrana, která ve svém destruktoru odstraní ukazatele na sebe ze seznamů uzlů a z grafu. Po odstranění hrany je odstraněn i ukazatel na instanci uzlu ze seznamu vrcholů grafu.

Celý proces manipulace se seznamem druhého prvku probíhá bez rekurze nebo volání metod druhého prvku. Díky tomu, že oba prvky mají stejnou bázovou třídu, která dané seznamy obsahuje, může instance bázové třídy jednoho prvku (například hrany) přímo přistupovat k seznamům druhého prvku (například uzlu nebo grafu) a měnit ho.

Nákres struktury bázové třídy je na obrázku 5:



Obrázek 5: Schéma bázové třídy

Pole seznamů je inicializováno v konstruktoru bázové třídy a počet prvků závisí na na typu daného objektu

- graf: 2 prvky (seznam hran a uzlů)
- uzel: 2 prvky (seznam vstupních hran a seznam výstupních hran)
- hrana: žádný prvek (předchůdce a následník jsou uchovávány jako ukazatele)

Protože prvky seznamů, které jsou uloženy v tomto poli, jsou typu bázové třídy, je nutné je vždy pro dané použití přetypovat.

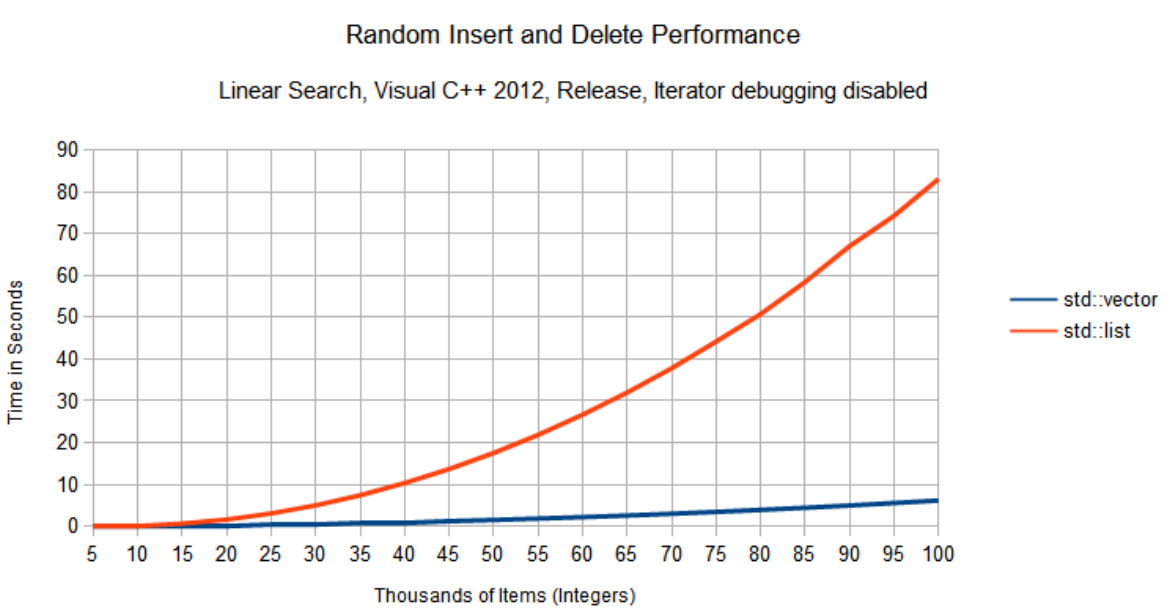
Jak je patrné z obrázku 6, bázová třída poskytuje pro práci se seznamy rozhraní s viditelností `public`. Toto rozhraní umožňuje:

- získat ukazatel na instanci seznamu dle indexu
- odstranit seznam
- přidat nový seznam nebo seznamy

Pro reprezentaci seznamů prvků grafu se nabízejí třídy `Vector` a `List` z STL. Třída `Vector` má výhodu náhodného přístupu k prvkům, ale na druhou stranu je značně pomalá při odebírání nebo přidávání prvků doprostřed seznamu. `List` sice nemá náhodný přístup (implementuje pouze dopředný a zpětný iterátor), ale operace nad prvky uvnitř množiny jsou značně rychlejší.

Je nutné také uvážit, zda je možné využít náhodný přístup iterátorů třídy `Vector` a odhadnout, jak často se během životního cyklu grafu budou mazat nebo přidávat prvky doprostřed seznamu. Objekty grafu jsou identifikovány pouze svojí adresou v paměti (uvažujeme ryze o grafu zatím bez jakékoli vazby na prvky databáze nebo konečného automatu). V některých případech bude potřeba určité prvky řadit dle určitého klíče

(například při výstupu uživateli řadit dle abecedy) a k těmto seřazeným prvkům náhodně přistupovat. V této úvaze ovšem vzniká spor: chceme využít náhodný přístup, který poskytuje iterátor `Vectoru`, ale zároveň provádět změny uvnitř seznamu, které má lépe optimalizován



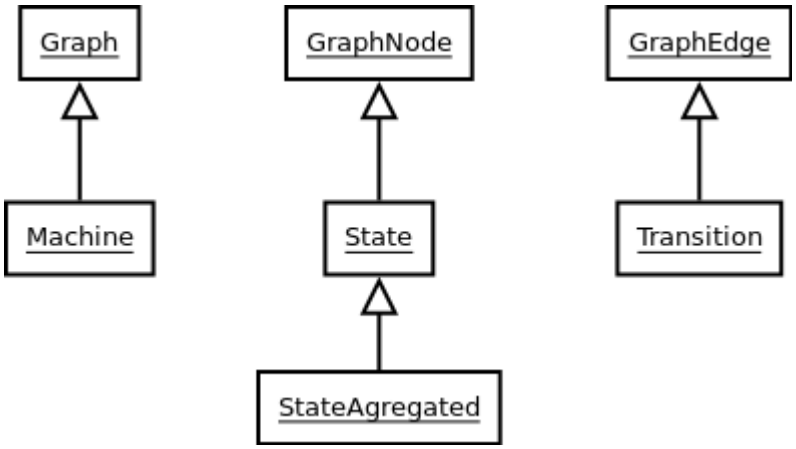
Obrázek 6: Časová náročnost náhodného zápisu do instance Vector a List [15]

`List`. Rozdíl při náhodném mazání a přidávání prvku je patrný z grafu na obrázku 6:

Protože vznikl konflikt, který se stal „začarovaným kruhem“, bylo potřeba se na problém podívat z trochu jiného úhlu pohledu. Předpokládejme, že bude docházet k velkému množství úprav a budeme tedy potřebovat náhodné mazání. V tomto směru je jednoznačný vítěz `List`. Dále vezměme na vědomí, že iterátory `Vectoru` se zneplatní, pokud dojde k realokaci paměti. Je předpoklad, že k realokaci bude docházet a bude potřeba udržet stále platné iterátory. Zde je opět jedinou volbou `List`. Pro ukládání seznamů byl tedy zvolen `List`.

6.3 Struktura automatu a mechanismus přechodu

Jak bylo napsáno výše, automat je založen na reprezentaci pomocí grafu. Na obrázku 7 je znázorněna hierarchie základních tříd automatu vzhledem k třídám grafu.



Obrázek 7: Základní třídy automatu

Třída `Machine` reprezentuje celý automat. Obsahuje pouze metodu, která spustí běh automatu nad zadanými vstupními a výstupními daty.

Instance typu `State` jsou stavy automatu, do kterých se může na základě vstupních dat automat překlopit. Zvláštním případem je `StateAgregated`, který obsahuje zanořenou třídu `Machine`, a realizuje tak zásobník.

Stavy jsou mezi sebou propojeny přechody typu `Transition`. Přechody obsahují metodu pro vyhodnocení možnosti přechodu do cílové stavu a metodu, která realizuje vlastní přechod. Během něj se zároveň načítají vstupní data předávaná cílovému stavu. Ten potom z načtených informací generuje výstupní data.

6.3.1 Pomocné třídy automatu

Kromě hlavních tříd, které slouží k reprezentaci automatu existuje ještě několik pomocných tříd. Tyto jsou využívány pro realizaci přechodu, generování výstupu, uchování vstupních dat a podobně. Konkrétně to jsou:

- třída pro uchování vstupních dat
- třídy pro načítání dat (potomci třídy `Loader`)
- filtrační třídy (potomci třídy `Filter`)
- testovací třídy (potomci třídy `Test`)
- třídy pro generování dat (potomci třídy `Processor`)
- třídy pro generování dat vystupujících ze zásobníku (potomci třídy `Assembler`)
- třídy pro reprezentaci instance stavu

V průběhu zpracování vstupních dat automatem jsou tato data uložena ve specializované třídě `InputData`. Třída uchovává informace o délce dat, aktuální pozici v datech, ukazatel na vlastní data (typu `std::string`) a ukazatel na počítadlo instancí.

Obrázek 8: Informace v InputData

Ukazatel na řetězec je použit kvůli úspoře paměti. Při zpracování dat vzniká při každém přechodu nová kopie instance vstupních dat. Pokud by se jednalo o dlouhý řetězec, vzniklé instance by zabíraly zbytečně mnoho paměti. Toto řešení používá pouze jednu kopii vstupních dat a při vytváření nové instance se vždy zvýší počítadlo instancí a naopak v destruktoru se počítadlo o jedničku sníží. Pokud počítadlo klesne na nulu, pak je zničen i řetězec vstupních dat.

Z třídy `InputData` načítá informace instance typu `Loader`. Jedná se o ryze virtuální třídu, a načítat data mohou až její potomci. Během načítání dat navíc vytvoří novou instanci vstupních dat s pozicí v datech posunutou o načtený segment. Tato třída také obsahuje metodu pro přeskakování počátečních bílých znaků.

Přímo v knihovně automatu jsou definováni potomci:

- LoaderChar – načte vždy jeden znak
- LoaderMultichar – načte přesný počet znaků
- LoaderWord – načte slovo ukončené nastavenými znaky s možností escapování těchto znaků

Po načtení dat ze vstupního řetězce je potřeba rozhodnout, zda je možné provést přechod do následujícího stavu. Než se tak stane, je občas vhodné provést úpravu tohoto kusu textu, aby se snáze vyhodnocoval. Tyto úpravy provádějí potomci třídy `Filter`. V knihovně je pouze jediný potomek této třídy, a sice `FilterCase`. Ten převede, dle nastavení, všechna ASCII písmena na velká (upper case), nebo malá (lower case).

Jakmile jsou data načtena a přefiltrována, dá se otestovat, zda vyhovují podmínkám přechodu. Kontrolu správnosti dat provádí třída `Test`. Výstupem z této třídy je logická hodnota indikující úspěšnost testu. V základu jsou definovány dva testy:

- `TestLength` – zkontroluje délku načtených dat. Test je možné nastavit, aby vracel logickou jedničku, pokud se délka načtených dat shoduje s kontrolní délkou nebo je menší či je větší
- `TestEquals` – načtený fragment dat musí přesně (znak po znaku) odpovídat referenčnímu vzorku, který je instanci přiřazen

Pokud načítaná data vyhovují testům, lze je zpracovat. Ke tomu slouží potomci třídy `Processor`. Tato třída generuje výstupní data na základě načítaných informací ze vstupu pomocí `Loaderů`. Kromě vlastního generování obsahuje ještě metodu pro vrácení dat do stavu před generováním. Tato metoda ovšem nemusí být vždy implementována, protože některé operace nelze vrátit zpět. Jediným potomkem této třídy je ProcessorString, který načtený řetězec připojí na konec generovaných dat (data musí být samozřejmě řetězcového typu). Pokud je volána metoda pro vrácení do původního stavu, pak je generovaný řetězec zkrácen o načtený počet znaků.

Obdobnou funkci jako `Processor` má třída `Assembler`. Rozdíl mezi těmito dvěma třídami je v tom, že `Processor` zpracovává data načtená stavem, kdežto `Assembleru` jsou předávána data načítaná agregovaným automatem. Navíc obsahuje metodu pro přípravu dat zanořenému DKAZ. Tato zvláštní metoda vytvoří prázdná vstupní data a po zpracování jsou tato data připojena ke generovaným datům.

Velmi specifickou funkci mají instance tříd `StateInstance` a `StateAgregatedInstance`. Tyto dvě třídy fungují jako články obousměrně zřetězeného seznamu a reprezentují „historii průchodu automatem“. Během překlopení do nového stavu automatu, je vygenerována instance jedné z těchto tříd (`StateInstance` pro klasické stavy, `StateAgregatedInstance` pro stavy se zanořeným automatem) a je připojena na konec řetězu. Pokud se automat dostane do „slepé uličky“ a nemá možnost ze svého aktuálního stavu pokračovat dál, může se pomocí této historie vrátit o libovolný počet stavů zpět (včetně rekonstrukce původních dat) a pokusit se jít jinou cestou. Toto zřetězení umožňuje za určitých okolností vytvořit a spustit i nedeterministický konečný automat. Těmito určitými okolnostmi jsou zejména vnitřní uspořádání stavů a vhodná vstupní data.

6.3.2 Mechanismus překlápění stavů a generování dat

Mnohokrát byla zmíněna data generovaná zpracováním vstupního řetězce, ale nebyla popsána jejich struktura. Důvodem je, že žádná pevná struktura výstupních dat neexistuje. Při spuštění automatu jsou tato data, která budou generována, předána pouze jako obecný ukazatel na `void`. Až při manipulaci s nimi třídou `Processor` nebo `Assembler` je tento ukazatel přetypován na správný datový typ. Tato neurčitost generovaných dat je důvodem pro ryzí virtualitu těchto dvou tříd. Ukazatel na `void` si mezi sebou předávají stavy uvnitř automatu během překlápění.

Následuje algoritmus překlápění stavů, u kterého si je třeba uvědomit rozdíl mezi stavem (třídy `State` a `StateAgregated`) a instancí stavu, neboť tyto pojmy mohou být značně matoucí. Pro připomenutí těchto rozdílů viz kapitoly 6.3 a 6.3.1.

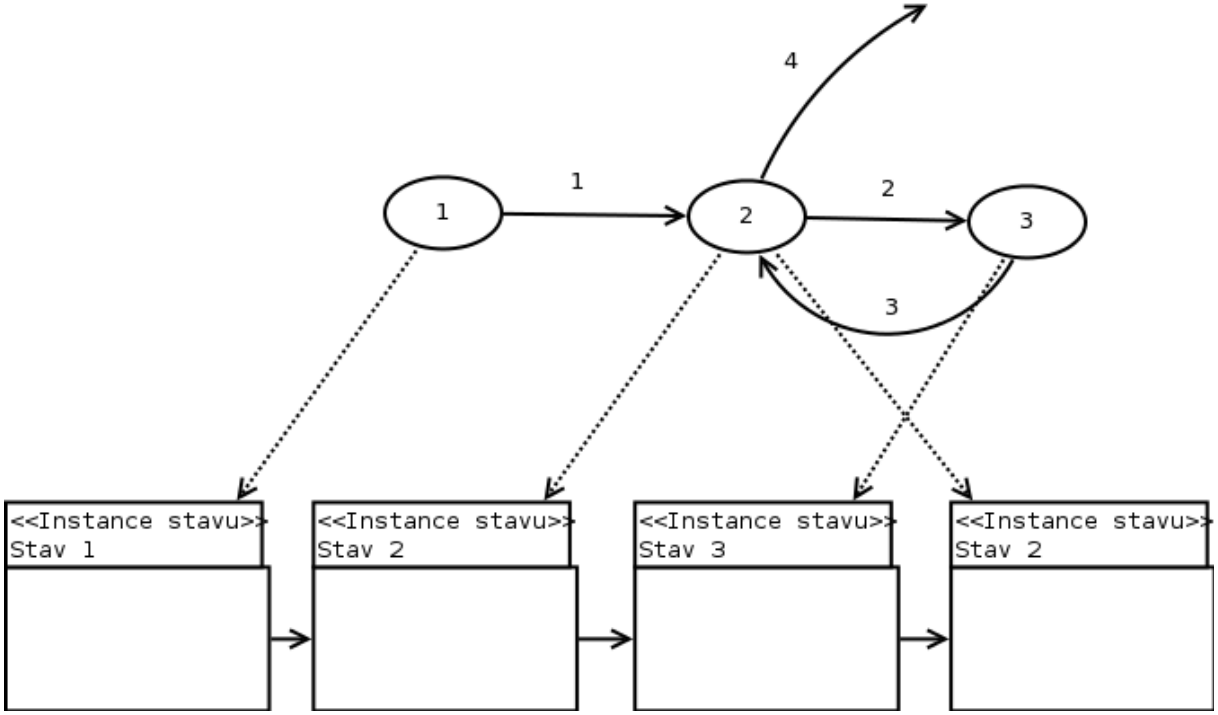
1. zavolej spouštěcí metodu a předej jí ukazatel na generovaná data a na vstupní řetězec
2. vytvoř instanci vstupního stavu
3. nastav ukazatel na vstupní instanci jako aktuální
4. je ukazatel na aktuální instanci NULL? Pokud ne, pokračuj na 5, pokud ano, pokračuj na 11
5. zavolej překlopení za aktuální instance stavu a návratovou hodnotu nastav jako aktuální instanci
    a) vezmi stav automatu (třída `State` nebo její potomek) a zavolej metodu přechodu
        1. iteruj nad seznam přechodů a najdi iterátor ukazující na vybraný přechod
            a) pokud je zvednutá vlajka nalezení vhodného přechodu nebo byly zkontrolovány všechny přechody pokračuj na 5.a)1.e)
            b) zkontroluj dostupnost aktuálního přechodu
                1. načti data pomocí třídy `Loader`
                2. spusť nad načteným řetězcem filtry
                3. spusť nad přefiltrovanými daty testy

4. pokud všechny testy skončily úspěchem, pak je přechod dostupný, pokud byl alespoň jeden test neúspěšný, pak přechod dostupný není
   c) pokud je přechod dostupný, zvedni vlajku nalezení, pokud ne, tak pokračuj na další přechod
   d) vrať se na 5.a)1.a)
   e) vrať aktuální iterátor
2. pokud iterátor ukazuje na jednu pozici za poslední prvek seznamu přechodů pak
   a) pokud je stav koncový, vrať hodnotu NULL
   b) pokud není stav koncový, vyvolej výjimku a pokračuj na 6
3. aktivuj přechod
   a) vytvoř novou kopii vstupních dat
   b) načti data pomocí třídy `Loader`
   c) uchovej neupravená načtená data
   d) přefiltruj data
   e) uchovej přefiltrovaná data
   f) vrať novou instanci vstupních dat a kolekci obsahující
4. vytvoř novou instanci stavu
   a) inicializuj výchozí vlastnosti instance stavu
   b) pokud původní stav obsahuje zanořený automat, zařaď tento automat na vrch zásobníku a spusť zanořený automat
   c) pokud původní stav má nastavený `Processor` respektive `Assembler`, zavolej jejich příslušné metody a proveď generování dat
5. vrať nově vytvořenou instanci stavu

b) vezmi ukazatel na následující článek řetězu
c) pokud je ukazatel na následující článek řetězu historie NULL, pak vrať NULL
d) v opačném případě vrať poslední článek řetězu

6. pokud nebyla zachycena výjimka, pokračuj na 4
7. pokud není možné se vrátit o jeden krok v řetězu zpět, pokračuj v distribuci výjimky (chybové ukončení běhu)
8. načti předchozí článek řetězu historie a nastav ho jako aktuální instanci stavu
9. proveď rekonstrukci původních dat a odříznutí následujících článků řetězu
   a) zavolej metodu pro rekonstrukci dat následujícího článku řetězu
      1. zavolej rekurzivně tuto metodu na následujících článcích řetězu
      2. vyžádej si `Processor` nebo `Assembler` původního stavu (třída `State` nebo její potomek)
      3. proveď rekonstrukci dat
   b) znič instanci následujícího článku (provede se rekurzivně na konec řetězu)
   c) nastav ukazatel na následující článek na NULL
10. pokračuj na 4
11. vrať první instanci stavu v řetězu

Výsledkem tohoto algoritmu jsou vygenerovaná data a řetězec instancí stavů, který reprezentuje historii změn. Tento řetězec může vypadat například jako na obrázku 9.



Obrázek 9: Zřetězení instancí stavů

Na tomto obrázku je příklad průchodu automatem o třech stavech. Vždy při aktivaci stavu dojde k vytvoření jeho nové instance. Protože stav 2 byl aktivní dvakrát (poprvé přechodem ze stavu 1 a podruhé přechodem ze stavu 3), má tedy v řetězci dvě instance. Každá instance si uchovává ukazatel na svůj původní stav. Naopak stav nemá žádné informace o svých instancích.

6.4 Konstrukce automatu z externího zdroje

Vzhledem k tomu, že vytvoření automatu na úrovni zapracování přímo do programu by bylo značně pracné a neflexibilní, je výhodné mít možnost definovat jeho strukturu z externího zdroje. Protože ho lze do jisté míry chápat jako strukturovanou jednotku, byl pro tento externí zdroj zvolen formát XML, kde uzly reprezentují jednotlivé prvky a parametry automatu. DTD tohoto XML souboru je v příloze 1.

Dokument se skládá z několika typů uzlů. Každý tento uzel specifikuje určité prvky, nebo vlastnosti výsledného automatu.

6.4.1 Prvek FA

Kořenovým prvkem celého XML souboru je `FA`. Má jediný atribut. Ten je povinný a obsahuje jméno automatu, který je brán jako hlavní. Tento hlavní automat pak bude sloužit jako návratová hodnota třídy zpracovávající XML soubor.

6.4.2 Prvek MACHINE

Uvnitř kořenového uzlu se nacházejí uzly `MACHINE`. Ty obsahují definice jednotlivých automatů, z nichž se následně poskládá výsledek. Každý tento automat musí být pojmenován, to sice nemá na výsledný automat žádný vliv, ale děje se čistě z důvodu sestavení. Prvek musí mít nastaven atribut STARTER určující stav, který bude použit jako výchozí. Obsahem uzlu `MACHINE` jsou elementy definující stavy a přechody mezi stavy.

6.4.3 Prvek STATE

Stav automatu je definovaný pomocí elementu `STATE`. Stejně jako automat musí být tento prvek pojmenovaný, aby bylo možné stavy jednoznačně identifikovat. Tato identifikace se použije pro pospojování stavů pomocí přechodů. Uvnitř uzlu stavu může být zanořený `PROCESSOR`.

6.4.4 Prvek PROCESSOR

Prvek `PROCESSOR` oznamuje konstrukční třídě, že stav bude generovat výstupní data. Jeho součástí je atribut `CLASS` obsahující identifikátor potomka třídy `Processor`. Protože může být potřeba instanci této třídy nastavit, je toto nastavení umožněno prostřednictvím elementů `PARAM` uvnitř prvku `PROCESSOR`.

6.4.5 Prvek PARAM

Prvky `PARAM` se vyskytují ve všech elementech, jejichž programové vyjádření je možné nějakým způsobem parametrizovat (nastavují třídy, které jsou potomekm třídy `ParamLoadable`). Parametr obsahuje dva povinné atributy, a sice `NAME` a `VALUE`. Obsahem `NAME` je jméno nastavovaného parametru a `VALUE` obsahuje jeho hodnotu.

6.4.6 Prvek STATEAGREGATED

Po popsání prvku `STATE` a jeho obsahu, můžeme přistoupit k popisu `STATEAGREGATED`, který je jeho rozšířením a reprezentuje zanořený automat (zásobník). Tento prvek je naprosto totožný se `STATE`, ale na rozdíl od něj neobsahuje potomka `PROCESSOR`, nýbrž uzel jménem `ASSEMBLER`.

6.4.7 Prvek ASSEMBLER

Tento prvek nahrazuje `PROCESSOR` u stavu s agregovaným automatem. Stejně jako `PROCESSOR` může obsahovat parametry, které nastavují jeho chování. Jak bylo psáno v kapitole 6.3.1, třída Assembler, popisovaná tímto prvkem slouží ke vložení dat načtených zanořeným automatem do dat generovaných automatem o úroveň výše.

6.4.8 Prvek TRANSITION

Stavy, které jsou vytvořeny a nastaveny je potřeba propojit přechody. K popisu přechodů slouží uzel typu `TRANSITION`. Má dva povinné atributy, jejichž hodnotou je jméno uzlu výchozího, nebo cílového. Na rozdíl od stavů má přechod mnohem více nastavení.

6.4.9 Prvek LOADER

Jediným povinným prvkem je `LOADER`, s jehož pomocí je nastavována třída typu `Loader`. Může opět obsahovat nastavení pomocí parametrů. Jeho jediným atributem je `CLASS`, který identifikuje typ třídy načítající data.

6.4.10 Prvek FILTER

Aby data načtená Loaderem mohla být filtrována, je potřeba vytvořit a nastavit filtry. Toto nastavení provádí prvek `FILTER`. Těchto prvků, může být obsaženo v definici přechodu několik (nebo i žádný). Stejně jako `LOADER` obsahuje atribut `CLASS` a může obsahovat potomky typu `PARAM`.

6.4.11 Prvek TEST

Posledním z rodiny prvků je `TEST`, který obsahuje definici testů vstupních dat. Kromě jiného jména značky je jinak naprosto totožný s prvkem `FILTER`.

6.4.12 Zpracování souboru definic

Jak bylo uvedeno výše, program umožňuje načtení automatu ze souboru. Než je toto načítání zahájeno, je nutné nastavit výrobní (factory) třídy pro některé složky automatu. Těmito složkami jsou:

- loadery
- testy
- filtry
- procesory
- assemblery

Tyto specializované třídy slouží k vytvoření instancí tříd na základě identifikačního řetězce. Instance mohou být buď čisté bez modifikací, nebo i přednastavené, například na často používané hodnoty. V okamžiku, kdy jsou tyto třídy nastaveny, může program přistoupit k načtení XML souboru.

Pro zpracování zdrojových dat byla původně zvažována knihovna LibXML++. Od ní ale bylo nakonec upuštěno, protože má několik závislostí na knihovnu Glib, které by mohly způsobit problém při portování výsledného projektu pro platformu Windows případně Mac OS.

Pro načtení dokumentu s definicemi do paměti byla nakonec vybrána open source knihovna PugiXML. Ta umožňuje objektově orientovanou práci s DOM načteného souboru, a to včetně vyhledávání a pohybu v dokumentu pomocí výrazů Xpath.

Na rozdíl od původní knihovny LibXML++ je práce s DOM značně zjednodušena a neposkytuje některé vlastnosti, například práci s vícebajtovým kódováním (UTF-8 a podobně) nebo načítání dokumentu z proudu dat. Značnou výhodou je ovšem velikost knihovny. Celá tato knihovna se skládá ze tří souborů, které je možné přímo přidat do projektu bez nutnosti linkovat externí zkompilované SO nebo DDL soubory.

Zpracování vstupních dat probíhá ve dvou fázích. V první fázi se projde seznam automatů definovaných v souboru a vytvoří se jejich prázdné instance. Tento krok je nutný, protože při zanořování automatů může vzniknout situace, kdy bude potřeba použít automat, který ještě nebyl načten. Pokud by byl vyžadován dosud neexistující automat, vedlo by to nutně vedlo ke komplikacím, nebo v horším případě k pádu celého programu. Ve druhé fázi se seznam automatů prochází znovu, ale tentokrát se u každého automatu zpracuje i obsah a vytvoří se jeho vnitřní struktura. Po dokončení druhé fáze je vrácen ukazatel na automat, jenž je označen jako hlavní. Tento automat je potom nastaven a připraven k okamžitému použití.

7 POPIS DATABÁZE UVNITŘ PROGRAMU

Databázové struktury, které načte automat popsaný v kapitole 6 je potřeba během zpracování programem nějakým způsobem uchovávat. Problematická je ale potřeba uchovat rozdílné struktury tak, aby práce s nimi byla sjednocená, a aby v určitých částech programu využívaly jednotné rozhraní. Dalším požadavkem na tyto třídy je možnost převedení celé sestavy v případě potřeby zpět na SQL kód, případně do přenosového formátu. Tyto operace musí být proveditelné i pro jednotlivé části struktury (klíč, sloupec a podobně), aby bylo možné generovat příkazy ALTER. Bylo tedy rozhodnuto, že všechny tyto třídy musí mít společného předka, který poskytuje toto společné rozhraní.

Problém může nastat na začátku zpracování SQL kódu, když ještě nevíme, o jakou strukturu nejvyššího řádu (tabulka, pohled,...) se jedná. To je možné vyřešit buď obalovou třídou, která bude schopna „přepínat“ tyto struktury, nebo přímo na úrovni automatu, kdy se vytvoří daná struktura až ve chvíli, kdy je jisté, o který typ se jedná. Toto řešení je mnohem jednodušší.

7.1 Třídy popisu

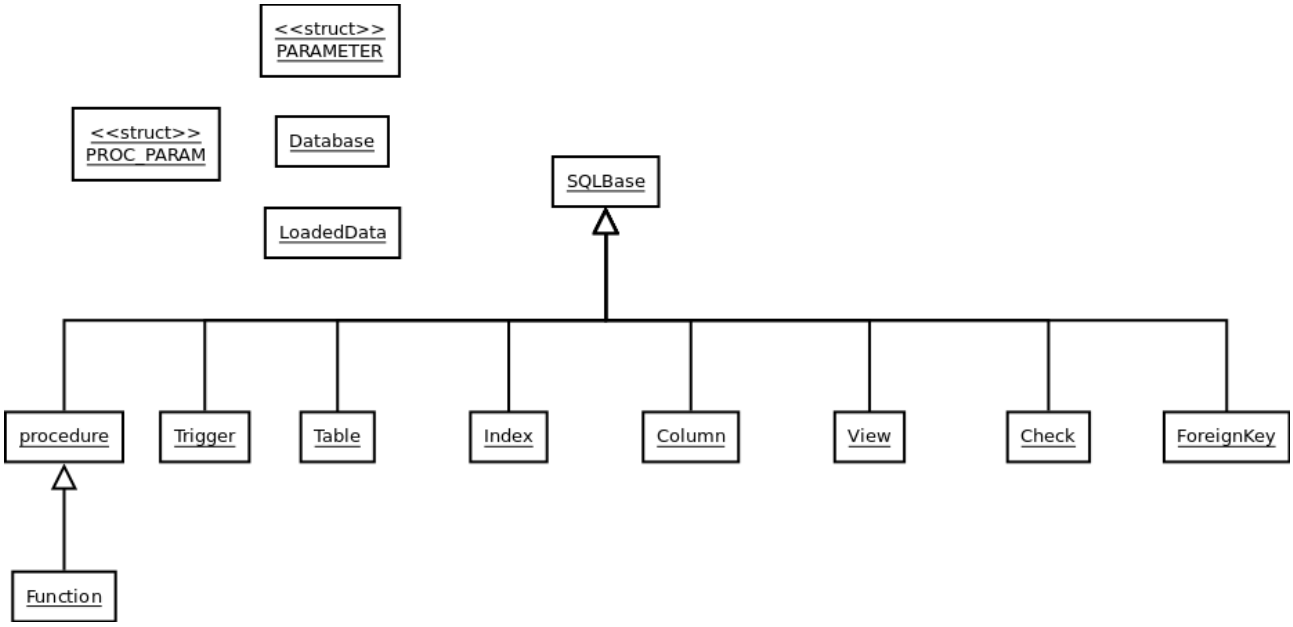
Z toho tedy vyplývá, že bude existovat několik typů tříd sloužících pro agregaci s různým stupněm zanoření. Podle tohoto stupně můžeme tedy třídy v základu rozdělit na obalové, strukturální a popisové. Obalová třída, o níž bylo psáno výše, je pouze jedna a slouží pro uchování prvků databáze. Strukturální třídy reprezentují vlastní prvky, které jsou „viditelné“ na úrovni celé databáze. Jsou to tedy tabulky, pohledy, cizí klíče a podobně. V hierarchii zanoření nejnižší třídy jsou třídy opisové. Budou reprezentovat:

- sloupec
- klíč (index, primární klíč atd.)
- cizí klíč

Obal, databáze, strukturální třída a skupina popisových tříd (brána jako celek) mají jasnou hierarchii zanoření. Obalová (databázová) třída obsahuje strukturální třídy a ve struktuře jsou zanořeny třídy popisu.

Programovou hierarchii tříd popisuje obrázek 10.

Tento model naznačuje, že sloupec tabulky, který je používán indexem nebo cizím klíčem, musí být obsažen také v instancích těchto tříd. Tento odkaz existuje z důvodu zachování integrity dat a zabránění vzniku neplatných referencí na neexistující sloupce, tabulky a podobně. Vznik těchto neplatných vazeb by mohl být zapříčiněn například přímým vstupem z programu nebo načtením chybného zdrojového souboru a vedlo by k následnému neplatnému SQL dotazu.



Obrázek 10: Hierarchie tříd popisu databáze

V následujícím textu se nachází stručný popis tříd. Věnuje se pouze základním funkcím a úloze dané třídy v modelu. Kompletní popis je dostupný v dokumentaci knihovny přímo v programu.

7.1.1 Struktura PARAMETER

Než bude přistoupeno k popisu jednotlivých tříd, je potřeba popsat strukturu `PARAMETER`, která obsahuje pouze dvě hodnoty:

- `std::string value`
- `bool switcher`

Hodnota `value` uchovává hodnotu daného parametru pouze v případě, že platí `switcher = FALSE`. Pokud je `switcher` nastaven na hodnotu `TRUE`, pak je `value` ignorována a daný parametr je pouze přepínačem (lze využít například pro indikaci dočasných tabulek a podobně).

7.1.2 Třída SQLBase

Výchozím rodičem pro všechny ostatní třídy, kromě `LoadedData` a `Database`, je třída `SQLBase`. Tato třída je potomkem `GraphNode` a obsahuje pouze základní vlastnosti a metody, které jsou společné všem dalším prvkům popisující databázi. Jedná se zejména o jméno prvku, komentář a pak parametry, jež je možné pomocí páru klíč-hodnota ukládat a měnit. K reprezentaci těchto hodnot slouží datový typ `ParamMap` definovaný pomocí typu `std::map`, kde klíčem je řetězec `std::string` a hodnotou je datový typ `PARAMETER` popsaný v 7.1.1.

Pro manipulaci s parametry slouží metody `createParameter()`, `deleteParameter()`, `getParameters()` a `isParameter()`.

První jmenovaná metoda vytvoří parametr daného jména a vrací referenci na novou instanci struktury `PARAMETER`. Další metoda v pořadí, `deleteParameter()`, smaže parametr, který je definován svým jménem. Metoda `getParameters()` vrací přímo referenci na instanci `ParamMap`, kde jsou všechny parametry prvku zapsány. Poslední metoda, `isParameter()` vrací TRUE, pokud parametr jména předaného v parametru existuje. V opačném případě vrací FALSE.

Pro práci s uzly grafu poskytuje vzhledem k třídě `GraphNode` navíc metodu `getParent()`. Protože prvky v databázi jsou vůči sobě vždy v pevně definovaném vztahu, je možné vždy zjistit jejich vlastníka. U prvků nejvyšší úrovně (pohled, funkce, procedura a tabulka) vrací tato metoda vždy NULL, protože žádného předka nemají. Ovšem u prvků, které je možné zanořovat (sloupce, indexy, atd.), je návratovou hodnotou ukazatel na mateřský prvek. O problematice reprezentace závislostí v paměti pojednává kapitola 7.2.

7.1.3 Třída Database

Nejvyšší třídou v hierarchii zanoření je `Database`. Slouží k uchování seznamu tabulek, pohledů a dalších prvků na úrovni databáze. Pomáhá udržovat integritu databáze a hlídá existenci tabulek v cizích klíčích. Obsahuje seznam tabulek, pohledů a cizích klíčů, jež se v tabulkách vyskytují.

Metody této třídy jsou určeny zejména pro manipulaci s tabulkami a cizími klíči. V případě tabulek umožňuje jejich tvorbu i úpravu. U cizích klíčů lze získat pouze seznam, vytvářet je pomocí tohoto objektu přímo nelze.

Metody sloužící k filtrování elementů databáze jsou vždy dvě pro každý typ prvku. Jedna metoda filtruje a vrací kompletní seznam těchto prvků a je bezparametrická. Je vždy označena anglickým názvem dané databázové struktury, který je v množném čísle. Tedy například `tables()`, `procedures()` a podobně.

Druhým typem filtračních metod jsou ty, které vrací konkrétní prvek a mají jeden parametr typu `std::string`. V případě, že prvek nenaleznou, tak vrací hodnotu `NULL`. Tyto metody pracují tak, že zavolají svoji variantu vracející kompletní seznam prvků a pak v tomto seznamu hledají prvek s požadovaným jménem.

7.1.4 Třída Table

Tabulku popisuje třída `Table`. Tato třída může obsahovat několik zanořených členů, které definují její popis, a ze kterých bude následně vygenerována hlavní část informací o struktuře tabulky:

- `Column` – sloupec tabulky
- `Index` – index dat
- `ForeignKey` – cizí klíč (vazba na jinou tabulku)
- `Check` – kontrolní kód

Pro práci s tabulkovými prvky poskytuje třída paletu metod, které jsou celkem čtyř typů:

1. vytvářecí
2. mazací
3. filtrační
4. dotazovací

Metody, které spadají do první kategorie slouží k vytváření těchto podřízených prvků. Jejich návratová hodnota je ukazatel na nový prvek a přebírají vždy alespoň jeden parametr, kterým je jméno nového prvku. Je nutné poznamenat, že třída nekontroluje konflikty jmen. To je ponecháno na databázi a jiných částech programu. Některé prvky potřebují pro vytvoření více parametrů. Jedná se o indexy a cizí klíče. Index požaduje pro konstrukci navíc seznam sloupců tabulky a cizí klíč navíc ještě ukazatel na referenční tabulku a seznam sloupců z referenční tabulky.

Kategorie metod pro mazání vyžaduje vždy pouze jediný parametr, a sice jméno prvku, a nemají žádnou návratovou hodnotu. Při mazání prvku jsou vždy smazány i všechny jeho závislosti. Tedy například při smazání sloupce se kontroluje, jestli by po něm nezůstal index, který na žádný sloupec už nemá vazbu. Pokud by se tak stalo, pak se index smaže taky. Toto kaskádovité mazání prvků zajišťuje destruktor mazané třídy.

Stejně jako u databáze se dělí filtrační metody na další dvě skupiny. Jednak hromadné, kdy jsou vyfiltrovány všechny prvky daného typu, a jednak konkrétní, které filtrují jeden konkrétní prvek daného typu podle jeho jména.

Poslední kategorie je skupina dotazovacích metod. Tyto metody vrací logickou hodnotu a mají jako parametr název hledaného prvku. Jejich účelem je zjistit, zda se hledaný prvek v tabulce nachází.

7.1.5 Třída View

Druhá struktura, která v databázi spadá do kategorie nejvyšší úrovně je pohled reprezentovaný třídou `View`. Tato třída obsahuje kromě jména a několika parametrů (implementované na úrovni `SQLBase`) pouze textový řetězec obsahující definici pohledu a slouží pouze k uchování definice pohledu, nijak nekontroluje správnost kódu. Ověření správnosti (a případné přenositelnosti) je necháno na uživateli, který program, jež je výstupem této práce, používá.

7.1.6 Třída Column

Hlavním prvkem, který dělá tabulku schopnou ukládat data je sloupec. Třída pro popis sloupce se jmenuje `Column` a kromě jména uchovává několik dalších příznaků a informací.

Pro vytvoření sloupce vyžaduje konstruktor jméno nového sloupce a datový typ. Datový typ je uložen v řetězcové vlastnosti dataType. Jeho velikost není ve třídě nijak odlišena, a pokud má být zapsána, tak jako součást jména datového typu. Řetězec uložený v této vlastnosti může být tedy například jak `INT`, tak i `INT(11)`.

Třída obsahuje ještě následují informace a příznaky, které určují nepovinné parametry sloupce. Těmito parametry jsou:

- příznak, jestli hodnota může být prázdná (`NULL` nebo `NOT NULL`)
- příznak, jestli sloupec používá výchozí hodnotu
- výchozí hodnota
- extra hodnota, která může být použita například pro zaznamenání automatické inkrementace v MySQL

A následně přepínače a parametry, které se týkají konkrétního datového typu. Může to být například typ kódování a porovnávání u řetězců, „bezznaménkovost“ u čísel a podobně. Tyto parametry jsou spravovány na úrovni `SQLBase` v seznamu dodatečných parametrů.

Třída poskytuje také metodu pro zjištění rodičovské tabulky. Ta uvnitř volá rodičovu metodu `getParent()`, jejíž návratovou hodnotu pouze přetypuje na ukazatel na typ `Table`.

Dále obsahuje metodu getIndexes(), která vrací seznam indexů, v nichž je sloupec použit, a metodu getForeigns() vracející seznam cizích klíčů. Tato metoda přebírá jeden parametr, s jehož pomocí lze odlišit cizí klíče rodičovské tabulky a cizí klíče, které mají na rodičovskou tabulku pouze referenci.

Poslední metoda, kterou sloupec poskytuje slouží k navigaci uvnitř tabulky. Její jméno je `prev()` a vrací ukazatel na sloupec, nacházející se před sloupem, s jehož instancí pracujeme. Tato nebyla součástí původního návrhu a byla dopsána až dodatečně se vzniklou potřebou zachovat pořadí sloupců ve změnách.

7.1.7 Třída Index

Pokud se ve sloupci často hledá, měl by být nad ním sestaven vyhledávací index, který toto hledání značně urychluje. Tyto struktury popisuje třída Index, jež v sobě uchovává typ indexu a seznam indexovaných sloupců.

Typ indexu musí být jedním z tohoto seznamu:

- `KEY` – dva možné zápisy obyčejného indexu

- `UNIQUE` – prvky tohoto indexu musí být v celé tabulce unikátní
- `PRIMARY KEY` – primární klíč tabulky

Fulltextové indexy zatím podporovány nejsou.

Stejně jako třída sloupce, obsahuje Index ve svém rozhraní metodu `getTable()`, která vrací rodičovskou tabulku. Ke zjištění seznamu sloupců v indexu je nadefinována metoda `getColumns()` vracející seznam typu `ColumnList`.

7.1.8 Třída Check

Jak bylo psáno v kapitole 4, vkládaná data do tabulky je možné kontrolovat pomocí testovacích podmínek. K reprezentaci takovýchto podmínek slouží třída `Check`, která obsahuje pouze řetězec s kontrolními podmínkami. Třída nezaručuje existenci sloupců a operátorů použitých v tomto kódu. Slouží jen jako kontejner na tento kód. Ve svém rozhraní má jedinou metodu, a sice `getTable()`.

7.1.9 Třída Trigger

Podobně jako `Check`, obsahuje třída Trigger výkonný kód triggeru, kromě toho ale ještě informaci o tom, kdy a při jaké operaci se má trigger spustit.

Typ operace je určen hodnou z výčtu `TRIGGER_OPERATION`:

- `INSERT` – operace vložení nového (nových) řádků
- `UPDATE` – aktualizace dat
- `DELETE` – smazání řádků

Čas je definován pomocí enumeračního typu `TRIGGER_TIME`:

- `BEFORE` – trigger se spouští před operací
- `AFTER` – trigger se spouští po operaci

Stejně jako ostatní tabulkové prvky, obsahuje také metodu `getTable()`.

7.1.10 Třída Procedure

Pro uchování informací o procedurách existuje třída `Procedure`. Stejně jako třída `Check` nebo `Trigger` uchovává pouze zdrojový kód procedury a neručí za jeho správnost. Oproti těmto třídám navíc ještě obsahuje informaci o přístupu k datům a seznam parametrů přijímaných procedurou.

Typ přístupu k datům je v paměti reprezentován výčtovým typem `DATA_ACCESS_MODS`, který je součástí veřejného rozhraní třídy a může nabývat hodnot:

- `DA_NOSQL`
- `DA_CONTAINSQL`
- `DA_READSQLDATA`
- `DA_MODIFYSQLDATA`

Tyto parametry jsou uloženy pomocí struktury `PROC_PARAM` v kontejneru `std::list`. Přístup a manipulace s nimi je zajištěna kolekcí těchto metod:

- `addParameter()` – přejímá jméno, datový typ a výchozí hodnotu a vytvoří nový parametr procedury
- `clearParameters()` – vymaže všechny parametry
- `dropParameter()` – smaže parametr, jehož jméno se shoduje se jménem, které je parametrem metody
- `hasParameter()` – má jediný parametr, který je řetězec obsahující jméno zkoumaného parametru. Návratová hodnota je `BOOL` a vrací `TRUE`, pokud parametr daného jména procedura obsahuje
- `setParameters()` – její parametr je stejného typu jako seznam parametrů uvnitř

třídy a slouží k hromadnému nastavení parametrů procedury, kterou třída reprezentuje

- `getParameter()` – vrací referenci na parametr podle jména, které je předáno metodě
- `getParameters()` – vrací kopii seznamu parametrů

Pro poslední dvě metody byla značně nevhodně zvolena jména, neboť přetěžují metody pro práci s parametry, které definuje rodičovská třída. Naštěstí se jedná o vadu pouze kosmetickou, protože nepoužívají stejnou návratovou hodnotu a nevznikl kvůli tomu žádný závažný problém.

7.1.11 Struktura PROC_PARAM

Struktura pro uchování informací o parametrech procedury respektive funkce obsahuje tři řetězcové hodnoty a jednu výčtovou:

- `name` – jméno parametru
- `dataType` – datový typ parametru
- `defVal` – výchozí hodnota parametru
- `paramType` – typ parametru, který je určen výčtovým typem `PROC_PARAM_TYPE` a nese informaci o směru parametru (vstupní, výstupní nebo obojí)

Hodnoty `name` a `dataType` jsou pro správnou funkci vygenerovaného SQL povinné.

7.1.12 Třída Function

Informace o funkcích uchovává třída `Function`, jež je potomkem `Procedure`. Jediná věc, o kterou svého předka rozšiřuje je hodnota obsahující návratový datový typ.

7.1.13 Třída ForeignKey

Poslední třídou z reprezentace databázových prvků je cizí klíč, tedy třída `ForeignKey`. Ta obsahuje kromě lokálních sloupců také odkaz na další tabulku (tato tabulka může být i stejná jako ta, kde se cizí klíč nachází) a sloupce v této tabulce. Musí platit, že v okamžiku exportu se počet sloupců v místní tabulce musí rovnat počtu sloupců v cílové tabulce. Jinak by databázový server při zpracování definičních dat opět nahlásil chybu.

Má dvě vlastnosti, a sice `onDelete` a `onUpdate` uchovávající informaci o chování cizího klíče, pokud se něco stane s referenční tabulkou, respektive sloupcem. Tyto dvě hodnoty jsou pro jednoduchost uchovány pouze jako řetězec. Dále má rozhraní pro práce se sloupci (místními i referenčními), metody pro zjištění a nastavení referenční tabulky a standardní metodu `getTable()`, která vrací rodičovskou tabulku.

7.1.14 Třída LoadedData

Speciální třídou je třída `LoadedData`. Instance této třídy slouží jako mezičlánek mezi SQL kódem získaným z databázového serveru a reprezentací pomocí výše popsaných tříd. Jako jediná třída není vázána na `SQLBase` a je používána pro uchování dat, které zpracuje konečný automat popsaný v kapitolo 6.

7.2 Implementace modelu

Výše popsané třídy budou obsaženy, stejně jako třídy automatu, v samostatné knihovně. Jednotlivé třídy budou uvnitř vlastního namespace, aby při dalším použití mimo tento projekt nedocházelo ke kolizím jmen tříd a typů.

Jednotlivé prvky se budou předávat zpravidla pomocí ukazatelů a kopírovat se uvnitř obalových instancí. Po přidání objektu z vnějšku bude tedy možné původní instanci odstranit z paměti. To zjednoduší kontrolu nad pamětí existujících objektů a nebude tedy hrozit opomenutí dealokace použitých zdrojů.

7.2.1 Reprezentace závislostí v paměti

Kvůli reprezentaci pomocí ukazatelů bude nutné uchovávat závislosti mezi objekty oboustranně. To znamená, že například objekt, který reprezentuje index bude mít v sobě ukazatel na sloupce, které jsou v něm zahrnuty, a stejně tak tyto sloupce budou obsahovat ukazatel na tento index. Díky tomuto řešení bude snadnější hlídat vnitřní integritu celé struktury databáze. Předejde se tak tomu, aby sloupec obsahoval neexistující index a naopak klíč indexoval smazaný sloupec.

Pro realizaci tohoto modelu závislostí se nabízejí zejména tři způsoby:

- zpřístupnění referencí jiné třídě
- ohlašování změny prvku pomocí posílání zpráv mezi instancemi
- křížové volání metod odebrání (odebíraný prvek ohlásí všem svým závislostem, že je rušen)
- reprezentace databáze pomocí grafu

Protože je celý projekt programován v C++, je možné použít pro první možnost konstrukci `friend`. Tato konstrukce umožňuje přistupovat k privátním prvkům jedné třídy z třídy jiné a toto zpřístupnění je možné omezit pouze na určité metody.

Použití konstrukce `friend` je ovšem z hlediska OOP poměrně nečisté, neboť se tím porušuje zapouzdření tříd. Další nevýhoda tohoto způsobu je čistě estetická kvůli znepřehlednění kódu. Tato možnost je tedy zavržena, a to zejména z prvního důvodu.

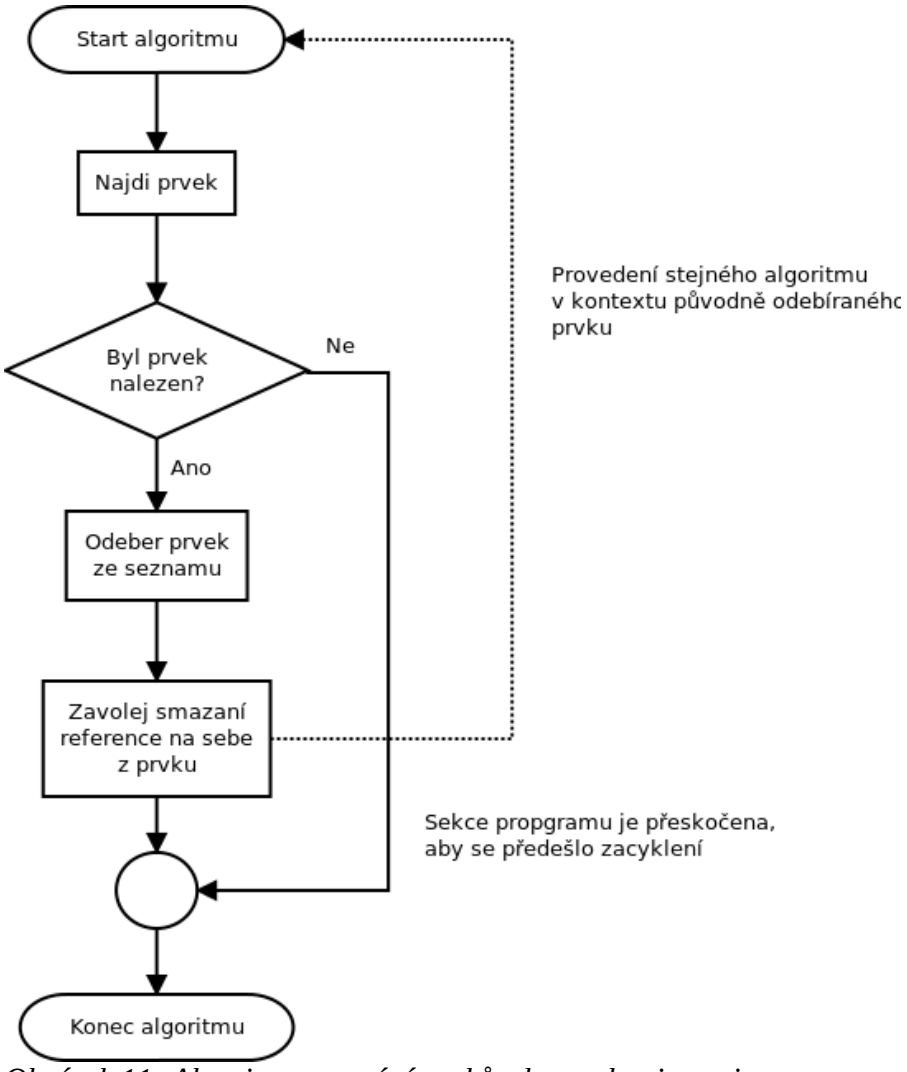
Druhou zde zmíněnou možností je použití specializované bázové třídy, která bude umožňovat zasílání zpráv mezi instancemi. Jedná se o robustní řešení, které má spoustu výhod, jako možnost použití při více vláknovém zpracování nebo nasazení paralelní práce na více fyzických strojích. Nevýhoda vychází ze sekvenční struktury tohoto konkrétního programu, je zbytečné posílání zpráv více objektům přes zprostředkovatelskou třídu. Použití této třídy jako bázové by opět vedlo ke znepřehlednění kódu a zbytečným komplikacím během implementace.

Třetí návrh, tedy křížové volání metod, je značně univerzálnější a nezávislý na programovacím jazyku. Nevýhodou je ale možnost vzniku zacyklení při špatné implementaci a větší režie, která vzniká při tomto křížovém volání. Příklad tohoto problému lze uvést z reálného světa: mějme třídu Osoba, která vyjadřuje konkrétního člověka, a třídu Vztah vyjadřující vztah mezi dvěma různými lidmi. Uveďme, že každá osoba ví, jaké vztahy má navázány (instance obsahuje ukazatele na tyto vztahy) a stejně tak každá instance vztahu ví, na které dvě osoby se váže. Nyní uvažujme, že máme dvě instance třídy Osoba, například Petr a Jana, a tyto dvě instance jsou manželé, a jsou tedy spojeny instancí třídy Vztah. V jejich manželství ovšem dojde k problémům a jeden z nich chce požádat o rozvod. V případě, že v implementaci „rozvodu“ byla chyba, mohlo by dojít k zacyklení, které by šlo popsat jako hádku, kdo podá žádost o rozvod.

Diagram popisující jednu z možností, jak správně implementovat tento algoritmus, je na obrázku 11.

Jak bylo naznačeno na příkladu použití algoritmu odebírání, je možné s tímto algoritmem použít návrh číslo čtyři – reprezentaci pomocí grafu (v příkladu osoby odpovídají uzlům a vztahy hranám). V tomto grafu uzly vyjadřují jednotlivé prvky databáze (tabulky, sloupce, index) a hrany vyjadřují závislost mezi nimi. Pokud se objekty grafu implementují pro tento specifický účel, dají se rozlišit typy uzlů (typy databázových struktur a jejich vlastností) pomocí příznaku, a sestavovat tak podgrafy reprezentující určitý typ závislosti. Z uživatelského hlediska budou tyto podgrafy vyhledávací filtry schopné zobrazit pouze indexy tabulek, síť cizích klíčů apod.

Příkladem kombinace možnosti 3 a 4 může být odebrání uzlu reprezentující sloupec tabulky. Po odebrání sloupce tabulky se vyfiltrují indexy (klíče) této tabulky společně s hranami na sloupce, které jsou indexovány. V případě, že některý uzel je izolovaný (není zde žádná hrana na uzel sloupce), pak se klíč může smazat.



Obrázek 11: Algoritmus mazání prvků s kontrolou integrity

S ohledem na výše uvedené výhody a nevýhody jednotlivých možností byla vybrána třetí možnost v kombinaci s reprezentací pomocí grafu. Pro reprezentaci závislostí byl použit stejný graf, s jehož pomocí je sestavena vnitřní struktura konečného automatu. Pro bližší informace o třídách a stavbě grafu viz kapitolu 6.2 na straně 31.

7.2.2 Provázání grafu s popisem databáze

Vzhledem k požadavkům na třídu `Database`, bude tato třída potomkem třídy `Graph`. Její metody pro filtraci uzlů dle jednotlivých typů databázových prvků budou využívat rozhraní svého předka. Aby bylo toto filtrování možné, je třeba rozlišit typy uzlů a v některých případech i hran. K tomuto účelu bylo využito enumeračních typů s hodnotami nastavenými tak, aby bylo možné realizovat bitový součin a součet.

Pro odlišení uzlů je definován výčtový typ `NODE_TYPES` s hodnotami:

- `NODE_TABLE = 1`
- `NODE_COLUMN = 2`
- `NODE_VIEW = 4`
- `NODE_FOREIGN_KEY = 8`
- `NODE_INDEX = 16`

Pro odlišení hran je pak typ `MATE_TYPES`

- MATE_MAJOR_PARENT = 1 (hrana na hlavního předka v grafu, například z tabulky na databázi)
- MATE_INDEXED = 2 (hrana z indexu na indexovaný sloupec)
- MATE_FOREIGN_REF = 4 (hrana z cizího klíče na referencovanou tabulku nebo sloupec, tzn. za klíčovým slovem REFERENCE)
- MATE_FOREIGN_TARGET = 8 (hrana z cizího klíče na tabulku nebo sloupec, nad kterým je tento klíč vytvořen)

Ostatní prvky databáze (sloupce, tabulky,...) jsou pak potomkem třídy GraphNode. V konstruktoru těchto prvků se předává ukazatel na instanci databáze a jméno databázového prvku. Uvnitř konstruktoru se poté nastaví příslušné ohodnocení uzlu dle typu vytvářeného prvku (hodnota z NODE_TYPES). Vytváření těchto prvků se provádí pomocí nadřazených tříd metodami, které po vytvoření instance vytvoří také příslušnou hranu s ohodnocením typu vazby mezi těmito prvky. Pokud by bylo potřeba dělat specifické operace, nepodporované rozhraním těchto tříd, je možné je provést mimo metody, ale některá nastavení je nutné udělat ručně.

7.3 Reprezentace struktury v souborech

Pro potřeby uchování informací o struktuře databáze a změn v ní je třeba ukládat data do souboru. Jako jeho formát byl vybrán XML, protože je snadno zpracovatelný pomocí široké palety knihoven a je přímo určen pro uchování strukturovaných dat, což databáze je.

Tyto XML soubory se budou skládat z prvků:

- DATABASE
- TABLE
- VIEW
- FOREIGNKEY
- COLUMN
- KEY
- CHECK
- FUNCTION
- PROCEDURE
- TRIGGER
- REFCOLUMN
- TARCOLUMN
- PARAMETER
- PROCPARAM
- CODE

Všechny prvky mají jeden společný nepovinný atribut. Tímto atributem je řetězcová hodnota comment, pomocí které je možné okomentovat daný element.

V následujícím textu budou popsány jednotlivé elementy: jejich význam, pozice v dokumentu a atributy, které mohou obsahovat.

7.3.1 Prvek DATABASE

Kořenovým prvkem je DATABASE, nemá žádné atributy a slouží pouze jako obalový element pro celý dokument.

Možní potomci jsou:

- TABLE
- VIEW
- PROCEDURE
- FUNCTION

7.3.2 Prvek TABLE

Prvek `TABLE` reprezentuje tabulku. Vyskytovat se může jako přímý následník buď prvku `DATABASE`, nebo prvků `ADD`, `REMOVE` a `CHANGE`, které jsou přímými následníky prvku `DATABASE`.

Tento prvek má atributy atributy:

- `name` (řetězec) – jméno tabulky
- `storagetype` (řetězec) – typ úložiště
- `collate` (řetězec) – porovnávání
- `charset` (řetězec) – výchozí znaková sada

Uvnitř elementu TABLE mohou být zanořené prvky:

- `COLUMN`
- `INDEX`
- `TRIGGER`
- `FOREIGNKEY`

7.3.3 Prvek VIEW

Prvek `VIEW` má stejné možnosti výskytu jako `TABLE`, který je popsaný v předchozí podkapitole. Na rozdíl od něj ale má libovolné atributy, a to z toho důvodu, aby se mohla definovat nestandardní rozšíření databázových systémů (například u MySQL se jedná o atribut `definer`, `algorithm` a podobně). Jediný povinný atribut je řetězec `name`, který obsahuje jméno pohledu. Prvek neobsahuje žádné další zanořené elementy. Pouze sekvenci CODE, kde je obsažena definice pohledu.

7.3.4 Prvek FOREIGNKEY

Posledním prvkem, který reprezentuje strukturu a může se vyskytovat v kořenovém elementu je `FOREIGNKEY` definující cizí klíč. Jak už bylo řečeno, může se vyskytovat jako přímý následník elementu TABLE.

FOREIGNKEY musí obsahovat tyto atributy:

- name (řetězec) – jméno klíče
- target (řetězec) - jméno tabulky, ze které klíč vychází
- reference (řetězec) – jméno referenční tabulky

Dále pak má dva nepovinné atributy definující akce, které se mají provést v případě určitých událostí:

- ondelete (řetězec) – událost, pokud se rodičovský záznam smaže
- onupdate (řetězec) – událost, pokud dojde ke změně referencovaných sloupců

Jako potomci tohoto prvku mohou být pouze elementy:

- REFCOLUMN
- TARCOLUMN

7.3.5 Prvek COLUMN

Pro reprezentaci sloupce tabulky slouží prvek `COLUMN`, který se může nacházet uvnitř prvků TABLE, `ADD`, `REMOVE` nebo `CHANGE`. Jeho povinnými atributy jsou:

- `name` (řetězec) – jméno sloupce
- `datatype` (řetězec) – datový typ i s případným rozsahem

Sloupec pak může definovat několik nepovinných atributů upřesňujících jeho strukturu:

- `notnull` (logická hodnota) – přepínač možnosti prázdné hodnoty
- `extra` (řetězec) – extra nastavení (například auto_increment)
- `default` (řetězec) – výchozí hodnota

7.3.6 Prvek KEY

Indexy sloupců jsou v XML souboru vyjádřeny pomocí elementu KEY. Ten je zanořen v kontejneru TABLE nebo v některém modifikačním kontejneru a obsahuje hodnoty, které jsou potřeba pro vytvoření klíče. Jedná se o atributy:

- name (řetězec) – jméno indexu
- type (řetězec) – typ indexování (například BTREE a podobně)

Uvnitř uzlu KEY se může vyskytovat pouze element TARCOLUMN odkazující na indexované sloupce.

7.3.7 Prvek CHECK

Kontejnerem pro kontrolní kód vkládaných dat je prvek CHECK. Nedefinuje žádný parametr a jeho obsahem je sekce CODE se SQL kódem, který se použije pro kontrolní kód při manipulaci s daty tabulky.

7.3.8 Prvek PROCEDURE

Definice procedury je ukládána v prvku PROCEDURE, který obsahuje pouze atribut name se jménem procedury. Ostatní informace nesou následníci tohoto uzlu. Těmito následníky jsou:

- PROCPARAM – obsahuje informace o parametru
- CODE – obsahuje výkonný SQL kód procedury

7.3.9 Prvek FUNCTION

Velmi podobný prvek k proceduře je FUNCTION. Je téměř totožný. Jediný rozdíl oproti proceduře je v atributu navíc. Tím je DATATYPE, který program informuje o návratovém typu funkce.

7.3.10 Prvek PROCPARAM

Aby bylo možné ukládat informace o parametrech procedur a funkcí, je definován element PROCPARAM. Všechny informace o parametru jsou uloženy v atributech tohoto prvku:

- NAME – jméno parametru
- DIRECTION – informace, zda se jedná o vstupní, výstupní nebo vstupně výstupní parametr
- DATATYPE – datový typ

7.3.11 Prvek TRIGGER

Informaci o podobě triggeru uchovává prvek TRIGGER s atributy:

- NAME – jméno triggeru
- TIME – čas, kdy se má trigger spustit
- OPERATION – operace s daty, která se má provést

Uvnitř tohoto uzlu se nachází jeden prvek CODE, jehož textový obsah je SQL kód prováděný triggerem.

7.3.12 Prvek TARCOLUMN

V kapitole 7.3.4 a 7.3.6 jsou popsány prvky, které k sobě vážou sloupce. Výčet sloupců tabulky, k nimž jsou tyto prvky vázány je realizován pomocí značky TARCOLUMN. Je to nepárový tag s jediným atributem NAME. Ten je povinný a jeho obsahem je jméno sloupce, který je v dané struktuře použit.

7.3.13 Prvek CODE

Některé prvky potřebují uchovávat zdrojový kód. Aby mohl být zdrojový kód uchován, je definován prvek `CODE`. Nemá žádný atribut a jeho obsahem je SQL kód, který je potřeba uchovat.

7.3.14 Prvek REFCOLUMN

Stejnou definici má i elementy typu `REFCOLUMN`. Obsah je naprosto stejný jako `TARCOLUMN`. Liší se pouze ve významu této značky. Zatímco `TARCOLUMN` ukazuje na sloupce tabulky, k níž nadřazený prvek patří, tak `REFCOLUMN` nese informace o sloupcích referenční (cizí) tabulky. Význam má tento prvek pouze pro nastavení cizích klíčů.

8 GENEROVÁNÍ REPREZENTACE DATABÁZE

Reprezentace databáze, popsaná v kapitole 7, musí být generována na základě skutečných struktur uvnitř databázového systému.

V některých případech je získání popisu relativně jednoduché. Jako příklad je možné uvést MySQL, která zpřístupňuje tyto informace ve virtuální databázi `information_schema` [20]. V ní jsou uloženy všechny informace o strukturách přehledně v tabulkách, ze kterých lze jednoduchými filtračními dotazy tato data získat. `Information_schema` byla přidána do MySQL až ve verzi 5.0.2 [21]. Vzhledem k tomu, že byla vydána 1. prosince 2004 [21], ale lze předpokládat, že starší verze než tato už nejsou používány, a nebudou tedy představovat žádný problém.

Bohužel některé další databázové systémy takto vstřícné nejsou. Pro zjištění datových struktur je potřeba zpracovat surový SQL kód. K tomuto procesu je použit konečný automat z kapitoly 6. Jako příklad je možné poukázat na SQLite, který tyto informace schraňuje v tabulce `sqli_master` [16] a nebo PostgreSQL, kde je nutné použít utilitu `pg_dump` Chyba: zdroj odkazu nenalezen.

Vzhledem k nedostatku času je z databázových adaptérů implementován pouze adaptér pro MySQL. Pro další rozvoj programu je pak na konstrukci ostatních adaptérů připraven konečný automat popsaný v kapitole 6.

8.1 Abstraktní adaptér

Aby bylo možné jednotně přistupovat k různým databázovým systémům, musí existovat jednoznačně definované rozhraní, které další třídy implementují. Tím je abstraktní třída `Adapter` obsahující čtyři ryze virtuální metody:

- `readData()`
- `writeData()`
- `writeDiff()`
- `setConfig()`

První metoda slouží k načítání dat z databáze. Má parametr, který je ukazatelem na prázdnou reprezentaci databáze (čili typ `Database*`), a tento ukazatel je zároveň návratová hodnota.

Druhá metoda dělá pravý opak. Přijímá opět parametr, jenž je ukazatelem na databázi a bude se zapisovat na server, ale na rozdíl od `readData()` nemá žádnou návratovou hodnotu (typ `void`).

Podobnou úlohu má i `writeDiff()`. Nezapisuje ale celou databázi, ale musí být schopna vygenerovat takový příkaz, který zapíše pouze změny v databázi.

Poslední metodou je `setConfig()`. Jak napovídá její název, slouží ke konfiguraci adaptéru. Jako parametr je předán XML uzel obsahující informaci o konfiguraci. Jméno uzlu je určeno statickou konstantou `NODE_PARAM` a názvy jeho atributů jsou obsaženy taktéž ve statických konstantách `ATTR_NAME` (jméno parametru) a `ATTR_VALUE` (hodnota parametru).

Často je potřeba roztřídit uzly reprezentující strukturu databáze podle typu operace, jež byla provedena. Z tohoto důvodu je připravena šablonová metoda `SortItems()`. Té je předán seznam prvků jednoho typu (například tabulky, pohledy, sloupce atd.) a nad elementy v tomto seznamu je provedeno třídění dle hodnoty `compareResult`.

Návratovou hodnotou šablonové metody je seznam (`std::vector`) obsahující prvky typu seznamu, který byl předán v parametru. Jednotlivé seznamy, tak jak jdou za sebou a obsahují prvky:

1. které byly přidány

2. které byly smazány
3. uvnitř kterých proběhla změna
4. které byly změněny
5. které jsou pouze logické, a se kterými nebylo nic provedeno

Tento výčet je v programu reprezentován enumeračním typem `SORTED_LIST_INDEX`. Tento enumerační typ je součástí třídy `Adapter`.

8.2 MySQL

Protože zpracování dat ze systému MySQL je nejjednodušší, bude popsán jako první a spolu s ním bude vysvětlen princip fungování třídy `MySQLAdapter` sloužící k načítání z MySQL.

Virtuální databáze `information_schema` obsahuje mnoho tabulek uchovávajících veškeré informace o strukturách, k nimž má přihlášený uživatel přístup. Pro analýzu těchto dat bude potřeba jen několik z nich:

- `COLUMNS` – sloupce tabulek
- `KEY_COLUMN_USAGE` – indexované sloupce
- `PARAMETERS` – parametry procedur a funkcí
- `REFERENTIAL_CONSTRAINTS` – informace o cizích klíčích
- `ROUTINES` – procedury a funkce
- `TABLES` – tabulky
- `TABLE_CONSTRAINTS` – indexy a cizí klíče
- `TRIGGERS` – triggery
- `VIEWS` – pohledy

Při načítání struktur se třída adaptéru nejprve pomocí metody `connect()` připojí k serveru a nastaví jako aktivní databázi `information_schema`. Pokud se připojení zdaří, pokračuje se vlastním načítání dat. Tato data jsou načítána v pořadí:

1. tabulky
2. pohledy
3. cizí klíče
4. triggery
5. procedury a funkce

Přestože jsou cizí klíče součástí tabulek, musí být načítány až ve chvíli, kdy jsou všechny tabulky vytvořené. Pokud by se tvořil cizí klíč s referencí na ještě neexistující tabulku, došlo by k chybě.

Při načítání tabulek se získávají informace z `TABLES`, `COLUMNS`, `TABLES_CONSTRAINTS` a `KEY_COLUMN_USAGE`. Nejprve se načtou a vytvoří sloupce tabulky, poté se vytvoří indexy a nakonec se přiřadí indexované sloupce k indexům. Při načítání indexů jsou vynechávány ty indexy, které mají v sloupci `CONSTRAINT_TYPE` (uchovává typ klíče [3]) nastavenu hodnotu `FOREIGN KEY`.

Oproti tabulkám je načítání pohledů jednodušší. Pohled je definován pouze svým jménem a výkonným kódem, který je formou `SELECT` dotazu uložen v samostatném sloupci. Tyto dvě hodnoty stačí načíst a z nich vytvořit objekt reprezentující pohled.

Po načtení pohledů přicházejí na řadu cizí klíče, načítané z tabulky `TABLE_CONSTRAINTS` s hodnotou `FOREIGN KEY` ve sloupci `CONSTRAINT_TYPE` a rozšiřující informace, které jsou potřeba pro úplnou definici cizího klíče, jejichž místo uložení je v tabulce `REFERENTIAL_CONSTRAINTS`. Po načtení potřebných informací se vytvoří instance cizího klíče a zapíší se místní a referenční sloupce.

Předposlední načítanou položkou jsou triggery. Zpracování každého řádku probíhá tak, že se zjistí tabulka triggeru, vytvoří se nový trigger a zapíší se potřebné hodnoty:

- čas provedení triggeru (před nebo po operaci)
- operace spouštějící trigger
- výkonný kód

Po zpracování a zanesení těchto informací se pokračuje na další záznam.

Na konec přijde na řadu zpracování procedur a funkcí. Protože tyto dvě struktury jsou si velmi podobné, probíhá jejich načítání společně. Nejprve se nastaví společné vlastnosti procedury i funkce (jméno, parametry, kód atd.) a poté se vyhodnotí, jestli se jedná o funkci. Pokud ano, načte se ještě návratový datový typ.

Po načtení všech struktur dojde k odpojení od databáze zavoláním metody `disconect()`.

Zápis databáze na server je velmi podobný. Při připojování k serveru se ovšem neaktivuje virtuální databáze, ale přímo databáze, do které se bude zapisovat. Než se provede samotný zápis dat, odešle se na server příkaz:

```
SET foreign_key_checks = 0
```

Příklad 21: Vypnutí kontroly cizích klíčů

Tento příkaz vypne kontrolu cizích klíčů a neintegrita dat je dočasně ignorována.

Po odeslání příkazu 21 jsou na serveru postupně vytvářeny datové struktury v pořadí:

1. tabulky
2. pohledy
3. triggery
4. procedury
5. funkce
6. cizí klíče

V tomto případě jsou procedury a funkce zdánlivě zapisovány odděleně. Ve skutečnosti ale obě metody volají jednu metodu, které předají jako parametr zpracovávanou funkci nebo proceduru společně s informací o typu rutiny.

Po dokončení tvorby struktur je odeslán na server příkaz velmi podobný příkazu 21, ale s tím rozdílem, že místo nuly nastavujeme jedničku.

Jakmile je příkaz odeslán, dojde k ukončení spojení se serverem a návrat z metody zapisující data.

Nejdůležitější funkcí adaptéru je ale schopnost zápisu rozdílových balíčků databází. Tvorba těchto balíčků je popsána v kapitole 9. Při zápisu změn je vždy volán příkaz 21, aby se zabránilo výjimkám vyvolaných z MySQL serveru vlivem neintegrity dat. Změny ve strukturách jsou zapisovány v pořadí:

1. tabulky
2. funkce
3. procedury
4. pohledy

Na rozdíl od čtení a zápisu celé databáze jsou zde zapisovány změny triggerů a cizích klíčů společně s tabulkami. Je to opět kvůli udržení integrity, ale v tomto případě integrity datových struktur (zabránění existence cizího klíče ukazujícího na neexistující tabulku a podobně).

Změny jsou většinou prováděny v pořadí:

1. mazání
2. vytváření
3. modifikace stávajících

Jedinou výjimkou z tohoto pořadí jsou tabulky, kde je situace kvůli cizím klíčům poněkud složitější. Zpracování těchto změn se řídí pořadím:

1. smazání starých cizích klíčů
2. smazání starých tabulek

3. vytvoření nových tabulek
4. změna stávajících tabulek
    1. smazání starých indexů
    2. smazání starých sloupců
    3. vytvoření nových sloupců
    4. vytvoření nových indexů
5. vytvoření nových cizích klíčů

Díky tomuto postupu je minimalizována šance vzniku „sirotků“, kteří porušují integritu datových struktur.

8.3 XML Adaptér

Pro zápis načtené databáze do souboru slouží XML adaptér, který zastupuje třída `XMLAdapter`. Tento adaptér se nepřipojuje ke skutečné databázi. Slouží pouze k zápisu a čtení dat z XML dokumentů. Ty mohou být jak soubory uložené na pevném disku, tak i načtené v paměti, odkud je adaptér vezme a naparsuje.

Tento adaptér obsahuje dvě veřejné vlastnosti. První vlastnost je řetězec `file` obsahující buď adresu zpracovávaného souboru, nebo načtený XML dokument. Druhá vlastnost je logická hodnota `useString`. Hodnota `useString` adaptéru poskytuje informaci, zda má hledat soubor na disku (hodnota FALSE), nebo jestli je má načíst z vlastnosti `file`.

Práci s XML dokumentem provádí open source knihovny PugiXML, která byla zmíněna už dříve v textu.

Názvy jednotlivých prvků (jejich seznam a význam je v kapitole 7.3) má třída uložené jako statické řetězcové konstanty. Tímto způsobem jsou uloženy:

- názvy uzlů
- názvy atributů
- textová reprezentace enumeračních typů

Tento adaptér poskytuje všechny metody definované svým předkem plně funkční (na rozdíl od třídy LocalAdapter, o které je pojednáváno později). Je nutné však poznamenat, že funkce metod `writeData()` a `writeDiff()` jsou naprosto shodné. Obě metody zapíší předanou reprezentaci databáze do souboru.

Metoda konfigurace je prázdná, protože adaptér je nastavován přímo v programu a není nutné jeho konfiguraci nikde načítat.

Načítání i ukládání funguje velmi podobně, proto zde bude popsán pouze zjednodušený princip. Případní zájemci o detailnější informace nechť nahlédnou do zdrojového kódu.

Vstupně výstupní operace probíhají postupně, podle typu elementu. Každý typ elementu má své specializované metody, které čtení (respektive zápis) provádějí. V tomto případě tvoří výjimku procedury a funkce. Protože tyto dvě struktury jsou velmi podobné, obsluhuje je kolekce stejných metod.

Dále existují metody pro čtení/zápis univerzálních informací mající téměř všechny prvky společné. Jsou to jméno, komentář a specifické parametry, které jsou určené pro nestandardní nastavení různých databázových systémů (například parametr `ENGINE` u MySQL).

Obecný princip čtení probíhá tak, že se vezme kontejnerový prvek (zpravidla kořenový element nebo uzel reprezentující tabulku) a iteruje se nad jeho přímými potomky požadovaného typu. Využívá se vlastnosti prezentace XML uzlů knihovny Pugi. Pokud element požadovaného typu neexistuje, je sice vytvořena instance uzlu, ale při logickém porovnání vrací hodnotu FALSE. Stejná situace nastane, když se iteruje nad množinou sousedů (siblings) a při přechodu na další prvek se dostaneme mimo jejich množinu. Při

logickém porovnání je opět vrácena logická hodnota FALSE.

9 ZJIŠTĚNÍ A UCHOVÁNÍ ZMĚN

Pro získání informací o strukturách, které se změnily je potřeba mít v paměti dvě databáze. Tyto dvě databáze je potřeba porovnat a z tohoto porovnání získat požadované informace.

Stará verze databáze je načítána poskládáním výsledků starších porovnávání (pomocí XML adaptéru) a nová verze je načítána přímo z databázového serveru.

9.1 Porovnání databází

Stará a nová databáze je porovnána pomocí třída `Comparator`. Porovnání dat probíhá postupně podle typu struktur:

1. triggery
2. funkce
3. procedury
4. pohledy
5. tabulky

Hledání změn probíhá ve dvou fázích: nejprve se hledají smazané prvky a zároveň se vytvoří seznam struktur, které zůstaly, a v druhé fázi se hledají nově vytvořené prvky. Výsledek porovnání je zapsán do odpovídajících seznamů a předán nadřazené metodě, která porovnání zavolala.

Protože bylo nutné označit struktury, se nimiž se pracovalo, byla přidána třídě `SQLBase` nová vlastnost `compareResult`, která je výčtového typu a může nabývat hodnot:

- `CR_NOCOMPARE` – označuje prvky u kterých k žádným změnám nedošlo (standardní hodnota)
- `CR_ADDED` – označuje prvky, které byly nově vytvořeny
- `CR_CHANGED` – označuje prvky, uvnitř kterých došlo ke změně
- `CR_MODIFIED` – označuje prvky, které byly přímo změněny
- `CR_DELETED` – označuje prvky, které byly smazány
- `CR_LOGIC` – označuje prvky, které jsou potřeba pro úplné informace v XML dokumentu

Poslední hodnota se používá zejména pro uchování cizích klíčů. Jedná se například o případ, kdy je mezi dvěma tabulkami vytvořen nový cizí klíč, ale změněna je pouze tabulka s cizím klíčem (vytvoření cizího klíče). Pro uchování úplné informace o tomto klíči se tedy vytvoří referenční tabulka pouze se sloupci, na něž odkazuje daný cizí klíč, a všem těmto prvkům se nastaví výsledek porovnání na hodnotu `CR_LOGIC`. Tato hodnota pak pří zapisování změn adaptéru oznámí, že si struktury nemá všímat, protože existuje pouze pro zachování integrity dat.

Výsledkem tohoto procesu je delta balíček obsahující pouze změny, jež se udály od posledního uložení. Balíček je následně uložen do souboru a společně s dalšími informacemi (například kdo a kdy balíček vytvořil) zabalen.

Poskládáním série těchto balíčků vzniká aktuální verze databáze.

9.2 Ukládání získaných dat

Informace o rozdílu databáze je potřeba uložit tak, aby bylo možné provést výměnu takto získaných dat s ostatními členy týmu. Zároveň je nutné, aby každý balíček mohl být jednoznačně identifikován a bylo možné poznat, jestli nebyl během přenosu poškozen.

K identifikaci balíčku slouží jednak jméno souboru, které je zároveň kontrolním součtem dat, a jednak meta informace uložené uvnitř souboru. Tyto meta informace obsahují:

- kdo balíček vytvořil
- kontakt na tvůrce (emailová adresa)
- kdy byl soubor vytvořen
- informace o předcích balíčku
- identifikátor větve (viz 9.2.3)

Důležité je si uvědomit, že jeden balíček může mít vícero předků. Tato situace nastává v případě slučování dvou větví a musí splňovat tu podmínku, že nesmí existovat konflikt. O této problematice pojednává kapitola 9.4.

9.2.1 Konfigurace

Před použitím programu je nutná konfigurace. Ta se týká nastavení informací o databázi (adresa serveru, přihlašovací údaje atd.), nacionále uživatele a informace o umístění balíčku na lokálním počítači.

Protože ruční konfigurování by bylo značně pracné a neefektivní, podporuje program načítání konfiguračního souboru ve formě XML. Tento dokument obsahuje všechna data potřebná pro chod programu a jeho jméno se zadává jako parametr při spuštění.

Dokument je možné vytvořit buď ručně, nebo přímo za asistence programu, jenž se uživatele postupně ptá na potřebné informace a následně vytvoří daný konfigurační soubor, který uloží na specifikované úložiště (standardně adresář `config`).

Konfigurace také obsahuje aktuálně používaný balíček. Díky této informaci je možné navázat další článek na konec řetězu.

9.2.2 Třída delta balíčku

Delta balíček je v paměti reprezentován třídou `DiffPackage`. Ta obsahuje veškeré informace potřebné pro rekonstrukci databáze z řetězu instancí těchto objektů a poskytuje k tomu náležité metody.

Kromě řetězcových identifikátorů balíčků, které jsou předkem delta balíčku, obsahuje také seznamy ukazatelů jak na skutečné instance předků, tak i na instance potomků.

Navíc umožňuje data načítat a ukládat do ZIP souboru. Jakmile jsou data uložena, balíček se zamkne pouze pro čtení a není v něm možné dělat žádné změny. Tento přístup je nutný kvůli zachování zachování platnosti jména souboru s delta balíčkem: pokud by se obsah změnil, bylo by nutné změnit i jméno souboru, protože jméno je zároveň kontrolním SHA-1 součtem dat. Toto by vyžadovalo změnit i veškeré odkazy na tento soubor. Pokud by se všechny soubory nacházely na jednom fyzickém stroji, pak by to nebyl příliš velký problém všechny balíčky dohledat a provést náležité úpravy. Vývoj však zpravidla probíhá ve více lidech na různých strojích, které jsou často vzdáleny od sebe stovky kilometrů, a proto není možné rozumným způsobem tyto změny provést. Změna jednoho souboru by tak rozbila řetěz svých následníků a způsobila by nekonzistenci dat.

Kromě instančních vlastností a metod obsahuje také několik statických členů. Jsou to zejména globální seznamy, které se používají pro správu načtených balíčků a metody určené pro manipulaci s nimi. Jejich význam a využití je popsáno níže v kapitole 9.2.5 věnující se zřetězení existujících balíčků.

Metody, které jsou součástí instance dovolují procházet řetěz balíčků, manipulovat s nimi i sestavovat a vyhledávat posloupnosti těchto balíčků:

- `addParent()`
- `assembleDatabase()`
- `getDatabase()`
- `getDeltaChain()`
- `getMergeables()`
- `meta()`

- `setCurrentTime()`
- `setDatabase()`
- `writeFile()`

Metoda `addParent()` má jediný parametr, kterým je ukazatel na instanci jiného balíčku. Jedná se o metodu využívanou při slučování větví, o němž je pojednáno v kapitole 9.4. Metoda slouží k přidání dalšího předka do seznamu.

Při slučování větví se také využívají metody `getMergeables()` vracející seznam větví, které je možné sloučit (jejich nejaktuálnější balíčky ještě nebyly zapsány do naší vývojové větve), a `getDeltaChain()`, jejíž úloha je nalézt posloupnost balíčků aktuální a slučované větve. Problematika a průběh tohoto procesu je včetně algoritmu vysvětlen taktéž v kapitole 9.4.

Patrně nejdůležitější a nejpoužívanější metodou je `assembleDatabase()`. Tato bezparametrická metoda je používána k rekonstrukci databáze z posloupnosti delta balíčků. Detailní popis její funkce je v kapitole 9.3.

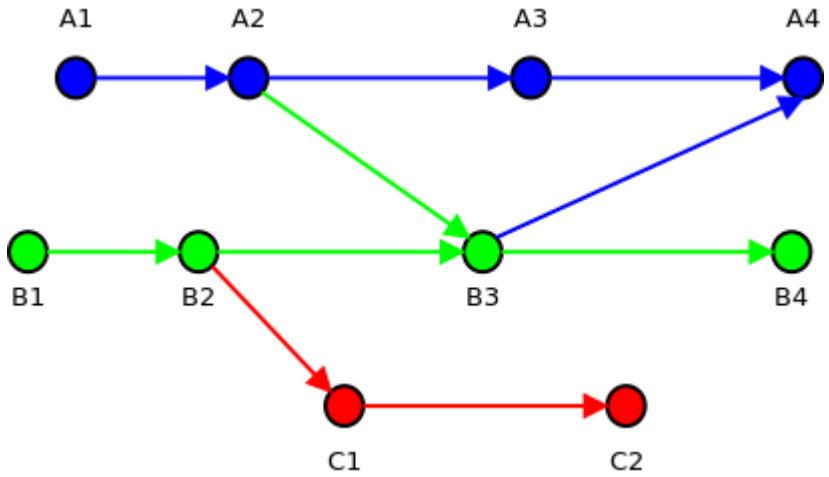
Dalšími metodami jsou přístupy k informacím o balíčku, jedná se o `getDatabase()`, `setDatabase()` a `meta()`. První dvě jsou určeny k manipulaci s rozdílovou databází, která je součástí balíčku. Poslední zmíněná vrací referenci na meta informace, obsahující údaje o tvůrci, čas a podobně (viz začátek kapitoly 9.2).

Metoda `setCurrentTime()` má využití při tvorbě nového balíčku pro nastavení času tvorby. Čas je uchován v meta informacích. Pokud je balíček označen pouze pro čtení, nemá tato metoda žádný účinek.

Poslední metodou týkající se instancí je `writeFile()`. Má jediný parametr typu `std::string` a obsahuje adresář, kam se má balíček uložit. Tato metoda provede převod reprezentace databáze do XML dokumentu (viz kapitola 7.3), do dalšího XML převede meta data a nakonec vše uloží do ZIP souboru. Tento proces je popsán detailněji v kapitole 9.2.4. Návratovou hodnotou je pak jméno souboru (bez přípony), do kterého byl obsah uložen.

9.2.3 Větvení změn

Jak bylo naznačeno, vývoj probíhá na různých místech, kdy se autoři nemusí vůbec vidět, pouze si vyměňují informace o změnách. Není možné tedy změnu provedenou uživatelem A okamžitě aplikovat na stanici uživatele B. Možné řešení je využití samostatných vývojových větví, které jsou po výměně balíčků sloučeny.



Obrázek 12: Větvení změn

Nelze ale použít přístup k větvím jako používá například Git, kdy větve vznikají pouze při konfliktu dvou commitů při přístupu k jednomu souboru (pokud nejsou vytvořeny ručně pomocí náležitého příkazu) [17]. Je to z toho důvodu, že na databázi nelze vždy nahlížet jako

na soustavu adresářů a souborů (i když by ve většině případů tato analogie mohla fungovat). Nesmíme zapomínat na cizí klíče, které jednotlivé struktury propojují, a proto na databázi nahlížíme jako na jeden velký celek.

Každý uživatel tedy musí vést vlastní větev vývoje databáze, jež je příležitostně (když dojde k výměně balíčků) synchronizována s ostatními. Nová větev může být vytvořena jako prázdná (větve A a B na obrázku 12 – například při vytvoření nového projektu), nebo jako odštěpení z existující větve (větev C na témže obrázku – například při vstupu nového vývojáře do projektu).

Tyto větve jsou identifikovány pomocí osmi bytového čísla zapsaného v hexadecimálním tvaru. O generování se stará statická metoda `generateBranchName()`, která je členem třídy `DiffPackage`. Vzhledem k předpokládanému malému množství větví je identifikátor generován z aktuálního časového údaje zjištěného pomocí funkce time(), jež je součástí STL [26]. Po vygenerování je provedena kontrola, zda nedošlo ke konfliktu. Pokud ke kolizi identifikátorů došlo, informace o čase se inkrementuje o jedničku a proces kontroly proběhne znovu. K převodu čísla na hexadecimální tvar je použita funkce `sprintf()`, taktéž součást STL [26].

Slučování větví probíhá ručně zvláštním příkazem a je detailně popsáno níže v kapitole 9.4, která je celá věnována této problematice.

9.2.4 Formát přenosového souboru

Jako formát delta balíčku byl zvolen kompresní formát ZIP. Tato volba proběhla za subjektivního zhodnocení, protože autor této práce s knihovnou již pracoval, a vývoj tedy mohl probíhat rychleji.

Uvnitř archivu se nacházejí tyto dva soubory:

- XML soubor s meta informacemi o balíčku (pro DTD viz přílohu 3)
- XML soubor s rozdílovou databází

Jméno souboru je samotné SHA-1 kontrolním součtem celého archivu. Tato metoda sice snižuje přehlednost, ale zajišťuje jednoduché porovnání změn, odhalení duplicit a jednoduchou ochranu proti neautorizovaným změnám.

Pro ukládání ZIPů je použita open source knihovna libZip2 poskytující všechny potřebné funkce pro práci se soubory tohoto typu [22].

Pro práci s daty v souboru se používá několik privátních metod a navíc metoda writeFile(), popsaná výše v kapitole 9.2.2.

Tyto privátní metody slouží k načítání delta balíčku ze souboru a ke generování meta informací při ukládání do souboru. Jak bylo řečeno, meta informace jsou zapsány v XML souboru. Veškeré názvy uzlů, atributů a hodnot atributů jsou definovány jako veřejné řetězcové konstanty a jejich přehled se nachází v hlavičkovém souboru `DiffPackage.h`, který je součástí aplikace `Tribble` a popis struktury XML dokumentu (DTD) je v příloze 2.

Poté co je celý soubor uložen na disk, je balíček označen jako pouze pro čtení. Od té chvíle v něm sice lze provádět některé změny, ale není možné je uložit.

9.2.5 Tvorba řetězu změn

Jednotlivé soubory s delta balíčky jsou uloženy v jednom adresáři. Program tento adresář prozkoumá, načte všechny balíčky a zapíše si je do paměti. Pro uchování balíčků v paměti je použit návrhový vzor Pool [25]. Tento návrhový vzor se používá k uchování kontroly nad počtem instancí, kdy je potřeba, aby se nevytvářely nové, ale použily se již dříve konstruované [25]. Protože balíčků je vždy známý počet, je tento návrhový vzor ideální k jejich správě.

Existuje jediný okamžik, kdy se za běhu může počet instancí zvětšit, a to při tvorbě nového delta balíčku. V tom případě se počet instancí zvětší o jedničku a nový balíček je po

uložení zapsán do poolu.

Instance všech jsou tedy balíčků uloženy ve statické vlastnosti třídy. Jedná se o typu `std::map<string, DiffPackage>`, kde klíčem je jméno souboru (SHA-1 hash souboru).

Načítání balíčků z adresáře probíhá ve dvou fázích:

1. načtení balíčků z adresáře
2. vytvoření vazeb mezi balíčky
    1. zpětné řetězení
    2. dopředné řetězení
    3. sestavení větví

K analýze adresáře a vypsání souborů je použita komponenta `Filesystem` multiplatformní knihovny `Boost` [23][24]. Celá knihovna je velmi rozsáhlá a její důkladné popsání by vydalo na samostatnou práci. Pro potřeby tohoto projektu postačí vědět, že použitá část knihovny umožňuje pracovat se souborovým systém, a to bez nutnosti se vázat na operační systém. Její použití je stejné jak pro platformu Windows, Linux i MacOS [24].

Pomocí třídy `directory_entry` [23][24] se procházení adresářem a hledají se všechny ZIP soubory, jejichž jméno je dlouhé 40 znaků (délka řetězcové reprezentace SHA-1). Každý z těchto souborů je načten a zařazen pod svým jménem (kontrolním součtem) do poolu.

Pro vlastní načtení dat balíčku existuje privátní statická metoda `loadFile()`, které se předá jméno souboru s balíčkem a ona se postará o načtení dat. Pokud načítání proběhne správně, pak vrací logickou hodnotu TRUE, v opačném případě vrací FALSE.

Neúspěch při načítání může nastat při těchto případech (případy jsou seřazeny podle pořadí, v jakém mohou nastat v programu):

1. nečitelný ZIP archiv
2. nepodaří se načíst informace o souboru s meta daty
3. nepodaří se otevřít soubor s meta daty
4. nepodaří se načíst data z meta souboru
5. nepodaří se načíst informace o souboru s databází
6. nepodaří se otevřít soubor s databází
7. nepodaří se načíst data ze souboru s databází

V případě neúspěchu je vyhozena výjimka a program skončí neúspěchem. Pokud se načítání těchto dat zdaří, pak se vytvoří nový delta balíček, kterému se nastaví načtená data a zapíše se do poolu, pod své jméno.

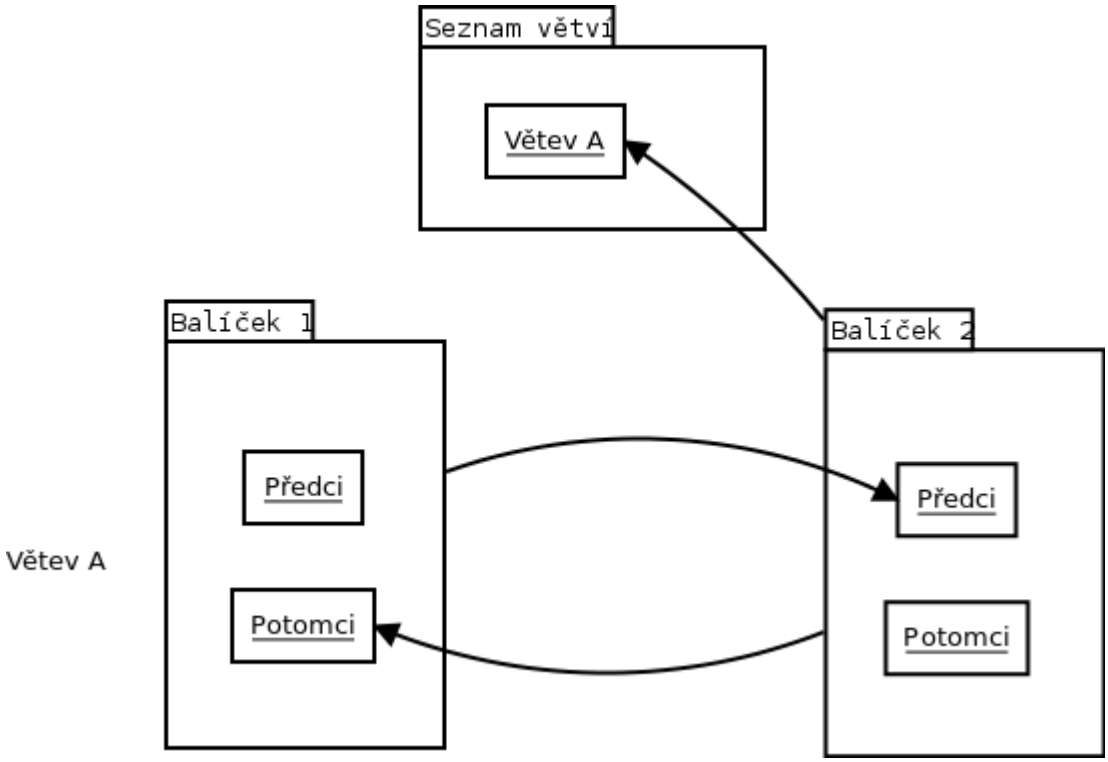
V druhé fázi jsou vytvořeny ukazatelové vazby rodič-potomek mezi jednotlivými instancemi a je vytvořen seznam větví. Vazby jsou vytvářeny tak, že se iteruje postupně nad všemi balíčky a kontrolují se identifikátory předků aktuálního balíčku. Do seznamu potomků každého předka je zapsán ukazatel na aktuální balíček a stejně tak naopak je zapsán do aktuálního balíčku ukazatel na každého předka.

Díky tomu, že existují vazby vytvořené v druhé fázi, je možné se jednoduše pohybovat mezi jednotlivými balíčky. Tento pohyb probíhá po „ose“ řetězu dopředu nebo zpět. Této vlastnosti se využívá zejména při sestavování databáze z delta balíčků.

Zároveň je vygenerován seznam větví, který je uložen ve statické vlastnosti `branches` typu `BranchList` (`std::map<std::string, DiffPackage*>`). Klíčem k této mapě je identifikátor větve a pointer, který je hodnotou a ukazuje na nejnovější balíček (takzvaný HEAD) této větve. Zápis do seznamu větví provádí statická metoda `writeBranchId()`.

Této metodě je jako parametr předán ukazatel na balíček. Nejprve se zkontroluje, zda větev, do které balíček patří, už existuje. Pokud větev neexistuje, vytvoří se o ní záznam a balíček se zapíše jako aktuální. V opačném případě je porovnán čas vytvoření, který je

uložen v meta informacích, (viz kapitolu 9.2) a pokud je čas zpracovávaného balíčku větší (balíček byl vytvořen později), pak v seznamu nahradí stávající ukazatel za adresu tohoto novějšího balíčku. Grafické znázornění modelu závislostí je na obrázku 13.



Obrázek 13: Datová struktura balíčku

Protože se tento proces porovnávání a zápisu větví provede se všemi načítanými balíčky, máme zaručeno, že v seznamu budou jak všechny větve, tak i jejich nejaktuálnější balíčky, které máme k dispozici.

9.3 Aplikace řetězu změn

Při sestavování řetězu změn se provádějí dva kroky: nejprve se postupuje „odshora dolů“, to znamená od aktivního balíčku ke kořenům, a následně se vrací zpět „odspoda nahoru“. V první fázi dochází k vyhodnocování akce k provedení. Provedená akce se odvíjí od počtu balíčků, které jsou předkem momentálně zpracovávaného. Pokud balíček nemá předka, jedná se o kořenový balíček a je proveden zápis do paměťové reprezentace. Pokud existuje právě jeden předek, je „probubláno“ o úroveň níž (na úroveň předka) a opět dojde k vyhodnocení akce, která se provede. Toto se opakuje dokud není dosaženo kořene.

V případě dvou balíčků, které jsou předky se jedná o sloučení (merge) dvou vývojových větví stromu. Sloučení větví je poněkud komplikovanější mechanismus, než sestavení jedné větve a je mu věnována kapitola 9.4, která se nachází níže.

9.3.1 Lokální adaptér

Při sestavování databáze z balíčků bylo potřeba zapisovat zjištěné změny nikoliv do „fyzického“ úložiště (XML soubor pro přenos nebo databázový systém), ale do paměťové reprezentace. Z tohoto důvodu byl vytvořen takzvaný lokální adaptér (třída `LocalAdapter`). Tento speciální typ adaptéru je schopen zapisovat změny přímo do paměťové reprezentace. Stejně jako ostatní adaptéry je potomkem ryze virtuální třídy `Adapter`. Na rozdíl od nich ovšem implementuje z této bázové třídy pouze metodu writeDiff(), která zapisuje změny. Ostatní metody (`readData()`, `writeData()`

a `setConfig()`) nedělají nic, neboť nejsou potřeba.

Lokální adaptér má pouze jediný parametr. Je jím `target` typu ukazatel na instanci `Database`. Do tohoto parametru je přiřazen pointer na instanci reprezentace, v níž probíhá sestavování výsledné databáze, která bude následně po kompletním sestavení vrácena.

9.4 Slučování větví

Významnou funkcí při práci s databází je slučování dvou větví. Po výměně balíčků s kolegou dojde k vytvoření nebo k aktualizaci existující paralelní větve vývoje. Tyto změny je potřeba promítnout do lokální databáze, aby pracovník měl k dispozici změny, které jeho kolega provedl u sebe. Tento proces se nazývá sloučení větví (anglicky merge).

Po dokončení běhu této funkce jsou spojeny vždy větve. Pokud je potřeba spojit více větví, je nutné je slučovat postupně. Omezení pro sloučení pouze dvou větví bylo zvoleno zejména z realizačních důvodů, neboť tato operace nad více než dvěma větvemi by bylo jednak náročné na realizaci (vymyšlení vhodného algoritmu) a jednak poměrně nepřehledné při řešení konfliktů.

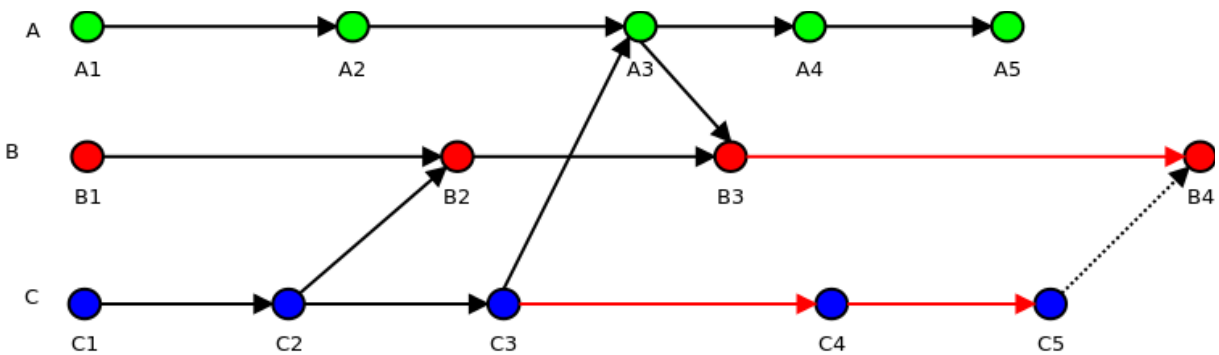
Během tohoto procesu je vytvořen zvláštní delta balíček. Tento balíček se odlišuje od ostatních tím, že má v meta datech uvedeny odkazy na dva rodičovské balíčky, a soubor s rozdílovou databází neobsahuje změny oproti minulému uložení. Místo těchto změn obsahuje informace o změnách, které se přenášejí z druhé větve.

Je programově zaručeno, že v případě více rodičovských balíčků je první v seznamu uveden balíček stejné (hlavní) větve. Druhý balíček pak pochází z větve, se kterou je hlavní větev slučována. Tato větev bude dále v textu nazývána vedlejší větví.

Algoritmus sloučení vychází z myšlenky sloučení předchozích změn od posledního provedení merge:

1. Vyber posloupnost balíčků druhé větve od posledního sloučení
2. Vytvoř prázdnou delta databázi
3. Nastav proměnnou i na hodnotu 0
4. Vyber i-tý balíček ze seznamu a zapiš informace o změnách do delta databáze
5. Zvětši proměnnou i o jedničku
6. Pokud je i menší než počet balíčků pokračuj na krok 4
7. Konec

Problém se vyskytl při zjišťování balíčku, který je jako poslední zapsaný do hlavní větve. Není možné jít po hlavní větvi v historii zpět a kontrolovat, zda u balíčků s více předky je druhý balíček z vedlejší větve. Toto by fungovalo, pokud by vývojové větve byly pouze dvě. Pokud je větví vícero, může nastat situace, kdy je vedlejší větev slučována nepřímo, přes větev jinou, jak to ukazuje obrázek 14.



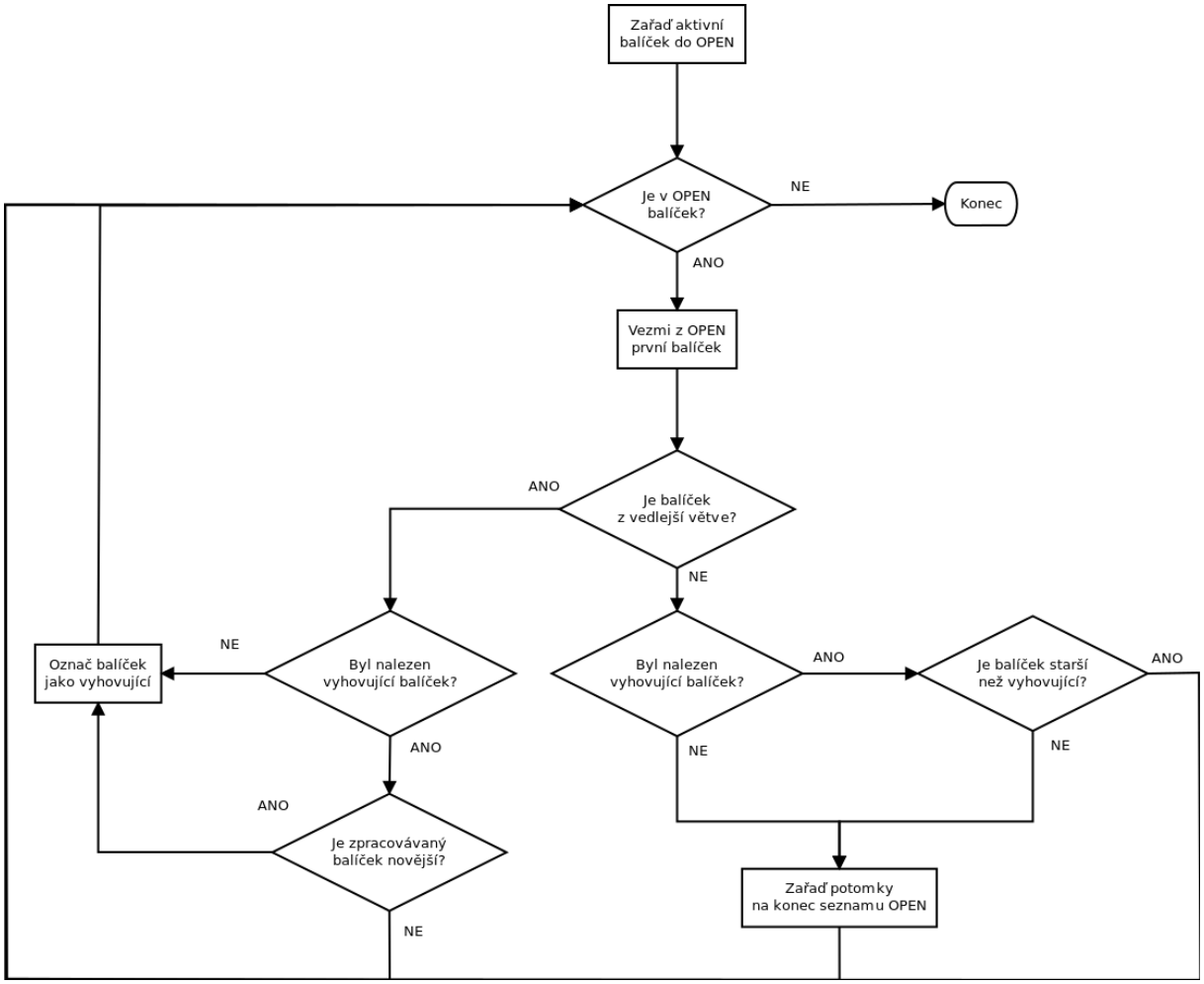
Obrázek 14: Nepřímé sloučení větví

Na tomto obrázku plní úlohu hlavní větve vývojová linie B a vedlejší větví je C. Pokud se postupovalo pouze jednoduše zpět v historii hlavní větve, tak označil by se jako nejbližší sloučený balíček C2. Při bližším prozkoumání ale nalezneme druhou spojnici, která

vede přes uzel A3 do prvku C3 a ten je tedy poslední z větve C, jehož změny se doposud promítly do vývoje hlavní větve.

Aby bylo zaručeno, že bude nalezen skutečně „nejmladší“ balíček, musí se použít vhodný algoritmus. Tento algoritmus pracuje tak, že se aplikuje prohledávání grafu, který provázané větve vytvářejí. Přestože se používá prohledávání do šířky, není zaručeno, že první nalezený balíček z vedlejší větve, je skutečně nejnovější. Tato skutečnost vychází z proměnlivého časového intervalu mezi jednotlivými balíčky. Může tedy existovat situace, v níž spojnice mezi dvěma delta balíčky pokrývá časové období jednoho týdne a paralelně s ní je série pěti balíčků, které vznikly v průběhu jediného dne.

Po nalezení prvního vyhovujícího balíčku tedy hledání probíhá dál. V této části hledání je však aplikováno jedno omezení: pokud se narazí na balíček, který je starší než posledně nalezený vyhovující, pak se prohledávání této části zastaví a pokračuje se na další rozpracovanou větev. Pokud je nalezen vyhovující balíček, který je novější, pak se označí jako poslední vyhovující a pokračuje se v hledání, dokud existují neprohledané prvky. Pro názornost je tento algoritmus zobrazen na obrázku 15.



Obrázek 15: Algoritmus hledání nejnovějšího balíčku

Je nutné si uvědomit, že toto sloučení je provedeno pouze na hlavní větvi. Na vedlejší větví se výsledek tohoto procesu nijak neprojeví a je nutné, aby ho druhý uživatel provedl samostatně. Názorný příklad je na obrázku 12: v uzlu B3 je sloučena větev B (uzel B2) a A (uzel A2). Uzel B3 je pak výsledkem tohoto spojení a obsahuje informace z obou větví vývoje. Větev A je naproti tomu stále bez informací o dění v ostatních větvích, až do uzlu A4, kde dojde ke sloučení s větví B (uzel B3).

9.4.1 Konflikty

Při slučování se může stát, že obě větve nějak různě modifikovaly stejný prvek (například v jedné větvi byl tabulce přidán sloupec a ve druhé větvi byla celá tabulka smazána). Tento stav se nazývá konfliktem a zpravidla není možné ho vyřešit automaticky. O vyřešení takové situace rozhodne uživatel, který vybere jestli bude použita hlavní nebo vedlejší větev.

Ke zjištění, zda se slučované větve nacházejí v konfliktu slouží třída `ConflictFinder`, která je součástí knihovny `SQLRepresenter`. V předchozím textu už bylo ve zjednodušené formě naznačeno, jak kontrola konfliktu probíhá:

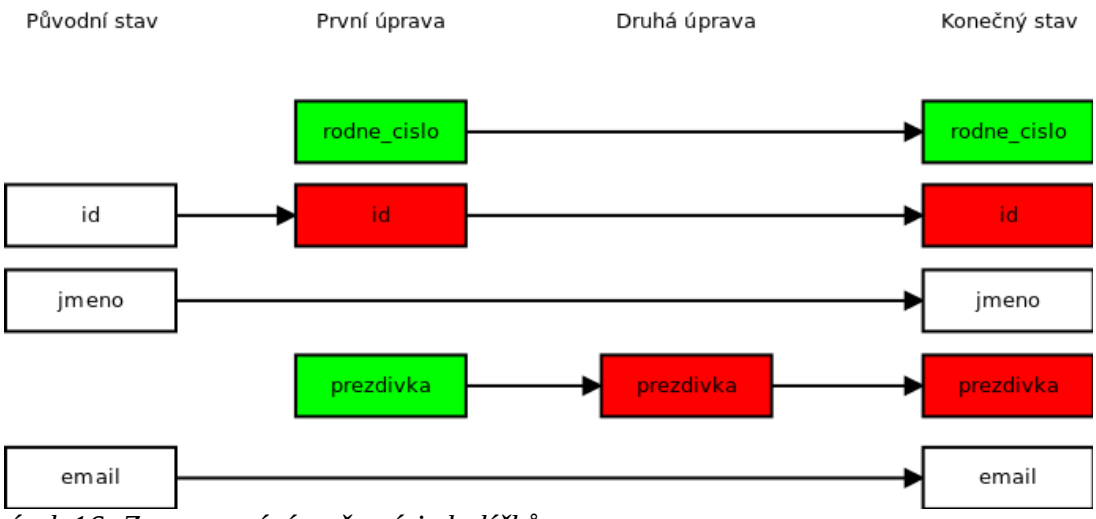
1. Vytvoří se posloupnost delta databází pro obě větve, tak jak jdou za sebou
2. Databáze se porovnají tak, že pokud se v obou databázích manipuluje se stejnou entitou, pak dochází ke konfliktu a zapíše se do připravené reprezentace
3. Výsledná reprezentace databáze není prázdná, pak došlo ke konfliktu

Porovnání databází probíhá tak, že se začne procházet seznam balíčků od nejstaršího po nejnovější (nejprve hlavní a poté vedlejší větev). Každá změna v balíčku je zaznamenána do reprezentace databáze, tak že se vyhledá, jestli daná entita již existuje (jestli se s ní už pracovalo). Pokud existuje, pracuje se s ní, a pokud neexistuje, vytvoří se nová. Do takto vzniklého balíčku se přiřadí aktuální hodnoty entity z delta balíčku. Pokud však je vlastnost `compareResult` nastavena na hodnoty `CR_LOGIC` a `CR_NOCOMPARE`, tak se entita ignoruje, protože hodnota `CR_LOGIC` označuje prvek, který je v reprezentaci pouze pro zachování integrity dat a `CR_NOCOMPARE` by se v delta databázi neměl vůbec vyskytovat.

Zpracování této série je graficky znázorněno na obrázku 16. Jedná se o příklad, kde se zpracovávají změny v jedné tabulce. Význam barev je následující:

- bílá – prvek se zatím nijak nemění
- zelená – prvek byl vytvořen
- červená – prvek byl smazán

Ve čtvrtém řádku je možné si všimnout nového sloupce „prezdivka“, který je v následujícím sloupci opět smazán. Přestože se prakticky nic nezměnilo (v původní verzi sloupec nebyla v poslední opět není), je tento stav zaznamenán. Jedná se o zjednodušení, neboť pokud by se měly tyto případy pokaždé kontrolovat, muselo by se jednak pracovat s celou databází, a zároveň by se musel každý prvek, který je ve výsledku označen jako smazaný, kontrolovat, jestli na počátku existoval nebo ne. Toto by značně zkomplikovalo celou kontrolu konfliktů. Proto byl zvolena metoda, kdy se zaznamenají i prvky, jež se na výsledku nijak neprojevily. S touto vlastností je ve zbytku programu počítáno a je na ní brán ohled.



Obrázek 16: Zaznamenání změn série balíčků

Zpracování těchto změn probíhá v každém balíčku postupně po jednotlivých prvcích. K tomuto účelu má třída ConflictFinder pět specializovaných metod:

- `buildBase()`
- `buildTables()`
- `buildTable()`
- `buildIndexes()`
- `buildForeigns()`

První metoda slouží k sestavení těch prvků databáze, které nejsou nijak zanořeny. To znamená pohledy, procedury a funkce. Do této kategorie prvků sice spadají i tabulky, ale ty mají svoji vlastní metodu `buildTables()`. Tato metoda jednak sestaví tabulky (myšleno nejvyšší strukturu tabulky) a jedna volá metodu `buildTable()`, která sestavuje jednotlivé „nižší“ prvky tabulky bez dalších vazeb (to znamená triggery, checky a sloupce).

Metodu `buildTable()` by bylo obtížné napsat tak, aby podporovala zároveň prvky s křížovými vazbami (indexy a cizí klíče). Proto byly vytvořeny dvě specializované metody, aby se zpracování zjednodušilo.

Metoda `buildIndexes()` sestavuje vyhledávací indexy. Nejprve načte nebo vytvoří index daného jména a odstraní veškeré stávající vazby na sloupce. Poté iteruje nad sloupci, které jsou zapsány v delta balíčku, a zapisuje jejich protějšky ze sestavené databáze do seznamu sloupců indexu. Pokud hledaný sloupec neexistuje, pak byl pravděpodobně při předchozím zpracování přeskočen, protože měl hodnotu porovnání nastaven na `CR_LOGIC`. V tomto případě se tedy vytvoří nový sloupec s daným jménem a nastaví se mu porovnání na logický člen delta databáze.

Obdobně se zapisují cizí klíče. Při tomto procesu se však na začátku maže také hrana směřující na referenční tabulku a vazby na referenční sloupce. Proces kopírování probíhá opět stejně jako u indexu.

Protože zpracování indexů a cizích klíčů má jeden společný proces a v případě cizích klíčů se jeden proces opakuje, byly pro tyto kusy kódu vytvořeny speciální pomocné metody, aby nedošlo k zbytečné duplikaci zdrojového kódu. Těmito metodami jsou `removeEdges()`, jež jako parametr dostává seznam hran. V těle metody se iteruje nad hranami a probíhá jejich mazání. Destruktor třídy hrany se pak postará o to, aby nevznikaly žádné neintegrity a nikde uvnitř grafu nezbyly ukazatele do neplatné paměti.

Druhou metodou je `copyColumns()`. Ta je určena pro kopírování sloupců z jednoho cizího klíče do druhého. V parametrech je předán zdrojový a cílový (cizí) klíč, dále pak tabulka sloužící jako zdroj sloupců a nakonec typ vazby, který bude použit pro nově

vytvořenou hranu. Podle něj se vyhodnotí, jestli se budou zpracovávat místní nebo referenční sloupce. Načte se tento seznam sloupců a proběhne jejich zápis. Opět se uplatňuje mechanismus, že pokud sloupec neexistuje, je vytvořen nový s hodnotou `CR_LOGIC`.

Aby bylo možné správně porovnat všechna data, jsou uzly grafu reprezentující jednotlivé prvky databáze kopírovány včetně všech svých vlastností (těmito vlastnostmi nejsou myšleny vazby na další elementy). Protože při zápisu o změně se zapíší všechny informace o novém stavu prvku, nemůže vzniknout situace, kdy výchozí hodnoty překryjí starší informace, které jsou také relevantní. Tím je myšleno, že pokud se například u sloupce změní pouze datový typ, je zaznamenán kompletně celý nový stav tohoto sloupce, tedy včetně jména, výchozí hodnoty, přepínače NOT NULL a dalších informací.

Po sestavení změnových databází pro obě větve lze provést vyhledání konfliktů. Při tomto procesu se opět samostatně zpracovávají v první fázi procedury, funkce a pohledy a ve druhé fázi pak tabulky a jejich podřízení členové. Během vyhledávání se vždy vezme jedna databáze, respektive tabulka, a vyhledají se všechny prvky stejného typu (například procedury nebo sloupce v případě tabulky). Poté proběhne pokus o vyhledání prvku stejného jména v druhé databázi. Pokud není druhý prvek nalezen, pak v druhé vývojové větvi nedošlo k jeho změně a nemůže tedy vzniknout konflikt.. V tomto případě se tedy pokračuje se na další položku prvního seznamu. V případě, že prvek nalezen byl, je nutné určit, zda došlo ke konfliktu, či nikoliv. K tomuto účelu slouží metoda `isConflict()`, které jsou jako parametry předávány oba nalezené prvky a v těle této metody proběhne vyhodnocení, jestli jsou prvky v konfliktu, či nikoliv.

Vyhodnocení, že jsou objekty v konfliktu, nastane jednak při následujících kombinacích hodnot porovnání:

- `CR_DELETED`, `CR_ADDED`
- `CR_DELETED`, `CR_MODIFIED`
- `CR_DELETED`, `CR_CHANGED`
- `CR_MODIFIED`, `CR_MODIFIED`

A jednak pokud se prvky nerovnají. V tomto druhém případě vzniku konfliktu se předpokládá, že oba prvky byly změněny a byly změněny jiným způsobem.

Pokud je vyhodnocení konfliktu kladné, je informace o prvku zanesena do konfliktní databáze. Tato informace je ve formě prvku daného typu a daného jména. Více informací (například datový typ sloupce atd) není pro tento účel potřeba. Kromě této konfliktní databáze je ještě zapsán do seznamku konfliktů, který je popsán níže.

Výsledkem celé operace je reprezentace konfliktní databáze a seznam konfliktů. Pokud je tato reprezentace prázdná (neobsahuje žádné prvky), pak nedošlo ke konfliktu a není potřeba ruční zásah uživatele. V opačném případě nelze zpracovat změny plně automaticky a je na uživateli, jak daný konflikt vyřeší.

Řešení konfliktu probíhá tak, že jsou uživateli postupně předkládány nalezené konflikty a je požádán o výběr operace, která bude použita. Více informací o průběhu ručního řešení konfliktů je v kapitole 10, která se věnuje uživatelskému rozhraní.

Aby bylo možné jednoduchým způsobem aplikovat řešení konfliktů při sestavování, byly prvkům databáze přidány následující čtyři vlastnosti:

1. databázi přibyla vlastnost `name` (řetězec), která se využívá pro uložení jména balíčku, ze kterého databáze pochází
2. databázi přibyla vlastnost `nextId` (celé číslo), která uchovává další identifikační číslo prvku
3. třídě `SQLBase`, ze které jsou odvozeny všechny databázové prvky, byla přidána vlastnost `id` (celé číslo), které je identifikačním číslem prvku
4. třídě `SQLBase` byla přidána řetězcová vlastnost `dbName`, která obsahuje informaci o jméně rodičovské databáze

Může se zdát, že nová hodnota popsaná v položce 4 je zbytečná, protože jméno databáze je možné zjistit z reference na mateřskou databázi. Při zpracování konfliktů ale vzniká potřeba uchovat informaci, ve kterém balíčků problém vznikl a je ho potřeba zapsat. V tuto chvíli se hodnota `SQLBase::dbName` nerovná `Database::name`.

Výše v textu byl zmíněn seznam konfliktů. Tento seznam je složen z několika datových typů, které jsou do sebe zanořeny a jsou definovány v hlavičkovém souboru `ConflictFinder.h`. Od nejhlubšího zanoření po nejvyšší to jsou typy:

1. `CONFLICT_SOLVE`
2. `ConflictList`
3. `Conflicts`

První položka v seznamu, `CONCLICT_SOLVE`, je struktura obsahující informace o konfliktu, které jsou potřeba pro jeho vyřešení a také pro vyhodnocení, zda konflikt byl vyřešen, nebo zda ještě čeká na vyřešení. Členové struktury jsou:

- `int id` – celé číslo a obsahuje identifikátor prvku vedlejší větve, ze kterého konflikt vznikl
- `bool useCurrent` – přepínač použití konfliktní změny
- `bool isSolved` – stavový člen obsahující informaci, zda byl daný konflikt už vyřešen

Položka `useCurrent` je přímo informace pro řešení konfliktu. V případě, že je nastavena na hodnotu FALSE, pak je konfliktní změna z vedlejší větve aplikována na hlavní větev. Pokud je tomu tak, jsou přepsány některé změny, které byly na hlavní větvi provedeny změnami z vedlejší větve.

Pokud je však nastavena na TRUE, tak je při sestavování databáze (které je popsáno na začátku této kapitoly) tato konfliktní změna přeskakována.

Druhý datový typ, kterým je `ConflictList`, je šablonová třída `std::vector`, která je typu `CONFLICT_SOLVE`. Je nutné, aby všechny položky v tomto seznamu měly rozdílné identifikační číslo prvku. Pokud by došlo ke shodě, mohla by tato situace vést k zacyklení procesu hledání konfliktů.

Posledním datovým typem vytvořeným pro ukládání záznamu o konfliktech je `Conflicts`, který je mapou `std::map<std::string, ConflictList>`. Do této mapy jsou zaznamenávány nalezené konflikty a automaticky jsou tříděny podle jména balíčku (klíč mapy), ve kterém byly nalezeny.

Vzhledem k tomu, že se seznamem konfliktů pracuje poměrně často, byly vytvořeny tři metody, které jsou pak následně využívány, aby nedošlo k duplikaci kódu. Seznam metod je:

- `findSolve()`
- `useCurrent()`
- `isConflictSolved()`

Všechny tyto metody přijímají dva parametry:

1. `std::string name`
2. `int id`

První parametr určuje balíček, z něhož změna pochází, a druhý je identifikačním číslem prvku, ze kterého konflikt vznikl. Metoda `findSolve()` je nejdůležitější a slouží k vyhledání konfliktu dle zadaných parametrů. Pokud daný konflikt není nalezen, pak se vrací výchozí hodnoty, která jsou:

- `useCurrent = fasle`
- `solved = false`
- `id = id`

Poslední přiřazovaná hodnota je identifikátor změny, pro kterou se konflikt hledá.

Další dvě metody pouze volají `findSolve()` a vracejí jednu hodnotu z výsledku

hledání. Metoda `useCurrent()` vrací hodnotu stejného jména a metoda `isConflictSolved` vrací hodnotu `solved`.

Při sestavování seznamu řešení konfliktů se může stát situace, kdy je potlačena konfliktní změna překrývající změnu, která je také konfliktní. Proto se po zpracování seznamu změn provede znovu kontrola (tentokrát i s již s aplikovanými nastaveními). Tento proces se opakuje tak dlouho, dokud není konfliktní databáze prázdná.

V okamžiku, kdy jsou všechny konflikty odstraněny, se může vytvořit balíček a provést zápis dat do souboru. Tento postup je téměř stejný jako normální tvorba rozdílového balíčku. Jediná změna tkví v obsahu delta databáze. Oproti klasickému zápisu, kdy jsou uloženy změny oproti předchozímu stavu, je obsahem souboru `diff.xml` zápis změn, které se musí provést, aby byly dorovnány rozdíly oproti vedlejší větvi. Tyto rozdíly se sestaví pomocí spuštění metody `build()` instance třídy `ConflictFinder`, která má přístup k seznamu řešených konfliktů.

Nakonec je aktualizována konfigurace profilu a vše je uloženo do příslušných souborů.

10 UŽIVATELSKÉ ROZHRANÍ

Rozhraní pro komunikaci s uživatelem je vytvořeno pouze pomocí polointeraktivní konzole. Pojmem polointeraktivní je myšlen kompromis, kdy se pro provedení operace použije kombinace parametrů předaných při spuštění programu a dotazů a odpovědí za běhu. Tyto dotazy a odpovědi probíhají přes standardní vstup a výstup během běhu programu.

Protože příkazy, které jsou pod tímto odstavcem, jsou všechny psány v angličtině, byla jako jazyk rozhraní také zvolena angličtina. Podporované příkazy jsou:

- create conf <jméno konfigurace>
- create diff <jméno konfigurace>
- show conf <jméno konfigurace>
- db export <jméno konfigurace> <jméno souboru>
- merge <jméno konfigurace>
- help

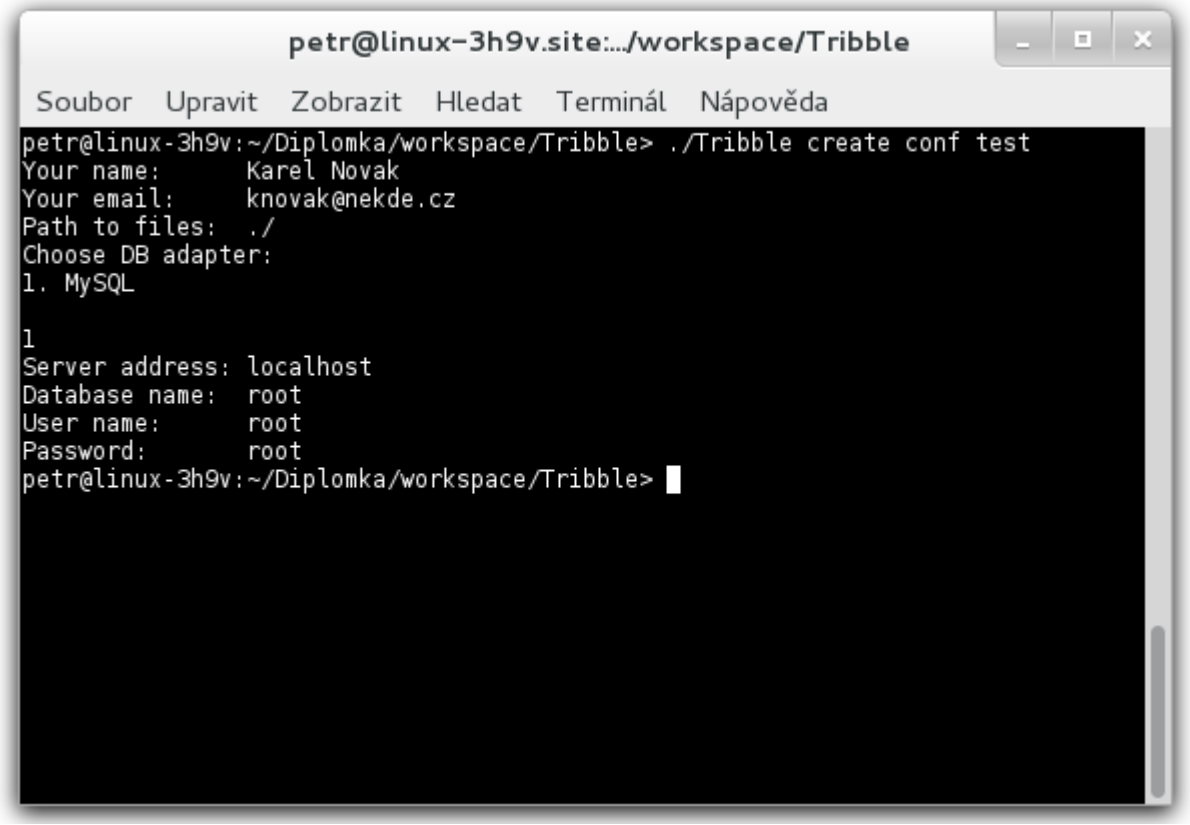
10.1 Create conf

Příkaz create conf vytvoří nový konfigurační soubor. Jako poslední parametr je jméno profilu k tomuto, podle kterého bude soubor pojmenován.

Následně se program ptá na informace o uživateli, které jsou potřeba k vytvoření profilu:

- jméno
- emailová adresa
- adresář, kde se budou ukládat delta balíčky
- databázový adaptér, která se použije
- nastavení databázového adaptéru

  Nastavení databázového adaptéru se může lišit v závislosti na vybrané databázi.

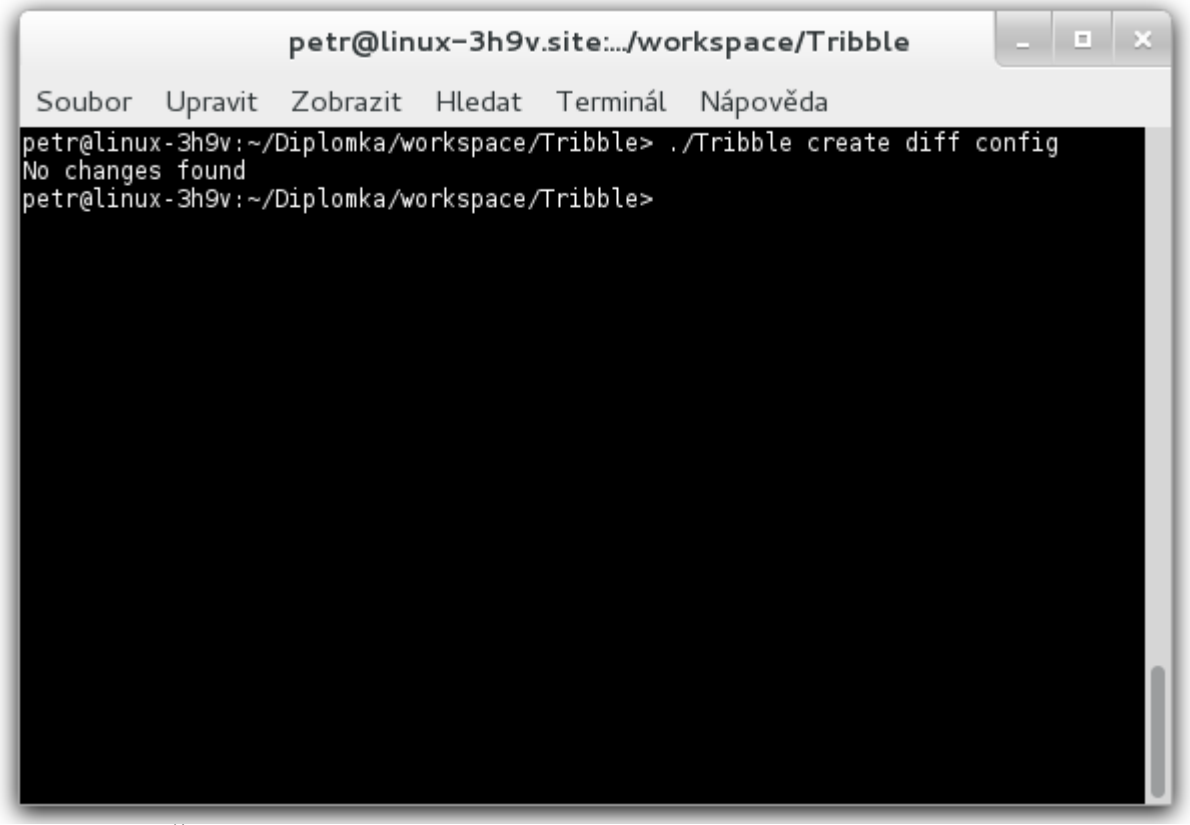


Obrázek 17: Tvorba nového profilu

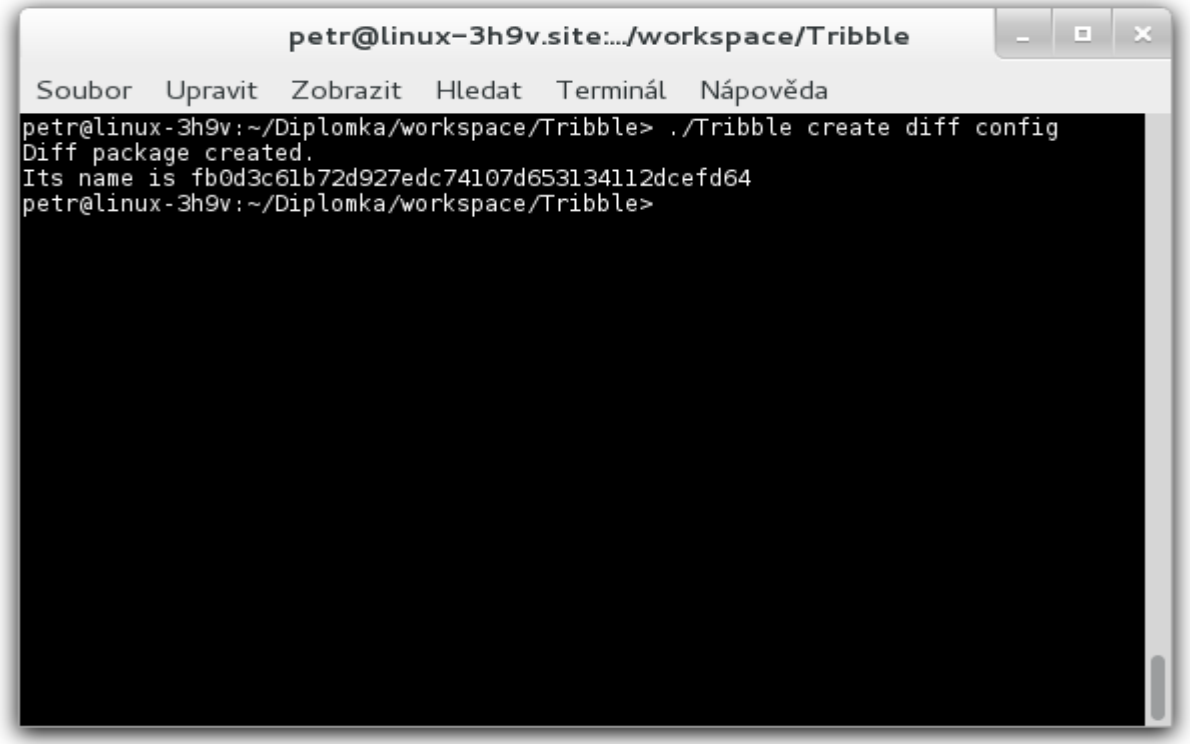
Po zadání požadovaných hodnot je do úložiště konfigurací zapsán nový soubor s právě vyplněnými daty.

10.2 Create diff

K vytvoření nového delta balíčku se použije příkaz create diff následovaný názvem souboru profilu. Po zadání tohoto příkazu program sestaví z dostupných balíčků uložený obraz databáze a pomocí zvoleného adaptéru připojí k databázovému serveru. Poté ze serveru získá aktuální strukturu skutečné databáze a provede porovnání. Pokud jsou obě databáze stejné, pak se nic nestane, pouze se vypíše informační hláška. V opačném případě se vypíše notifikace, že porovnání proběhlo a vypíše se také jméno nového souboru.



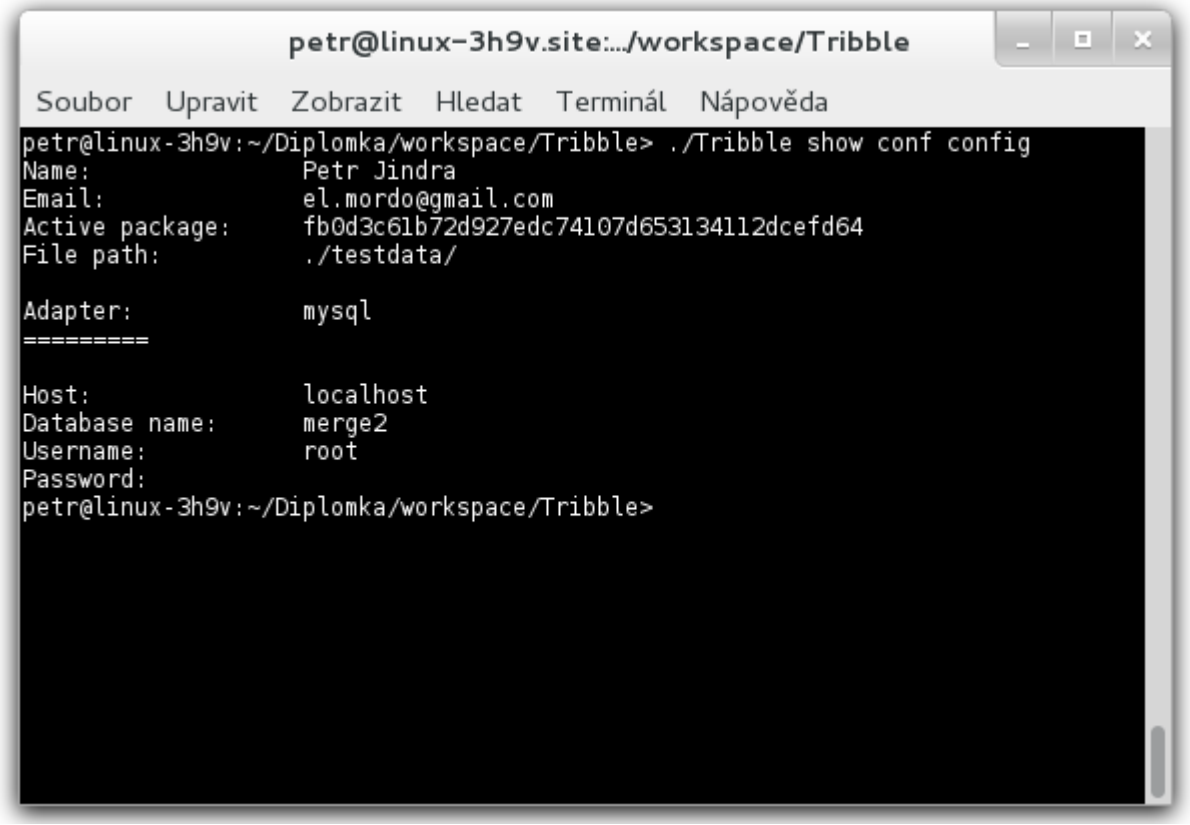
Obrázek 18: Žádné změny nebyly nalezeny



Obrázek 19: Vytvoření nového balíčku

10.3 Show conf

Pokud se vyskytne potřeba zobrazit konfiguraci, je možné k tomu využít příkaz show conf. Po zadání tohoto příkazu se vypíší jak obecná data, jako je jméno a email, tak i konfigurace databázového připojení.

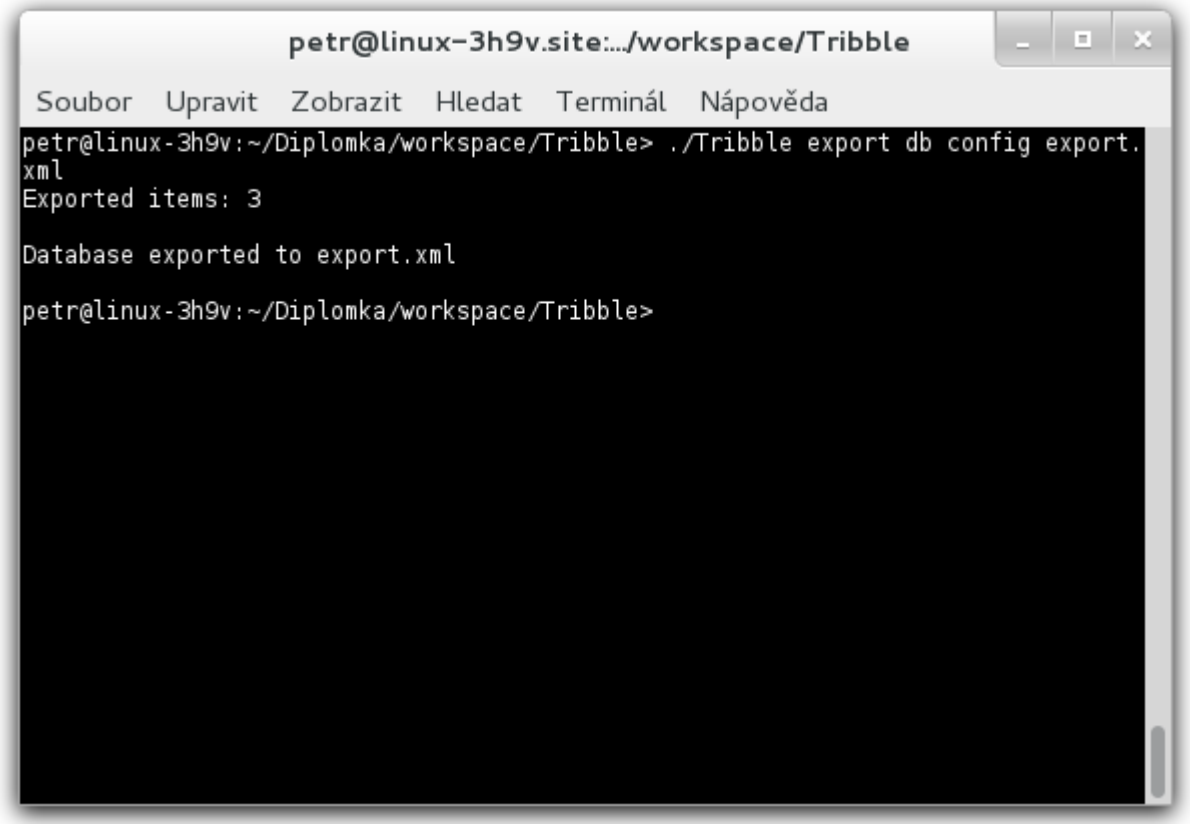


Obrázek 20: Zobrazení konfigurace

10.4 Export db

V případě nutnosti vyexportovat celou databázi tak, jak je na serveru, se použije příkaz export db. Během zpracování příkazu se program připojí k databázovému serveru, načte všechna data do reprezentace a tu pak zapíše do XML souboru, který uživatel určí jako poslední parametr při spuštění programu.

Po zpracování je na standardní výstup vypsána informace, kolik prvků bylo vyexportováno, a do kterého souboru byla databáze uložena.



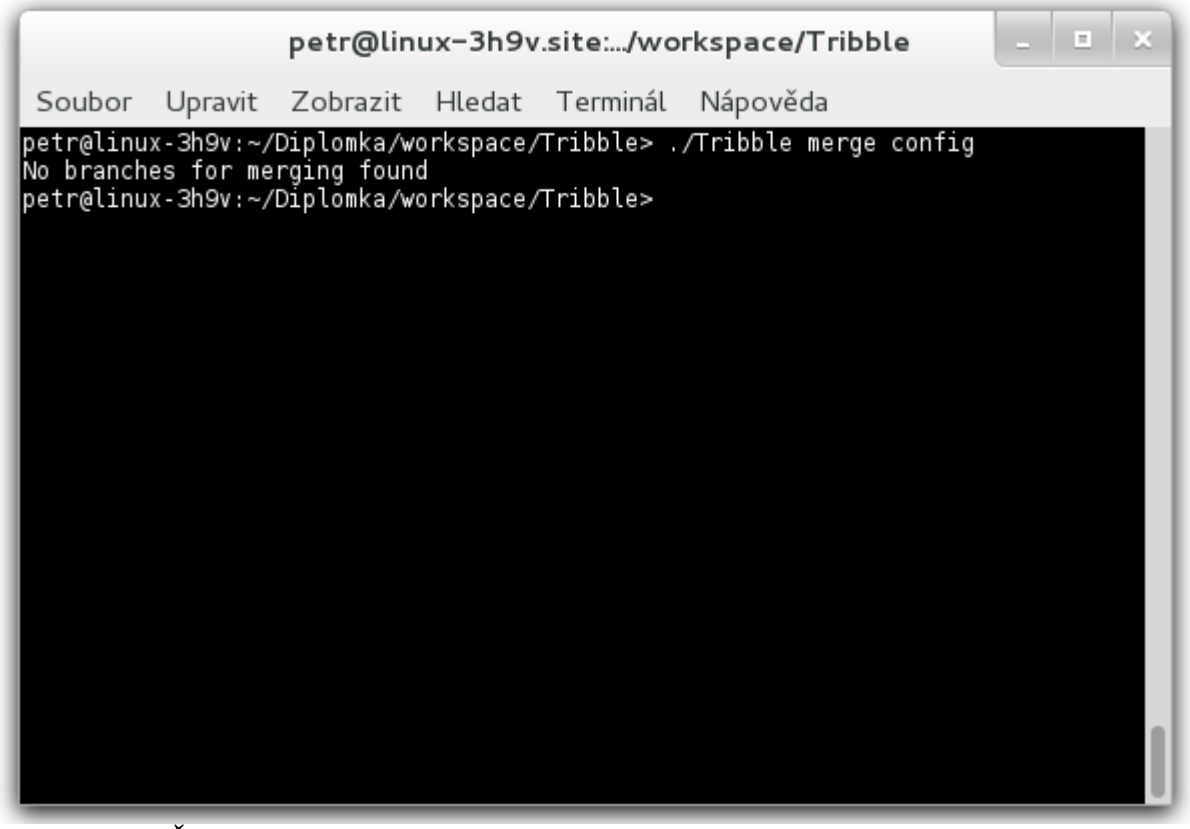
Obrázek 21: Export databáze

10.5 Merge

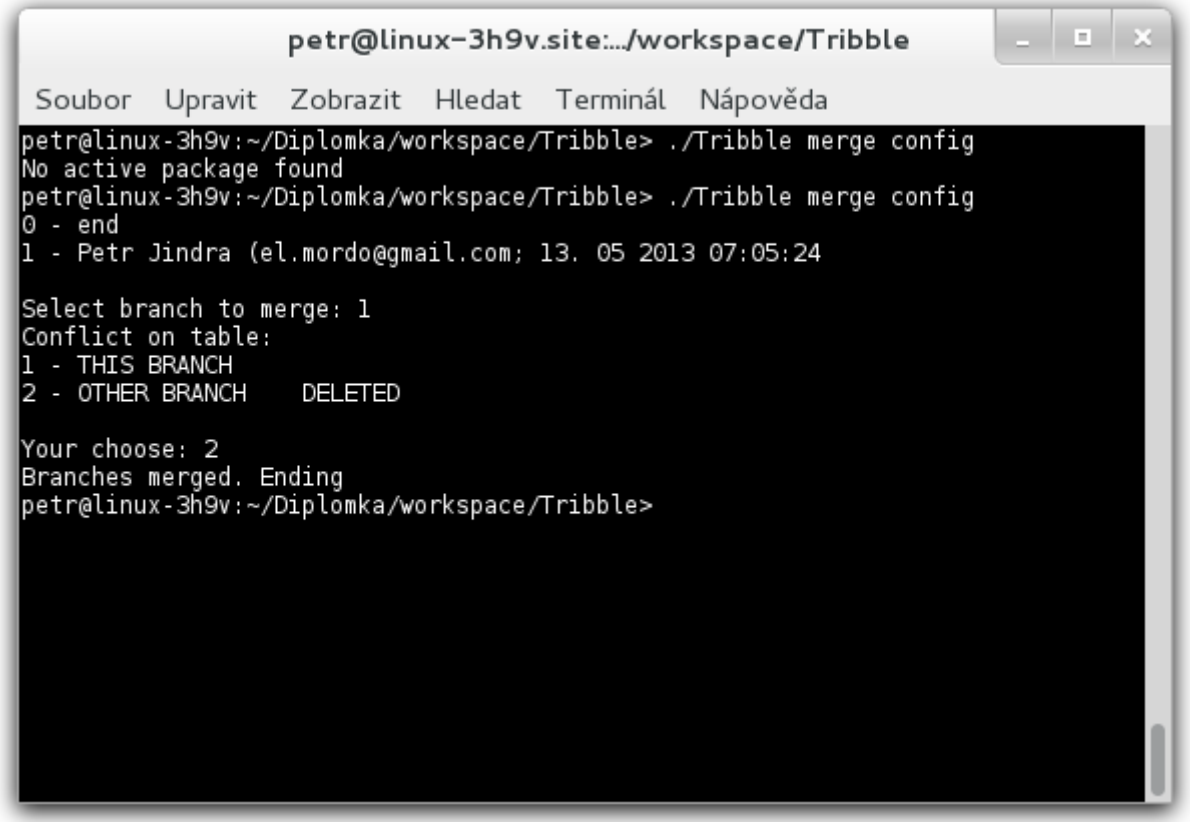
Pro sloučení dvou větví je připravena metoda merge. Metoda nejprve zkontroluje dostupné větve, které se mohou s aktuální větví sloučit a poté se pokusí sloučení realizovat.

V případě, že dojde ke konfliktům, je uživatel vyzván k jejich ručnímu řešení. Po dokončení slučování se vypíše notifikace a dojde k ukončení běhu programu.

Pokud není žádná dostupná větev nalezena, vypíše se notifikace a program skončí.



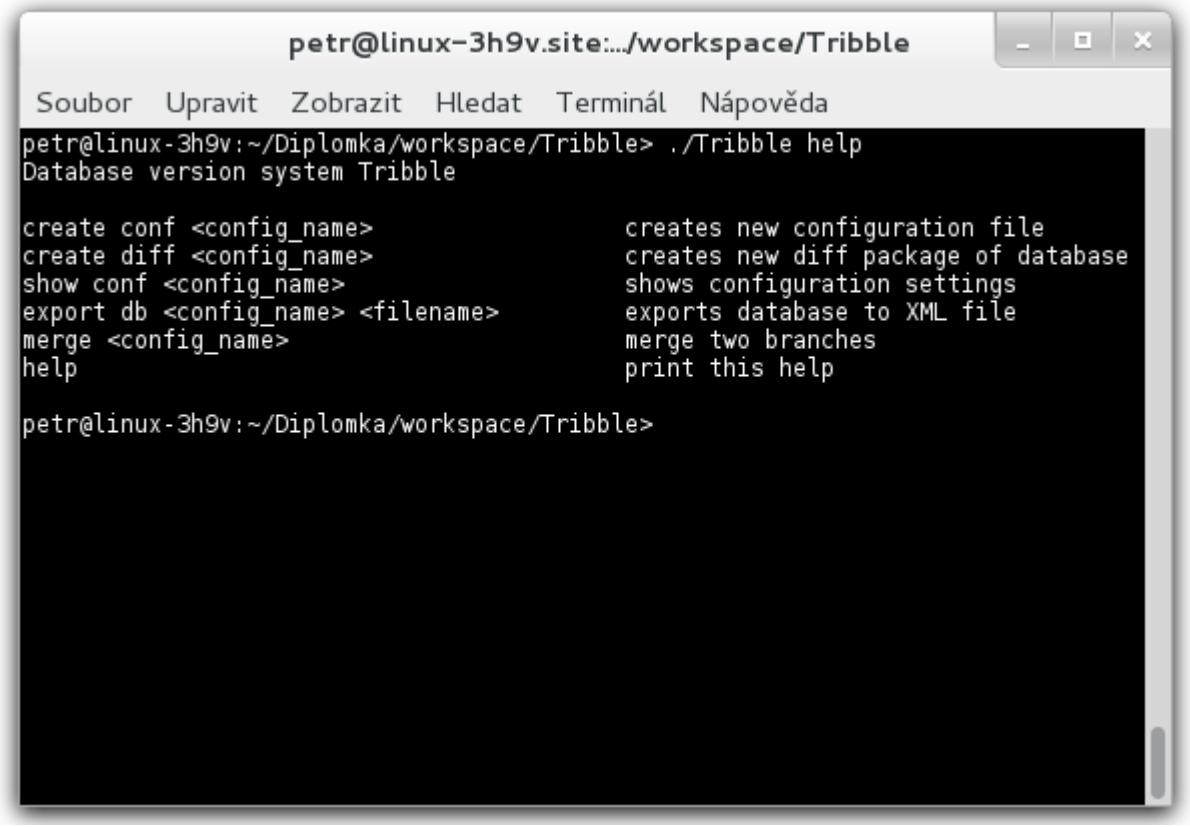
Obrázek 22: Žádné větve ke sloučení



Obrázek 23: Sloučení dvou větví s konfliktem

10.6 Help

Pokud uživatel zadá chybný příkaz nebo příkaz help, pak se vypíše nápověda se seznamem příkazů.



Obrázek 24: Nápověda programu

11 ZÁVĚR

Na začátku projektu byly vyhodnoceny největší problémy, které mohou nastat. S těmito problémy jsme se během vývoje skutečně setkali a bylo je nutné vyřešit, aby práce mohla pokračovat dál.

Zpracování SQL kódu bylo ve finále vyřešeno zásobníkovým konečným automatem. Vzhledem k nedostatku času nakonec nebyl realizován žádný adaptér, který by tento automat využíval. Přesto je připraven, jako samostatná knihovna, pro pozdější využití nebo pro samostatné využití v některém dalším projektu.

Připojování a komunikace s databází bylo vyřešeno pomocí abstraktní třídy, která definuje jednotné rozhraní ke všem možným typům databází. Navíc umožňuje i unifikovaný postup nastavení připojení z konfiguračního souboru.

K porovnávání změn byla navrhnuta specializovaná třída, která považuje starou i novou databázi za množiny a pomocí svých algoritmů popsaných v kapitole 9.1 provede operaci rozdílu těchto množin.

Tím se dostáváme k ukládání dat na disk. Pro tento účel byly zvoleny XML soubory, které jsou zabalené v souboru, jehož název je SHA-1 hash v šestnáctkové reprezentaci, a využity byly open sourcové knihovny libZip2 a libCrypto.

Opětovné načítání balíčků je prováděno najednou pro celý adresář, v němž jsou balíčky uloženy. Během procesu načítání jsou zároveň balíčky roztříděny podle větví a dojde mezi nimi k vytvoření vazeb rodič-potomek.

Při požadavku na strukturu databáze je její reprezentace v paměti konstruována stejným způsobem, jakým byla ukládána. Na začátku je prázdná databáze a postupně jsou na ni aplikovány změny tak, jak byly zapisovány balíčky na disk.

Podle předpokladu, uvedeného v kapitole 2, bylo skutečně jednou z největších výzev sloučení větví a vyhodnocení konfliktních změn. Tyto konflikty musely být nejprve nalezeny a poté muselo být provedeno jejich odstranění. Nalezení konfliktů bylo vyřešeno tak, že se nejprve zjistily všechny změny provedené na obou větvích od posledního sloučení a poté byla nad oběma větvemi provedena upravená množinová operace průnik. Pro detailní popis operace viz kapitolu 9.4. Průnikem jsou pak konflikty, které je potřeba ručně odstranit.

Odstraňování konfliktů bylo vyřešeno pravděpodobně nejjednodušší možnou cestou. Uživatel je dotázán na každý konflikt, který byl nalezen, a je požádán aby vybral, zda se použije změna, kterou provedl on, nebo zda se provede změna z vedlejší větve.

Jak bylo vytyčeno v zadání práce, výstupem projektu je program, který je schopen automaticky zjišťovat změny provedené od posledního spuštění a ukládat je do rozdílových balíčků. Program přímo neumožňuje distribuci změn ostatním členům týmu. Tento proces je nechán na jiné aplikace a prostředky. Klasickým příkladem takového prostředku verzovací systém pro soubory, který může sledovat adresář s balíčky a zapisovat ho do repozitáře na serveru, odkud si změny stáhnou kolegové.

Pro další rozvoj projektu bude nutné naprogramovat více databázových adaptérů, jako je PostreSQL, MS SQL Server nebo knihovna Sqlite. K tomuto účelu je připravená knihovna s konečným automatem, která byla zmíněna výše.

Dále by bylo vhodné implementovat další funkce, jako je například vizualizace průběhu vývoje větví a vizualizace provedených změn. S tímto úzce souvisí GUI, které by tyto informace zobrazilo podstatně přehledněji, než momentálně použitá konzole.

Velké použití by patrně nalezla i možnost přenášet vybraná testovací data. Aby bylo možné toto provést, bude nutné upravit konfigurační soubor. Ten bude obsahovat informace, která data uložit do balíčku, a která ne. Tato funkce bude další velká výzva.

Dle mého osobního názoru byly všechny body zadání splněny, a i přes tento fakt stále existují velké možnosti vývoje, kterým se může program vydat. Využití může nalézt jak pro výukové účely, tak i pro nasazení na skutečných projektech, kde má potenciál ušetřit spoustu

zbytečného času, který by programátoři strávili psaním SQL kódu migrací.

SEZNAM POUŽITÉ LITERATURY

[1] Ruckusing: "Database Migrations" for PHP5, ala Ruby on Rails ActiveRecord Migrations. [online]. [cit. 2012-11-28]. Dostupné z: http://code.google.com/p/ruckusing/wiki/CompleteExamples

[2] SHELDON, Robert. *SQL: začínáme programovat*. 1. vyd. Praha: Grada, 2005, 499 s. ISBN 80-247-0999-6.

[3] GRUBER, Martin. *Mistrovství v SQL*. Praha: SoftPress, c2004, 480 s. ISBN 80-86497-62-3.

[4] SCHNEIDER, Robert D. *MySQL: oficiální průvodce tvorbou, správou a laděním databází*. 1. vyd. Praha: Grada, 2006, 372 s. ISBN 80-247-1516-3

[5] MOMJIAN, Bruce. *PostgreSQL: praktický průvodce*. Vyd. 1. Brno: Computer Press, 2003, xxii, 402 s. ISBN 80-7226-954-2

[6] LACKO, Ľuboslav. *Oracle: správa, programování a použití databázového systému : platné pro verze Oracle9i, 8i a 8*. Vyd. 1. Praha: Computer Press, 2002, xiv, 464 s. ISBN 80-7226-699-3

[7] MORKES, David. *Microsoft SQL Server 2000: tvorba, úprava a správa databází*. 1. vyd. Praha: Grada, 2004, 228 s. ISBN 80-247-0732-2.

[8] FAISON, Ted. *Object-Oriented State Machines*. vyd. 116 Agostino Irvine, Ca 92714 , (714) 261-2752 (H) (714) 753-0777 (W) , June 7, 1993

[9] SQL Power Architect: Data Modeling & Profiling Tool. SQL POWER SOFTWARE. [online]. [cit. 2012-11-23]. Dostupné z: http://www.sqlpower.ca/page/architect

[10] HIPP, D. Richard. SQLite. [online]. [cit. 2013-05-20]. Dostupné z: http://www.sqlite.org/about.html

[11] SHOW CREATE TABLE Syntax. ORACLE CORPORATION. [online]. [cit. 2013-05-20]. Dostupné z: http://dev.mysql.com/doc/refman/5.6/en/show-create-table.html

[12] PostgreSQL 9.2.4 Documentation. POSTGRESQL GLOBAL DEVELOPMENT GROUP. [online]. [cit. 2013-05-20]. Dostupné z: http://www.postgresql.org/docs/9.2/static/index.html

[13] PostgreSQL 9.2.4 Documentation. POSTGRESQL GLOBAL DEVELOPMENT GROUP. [online]. [cit. 2013-05-20]. Dostupné z: http://www.postgresql.org/docs/8.4/static/app-pgdump.html

[14] SQLite: Frequently Asked Questions. HIPP, D. Richard. [online]. [cit. 2013-05-20]. Dostupné z: http://www.sqlite.org/faq.html#q7

[15] Fun with C++: std::list vs. std::vector Performance. [online]. [cit. 2013-05-20]. Dostupné z: http://yaserzt.com/blog/archives/615

[16] http://www.sqlite.org/faq.html

[17] CHACON, Scott. *Pro Git*. Praha: CZ.NIC, c2009, 263 s. ISBN 978-80-904248-1-4

[18] Fun with C++: std::list vs. std::vector Performance. In: [online]. [cit. 2013-05-20]. Dostupné z: http://yaserzt.com/blog/wp-content/uploads/2012/09/yzt_vector_vs_list_chart_01.png

[19] MLÝNKOVÁ, Irena a Jaroslav POKORNÝ. *XML technologie: principy a aplikace v praxi*. 1. vyd. Praha: Grada, 2008, 267 s. ISBN 978-80-247-2725-7

[20] INFORMATION_SCHEMA Tables. ORACLE. [online]. [cit. 2013-05-20]. Dostupné z: http://dev.mysql.com/doc/refman/5.0/en/information-schema.html

[21] Changes in MySQL 5.0.2. ORACLE CORPORATION. [online]. [cit. 2013-05-20]. Dostupné z: http://dev.mysql.com/doc/relnotes/mysql/5.0/en/news-5-0-2.html

[22] Libzip: libzip(3). BARON, Dieter a KLAUSNER. [online]. [cit. 2013-05-20]. Dostupné z: http://www.nih.at/libzip/libzip.html

[23] Boost C++ library: Filesystem Library Version 3. [online]. [cit. 2013-05-20]. Dostupné z: http://www.boost.org/doc/libs/1_53_0/libs/filesystem/doc/index.htm

[24] The Boost C++ Libraries. SCHÄLING, Boris. [online]. [cit. 2013-05-20]. Dostupné z: http://en.highscore.de/cpp/boost/

[25] PECINOVSKÝ, Rudolf. *Návrhové vzory: [33 vzorových postupů pro objektové programování]*. Vyd. 1. Brno: Computer Press, 2007, 527 s. ISBN 978-80-251-1582-4

[26] PRATA, Stephen. *Mistrovství v C++*. 2. aktualiz. vyd. Brno: Computer Press, 2004, xxvi, 1006 s. ISBN 80-251-0098-7

PŘÍLOHA 1 – DTD XML SOUBORU AUTOMATU

```
<!ENTITY % class "class CDATA #REQUIRED">
<!ENTITY % name "name CDATA #REQUIRED">
<!ENTITY % isfinal "isfinal (true|false) #REQUIRED">
<!ELEMENT fa (machine+)>
<!ATTRLIST fa
    starter CDATA #REQUIRED
>
<!ELEMENT machine (state+,stateagregated*,transition*)>
<!ATTRLIST machine
    starter CDATA #REQUIRED
>
<!ELEMENT state (processor)>
<!ATTRLIST state
    %name
    %isfinal
>
<!ELEMENT processor (param*)>
<!ATTRLIST processor
    %class
>
<!ELEMENT param EMPTY>
<!ATTRLIST param
    %name
    value CDATA #IMPLIED
>
<!ELEMENT stateagregated (assembler)>
<!ATTRLIST stateagregated
    agrmachine CDATA #REQUIRED
>
<!ELEMENT assembler (param*)>
<ATTRLIST assembler
    %class
```

```
>
<!ELEMENT transition (loader,filter*,test*)>
<!ATTRLIST transition
    from CDATA #REQUIRED
    to CDATA #REQUIRED
    priority CDATA #REQURED
>
<!ELEMENT loader (param*)>
<!ATTRLIST loader
    %class
>
<!ELEMENT filter (param*)>
<!ATTRLIST filter
    %class
>
<!ELEMENT test (param*)>
<ATTRLIST test
    %class
```

>PŘÍLOHA 2 – DTD XML SOUBORU STRUKTURY DATABÁZE

```
<!ENTITY % comment "comment CDATA #IMPLIED">
<!ENTITY % name "name CDATA #REQUIRED">
<!ELEMENT database (table,view,procedure,function)*>
<!ATTLIST database
    %comment
    >
<!ELEMENT table (column,index,trigger,check,foreignkey)*>
<!ATTLIST table
    %comment
    %name
    storage CDATA #IMPLIED
    charset CDATA #IMPLIED
    collate CDATA #IMPLIED
    >
<!ELEMENT view (code)>
<!ATTLIST view
    %comment
    %name
    >
<!ELEMENT foreignkey (refcolumn+,tarcolumn+)>
<!ATTLIST foreignkey
    %comment
    %name
    table CDATA #REQUIRED
    reftable CDATA #REQUIRED
    ondelete CDATA #IMPLIED
    onupdate CDATA #IMPLIED
    >
<!ELEMENT column EMPTY>
<!ATTLIST column
    %name
    %comment
```

```
    datatype CDATA #REQUIRED
    extra CDATA #IMPLIED
    notnull CDATA #IMPLIED
    default CDATA #IMPLIED
    >
<!ELEMENT key (tarcolumn)>
<!ATTLIST key
    %comment
    %name
    type CDATA #REQUIRED
    method CDATA #IMPLIED
    >
<!ELEMENT refcolumn EMPTY>
<!ATTLIST refcolumn
    %name
    >
<!ELEMENT tarcolumn EMPTY>
<!ATTLIST tarcolumn
    %name
    >
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
    %name
    value CDATA #IMPLIED
    >
<!ELEMENT code CDATA>
<!ELEMENT trigger (code)>
<!ATTRLIST trigger
    %name
    %comment
    time CDATA #REQUIRED
    operation CDATA #REQUIRED
>
```

```
<!ELEMET procedure (code,parameter*)>
<!ATTRLIST procedure
    %name
    access CDATA #REQURED
>
<!ELEMENT function (code,parameter*)>
<!ATTRLIST function
    %name
    datatype CDATA #REQUIRED
    access CDATA #REQURED
>
<!ELEMENT procparam EMPTY>
<!ATTRLIST procparam
    %name
    datatype CDATA #REQUIRED
    direction (im|out|inout) #REQUIRED
>
<!ELEMENT check (code)>
<!ATTRLIST check
    %name
>
```

PŘÍLOHA 3 – DTD XML SOUBORU META DAT BALÍČKU

```
<!ELEMENT meta (user,email,time,branch,conflicts?,parents)>
<!ELEMENT user CDATA>
<!ELEMENT email CDATA>
<!ELEMENT time (year,month, day,hour,minute,second)>
<!ELEMENT branch CDATA>
<!ELEMENT conflicts (conflict*)>
<!ELEMENT parents (parent*)>
<!ELEMENT year CDATA>
<!ELEMENT month CDATA>
<!ELEMENT day CDATA>
<!ELEMENT hour CDATA>
<!ELEMENT minute CDATA>
<!ELEMENT second CDATA>
<!ELEMENT conflict CDATA>
<!ATTRLIST conflict
    id CDATA #REQUIRED
    db CDATA #REQUIRED
    method (current|other) #REQURED
>
<!ELEMENT parent CDATA>
```