PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Vytvoření projektu pro zpracování výměnného formátu Informačního systému katastru nemovitostí v jazyce Java

2014

Petr Freiberg

Anotace

*Prostřednictvím výměnného formátu katastrální pracoviště předávají nebo přebírají data Informačního systému katastru nemovitostí o objektech katastru nemovitostí, a to jak popisné, tak i grafické informace. Bakalářská práce se zabývá načtením CSV ze souborů výměnného formátu, zpracováním a uložením do Oracle Database. Ukládání do objektově relační databáze je realizováno přes SQL*Loader, nebo Java Database Connectivity. Výsledkem je volně šiřitelný program napsaný v jazyce Java, vydaný pod licencí MIT, který je již v době psaní práce používán v ostrém provozu firmou Corpus Solutions a.s. Výsledky práce mohou být díky využití rozmanitých programátorských technik inspirací pro ty, kteří se chystají vybudovat rozsáhlý, avšak snadno udržitelný program zpracovávající velké množství textových dat.*

V rámci univerzity děkuji vedoucí bakalářské práce doc. Ing. Lence Carr-Motyčkové, CSc. za zastřešení mnou navržené práce a Mgr. Petrovi Krajčovi, Ph.D., že mne v druhém ročníku zasvětil do Javy SE. Ve firmě Corpus Solutions a.s. pak Ing. Zdeňkovi Zíchovi za excelentní analytickou podporu, Janu Šnajdarovi a Ing. Petru Šomkovi za neocenitelné programátorské rady, a v neposlední řadě také Stanislavu Horehleďovi za vedení projektu.

Obsah

1. Úvod **9**

2. Příprava projektu **11**
2.1. Předchozí stav . . . 11
2.2. Spolupráce s firmou Corpus Solutions a.s. . . . 11
2.3. Analýza a vedení projektu . . . 12

3. Milníky a cíle **13**
3.1. Milníky . . . 13
3.1.1. Verze 1.0 . . . 13
3.1.2. Verze 2.0 . . . 13
3.2. Cíle . . . 13
3.2.1. Rychlost . . . 13
3.2.2. Kompletnost a validita . . . 14
3.2.3. Použitelnost . . . 14
3.2.4. Rozšiřitelnost a srozumitelnost . . . 15
3.2.5. Pružnost . . . 15
3.2.6. Robustnost . . . 15

4. Struktura výměnného formátu **16**
4.1. Kódování souboru . . . 16
4.2. Části výměnného formátu . . . 16
4.2.1. Hlavička . . . 16
4.2.2. Datový blok . . . 17
4.2.3. Konec souboru . . . 18
4.3. Rozdíl mezi stavovými a změnovými daty . . . 18

5. Programovací jazyk a použité nástroje **19**
5.1. Java SE 1.7 . . . 19
5.2. Eclipse . . . 19
5.3. Apache Maven . . . 19
5.4. GitHub . . . 20
5.5. Dropbox . . . 20

6. Licence MIT **21**

7. Načtení a zpracování dat **22**
7.1. Doménové třídy . . . 22
7.2. Načtení souboru do doménových tříd . . . 23
7.2.1. Lexikální analýza . . . 23
7.2.2. Speciální případy . . . 24
7.2.3. Syntaktická analýza . . . 25

7.3. Dávkové zpracování 26

8. Vícevláknové zpracování **28**
8.1. Technika producenta a konzumenta 28

9. Exportování dat **31**
9.1. Abstraktní továrna 31
9.2. Oracle Database 32
9.3. Export přes SQL*Loader 32
9.3.1. Kontrolní soubor SQL*Loaderu 32
9.3.2. Vstupní data a datové soubory 34
9.3.3. Průběh exportu přes SQL*Loader 34
9.3.4. Návrhový vzor Template method 35
9.3.5. Import vyexportovaných dat 36
9.4. Export přes Java Database Connection 37
9.4.1. Změnová data a JDBC 37
9.4.2. Analýza 38
9.4.3. Algoritmy pro zpracování změnových dat 38
9.4.4. Průběh exportu přes JDBC 38
9.4.5. Rozhraní a společná logika pro JDBC (1. úroveň) 39
9.4.6. Algoritmy zpracovávající doménové třídy (2. úroveň) . . . 41
9.4.7. Exportéry jednotlivých doménových tříd (3. úroveň) 42
9.4.8. Optimalizace a bezpečnost 43

10. Programátorská dokumentace **44**
10.1. Struktura projektu 44
10.2. UML diagram a třídy programu 45
10.2.1. cz.pfreiberg.knparser 45
10.2.2. cz.pfreiberg.knparser.domain 46
10.2.3. cz.pfreiberg.knparser.domain.* 46
10.2.4. cz.pfreiberg.knparser.parser 47
10.2.5. cz.pfreiberg.knparser.exporter 47
10.2.6. cz.pfreiberg.knparser.exporterfactory 47
10.2.7. cz.pfreiberg.knparser.exporter.oracleloaderfile 48
10.2.8. cz.pfreiberg.knparser.exporter.oracledatabase 48
10.2.9. cz.pfreiberg.knparser.util 50

11. Uživatelská dokumentace **51**

12. Výsledky testů a ostrý provoz **52**
12.1. Externí testování na ČÚZK 52
12.2. Výsledky pro verzi 1.0 53
12.3. Výsledky pro verzi 1.1 53

12.4. Výsledky pro verzi 2.0 54

13. Budoucí práce na programu **55**

Závěr **56**

Reference **57**

A. Licence použitých zdrojů **58**

B. Obsah přiloženého CD a USB flash paměti **59**
B.1. Struktura CD 59
B.2. Struktura USB flash paměti 59

C. Spuštění v programu VirtualBox **60**
C.1. Postup spuštění programu 60
C.2. Poznámky 61

Seznam obrázků

1. Jednovláknové zpracování souboru 29
2. Vícevláknové zpracování souboru 29
3. UML diagram projektu. Pro přehlednost bez abstraktních tříd z JDBC exportéru, které nezpracovávají doménovou třídu `Parcely`. 45

Seznam tabulek

1. Skupiny datových bloků 22

1. Úvod

Přestože se dnes zájem o data katastru nemovitostí[1] zaměřuje téměř výhradně na veřejně dostupná data projektu RÚIAN[2], který je jedním ze základních registrů veřejné správy, existují případy užití, ve kterých je informační hodnota dat RÚIAN nedostatečná. Tyto nedostatky jsou patrné především tehdy, když se snažíme dohledat informace o vlastnictví.

Český úřad zeměměřický a katastrální vedle RÚIANu provozuje ISKN[3], který lze považovat za jeden z datově nejobsáhlejších systémů státní správy. Jedná se o informační systém, jehož hlavním cílem je usnadnit výkony spojené s vykonáváním státní správy katastru nemovitostí, a to tak, že shromažďuje, spravuje a poskytuje informace o nemovitostech, jejich geometriích, poloze či vlastnických právech. ISKN je podstatně starším systémem než RÚIAN – 2001 oproti roku 2012.

Ačkoliv se určitá data z ISKN automaticky kopírují do RÚIANu (například parcely mimo zjednodušené evidence[4]), tak přesto poskytuje podstatně rozsáhlejší informační bázi. Data lze z ISKN získat dvojím způsobem:

- dálkovým přístupem
- výměnným formátem

Dálkový přístup je určen těm, kteří mají připojení k síti Internet a jsou na Českém úřadu katastrálním a zeměměřičském (dále ČÚZK) zaregistrováni. Data jsou poskytnuta buď zdarma (v případě, že se jedná o státní správu), nebo za poplatek.

Výměnný formát, jehož zpracováním se tato práce zabývá, je poskytován (opět za stejných podmínek) ve formě souborů nového výměnného formátu. Oficiální podpora předchozí verze byla ČÚZK zaručena do roku 2003. Výhodami oproti předešlé verzi je například větší množství tabulek a tedy i informací, velmi dobrá provázanost mezi tabulkami, dostupnost číselníků a mnoho dalšího. Pokud bude v textu zmíněn výměnný formát (a nebude explicitně uvedeno jinak), pak je tím myšlen nový výměnný formát.

Vytvoření volně šiřitelného programu, který by soubory výměnného formátu načetl a umožnil jejich vyexportování více způsoby, bylo velmi žádoucí. V rámci této bakalářské práce vznikl na poptávku firmy **Corpus Solutions a.s.** program,

[1] Veřejný soubor údajů o nemovitostech v České republice. Obsahuje jejich popis, geometrické a polohové určení, vlastnické práva, ... Tyto informace slouží od ochrany práv nemovitostí přes daňové a poplatkové účely po oceňování nemovitostí a vědecké či statistické potřeby.

[2] Registr územní identifikace, adres a nemovitostí

[3] Informační systém katastru nemovitostí

[4] Zemědělské a lesní parcely s omezenou množinou údajů, které byly právoplatným vlastníkům odebrány během kolektivizace a sloučeny do větších celků. Hranice těchto parcel nejsou součástí katastrální mapy.

který umožňuje načtení dat výměnného formátu a uložení do Oracle Database (prostřednictvím SQL*Loaderu, nebo Java Database Connectivity). Velmi důležitým aspektem programu, vyvíjeným pod názvem **KnParser**, je, že jej lze velmi snadno rozšířit, aby dokázal data exportovat do jiných systémů. Ve spojení s licencí MIT nad ním tedy mohou i další jedinci a subjekty vybudovat své projekty.

2. Příprava projektu

O katastru nemovitostí vzniklo několik bakalářských i diplomových prací. V následující kapitole si ukážeme, v čem se tato bakalářská práce liší a jak došlo ke vzniku jejího zadání.

2.1. Předchozí stav

Základními datovými sadami (mezi něž spadají i data katastru nemovitostí) se dlouhodobě zabývá oddělení geomatiky na Západočeské univerzitě v Plzni. Série článků a dále bakalářských a diplomových prací vyústila v projekt „Otevřený katastr“[5], který si dal za cíl vytvořit rozhraní pro přístup k datům katastru nemovitostí. V rámci něj vznikly nástroje pro import dat výměnného formátu do prostorové databáze PostGIS, což je rozšíření databázového systému PostgreSQL o geografické objekty. Druhým příspěvkem je pak vizualizace katastrálních dat prostřednictvím mapového serveru UMN MapServer, jenž umožňuje sdílet geografické informace na internetu.

Dalším nástrojem, který se dotýká tématiky zpracování dat katastru, je „ruian2pgsql“[6]. Jak už vyplývá z názvu, jedná se o program, který pracuje nad RÚIANem. Vychází tedy jen z omezené množiny informací.

Samostatně se zpracováním výměnného formátu zabývali i další jedinci.[7]

Žádný aktuální projekt si však nedal za cíl vypracovat volně šiřitelný program, který umožní data nejen načíst, ale především snadno převádět do dalších systémů. Ostatní projekty si nejčastěji dávaly za cíl ukládání a zpracování dat v geodatabázích[8]. Případně pracovaly i s omezenou množinou dat, kdy data sloužila pouze jako prostředek k představení funkčnosti projektu, jehož hlavním cílem nebylo zpracování výměnného formátu, ale kupříkladu ukázka vizualizace pozemků[9].

2.2. Spolupráce s firmou Corpus Solutions a.s.

Práce na projektu mi byla nabídnuta, jakožto zaměstnanci firmy Corpus Solutions a.s., v polovině října 2013. Součástí nabídky bylo, že mohu program vyvíjet

[5] JEDLIČKA, K., JEŽEK, J., PETRÁK, J. Otevřený katastr - svobodné internetové řešení pro prohlížení dat výměnného formátu katastru nemovitostí. 2007. Stať ve sborníku. s. 111-117. České vysoké učení technické v Praze, Fakulta stavební.

[6] https://github.com/fordfrog/ruian2pgsql

[7] DROZD, Pavel. Geodatabáze katastru nemovitostí [online]. 2008. Diplomová práce. Masarykova univerzita, Fakulta informatiky.

[8] ORÁLEK, Jakub. Možnosti využití nekomerčního geografického software pro tvorbu prostorového rozhraní informačního systému malé obce [online]. 2006. Diplomová práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd.

[9] RŮŽIČKA, Zdeněk. Vizualizace změn druhů pozemků u parcel katastru nemovitostí [online]. 2010. Bakalářská práce. České vysoké učení technické v Praze, Fakulta stavební.

pod svobodnou licencí a v rámci mé bakalářské práce.

Výhody jsou zřejmé: pracuje se na reálném problému a práce bude mít opravdový přínos. Neméně důležitý je i fakt, že se jedná o propojení akademické a firemní sféry.

2.3. Analýza a vedení projektu

Před zahájením projektu byl analýzou pověřen zaměstnanec firmy Ing. Zícha. Jeho úkolem bylo definovat vlastnosti, které má výsledný program splňovat. Během vývoje pak provádět průběžné testy na ČÚZK a především u verze 2.0 pomoci vytvořit postupy zpracování dat nad databází. Bez rozsáhlých znalostí, které o katastru nemovitostí má, by mohl tento projekt jen těžko vzniknout.

Zastřešením projektu byl v rámci firmy pověřen Stanislav Horehleď.

3. Milníky a cíle

Před začátkem vývoje byly vytyčeny dva milníky, které bylo potřeba dodržet a několik cílů, kterých by měl hotový program dosáhnout.

3.1. Milníky

Milníky bylo potřeba definovat jasně a s ohledem na termíny. Na jejich dodržení spoléhaly v rámci firmy další týmy, které s daty zpracovanými programem dále pracovaly.

3.1.1. Verze 1.0

Program zpracovává všechny datové bloky výměnného formátu a data exportuje ve formátu, který umí zpracovat SQL*Loader.

Termín: polovina ledna 2014

3.1.2. Verze 2.0

Rozšíření o přímé exportování dat do Oracle Database přes Java Database Connectivity.

Termín: začátek března 2014

3.2. Cíle

Cíle nebyly jako milníky zatíženy termíny, ale počítalo se s tím, že se jim bude program během celého vývoje pomalu přibližovat, až je ideálně ve verzi 2.0 naplní.

3.2.1. Rychlost

V každém čtvrtletí ČÚZK vydává aktualizovaná stavová data[10], které dnes mají okolo 145 GB (a jejich velikost pomalu narůstá). Měsíčně jsou pak vydávána změnová data[11]. Jejich velikost závisí na počtu změn v daném měsíci. Například za únor 2014 měly velikost 6 GB. Uvedená velikost jak u stavových, tak změnových dat, platí za celý ČÚZK:

- ČÚZK tvoří 97 katastrálních pracovišť
- každé pracoviště poskytuje svou část dat

[10] Všechna data platná k určitému časovému okamžiku
[11] Data, která se změnila za dané časové období. Například mezi únorem a březnem 2014.

- tuto část předává ve třech souborech výměnného formátu (popisné informace, geodetické informace a definiční body parcel)

Velikost je tedy součtem všech katastrálních pracovišť. Vzhledem k objemu musí být zpracování rychlé a efektivní. Kritickou částí programu je:

- průchod souborem výměnného formátu a načítání jednotlivých řádků
- zpracování řádku
    - získání informací z textového řetězce
    - určení typu řádku (aby bylo možné rozhodnout o místě uložení)
    - dočasné uložení řádku v programu
- exportování zpracovaných dat

Jak jsem uvedl výše, každé katastrální pracoviště rozděluje data do třech souborů. Přesto jsou jednotlivé soubory poměrně objemné. Největší soubory výměnného formátu dosahují velikosti až 4,5 GB a obsahují přes 40 miliónů řádků. Omezením, které soubory o takovém objemu přinášejí, je nemožnost jejich zpracování v celku. To řeší implementace technologie dávkového zpracování.[12]

Velký vliv na rychlost má implementace vícevláknového zpracování. Jedno vlákno se stará o načítání a ukládání dat, druhé pak o export.

3.2.2. Kompletnost a validita

Povaha dat, které program zpracovává, klade vysoké nároky na kompletnost a validitu. Výpadek byť jediného řádku může znehodnotit celkový import dat. Samozřejmě nelze zaručit, že během přímého importu dat do databáze například nedojde k výpadku sítě. Pokud se tak však stane, je nutné, aby se o tom uživatel, který import provádí, dozvěděl. Základem je víceúrovňové žurnálování, ke kterému je použita knihovna `log4j`[13].

3.2.3. Použitelnost

Jelikož se jedná o specializovaný nástroj, který bude používat poměrně úzká skupina uživatelů, je výsledná podoba odrazem konzultací s nimi vedenými. Jako nejlepší řešení se jeví předávání parametrů přes příkazový řádek a textový soubor s nastavením. Příkazovým řádkem se předávají často se měnící parametry (cesta k souboru, typ exportu). Textovým souborem pak počet řádků, které budou v jedné dávce zpracovány a případně údaje potřebné k napojení do databáze.

[12]Zde uvažujeme běžné podmínky. Pokud by program běžel na serveru s velkým množstvím operační paměti, je velmi jednoduché v přiloženém souboru s nastavením upravit velikost jednotlivých dávek.

[13]Žurnálovací knihovna pro Javu, který je vyvíjena v rámci Apache Software Foundation. Program ji využívá k ukládání informací o své činnosti a běhu.

3.2.4. Rozšiřitelnost a srozumitelnost

Poptávka o export do dalších formátů by měla být splnitelná s co nejmenší pracností. S tím jde ruku v ruce srozumitelnost. Program je strukturován do několika bloků, kdy třídy v jednotlivých blocích spolu logicky souvisí. Dodržování konvencí jazyka Java a jednotného pojmenování napříč programem, použití návrhových vzorů[14] (Abstraktní továrna[15], ...) a dalších prvků přispívá k tomu, že jedinci a subjekty, kteří se nepodíleli na samotném vývoji, mohou snadno upravovat program a vybudovat nad ním vlastní projekty.

3.2.5. Pružnost

Tento požadavek úzce souvisí s předchozím bodem. Výměnný formát je stále ve vývoji, stejně jako právní systém České republiky. Na sklonku roku 2013 kupříkladu ČÚZK reagoval na novelu katastrálního zákona, která sebou mimo jiné přinesla i úpravy ve výměnném formátu. Byl přidán nový datový blok „Vazba JPV k jinému věcnému právu", další datové bloky („Vlastnictví", „Parcely", ...) byly rozšířeny o nové položky a tak dále. Případné úpravy, které mohou znamenat i rozšíření o načítání a ukládání nových datových bloků, tedy musí být co nejsnadnější.

3.2.6. Robustnost

Program implementuje základní ochrany při práci s externími zdroji (soubor s daty a nastavením, komunikace s databází). Při zpracování poškozeného řádku, kdy například nejsou správně uzavřeny uvozovky, je snaha tento problém eliminovat v rámci řádku, upozornit na konkrétní řádek uživatele a pokračovat ve zpracovávání.

Pokud nastane chyba, ze které není možného zotavení (například nejsou uvedeny informace pro připojení k databázi a je vybrán export do databáze), program informuje uživatele a standardně se ukončí. Stejně se zachová i ve chvíli, kdy se jedná o chybu, u které nemůžeme zaručit, že budou data vyexportována validně (každý výměnný soubor ve své hlavičce obsahuje informace o znakové sadě. Pokud není rozpoznána, je program ukončen).

[14] Návrhový vzor je obecným řešením určitého typu problému.

[15] Umožňuje až za běhu programu rozhodnout, jaké konkrétní objekty budou vytvářeny. V našem případě do jakého formátu/systému budou data vyexportována.

4. Struktura výměnného formátu

Informační systém katastru nemovitostí využívá výměnný formát ke vzájemné výměně informací s jinými systémy zpracování dat. Kapitola je stručným úvodem do tohoto formátu. Pro podrobný popis čtenáře odkazuji na oficiální dokumenty ČÚZK[1, 2].

4.1. Kódování souboru

Soubor výměnného formátu (.VFK) je dodáván s kódováním češtiny dle ISO 8859-2. Ve výjimečných a zdůvodněných případech také ve Windows-1250.

4.2. Části výměnného formátu

Jedná se o textový soubor, který se skládá z:

- hlavičky
- datových bloků
- koncového znaku (&K)

4.2.1. Hlavička

Sestává se z několika řádků, které obsahují informace o souboru. Každý jednotlivý řádek je uvozen návěstím `&H`:

- označení verze výměnného formátu
- datum a čas vytvoření souboru
- původ dat
- označení kódové stránky
- seznam skupin datových bloků souboru
- jméno osoby, která soubor vytvořila
- časovou podmínku použitou pro vytvoření souboru
- omezující podmínku – katastrální území
- omezující podmínku – oprávněné subjekty
- omezující podmínku – parcely
- omezující podmínku – polygon

Program z hlavičky využívá „Označení kódové stránky“ pro určení druhu kódování a „Omezující podmínku - katastrální území“, ze které zjišťuje, zda se jedná o stavová, nebo změnová data. Tato informace je důležitá při exportu přes SQL*Loader, kdy mají v případě změnových dat výstupní soubory prefix "TMP_".

Příklad hlavičky z ukázkových dat ČÚZK:

```
&HVERZE;"5.0"
&HVYTVORENO;"13.01.2014 11:46:30"
&HPUVOD;"ISKN"
&HCODEPAGE;"WE8ISO8859P2"
&HSKUPINA;"NEMO";"JEDN";"BDPA";"VLST"
&HJMENO;"Kokeš Petr Ing."
&HPLATNOST;"13.01.2014 11:41:00";"13.01.2014 11:41:00"
&HZMENY;0
&HNAVRHY;0
&HPOLYG;0
&HKATUZE;KOD N6;OBCE_KOD N6;NAZEV T48;PLATNOST_OD D;PLATNOST_DO
```

4.2.2. Datový blok

Každý datový blok se skládá z jednoho uvozujícího řádku, který obsahuje seznam atributů s datovými typy. Po něm následují samotné datové řádky. Pořadí hodnot v datových řádcích odpovídá uvozujícímu řádku.

Uvozující řádek datového bloku začíná návěstím `&B` (blok výměnného formátu). Následuje zkratka, která identifikuje daný blok. V příkladu uvedeném níže je to `ZPOCHN` (datový blok „ZP_OCHRANY_NEM“, Číselník způsobů ochrany nemovitosti). Za ní jsou seřazeny jednotlivé atributy, které jsou od sebe odděleny středníkem. Atribut se skládá z názvu a typu.
Pokud se jedná o číselný (N) nebo textový (T) atribut, pak za typem dále najdeme číselnou hodnotu, která udává:

- číslo: maximální počet číslic. V případě desetinného čísla číslice před desetinnou tečkou určuje maximální počet číslic položky a číslice za desetinnou tečkou určuje počet desetinných míst.
- text: maximální délku textového řetězce

Uvozující řádek:

```
&BZPOCHN;KOD N4;NAZEV T60;PLATNOST_OD D;PLATNOST_DO D;POZEMEK
T1;BUDOVA T1;JEDNOTKA T1;NEMOCHR N3
```

Datový řádek je uvozen návěstím `&D` (data výměnného formátu). Stejně jako u uvozovacího řádku následuje zkratka, která identifikuje daný blok. Za ní jsou seřazena vlastní data, která jsou od sebe oddělena středníkem. Jeden takto

zpracovaných řádek se rovná jednomu objektu. Konec řádku je označen sekvencí znaků `<CR><LF>`[16].

V datových řádcích se vyskytují tři datové typy:

- **Text**: umístěn v uvozovkách. V případě, že se v textové položce vyskytují uvozovky, jsou zdvojeny.
- **Číslo**: kladná čísla jsou exportována bez znaménka, záporná čísla jsou exportována se znaménkem mínus před číslem. Nevýznamné nuly před ani za číslem se neexportují (0.53 jako .53 a 17.00 jako 17). Desetinná čísla jsou oddělována desetinnou tečkou.
- **Datum**: umístěno v uvozovkách. Má tvar DD.MM.YYYY HH:MI:SS. Například 22. 11. 2013 06:25:17.

Řádky s daty:

```
&DZPOCHN;8;"národní park - III.zóna";"01.01.1993
00:00:00";"";"a";"a";"a";2
&DZPOCHN;20;"vnitřní území lázeňského místa";"15.10.1998
00:00:00";"";"a";"a";"a";10
```

4.2.3. Konec souboru

Konec souboru je uvozen návěstím `&K`.

4.3. Rozdíl mezi stavovými a změnovými daty

Stavová data jsou platná k určitému datumu a obsahují informace o všech objektech daného katastrálního území. Například ze stavových dat k 1. 10. 2013 dokážeme určit, jak přesně vypadalo katastrální území k tomuto dni.

Změnová data obsahují pouze objekty, které byly dotčeny změnou během období udaného v hlavičce. Například změnová data za období 1. 10. 2013 až 1. 11. 2013 obsahují jen ty objekty, které na daném katastrálním území v tomto časovém rozmezí prošly změnou.

Ze změnových dat tedy vyčteme vývoj v čase, ze stavových ne. Avšak pouze ze změnových dat nezjistíme celkový stav katastrálního území. Mají tedy význam pro ty, kteří již mají stav k určitému datu (za stavových dat) a údaje chtějí pouze aktualizovat. To je výhodnější jak z finančního, tak praktického hlediska. Stavová data se importují do prázdné databáze, změnovými naproti tomu pouze aktualizujeme již vložené hodnoty.

K tomuto se ještě podrobněji vrátíme v kapitole 9., která pojednává o exportování dat přes Java Database Connectivity.

[16] Carriage return a line feed dohromady tvoří sekvenci znaků, která v počítačovém souboru běžně označuje konec řádku

5. Programovací jazyk a použité nástroje

Výběr správných nástrojů pro podporu vývoje ve zvoleném programovacím jazyku dokáže ušetřit spoustu času. Nyní stručně popíši jednotlivé nástroje, které jsem použil při tvorbě programu. Uvedu i důvody, proč jsem se pro ně rozhodl. Pro podrobnější informace čtenáře odkazuji na oficiální stránky či dokumentace.

5.1. Java SE 1.7

Za programovací jazyk jsem zvolil Javu SE v poslední verzi 1.7. Jedná se o objektově orientovaný programovací jazyk, který spravuje Oracle Corporation. Díky své přenositelnosti, poměrně jednoduché syntaxi (která je podobná jazykům vycházejícím z jazyka C) a velkému množství knihoven se stala v poslední dekádě jedním z nejpopulárnějších jazyků.

Důvod:
Pro Javu jsem se rozhodl, protože je:

- mým primárním jazykem (se kterým mám bohaté zkušenosti)
- převládajícím programovacím jazykem v rámci firmy Corpus Solutions a.s. (případné specifické problémy tedy mohu konzultovat)
- díky své rozšířenosti a popularitě mezi vývojáři dobrou volbou pro volně šiřitelný projekt

5.2. Eclipse

Samotný vývoj probíhal ve vývojovém prostředí Eclipse, verze Kepler. Jedná se o prostředí určené především pro vývoj v jazyce Java. Díky velkému množství modulů lze rozšířit o spoustu dodatečných funkcí.

Důvod:
Jedná se o kvalitní vývojové prostředí, které má dobře implementovanou integraci Apache Maven.

5.3. Apache Maven

Nástroj, jehož cílem je usnadnit správu, řízení a sestavování programů. Poskytuje mnoho výhod od vynucování dodržování osvědčených konvencí a postupů, přes rozsáhlou databázi knihoven (u kterých se nemusíme starat o aktualizaci či závislosti) po úzkou návaznost na Javu.

Důvod:
Rozsáhlá databáze knihoven a provázanost s Javou.

5.4. GitHub

Pro vedení projektu jsem zvolil jeden z nejrozšířenější serverů - GitHub. Jedná se o webovou službu, která poskytuje hosting pro Git repositáře. Git je systémem pro správu verzí, který vytvořil Linus Torvalds původně pro potřeby vývoje linuxového jádra.

Důvod:
Použití GitHubu poskytlo:

- přehled o aktuálním stavu programu pro všechny zúčastněné
- zálohu zdrojových kódu
- historizaci vývoje
- statistiky (množství přidaného/odebraného kódu, ...)
- transparentní vývoj od samého začátku projektu

5.5. Dropbox

Webové úložiště, které nabízí snadnou a rychlou cestu pro ukládání a sdílení souborů v síti Internet. Dropbox synchronizuje soubory mezi vybranou složkou v počítači a svým webovým úložištěm. Využil jsem jej k zálohování dokumentace projektu.

Důvod:
Dropbox považuji za kvalitní nástroj pro zálohování a synchronizaci dat. V základní verzi je zdarma a pro potřeby dokumentace bakalářské práce poskytuje dostatečný prostor.

6. Licence MIT

Svobodná licence, která má svůj původ na Massachusettském technologickém institutu.

Povinnost

- text licence musí být dodáván spolu s daným softwarem

Oprávnění

- komerční využití
- software lze distribuovat
- software lze modifikovat (a úpravy nemusí být poskytnuty veřejnosti)
- text licence je stručný a jasný

Zaručuje

- software je poskytován bez záruk a autoři softwaru (či držitelé licence) nenesou zodpovědnost za případné škody

Důvod:
Ke zvážení byly samozřejmě i další varianty. Uveďme si dvě populární licence a důvody, proč jsem je nezvolil namísto MIT.

Licence GPL přikazuje případné modifikace poskytnout veřejnosti. GPL by tedy mohla být překážkou pro ty, kteří chtějí do programu vkládat vlastní „know-how", ale nechtějí jej dále šířit.

Licence Apache vyžaduje větší změny hlásit původním autorům. Pro mé potřeby nadbytečná komplikace.

7. Načtení a zpracování dat

Cílem následujících tří kapitol je seznámit čtenáře s implementací jednotlivých stěžejních částí programu. Účelem není popisovat jednotlivé třídy, ale vysvětlit proč a jak byly dané problémy řešeny.

7.1. Doménové třídy

Prvním krokem bylo vytvoření doménových tříd, do kterých se budou ukládat datové bloky souboru výměnného formátu. Jeden datový blok je ukládán do právě jedné třídy. Bloků je dohromady 75, tříd je tedy stejný počet. Vzhledem k tomu, že ČÚZK každý blok přiřazuje do jedné z dvanácti skupin, zachoval jsem toto členění pro větší přehlednost i u doménových tříd. Dělení na skupiny je v programu reprezentováno balíčky.

Jméno skupiny	**Kód**
Nemovitosti	NEMO
Jednotky	JEDN
Bonitní díly parcel	BDPA
Vlastnictví	VLST
Jiné právní vztahy	JPVZ
Řízení	RIZE
Prvky katastrální mapy	PKMP
BPEJ	BPEJ
Geometrický plán	GMPL
Rezervovaná čísla	REZE
Definiční body	DEBO
Adresní místa	ADRM

Tabulka 1. Skupiny datových bloků

Atributy doménových tříd odpovídají uvozujícímu řádku datového bloku. Data se ukládají do objektů, ne primitivních datových typů. To z důvodu, abych mohl pracovat s hodnotou NULL. Třídy kromě atributů a přístupům k nim obsahují již jen překrytou metodu[17] `toString()`, která vrací textovou reprezentaci objektu.

Jmenná konvence byla pro snadnější orientaci převzata z dokumentace ČÚZK. Prošla pouze úpravou, aby odpovídala konvencím Javy. Například

[17]Potomek definuje vlastní implementaci metody, která byla poskytnuta rodičovskou třídou.

skupina „Prvky katastrální mapy“ se nachází v programu jako balíček `domain.prvkykatastralnimapy` a datový blok „SPOJENI_B_POLOH“ jako třída `SpojeniBPoloh.class`.

Problém, se kterým jsem se setkal při tvorbě doménových tříd, byly časté překlepy a přehlédnutí. Nutnost vytvořit během krátkého časového období velký počet doménových tříd (kdy některé třídy mají přes třicet atributů s často lehce zaměnitelnými názvy), způsobila, že jsem do programu zanesl poměrně velké množství drobných i větších chyb. U drobných chyb se nejčastěji jednalo o překlepy v názvech, u větších pak o chybějící atributy či špatné datové typy. Pokud bych takovémuto úkolu čelil znova, volil bych důkladnější průběžné kontroly ideálně v kombinaci s dalším člověkem.

7.2. Načtení souboru do doménových tříd

Postupu načtení souboru do doménových tříd říkáme parsování. Parser je pak množina algoritmů, která dokáže ze souboru získat informace a uložit je do patřičných atributů. Jedná se o kritickou část, která musí být po stránce rychlosti dobře optimalizovaná a nesmí obsahovat chyby. Parsování se skládá z lexikální a syntaktické analýzy.

7.2.1. Lexikální analýza

Jako první je spuštěn lexikální analyzátor. V našem případě jej představuje instance třídy `BufferedReader` ve spolupráci s metodami na zpracování CSV. Analyzátor načte jeden řádek, ten prochází a podle mu známých pravidel jej rozdělí na posloupnost tokenů. Jeden token reprezentuje právě jednu hodnotu oddělenou středníkem.

Z datové řádku:

```
&DZPOCHN;8;"národní park - III.zóna";"01.01.1993
00:00:00";"";"a";"a";"a";2
```

Získáme po lexikální analýze tyto tokeny:

```
&DZPOCHN
8
"národní park - III.zóna"
"01.01.1993 00:00:00"
""
"a"
"a"
"a"
2
```

Vidíme, že zpracování CSV není obtížné. Hodnoty od sebe odděluje středník

a zpracování je tedy přímočaré. Nicméně aby analyzátor fungoval korektně ve všech situacích, musel jsem ošetřit několik speciálních případů.

7.2.2. Speciální případy

V kapitole 4.2.2. jsme si řekli, že textové i datumové hodnoty jsou v datovém řádku obaleny uvozovkami. Ty určují začátek/konec hodnoty a při analýze tak pomáhají určit výslednou podobu daného tokenu. V případě, že se uvozovky vyskytují i uvnitř textu (což není nic neobvyklého), jsou zdvojeny. S tímto musí analyzátor počítat. Když narazí na uvozovku, podívá se na další znak a pokud se jedná o uvozovku a nachází se v textové hodnotě, rozhodne o jejím přidání do aktuálního tokenu. Nepřidává ji samozřejmě ve zdvojené podobě: první slouží jako tzv. „escape character“ a je vypuštěna, druhou pak přidá.

Podívejme se na upravený datový řádek z předchozí ukázky:

```
&DZPOCHN;8;"národní park - ""Šumava"" - III.zóna";"01.01.1993
00:00:00";"";"a";"a";"a";2
```

Vidíme, že nám přibyl v uvozovkách název národního parku. Název je stále součástí stejného tokenu. Pokud by pravidlo nebylo implementováno, lexikální analyzátor nám vrátí tyto tokeny:

```
&DZPOCHN
8
"národní park - Šumava - III.zóna"
"01.01.1993 00:00:00"
""
"a"
"a"
"a"
2
```

V exportovaných datech chybí uvozovky kolem názvu národního parku. Mají totiž pouze jeden význam: určovat ohraničení textu a datumu. Uvnitř textu jsou tedy vypuštěny (navzájem se vyruší).

Mnohem závažnější chybou by bylo, pokud bychom nerozlišovali, v jakém kontextu se nachází středník, který funguje jako oddělovač:

```
&DZPOCHN;8;"národní park; III.zóna";"01.01.1993
00:00:00";"";"a";"a";"a";2
```

```
&DZPOCHN
8
"národní park"
" III. zóna"
"01.01.1993 00:00:00"
""
```

```
"a"
"a"
"a"
2
```

Analyzátor po textu „národní park“ považuje token za ukončený. Středník pro něj není součástí textu, slouží pouze jako oddělovač. Vidíme, že v případě oddělovače je velmi důležité mít správný kontext předchozího zpracování.

V rámci textového řetězce se samozřejmě může vyskytovat i znak nového řádku. Ve chvíli kdy analyzátor načte na vstup řádek, musí se podívat, zda aktuální řádek nenavazuje na předchozí. To je vyřešeno nahlédnutím do vyrovnávací paměti, která se uvolní až ve chvíli, kdy je řádek zapsán do doménové třídy. A řádek je zapsán pouze tehdy, kdy je po jeho zpracování nastaven příznak o uzavření uvozovek. Pokud tedy vyrovnávací paměť není prázdná, víme, že aktuální řádek patří k předchozímu a po jeho zpracování tokeny připojíme k předchozím.

Tímto způsobem jsou zpracovány všechny řádky v souboru, avšak při následné syntaktické analýze:

- z hlavičky využíváme pouze informaci, o jaký typ dat se jedná (stavová/změnová)
- druh kódování je načten a zpracován již při inicializaci parseru, a je tedy při samotném parsování ignorován
- uvozovací řádky jsou ignorovány všechny, protože pro nás nemají žádnou informační hodnotu[18]
- koncový řádek je taktéž ignorován, protože si konec souboru hlídá vnitřní logika `BufferedReaderu`

Jestliže máme soubor, v němž je 99,9998% řádků datových (po odečtení několika řádků hlavičky, uvozujících řádků jednotlivých datových bloků a jednoho koncového řádku), nevyplatí se řešit jejich filtrování na úrovni lexikální analýzy.[19]. Zahazujeme je tedy až při syntaktické analýze.

7.2.3. Syntaktická analýza

Ve chvíli kdy je ukončena lexikální analýza, započne syntaktická analýzu. Ta dostává na vstup množinu tokenů (v Javě předávány jako pole řetězců). První

[18]Definice doménových tříd nelze v Javě rozumně vytvářet až za běhu, kde bychom mohli využít informace o typech a velikostech atributů jednotlivých datových bloků z uvozovacích řádků.

[19]Pokud bychom kupříkladu přidali podmínku, že řádek začínající &B (uvozující řádek) ignorujeme a dále nezpracováváme, musely by touto „zbytečnou“ podmínkou projít i ostatní řádky, kterých se to z drtivé většiny netýká.

token obsahuje návěstí a zkratku dané hlavičky, nebo datového bloku, nebo se jedná o návěstí konce řádku a neobsahuje nic dalšího.

Vyhledávání probíhá nad hašovací tabulkou, která je vytvořena při inicializaci parseru, a jako klíč je použit právě onen první token. Pro snadnější údržbu a lepší přehlednost ji byla dána přednost před konstrukcí `switch`[20]. Jelikož Java 1.7 neumožňuje vytvořit hašovací tabulku, která by jako hodnotu brala metodu, byl použit návrhový vzor **Příkaz**[21] s rozhraním jakožto objektem.

Při vyhledávání syntaktický analyzátor zahodí vše kromě datových řádků a řádku hlavičky, jenž udává typ dat. Pro jednoduchost však opomíjejme i jej, a pracujme pouze s datovými řádky, které obsahují data pro naplnění doménových tříd.

Jako hodnota klíče je vrácena statická třída určená pro „naplnění“ dané doménové třídy. Pro každou doménovou třídu tedy máme jednu statickou třídu. Statické třídě jsou předány tokeny (které jsou stále uloženy v poli řetězců). Ta pole projde, tokeny převede na správné datové typy a uloží do nové instance doménové třídy. Instance obsahující data parsovaného řádku je pak uložena do kolekce. Tímto krokem končí zpracování jednoho řádku.

Parsování souboru probíhá, dokud je počet zpracovaných řádků menší než maximální velikost jedné dávky, nebo není dosaženo jeho konce.

7.3. Dávkové zpracování

Implementoval jsem jej z důvodu, že část souborů výměnného formátu nelze zpracovat najednou (omezení na straně operační paměti). V základním nastavení je tedy načteno, zpracováno a uloženo do kolekce maximálně 100 000 řádků. Poté je parsování pozastaveno a dávka předána do vyrovnávací paměti, kde si ji přebírají třídy zodpovědné za exportování dat. Parser si samozřejmě musí uchovávat aktuální pozici v souboru.

Tvrzení, že najednou zpracováváme maximálně 100 000 řádků, není úplně přesné. Jelikož bylo pro navýšení rychlosti do programu zabudováno vícevláknové zpracování, jsou jednotlivé dávky ukládány až do desetistupňové vyrovnávací paměti. Můžeme tedy pracovat až s miliónem řádků. Implementace vícevláknového zpracování je popsána v následující kapitole.

V předchozím textu jsme si popsali postup načtení dat do doménových tříd. Než se přesuneme dále, rád bych na závěr uvedl několik drobností:

BigInteger

Některé hodnoty uložené v datovém souboru byly příliš velké na to, aby je bylo možné uchovávat v primitivních datových typech[22]. V těchto případech jsem po-

[20] Konstrukce switch byla používána ve verzi 1.0. Potvrdilo se však, že u většího počtu hodnot se stává nepřehlednou.

[21] Objekt je použit k reprezentaci a zapouzdření všech informací nutných k pozdějšímu zavolání metody.

[22] Atributy těchto typů nesou elementární, z hlediska Javy atomické, hodnoty.

užil datový typ `BigInteger`. Jedná o neměnný objekt[23] s libovolnou přesností. Mohl jsem díky tomu přesně určit, jak velké číslo mají jednotlivé atributy pojmout.

Datum

Pro ukládání datumu byla použita i přes obecně známé nedostatky standardní knihovna `java.util.Date`. Pro uchovávání datumu je rychlostně dobře optimalizovaná a pro naše potřeby dostačující.

> „`Java.util.Date` je důkazem, že i skvělí programátoři umí věci zpackat. A `java.util.Calendar`, jež měla dát podle Sunu všechno do pořádku, potvrzuje, že průměrní programátoři to umí zbabrat taky.“
>
> — gustafc, StackOverflow.com

Velikost dávky

Jistý čas jsem věnoval hledání optimální velikosti jedné dávky. Příliš malé dávky rychle zaplní desetiúrovňovou vyrovnávací paměť a parser bude muset čekat, než mu třídy běžící v druhém, exportovacím, vlákně uvolní místo. Problém s malými dávkami by mohly mít i exportní třídy, které by často musely otevírat a zavírat soubor (plus režie, která probíhá, než začne ukládání). Naopak velké dávky by mohly zaplnit operační paměť. Výchozí velikost 100 000 řádků byla zvolena po zralé úvaze podložené několika testy.

Jmenná konvence doménové třídy a příslušné plnící třídy

Pro přehlednost byla zavedena jmenná konvence, kterou si předveďme na zpracování datového bloku „PARCELY“:

- doménová třída: `Parcely.class`
- plnící (statická) třída: `ParcelyParser.class`

[23] Žádná metoda nemůže změnit jeho jednou nastavenou hodnotu.

8. Vícevláknové zpracování

Verze programu 1.0 pracovala pouze s jedním vláknem. Dávka byla načtena a následně exportována. Výhodou byla jednoduchost. Nevýhody však převažovaly. Procesor nebyl rovnoměrně vytížen, protože se pracovalo jen s jedním jádrem a vzhledem k tomu, že se program neustále přepínal mezi parsováním a exportováním, byly velmi patrné propady využití i v rámci jednoho jádra. Východiskem z této situace byla implementace „producenta a konzumenta".

8.1. Technika producenta a konzumenta

Technika je založena na komunikaci dvou vláken, kdy producent periodicky získává data a ukládá je do sdílené paměti a konzument je postupně odebírá.

V našem případě je producentem parser a sdílenou pamětí instance třídy `ArrayBlockingQueue`[24] s velikostí pole deset, jež implementuje rozhraní `BlockingQueue`[25]. Vždy po zpracování 100 000 řádků zkusíme uložit dávku do paměti. Pokud je plná musíme počkat a parsování je po dobu čekání pozastaveno. Můžeme uložit až deset dávek (viz velikost pole), tedy jeden milión řádků.

Konzument, který je dle vybraného exportu reprezentován exportními třídami pro SQL*Loader, nebo Java Database Connectivity, pak data postupně odebírá a zpracovává.

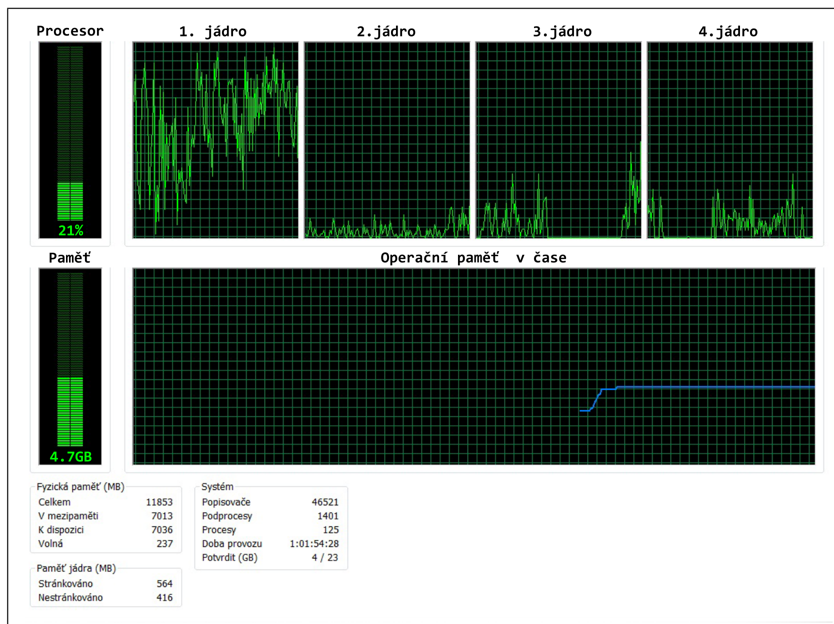
Podmínkou je, aby producent nezapisoval do plné a konzument nečetl z prázdné paměti. To je realizováno čekáním, které zajišťuje logika třídy `ArrayBlockingQueue`. Pokud se vlákno konzumenta pokusí načíst data z prázdné fronty, je zablokováno do té doby, než producent do fronty vloží data. Producent je zase zablokován tehdy, když je fronta plná a to tak dlouho, dokud konzument nezpracuje část (nebo všechna data) a frontu neuvolní.

Rozšíření o vícevláknové zpracování bylo přidáno do verze 2.0 a znamená časovou úsporu při exportu přes SQL*Loader (kdy se data ukládají do souborů) o 40-45%. U exportu přes databázi k výraznému zrychlení nedošlo, protože „úzkým hrdlem" je odesílání dat do databáze.

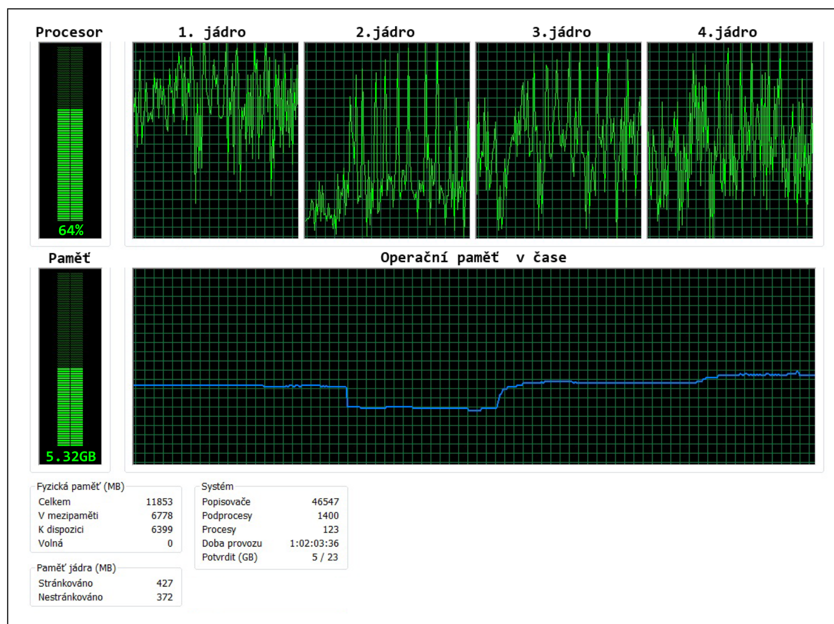
Velikost vyrovnávací paměti byla určena s přihlédnutím na potřeby SQL*Loaderu. Implementace producenta-konzumenta by ztrácela smysl, pokud by byla vyrovnávací paměť stále plná a producent by musel čekat.

[24]Blokující fronta založená na poli fixní velikosti (velikost se určuje při inicializaci). Funguje na principu FIFO (první dovnitř, první ven). Na začátku fronty je prvek, který je v ní nejdelší dobu.

[25]Rozhraní `BlockingQueue` je součástí balíku `java.util.concurrent`. Poskytuje operace, které při pokusu načíst prvek z prázdné fronty pozastaví vlákno a čekají, dokud ve frontě není alespoň jeden prvek. A naopak při pokusu uložit prvek do plné fronty čekají, dokud se neuvolní místo. Operace nad frontou jsou vláknově bezpečné: jsou atomické využívajíce interních zámků a dalších synchronizačních mechanismů.



Obrázek 1. Jednovláknové zpracování souboru



Obrázek 2. Vícevláknové zpracování souboru

Jak vidíme na obrázcích, program je na procesoru mnohem lépe škálován. Průměrné využití procesoru vzrostlo z 20% na 60% a zátěž se rozložila na všechna jádra. Můžeme vidět i průměrný nárůst u operační paměti o 400 MB, záleží na zaplnění vyrovnávací paměti. To je vzhledem k navýšení rychlosti přijatelná ztráta[26].

[26] V případě potřeby můžeme v konfiguračním souboru upravit velikost dávky a tím i množství využité operační paměti.

9. Exportování dat

Cílovým systémem pro ukládání dat z doménových tříd je Oracle Database. Data jsou do databáze ukládány přes SQL*Loader ve verzi 1.0, nebo Java Database Connectivity (dále JDBC) ve finální verzi 2.0. Typ exportu zadává uživatel jako parametr při spuštění. Právě na základě hodnoty tohoto parametru jsou za běhu vybrány třídy odpovědné za export. Řešení je přitom natolik obecné, že lze snadno rozšířit o další způsoby exportu.

9.1. Abstraktní továrna

Abych datům z doménových tříd mohl na základě uživatelova výběru dynamicky přiřadit cestu, kterou budou během exportu putovat, využil jsem návrhového vzoru Abstraktní továrny. Ten poskytuje postup jak zapouzdřit skupinu továren s podobným úkolem, aniž bychom specifikovali konkrétní továrnu. Ta je určena až za běhu programu.

Nenechme se zmást spojením „konkrétní továrna“. V našem případě jednoduše představuje třídu, která vytváří instance tříd, jejž mají implementovanou logiku jak exportovat data. Ukažme si, jak je tento návrhový vzor implementován přímo v programu.

Na začátku máme abstraktní továrnu (rozhraní) `ExporterFactory`. Jelikož je abstraktní, neříká nic o tom, kde se data exportují. Je implementována továrnami (třídami) `OracleLoaderFileExporterFactory` a `OracleDatabaseJdbcExporterFactory`. Před slovo `ExporterFactory` přibyly prefixy `OracleLoader` a `OracleDatabase`. Tedy již mluvíme o konkrétních továrnách, protože jak jejich název vypovídá, `OracleLoaderFileExportertFactory` exportuje data přes SQL*Loader a `OracleDatabaseJdbcExporterFactory` exportuje data přes JDBC. Obě konkrétní továrny obsahují implementaci metod z abstraktní továrny. Ty jsou velmi jednoduché. Každé doménové třídě je definována právě jedna metoda, která pro ni vrací exportér do daného systému. Uveďme si příklad pro `Parcely`:

```
public Exporter getParcelyExporter(List<Parcely> parcely)
```

V závorkách vidíme, že metoda dostává na vstup kolekci parcel a vrací pro ně `Exporter` (což je opět rozhraní; z jeho názvu můžeme vyčíst, že sdružuje všechny exportovací třídy). A to je vše, vrácená instance `Exporter` obsahuje logiku, která popisuje, jak data exportovat přes SQL*Loader, nebo JDBC (v závislosti na tom, jaký typ exportu uživatel zadal jako parametr).

Ukažme si, jak snadné je rozšířit export dat o další způsob. Kupříkladu by byl požadavek, abychom data přes vysílač rozesílali do okolí (ač se jedná o požadavek nesmyslný, uvidíme, že i ten dokážeme snadno splnit). Každý, kdo by žil v okolí a vysílání přijímal, by pak data mohl nějakým způsobem zpracovat.

Pro splnění tohoto požadavku musíme udělat jediné, vytvořit továrnu

`BroadcasterExporterFactory`, která se stává dalším potomkem abstraktní továrny `ExporterFactory` a implementuje její metody. Ty pak vrací `Exporter`, který umí odesílat data přes vysílač.

Návrhový vzor Abstraktní továrny je velmi dobrým způsobem pro budování dynamických systémů. Ač se nejedná o obzvlášť obtížnou techniku, tak rozhodně není triviální a je potřeba se nad implementací zamyslet. Jsem rád, že jsem si mohl tuto metodu vyzkoušet na reálném problému, protože se jedná o velice praktický přístup, který se mi bude v budoucnu jistě hodit.

V této chvíli již program umí vybrat cestu pro data z doménových tříd. V poslední části probereme, jak je implementovaná samotná logika exportovacích tříd pro SQL*Loader a JDBC, kterou jednotlivé konkrétní továrny v podobě instance `Exporter` vrací.

9.2. Oracle Database

Jedná se o objektově relační databází, kterou stejně jako Javu vyvíjí Oracle Corporation. Podporuje relační dotazovací jazyk SQL, imperativní programovací jazyk PL/SQL a spoustu dalšího. Velmi důležitá je pro práci nad daty katastru nemovitostí integrovaná množina funkcí Oracle Spatial. Ta umožňuje uložení, přístup a práci nad prostorovými daty. O typu databáze rozhodla firma Corpus Solutions a.s. s ohledem na databázové specialisty, které zaměstnává, a kteří budou s importovanými daty dále pracovat.

9.3. Export přes SQL*Loader

SQL*Loader je primárním nástrojem pro rychlé uložení dat z externích souborů do tabulek v Oracle Database. Jedná se o jednu ze stěžejních funkcí tohoto databázového systému a je dostupná ve všech konfiguracích.

Stavová data nelze kvůli jejich velikosti v rozumném čase uložit do databáze přes JDBC. Proto je využit SQL*Loader, který je k importům o takovýchto objemech určen.

9.3.1. Kontrolní soubor SQL*Loaderu

Jedná se o textový soubor psaný speciální syntaxí, která SQL*Loaderu určuje, kde najít data k importu, jak je zpracovat, interpretovat a kam je uložit. Poskytuje širokou škálu nastavení.

Každá doménová třída má jeden kontrolní soubor, protože odpovídá jedné tabulce v databázi. Na základě požadavků firmy jsem definoval jednotný styl, který využívají všechny doménové třídy.

Například soubor pro doménovou třídu `Adresa` vypadá takto:

```
LOAD DATA
CHARACTERSET EE8MSWIN1250
```

```
INFILE "ADROBJ.TXT" "STR'|NAK'"
APPEND
INTO TABLE ADROBJ
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(
KOD NULLIF ( KOD = "NULL" ),
OBJEKT_KOD NULLIF ( OBJEKT_KOD = "NULL" ),
ULICE_KOD NULLIF ( ULICE_KOD = "NULL" ),
CIS_ORIENT CHAR(4) NULLIF ( CIS_ORIENT = "NULL" ),
PSC NULLIF ( PSC = "NULL" ),
PLATNOST_OD DATE "DD.MM.YYYY HH24:MI:SS" NULLIF ( PLATNOST_OD =
"NULL" ),
PLATNOST_DO DATE "DD.MM.YYYY HH24:MI:SS" NULLIF ( PLATNOST_DO =
"NULL" ),
ULICE_NAZEV CHAR(48) NULLIF ( ULICE_NAZEV = "NULL" )
)
```

Zajímají nás především tyto řádky:

- `CHARACTERSET`: kódování souboru
- `INFILE`: název souboru a koncová sekvence jednotlivých záznamů[27]
- `INTO TABLE`: tabulka, která bude těmito daty naplněna
- `FIELDS TERMINATED BY`: oddělovač jednotlivých hodnot
- `OPTIONALLY ENCLOSED BY`: znak, kterým jsou hodnoty obaleny

Dále již následují definice jednotlivých sloupců. Ty nám určují do jakých sloupců se mají data ukládat. Podíváme se na sloupec uchovávající číslo, text a data.

- `KOD NULLIF ( KOD = "NULL" )`: sloupec uchovávající **číslo** se skládá z názvu a výchozí hodnoty, pokud je vkládaná položka NULL
- `CIS_ORIENT CHAR(4) NULLIF ( CIS_ORIENT = "NULL" )`: sloupec uchovávající **text** se skládá z názvu, maximální délky, které může řetězec nabýt a výchozí hodnoty, pokud je vkládaná položka NULL
- `PLATNOST_OD DATE "DD.MM.YYYY HH24:MI:SS" NULLIF ( PLATNOST_OD = "NULL" )`: sloupec uchovávající **datum** se skládá z názvu, formátu datumu a výchozí hodnoty, pokud je vkládaná položka NULL

[27] Bylo nutné vybrat takovou sekvenci (posloupnost znaků), která bude vyjadřovat ukončení řádku, ale zároveň se v souborech výměnného formátu nevyskytuje. Zvolil jsem svislou čáru, ta se však datech může vyskytovat. Doplnil jsem tuto sekvenci netisknutelný znakem `negative-acknowledge`, který nejde zadat z klávesnice. Problém s koncovou sekvencí byl tímto elegantně vyřešen.

Pokud bychom chtěli, můžeme v tomto souboru zapsat i vytvoření jednotlivých sloupců. To však nebylo mým cílem, protože firma Corpus Solutions a.s. již měla připravenu databázi, do které se budou data ukládat. Při ukládání stavových dat přes SQL*Loader jsou použity prázdné tabulky, které nemají definovány ani primární klíče. Ty jsou společně s dalšími prvky doplněny až poté, co proběhne import.

Vytvoření databáze, která dokáže korektně pracovat se všemi daty, které výměnný formát přenáší, je složitou záležitostí. To především s ohledem na pochopení souvislostí a návazností mezi jednotlivými tabulkami. Troufám si říci, že by to samo o sobě vydalo na velkou část další bakalářské práce. Databázi měl na starost Ing. Zícha a my se tedy tímto aspektem dále nebudeme zabývat.

9.3.2. Vstupní data a datové soubory

SQL*Loader načítá data ze souboru, který je uveden v kontrolním souboru. Data jsou v souborech organizovány do záznamů.

Záznamy v datovém souboru doménové třídy `Adresa` jsou strukturovány takto:

```
"780481","778371","NULL","NULL","38301","03.12.2013
00:00:00","NULL","NULL"|NAK
"780499","778389","NULL","NULL","38301","03.12.2013
00:00:00","NULL","NULL"|NAK
"780502","778397","NULL","NULL","38301","03.12.2013
00:00:00","NULL","NULL"|NAK
```

Jednotlivé záznamy odpovídají předpisu kontrolního souboru. Jsou v uvozovkách, odděleny čárkou, ...

Musel jsem tedy vytvořit takové exportéry, které dokážou data z doménových tříd převést do podoby, která bude odpovídat tomu, co definují kontrolní soubory.

9.3.3. Průběh exportu přes SQL*Loader

Nyní si projdeme jednotlivé kroky, které nastanou, když uživatel vybere export přes SQL*Loader, program zpracuje data (buď všechna, nebo jednu dávku) a předá je na export. Jak víme, na exportu je dynamicky rozhodnuto, jaké třídy za něj budou zodpovídat.

Tvorba kontrolních souborů

Při exportu přes SQL*Loader jsou jako první vytvořeny kontrolní soubory. Každá doménová třída má právě jednu exportovací třídu. Jednotlivé kontrolní soubory se liší ve jméně tabulky a sloupcích, do kterých se budou ukládat atributy dané třídy. Rozhodl jsem si pro usnadnění vytvořit pomocné metody, které urychlí tvorbu jednotlivých kontrolních souborů. Metody poskytuje třída `OracleLoaderFileExporter`, která vytvoří vzorový soubor, do kterého si

jednotlivé exportéry doplní informace o názvu tabulky a jednotlivých sloupcích.

Při exportu pak exportér využije logiky, kterou mu poskytuje rodičovská třída `OracleLoaderFileExporter` a vytvoří kontrolní soubor. Ten je předán třídě `FileManager`, která se postará o uložení souboru na místo, které uživatel specifikoval při spuštění programu.

Pokud exportujeme změnová, a ne stavová data, je přidán k názvu souboru prefix „TMP_“.

Data doménových tříd

Dále jsou exportována samotná data, která uchovávají doménové třídy. Zde je zpracování poměrně jednoduché. Celá kolekce instancí[28] dané doménové třídy se společně s kódování předává `FileManageru`, který kolekci prochází a nad každou instancí volá metodu `toString()`, která vrací textový řetězec, který reprezentuje obsah atributů dané instance.

Původně jsem zamýšlel pro export dat využít knihovnu `org.apache.commons.io.FileUtils`, ta má však jeden zásadní nedostatek. Vždy po přidání řádku do souboru automaticky zapíše znak nového řádku. Znak je přidán i po zpracování poslední instance. To při importu přes SQL*Loader způsobuje u zpracování posledního řádku datového souboru chybu, jelikož tento řádek je prázdný. Napsal jsem si tedy vlastní třídu, která využívá `FileUtils` pouze k otevření výstupního proudu, ale samotné ukládání zastává sama.

Problém posledního řádku jsem vyřešil způsobem, že se u zpracování poslední instance kolekce znak nového řádku vypouští. Pokud máme v další dávce nová data, která se připojují do stejného souboru, tak se řádek přidá až v této chvíli. Výjimkou je samozřejmě prvotní naplnění souboru, kdy se nový řádek nevkládá.

9.3.4. Návrhový vzor Template method

Při tvorbě exportovacích tříd jsem již během krátké chvíle zjistil, že vzniká velké množství redundantního kódu. Každá exportovací třída si sama zajišťovala vytvoření celého kontrolního souboru (včetně částí, které jsou společné pro všechny třídy) i ukládání přes třídu `FileManager`. Když se například na průběžných testech projevil problém s posledním řádkem u knihovny `FileUtils`, kterou jsem se rozhodl nahradit vlastním řešením, bylo nutné její volání nahradit ve všech exportovacích třídách. Jakékoliv, byť drobné úpravy, tedy znamenaly spoustu času, velkou náchylnost k zanesení chyby a značily problém v návrhu.

K vyřešení problému jsem se rozhodl po úvaze využít návrhového vzoru `Template method`.

[28] Ještě jednou připomínám, že jeden datový řádek souboru výměnného formátu je roven jedné instanci. Kolekce instancí jednoho typu doménové třídy je tedy rovna jednomu a více řádkům daného datového typu.

Template method

Návrhový vzor je založen na třídě předka, kde definujeme kostru algoritmu, která obsahuje jednu (i více) abstraktních metod. Jeho potomci pak přepíší tyto metody, doplní tak kostru a vznikne plnohodnotný algoritmus.

V našem případě se jednotlivé exportovací třídy liší pouze v názvu tabulky a sloupcích, které je potřeba zadat při tvorbě kontrolního souboru.

Třída `OracleLoaderFileExporter` představuje předka exportovacích tříd, který definuje kostru algoritmu. Velká část kontrolního souboru je pro všechny třídy stejná (definice konce řádku, kódování, ...), takže tato část je vytvořena již v samotné kostře. V názvu a sloupcích se třídy liší. V `OracleLoaderFileExporter` je tedy tato část právě onou abstraktní metodou, kterou přepisují potomci třídy a doplňují tak algoritmus o svou specifickou část.

Aplikováním návrhové vzoru došlo k velké úspoře zdrojového kódu a údržba programu se stala mnohem snadnější a odolnější proti zanesení chyb. Pokud nyní například dojde ke změně koncového řádku, není již třeba tuto změnu přepisovat ve všech exportovacích třídách (kterých je 75), ale pouze v jejich společném předkovi.

Pokud šlo o využití návrhového vzoru Abstraktní továrny, tak ten jsem měl promyšlen a počítal jsem s ním již od začátku. Avšak pro Template method jsem se rozhodl až ve chvíli, kdy jsem narazil na uvedený problém. Díky tomu, že jsem jej poté využil i u exportu přes JDBC, kde jeho implementace znamenala použití poměrně pokročilé generiky, považuji jej, pokud jde o můj osobní rozvoj, jako jeden z největších přínosů bakalářské práce.

9.3.5. Import vyexportovaných dat

Po zpracování celého souboru výměnného formátu nalezneme na výstupní cestě, kterou jsme zadali při spuštění, soubory pro SQL*Loader. Kontrolní soubory mají koncovku „.CFG“ a datové soubory koncovku „.TXT“. Například pro `Parcely` máme dvojici:

- PARCELY.CFG (kontrolní soubor)
- PARCELY.TXT (datový soubor)

V této chvíli je práce programu u konce[29] a standardně se ukončí.

Nad takto vyexportovanými soubory pak zavoláme script, který spustí SQL*Loader a importuje data do databáze. Zjednodušeně vypadá takto:

```
sqlldr JMENO/HESLO@DB control=PARCELY.CFG
```

[29] Při vypínání jsou zároveň uloženy poslední záznamy žurnálovací knihovny `log4j` do logu „KnParser.log“, který poskytuje detailní informace o běhu programu.

9.4. Export přes Java Database Connection

JDBC představuje vrstvu mezi programem napsaným v Javě a databází. Odstiňuje tak programátora od specifického rozhraní databáze a poskytuje jednotný přístup k relačním databázím. Na základě toho s jakou databází chceme komunikovat (Oracle Database, PostgreSQL, ...), vybereme JDBC ovladač, který zajišťuje přístup ke konkrétnímu databázovému serveru.

JDBC poskytuje metody pro navázání spojení s databází, vytváření a vykonávání SQL dotazů (`INSERT`, `SELECT`, ...) či procházení vrácených záznamů. Podporuje také transakční zpracování, ochranu před SQL injection a vykonávání SQL dotazů v dávkách.

Největší výhodou JDBC je, že prostou výměnou jednoho JDBC ovladače za druhý můžeme v rámci programu pracovat nad různými databázemi. To vše bez jakýchkoliv dalších úprav v kódu.

Export dat přes JDBC byl poslední a pravděpodobně nejtěžší částí bakalářské práce.

9.4.1. Změnová data a JDBC

V kapitole o struktuře výměnného formátu jsem popsal hlavní rozdíl mezi stavovými a změnovými daty. Stavová data jsou velmi objemná a obsahují všechny informace potřebné k rekonstrukci katastrálního území k datu, ke kterému byla vyexportována. Naproti tomu změnová data tak objemná nejsou, ale zachycují pouze změny, ke kterým došlo ve vybraném časovému období.

Záměr byl následující:

- třídy pro export přes SQL*Loader zpracovávají stavová data, generují z nich kontrolní a datové soubory, které jsou následně importovány do databáze (to bylo splněno ve verzi 1.0, potažmo 1.1). Implementaci popisuje předchozí podkapitola.
- třídy pro export přes JDBC zpracovávají změnová data a aktualizují data v databázi (to bylo cílem verze 2.0)

Hlavním rozdílem je, že stavová data se exportují přes SQL*Loader do prázdné databáze. Jedná se tedy o jednoduchý import. Změnová data, která se mají zpracovávat přes JDBC, však potřebují při exportu logiku. Při aktualizaci záznamů změnovými daty je potřeba například rozhodnout, která informace o určité parcele, které je uložena v databázi a zároveň je popisována i ve zpracovávaném souboru výměnného formátu, je aktuálnější.

Zde chci upozornit na jednu věc: stavová i změnová data jsou po stránce struktury **totožná**. Liší se pouze v rozsahu, který popisují. Pokud bychom chtěli, můžeme do databáze přes JDBC importovat i kompletní stavová data (a s využitím stejné logiky). To je však kvůli jejich objemu nerealizovatelné. Proto je zpracování přes JDBC primárně určeno pro ta změnová.

9.4.2. Analýza

Při vývoji exportéru přes SQL*Loader nebylo zapotřebí větších konzultací. U exportu změnových dat přes JDBC však byla spolupráce naprosto nezbytná.

Rád bych zde ještě jednou poděkoval Ing. Zíchovi, který mi byl v této fázi velmi nápomocen. Společně jsme vytvořili čtyři typy algoritmů, které zajišťují aktualizaci všech dvanácti skupin datových bloků.

9.4.3. Algoritmy pro zpracování změnových dat

Jak již víme, každý datový blok je uložen právě v jedné doménové třídě. Avšak datové bloky se nezpracovávají stejně, pokud nad nimi chceme v databázi provádět aktualizaci. Doménová třída má pro své zpracování přes JDBC vybrán právě jeden ze čtyř typů algoritmů.

- **Historizační doménové třídy**: třídy, kde datové bloky, které se do nich ukládají, podléhají principu historizace. Udržujeme tedy minulost a přítomnost.
- **Stavové doménové třídy typu 2**: třídy, kde datové bloky, které se do nich ukládají, podléhají principu historizace. Udržujeme tedy minulost a přítomnost. Oproti historizačním tabulkám je však až za běhu rozhodnuto, který atribut bude použit jako primární klíč.
- **Stavové a číselníkové doménové třídy**: třídy, kde datové bloky, které se do nich ukládají, nepodléhají principu historizace. Udržujeme tedy pouze aktuální stav.
- **Hranice parcel**: tímto algoritmem je exportována pouze jediná doménové třída a to třída `HraniceParcel`. Tato doménová třída podléhá principu historizace, avšak navíc bereme v potaz hodnoty pro dvě parcely, které tato hranice odděluje.

9.4.4. Průběh exportu přes JDBC

Jelikož jsem byl dostatečně poučen z tvorby velkého množství exportovacích tříd pro SQL*Loader, snažil jsem se nejprve najít vzor mezi jednotlivými třídami. Ten pak abstrahovat do abstraktní třídy, která bude na jednom místě poskytovat logiku.

Rozhodl jsem se pro už jednou úspěšně použitý návrhový vzor Template method. Složitější to však bylo kvůli tomu, že existuje několik způsobů, jak se doménová třída může zpracovat. Postup byl následující:

- určit doménové třídy, které mají společné rysy a budou se zpracovávat jedním typem algoritmu

- vytvořit ve spolupráci se Ing. Zíchou tento algoritmus
- vytvořit jednotlivé exportéry vybraných doménových tříd a ty označit jako potomky abstraktní třídy, která obsahuje daný algoritmus
- v exportérech pak přepsat metody, které tato abstraktní třída nedefinuje (jako v případě SQL*Loaderu jde o zkompletování algoritmu)

Ač se jednotlivé abstraktní třídy liší v algoritmu, mají i několik společných rysů. Ty jsem se rozhodl abstrahovat do hlavního předka, která bude zastřešovat celý import do JDBC. Jedná se tedy o třístupňovou hierarchii.

9.4.5. Rozhraní a společná logika pro JDBC (1. úroveň)

Jako první jsem vytvořil rozhraní `JdbcOperations`, které poskytuje společné metody pro práci nad JDBC. Navázání spojení na databázi, uzavření spojení, získání primárních klíčů, `INSERT/DELETE`, ... bude potřebovat každý exportovací algoritmus.

`OracleDatabaseJdbcExporter` je třídou, která implementuje drtivou většinu společných metod, které předepisuje rozhraní. Patří v oné třístupňové hierarchii na nejvyšší příčku a je rodičem abstraktních tříd, které obsahují jednotlivé algoritmy.

Třída obstarává napojení na databázi, načtení jmen primárních klíčů pro danou doménovou třídu a exekuci všech SQL dotazů nad databází (s výjimkou exportéru pro `HraniceParcel`).

Zajímavostí jsou zmíněné primární klíče. Pro ty je ve firemní databázi vytvořena tabulka, ve které je pro každou tabulku, do které se ukládá doménová třída, uvedeno, jaké má primární klíče. Například pro tabulku `PARCELY`, do které se ukládá doménová třída `Parcely`, je to primární klíč `ID`. Ve chvíli kdy zpracováváme instanci doménové třídy `Parcely`, tedy musíme načíst hodnotu v atributu `Id`.

Je tedy potřeba za běhu dynamicky vybrat, jaká metoda bude zavolána. Přišel jsem s následujícím řešením:

- načtu název primárního klíče (nebo klíčů) tabulky, do které se ukládá daná doménová třída
- nad vráceným záznamem zavolám metodu, která jej převede do velbloudí notace[30] a přidá prefix „get“
- příklad: tabulka `ADRESY`, do které se ukládá doménová třída `Adresy`, má primární klíč `ADRESY_ID`. Takto vrácený název tedy nejdříve převedu do velbloudí notace `AdresyId`, a poté k němu přidám prefix `get`. Vznikne tak název metody `getAdresyId`.

[30]Způsob zápisu, kdy se několikaslovný název píše dohromady bez mezer, přičemž každé slovo začíná velkým písmenem. Tímto způsobem jsou tvořena jména metod v Javě.

- ve chvíli kdy v exportéru zpracovávám instanci doménové třídy, mohu dynamicky vyvolat metodu, která mi vrátí hodnotu či hodnoty atributů, které představují primární klíč. Využívám tedy principu reflexe[31].

Velkou výhodu, které toto řešení přináší, je, že pokud dojde v databázi ke změně primárního klíče, tak není zapotřebí žádných úprav v programu.

Chci zde ještě upozornit na jeden rys, který šetří prostředky a má pozitivní vliv na rychlost. Pokud například ukládáme do databáze 50 tisíc instancí doménové třídy `Parcely` (tedy 50 tisíc zpracovaných řádků datového souboru), tak jména primárních klíčů načítáme a metody z nich vytváříme pouze u první instance.

[31]Součástí Javy je Java Reflection API. Díky němu můžeme za běhu získávat informace o třídách, jejich atributech a metodách, aniž by bylo nutné znát tyto informace v době kompilace.

9.4.6. Algoritmy zpracovávající doménové třídy (2. úroveň)

V této chvíli jsem již měl vytvořený základ, nad kterým jsem mohl začít budovat jednotlivé algoritmy.

Algoritmus historizačních doménových tříd

Pod něj spadají například `Parcely`. Ukažme si funkčnost algoritmu na této doménové třídě.

Algorithm 1 Historizační doménové třídy

---

if $DATUM_ZANIKU \neq$ NULL **then**
  if v tabulce PARCELY_MIN není záznam se stejným PK a DATUM_VZNIKU **then**
    instanci doménové třídy uložím do PARCELY_MIN
  end if
  if v tabulce PARCELY existuje záznam se stejným PK a DATUM_VZNIKU **then**
    smažu jej
  end if
else
  if v tabulce PARCELY existuje záznam se stejným PK a $DATUM_VZNIKU <$ datum vzniku instance **then**
    smažu záznam z tabulky PARCELY
    instanci doménové třídy uložím do tabulky PARCELY
  else
    if v tabulce PARCELY neexistuje záznam se stejným PK **then**
      instanci doménové třídy uložím do tabulky PARCELY
    end if
  end if
end if

Algoritmus je celý realizován v abstraktní třídě `HisOracleDatabaseJdbcExporter` a exportéry doménových tříd (3. úroveň hierarchie) do něj pouze doplní `INSERT` atributů doménové třídy a název tabulky.

Algoritmus stavových doménových tříd typu 2

Pod něj spadá v dnešní době pouze doménová třída `SpojeniBMapy`. Algoritmus je stejný jako u historizačních tabulek, a proto je abstraktní třída `Stav2OracleDatabaseJdbcExporter` potomkem `HisOracleDatabaseJdbcExporter` a využívá jeho algoritmus. Rozšiřuje jej však vlastním výběrem primárního klíče. Nezískává ho z databáze jako ostatní třídy, ale rozhoduje o něm podle hodnoty určitých atributů zpracovávané instance.

Algorithm 2 Rozšíření historizačního algoritmu

```
if OP_ID ≠ NULL then
  PK = OP_ID
else
  if DPM_ID ≠ NULL then
    PK = DPM_ID
  else
    if HBPEJ_ID ≠ NULL then
      PK = HBPEJ_ID
    end if
  end if
end if
```

Algoritmus stavových a číselníkových doménových tříd

Pod něj spadají například `Listiny`. Ukažme si funkčnost algoritmu na této doménové třídě.

Algorithm 3 Stavové a číselníkové doménové třídy

```
if v tabulce LISTINY je záznam se stejným PK then
  smažu jej
end if
instanci doménové třídy uložím do tabulky LISTINY
```

Jak vidíme, zpracování stavových tabulek není historizující.

Algoritmus pro HraniceParcel

Vychází z historizačního algoritmu, kde jsou staré záznamy ukládány do historické tabulky. Je však dále rozšířen o poměrně netriviální část, která pracuje s identifikačními čísly parcel, které od sebe tato hranice odděluje. Nebudu tedy tento algoritmus dále rozebírat a případné zájemce odkazuji na prostudování třídy `HraniceParcelOracleDatabaseJdbcExporter`, která jej implementuje.

9.4.7. Exportéry jednotlivých doménových tříd (3. úroveň)

Na nejnižší úrovni stojí exportéry jednotlivých doménových tříd, ve kterých je vždy specifikován jen název tabulky a `INSERT`. Můžeme tedy vidět, že neobsahují žádnou logiku a obsahují jen nezbytné minimum kódu. Pokud bychom například chtěli změnit algoritmus, jakým se exportují stavové tabulky, uděláme to pouze na jednom místě (v abstraktní třídě `StavOracleDatabaseJdbcExporter`). Díky tomu jsou jakékoliv úpravy velmi snadné a rychlé.

9.4.8. Optimalizace a bezpečnost

Během vývoje verze 2.0 jsem strávil poměrně velké množství času samostudiem optimalizačních technik u JDBC.

Použití PreparedStatement

Základem je použití rozhraní `PreparedStatement`.

- reprezentuje předkompilovaný SQL dotaz a je tedy rychlý pro opakované použití (kdy měníme pouze hodnoty parametrů dotazu)
- databáze ukládá do mezipaměti postup, jakým vykonala předchozí dotaz. To umožňuje, aby databáze na další `PreparedStatement` použila stejný plán zpracování. `PreparedStatement` pracuje pouze se změnou parametrů a jeví se tedy databázi jako stále stejný SQL dotaz.
- poskytuje ochranu před SQL injection automatickým escapováním uvozovek a dalších speciálních znaků
- poskytuje oddělení kódu dotazu a hodnot parametrů, což zlepšuje čitelnost a údržbu kódu

Pro každý exportér se vytváří `PreparedStatement` pouze jednou, na začátku, a pak jsou pro potřeby ukládání hodnot do databáze pouze měněny hodnoty parametrů.

Počet dotazů na databázi

Algoritmy zpracovávající změnová data potřebují komunikovat s databází, aby kupříkladu zjistily, zda se tam již hodnota s daným primárním klíčem nevyskytuje. Snažil jsem se po této stránce udělat algoritmy co nejúspornější a počet dotazů snížit na nezbytné minimum.

Například pokud má algoritmus číselníkovým doménových tříd vyhledat záznam se stejným primárním klíčem, a pokud jej najde, tak smazat, znamená to dva dotazy. `SELECT`, který zjistí, zda se v databázi záznam vyskytuje a pokud ano, tak dotaz `DELETE`, který ho případně smaže. Tuto kombinaci dotazů lze nahradit samotným dotazem `DELETE`, který smaže hodnotu z databáze, nebo neprovede nic.

Transakce

Program využívá transakcí. Transakce je uspořádaný balík databázových operací, který je databází prováděn jako jediná operace. Provedou se tedy buď všechny databázové operace, nebo žádná.

Navázání spojení

Spojení se navazuje pouze jednou a následně je v průběhu celého importu doménové třídy do databáze udržováno. Po skončení importu je společně s dalšími zdroji korektně uvolněno.

10. Programátorská dokumentace

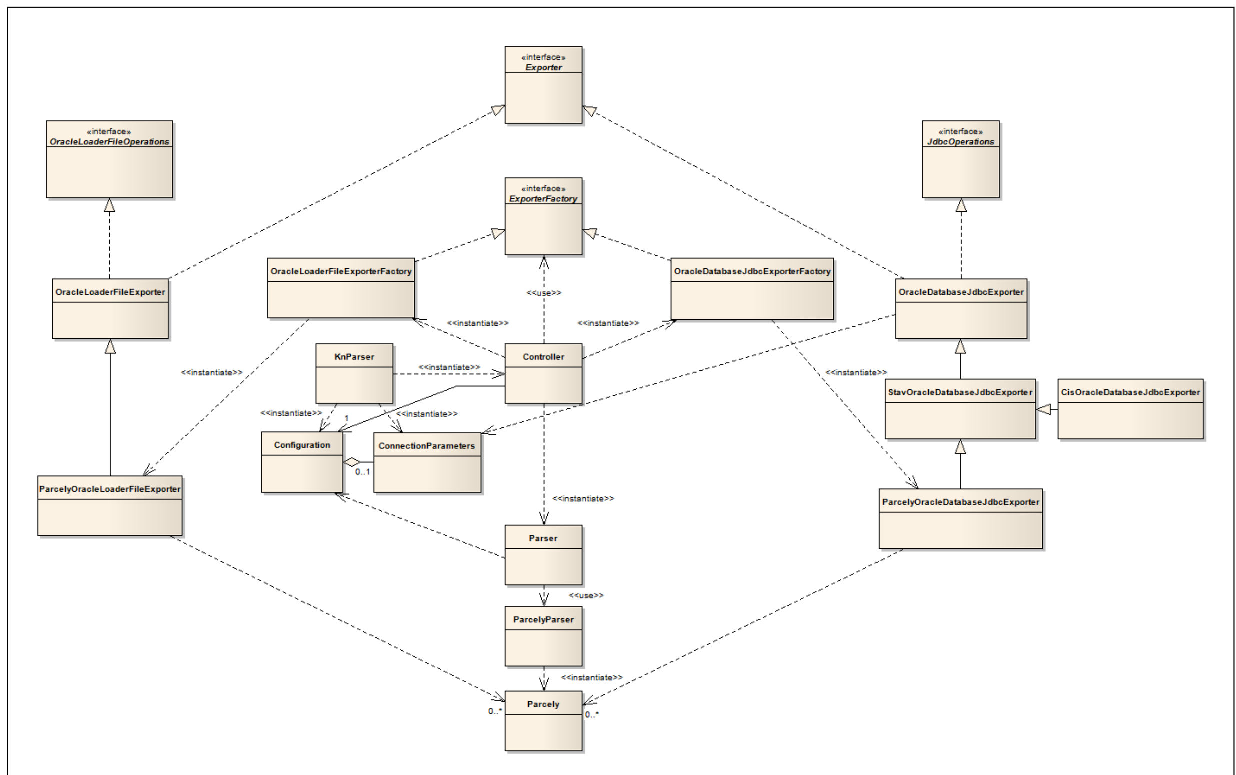
V této části si podrobně projdeme rozčlenění programu. Popíšeme si strukturu projektu a jednotlivé kategorie tříd. U třídy si vždy uvedeme krátký popisek, který vysvětlí, jakou funkci v programu zastává (to slouží především k udržení kontextu, pro podrobnější informace odkazuji na předchozí kapitoly).

10.1. Struktura projektu

Projekt se skládá z následujících položek

- **src/** obsahuje veškeré zdrojové kódy programu. Ty jsou uspořádány do struktury podadresářů. Podadresáře reflektují jednotlivé balíčky, které dodržují jmennou konvenci Javy.
- **lib/** použité knihovny (ojdbc7.jar: JDBC ovladač pro Oracle Database)
- **.settings/** specifická nastavení projektu v Eclipse
- **.classpath** soubor obsahující informace pro JDT ke správnému zkompilování projektu
- **.gitignore** soubor obsahující názvy souborů (případně celé cesty), které nemají být nahrány na GIT
- **.project** soubor popisující projekt (v případě otevření zajišťuje korektní načtení projektu)
- **LICENSE** licence MIT
- **README.md** krátký popis projektu a informace o autorovi
- **pom.xml** XML soubor obsahující informace pro Maven (popisuje závislosti na externích knihovnách, informace potřebné ke zkompilování programu, ...)
- **src/main/java/log4j.properties** obsahuje nastavení pro žurnálovací knihovnu `log4j`

10.2. UML diagram a třídy programu



Obrázek 3. UML diagram projektu. Pro přehlednost bez abstraktních tříd z JDBC exportéru, které nezpracovávají doménovou třídu `Parcely`.

10.2.1. cz.pfreiberg.knparser

Nejvyšší úroveň hierarchie. Obsahuje základní třídy programu.

- **KnParser** je vstupní bod programu. Zpracuje parametry, se kterými byl program spuštěn, vytvoří `Controller`, který zodpovídá za zpracování výměnného formátu, a předá mu kontrolu nad během.
- **Configuration** zapouzdřuje parametry, se kterými byl program spuštěn.
- **ConnectionParameters** zapouzdřuje parametry nutné pro připojení k databázi.
- **Controller** je řídící částí programu. Využívá `Parser` k naplnění doménových tříd ze souboru. Následně iniciuje jejich zpracování a export přes SQL*Loader, nebo Oracle Database.

- **KnParser.properties** je textový konfigurační soubor obsahující informace nutné k napojení do databáze. Je vyplněn uživatelem v případě, že chce exportovat data do databáze.

10.2.2. cz.pfreiberg.knparser.domain

Třídy společné pro všechny doménové třídy.

- **DomainWithDate** je rozhraním, které implementují doménové třídy obsahující atributy datumu vzniku a zániku.
- **Vfk** představuje jednu dávku dat, kterou `Parser` získal ze souboru. Obsahuje tedy kolekce instancí doménových tříd, které jsou předávány ke zpracování.

10.2.3. cz.pfreiberg.knparser.domain.*

V hierarchii se nacházejí ve dvanácti balíčcích všechny doménové třídy a kopírují tak rozdělení dané ČÚZK. Každá doménová třída obsahuje atributy, kde jsou uloženy data datového bloku a metodu `toString()` pro vrácení obsahu v podobě textového řetězce.

Vzhledem k velkému počtu tříd (75) a stejné funkčnosti vypíši pouze první tři balíčky a odkazuji na zdrojové kódy na GitHubu[32], kde lze snadno dohledat všechny doménové třídy.

Pro podrobné informace co jednotlivé třídy představují, jaká data uchovávají a k čemu slouží, odkazuji na oficiální dokument ČÚZK (od strany 25).

Adresní místa

- **Adresa** uchovává datový blok „Adresa“.
- **BudObj** uchovává datový blok „Odkazy objektů na adresy“.

Bonitní díly parcel

- **BonitDilyParc** uchovává datový blok „Bonitní díly parcel“.

Bonitní půdně ekologické jednotky

- **HraniceBpej** uchovává datový blok „Hranice BPEJ“.
- **OznaceniBpej** uchovává datový blok „Označení BPEJ“.

⋮

[32] https://github.com/pfreiberg/knparser/

10.2.4. cz.pfreiberg.knparser.parser

Balíček obsahuje třídy, které soubor výměnného formátu ukládají do doménových tříd.

- **Parser** je třída zpracovávající výměnný formátu katastru nemovitostí. Z konfiguračního souboru načítá počet řádků, které je možné zpracovat v jedné dávce. Ze souboru pak kódování. Po načtení a zpracování jednoho řádku je určeno, do jaké doménové třídy má být uložen. Po zpracování daného počtu řádku (nebo dosažení konce souboru) jsou pak již instance doménových tříd (zapouzdřeny ve třídě `Vfk`) předány `Controlleru`.

- **ParserException** představuje výjimky reprezentující chybové stavy při zpracování souboru.

- **AdresaParser, BudovyParser, . . .** : pro každou doménovou třídu existuje její protějšek v podobě plnící třídy. Je použita jmenná konvence, kdy vezmeme název doménové třídy a přidáme postfix „Parser“. Těmto plnícím třídám `Parser` předává pole typu `String` (obsahující množinu tokenů zpracovaného řádku), které je uloženo do instance nové doménové třídy.

10.2.5. cz.pfreiberg.knparser.exporter

Třídy stojící nad všemi exportéry (ať už se exportuje přes SQL*Loader, nebo JDBC).

- **Exporter** je rozhraní, které implementují všechny exportovací třídy. Představuje obecný exporter, který nemá předepsanou žádnou funkčnost.

10.2.6. cz.pfreiberg.knparser.exporterfactory

Třídy realizující návrhový vzor abstraktní továrny.

- **ExporterFactory** je rozhraní představující abstraktní továrnu, které předepisuje metody pro vrácení exportéru pro každou doménovou třídu.

- **OracleLoaderFileExporterFactory** je třída představující konkrétní továrnu, jež vrací exportéry realizující export přes SQL*Loader.

- **OracleDatabaseJdbcExporterFactory** je třída představující konkrétní továrnu, jež vrací exportéry realizující export přes JDBC do Oracle Database.

10.2.7. cz.pfreiberg.knparser.exporter.oracleloaderfile

Třídy realizující export přes SQL*Loader.

- **OracleLoaderFileOperations** je rozhraní předepisující základní metody pro export přes SQL*Loader. Například vytvoření šablony kontrolního souboru, naplnění datového souboru, ...

- **OracleLoaderFileExporter** je abstraktní třída poskytující metody pro export přes SQL*Loader a předepisuje šablonu, do které si jednotlivé exportovací třídy samotných doménových tříd dosazují svou specifickou metodu (návrhový vzor Template method).

- **AdresaOracleLoaderFileExporter,**
BudovyOracleLoaderFileExporter: pro každou doménovou třídu existuje její protějšek v podobě exportovací třídy pro SQL*Loader. Stejně jako u plnících tříd je použita jmenná konvence, která nám pomáhá udržovat v projektu přehled. Vezmeme název doménové třídy a přidáme postfix `OracleLoaderFileExporter`.
Každá exportovací třída pro SQL*Loader obsahuje volání předka `OracleLoaderFileExporterFactory`, kde předává název souboru, do kterého exportuje (ten koresponduje s názvem doménové třídy). Dále volá metodu zajišťující export doménové třídy, přičemž přepisuje abstraktní metodu v šabloně a kompletuje tak algoritmus, který `OracleLoaderFileExporterFactory` provádí.

10.2.8. cz.pfreiberg.knparser.exporter.oracledatabase

Třídy realizující export přes JDBC do Oracle Database.

- **JdbcOperations** je rozhraní předepisující základní metody pro export přes JDBC. Například navázání spojení na databázi, uzavření spojení, načtení primárních klíčů určité tabulky, vložení záznamu do tabulky, ...

- **OracleDatabaseJdbcExporter** je abstraktní třída poskytující společnou logiku pro práci nad Oracle Database. Zajišťuje navázání spojení, načtení primárních klíčů, ... Poskytuje zjednodušení SQL dotazů pro SELECT a DELETE (potomci pouze předávají informace, co a odkud chtějí získat/smazat).

- **HisOracleDatabaseJdbcExporter** je abstraktní třída poskytující logiku pro historizační tabulky. Stejně jako u třídy `OracleLoaderFileExporter` je využito návrhové vzoru Template method, kdy jednotlivé exportéry doménových tříd, které spadají pod zpracování historických tabulek, pouze doplní do algoritmu svou specifickou část.

- **Stav2OracleDatabaseJdbcExporter** je abstraktní třída poskytující logiku pro stavové tabulky typu 2. Postup zpracování doménových tříd spadajících do této kategorie je z velké části stejný jako u historizačních tabulek. Třída je tedy potomkem `HisOracleDatabaseJdbcExporter`, ale implementuje vlastní postup získání primárního klíče (ten vybírá dynamicky až během exportu instance doménové třídy).

- **StavOracleDatabaseJdbcExporter** je abstraktní třída poskytující logiku pro stavové tabulky. Stejně jako `HisOracleDatabaseJdbcExporter` poskytuje algoritmus pro zpracování určitého typu doménových tříd. Liší se od sebe pouze v algoritmu.

- **CisOracleDatabaseJdbcExporter** je abstraktní třída poskytující logiku pro číselníky. Postup zpracování doménových tříd spadajících do této kategorie je stejný jako u stavových tabulek. Tato třída je tedy potomkem `StavOracleDatabaseJdbcExporter` (dělení je zde pouze formální, aby byly dodrženy skupiny předepsané ČÚZK).

- **HraniceParcelOracleDatabaseJdbcExporter** je speciální třídou, jejíž export vyžaduje specifický algoritmus, který se liší od všech ostatních exportovacích tříd. Protože se jedná o jedinou třídu s takovýmto algoritmem, je celá logika umístěna v samotné třídě exportéru, a ne v nadřazené abstraktní třídě.

- **AdresyOracleDatabaseJdbcExporter, BudovyOracleDatabaseJdbcExporter, . . .** : pro každou doménovou třídu existuje její protějšek v podobě exportovací třídy přes JDBC. Stejně jako u exportu přes SQL*Loader dodržují jednotlivé exportovací třídy jmennou konvenci. Vezmeme název doménové třídy a přidáme postfix `OracleDatabaseJdbcExporter`.
  Každá takováto třída patří do jedné ze čtyř skupin (s výjimkou exportéru pro `HraniceParcel`) a využívají algoritmů, které předepisují jejich předci:

  - `HisOracleDatabaseJdbcExporter`
  - `Stav2OracleDatabaseJdbcExporter`
  - `StavOracleDatabaseJdbcExporter`
  - `CisOracleDatabaseJdbcExporter`

  Všechny exportovací třídy mají společnou strukturu: do svého předka předávají název tabulky a přepisují abstraktní metodu pro ukládání hodnot do databáze.

- **OracleDatabaseParameters** zapouzdřuje parametry, které využívají algoritmy výše jmenovaných abstraktních tříd.

- **JdbcException** třída reprezentující chybové stavy při zpracování souboru přes JDBC.

10.2.9. cz.pfreiberg.knparser.util

Balík tříd, které poskytují pomocné funkce pro ostatní třídy programu.

- **EncodingCzech** je výčtová třída představující dva druhy kódování, ve kterých se soubory výměnného formátu dodávají (iso-8859-2 a windows-1250). Výstup je realizován ve windows-1250.

- **FileManager** zajišťuje ukládání na disk. Vytváří soubory pro ukládání dat a konfiguračních souborů. Následně tyto soubory naplní předanými daty.

- **Operations** je výčtová třída představující matematické symboly porovnávání.

- **VfkUtil** je pomocná třída s metodami pro práci nad výměnným formátem. Obsahuje metody pro získání kódování souboru (a jeho konvertování pro potřeby `BufferedReaderu`), převod tokenů na datové typy atributů doménových tříd, formátování atributů doménových tříd pro SQL*Loader, ...

- **LogUtil** je pomocná třída vytvářející části logů pro žurnálovací knihovnu `log4j`.

11. Uživatelská dokumentace

Program je distribuován v souboru typu JAR[33]. Kromě přeložených zdrojových kódů jsou v něm přiloženy i knihovny nutné pro správnou funkčnost.

Program se spouští přes příkazovou řádku v podobě:

```
java -jar KnParser.jar --input cesta_k_souboru --output
cesta_pro_vystup
```

V případě `--input` a `--output` jde o parametry, kdy za `--input` uvedeme cestu k souboru a za `--output` výstupní místo. Pokud mají parametry tuto podobu, je použito exportování do formátu pro SQL*Loader.

Pokud chceme data exportovat přímo do databáze, tak přidáme parametr `--database`:

```
java -jar KnParser.jar --database --input cesta_k_souboru
```

Vzhledem k tomu, že data importujeme přímo do databáze, tak parametr `--output` můžeme vynechat (když ho zadáme, bude ignorován). V konfiguračním souboru `KnParser.properties`, přiloženém v JARu, musíme nastavit údaje nutné k připojení do databáze. V konfiguračním souboru také můžeme nastavit maximální velikost jedné zpracované dávky. Soubor má následující strukturu:

```
numberOfRows=100000
url=my-database:1234:parser
username=user
password=secret_password
```

`NumberOfRows` udává velikost dávky, `url` adresu databáze a `username/password` pak přihlašovací jméno/heslo.

[33] Standardní kompresní souborový formát používaný na platformě Java. Je založen na kompresi ZIP.

12. Výsledky testů a ostrý provoz

Celý vývoj programu provázelo průběžné testování. To se kromě běžného testování na úrovni objektů a tříd, které má za úkol ověřit funkčnost kódu a správnost algoritmů, skládalo i ze série testů na ČÚZK.

12.1. Externí testování na ČÚZK

Testování na katastrálních pracovištích ČÚZK prováděl Ing. Zícha. Šlo o série tří testů, které sloužily především k odhalení chyb a nedostatků a ověřovaly rychlost programu.

Jednalo se o tyto testy:

1. Ověření na testovacím souboru
   - provedena kontrola počtu importovaných záznamů
   - zkontrolován každý první záznam v každé importované tabulce
   - prověřen importní log na výskyt chyb při importu dat pomocí SQL*Loaderu
2. Ověření na katastrálním pracovišti 508 a 509 (stavová data k 1. 10. 2013)
   - prověřen importní log na výskyt chyb při importu dat pomocí SQL*Loaderu
   - prověřeny hodnoty sloupců, které u všech záznamů mají hodnotu NULL
3. Ověření na datech celé České republiky
   - nahrána stavová data a ty pak třikrát aktualizována změnovými daty (každá aktualizace představovala změny za jeden měsíc). Počet záznamů pak byl srovnán s počtem záznamů z databáze katastru nemovitostí se stavem k 1. 10. 2013.
   - prověřen importní log na výskyt chyb při importu dat pomocí SQL*Loaderu
   - prověřeny hodnoty sloupců, které u všech záznamů mají hodnotu NULL
   - provedeno kompletní porovnání všech hodnot vybraných nejdůležitějších tabulek (`PARCELY`, `BUDOVY`, `JEDNOTKY`, `VLASTNICTVI`, `OPRAV_SUBJEKTY`, `HRANICE_PARCEL`...)

12.2. Výsledky pro verzi 1.0

Jak jsme si uvedli již na začátku, ve verzi 1.0 měl program zpracovávat všechny datové bloky výměnného formátu a ukládat je do formátu, který dokáže zpracovat SQL*Loader. Termín byl určen na polovinu ledna 2014.

Požadavky pro tuto verzi byly splněny. Verze 1.0 byla vydána na GitHubu k 8. 1. 2014[34] a vyhovovala kladeným požadavkům. Měla však několik problémů:

- zpracování stavových dat pro celou Českou republiku trvalo přibližně 21 a půl hodiny. Což byla poměrně dlouhá doba oproti očekávaným 10 až 12 hodinám[35].

- ve verzi 1.0 jsem ještě stále používal pro export dat knihovnu `FileUtils`, což způsobovalo problém s posledním, „bonusovým", řádkem. To sice nebránilo fakticky importovat takto vytvořená data do Oracle Database, ale během importu se vytvářely chybové soubory (které upozorňovaly právě na tento problém).

Rozhodl jsem se tedy před pokračováním práce na verzi 2.0 vydat minoritní verzi 1.1, která bude tyto problémy řešit.

12.3. Výsledky pro verzi 1.1

Verze 1.1 byla vydána o týden později[36] a řešila výše zmíněné problémy. Zpracování a exportovat dat se zrychlilo z 21,5 na 10,5 hodin. Toho jsem dosáhl především přepracováním metod pro ukládání získaných tokenů do doménových tříd. Ve verzi 1.0 během ukládání často docházelo k „vyhazování" výjimek, které nesloužily pouze k opravdovému k ošetření výjimečných stavů, ale i pro určení běhu programu. K odhalení míst v programu, kde se mnohokrát opakují a program na nich tráví spoustu času, jsem použil profilovací nástroj JVMMonitor[37].

Došlo i k dalším úpravám, které měly na zvýšení rychlosti taktéž menší vliv (upravení výchozí velikosti instance `StringBuilderu`, kontrola existence kontrolních souborů SQL*Loaderu, ...).

„Bonusový řádek" byl opraven již dříve zmíněnou implementací vlastního řešení. Verze 1.1 byla dále využita k čištění kódu, opravě prefixu u kontrolních souborů při exportu změnových dat a dalším drobným úpravám.

Verze 1.1 byla v této chvíli připravena na nasazení do ostrého provozu a firma

[34] https://github.com/pfreiberg/knparser/releases/tag/v1.0

[35] Tento předpoklad byl založen na časech, který dosahuje interní program Úřadu pro zastupování státu ve věcech majetkových.

[36] https://github.com/pfreiberg/knparser/releases/tag/v1.1

[37] Profilování označuje snahu najít místa v programu, která jsou vhodná k optimalizaci. JVMMonitor je pak profilovací nástroj, který se formou rozšíření přidává do Eclipse, a dokáže za běhu programu sledovat využití paměti, počty volání jednotlivých metod a čas v nich strávený, ...

Corpus Solutions a.s. ji již v lednu 2014 použila k importu stavových dat katastru nemovitostí do svých databází.

12.4. Výsledky pro verzi 2.0

Verze 2.0 byla vydána 16. 3. 2014. Mezi její hlavní rysy patří:

- přidání exportu přes JDBC (a tedy umožnění aktualizací přes změnová data)
- zrychlení exportu do formátu pro SQL*Loader o 45%-50% (díky implementaci vícevláknového zpracování)
- zjednodušení budoucích rozšíření programu (mezi verzí 1.1 a 2.0 prošel program velkou revizí a čištěním kódu)
- přidání žurnálovací knihovny `log4j`

Jediným problémem, se kterým se tato verze potýká, je rychlost exportu dat přes JDBC. Úzkým hrdlem zde samozřejmě není parsování, ale databáze. Ač je v programu zabudováno poměrně velké množství optimalizačních prvků, tak se při zpracovávání změnových dat generuje poměrně velké množství dotazů mezi databází a programem. Předpokládalo se, že program dokáže změnová data zpracovat a uložit do databáze přibližně za den a půl, nyní však dosahuje výsledku okolo tří dní. Na vyřešení tohoto neduhu se však pracuje a v nejbližší době bude vydána verze 2.1, která toto vyřeší.

S výjimkou nižší rychlosti při exportu přes JDBC však program splňuje ve verzi 2.0 vše, co se očekávalo.

13. Budoucí práce na programu

Jak jsme si řekli již na začátku, výměnný formát katastru nemovitostí je stále ve vývoji. Lze tedy očekávat, že již v průběhu roku 2014 provede ČÚZK úpravy, které bude zapotřebí reflektovat v rámci programu. S přidáváním dalších funkcionalit do **veřejné** verze se však v současné době nepočítá. Slovo veřejné zdůrazňuji záměrně, jelikož v rámci firmy Corpus Solutions a.s. postupně vzniká odnož programu, která je rozšiřována v podobě zásuvných modulů o další funkcionality. Zde můžeme vidět velký přínos licence MIT. Jedinci i firmy mohou nad programem budovat své vlastní projekty a zároveň si mohou zachovat své „know-how“. Ať už z důvodů bezpečnostních[38] nebo finančních.

Veřejná verze programu tedy poběží v servisním režimu, který bude reflektovat případné změny ve výměnném formátu. Kromě toho samozřejmě bude zahrnuta do verze 2.1, která sebou přinese optimalizaci pro export přes JDBC.

Věřím, že se podařilo vybudovat kvalitním základ, který umožňuje široké veřejnosti zpracovávat data katastru nemovitostí a budovat nad ním další rozšíření.

[38]Firmy se snaží držet v tajnosti své databázové systémy, aby případný pokus o kompromitaci těchto systémů byl neúspěšný. Části kódu, které pracují nad databází, mohou ledacos prozradit o jejich struktuře a tím usnadnit útočníkům práci.

Závěr

Cílem práce bylo vytvořit volně šiřitelný program, který umožní načtení a zpracování souborů výměnného formátu katastru nemovitostí, a jejich následné exportování do různých systémů. Důležitým aspektem bylo taktéž snadné rozšíření o nové způsoby exportu. Vznikl tak program šířený pod licencí MIT, který načítá libovolně velké soubory a exportuje je do formátu pro SQL*Loader, či je přímo exportuje přes JDBC do Oracle Database.

Spolupráce na bakalářské práci s firmou Corpus Solutions a.s. probíhala na velmi dobré úrovni a ke vzájemné spokojenosti. Během vývoje programu nedocházelo k časovým prodlevám a byly tak splněny termíny, které byly na začátku vytyčeny. Při vývoji a testování se pracovalo s reálnými daty katastru nemovitostí. Ověření na kompletních datech pro celou Českou republiku považuji za kvalitní zátěžové testy.

Na závěr bych rád konfrontoval výsledný program s cíli, které byly na začátku stanoveny:

- **Rychlost**: rychlost při exportu do formátu pro SQL*Loader předčila očekávání. Původní předpoklad deseti hodin se mi povedlo pokořit o celé čtyři a půl hodiny. Pokud jde o export přes JDBC, tak tam má program stále prostor ke zlepšení a jak jsem uvedl v předchozí kapitole, v nejbližší době bude vydána minoritní verze, která se tímto bude zabývat.
- **Kompletnost a validita**: využití žurnálovací knihovny `log4j` společně s důsledným sledováním nejrůznějších hraničních a výjimečných stavů poskytuje kvalitní základ pro kompletní a validní zpracování dat.
- **Použitelnost**: od začátku vývoje se počítalo s jednoduchým ovládáním přes příkazovou řádku. Naplnění tohoto cíle nezpůsobilo žádný větší problém.
- **Rozšiřitelnost a srozumitelnost**: rozsáhlé úpravy a vylepšení v kódu, které sebou přinesla verze 2.0, poskytují základ, na kterém mohou další subjekty budovat své projekty.
- **Pružnost**: zakomponování novely katastrálního zákona na sklonku roku 2013 se povedla bez problémů. Věřím tedy, že tomu tak bude i nadále.
- **Robustnost**: program obsahuje základní ochrany při práci s externími zdroji (soubory, databáze). Pokud je to možné, snaží se eliminovat chyby a zotavit se z nich s co nejmenším vlivem na kvalitu zpracování dat.

Katastr nemovitostí považuji za projekt, který díky objemu dat, jež zpracovává, nemá v České republice obdoby. Pevně věřím, že se tato práce může zařadit po boku dalších bakalářských a diplomových prací, které se zabývají tímto dozajista velmi zajímavým systémem.

Reference

[1] ČESKÝ ÚŘAD ZEMĚMĚŘICKÝ A KATASTRÁLNÍ, Praha. *Struktura výměnného formátu informačního systému katastru nemovitostí České republiky, úplné znění.* 2013. 101 s.

[2] ČESKÝ ÚŘAD ZEMĚMĚŘICKÝ A KATASTRÁLNÍ, Praha. *Struktura výměnného formátu informačního systému katastru nemovitostí České republiky, dodatky (č.1 - č.15).* 2003 - 2013.

[3] METSKER S. J. *Building Parsers With Java.* Addison-Wesley Professional 2001. 371 s. ISBN-10: 0201719622.

[4] FREEMAN E. a kol. *Head First Design Patterns.* 1 edition, O'Reilly Media 2004. 678 s. ISBN 10: 0-596-00712-4.

[5] MENON R. M. *Expert Oracle JDBC Programming.* 1 edition, Apress 2005. 708 s. ISBN-10: 159059407X.

[6] ORACLE CORPORATION. *Java Platform, Standard Edition 7 API Specification. [Cit. 25.11.2013.]* http://docs.oracle.com/javase/7/docs/api/

[7] ORACLE CORPORATION. *ORACLE CORPORATION: MySQL Documentation. [Cit. 25.11.2013.]* http://dev.mysql.com/doc/

A. Licence použitých zdrojů

Ke korektnímu běhu programu je zapotřebí několika knihoven, které jsou šířeny pod dvěma licencemi: Apache License, verze 2.0 a Oracle Technology Network ("OTN") License.

- Commons IO 2.4 je v programu použita pro potřeby práci se soubory.
- Apache Log4j 1.2.17 je v programu použita pro potřeby žurnálovaní.

Obě dvě knihovny jsou šířeny pod licencí Apache License, verze 2.0 a o jejich stáhnutí z repositářů stará Maven. Nejsou tedy přiloženy fyzicky k programu.

- Oracle Database 12c Release 1 JDBC Driver je v programu použita pro práci nad Oracle Database.

Knihovna je šířena pod licencí Oracle Technology Network ("OTN") License. Ta neumožňuje její šíření přes veřejné repositáře systému Maven. Dovoluje však knihovnu přiložit k programu, který ji využívá.

B. Obsah přiloženého CD a USB flash paměti

K bakalářské práci je přiloženo doprovodné CD, a dále USB flash paměť s testovacím prostředím.

B.1. Struktura CD

`bin/KnParser.jar`
: Spustitelný program distribuovaný v souboru typu JAR.

`doc/Petr Freiberg - bakalářská práce.pdf`
: Dokumentace práce ve formátu PDF.

`doc/Petr Freiberg - bakalářská práce.zip`
: Zdrojový text dokumentace v LaTeX včetně všech externích závislostí (v ZIP archivu).

`src/knparser.zip`
: Zdrojové soubory programu v projektu IDE Eclipse (v ZIP archivu).

`/README.txt`
: Popis obsahu CD.

B.2. Struktura USB flash paměti

`/Testovací prostředí.zip`
: Virtuální disk obsahující testovací prostředí (v ZIP archivu).

`/README.txt`
: Popis obsahu USB flash paměti.

C. Spuštění v programu VirtualBox

Pro potřeby testování programu jsem vytvořil virtuální prostředí formou virtuálního disku pro program VirtualBox (Oracle VM VirtualBox). Jedná se o virtualizační nástroj pro Linux, Windows a OS X. Je zdarma ke stažení na oficiálních stránkách `https://www.virtualbox.org/wiki/downloads`.

Testovací prostředí je připraveno pro zpracování přes SQL*Loader. Před importem přes JDBC mu byla dána přednost z těchto důvodů:

- pro JDBC by bylo nutné vytvořit kompletní databázovou strukturu (včetně primárních i cizích klíčů) a navíc vytvořit speciální tabulky[39], které navrhla pro své potřeby firma Corpus Solutions a.s.
- soubor vyměněného formátu obsahuji v některých ohledech navzájem duplicitní data (na odstraňování duplicit má firma Corpus Solutions a.s. vlastní skripty a postupy). Docházelo by tedy k porušení primárního klíče.
- při importu do databáze přes SQL*Loader není třeba definovat primární klíče a tedy ani řešit duplicity. Navíc vzhledem k tomu, že importujeme stavová data, lze skript opakované použít (databázi se celá smaže a je připravena na nový import dat).

C.1. Postup spuštění programu

Testovací prostředí se nachází na doprovodné USB flash paměti v kořenovém adresáři pod názvem `Testovací prostředí.zip`.

- Rozbalíme ZIP archiv.
- V rozbaleném archivu se nachází složka `Oracle Developer Days`. Otevřeme ji.
- Dvojklikem přidáme soubor `Oracle Developer Days.vbox` do VirtualBoxu (po instalaci VirtualBoxu se nám soubory s příponou `.vbox` automaticky asociují s VirtualBoxem) a spustíme.
- Na přihlašovací obrazovce zadáme uživatelské jméno `oracle` a heslo `oracle`.
- Na ploše se nachází skript `run`. Po spuštění dvojklikem se zobrazí nabídka s několika možnostmi, jak skript spustit. Zde doporučuji „Run in Terminal“, protože se vše, co skript vykonává, vypíše do konzole. Skript vyčistí databázi a výstupní soubory, spustí program `KnParser` a zpracuje s ním data výměnného formátu. Nakonec je na tyto data zavolán SQL*Loader.

[39] Příkladem speciální tabulky může být například tabulka definující primární klíče pro jednotlivé doménové třídy.

- Po doběhnutí skriptu zapneme `SQL Developer`, který se nachází taktéž na ploše. V levém sloupci vidíme seznam databází. Vybereme pravým tlačítkem databázi `SCOTT` a dáme „Connect“. Heslo je `oracle`.

- Poté, co se program připojí na databázi, rozbalíme opět v levém sloupci položku `Tables`, která se nachází ve stromové struktuře databáze `SCOTT`. Nyní již můžeme procházet jednotlivé tabulky s importovanými daty.

C.2. Poznámky

Na ploše se kromě spustitelného skriptu `run` nachází i další položky:

- **Scripts**: složka obsahuje skripty pro vytvoření, smazání a vyčištění tabulek v databázi

- **Data**: dva soubory výměnného formátu, které obsahují stavová data. Soubory poskytla firma Corpus Solutions a.s.

- **KnParser.jar**: program, který popisuje bakalářská práce

- **output**: složka, do které `KnParser` exportuje zpracovaná data připravená pro import přes SQL*Loader

- **logs**: složka s logy, které generuje SQL*Loader pro každý zpracovaný soubor

- **bads**: složka s daty, které se SQL*Loaderu nepovedlo importovat

Po doběhnutí skriptu na ploše najdeme žurnálovací soubor `KnParser.log` s detailními informacemi o běhu programu.

Pro prohlížení textových souborů můžeme použít textový editor `gedit`, který zapneme přes zástupce `Text Editor`. Pro správné kódování českých znaků je zapotřebí soubory otevírat přes `File - Open` a dole zvolit v `Character Coding` kódování „Central European (WINDOWS-1250)“.

Skript lze spouštět opakovaně, protože se vždy vyčistí databáze a smažou výstupní soubory.

Virtualizované prostředí obsahuje citlivá data. Využijte je, prosím, výhradně k otestování funkcionality programu.