

Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce
Migrace databáze

Bc. Martin Lukeš

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Otevřená informatika, strukturovaný, Navazující magisterský

Obor: Softwarové inženýrství a interakce

24. listopadu 2014

Poděkování

Chtěl bych poděkovat vedoucímu své diplomové práce Ing. Ondřeji Mackovi za pomoc s vypracováváním této práce. Dále bych chtěl poděkovat svým kolegům z týmu Migdb, obzvláště Martinu Mazanci, kteří svými připomínkami napomáhali k zkvalitnění této práce a zahlazení některých nepřesností. V neposlední řadě bych chtěl poděkovat firmě CollectionsPro s.r.o, jež přišla s původní myšlenkou, která vedla k vytvoření Migdb týmu.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v příloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 22. 5. 2014

.....

Abstract

This work is concerned with specifying the contract and implement the transformation changes of the application model into changes of the database model. It also deals with the automation of the derivation of the changes applied to one model leading to another without loss or with minimal loss of stored data.

Abstrakt

Tato práce se zabývá upřesněním kontraktu a realizací transformací změn aplikačního modelu na změny modelu databázového. Dále se zabývá automatizací odvození změn vedoucích z jednoho modelu k druhému bez ztráty či s minimální ztrátou uložených dat.

Obsah

1	Úvod	1
1.1	Motivace	1
2	Projekt Migdb	3
2.1	Framework Migdb	3
2.2	Metamodel struktury aplikace	5
2.3	Databázový metamodel struktury	5
2.3.1	Operace nad aplikačním modelem	6
2.3.1.1	Historický vývoj	6
2.3.1.2	Seznam aplikačních operací	7
2.3.1.3	Rozdělení aplikačních operací	13
2.3.2	Delta notace	13
2.3.2.1	Vlastnosti operací	18
2.4	Databázové operace	18
2.4.1	Ukázka kódu	21
2.5	ORMo (ORM operací)	22
2.5.1	Diff elementy	25
2.6	Rozpoznávání operací	26
2.7	Obecné principy model matching	26
2.7.1	Graph matching	27
2.8	Vytvořený algoritmus rozpoznávání operací	28
2.8.1	Návrh ze studia článků	28
2.8.2	Implementace	28
2.9	alternativní algoritmus	29
3	Popis problému, specifikace cíle	35
4	Testování projektu Migdb	37
5	Ukázka zdrojového kódu práce	39
6	Obsah přiloženého CD	43
7	Závěr	45
7.1	Další poznámky	47
7.1.1	České uvozovky	47

8	Seznam použitých zkratk	49
9	UML diagramy	51
10	Instalační a uživatelská příručka	53
11	Obsah přiloženého CD	55
	Literatura	57

Seznam obrázků

2.1	Rozdělení vrstev softwaru - převzato z [Maz14]	5
2.2	Rozdělení vrstev softwaru - převzato z [Tar12] a částečně upraveno	6
2.3	Aplikační metamodel v počátku vývoje obrázek převzat z [Luk11]	7
2.4	Aplikační metamodel v průběhu vývoje obrázek převzat z [Jez12]	8
2.5	Rootové elementy aplikačního modelu	9
2.6	Struktura aplikačního metamodelu	10
2.7	Struktura databázového metamodelu	11
2.8	Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Tar12]	30
2.9	Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Luk11]	31
2.10	AddParent(Teacher, UniversityTeacher)	31
2.11	Výsledek po operaci AddParent	32
2.12	Výsledný stav po aplikaci RemoveParent	32
2.13	Seznam diff elementů	33
2.14	Typy graph matchingu	33
6.1	Seznam přiloženého CD	43
11.1	Seznam přiloženého CD — příklad	55

Seznam tabulek

Kapitola 1

Úvod

1.1 Motivace

V průběhu poslední dekády je vyvíjeno více nového softwaru než kdy předtím a současně je i stávající software stále více a častěji modifikován, ať už je to zapříčiněno existencí rozsáhlého legacy systému, špatného návrhu či upravováním funkcionality softwaru. Dá se předpokládat, že díky masivnímu rozšíření informačních technologií, obzvláště mobilních tento trend nejenže bude pokračovat, ale bude i dále na vzestupu.

Díky nutnosti zpracování a ukládání velkého množství dat se již od padesátých let dvacátého století prosazovaly myšlenky vedoucí k vytvoření speciálních systémů k těmto účelům určeným - tento software se v české odborné literatuře nazývá systém řízení báze dat (SRBD).

Kvůli nutnosti dokumentace a komunikace mezi vývojáři vznikají různé typy modelů. Objektový model aplikace popisuje strukturu aplikace a je doplněn modelem databázovým modelem popisujícím stav databáze. Nejrozšířenějším typem databáze jsou v nynější době databáze relační, která uspořádává data podle relačního modelu. Relační model je dle [Val14a] formální abstrakce využívající relaci jakožto jediný konstrukt. Relace je uspořádaná n -tice souvislých dat $R(A_1:D_1, A_2:D_2, \dots, A_n:D_n)$, přičemž "R" je název relace, A_i jsou názvy atributů a D_i jsou k nim přidružené typy. K zajištění konzistence jsou v SRDB používána Integritní omezení (IO). IO jsou dle [Val14b] tvrzení vymezující korektnost DB, stupeň souladu datového obrazu s předlohou (jaká data v databázi mohou být a jaká již ne).

Relační algebra definuje pomocí relací a integritních omezení strukturu databáze. Dotazovacím aparátem pro data definovaná pomocí relační algebry je relační algebra. Relační algebra využívá operátorů \cup (sjednocení), \cap (průnik), \setminus (množinový rozdíl), \times (kartézský součin), selekce značená $R(\varphi) = \{ u \mid u \in R \text{ a } \varphi(u) \}$ a projekce značená $R[C] = \{ u[C] \mid u \in R \}$ a operace přirozené spojení $T(C) = R * S = \{ u \mid u[A] \in R \text{ a } u[B] \in S \}$. [?] dále definuje relačně úplný jazyk jako takový, který umožňuje realizovat relační algebru. Takovým jazykem je například jazyk SQL (Structured Query Language). Relační databáze implementuje relaci tabulkou, její atributy A_i jednotlivými sloupci s typy D_i . V dotazovacím jazyce SQL nahrazuje projekci $R[a_1, \dots, a_n]$ operací `SELECT a1, ... an FROM R`, operaci selekce $R[a]$ klauzulí `where` v dotaze `select SELECT * FROM R WHERE a = 1`;

Aby byla aplikace funkční, je nutné zajistit konzistenci mezi databázovým a aplikačním modelem. Tento problém byl již vyřešen a jeho řešení bývá v odborných kruzích nazýváno objektově relační mapování (ORM) [wc14b]. Dnešní implementace ORM jsou schopny nejen

transformovat aplikační model na model databázový, ale také vyjádřit změnu v struktuře aplikace pomocí skriptů Data definition Language (DDL), podmnožiny jazyka SQL. Tyto skripty pozmění model databázový tak aby odpovídal modelu aplikačnímu. Tato konzistence je zaručena automatickou transformací aplikačního modelu na model databázový, což vývojářům softwaru šetří čas strávený vývojem softwaru.

Problém nastává, jakmile zahrneme do zachování nejen strukturu dat, ale i samotná data. Změnit strukturu dat a zároveň transformovat data tak, aby měla stejnou vyjadřovací schopnost jako původní data - považujeme změny aplikačního modelu jako smazání třídy, atributu apod za změny zachovávající informaci.

Kapitola 2

Projekt Migdb

Tato diplomová práce byla napsána v rámci projektu Migdb, který vznikl díky spolupraci akademické sféry se sférou komerční v roce 2011. Komerční sféra byla v tomto případě zastoupena firmou CollectionsPro, s.r.o (dále jen CP) a akademická potom Katedrou počítačů fakulty Elektrotechnické na pražském Českém vysokém učení technickém. Ambiciózním cílem tohoto projektu bylo od počátku projektu definování ucelené množiny změn, tj. operací, kterými mohou vývojáři změnit model aplikace a transformovat tyto změny do spustitelného SQL skriptu, který změní strukturu databáze a přesune data do odpovídajících elementů z modelu aplikace.

Tato diplomová práce se zabývá se zkoumáním změn aplikačního modelu, jejich popisem a rozpoznáváním změn vedoucích od jednoho aplikačního modelu vedoucích k druhému. Dále pak dokončuje a upřesňuje kontrakt takzvaných operací nad aplikačním modelem a popisuje jejich transformaci na změny modelu databázového a následným vygenerováním SQL příkazů spustitelných nad relační databází PostgreSQL.

V rámci Migdb bylo v posledních letech vytvořeny a obhájeny 4 bakalářské práce a 2 diplomové práce členů Migdb.

Jednalo se o tyto bakalářské práce - bakalářská práce mé osoby [Luk11], jež pojednávala o problematice mapování aplikačního modelu na model databázový, bakalářské práce Jiřího Ježka [Jez12] popisující aplikační model a jeho transformace, bakalářská práce Petra Taranta [Tar12] popisující databázový model a jeho transformace a poslední bakalářskou prací je práce popisující testování projektu [Luk13].

V roce 2014 byla obhájena diplomová práce Petra Taranta [Tar14] formálně specifikující aplikační operace a diplomová práce Martina Mazance [Maz14] definující doménově specifikující jazyk aplikačních operací.

Projekt Migdb byl započat v spolupráci se společností Collections Pro. Výsledky dosavadní práce byly v roce 2012 prezentovány jako case-study na prestižní modelové konferenci Code Generation 2012 [PMH12] v Anglickém Cambridge.

2.1 Framework Migdb

Tato kapitola je věnována krátkému představní frameworku Migdb. Projekt Migdb se věnuje evolučním procesem v průběhu vývoje software. Software použitý v této práci je reprezentován na obr. 2.1 převzatého z [Maz14]. Objektová vrstva každé aplikace je reprezentována jejím

modelem obsahujícím entity aplikace. Databázová vrstva zajišťující perzistenci dat obsahuje jednak schéma definující strukturu databáze, ale také samotné perzistované instance dat.

Proces evoluce změn v životním cyklu je definován v [Jez12] následovně:

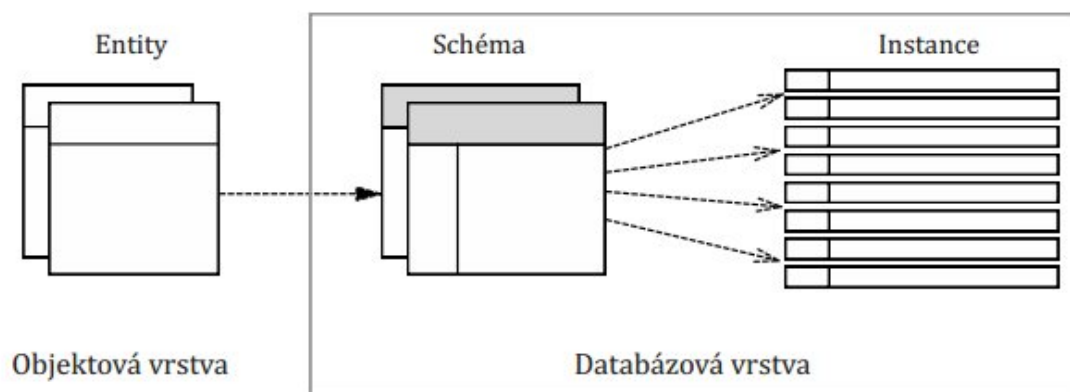
1. redefinice entit na objektové vrstvě
2. opětovná tvorba databázového schématu dle redefinovaných entit
3. sběr potřebných informací pro migraci dat a příprava migračního skriptu
4. ověření proveditelnosti migračního skriptu
5. migrace dat

Po bližší analýze potřeb společnosti CP a po ověření nadbytečnosti vrstvy ověřující proveditelnost migrovaných skriptů byl tento koncept přepracován do formy zobrazené na obrázku 2.2. Na obrázku vidíme aplikační a databázový metamodel, které jsou uloženy v souborech s příponou *ecore*. Instancemi aplikačního APP metamodelu jsou jednotlivé generace aplikačního modelu. Instancemi databázového RDB metamodelu jsou potom generace RDB modelů. Aplikační i databázové modely jsou uloženy ve formě *.xmi* souborů. Změny, které se provádějí v aplikaci jsou transformovány na databázové operace aplikovatelné na RDB model. Databázové operace jsou potom transformovány na SQL skripty. V původní představě byl do frameworku zapojen i exekutor těchto skriptů nad databází, ale po konzultaci s CP byl vzhledem k znovupoužitelnosti SQL souborů z frameworku vypuštěn.

V rámci projektu jsem vytvořil ve své bakalářské práci [Luk11] ORM transformaci aplikační struktury na databázovou a následně vygenerování SQL skriptu vytvářející strukturu aplikace. Tento nástroj není ve frameworku použit při nasazení, ale byl použit při testování frameworku - viz kapitola 4. Framework Migdb pracuje oproti původní sekvenční představě viz 2.1 iteračně. V první iteraci je první aplikační operace aplikována na aplikační model, transformována do databáze, kde je její obraz (sekvence databázových operací) aplikován na databázový model. Po provedení první iterace prochází tímto cyklem druhá operace, potom třetí ...

Ačkoliv Evoluce na databázové úrovni nemá vliv na tvar vygenerovaných SQL skriptů, ale po konzultaci s CP jsme usoudili i po zkušenostech z praxe s databází PostgreSQL, že evoluce databázového modelu odhalující chyby je užitečným uspořením času vývojáře, protože aplikace některých SQL skriptů může trvat(a typicky trvá) delší čas než průběh evoluce RDB modelu a dobrým nahrazením přímé exekuce SQL skriptů, proto byl tento modul ve frameworku zachován.

V průběhu vývoje existoval koncept *ComposedOperation* a *AtomicOperation*, kdy každá operace byla buď *Composed* nebo *Atomic*, každá *Composed* operace byla na aplikační vrstvě nejdříve dekomponována na set atomických, které se později vykonaly a mapovaly přes ORM o na databázové operace. Tento koncept jsme zavrhlí, protože jsme nedokázali dekomponovat správně některé operace a obzvláště pořadí ORM obrazu nám dělalo problémy. Nicméně v nynější chvíli by se tento koncept mohl navrátit po přidání konceptu *mirrored operations*, ale asi by bylo nutné předefinovat či přidat některé aplikační operace.



Obrázek 2.1: Rozdělení vrstev softwaru - převzato z [Maz14]

2.2 Metamodel struktury aplikace

Metamodel struktury aplikace zachycuje vztahy mezi jednotlivými objekty tvořícími aplikaci. Tento model byl vytvořen již v raných fázích projektu Migdb a byl často upravován, většinou zjednodušován. V raných fázích projektu 2.6 viz [Luk11] byl metamodel struktury aplikace ekvivalentní metamodelu aplikačnímu. Postupem času byl aplikační metamodel obohacen o strukturu operací zobrazenou na obr. 2.4 a seznam Diff elementů viz. seznam 2.13. Ačkoliv tušíme, že tento způsob uchování metamodelů v jednom souboru ideální, vybrali jsme si ho kvůli snadnější údržbě.

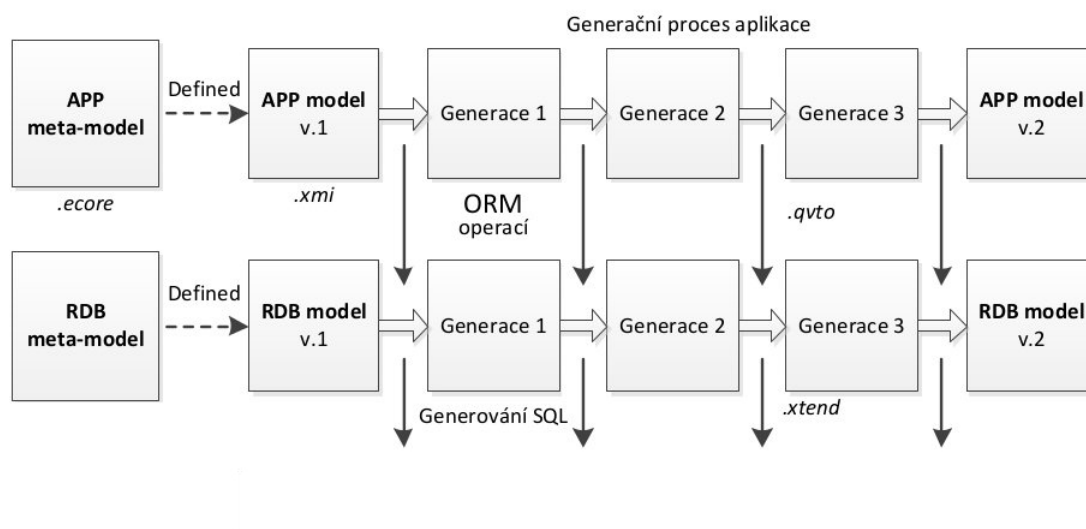
Na obrázku 2.5 jsou znázorněny kořenové elementy nynějšího aplikačního modelu - každý aplikační model musí obsahovat jeden container - potomka třídy ModelRoot. Na obrázku 2.6 jsou zobrazeny elementy patřící do Struktury aplikačního modelu. Při vývoji byla přejmenována třída Class, kterou je možno vidět na 2.3 z původního metamodelu na StandardClass. Oproti aplikačnímu metamodelu [Jez12] byly odstraněny entity EmbeddedClass a její předek GeneralClass, dále byla zjednodušena třída Property, u níž ubyly atributy defaultValue, sequenceName a atribut isId. Atribut isId byl nahrazen přímou referencí na idProperty ve třídě StandardClass který byl nahrazen referencí.

Koncept generace modelů byl zachován, ale tyto generace nejsou obsaženy z implementačních a testovacích důvodů v jednom souboru, ale ve více souborech.

2.3 Databázový metamodel struktury

Stejně jako metamodel struktury aplikace byl i metamodel databázové struktury původně ekvivalentní databázovému metamodelu a následně byly do metamodelu vloženy databázové operace. Metamodel databázové struktury se ukázal jako celkem dobře definovaný a proto změny na něm udělané ho zjednodušovaly.

Na obr. 2.7 vidíme aktuální matamodel databázové struktury, na obr. 2.9 vidíme model na počátku vývoje převzatý z [Luk11] a na obrázku 2.8 model v pozdější fázi vývoje převzatý z [Tar12].



Obrázek 2.2: Rozdělení vrstev softwaru - převzato z [Tar12] a částečně upraveno

Nejvýraznější změnami databázového metamodelu struktury jsou - stejně jako v metamodelu aplikační struktury odstranění generace modelů, odstranění elementu `UnderlyingIndex`, odstranění elementu `ColumnConstraint`, nahrazení elementu `NotNullConstraint` atributem boolean v `Column` a snížení kardinality `Sequence` obsažených ve schématu z * na 1.

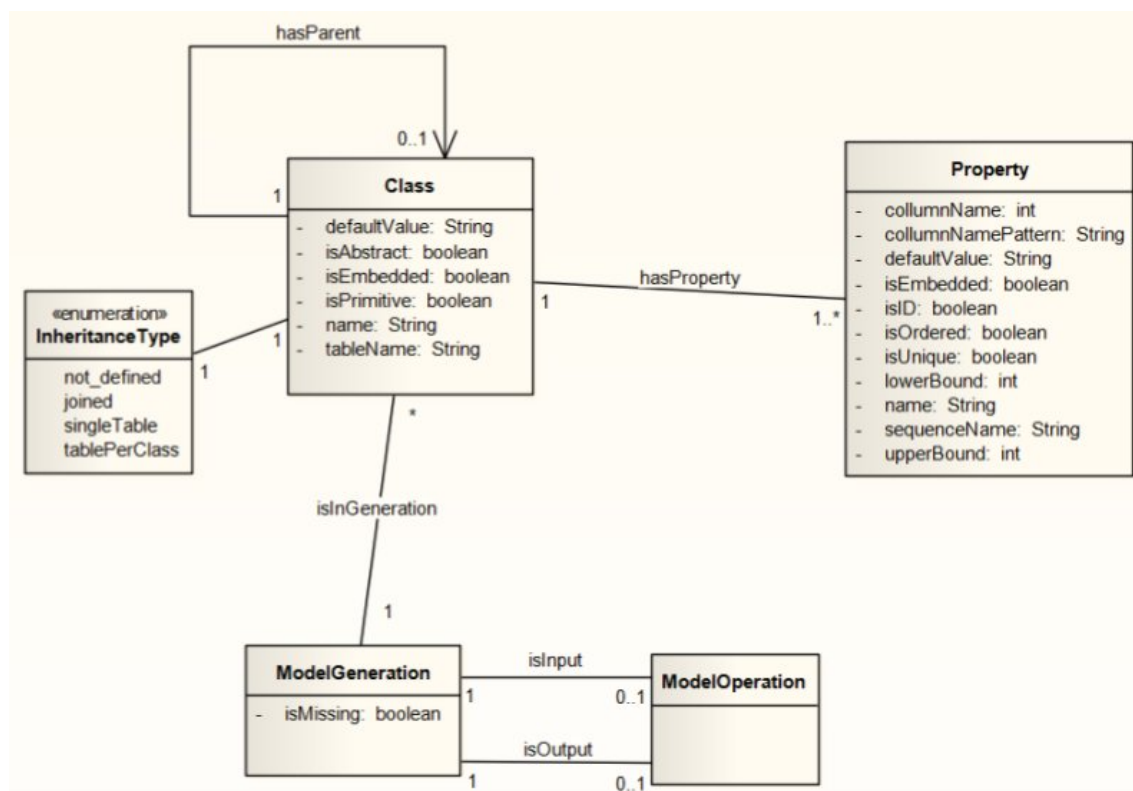
Myšlenka generace modelů byla zachována, ale jejich případné uchovávání bylo zvoleno ve více oddělených souborech. Element `UnderlyingIndex` byl shládán nadbytečným, stejně jako element `ColumnConstraint`. Neatributový element `NotNullConstraint` byl shledán příliš informačně chudým a byl nahrazen atributem `isNillable` zachovávajícím stejnou informační hodnotu jako element v původních modelech.

2.3.1 Operace nad aplikačním modelem

2.3.1.1 Historický vývoj

V průběhu modelování operací nad aplikačním modelem jsme se snažili, aby tyto operace byly jednoznačné (strojově zpracovatelné) v rámci daného kontextu, dále vzhledem k nutnosti textového zápisu uživatelem o minimalističnost zápisu. Tyto dva koncepty jdou obecně proti sobě, proto jsme došli k jistému jejich kompromisu uživatelské jednoduchosti zápisu a jednoznačnosti.

Operace v aplikačním modelu se vyvíjely a měnily se jejich parametry, ale současně se měnil i seznam dostupných operací nad aplikačním modelem. Z operací v první verzi modelu byly odstraněny operace `MoveProperty`, `AddPrimitiveClass`, `SetOpposite` a `SetType`. Operace `AddPrimitive` byla označena za nadbytečnou, protože není cílem modifikace modelu změna seznamu primitivních tříd, který bývá definován použitým programovacím jazykem a

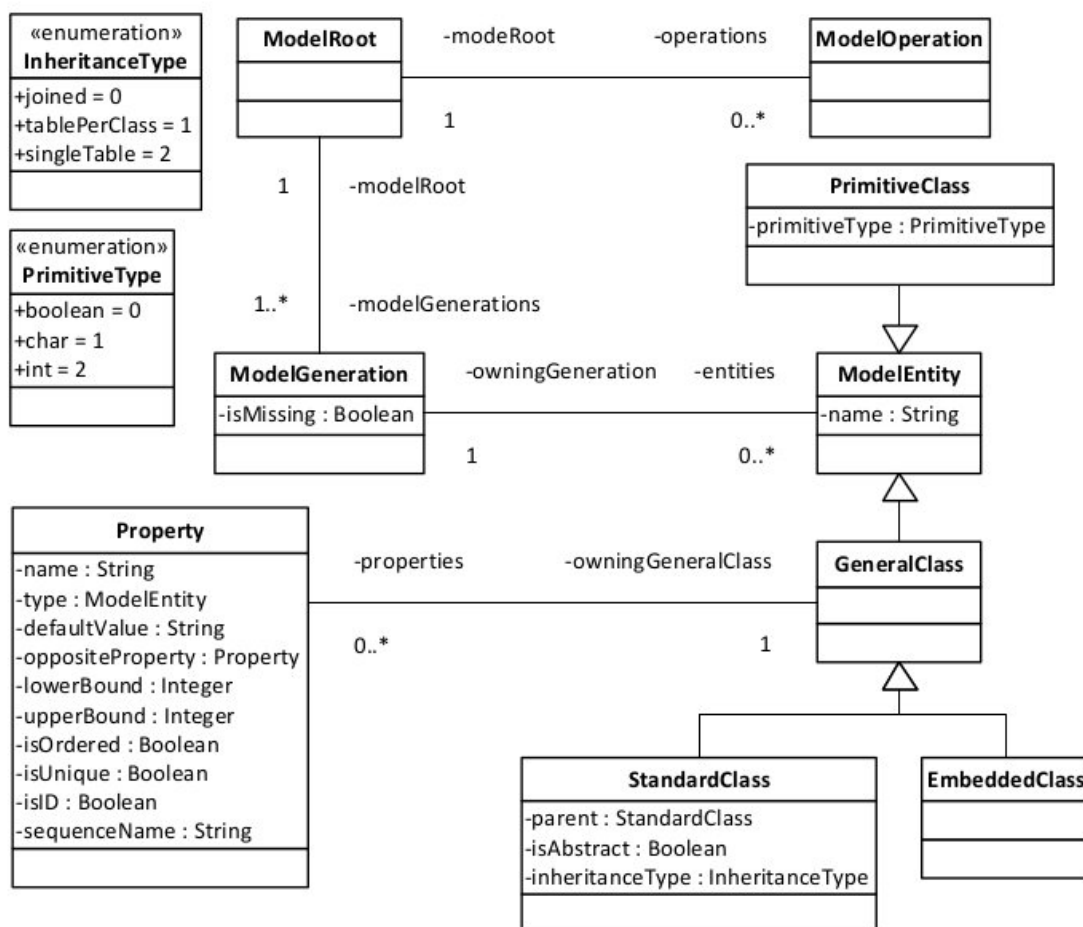


Obrázek 2.3: Aplikační metamodel v počátku vývoje obrázek převzat z [Luk11]

tudíž by měl tento seznam být v vstupní generaci. V průběhu analýzy operace SetOpposite bylo zjištěno, že tato operace má smysl na strukturální úrovni, ale není možné ji aplikovat na obecná data, proto byla tato operace nahrazena dvojicí operací ChangeUniToBidir a ChangeBiToUnidir, které plní nároky kladené na původní operaci a jsou aplikovatelné na datové úrovni. Operace SetType byla prozkoumána, ale nebyla exaktně popsána, nebylo nalezeno její mapování na operace v databázi ani validační podmínky nutné k úspěšné aplikaci operace na aplikační model. Předpokládáme, že by tato operace měla souviset s dědičnými hierarchiemi.

2.3.1.2 Seznam aplikačních operací

Operace jsou uvedeny v následujícím seznamu. Kromě regulérních operací, které může vytvořit uživatel jsou v tabulce zde uvedeny i virtuální operace DistributeProperty, MergeProperty a operace ExportProperty, které jsou používány jako pomocné v implementaci složitějších reduktivních a expanzivních operací a manipulují s Property v rámci dědičné struktury tříd. Tyto operace mohou narozdíl od nevirtuálních operací být aplikovány na model, který je z nějakého hlediska nevalidní. Například operace MergeProperty počítá s hierarchií s kolizní property v třídě předka a potomka. Hlavním přínosem virtuálních operací je zabránění duplikace v definici ORM o mapování a zjednodušení kódu.



Obrázek 2.4: Aplikační metamodel v průběhu vývoje obrázek převzat z [Jez12]

Operace I: AddStandardClass(name, isAbstract, inHeritanceType)

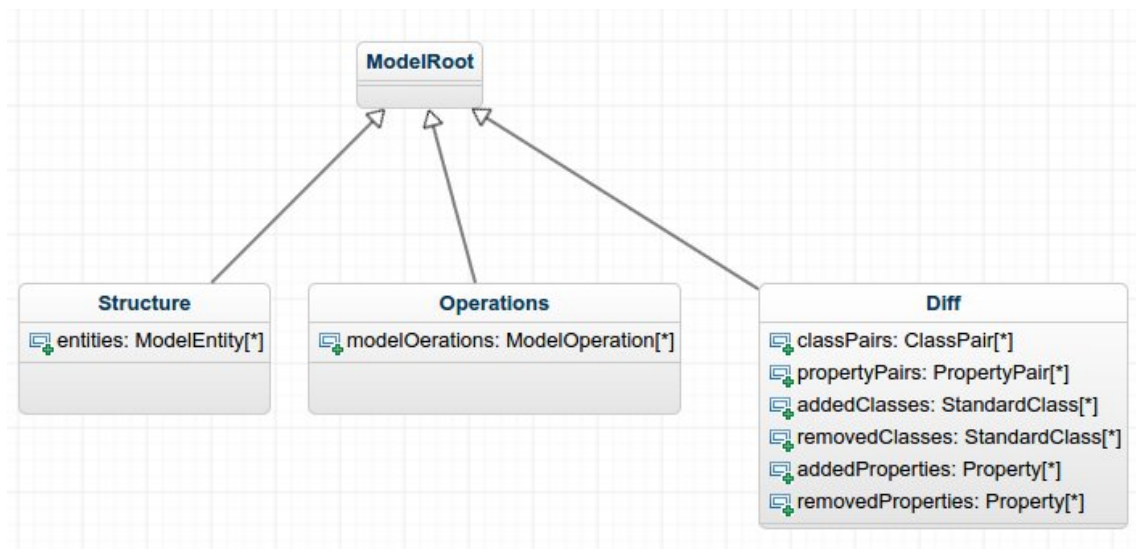
- Validační podmínky - neexistuje třída s jménem nově vznikající
- Operace vytvoří novou třídu a její id odvozené z názvu třídy

Operace II: RenameEntity(name, newName)

- Validační podmínky - existuje třída s původním jménem, neexistuje třída s novým jménem
- Operace změní název třídy na nový

Operace III: SetAbstract(name, isAbstract)

- Validační podmínky - existuje třída s daným jménem
- Operace nastaví třídě atribut abstract na danou hodnotu



Obrázek 2.5: Rootové elementy aplikačního modelu

Operace IV: RemoveEntity(name)

- Validační podmínky - existuje třída s daným jménem, neexistuje link na tuto třídu, třída neobsahuje žádné property, neexistuje pro tuto třídu žádný potomek
- Operace odstraní entitu (standardní třídu) z modelu

Operace V: AddProperty(owningClassName, name, typeName, lowerBound, upperBound, isOrdered, isUnique)

- Validační podmínky - zadané bounds jsou validní, v hierarchii dědičnosti neexistuje kolizní property se stejným jménem
- Operace vytvoří v dané třídě novou property se zadanou horní mezí, dolní mezí, typem, seřaditelností a unikátností

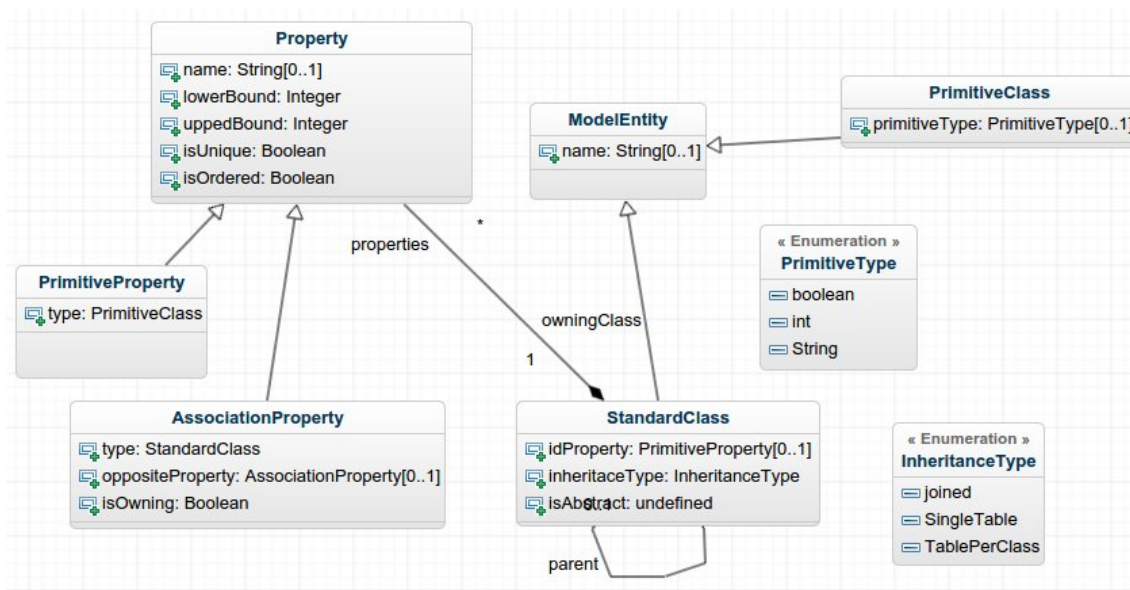
Operace VI: RenameProperty(owningClassName, name, newName)

- Validační podmínky - existuje přejmenovaná property v dané třídě, neexistuje property v dané třídě nového jména
- Operace změní název property v dané třídě ze starého na nový

Operace VII: RemoveProperty(owningClassName, name)

- Validační podmínky - musí existovat vlastnická třída property a v ní odstraňovaná property
- Operace odstraní property z dané třídy

Operace VIII: SetBounds(upperBound, lowerBound)



Obrázek 2.6: Struktura aplikačního metamodelu

- Validační podmínky - bounds musí být validní a musí existovat daná třída a property
- Operace nastaví horní a dolní mez property na nové hodnoty

Operace IX: SetOrdered(owningClassName, name, isOrdered)

- Validační podmínky - musí existovat daná třída a property
- Operace nastaví property seřaditelnost

Operace X: SetUnique(owningClassName, name, isUnique)

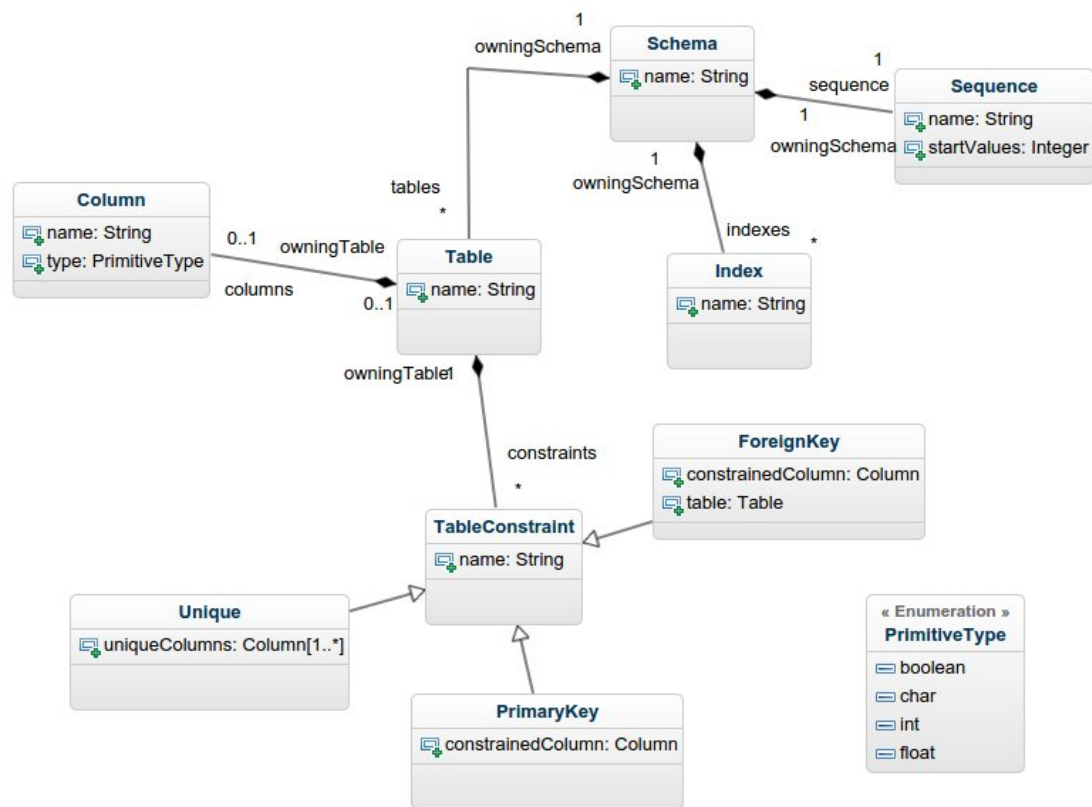
- Validační podmínky - musí existovat daná třída a property
- Operace nastaví property unikátnost

Operace XI: AddParent(className, parentClassName)

- Validační podmínky - musí existovat rodičovská třída a třída potomka, třída potomka nesmí mít nastaveného rodiče
- Operace nastaví třídě předka a přesune namerguje kolizní atributy do rodičovské třídy

Operace XII: RemoveParent(className, parentClassName)

- Validační podmínky - musí existovat třída child a mít nastavenou rodičovskou třídu
- Operace odstraní třídě rodičovskou třídu a rozdistribuje (použije virtuální operaci distribute property) property z rodičovské třídy do třídy původního potomka



Obrázek 2.7: Struktura databázového metamodelu

Operace XIII: `ExtractClass(sourceClassName, extractClassName, associationPropertyName, oppositePropertyName, propertyNames)`

- Validační podmínky - musí existovat zdrojová třída, neexistuje property s jménem linku na nově vzniklou třídu, existují exportované property
- Operace vytvoří novou třídu, kterou napojí na původní třídu, exportuje (využije virtuální operaci `export property`) do nově vzniklé třídy vyjmenované property

Operace XIV: `InlineClass(targetClassName, associationPropertyName)`

- Validační podmínky - musí existovat cílová třída a musí existovat asociační property typu `Inlinované třídy`, která má upper bound 1
- Operace exportuje všechny property z inlinované třídy do cílové třídy přes specifikovanou unidirectional asociaci

Operace XV: `ChangeUniToBidir(className, associationPropertyName, oppositePropertyName)`

- Validační podmínky - v dané třídě musí existovat asociační property s daným jménem a nesmí mít nastavenou `opposite property`

- Operace vytvoří nový zpětný link s `oppositePropertyName` k property `targetClassName` a nastaví správně data do `oppositePropertyName`

Operace XVI: `ChangeBiToUnidir(className, associationPropertyName)`

- Validační podmínky - v dané třídě musí existovat asociační property s daným jménem a musí mít nastavenou `opposite property`
- Operace odstraní opoziční property

Operace XVII: `CollapseHierarchy(superClassName, subClassName, isIntoSub)`

- Validační podmínky - musí existovat subclass a superclass, subclass musí mít nastavenou superclass jako parenta
- Operace exportuje (aplikuje virtuální operaci) všechny property z jedné třídy do jejího předka a třídy spojí, upraví dědičné vazby

Operace XVIII: `ExtractSubClass(sourceClassName, extractedClassName, extractedPropertyNames)`

- Validační podmínky - musí existovat třída s name `sourceClassName` a nesmí existovat třída s jménem `extractedClassName`, v třídě `sourceClass` musí existovat property s názvy z kolekce `extractedPropertyNames`
- Operace vytvoří třídě nového potomka a exportuje (aplikuje virtuální operaci `export property`) do něj vyjmenované property

Operace XIX: `ExtractSuperClass(sourceClassesName, extractParentName, propertyNames)`

- Validační podmínky - musí existovat třída s name `sourceClassName` a nesmí existovat třída s jménem `extractedParentName`, v třídě `sourceClass` musí existovat property s názvy z kolekce `propertyNames`
- Operace vytvoří třídě nového předka a přesune do něj vyjmenované property, pokud měla původní třída předka nastaví tohoto předka rodičem nově vzniklé třídy

Operace XX: `PullUpProperties(childClassName, pulledPropertiesNames)`

- Validační podmínky - musí existovat `childClass` a mít nastavenou rodičovskou třídu, v třídě potomka musí existovat properties z kolekce `pulledPropertiesNames`, v okolních subhierarchiích nesmí existovat properties z této kolekce
- Operace exportuje (aplikuje virtuální operaci `export property`) property do rodičovské třídy

Operace XXI: `PushDownProperties(childClassName, pushedPropertiesNames)`

- Validační podmínky - musí existovat class s `childClassName` a mít nastavenou `parentClass`, v třídě potomka musí existovat properties z kolekce `pushedPropertiesNames`
- Operace exportuje (aplikuje virtuální operaci `export property`) vyjmenované property do třídy potomka a přesune JEN data potomka

Operace XXII: ExportProperty(exportedPropertyName, className)

- virtuální operace
- Operace přesune property a data v ní obsažená v rámci hierarchie do cílové třídy

Operace XXIII: DistributeProperty(distributedPropertyName, className)

- virtuální operace
- Operace zduplikuje strukturu v rámci hierarchie dané property do cílové třídy a přesune data přiřazená této třídě

Operace XXIV: MergeProperty(mergedPropertyName, className)

- virtuální operace
- Operace přesune data zdrojové property do cílové property a smaže strukturu původní property

2.3.1.3 Rozdělení aplikačních operací

Operace nad aplikačním modelem je možné dělit podle dvou kritérií - 1. nad jakým typem modelové entity pracují, 2. jaký je charakter/význam pro tuto entity daná operace má. Operace byly rozděleny podle obou kritérií spíše formálně. Všechny operace jsou potomkem generické operace ModelOperation. Rozdělení podle druhého kritéria vzniklo až po přidání funkcionality rozpoznávání operací.

První kritérium dělí aplikační operace na operace pracující s třídami a operace pracující pouze s properties daných tříd. Příkladem operací pracujících s třídami jsou operace AddStandardClass, AddParent a RemoveEntity. Příkladem operací pracujících s properties jsou operace AddProperty, RemoveProperty, SetAbstract.

Podle druhého kritéria je možné rozdělit operace nad aplikačním modelem do 5 skupin - konstruktivní, destruktivní, expanzivní, reduktivní a modifikační operace. Konstruktivní operace jsou takové, které po své aplikaci vytvoří 1 novou entitu v výsledném modelu, která nemá žádné vazby na jiné entity. Příklady aditivní operace je operace AddClass.

Destruktivní operace je opak konstruktivní, v vstupním modelu existuje entita a ta je aplikací destruktivní operace odstraněna. Příkladem destruktivní operace je operace RemoveProperty.

Operace expanzivní přidává do výstupního modelu jednu entitu, čímž se podobá operaci konstruktivní, nicméně zároveň je vázána na jinou entitu stejného typu a zmenšuje její obsah. Příkladem expanzivní operace je ExtractClass.

Reduktivní operace entitu z vstupního modelu odstraní a zároveň entitě, která je pro operaci řídicí změni obsah. Příkladem této operace je InlineClass. Reduktivní operace jsou inverzní k operacím expanzivním.

2.3.2 Delta notace

V [Cic08] jsou definovány operace pomocí delta notace. Delta notace je taková, která ukazuje všechny změny mezi danými dvěma artefakty stejné úrovně abstrakce. Změnami můžeme rozumět přidáním elementu, odstraněním elementu a modifikací elementu.

Nejznámějším a nejrozšířenějším typem delta notace je výstup linuxového příkazu diff. Každý rozdíl v delta notaci příkazu diff je dle [wc14a] definován následně:

```
popis změny
<řádek z prvního souboru
<řádek z prvního souboru...
---
>řádek ze druhého souboru
>řádek ze druhého souboru...
```

Popis změny může nabývat tvaru: LaR, FcT nebo RdL.

LaR

Ve druhém souboru jsou navíc řádky R patřící za řádek L prvního souboru. Např. 8a12, 15 znamená, že ve druhém souboru jsou navíc řádky 12-15 a patří za řádek 8 v prvním souboru.

FcT

Řádky F z prvního souboru byly změněny. Ve druhém souboru jsou jim odpovídající řádky T. Např. popis 5,7c8,10 znamená, že se liší řádky 5-7 v prvním souboru a jim odpovídající jsou řádky 8-10 ve druhém souboru.

RdL

Ve druhém souboru chybí řádky R z prvního souboru. Tyto řádky by patřily za řádek L druhého souboru. Např. 5,7d3 znamená, že za řádkem 3 ve druhém souboru chybějí řádky 5-7 prvního souboru.

Příklad Delta notace za pomoci linuxových nástrojů diff dvou souborů

Listing 2.1: Man1.java

```
class Man {
    private String name;

    public Man(String name){
        this.name = name;
    }
}
```

Listing 2.2: Man2.java

```
class Man {
    private String name;

    private String surname;
```

```

    public Man(String name){
        this.name = name;
    }

    public Man(String name, String surname){
        this(name);
        this.surname = surname;
    }
}

```

Listing 2.3: Diff Man1 Man2

```

3a4,5
>         private String surname;
>
5a8,12
>     }
>
>         public Man(String name, String surname){
>             this(name);
>             this.surname = surname;

```

Ukázka kódu 2.1 zobrazuje zdrojový kód třídy Man v první verzi. Ukázka 2.2 potom zdrojový kód třídy Man po první naší editaci, kdy jsme do dané třídy přidali nový atribut a nový konstruktor. Diff v delta notaci těchto dvou souborů je ukázán v 2.3. V delta notaci vidíme, že do prvního souboru byly za 3. řádek vloženy řádky 4-5 z druhého souboru - řádek definující nový atribut surname a oddělovací prázdný řádek, dále za 5. řádek byly vloženy řádky 8-12 z druhého souboru definující nový konstruktor a uzavírací závorka.

Delta notace je dle autora výhodná díky snadné rozložitelnosti velkých patchů na více menších patchů a také díky oddělení ve více konkurenčních patchích konfliktních operací od operací nazávislých.

Inverzní diff vidíme v 2.4, kde z souboru 2.2 byly odstraněny řádky 4-5, které by se jinak zařadily za řádek 3, dále byly odstraněny řádky 10-13, které by jinak patřily za řádek 7 souboru Man1.

Listing 2.4: patch *part_b*

```

4,5d3
<         private String surname;
<
8,12d5
<     }
<
<         public Man(String name, String surname){
<             this(name);
<             this.surname = surname;

```

Patch 2.3 bychom mohli rozdělit například na 2.5 a 2.6, protože změny jsou na různých řádcích a jsou zdánlivě nezávislé. Aplikací těchto dvou patchů v pořadí *part_a* a *part_b* vznikne 2.7, kdežto aplikace patchů v pořadí *part_b*, *part_a* dá vzniknout původnímu souboru 2.2

Listing 2.5: patch *part_a*

```
3a4,5
>     private String surname;
>
```

Listing 2.6: patch *part_b*

```
5a8,12
>     }
>
>     public Man(String name, String surname){
>         this(name);
>         this.surname = surname;
>
```

Listing 2.7: aplikace pořadí difů *part_apart_b*

```
class Man {
    private String name;

    private String surname;

}

    public Man(String name, String surname){
        this(name);
        this.surname = surname;
    public Man(String name){
        this.name = name;
    }

}
```

Delta notace umožňuje snadný forward i backward differencing, protože díky kontextové nezávislosti je možné snadno invertovat operace. Kontextová nezávislost znamená, že jakýkoliv patch je možné aplikovat na jakýkoliv model, přičemž očekávatelný výsledek je výstupní soubor nebo chyba. Přidávání řádků do souboru je vždy aplikovatelné. Náhrada a odstranění řádků ze souboru, ve kterém nefigurují naopak vyvolá chybu.

Aplikace patche 2.3 na 2.2 by vypadala 2.8. Atribut byl přidán na správné místo, ale konstruktor byl přidán na místo špatné - přidaná uzavírací závorka neuzavírá předchozí konstruktor, ale celou třídu. Navzdory tomu, že tento Java kód je nevalidní, patch bylo možné aplikovat.

Listing 2.8: patch aplikován na třídu Man2

```

class Man {
    private String name;

    private String surname;

    private String surname;

}

public Man(String name, String surname){
    this(name);
    this.surname = surname;
public Man(String name){
    this.name = name;
}

public Man(String name, String surname){
    this(name);
    this.surname = surname;
}
}

```

Všimněme si, že standardní unixový diff obsahuje redundantní informaci - řádek 2.5 3a4,5 musí obsahovat číslo řádku, za který se má navazující text diffu přidat, ale nemusí obsahovat, na které pozice v druhém souboru budou přidány, tato informace by byla dopočitatelná z aplikace všech předchozích řádků, přičtení počtu řádků přidávaných a odečtení řádků odebíraných. Takovouto změnou standardní delta notace dostaneme další možnou variantu delta notace.

Standardní delta notace obsahuje další redundantní prvky. Například v deltě mazající řádky je redundantní informací seznam mazaných řádků. Jeho odstraněním bychom získali plně operaci aplikovatelnou na jakýkoliv model. Tato změna nevytváří delta notaci - nechováva všechny informace o změně modifikovaných elementů. Pro ilustraci diff na 2.9 je reverzní vůči diffu 2.3. Odstraněním seznamu mazaných řádků a pozice v druhém souboru získáme diff 2.10. Z tohoto vyplývá, že delta notace není minimální, obsahuje redundantní informace. Prostým pozorováním můžeme zjistit, že oproti delta notaci nemůže v nově vzniklé minimální notaci být odvozena inverze 2.3 diffu 2.10, obrácený postup - transformace diffu 2.3 na 2.10 bez znalosti kontextových modelů 2.1 a 2.2 je naopak zachována z delta notace. Nově vzniklá notace je tudíž neidempotentní vůči operaci inverze bez znalosti kontextového vstupního a výstupního modelu.

Listing 2.9: diff modelů 2 a 1

```

4,5d3
<      private String surname;
<
10,13d7

```

```

<      public Man(String name, String surname){
<          this(name);
<          this.surname = surname;
<      }

```

Listing 2.10: diff modelů 2 a 1

```

4,5d
10,13d7

```

2.3.2.1 Vlastnosti operací

V [Cic08] Antonio Cincetti popisuje některé specifické vlastnosti jako je invertovatelnost a rozložitelnost operací. Je nutné říci, že operace zmiňované v literatuře pracují jen se strukturou dat, nikoliv s daty samotnými a jsou kontextově nezávislé - tyto operace jsou tvořeny téměř výlučně konstruktivními a destruktivními operacemi. Cincetti rozděluje zmiňuje, že v minulosti byly operace aplikovatelné na jeden konkrétní model(intensional) a modernější diferenční modely jsou tzv. extensional - je možné je aplikovat na jakýkoliv model, například na paralelní vývojové větve.

V projektu Migdb jsou operace invertovatelné se znalostí původního modelu. Například u operace RemoveParent(childClass) nezískáme parentClass přímo z operace, ale musíme ho dopočítat ze vstupního modelu.

Problémem je, že i po odvození inverze nemusí vést aplikace operace do stejného vstupního modelu. Pokud například na stav 2.10 aplikujeme operaci AddParent(Teacher, UniversityTeacher), získáme cílový stav 2.11. Pokud se chceme vrátit zpět z 2.11 do výchozího stavu, měli bychom aplikovat operaci RemoveParent(UniversityTeacher), nicméně aplikace této operace nepovede do výchozího stavu 2.10, ale do stavu 2.12. Po aplikaci operace RemoveParent bude v třídě UniversityTeacher navíc property class. Tento stav je zapříčiněn vývojem operace AddParent - v původní verzi operace nemohla být použita na jakékoliv třídy s kolizními atributy, ale shledali jsme tuto operaci nepoužitelnou - většinou přidáváme supertyp třídy, pokud je třída potomka speciálním typem třídy rodičovské, což se ale v drtivé většině případů projevuje kolizními atributy. Možným odstraněním tohoto problému by bylo přidání informací o distribuovaných properties do operace RemoveParent, čímž bychom se nicméně odklonili od cílu minimalizovat operace.

2.4 Databázové operace

Databázové operace reprezentují změny proveditelné na úrovni databáze. Mělo by z nich být možné generovat SQL kód jednoduchou Model-to-text transformací zajišťovanou modulem Generator.xtend.

Seznam operací s jejich validačními podmínkami:

Operace I: AddSchema(name)

- Vytvoří nové schéma s zadaným jménem
- Nesmí existovat schéma s zadaným jménem

Operace II: AddSequence(owningSchemaName, name, startValue)

- Vytvoří v cílovém schématu sekvenci s zadanou startovní hodnotou
- Musí existovat schéma, do kterého se vkládá, v něm nesmí existovat sequence s jménem name

Operace III: AddTable(owningSchemaName, name)

- V daném schématu vytvoří tabulku, id sloupec této tabulky a primární klíč odvozený z jména tabulky
- Musí existovat dané schéma, v němž nesmí existovat tabulka s jménem name

Operace IV: AddColumn(owningSchemaName, owningTableName, name, type)

- Vytvoří v daném schématu a tabulce column s zadaným primitivním typem
- Musí existovat dané schéma, tabulka a v dané lokaci nesmí existovat daný sloupec

Operace V: AddPrimaryKey(owningSchemaName, owningTableName, constrainedColumnName, name)

- Vytvoří v daném schématu a tabulce nad constrainedColumn Primární klíč s daným jménem
- Musí existovat dané schéma, daná tabulka, daná column, nesmí existovat constraint s daným jménem

Operace VI: AddForeignKey(owningSchemaName, owningTableName, constrainedColumnName, name, targetTableName)

- Vytvoří v daném schématu a tabulce cizí klíč s daným jménem, který referencuje IdColumn cílové tabulky
- Musí existovat dané schéma, daná tabulka, daná column, nesmí existovat constraint s daným jménem, musí existovat targetTable

Operace VII: AddUnique(owningSchemaName, owningTableName, constrainedColumnNames, name)

- Vytvoří v daném schématu a tabulce unique constraint s daným jménem nad zadanými sloupci
- Musí existovat dané schéma, musí existovat daná tabulka, musí existovat dané constrainované sloupce, nesmí existovat constraint s jménem name

Operace VIII: AddNotNull(owningSchemaName, owningTableName, constrainedColumnName)

- Nastaví v daném schématu a tabulce cílové property hodnotu notNull na true
- Musí existovat dané schéma, daná tabulka, daná column

Operace IX: RemoveNotNull(owningSchemaName, owningTableName, constrainedColumnName)

- Nastaví v daném schématu a tabulce cílové property hodnotu notNull na false
- Musí existovat dané schéma, daná tabulka, daná column

Operace X: RenameTable(owningSchemaName, name, newName)

- Změní cílové tabulce jméno na nové
- Musí existovat dané schéma, daná tabulka, nesmí existovat tabulka s novým jménem

Operace XI: RenameColumnowningSchemaName, owningTableName, name, newName

- Přenastaví v daném schématu a tabulce jméno z name na hodnotu newName
- Musí existovat dané schéma, daná tabulka, daná column

Operace XII: RemoveTableowningSchemaName, name

- Odstraní z daného schematu tabulku s jménem name
- Musí existovat dané schéma, daná tabulka

Operace XIII: RemoveColumnowningSchemaName, owningTableName, name

- Odstraní v daném schématu a tabulce column s jménem name
- Musí existovat dané schéma, daná tabulka, daná column, která nesmí na sobě mít constraint PrimaryKey, ForeignKey nebo Unique

Operace XIV: RemoveConstraintowningSchemaName, owningTableName, name

- Přenastaví v daném schématu a tabulce column s jménem name
- Musí existovat dané schéma, tabulka a column

Operace XV: RemoveSequenceowningSchemaName, name

- Odstraní v daném schématu sequence s jménem name
- Musí existovat dané schéma a daná sequence

Operace XVI: UpdateRowsowningSchemaName, sourceTableName, sourceColumnName, targetTableName, targetColumnName, selectionWhereCondition, safeWhereCondition

- v daném schématu updatuje hodnoty z tabulky sourceTable hodnoty z sourceColumns a nastaví je do taragetColumns tabulky targetTable pro instance splňující selectionWhereCondition, pozn. aby nebyly nullovány hodnoty, pro které nebyly vybrány hodnoty z sourceTable byla přidána safeWhereCondition
- Musí existovat dané schéma, v něm sourceTable, v ní sourceColumn, v dále musí v schématu existovat targetTable, v ní targetColumn, sourceColumn musí mít stejný typ jako targetColumn

Operace XVII: NillRowsowningSchemaName, tableName, columnName, whereCondition

- Nastraví sloupci v daném schematu a tabulce hodnoty null instancím splňující whereCondition

- Musí existovat dané schéma, daná tabulka, daná column

Operace XVIII: InsertRowsowningSchemaName, sourceTableName, sourceColumnName, targetTableName, targetColumnName, whereCondition

- v daném schématu zkopíruje z tabulky sourceTable hodnoty z sloupce sourceColumns instance splňující whereCondition a vloží je do targetColumns tabulky targetTable
- Musí existovat dané schéma, v něm sourceTable, v ní sourceColumn, v dále musí v schématu existovat targetTable, v ní targetColumn, sourceColumn musí mít stejný typ jako targetColumn

Operace XIX: DeleteRowsowningSchemaName, tableName, whereCondition

- Operace smaže instance z daného schématu, dané tableName instance splňující whereCondition
- Musí existovat dané schéma, v něm daná table

2.4.1 Ukázka kódu

V následujícím kódu je ukázáno validační query pro validaci operace AddTable

```
query RDB::ops::AddTable::isValid(structure : RDB::Structure,
  inout errorLog : ErrorLog, operationIndex : Integer) : Boolean {
  var existSchema : Boolean := checkExistSchema(
    self.owningSchemaName,
    structure,
    errorLog,
    operationIndex,
    getEvolutionRdbTransformationId());
  var notExistTable : Boolean := checkNotExistTable(
    self.owningSchemaName,
    self.name,
    structure,
    errorLog,
    operationIndex,
    getEvolutionRdbTransformationId());
  return existSchema and notExistTable;
}

helper checkExistSchema(schemaName : String, structure : Structure,
  inout errorLog : ErrorLog, operationIndex : Integer,
  transformationId : String) : Boolean{
  var existSchema : Boolean := structure.containsSchema(schemaName);
  if(not existSchema)then{
    var errorMessage : String := "Schema " + schemaName + " doesn't exist";
    errorLog.errors += _evolutionError(
```

```

                                operationIndex,
                                errorMessage,
                                transformationId);

    }endif;
    return existSchema;
}

helper checkNotExistTable(owningSchemaName : String, tableName : String,
    structure : Structure, inout errorLog : ErrorLog, operationIndex : Integer,
    transformationId : String) : Boolean{
    if(not structure.containsSchema(owningSchemaName))then{
        return false;
    }endif;
    var notExistTable: Boolean := not
        structure.containsTableInSchema(owningSchemaName, tableName);
    if(not notExistTable)then{
        var errorMessage : String := "Table " + tableName + " exists in schema "
            + owningSchemaName;
        errorLog.errors += _evolutionError(
            operationIndex,
            errorMessage,
            transformationId);

    }endif;
    return notExistTable;
}

```

Je možné zaznamenat, že helpery nejen vrací hodnotu, ale i loggují chyby do errorLogu pro daný index operace a danou transformaci, tudíž není možné volat helper checkExistTable a invertovat navrácenou hodnotu, ale bylo nutné napsat helpery kladné i záporné.

2.5 ORMo (ORM operací)

5 Ačkoliv ORM transformace vstupního aplikačního modelu funguje se všemi inheritance-Typy bylo nutné zjednodušit aplikační model tak, aby byla transformace ORMo implementovatelná, proto jsme v rámci týmu Migdb rozhodli o redukci počtu inheritanceTypů na jeden - nejvhodnější typ je nejspíše joined, který je pravděpodobně nejpoužívanějším.

Vzhledem k nedostatku času nebyla implementována myšlenka .q souborů, které kontrolují některé vlastnosti nejen modelu, ale i konkrétních dat, dále není mapováno omezení LB = 0, které by některé operace stížilo.

Algoritmus ORMo si bere všechna data z aplikačního modelu, čímž je seznam výstupních operací nezávislý na aplikaci databázových operací nad databázovým modelem, ale přebírá veškerou zodpovědnost za údržbu - tj vytvoření, odstranění a údržbu (úpravu) integritních omezení. V následujícím seznamu jsou specifikovány ORMo mapování jednotlivých operací.

Operace I: AddStandardClass(name, isAbstract, inHeritanceType)

- Vytvoří tabulku, id sloupec této tabulky a primární klíč

Operace II: AddProperty(owningClassName, name, typeName, lowerBound, upperBound, isOrdered, isUnique)

Primitivní typ a UB = 1 operace přidá do cílové tabulky sloupec pro primitivní property

Primitivní typ a UB != 1 operace přidá collection tabulku, datový sloupec, referenční sloupec a FK referencující vlastnickou tabulku

Neprimitivní typ a UB = 1 operace přidá do cílové tabulky a referenci na tabulku typu

Neprimitivní typ a UB != 1 operace vytvoří vazební tabulku pro neprimitivní property, vloží do ní referenční sloupce na vlastnickou tabulku a tabulku typu, na které vytvoří cizí klíč

Operace III: RenameEntity(owningClassName, name, newName)

- Operace změní název tabulky na nový, odstraní a vytvoří PK s novým jménem

Operace IV: SetAbstract(name, isAbstract)

isAbstract = true maže data, která náležejí pouze dané třídě

Operace V: RemoveEntity(name)

- operace smaže primární klíč, id property a tabulka odpovídající dané třídě

Operace VI: RenameProperty(owningClassName, name, newName)

primitivní typ a UB = 1 přejmenuje property v vlastnické tabulce

primitivní typ a UB != 1 přejmenuje datový sloupec, FK na vlastníka kolekce a tabulku kolekce

neprimitivní typ a UB = 1 přejmenuje sloupec ve vlastnické tabulce a cizí klíč referující tabulku typu

neprimitivní typ a UB != 1 přejmenuje vazební tabulku s referenčními sloupci na vlastníka a typ asociace + cizí klíče

Operace VII: RemoveProperty(owningClassName, name)

primitivní typ a UB = 1 odstraní sloupec z dané tabulky

primitivní typ a UB != 1 odstraní referenci na vlastnickou tabulku, sloupec z tabulky dané kolekce, datový sloupec a smaže kolekční tabulku

neprimitivní typ a UB = 1 odstraní referenci na tabulku vlastníka a referenční sloupec

neprimitivní typ a UB != 1 odstraní reference na vlastnickou tabulku a tabulku typu, datový sloupec a sloupec typu a smaže vazební tabulku

Operace VIII: SetOrdered(owningClassName, name, isOrdered)

isOrdered = true přidá sloupec ordering, přenastaví data a vytvoří unikátní constraint přes typový, referenční a ordering sloupec

isOrdered = false smaže ordering unique constraint a ordering sloupec

Operace IX: SetUnique(owningClassName, name, isUnique)

isUnique = true vytvoří unikátní constraint přes typový a referenční sloupec

isUnique = false smaže unique constraint

Operace X: AddParent(className, parentClassName)

- Aplikuje obraz operace MergeProperty na všechny kolizní property, přidá cizí klíč na rodičovskou třídu

Operace XI: RemoveParent(className)

- aplikuje obraz operace DistributeProperty na všechny property rodičovské třídy, odstraní cizí klíč, smaže data třídy potomka z tabulky rodiče

Operace XII: ExtractClass(sourceClassName, extractClassName, associationPropertyName, oppositePropertyName, propertyNames)

- vytvoří novou sekvenci, vytvoří novou tabulku a nové sloupce spolu se sloupcem pro opposite referenci na zdrojovou tabulku, aplikuje obraz operací exportProperty pro každou exportovanou property, vytvoří sloupec referencující nově vzniklou tabulku, updatuje mu hodnoty, smaže vygenerovanou sekvenci

Operace XIII: InlineClass(targetClassName, associationPropertyName)

- aplikuje obraz operací exportProperty, smaže association column a Inlinovanou tabulku

Operace XIV: PullUpProperties(childClassName, pulledPropertiesNames)

- aplikuje obraz operace export property do rodičovské třídy pro každou property s name z pulledPropertiesNames

Operace XV: PushDownProperties(childClassName, pushedPropertiesNames)

- aplikuje obraz exportProperty pro každou property s name z p vyjmenované property do třídy potomka a přesune JEN data potomka

Operace XVI: ExportProperty(virtuální operace)

primitivní typ a UB = 1

primitivní typ a UB !=1

neprimitivní typ a UB = 1

neprimitivní typ a UB !=1

Operace XVII: DistributeProperty(virtuální operace)

primitivní typ a UB = 1

primitivní typ a UB !=1

neprimitivní typ a UB = 1

neprimitivní typ a UB !=1

Operace XVIII: MergeProperty(propertyName, targetClassName)

primitivní typ a UB = 1 updatuje column v cílové tabulce smaže column ze zdrojové tabulky

primitivní typ a UB !=1 vloží řádky do tabulky collection, smaže FK z zdrojové collectionTable(případně odstraní UX a ORD constrainty), smaže data, reference column z zdrojové collection table a nakonec i zdrojovou collectionTable

neprimitivní typ a UB = 1 updatuje column v cílové tabulce smaže column ze zdrojové tabulky

neprimitivní typ a UB !=1 vloží řádky do tabulky collection, smaže FK z zdrojové collectionTable(případně odstraní UX a ORD constrainty), smaže data, reference column z zdrojové collection table a nakonec i zdrojovou collectionTable

2.5.1 Diff elementy

Kvůli nutnosti rozpoznávat operace vznikly v aplikačním modelu nové elementy. Kořenovým elementem diff modelu je Diff element. Tento element obsahuje kolekce elementů classpairs typů ClassPair, propertyPairs typu PropertyPair, a dále pak addedClasses a removedClasses typu DiffClass a addedProperties a removedProperties typu DiffProperty. Element ClassPair shlukuje zpárované zdrojové (source) a obrazové (reflection) třídy, dále pak referenci owningDiff na Diff element, v kterém jsou obsaženy a která je důležitá pro implementaci algoritmu a v neposlední řadě underlyingPairs - shodné páry Properties typu EqualPropertyPair, které jsou detekované danou operací. Podobně jako operace jsou i páry rozděleny do několika skupin.

Konstruktivní a destruktivní operace nemají svůj obraz v diff metamodelu jako ClassPair ani PropertyPair, protože tyto operace nemapují element ze vstupního modelu na element z výstupního modelu. Konstruktivní operace by jinak mapovaly prázdný vstup na element a destruktivní obráceně element na prázdný výstup.

Oproti jednodušším konstruktivním a destruktivním operacím jsou operace expanzivní, konstruktivní a modifikační v Diff modelu zobrazeny jako ExpansiveClassPair, ReductiveClassPair a ModifyingClassPair, mapují element vstupního modelu na element cílového modelu. Aby bylo možné rozpoznat specifický pár závislý na jiném páru, byla přidána třída ReplacingClassPair - nahrazující pár, který se používá jako pivot pro hledání expansivních, reduktivních a modifikačních párů. Od elementu ReplacingClassPair dědí elementy EqualClassPair - třída, která si uchovála jméno z původního modelu a element ReplacingClassPair - reprezentující třídu, která si neuchovala jméno, ale má změněný název. Podmínky získávání konkrétních typů párů a jejich pořadí specifikuje konkrétní rozpoznávací algoritmus.

Projevem subtraktivních a aditivních operací jsou elementy DiffClass a DiffProperty, které zaobalují třídy a property tak, aby bylo možné referencovat na jiný objekt než element Structure .

2.6 Rozpoznávání operací

Algoritmem pro rozpoznávání operací nazveme každý algoritmus, který nám pro každý vstupní model A a cílový model B najde uspořádaný seznam operací, jejichž postupná aplikace transformuje model A do modelu B. Tento algoritmus nemusí být deterministický.

Jedním z zajímavých faktů je poznatek, že seznam operací nemusí být jednoznačný a to i u jednoduchých změn. Pokud aplikujeme sekvenci operací Inline A, B + Rename B \rightarrow C na model X dostaneme stejný výstup jako aplikací operací Inline B, A + Rename A \rightarrow C, ještě zajímavějším poznatkem je, že nejsme schopni rozeznat rozdíl mezi aplikací sekvence operací Rename A, C + Inline C, B.

Samostatným tématem je pořadí operací a jeho permutace. Je zřejmé, že pořadí v seznamu operací operujících nad jinými elementy bude možné libovolně prohazovat. Také je samozřejmé, že seznam subtraktivních operací je také možné libovolně zpermutovat. Stejně tak seznam aditivních operací. Obecný princip seřazení kolekce operací není znám.

2.7 Obecné principy model matching

Nejtriviálnější implementovatelný algoritmus by mohl smazat zdrojový model pomocí destruktivních operací a následně vytvořit výsledný model pomocí operací konstruktivních, případně upravit atributy jednotlivých elementů pomocí operací modifikačních. Argumentem proti použití takového algoritmu je smazání jakýchkoliv dat, které v původní databázi byla. Takovýto algoritmus tudíž nemigruje žádná data, ale nahrazuje funkci ORM mapování integrované do většiny současných IDE. Proto se jím v této práci nezaobírám.

Jak je diskutováno v 19 a [DSK14] existuje několik požadavků na algoritmus řešící problém model matching. Tyto požadavky zahrnují přesnost, vysokou míru abstrakce na které je porovnávání provedeno, nezávislost na konkrétních nástrojích, doménách a jazycích (přelož independence from particular tools), použitelnost (efficiency) a minimální nutnost adaptace algoritmus pro daný problém. Tyto požadavky jdou proti sobě a je nutné preferovat některé na úkor jiných, proto není možné označit za nejlepší, ale je nutné vybrat si správný algoritmus v závislosti na řešeném problému.

V [DSK14] byly popsány algoritmy pro mapování shodných entit modelů (ModelMatching) a algoritmy pro získávání rozdílů modelů (Model Diff). Principem těchto modelů je párování elementů vstupního modelu s elementy z modelu cílového. Existují 4 obecné skupiny dělení matching algoritmů. 1 párování podle statického identifikátoru, 2. signature based matching, 3. similarity based matching a custom language specific matching.

Párování podle statického identifikátoru páruje elementy podle perzistentního identifikátoru, který je přiřazen každé entitě v době jejího vzniku, je neměnný a unikátní. Nejzákladnějším principem model matchingu je tedy párování entit na základě shodnosti jejich identifikátorů. Tento princip má výhody jednoduchosti implementace a rychlosti. Tento algoritmus není použitelný pro modely vytvořené nezávisle jeden na druhém či u technologií nepodporujících maintenance unikátních identifikátorů.

Algoritmus signature based matching byl navržen kvůli limitaci párování podle statického identifikátoru, tento algoritmus je založen na dynamickém vypočtení nestatické signatury jednotlivých features pomocí uživatelem definovaných funkcí specifikovaných pomocí nějakého dotazovacího jazyka. Tento princip tedy může být použit pro modely vzniklé nezávisle na

sobě. Nevýhodou je potom nutnost specifikovat query, které dopočítají signaturu.

Algoritmus Similarity based matching používá podobně jako signature based matching podobnost features jednotlivých elementů, kterou agreguje do skalární hodnoty. Tento princip se řadí mezi podtyp attribute graph matchingu. Každá feature modelu může mít jinou váhu pro porovnávání, například u podobnosti tříd má jméno vyšší důležitost nežli abstractnost dané třídy. Tento algoritmus musí být typicky doplněn o konfiguraci vah jednotlivých features elementů, kterou většinou píše vývojář. Zástupcem tohoto principu je framework EMF Compare, který je doplněn o defaultní konfiguraci vah. Výhodami je větší přesnost, nevýhodou je potom TRIAL ERROR metoda získávání vhodné konfigurace vah.

Algoritmy v kategorii Custom language specific matching jsou vytvořené přímo k využití daného modelovacího jazyka. Hlavní výhodou je, že algoritmus na dané doméně může začlenit do metody similarity based matchingu sémantické detaily, což vede k přesnějším výsledkům a redukuje prohledávaný stavový prostor. Jako příklad je uváděn jazyk UMLDiff, který při porovnávání dvou UML modelů může využít faktu, že dvě třídy nebo dva datové typy stejného jména tvoří po všech praktických stránkách pár(match). Nicméně výhoda začlenění sémantických detailů konkrétní domény je vykoupěno vysokou cenou - všechny ostatní kategorie algoritmů potřebují minimální neb téměř žádné úpravy od vývojáře, pro tuto kategorii vývojáře musí napsat celý matchovací algoritmus sám.

2.7.1 Graph matching

Problém model matching je podproblémem generičtějšího tasku graph matching, který studuje <http://www.sc.ehu.es/acwbecae/ikerkuntza/these/Ch2.pdf> a rozděluje a popisuje algoritmy pro graph matching. Problém je definován na obecné struktuře Graf, což je uspořádaná dvojice $G = (V, E)$, kde G je množina uzlů a E je množina hran grafu, přičemž $E \subset V \times V$. Grafy mohou být orientované či neorientované, mohou mít vícenásobné hrany.

Každý graf může přidávat informace do své struktury pomocí labelu (popisku nebo číslu) do hran a vrcholů, pokud je nutné přidat více informací, je možné přidat do hran a/nebo vrcholů atributy, potom hovoříme o vertex-attributed grafech a edge attributed grafech, případně attributed grafech. V některé literatuře jsou attributed grafy označovány jako labeled grafy. Graph matching je aplikován v mnoho oborů jako je počítačové vidění, analýza scény(scene analysis), chemie a molekulární biologie. V těchto oborech musí být vzorce nalezeny v daných datech.

Problém graph matchingu dvou grafů G_P (grafu patternu) a G_M (grafu modelu), přičemž se dělí podle převzatého obrázku 2.14 na matching nalezení přesné shody vzorku v hledaném grafu či matching hledání podobnosti grafu vzorku v hledaném grafu.

Matching hledání přesné shody je definován následně: Mějme grafy $G_P = (V_P, E_P)$ a $G_M = (V_M, E_M)$, přičemž $\|V_M\| = \|V_P\|$, úkolem je potom najít takové prosté zobrazení $f : V_D \rightarrow V_M$, takové, že $(u, v) \in E_P$ iff $(f(u), f(v)) \in E_M$. Pokud takové mapování existuje, nazveme ho exact graph matchingem(matching přesné shody).

Termín Inexact matching aplikovaný na některé problémy týkající se shodnosti grafů vyjadřuje/znamená, že není možné nalézt izomorfismus mezi dvěma grafy, aby byly matched (shodné?). To je stav, kdy oba grafy mají jiný počet vrcholů. Takže v těchto případech není očekávatelné hledání izomorfismu dvou grafů, ale v hledání největší možné shody mezi nimi (best matching). Toto vede k třídě problémů známé jako inexact graph matching. V tako-

vém případě hledáme nebijektivní korespondenci (přiřazení) mezi pattern grafem a model grafem. V následujícím textu předpokládejme $\|V_P\| < \|V_M\|$. Inexact matching je používán v oborech kartografie, rozpoznávání znaků a medicíně. Nejlepší korespondence graph matching problému je definována jako optimum nějaké objective function, která měří podobnost mezi matchovanými uzly a hranami. Tato funkce je nazvána fitness funkcí, případně energy function.

Formálně je tedy inexact matching definován takto: mějme dva grafy, G_M a G_P přičemž $\|V_M\| < \|V_D\|$ a cílem je nalezení mapování $f' : V_D \rightarrow V_M$ e $(u, v) \in E_P$ iff $(f(u), f(v)) \in E_M$.

Podtypem těchto úloh jsou problémy subgraph matching a subgraph izomorfizmu.

Složitost uváděných problémů uvádí autor u Exact graph matchingu jako P až NP kompletní, přičemž že u problémů této kategorie nebyla dokázána nejvyšší složitost NP complete. Pro složitost problémů u subgraph isomorphismu byla dokázáno, že patří do třídy NP complete. Pro složitost nepřesného graph matchingu bylo dokázáno, že patří do třídy NP-complete.

2.8 Vytvořený algoritmus rozpoznávání operací

2.8.1 Návrh ze studia článků

Vzhledem k obecné použitelnosti algoritmů pro graph matching nebyly tyto algoritmy shledány za vhodné k použití pro problém hledání sady aplikačních operací. První 3 popsané algoritmy model matchingu (1 párování podle statického identifikátoru, 2. signature based matching, 3. similarity based matching) nejsou taktéž vhodné k použití z důvodu, že k rozpoznání popsaných expanzivních a reduktivních operací je nutné rozpoznat 2 třídy, které se mapují na jednu třídu pro reduktivní operace a naopak jednu operaci, která se mapuje na 2 třídy. Problém rozpoznávání operací je tudíž nadskupinou problému model matchingu, protože matching páruje 1 ku jedné, ale na algoritmus řešící rozpoznávání operací musí řešit matching M entit ku N entitám.

Zmiňované algoritmy mě inspirovaly k vytvoření Custom language specific matching algoritmu pro tento problém, který si z zmiňovaných algoritmů bere hlavně poznámku u UML-Diffu - ze všech praktických důvodů považujeme třídy se stejným jménem jako matchující.

2.8.2 Implementace

Vzniklo několik implementací párovacích algoritmů. První a nejjednodušší používá párování tříd podle jména, následné rozdíly řeší rozpoznáním konstruktivních a destruktivních, případně některých operací modifikačních, ať už tyto operace pracovali s třídami nebo s property.

Složitější implementace algoritmu bylo páruje stejně jako jednodušší v první fázi shodné elementy - modely se mění, ale některé třídy jsou zachovány. Shodné elementy potom tvoří jakési pilíře pro operace konstruktivní a destruktivní, které se vážou na rozpoznání páry. Závislost rozpoznání

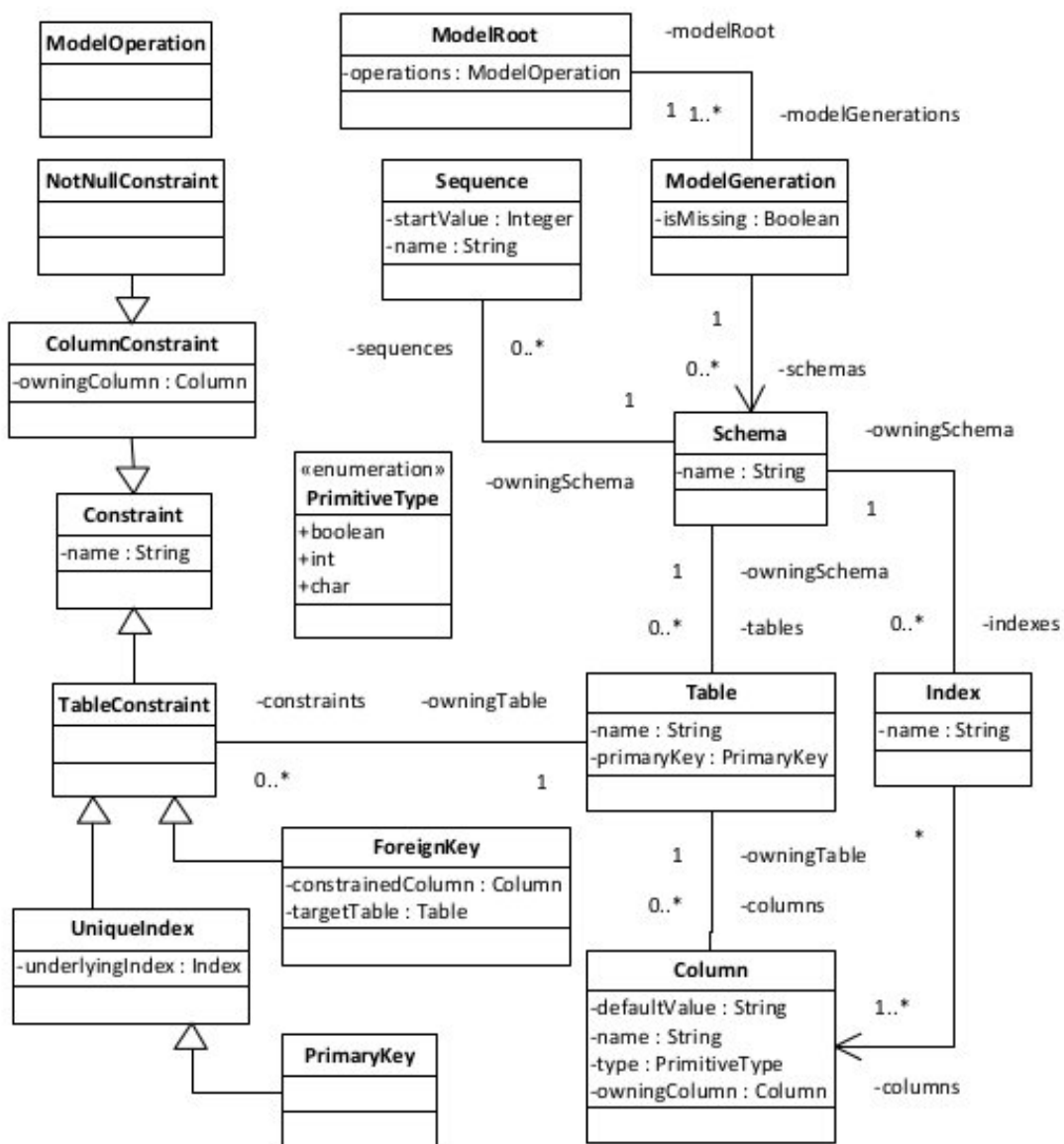
Algoritmus rozpoznávání operací byl napsán se snahou o zachování co největšího množství dat. Ačkoliv triviální algoritmus pro přechod z modelu A k modelu B by mohl pomocí subtraktivních operací zničit model A a následně složit model B pomocí aditivních operací

je zřejmé, že tento algoritmus neuchová žádná data. Proto byly zavedeny Konstruktivní a destruktivní operace.

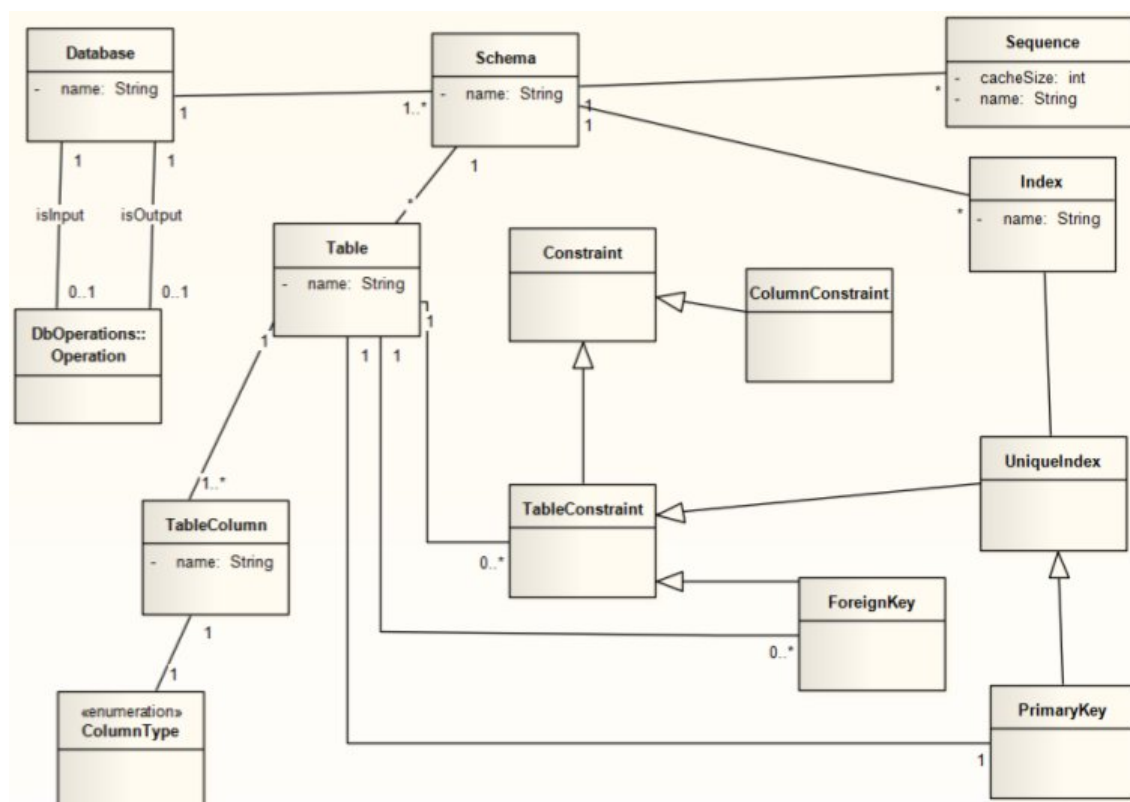
2.9 alternativní algoritmus

V ranné fázi byl napsán prototyp jiného rozpoznávacího algoritmu, který se snaží minimalizovat vzdálenost současného modelu od modelu cílového pomocí rozpoznávacích operací a aplikace operací. Výhodou tohoto přístupu je nalezení více alternativních cest, nevýhodou je potom velikost stavového prostoru. Algoritmus prochází těmito fázemi:

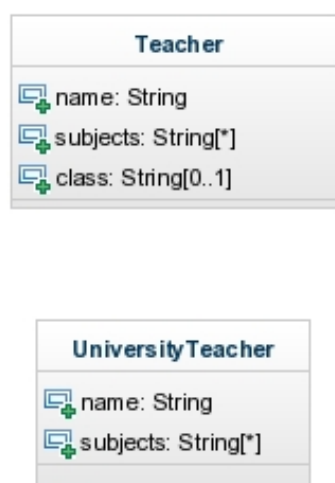
Spočítání vzdálenosti vstupního modelu od modelu cílového Nastavení nalezeného maxima na nula Pro každou operaci O zjištění, jestli má operace vhodné kandidáty na parametr nalezení nejvhodnějších parametrů operace



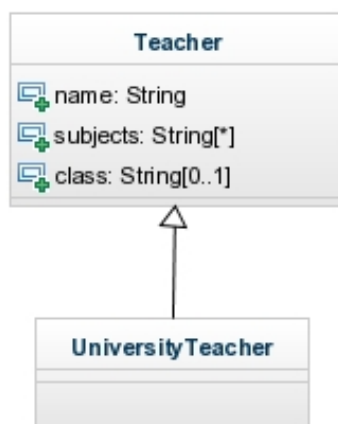
Obrázek 2.8: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Tar12]



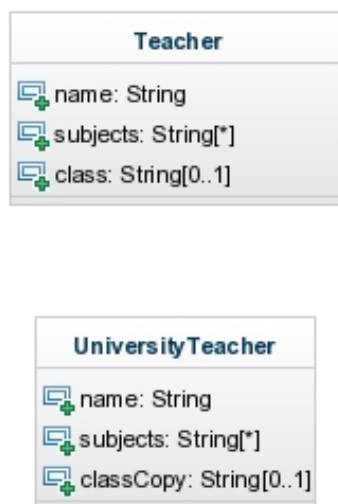
Obrázek 2.9: Struktura databázového metamodelu v průběhu vývoje, obrázek převzat z [Luk11]



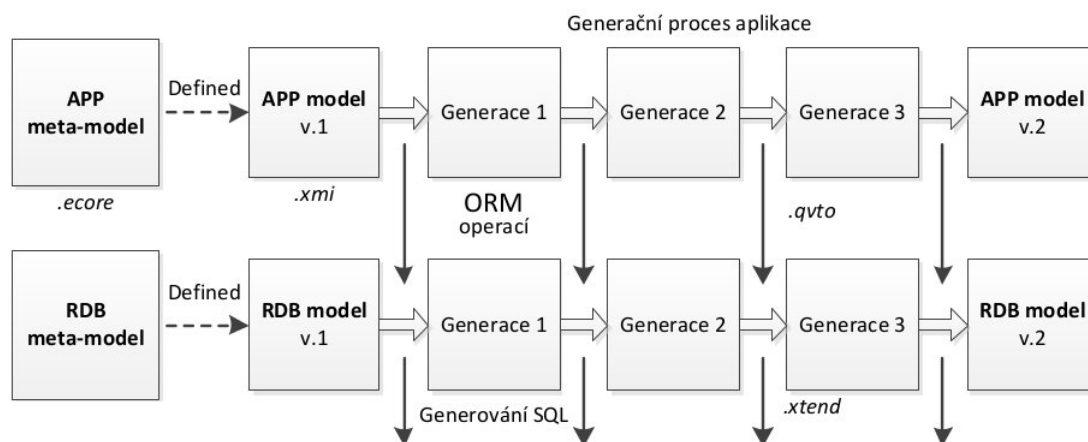
Obrázek 2.10: AddParent(Teacher, UniversityTeacher)



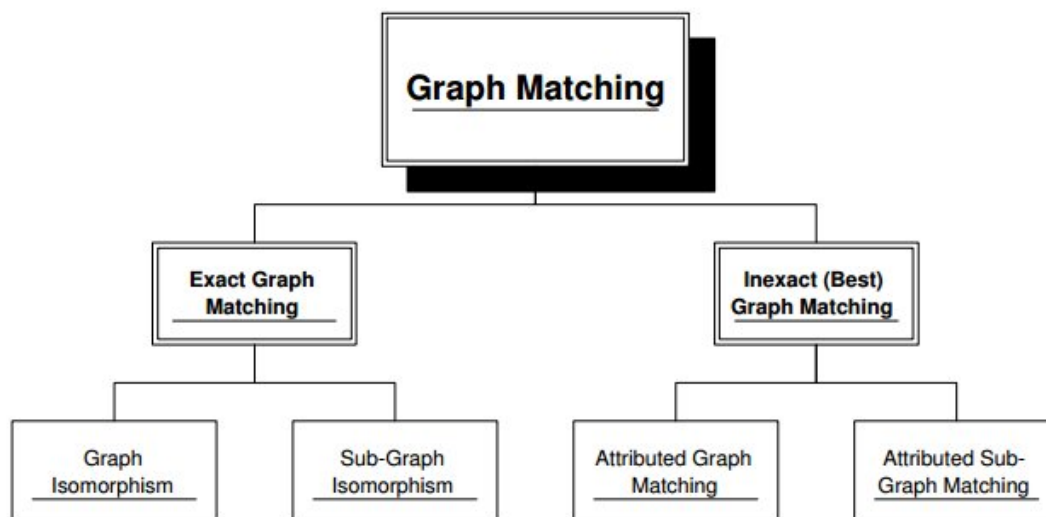
Obrázek 2.11: Výsledek po operaci AddParent



Obrázek 2.12: Výsledný stav po aplikaci RemoveParent



Obrázek 2.13: Seznam diff elementů



Obrázek 2.14: Typy graph matchingu

Kapitola 3

Popis problému, specifikace cíle

Tato diplomová práce si klade za cíl dokončit vývoj na projektu Migdb. Tj doimplementovat a otestovat ORM transformace vzniklé v předešlých fázích projektu, upravit a otestovat generátor SQL, případně upravit aplikační a databázový metamodel.

Dalším cílem, který jsem si před vypracováním diplomové práce stanovil bylo vytvoření a zdokumentování algoritmu generující z dvou vstupních modelů sekvenci operací, jejichž aplikací se model zdrojový transformuje na model koncový.

Kapitola 4

Testování projektu Migdb

V průběhu vývoje byl za účelem ověření správné funkcionality vytvořen projekt Migdb.testing.run, který spouští jednotlivé testy komponent aplikace.

V rámci projektu byly vytvořeny testy komponent Workflow obsažené v packagi migdb.testing.components.run, testy aplikační evoluce inkludované do Workflow test_app_atomic.mwe2 v packagi migdb.testing.app.atomic.run, testy databázové evoluce obsažené workflow test_rdb_atomic.mwe2 v packagi migdb.testing.rdb.atomic.run, testy validátorů aplikačního a databázového modelu obsažené v packagi migdb.testing.validators.run, test ORM transformace struktury aplikace na DB schema + ORMo transformace obsažené v workflow migdb.testing.orm.run, testy generování SQL schématu obsažené v packagi migdb.testing.generators.run, integrované testy celého frameworku migdb obsažené v packagi migdb.testing.migdb_executer a posledními testy jsou testy algoritmů rozpoznávajících operace obsažené v packagi migdb.testing.app.oracle.run.

Testy komponent testují správnou funkcionalitu komponenty Comparator, která musí správně porovnat očekávaný a reálný výstupní soubor xmi ostatních testů. Tyto testy jsou rozděleny do více workflow, protože některé musí být neúspěšné, jak už napovídá klíčové slovo fail v jejich názvu. Dalšími komponentami vzniklými v rámci projektu Migdb byly QVTO-Executor, TestWorkflow, DirectoryCleaner a TestComponent. Komponenta TestComponent vznikla, aby bylo přehlednější a efektivnější psát QVT testy Migdb, je složena z několika dalších načítacích, ukládacích a porovnávacích komponent a zkracuje zápis testů ve workflow asi 10 krát. Nebylo jasné, jak vytvořit testy na správnou funkcionalitu ostatních komponent - správné jejich zapojení do jiných testů bylo pro nás dostatečným testem.

Ostatní testy se řadí do xmi kategorie testů, tj testů, které mají jako vstupní model xmi soubor. Po spuštění xmi testů se v projektu migdb.testing.run vygeneruje složka output-tests, která obsahuje výstupní data jednotlivých testů. Výstupní data mohou obsahovat výstupní xmi soubory porovnávané s očekávaným xmi souborem nebo SQL souborem, který je možné aplikovat na databázi Postgresql.

Speciálním testem je test_code_generator.mwe2, tento test vygeneruje pro zadaný vstupní aplikační model a sadu aplikačních operací výstupní rdb model, sadu databázových operací, výstupní SQL generující strukturu databáze a SQL reprezentující výstupní databázové operace. Kromě těchto vygenerovaných dat je možné najít ke každému testu v adresáři test_data reálná data, s kterými může být test spuštěn. Reálná data jsou vytvořena jen pro testy komplexnějších operací(007 - 010), které jen nemění strukturu databáze, ale i manipulují s daty. Postup testování tohoto testu je - spuštění workflow vedoucí k vygenerování SQL db

schematu a SQL db operací. Spuštění SQL vytvářející strukturu databáze, spuštění SQL s daty uloženými v souboru data.sql z přidružené složky v adresáři test_data, aplikace vygenerovaného souboru transformačních SQL, zobrazení výstupních dat za pomoci souboru check_selects.sql ze složky s daty. Bohužel nebyl nalezen lepší způsob otestování než manuální spuštění a vizuální porovnání selectů s očekávaným výstupem po aplikaci dané operace na data.sql.

Kapitola 5

Ukázka zdrojového kódu práce

Ukázka helperu mergeProperty

```
helper AssociationProperty::ormMergeProperty(appStructure : APP::Structure,
      targetClassName : String, operations : RDB::Operations) : RDB::Operations {
  var defaultSchemaName : String := getDefaultSchemaName();
  var sourceClassName : String := self.owningGeneralClass.name;
  var typeClassName : String := self.type.name;
  var sourceTableName : String := sourceClassName.translate();
  var targetTableName : String := targetClassName.translate();
  var typeTableName : String := typeClassName.translate();
  var mergedColumnName : String := self.name.translate();
  var idSourceTableName : String := getDbIdColumnName(sourceTableName);
  var idTargetTableName : String := getDbIdColumnName(targetTableName);
  var idTypeTableName : String := getDbIdColumnName(typeTableName);

  if(self.upperBound = 1)then{
    var equalWhereCondition : String := getEqualityWhereCondition(
      sourceTableName,
      idSourceTableName,
      targetTableName,
      idTargetTableName);

    var safeUpdateCondition : String := getSafeCondition(
      defaultSchemaName,
      idTargetTableName,
      sourceTableName,
      idSourceTableName);

    var updateRows : RDB::ops::ModelOperation := _updateRows(
      defaultSchemaName,
      sourceTableName,
      mergedColumnName,
      targetTableName,
      mergedColumnName,
      equalWhereCondition,
```

```

safeUpdateCondition);
addOperation(updateRows, operations);
var fkName : String := getFkRefencingOppositeName(
    self.name,
    sourceClassName,
    typeClassName);
var removeFkConstr : RDB::ops::ModelOperation := _removeConstraint(
    defaultSchemaName,
    sourceTableName,
    fkName);

addOperation(removeFkConstr, operations);
var removeMergedColumn : RDB::ops::ModelOperation := _removeColumn(
    defaultSchemaName,
    sourceTableName,
    mergedColumnName);

addOperation(removeMergedColumn, operations);
//M x N association
}else{
    var sourceAssociationTableName : String := getAssociationTableName(
        self.name,
        sourceClassName);
    var targetAssociationTableName : String := getAssociationTableName(
        self.name,
        targetClassName);

    var sourceColumnNames : OrderedSet(String) := OrderedSet{
        idSourceTableName,
        idTypeTableName};
    var destinationColumnNames : OrderedSet(String) := OrderedSet{
        idTargetTableName,
        idTypeTableName};

    if(self.isOrdered)then{
        sourceColumnNames += getDbOrderingColumnName();
        destinationColumnNames += getDbOrderingColumnName();
    }endif;
    var transferData : RDB::ops::ModelOperation := _insertRows(
        defaultSchemaName,
        sourceAssociationTableName,
        sourceColumnNames,
        targetAssociationTableName,
        destinationColumnNames);

    addOperation(transferData, operations);
    if(self.isOrdered)then{
        var ordName : String := getUXOrderingName(sourceAssociationTableName);
        var removeOrdConst : RDB::ops::ModelOperation :=
            _removeConstraint(
                defaultSchemaName,

```



```

sourceAssociationTableName,
ordName);

addOperation(removeOrdConst, operations);
var removeOrdCol : RDB::ops::ModelOperation :=
    _removeColumn(
        defaultSchemaName,
        sourceAssociationTableName,
        getDbOrderingColumnName());

addOperation(removeOrdCol, operations);
}endif;
if(self.isUnique)then{
    var uxName : String := getUXName(sourceClassName, self.name);
    var removeUxConst : RDB::ops::ModelOperation :=
        _removeConstraint(
            defaultSchemaName,
            sourceAssociationTableName,
            uxName);

    addOperation(removeUxConst, operations);
}endif;
var fkSourceName : String :=
    getFKAssociationTableRefName(
        sourceAssociationTableName,
        sourceTableName);
var removeFkSourceConst : RDB::ops::ModelOperation :=
    _removeConstraint(
        defaultSchemaName,
        sourceAssociationTableName,
        fkSourceName);

addOperation(removeFkSourceConst, operations);
var fkTypeName : String :=
    getFKAssociationTableRefName(
        sourceAssociationTableName,
        typeTableName);
var removeFkTypeConst : RDB::ops::ModelOperation :=
    _removeConstraint(
        defaultSchemaName,
        sourceAssociationTableName,
        fkTypeName);

addOperation(removeFkTypeConst, operations);
var removeIdSourceRefCol : RDB::ops::ModelOperation :=
    _removeColumn(
        defaultSchemaName,
        sourceAssociationTableName,
        idSourceTableName);


addOperation(removeIdSourceRefCol, operations);
var removeIdTypeRefCol : RDB::ops::ModelOperation :=

```

```
                                _removeColumn(  
                                defaultSchemaName,  
                                sourceAssociationTableName,  
                                idTypeTableName);  
addOperation(removeIdTypeRefCol, operations);  
var removeSourceAssociationTable : RDB::ops::ModelOperation :=  
                                _removeTable(  
                                defaultSchemaName,  
                                sourceAssociationTableName);  
    addOperation(removeSourceAssociationTable, operations);  
}endif;  
return operations;  
}
```

Kapitola 6

Obsah příloženého CD

	index.html	- výchozí stránka projektu - z ní relativní html odkazy na dokumentaci, zdrojové texty a exe soubor
	readme.txt	- popis, co ve kterém adresáři je a jaký je účel jednotlivých souborů, postup spuštění
	install.txt	- postup instalace programu
	install (.bat)	- instalační dávka
	text/	- adresář obsahující vlastní text DP
	DP.pdf	- text DP v PDF/PS formátu (včetně obrázků)
	exe/	- adresář s přeloženým programem a exotickými .dll
	xxx.exe	- přeložený program
	data/	- data související s diplomovou prací
	...	
	src/	- zdrojové texty programu + exotické knihovny
	...	
	html/	- dokumentace v html včetně výstupu programu Doxygen (javadoc,...)
	...	- soubory dokumentace (html + obrázky)
	abstract	
	index.html	- krátký abstrakt
	...	- obrázky ke krátkému abstraktu (aby byly všechny potřebné v tomto adresáři)
	RabstrCZ	
	index.html	- rozšířený abstrakt v češtině
	...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)
	RabstrAJ	
	index.html	- rozšířený abstrakt v angličtině
	...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)

Obrázek 6.1: Seznam příloženého CD

Kapitola 7

Závěr

Po několikaletém teoretickém a praktickém studiu změn v aplikačním modelu a jejich projevu v modelu aplikačním se nám podařilo definovat ucelenou množinu operací, pomocí nichž je možné měnit aplikační model a definovat jejich mapování na operace databázové úrovně.

Nejjednodušším a teoreticky nejzajímavějším tématem našeho projektu je aplikační model, kterému byly věnovány celkově 2 bakalářské a 2 diplomové práce BP - [Tar12] a [Luk11], DP [Tar14], a [Maz14].

Implementačně nejnáročnějším a zároveň nejvíce teoreticky vyčerpaným tématem se stala samotná transformace aplikačních operací na operace databázové, čehož jsem využil a k implementaci, otestování a dospecifikaci tohoto mapování jsem přidal téma čerstvé a velmi málo prozkoumané. Tímto atraktivním tématem vzhledem k verzování modelů je automatické rozpoznávání operací provedených nad aplikačním modelem.

Mrzí mně, že jsem se nemohl více věnovat tématu rozpoznávání operací nad aplikačním modelem, takže jsem se nedostal k tématům jako je sémantické rozpoznávání operací vycházející pravděpodobně z ideí sémantického webu a implementací Resource Description Framework (RDF) a Ontology Web Language (OWL). Implementace sémantického rozpoznávání by mohla v budoucnu odhadnout změny nejen na základě syntaxe (struktury) databáze, ale i na základě významu názvů jednotlivých entit v databázi. Předpokládám, že potom by bylo možné detekovat mnohem jednoznačněji přejmenování entit - pokud by pomocí nějaké ontologie či podobného nástroje bylo jasné specifikováno, že zvíře je nadtypem býložravce a tento je nadtypem entity zebra, potom pokud v původním modelu existuje třída Zebra, která má jako rodičovskou třídu nastavenou třídu Býložravec a v výsledném modelu existuje třída Zebra s nadtřídou Zvíře a neexistuje třída Býložravec, potom by algoritmus měl snadněji detekovat přejmenování třídy Býložravec na třídu Zvíře i přes velkou strukturální podobnost s jinou třídou.

Je otázkou, jak by sémantičtější rozpoznávání operací bylo prospěšné v kontextu stále větších informačních systémech, kdy je občas těžké nazvat smysluplně entity aplikačního modelu, natož sémantiku jejich relace vůči jiným entitám.

V průběhu psaní této diplomové práce jsem si uvědomil, proč pro nás bylo občas obtížné definovat správné mapování aplikačních operací na operace databázové. Čím blíže bylo zaměření daného participanta bližší aplikačnímu modelu, tím více si tento participant přibližoval databázový model aplikačnímu a vznikaly tak entity jako jsou HasNoInstance s názvem rodičovské tabulky bez jakékoliv relace rodičovství existující v databázovém modelu. Druhou

chybou, kterou jsme dělali při zaměření se na aplikační metamodel bylo modelování aplikačních operací, které měly přespříliš složité mapování na databázové operace či někdy až nesmyslný vliv na data v modelu - příkladem bylo modelování operace SetOpposite, která v aplikačním modelu spojí dvě existující property tříd a až při implementaci mapování na databázové operace bylo zjištěno, že tato operace pozbývá smyslu. Tato odlehlost členů Migdb od databázového modelu vytvořila nemalé koncepční problémy, jejichž řešení bylo nutné s nemalým úsilím vymyslet a implementovat ve velmi pozdní fázi projektu. Pokud bych psal Migdb znovu od začátku, zaměřil bych se více na kooperaci jednotlivých částí, aby bylo pro mně psaní transformace ORMů snazší. Tato transformace je esenciální pro chod projektu a proto je její správná implementace alfa omegou na úspěch projektu.

Samotná definice operací aplikovatelných na databázový model není pevná, což vzhledem k malému vzorku sbíraných požadavků vede k nejednoznačným ORMů mapováním. Proto v navazujících pracích by bylo dobré udělat operační výzkum a získat větší množství požadavků na tyto operace.

Věcí, kterou bych udělal znovu jinak při psaní Migdb od začátku by bylo využití jiného jazyka než je QVT. Tento mapovací jazyk se našim potřebám hodil málo, proto jsme časem přestali používat jeho základní koncept - mapování a nahradili ho Java-like programovacím stylem queries a helperů. Naráželi jsme na stále větší problémy a QVT nám nepřinášelo moc užitku, nýbrž některé nevýhody. Jedna z těchto nevýhod je například automatické ukládání jakýchkoliv pomocných entit do výstupních modelů, které bylo nutné vyřešit (za účelem správného otestování) speciální transformací kopírující jen ty části výstupního modelu, které byly opravdovým výstupem, nikoliv meziproduktem. Tato transformace vzhledem ke svojí povaze samozřejmě zpomaluje exekuci Migdb Workflow.

Poslední otázkou, na kterou jsem neměl moc času hledat odpověď je portabilita projektu Migdb funkčního nad relační databází Postgresql na jiné relační databáze. Předpokládám, že by nebylo složité změnit generátor kódu pro jiné relační databáze - algoritmus vygenerování SQL kódu je poměrně přímočarý a pro Postgresql nebyl dlouhý. Struktura jiných relačních databází není vždy stejná, ale většinou se shoduje, takže ani modifikace databázového metamodelu by neměla být obtížná. Moje minimální zkoumání tohoto tématu odhalilo, že námi používaná databáze Postgresql má maximální délku 64 znaků, databáze Oracle má maximální délku identifikátoru 30 znaků a databáze Microsoft SQL server má maximum stanovené na 128 znaků. Z tohoto vyplývá, že převod Migdb na databázi Microsoft SQL server by nebyl tolik problematický z tohoto pohledu. 30 znaků pro Oracle nevypadá jako problematické - vývojáři mají málokdy třídy ukládané do databáze s jmény delšími než 30 znaků, problém nastává při zahrnutí service pro získávání názvů databázových entit k entitám z aplikačního modelu. Získání názvu cizího klíče, kolekce a asociační tabulky spojuje název atributu a tabulky s nějakým prefixem a odděluje tyto položky jména podtržítky.

Tudíž skutečné omezení délky názvu třídy může být základních 30 oslabeno o 3 (prefix FK či kolekce), oslabeno o 3 (podtržítka oddělující jednotlivé tři části jména - prefix a dvě tabulky) děleno 2 (dvě části). Aplikací těchto operací dostaneme horní hranici 12 znaků, která vypadá jako dostatečná, ačkoliv ne tolik komfortní. V této hranici jsme nicméně nezapočítaly Camel-podtržítkovou konverzi velkých písmen na malá předražená podtržítka. Předpokládejme, že každý identifikátor nebude mít víc jak 3 slova, tudíž musíme odečíst další 3 znaky. 9 nemusí být vždy dostatečná hranice vzhledem k velikosti nynějších systémů a nezapočítáváme do toho fakt, že třídy v aplikačním modelu mohou(a často jsou) prefixovány nějakým workspacem či balíčkem. Tudíž délka 9 nemusí být maximální horní hranice

jmen našich tříd a property. Je tedy zřejmé, že potom bude muset uživatel projektu Migdb nad databází Oracle používat krátké a naprosto neintuitivní názvy entit typu Vec102 či je nutné vymyslet jiný způsob překládání jmen do databázového modelu, který bude automatizovaný, jednoznačný, nezávislý na ostatních entitách v modelu a nejlépe pro člověka snadno získatelný bez pomoci nějakého překladače.

Není možné generovat zkrátit názvy entit - vedlo by to u kolizních názvů pomocí indexace, protože bez dalších pravidel jako například očiv případě potřeby odstranit Constraint není možné zjistit, ke které entitě

7.1 Další poznámky

7.1.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Kapitola 8

Seznam použitých zkratek

IDE Integrated Development Environment

ORM Object-relational mapping

EMF Eclipse modeling framework

SŘBD Systém řízení báze dat

IO Integritní omezení

⋮

Kapitola 9

UML diagramy

Tato příloha není povinná a zřejmě se neobjeví v každé práci. Máte-li ale větší množství podobných diagramů popisujících systém, není nutné všechny umísťovat do hlavního textu, zvláště pokud by to snižovalo jeho čitelnost.

Kapitola 10

Instalační a uživatelská příručka

Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Kapitola 11

Obsah příloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat příložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce.



Obrázek 11.1: Seznam příloženého CD — příklad

Na GNU/Linuxu si strukturu příloženého CD můžete snadno vyrobit příkazem:
`$ tree . >tree.txt`
Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případně **index.html** apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.

Literatura

- [Cic08] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Universit' a di L'Aquila, 2008.
- [DSK14] D. Di Ruscio R. F. Paige D. S. Kolovos, Alfonso Pierantonio. Different models for model matching: An analysis of approaches to support model differencing. [online], Citováno 16.11.2014.
- [Jez12] Jiří Jezek. Modelem řízená evoluce objektů, 2012.
- [Luk11] Martin Lukeš. Transformace objektových modelů, 2011.
- [Luk13] David Luksch. Katalog refaktoringu frameworku migdb, 2013.
- [Maz14] Martin Mazanec. Doménově specifický jazyk pro migdb, 2014.
- [PMH12] Jiří Jezek Pavel Moravec, Petr Tarant and David Harmanec. A practical approach to dealing with evolving models and persisted data. [online], [konference], publikováno 15.4.2012.
- [Tar12] Petr Tarant. Modelem řízená evoluce databáze, 2012.
- [Tar14] Petr Tarant. Migdb - formální specifikace, 2014.
- [Val14a] Michal Valenta. Relační algebra. [online], Citováno 16.11.2014.
- [Val14b] Michal Valenta. Integritní omezení. [online], Citováno 22.11.2014.
- [wc14a] wiki community. Diff definice. [online], Citováno 16.11.2014.
- [wc14b] wiki community. Orm definice. [online], Citováno 16.11.2014.