

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

16. 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	Aplikační metamodel	3
2.1.1	Operace nad aplikačním modelem	4
2.1.1.1	Cíle při modelování operací	4
2.1.1.2	Historický vývoj	4
2.1.1.3	Seznam aplikačních operací	5
2.1.1.4	Rozdělení aplikačních operací	10
2.1.2	Delta notace	10
2.1.2.1	Vlastnosti operací	15
2.2	ODBCHM operací	15
2.3	Modul Migdb	16
2.3.1	Diff elementy	16
2.4	Rozpoznávání operací	19
2.5	Obecné principy model matching	19
2.5.1	Graph matching	20
2.6	Vytvořený algoritmus rozpoznávání operací	22
2.6.1	Návrh ze studia článků	22
2.6.2	Implementace	22
2.7	alternativní algoritmus	22
3	Popis problému, specifikace cíle	23
4	Ukázka zdrojového kódu práce	25
5	Obsah přiloženého CD	27
6	Závěr	29
6.1	Další poznámky	29
6.1.1	České uvozovky	29
7	Seznam použitých zkratk	31
8	UML diagramy	33

9	Instalační a uživatelská příručka	35
10	Obsah přiloženého CD	37
	Literatura	39

Seznam obrázků

2.1	Aplikační metamodel v počátku vývoje z [Luk11]	4
2.2	Aplikační metamodel v průběhu vývoje z [Jez12]	5
2.3	Rootové elementy aplikačního modelu	6
2.4	Aplikační metamodel	7
2.5	Typy graph matchingu	21
5.1	Seznam přiloženého CD	27
10.1	Seznam přiloženého CD — příklad	37

Seznam tabulek

2.1	ODBCHM Seznam operací část 1	17
2.2	ODBCHM Seznam operací část 2	18

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 (SŘBD).

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 [Val10] formální abstrakce využívající relaci jakožto jediný konstrukt. Relace je uspořádaná n -tice souvisejících 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. 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 $T(C) = R * S = \{u \mid u[A] \in R \text{ a } u[B] \in S\}$. [Val10] 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, \dots, an FROM R`, operaci selekce $R[a = 1]$ 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 literatuře 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. Cílem tohoto projektu je definovat ucelenou množinu 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.

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.

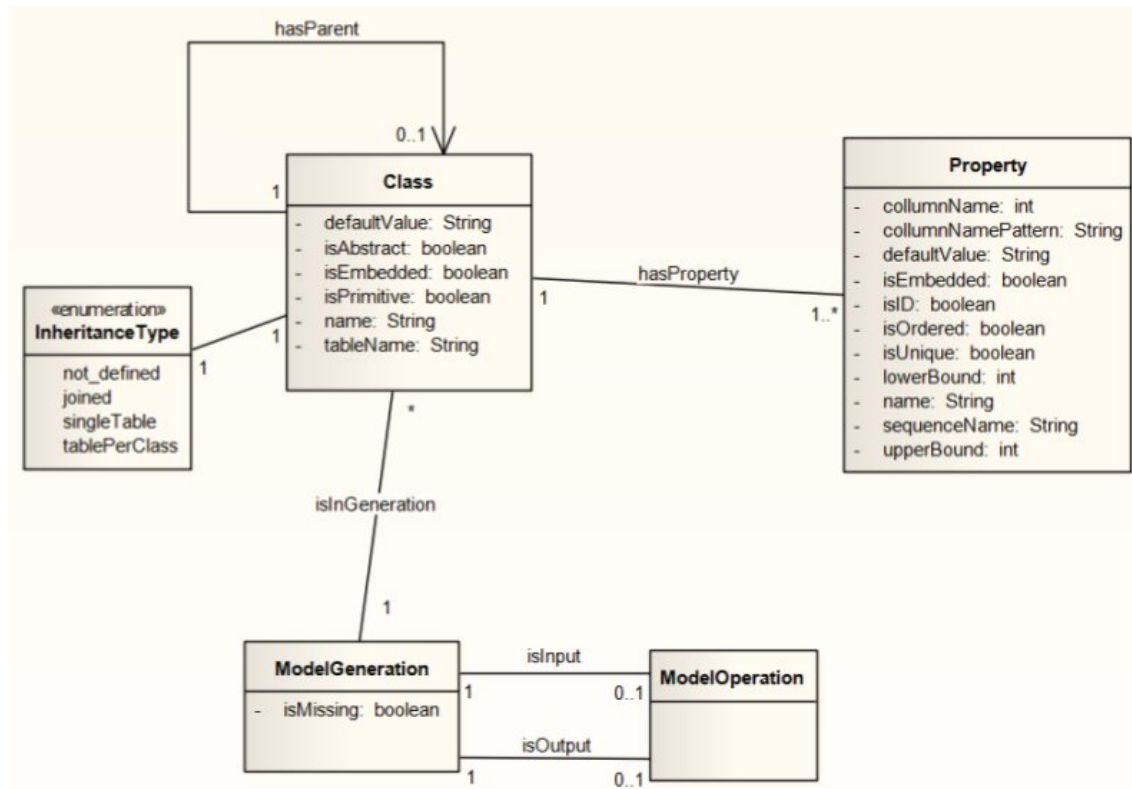
V rámci Migdb bylo v posledních letech vytvořeno 4 bakalářské práce a 2 diplomové práce členů Migdb. Jednalo se o práce mé osoby [Luk11], jež pojednávala o problematice mapování aplikačního modelu na model databázový práce Jiřího Ježka [Jez12] popisující aplikační model a jeho transformace, dále práce Petra Taranta [Tar12] popisující databázový model a jeho transformace a poslední práce pojednává o popisu testování projektu [Luk13].

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

2.1 Aplikační metamodel

Aplikační model zachycuje vztahy mezi jednotlivými objekty tvořícími aplikaci. Ačkoliv byl tento model vytvořen již v raných fázích projektu Migdb a byl často upravován, většinou zjednodušován. Na obrázku 2.3 jsou znázorněny kořenové elementy nynějšího aplikačního modelu - každý aplikační model musí obsahovat jeden container - potomka třídy ModelRoot. Na obrázku 2.4 jsou zobrazeny elementy patřící do Structury aplikačního modelu. Při vývoji byla přejmenována třída Class, kterou je možno vidět na 2.1 z původního metamodelu na StandardClass. Oproti aplikačnímu metamodelu [Jez12] byly odstraněny Entity EmbeddedClass a její předek GeneralClass, dále byla zjednodušena třída Property, u níž ubýly 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.



Obrázek 2.1: Aplikační metamodel v počátku vývoje z [Luk11]

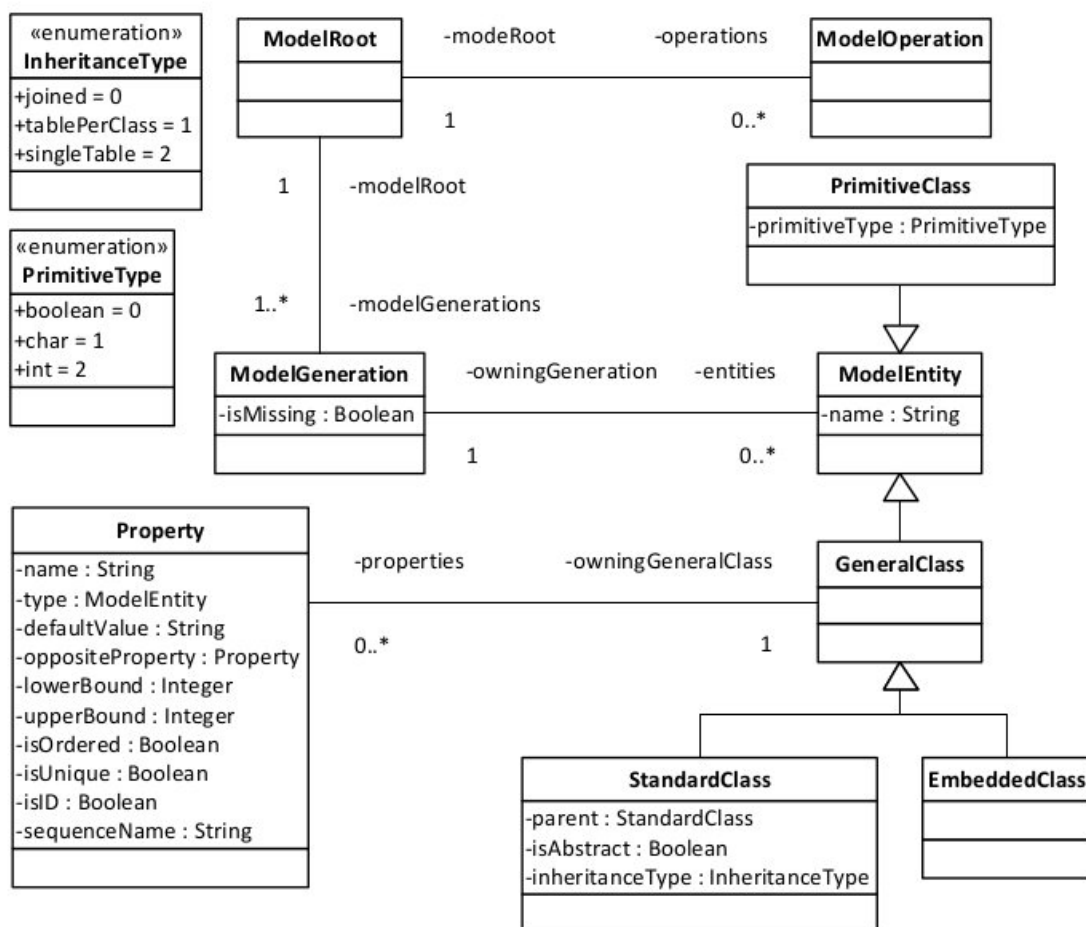
2.1.1 Operace nad aplikačním modelem

2.1.1.1 Cíle při modelování operací

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.

2.1.1.2 Historický vývoj

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 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



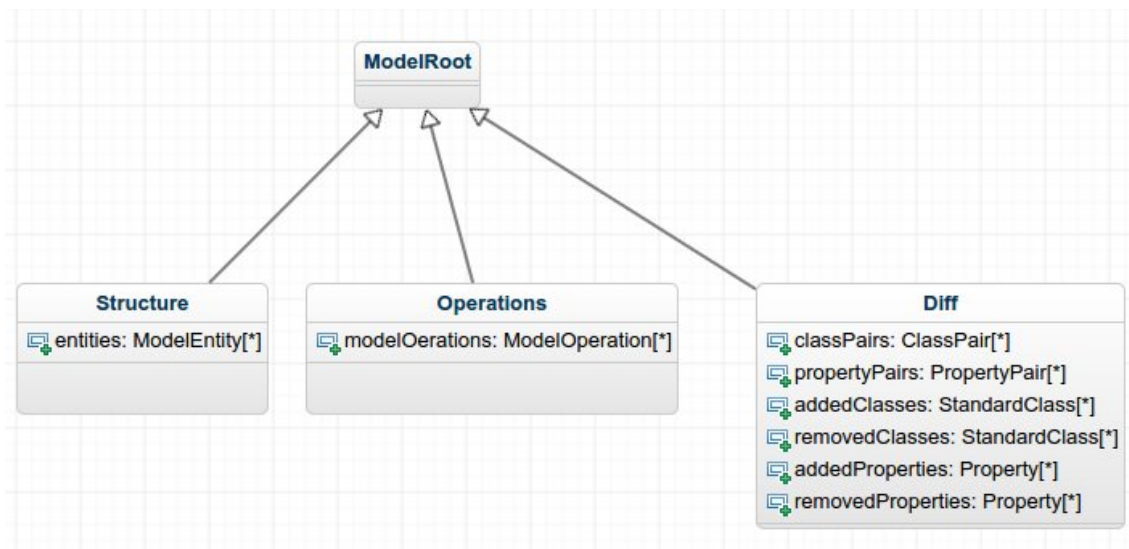
Obrázek 2.2: Aplikační metamodel v průběhu vývoje z [Jez12]

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. Je očekatelné, že by tato operace měla souviset s dědičnými hierarchiemi.

2.1.1.3 Seznam aplikačních operací

Operace jsou uvedeny v tabulkách 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. 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.

Operace I: AddStandardClass(name, isAbstract, inHeritanceType)



Obrázek 2.3: Rootové elementy aplikačního modelu

- 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)

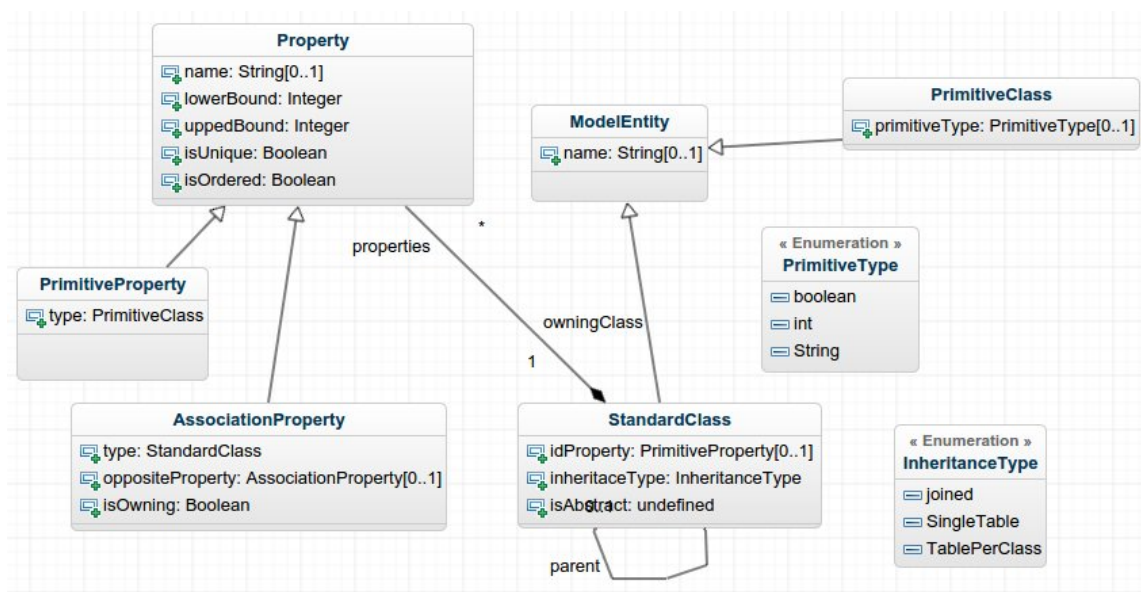
- Validační podmínky - existuje třída s daným jménem
- Operace nastaví třídě atribut abstract na danou hodnotu

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



Obrázek 2.4: Aplikační metamodel

Operace VI: RenameProperty(owningClassName, name, newName)

- Validační podmínky - existuje přejmenovaná property v dané třídě, neexistuje property v dané třídě nového jména
- Operace změny názvu 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)

- 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 překrývající 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

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 opositePropertyName

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

2.1.1.4 Rozdělení aplikačních operací

Operace nad aplikačním modelem je možné dělit podle dvou kritérií - 1. nad jakým typem entity pracují, 2. jaký je charakter/význam pro tuto entity daná operace má.

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 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 změny obsah. Příkladem této operace je `InlineClass`. Reduktivní operace jsou inverzní k operacím expanzivním.

Rozdělení operací nad aplikačním modelem je jen formální, neprojevovalo se změnou hierarchické struktury operací. Struktura operací byla zjednodušena - již neexistuje rozhodovatelná operace `ComposedOperation`, všechny operace jsou nyní defakto atomické. Tato změna byla zapříčiněna neschopností rozložit některé dekomponovatelné operace na operace atomické. Je zde nutné podotknout, že tento rozklad je v nynější chvíli možný, ačkoliv si vyžádal neformální obohacení aplikačního modelu o některé atomické operace jakými jsou `mergeProperty`, `distributeProperty` a `exportProperty`. Tyto operace v aplikačním modelu neexistují, ale v rámci transformace ODBCHM jsou v metodách použity/volány.

2.1.2 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.

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 $part_a part_b$

```

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

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

```

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 surename;
<
10,13d7
<         public Man(String name, String surename){
<             this(name);
<             this.surename = surename;
<         }

```

Listing 2.10: diff modelů 2 a 1

```

4,5d
10,13d7

```

2.1.2.1 Vlastnosti operací

V [Cic08] jsou popsány 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. (intensional vs extensional - první potřebuje konkrétní model ke své aplikaci, druhá ne, extensional je možné použít na paralelní vývojové větve).

V projektu Migdb na druhé straně existují operace, které jsou kontextově závislé, což je uživatelskou přívětivostí operací - jako zástupcem takové operace se dá uvést operace `AddParent(parentClass=B, childClass=A)`, která nezmiňuje všechny Property, které se mají odstranit, ale dynamicky si je dopočítává v závislosti na daném kontextu. Její inverzí je operace `RemoveParent(childClass=A)`. Vzhledem k minimalističnosti neexistuje k operaci `RemoveParent(childClass=A)` jednoznačná inverze bez daného kontextu.

Inverzi `AddParent(sourceClass=A parentClass=B)` získáme, pokud v kontextu přidruženém modelu aplikaci `RemoveParent` má třída A předka B. Nicméně i v takovém případě neplatí, že aplikace sekvence těchto dvou inverzních operací na vstupní model M vygeneruje vždy původní model. Příkladem tohoto neočekávaného chování je vstupní model `Man(int age, String name)`, `Person(int age, String name, String sureName)`. Aplikace operace `AddParent(Man, Person)` odstraní přebytečné atributy v třídě `man` a přidá rodiče této třídy, tj vznikne model : `Man()->Person, Person(int age, String name, String sureName)`. Nyní je aplikována operace `RemoveParent(Person)`, která sice získá z modelu jméno rodičovské třídy, ale nezíská seznam property, které se distribuují do třídy `Man`. Operace byly navrženy tak, aby maximalizovali objem zachovaných dat, proto operace `RemoveParent` distribuuje všechny atributy třídy `Person` do třídy `Man`. Výsledný model je tedy: `Man(int age, String name, String sureName)`, `Person(int age, String name, String sureName)` a odlišuje se od vstupního modelu o property `sureName` v třídě `Man`.

Stejná vlastnost platí pro dvojici `ChangeUniToBidir` a `ChangeBiToUnidir`, s rozdílem, že se nejedná o automaticky získanou rodičovskou třídu, ale opozitní property.

2.2 ODBCHM operací

Ačkoliv v průběhu projektu Migdb byla transformace operace z aplikačního modelu na sadu operací databázových označována jako ORM operací, rozhodli jsme se změnit název této transformace na `Operation Database Change Mapping`, kde dána databázová změna reprezentuje sadu DDL, DML operací, které vzniknou pomocí transformace. Ačkoliv na úrovni aplikační všechny operace fungují se všemi `inheritanceTypy` bylo nutné zjednodušit aplikační model tak, aby byla transformace ODBCHM 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 ODBCHM si bere všechna data z aplikačního modelu, čímž je nezávislý na aplikaci databázových operací nad databázovým modelem, ale předává veškerou zodpovědnost za údržbu - tj vytvoření a odstranění omezení. V tabulkách 2.1 a 2.1 nejsou uvedena vytváření a odstraňování unikátních constrainů pro unikátní či ordered kolekce primitivních a

neprimitivních typů.

V tabulkách

2.3 Modul Migdb

Původně modul mapování operace fungoval podle následujícího schématu:

Pro každou vstupní operaci AOp

1. validace AOp
2. aplikace AOp na vstupní model
3. Rozklad operace AOp na seznam db operací
4. Pro každou operaci DbOp rozkladu:
 - a) validace operace DbOp
 - b) aplikace operace DbOp Generování SQL z seznamu DB OP

Tento přístup narazil na některé problematické případy v průběhu testování aplikace výsledných SQL nad daty v databázi. Z tohoto důvodu a z důvodu přehlednější implementace bylo základní schéma modulu pozměněno a byl přidán koncept `mirroredOperations`.

Představme si, že uživatel potřebuje aplikovat operaci `ExtractClass(A, B, props)`, A je třída, z které se extrahuje, B je nově vzniklá třída, do které se budou přesouvat property z kolekce `props` a jejich data.

Tato operace pracuje nad aplikačním modelem následovně:

Vytvoří třídu B

Vytvoří v třídě A referenční property, která bude mít typ B

Pro každou property z kolekce `props` aplikuje operaci `MoveProp` - vytvoří v třídě B Property, přesune data do nově vzniklé property, smaže ji z původní třídy A

Krok 1 se v databázi projeví standardně. V kroku 2. je v databázi kromě vytvoření sloupce nutné vygenerovat data v nově vzniklé a referencovat je v tabulce kdy není možné oddělit v ODBCH krok 2, vytvoření referenční Property. V databázi je nutné vytvořit sloupec, nicméně tento sloupec musí obsahovat

2.3.1 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 rodin `ConstructiveClassPair`, `DestructiveClassPair` a `ModifyingClassPair`, ale aby bylo možné rozpoznat specifický pár závislý na jiném páru, byla přidána třída `ReplacingClassPair` - nahrazující pár, který se používá jako pivot pro hledání konstruktivních, destruktivních a modifikačních párů. Od elementu `ReplacingClassPair` dědí elementy `EqualClassPair` - třída, která si uchovávala jméno z původního modelu a element `ReplacingClassPair` - reprezentující třídu, která

Název operace	podmínky validnosti	Rozklad na db operace
AddStandardClass		Vytvoří tabulku, id sloupec této tabulky a primární klíč
RenameEntity	Existuje třída s původním jménem, neexistuje třída s novým jménem	Operace změny názvu tabulky na nový, odstraní a vytvoří PK s novým jménem
SetAbstract	Existuje třída s daným jménem	pro isAbstract = true maže data, která náleží pouze dané třídě
RemoveEntity	Existuje třída s daným jménem, neexistuje link na tuto třídu, třída neobsahuje žádné property, neexistuje pro tuto třídu žádný potomek	V Db je smazán primární klíč, property a tabulka
AddProperty	zadané bounds jsou validní, v hierarchii dědičnosti neexistuje kolizní property	operace přidá do cílové tabulky sloupec pro primitivní property S UB = 1 operace přidá tabulku, datový sloupec, referenční sloupec a referenci na vlastnickou tabulku pro primitivní property s UB = > 1 operace přidá sloupec pro neprimitivní property S UB = 1 do vlastnické tabulky a referenci na tabulku typu operace vytvoří vazební tabulku pro neprimitivní property s UB > 1, vloží do ní referenční sloupce na vlastnickou tabulku a tabulku typu, na které vytvoří cizí klíč
RenameProperty	Existuje přejmenovaná property v dané třídě, neexistuje property v dané třídě nového jména	pro danou property s UB = 1 a primitivním typem přejmenuje property v vlastnické tabulce pro danou primitivní property s UB > 1 přejmenuje datový sloupec, FK na vlastníka kolekce a tabulku kolekce pro danou asociační property přejmenuje sloupec v vlastnické tabulce a cizí klíč referující tabulku typu pro danou asociační property přejmenuje vazební tabulku s referenčními sloupci na vlastníka a typ asociace + cizí klíče
RemoveProperty	Musí existovat vlastnická třída property a v ní odstraňovaná property	pro primitivní property S UB = 1 odstraní sloupec z dané tabulky pro danou property primitivního typu s UB > 1 odstraní referenci na vlastnickou tabulku, sloupec z tabulky dané kolekce, datový sloupec a smaže kolekční tabulku pro danou asociační property s UB = 1 odstraní referenci na tabulku

Název operace	popis	parametry
CollapseHierarchy	Exportuje property z jedné třídy do jejího předka a třídy spojí, upraví dědičné vazby	superClassName, subClassName, isIntoSub
ExtractSubClass	Vytvoří třídu nového potomka a přesune do něj vyjmenované property	sourceClassName, extractedClassName, extractedPropertyNames
ExtractSuperClass	Vytvoří třídu nového předka a přesune do něj vyjmenované property, pokud měla původní třída předka nastaví tohoto předka rodičem nově vzniklé třídy	sourceClassesName, extractParentName, propertyNames
PullUpProperties	Exportuje property do rodičovské třídy	childClassName, pulledPropertiesNames
PushDownProperties	Exportuje vyjmenované property do třídy potomka a přesune JEN data potomka	childClassName, pushedPropertiesNames
ExportProperty(virtuální operace)	Exportuje property v rámci hierarchie a data do cílové třídy	exportedPropertyName, className
DistributeProperty(virtuální operace)	Zduplikuje strukturu dané property do cílové třídy a přesune data přiřazená této třídě	DistributedPropertyName, className
MergeProperty(virtuální operace)	Přesune data zdrojové property do cílové property a smaže strukturu původní property	mergedPropertyName, className

Tabulka 2.2: ODBCHM Seznam operací část 2

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`. Oproti jednodušším operacím aditivním a subtraktivním jsou operace destruktivní, konstruktivní a modifikační v `Diff` modelu zobrazeny do elementů `ConstructiveClassPair`, `DestructiveClassPair` a `ModifyingClassPair`.

2.4 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 -> C` na model `X` dostaneme stejný výstup jako aplikací operací `Inline B, A + Rename A -> 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.

2.5 Obecné principy model matching

Jak je diskutováno v 19 *Different Models for Model Matching: An analysis of approaches to support model differencing* 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 algoritmu 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 řešení problému.

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 a dobré si uvědomit, že funkci vytvoření DB modelu zvládá ORM mapování integrované do většiny současných IDE.

V literatuře (*Different Models for Model Matching: An analysis of approaches to support model differencing*) byly popsány algoritmy pro mapování shodných entit modelů (*Model-Matching*) 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.

2.5.1 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 2.5 na matching nalezení přesné shody vzorku v hledaném grafu či matching hledání podobnosti grafu vzorku v hledaném grafu.

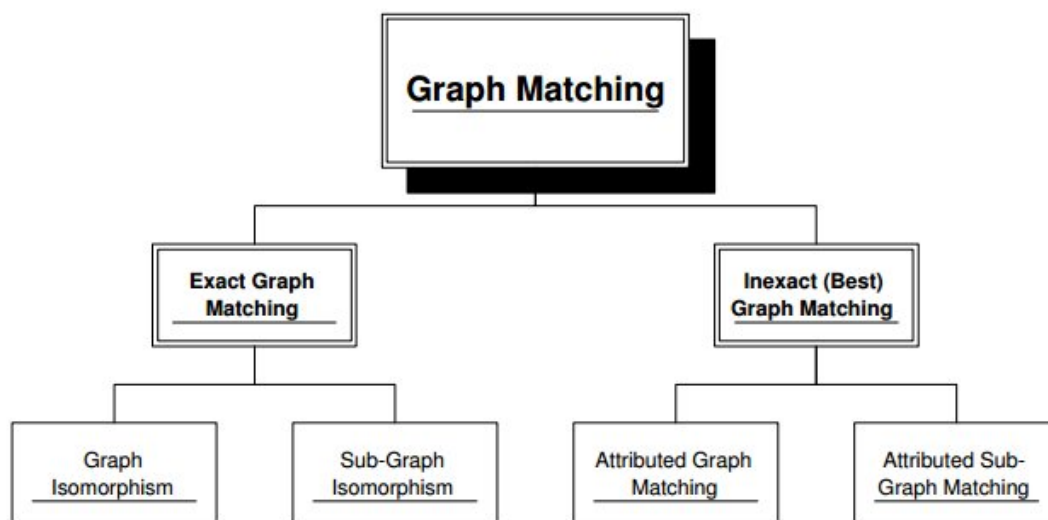
Matching hledání přesné shody je definován následně: Mějme grafy $G_P = (V_P, E_P)$ a $G_M = (V_M, E_M)$, přičemž $\|V_M\| = \|V_P\|$, úkolem je potom najít takové prosté zobrazení $f : V_D \rightarrow V_M$, takové, že $(u, v) \in E_P$ iff $(f(u), f(v)) \in E_M$. Pokud takové mapování existuje, nazveme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

Termín Inexact matching aplikovaný na některé problémy týkající se shodnosti grafů vyjadřuje/zneméná, 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 takovém 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.



Obrázek 2.5: Typy graph matchingu

2.6 Vytvořený algoritmus rozpoznávání operací

2.6.1 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 ná 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.

2.6.2 Implementace

Vzniklo několik implementací párovacích algoritmů. První a nejjednodušší používá párování tříd podle jména, následně rozdíl 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 rozpoznané páry. Závislost rozpoznání

Algoritmus rozpoznávání operací byl napsán se snahou o zachovávání 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.

2.7 alternativní algoritmus

V ranné fázi byl napsán prototyp jiného rozpoznávacího algoritmu, který se snaží minimalizovat vzdálenost současného modelu od modelu cílového pomocí rozpoznání 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

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

Ukážka zdrojového kódu práce

Kapitola 5

Obsah přiloženého 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 5.1: Seznam přiloženého CD

Kapitola 6

Závěr

Ačkoliv se mi nepodařilo dokončit tuto diplomovou práci v termínu, dokončil jsem implementační(viz ukázka kódu) a testovací části projektu, které jsem nestihl zdokumentovat. Proto bych byl rád, kdybych mohl věnovat následující půlrok přepracování textu diplomové práce.
[?]

6.1 Další poznámky

6.1.1 České uvozovky

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

Kapitola 7

Seznam použitých zkratek

IDE Integrated Development Environment

ORM Object-relational mapping

EMF Eclipse modeling framework

⋮

Kapitola 8

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 9

Instalační a uživatelská příručka

Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Kapitola 10

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 10.1: 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, Università di L'Aquila, 2008.
- [Jez12] Jiří Jezek. Modelem řízená evoluce objektů, 2012.
- [Luk11] Martin Lukeš. Transformace objektových modelů, 2011.
- [Luk13] David Luksch. Katalog refaktoringu frameworku migdb, 2013.
- [PMH12] Jiří Jezek Pavel Moravec, Petr Tarant and David Harmanec. A practical approach to dealing with evolving models and persisted data. Technical report, FEL ČVUT, CollectionsPro, 15.4. 2012. Dostupný na <http://www.codegeneration.net/cg2012/sessioninfo.php?session=37>.
- [Tar12] Petr Tarant. Modelem řízená evoluce databáze, 2012.
- [Val10] Michal Valenta. Databazové modely. Technical report, Fakulta informačních technologií ČVUT, 2010. Dostupný na https://users.fit.cvut.cz/valenta/doku/lib/exe/fetch.php/bivs/dbs_02_databazove_modely.pdf.
- [wc14a] wiki community. Diff definice. Technical report, Wikipedia.org, 16.11. 2014. Dostupný na <http://cs.wikipedia.org/wiki/Diff>.
- [wc14b] wiki community. Orm definice. Technical report, Wikipedia.org, 16.11. 2014. Dostupný na http://cs.wikipedia.org/wiki/Objektov%C4%9B_rela%C4%8Dn%C3%AD_mapov%C3%A1n%C3%AD.