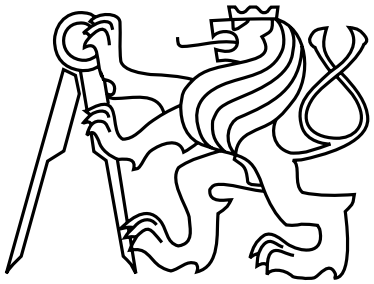
Zadání práce

Popište problematiku evoluce objektových modelů v souvislosti s jejich mapováním na model relační databáze. Vytvořte zjednodušený metamodel pro model tříd a model relační databáze. S jejich pomocí definujte a implementujte základní operace nutné pro zachování konzistence mapování při změně objektového modelu. Pro transformaci využijte jazyk QVT.

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Transformace objektových modelů

Martin Lukeš

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

26. května 2011

Poděkování

Zde bych chtěl poděkovat vedoucímu mé bakalářské práce, Ing. Ondřeji Mackovi za jeho ochotu, trpělivost a za cenné rady, které přispěly k dokončení této práce. Dále pak bych chtěl poděkovat členům firmy CollectionsPro, kteří se zapojili do projektu Migdb za jejich rady, co se týče implementační části.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém 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 26. května 2011 .....

Abstract

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její podstatu. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Tato bakalářská práce si klade za cíl poskytnout náhled do problematiky metamodelování ve spojitosti s mapováním mezi objekty na objekty z databázové domény - Object Relational Mappingu. Dále bude v práci implementován prototyp mapování realizující ORM transformaci za pomoci metodiky vývoje řízené modelem reprezentované jazykem QVT Operational - imperativní formy QVT standardu v vývojovém prostředí Eclipse.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

1.1 Osnova

Tato bakalářská práce byla zadaná jako počáteční fáze projektu Migdb (Migrace Databáze) zadaného externím zadavatelem - firmou CollectionsPro. Jejím účelem je seznámit čtenáře s problémem, analyzovat problém a zhodnotit kvalitu řešení pomocí vzorových příkladů. Konkrétní cíle této práce jsou:

- Popis databázového a aplikačního meta-modelu
- Rozbor vazeb mezi objektovým a databázovým světem
- Realizace ORM transformací

Práce je strukturována následujícím způsobem :

- 1. kapitola přibližuje čtenáři téma této bakalářské práce, vysvětluje motivaci a přibližuje doménu problému
- Kapitola 2 ukazuje použité technologie v rámci implementační části této práce, jedná se o vývojové prostředí, ale i samotné formáty dat, standardy
- Kapitola 3 popisuje metamodely aplikace a databáze
- Kapitola 4 si klade za cíl analýzu transformace pomocí matematického modelu a převedení tohoto modelu do konkrétní podoby jazyka QVT, který byl z tohoto důvodu vyjmut z kapitoly 2
- Kapitola 5 popisuje realizaci transformací
- 6. kapitola zhodnocuje provedené testy
- 7 kapitola jako závěrečná hodnotí tuto bakalářskou práci a spekuluje o dalším směřování projektu Migdb.

1.2 Motivace

V průběhu poslední dekády je vyvíjeno více nového software než kdy předtím, ale na druhé straně i stávající software je častěji a častěji modifikován, ať už je to dáno existencí rozsáhlého Legacy systému či špatného návrhu. Valná většina dnešních projektů potřebuje ukládat větší množství dat, povolit a řídit přístup více uživatelů k datům najednou, a proto je používán specializovaný software - databáze.

Kvůli nutnosti modifikace, dokumentace a komunikace mezi vývojáři vznikají různé typy modelů. Existuje objektový model aplikace, zachycující strukturu programu a model databázový. Mění/vyvíjí aplikace, mění se objekty v ní obsažené, mění se ukládané objekty a mění se i struktura databáze. Aby byla aplikace plně funkční, je nutné zachovat konzistenci mezi databázovým a aplikačním modelem, což se dá zajistit manuálně, ale v některých připadech je nutné razantně modifikovat velkou databázi. Čas strávený touto akcí s potřebnou precizností navyšuje náklady vývoje. Automatizace změny databázového schématu mívá v mnohých případech za následek ztrátu dat v databázi, což je málokdy přijatelná cena - opětovné vložení dat do databáze může být stejně náročnou záležitostí jako samotná modifikace databáze.

Proto je hnacím motorem tomuto projektu vytvoření prototypu a analýza podmínek nutných pro úspěšnou automatizaci procesu změny databázového schématu za pomoci metamodelování jakožto možné cesty vedoucí k cíli. Komplexnost dané problematiky má za následek zpracování jen části této problematiky v této bakalářské práci.

1.3 Objektové světy

Ačkoliv se v práci předpokládá čtenářova znalost objektového modelování a obecně práce s objekty, pro jistotu budou vysvětleny základní pojmy, které jsou dále rozvedeny v [7]. **Objektový model** představuje simulaci nějakého skutečného nebo smyšleného virtualního světa v světě tvořeném objekty. **Objekty**, s nimiž se v programech setkáváme, můžeme rozdělit do skupin, které mají nějaké společné vlastnosti. Tyto skupiny v programování označujeme jako **třídy**. Vlastní objekty označujeme jako **instance** příslušné třídy. Mohli bychom tedy říci, že instance (objekt) je konkrétní realizací obecného vzoru definovaného příslušné třídě.

Lidem (programátorům) je blízké použití tříd a jejich instancí - objektů jakožto základních stavebních jednotek programu. Každá třída je jakousi formou - předpisem k vytvoření svých instancí. Vlastnosti, které by měly mít instance dané třídy, jsou v objektovém světě reprezentovány pomocí atributů třídy.

Definice převzatá z [11]:
V OOP (Objektově orientovaném programování) je asociace neboli vazba definujícím aspektem vztahu mezi třídami objektů povolující instanci třídy zapříčinit vyvolání akce na instanci druhé. Tento vztah je strukturální povahy, protože specifikuje, že objekty jednoho typu jsou spojeny s objekty typu druhého.

Vazba "A has B" definuje vztah mezi objektem a jeho atributy. Překlad "A has B" do Češtiny je "A má B" nebo spíše "A vlastní B". Techniku vytváření tříd pomocí jiných již existujících nazýváme skládáním.

Pokud bychom například měli třídu Date s atributy month typu int, day typu int a year typu int, dále pak třídu Person s atributy birthday typu Date, name typu String (řetězec), pak by mezi těmito třídami byl vztah "Person has Date". Konkrétní instance Date by měla atributy například year 1976, month 4, day 21, instance třídy Person by mohla mít konkrétní atributy, například age roven 24 a name s hodnotou "Roman"a vlastnila by zmíněnou instanci třídy Date.

Krom atributů může a zpravidla má každá třída své metody. Metody třídy reprezentují reakci instanci nebo třídy na zasílání zprávy. Pokud zasíláme zprávu, obvykle říkáme, že voláme metodu dané třídy nebo instance. Pokud vynecháme metody statické, tak metody rozdělujeme do dvou skupin. První skupinou jsou metody, které vytvářejí novou instanci dané třídy, tyto metody jsou potom nazývány konstruktory. Druhou skupinou jsou potom metody, pomocí níž manipulujeme s objekty - primárně nevytváříme nové instance třídy, na jejíž instanci metodu voláme.

V objektovém světě může být mezi třídami vazba "A is B" znázorněná dědičností, kdy třída A dědí od B. Relace dědičnost je tranzitivní a antisymetrická. Pro vysvětlení těchto pojmů si vezměme dříve zmíněnou třídu Person, třídy Student a GoodStudent, přičemž třída Student dědí od třídy Person a třída GoodStudent dědí od třídy Student. Tranzitivita nám pro náš příklad ukazuje, když platí, že třída Student dědí od třídy Person a třída GoodStudent dědí od třídy Student, pak také GoodStudent dědí od třídy Person.

Soubor atributů a metod, které o sobě daná entita (třída, metoda nebo atribut), prozrazuje navenek nazýváme interface nebo též rozhraní. Rozhraní dělíme do dvou částí - signatura a kontrakt. **Signatura** zahrnuje identifikátory atributů, jejich typy, dále pak identifikátory metod, návratovou hodnotu metod, identifikátory parametrů s jejich typy, ale v neposlední řadě pak výjimky, které jsou vyhazované danými metodami. Signatura je zpracovatelná a kontrolovatelná překladačem. **Kontrakt** označuje souhrn dalších zásad, které je třeba při použití dané entity (třídy, atributu, metody) dodržet. Jejich kontrola již není proveditelná překladačem. Patří sem omezení na konkrétní hodnoty parametrů či například vzájemná implementační závislost metod. Nejen o problematice rozhraní pojednává více [6].

1.4 Databáze

Databáze, v anglické literatuře nazývaná jako Database Management System (DBMS) je specializovaný software sloužící k persistenci dat.

1.4.1 Vlastnosti a výhody použití DBMS

Databáze jsou uzpůsobeny k těmto úkonům (viz. [8])

- Query ability - schopnost databáze dotazovat se na vlastnosti dat. Tato vlastnost je realizována v relačních databázích jazykem SQL.
- Zálohování a replikace - tyto dvě vlastnosti zabraňují ztrátě dat.

- Transakční zpracování - Transakční zpracování nám zajišťuje možnost souběžného pracování více uživatelů a umožňuje zachování ACID vlastností Zkratka ACID je složená ze 4 slov - atomicity, consistency, isolation a durability, atomicity transakcí zajišťuje celé provedení transakce nebo její neprovedení, je zabráněno částečnému provedení transakce, consistency je vlastnost zabezpečující, že integritní omezení databáze nebudou porušena. Isolation transakce zabezpečuje, že modifikovaná data před ukončením transakce nebudou přístupná. Durability zajišťuje dlouhodobost commitovaných dat při chybě systému.
- Security - zabraňuje přístupu uživatelů bez příslušného oprávnění k přístupu k databázi či k úpravě databáze.

1.4.2 Typy DBMS

Bylo vyvinuto několik koncepčně odlišných modelů DBMS.

- Hierarchický model organizuje data do stromové struktury. Tento model je schopný popsat objekty reálného světa vazbou typu 1 x N.
- Síťový model je rozšířením modelu hierarchického. V síťovém modelu existují dva typy konstrukcí - set a record. Pomocí setu je možné zachytit vztah 1 x N, pomocí recordu je možné zachytit vztah M x N.
- Relační model (RDBMS) je definován pomocí termínů z predikátové logiky a teorie množin. Základní strukturou uložených dat v relačním modelu je tabulka obsahující sloupce. Informace o jednotlivých instancích relačního modelu jsou sdruženy do řádků. Po dlouhou dobu byl nejpoužívanějším typem DBMS, zvláště díky popularitě dotazovacího jazyka SQL (Structured Query Language).
- Objektově Orientovaný model (OODBMS) je spjat s rozvojem objektově orienntovaného programování. Umožňuje ukládání objektů do databáze. Jeho nevýhodou je vázanost na programovací jazyk, neexistence standardů - obzvláště dotazovacího jazyka.
- Objektově Relační model (ORDBMS) je hybridem mezi Objektovým a relačním typem, obohacuje relační model o definování vlastních datových typů uživatelem a některé další vlastnosti objektových DBMS, ale stále podporuje dotazovací jazyk SQL a proto se stává nástupcem relačního modelu.

V této práci je použita relační databáze, jelikož projekt zadavatele je vyvíjen již delší dobu a v době jeho vzniku byl nejrozšířenějším modelem relační model, který v nynější době zůstává jedním ze dvou nejpoužívanějších.

Definice v této sekci byly inspirovány [9] a [2], odkud byly definice převzaty a upraveny.

1.5 Objektově relační mapování

V nynější době většina aplikací je vyvíjena v některém z moderních objektově orientovaných programovacích jazyků (Java, C#, C++) a potřebuje ukládat data do databáze. Jak již bylo

řečeno v kapitole o databázích, nejrozšířenějšími typy databázových management systémů jsou nyní relační a objektově relační. Svět objektů byl vytvořen na základě potřeb a principů softwarového inženýrství, naopak relační koncept byl vytvořen na základě matematického modelu.

Oba koncepty sdílí některé myšlenky, ale objevují se v nich i mírné odlišnosti. Vzhledem k odlišnostem relačního a objektového světa vzniká mezera mezi těmito dvěma světy, v angličtině nazvaná „“Object-Relational Impedance Mismatch„“. Oba koncepty spolu koexistují již velmi dlouhou dobu a k odstranění této bariéry bylo vymyšleno právě Objektově relační mapování (ORM). Objektově relační mapování simuluje existenci objektů a jejich vzájemných vztahů v databázi pomocí tabulek, jejich sloupců a klíčů z databázové domény.

ORM je používáno již velmi dlouhou dobu a proto není divu, že vzniklo již mnoho jeho frameworků. Jako příklady se dá uvést Hibernate, EclipseLink, různé implementace JPA (Java Persistence API), ale dále je možné jmenovat i zástupce ORM frameworků pro php jako jsou například Doctrine a Propel. V rámci tohoto projektu nebyl vytvořen framework, ale byla vytvořena transformace, která mění aplikační model na model databázový.

1.6 Modelování

1.6.1 Model

Následující definice byly převzaty z [4]. Jakýkoliv **model** je abstraktní reprezentací nějaké reálné skutečnosti. Všechny modely zachovávají relevantní vlastnosti modelovaných objektů a abstrahují od méně důležitých detailů, čímž skrývají jejich komplexnost. Díky tomuto zjednodušení nám modely umožňují snadnější manipulaci s objekty, které reprezentují.

Modelování je snaha o zachycení nějaké entity z určitého, zjednodušujícího pohledu. V programování existuje velké množství modelů, většina z nich je zapsatelná textovou formou. Nicméně pro názornost a k zlepšení/zjednodušení komunikace mezi vývojáři byl vytvořen grafický jazyk pro vizualizaci, navrhování a dokumentaci programových systémů. Tento jazyk se nazývá UML (Unified Modeling Language) a byl standardizován organizací OMG (Object Management Group).

1.6.2 Metamodelování

Metamodelování a metamodel jsou pojmy, které jsou dnes často používány, dokonce tak často, že se mluví i o meta-metamodelech. Stejně jako při používání objektově orientovaného paradigmatu je i při používání metamodelů kladen důraz na efektivitu. Slovo metamodel je složeno z předpony meta (nad) a základu model. Metamodel je jakýsi pohled shora - popisující strukturu modelu, jakási "globalizace". Globalizace je často zaměňována s pojmem generalizace, nicméně tyto pojmy mají různý význam. Generalizace popisuje vztah mezi objektem typu A a objektem typu B. Tato tvrzení týkající se generalizace jsou ekvivalentní:

- A generalizuje (zobecňuje) B
- B specifikuje (upřesňuje) A

- B je speciální případ A

Naproti tomu globalizace nám nepopisuje vztah B "JE" (IS) specifičtější A. K ukázce metamodelu v IT nemusíme chodit nijak zvlášť do hloubky - vezměme si například problematiku XML (Extensible Markup Language) a DTD (Document Type Definition). XML dokument je validní vůči DTD, který ho popisuje. DTD je v tomto případě model daného XML dokumentu. Jazyk, v kterém je DTD zapsán nám tvoří metamodel (například Element, Atribut, sekce CDATA). Jako data by nám v tomto případě sloužil nějaký konkrétní XML dokument. Pro globalizaci platí tato ekvivalentní tvrzení:

- A popisuje B
- B je instancí A

Předpona meta je zavádějící a občas vytváří komunikační bariéru mezi vývojáři, neboť je možné se rychle ztratit v úvahách, co je vlastně onen metamodel, o kterém je právě řeč. V projektech existuje mnoho úrovní - téměř vždy je možné specifikovat více náš pohled a jít o úroveň níže. Na druhé straně cesta k abstraktnějším úrovním není nekonečná. Vrchní vrstva této architectury popisuje sama sebe.

Většina projektů je založena na tzv. čtyřúrovňové architektuře, viz OBR 1.1. Na levé straně obrázku je zobrazena závislost úrovní modelů, na pravé jsou potom příklady prvků, které patří do dané úrovně. V publikacích bývají úrovně značeny jako M0 - M3.

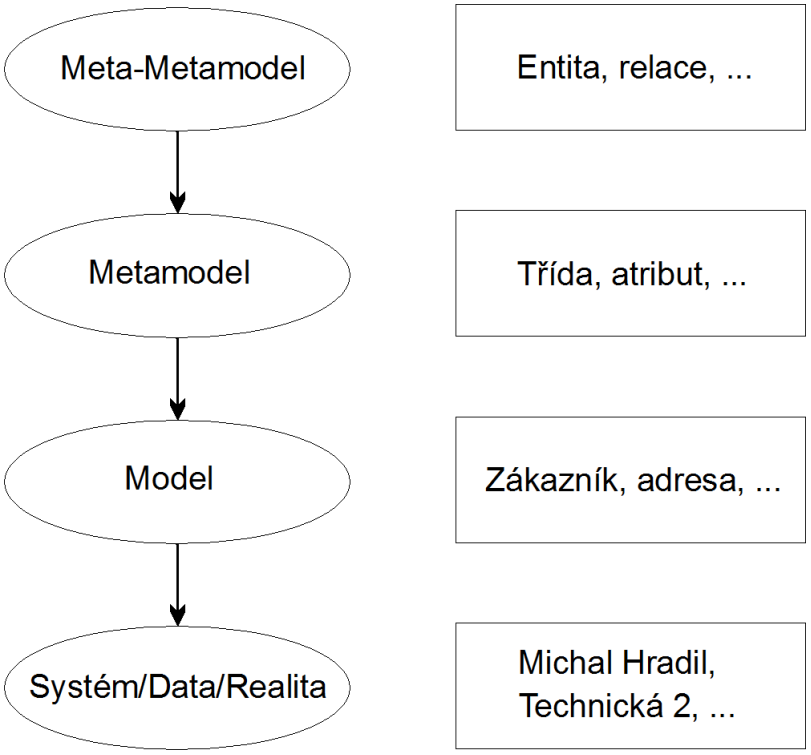
Na úrovni meta-metamodel se setkáváme s prvky jako je entita, relace. Tato úroveň nemá nad sebou již žádnou další, nicméně je popsatelná sama sebou. Do úrovně metamodel patří prvky třída, atd. Obecná struktura prvků úrovně metamodel je popsaná pomocí prvků úrovně meta-metamodel. Zástupci úrovně model jsou již konkrétní třídy jako je například třída Zákazník, Adresa ... nebo atributy Jméno, Věk, PSČ, atp. Poslední popsanou úrovní je úroveň, kde definujeme již konkrétní data. Pokud na úrovni metamodel existuje třída Zákazník(int : věk, String : jméno, String : příjmení) a dále třída Adresa(String : ulice , int : čísloDomu, int : PSČ), potom na nejnižší úrovni by mohla existovat instance třídy Zákazník: (27, Michal, Hradil) a instance třídy Adresa: (Technická, 2, 166 27)

1.6.3 Model Driven Architecture

Martin Fowler v [3] uvádí, že nějčastější použití grafického modelovacího jazyka UML je **tvorba náčrtku (UML as sketch)**. Toto využití zjednodušuje práci na projektech a komunikaci mezi vývojáři. V rámci zpětného inženýrství je možné náčrtky použít i k pochopení existujícího kódu.

Další variantou je **využití UML pro podrobný návrh (UML as Blueprint)**. Musí být splněn předpoklád, že v teamu existuje člověk či skupina osob, která vytvoří pomocí UML návrh a ten je následně programátory převeden do formy zdrojového kódu.

Jako poslední variantu Martin Fowler uvádí **využití UML jako programovacího jazyka**. Tato možnost je nejméně častá. Vyžaduje specializovaný software schopný generovat kód.



Obrázek 1.2: Data-model-metamodel-metametamodel

Model a modelování jsou pojmy zakořeněné v procesu vývoje software stejně jako používání jazyka UML. Přístupy k využití modelů jsou shodné s těmi, které uvádí Fowler pro využití UML.

Technika vývoje software postavená na tvorbě a změně modelů se nazývá Model Driven Engineering (MDE). Ačkoliv technika MDE staví do popředí vývoje model, nemůže použít všechny modely. Modely využívané v rámci MDE musí podle [4] musí být zapsaný pomocí dobře definovaného jazyka. **Dobře definovaný jazyk** je takový, který má jednoznačně definovanou strukturu (syntaxi) a význam (sémantiku) a je strojově zpracovatelný.

S MDE jsou spojeny pojmy Model Driven Development (MDD) a Model Driven Architecture (MDA). Vývoj MDA byl započat roku 2001 společností OMG. MDA, stejně jako MDD jsou registrované ochrané známky OMG. Podstatou MDA je vytvoření doménových modelů a jejich následné transformace bez závislostí na použitých algoritmech.

V MDA jsou definovány 4 úrovně modelů - Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) a Implementation Specific Model (ISM).

Computation Independent Model je výpočetně nezávislý model. Tento model je někdy nazýván Domain model, protože popisuje doménu problému pomocí doménového slovníku. Doménový slovník usnadňuje komunikaci mezi analytiky zabývajícími se návrhem a experty domény problému. Tento model je v MDA nástrojích opomíjen, jeho síla užitečnost není

v generování kódu.

Platform Independent model je platformě nezávislý model. V rámci MDA rozumíme platformou běhové prostředí, na kterém má náš systém běžet nebo technologii použité k jeho implementaci. Díky platformní nezávistlosti je pro tento model zaručena znovupoužitelnost při změně platformy nebo vývoji podobného projektu.

Platform Specific Model obsahuje informace potřebné k implementaci systému na dané platformě.

Implementation Specific model je potom popis (specifikace) systému v zdrojovém kódu.

Přechody mezi jednotlivými modely jsou nazývány transformace. Transformace Computation Independent Modelu do Platform Independent Modelu není většinou automatizovaná, často ji provádí člověk - softwarový architekt. Výhodou použití této architektury je snadná změna platformy, v které bude daný informační systém implementován. Je možné změnit jenom transformaci, ale zachovat PIM a poté díky transformaci získat PSM.

V průběhu vývoje se nemění jenom vstupní modely, občas je nutné modifikovat i transformace.

Kapitola 2

Použité technologie

2.1 Eclipse IDE

V této bakalářské práci byl použit Eclipse modeling framework sloužící k práci s modely a generování kódu, tento framework je doinstalován jako plugin do IDE Eclipse Helios. Používané formáty v rámci EMF jsou Ecore a Emfatic, formáty používané k uchovávání modelů. Oba dva jsou navzájem převoditelné, lze generovat z jednoho druhý.

Ecore je uložený na disku ve formě XMI (XML Metadata Interchange) souboru. XMI je standard OMG určený k výměně metadat přes XML viz [10]. **Metadata** jsou data popisující jiná data. Typický příklad metadat jsou kartotéky knih v knihovnách obsahující atributy jako je název knihy, autora, počet stran pro každou jednotlivou knihu. Popis obsahu webových stránek pomocí metatagů. XMI do sebe začleňuje 4 standardy:

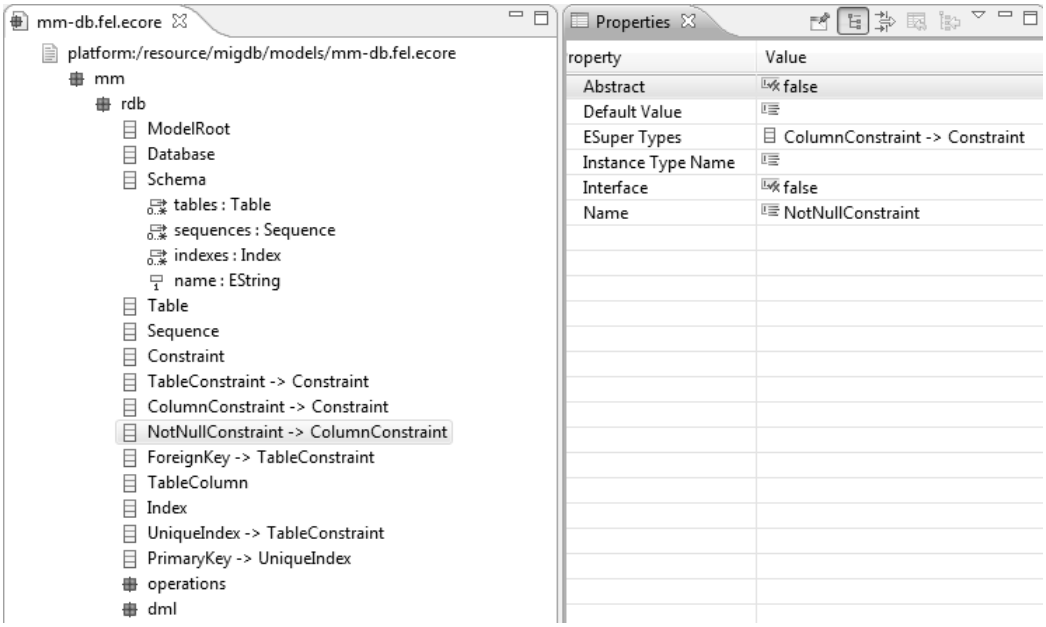
- XML - Extensible Markup Language, standard W3C konsorcia
- UML - Unified Modeling Language, modelovací standard organizace OMG
- MOF - Meta Object Facility, standard organizace OMG určený k specifikaci metamodelů.
- MOF Mapování do XMI

Ukázka XMI :

```
<eClassifiers xsi:type="ecore:EClass" name="Operation" abstract="true"/>
   <eClassifiers xsi:type="ecore:EClass" name="ComposedTableOperation"
   eSuperTypes="#//rdb/operations/Operation">
     <eStructuralFeatures xsi:type="ecore:EReference" name="operations"
     lowerBound="1" upperBound="-1"
     eType="#//rdb/operations/TableOperation" containment="true"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="TableOperation"
abstract="true" eSuperTypes="#//rdb/operations/Operation">
     <eStructuralFeatures xsi:type="ecore:EAttribute"
```

```
        name="tableName" lowerBound="1" eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="CreateTable"
    eSuperTypes="#//rdb/operations/TableOperation">
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="columnName"
        lowerBound="1" upperBound="-1" eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="columnType"
        lowerBound="1" upperBound="-1" eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore#//EString"/>
        <eStructuralFeatures xsi:type="ecore:EAttribute" name="primaryKeyName"
        lowerBound="1" eType="ecore:EDataType
        http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    </eClassifiers>
```

Jak vidno, XMI není určen pro textovou editaci, "výřečnost"formátu je dána následnictvím XML. Ecore je vhodný k základnímu seznámení se s modelovacími prostředky. Není nutné se učit novou syntaxi, není obtížné editovat ho v integrovaném prostředí pomocí myši a klávesnice vyplňováním předpřipravených atributů viz obr. 2.1. Nicméně jeho textová podoba není uzpůsobená pro rychlou editaci. Není možné si v něm zapisovat komentáře.



Obrázek 2.2: Ukázka Ecore v vývojovém prostředí Eclipse

Emfatic byl vyvinut pro rychlou editaci modelu. Jeho výhodou je snadná textová editace. Soubory emf jsou oproti XMI mnohem méně rozsáhlé. Mírnou nevýhodou je nutnost naučit se syntaxi emf, ale ta je podobná Javě, jednomu z nejpopulárnějších jazyků této doby.

O problematice emf pojednává [1].

Následující fragment kódu ilustruje formát emfatic :

```
    abstract class Operation {
    }

    class ComposedTableOperation  extends Operation {
        ordered val operations.Operation[1..*] operations;
    }

    abstract class TableOperation extends Operation {
        attr String[1] tableName;
}


    class CreateTable extends TableOperation {
        attr String[+] columnName;
        attr String[+] columnType;
        attr String[1] primaryKeyName;
    }
```

Kód zachycuje část databázového metamodelu a vyjadřuje stejnou informaci, která byla zachycena pomocí XMI. Pokud bysme přidali třídám metodu (například perform()) realizovali bysme návrhový vzor Composite nazývaný též Tree - abstraktní třída Operation má potomky ComposedOperation a abstraktní TableOperation. ComposedOperation obsahuje další operace. Stejně jako Java, i emfatic obsahuje některá klíčová slova **class**, **abstract**, **extends** a **String** se stejným významem. Oproti Javě ale také obsahuje klíčkové slovo **attr**, které v Javě nenalezneme a označuje atributy uvnitř třídy se stejným vztahem vůči vlastnící třídě jako stejný kód v Javě bez tohoto klíčového slova. Čísla a znaky v hranatých závorkách nám ukazují možné multiplicity obsažených elementů viz tabulka 2.3. Dolní a horní mez jsou obsaženy jako atributy v formátu ecore. Každý atribut se může vyskytovat v libovolné hodnotě v rozmezí mezi dolní mezí a horní. Atributy bez udané hodnoty mají implicitní hodnotu 0 .. 1, tudíž se mohou či nemusejí u instance dané třídy vyskytovat.

Hodnota	Dolní mez	Horní mez
bez hodnoty	0	1
[?]	0	1
[]	0	nekonečno (-1)
[∗]	0	nekonečno (-1)
[+]	1	nekonečno (-1)
[1]	1	nekonečno (-1)
[n]	n	n
[0..4]	0	4
[m..n]	m	n
[5..∗]	5	nekonečno (-1)
[1..?]	1	nespecifikováno (-2)

Obrázek 2.3: Tabulka znaků možných multiplicit převedená na skutečné hodnoty atributů lowerBound a upperBound

Emfatic kód

```
class Table{
    attr String[1] tableName;
}
```

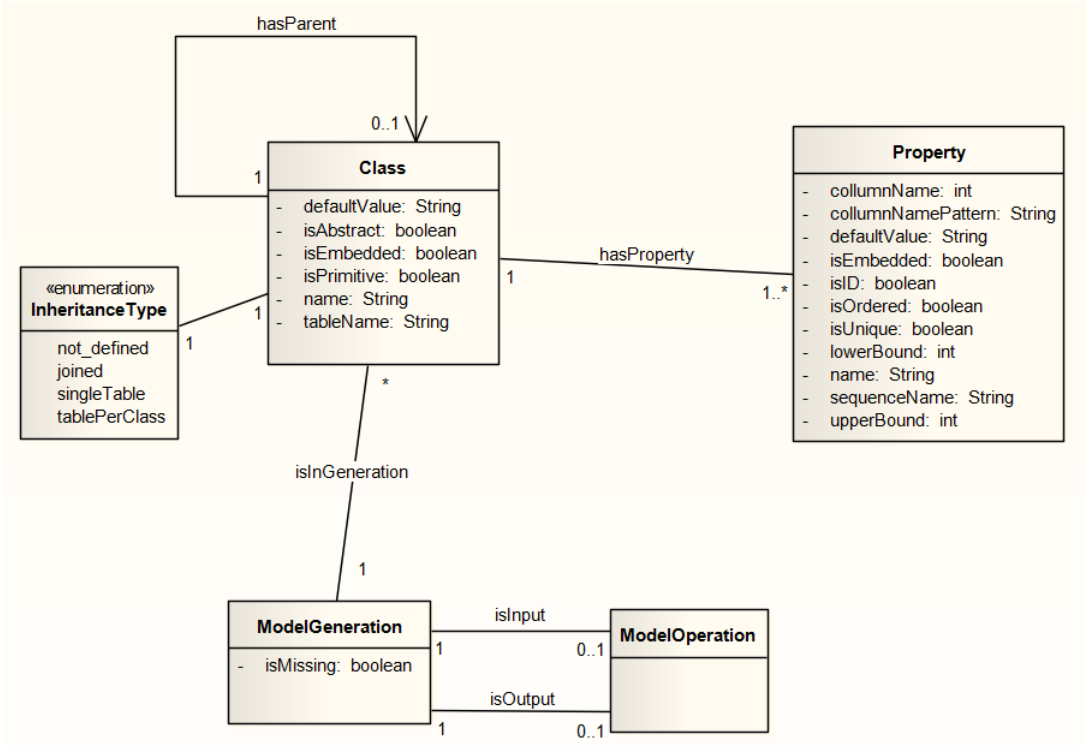
je ekvivalentní Javovskému kódu

```
    class Table{
        String tableName;
    }
}
```

Kapitola 3

Použité metamodely

Jak již bylo řečeno, modely, s kterými je v této bakalářské práci operováno jsou objektový aplikační (zkráceně nazývaný aplikační), databázový a SQL metamodel. Všechny jejich metamodely jsou postaveny pomocí technologie Emfatic a najdete je na přiloženém médiu, kde si je můžete prohlédnout i v Ecore. Aplikační metamodel je zachycen na obr 3.1. V transformaci je použit dále i SQL metamodel, který je ale velmi jednoduchý - obsahuje jednu třídu - Statement s atributem sqlCode typu String.



Obrázek 3.2: Obrázek objektového metamodelu aplikace

3.1 Aplikační metamodel

V aplikačním modelu existují třídy ModelGeneration, Class, Property, ModelOperation a její potomci. Třída Class obsahuje atributy:

- name
- tableName
- isPrimitive
- isEmbedded
- isAbstract

Atribut name určuje jméno třídy, atribut tableName budí zdání duplicity, nicméně je potřebný zejména pro třídy s jménem shodným s klíčovými slovy jazyka SQL. Třídy s atributem isPrimitive rovným true definují primitivní typy databáze jako je například Integer. Atribut isEmbedded ukazuje na fakt, že data instancí třídy budou uložena spolu s daty jiné třídy.

Property obsahuje atributy:

- name
- isEmbedded
- isOrdered
- isUnique
- lowerBound
- upperBound
- sequenceName
- upperBound
- collumnName
- defaultValue
- isID

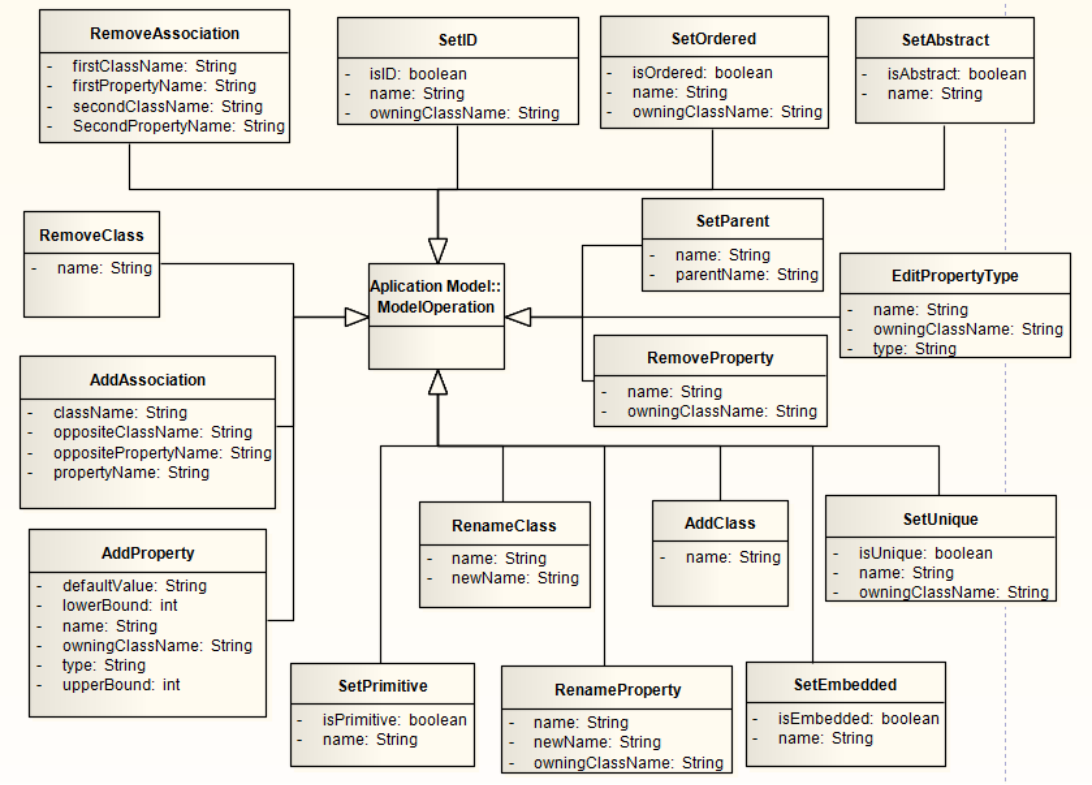
Atribut jméno identifikuje sloupec, atributy collumnName a collumnNamePattern jsou potřebné pro SQL klíčová slova. Atribut isEmbedded určuje, že třída má embedded Property. Atributy lowerBound a upperBound určují násobnost Property. Pokud je lowerBound 0, je Property volitelná. UpperBound vyšší, jak 1 signalizuje kolekci. Atributy isOrdered a isUnique se vážou na Property v kolekcích, isOrdered značí seřazené prvky, isUnique unikátní prvky v kolekci. SequenceName je použito v případě, že Property má být generována pomocí nějaké sequence, toto použití omezuje typ Property na celočíselné typy.

DefaultValue určuje implicitní hodnotu. Booleovský isID značí, zdali je property primárním klíčem či ne.

Význam atributu isMissing třídy ModelGeneration je spojen s evolucí a bude vysvětlen později.

Enumerace InheritanceType určuje, jakým způsobem bude třída v hierarchii dědičnosti uložená v databázi. Více o jednotlivých druzích InheritanceType bude řečeno v kapitole transformace.

Abstraktní třída ModelOperation zastupuje všechny uskutečnitelné operace s daným modelem. Každá operace má svůj vstupní model a model výstupní. Vstupní zaznamenává stav aplikace předq provedením operace, výstupní stav po provedení. Seznam všech zatím popsaných operací v aplikačním modelu je na obr 3.3. Tento seznam zahrnuje základní operace potřebné k vytvoření a modifikování modelů. Více o operacích bude řečeno v kapitole transformace.



Obrázek 3.4: Potomci třídy ModelOperation, doplnění aplikačního metamodelu

3.2 Databázový metamodel

Metamodel databáze je na obr 3.5. V databázovém metamodelu existují entity Schema, Database, Sequence, Table, TableColumn, Constraint, TableConstraint, Index, ColumnConstraint, ForeignKey, PrimaryKey a Operation. Většina těchto entit je pro čtenáře jistě známá,

ale některá rozhodnutí nebyla udělána v souladu s SQL standardem. Tato sekce je popisem databázové domény.

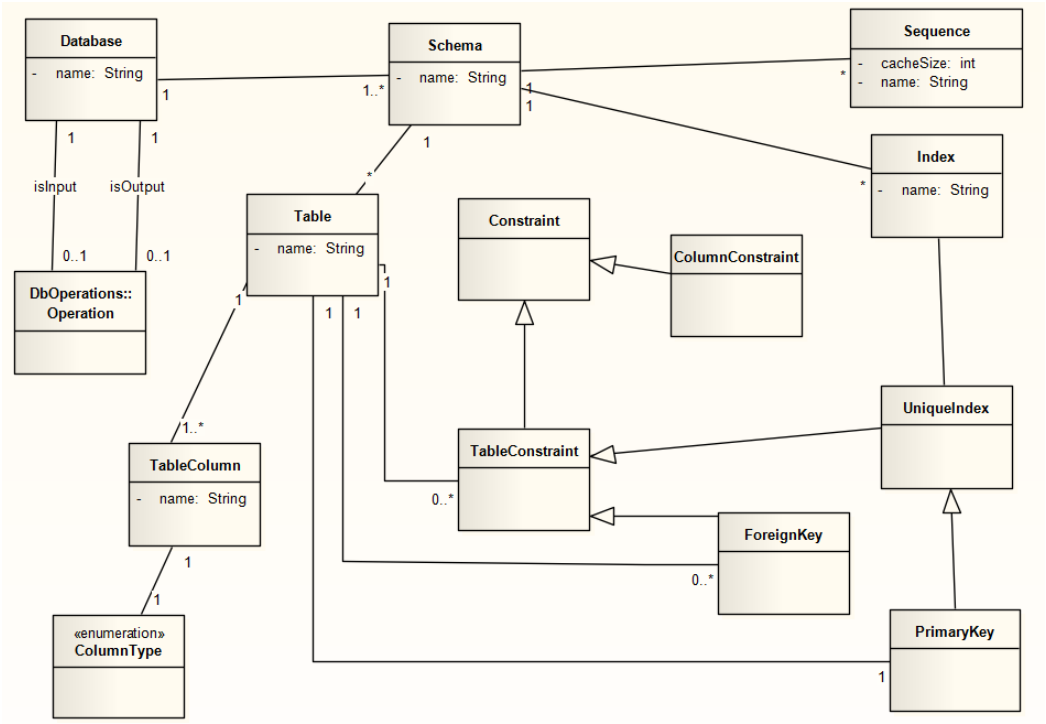
Entita **schema** nereferuje na databázové schéma jakožto strukturu databáze, ale definuje nám "namespace", ve kterém jsou uložené tabulky. **Databáze** nám reprezentuje jednotlivé databáze v významu úložiště dat nikoliv DBMS. **Sequence** je technologie určená pro generování číselných hodnot. Její atribut cacheSize určuje, kolik čísel má být připraveno v paměti k použití. Jako **table**, česky tabulka, je v relační databázi označována struktura, do níž ukládáme data. Každý řádek tabulky reprezentuje instanci nebo skupinu dat svázanou definicí tabulky. Každá tabulka musí k své existenci obsahovat minimálně jeden **TableColumn** - sloupec. Sloupec definuje jméno a typ dat, které se do něj dají uložit. Sloupce nám definují strukturu tabulky, které každý řádek musí splňovat. Pokud vkládáme do nějaké tabulky data, musíme použít SQL příkaz INSERT s typem dat definovaným pomocí jejich sloupců, abychom vložili do tabulky řádek.

Index napomáhá k zrychlení prohledávání tabulky, za tuto výhodu je nutné zaplatit zpomalením při zapisování do tabulky a zvětšením potřebného místa pro uložení. Indexování je mechanismus, který seřadí data podle jednotlivých sloupců. Tabulka se dá dále ovlivňovat pomocí jejich **Constraints** - podmínek, které musí platit. Známe dva druhy Constraints - **TableConstraints** a **ColumnConstraints**. Ačkoliv většinu Constraints je možné definovat jako TableConstraint i jako ColumnConstraint, jsou v této bakalářské práci rozděleny Constrainty jednoznačně. Do TableConstraints jsou zařazeny Constraints, které můžeme pomocí jazyka SQL vytvářet i jako ColumnConstraints (aspoň v dialektu PostgreSQL), patří mezi ně ForeignKey, UniqueIndex a jeho PrimaryKey. Constraint **UniqueIndex** nám zaručuje unikátnost záznamů v daném sloupci. Jeho potomek **PrimaryKey** má tu samou vlastnost jako UniqueIndex, nicméně může být v tabulce definován jen jednou (či vůbec) a je to feature, která nám pomáhá jednoznačně identifikovat data. **ForeignKey** zabezpečuje hodnotu daného sloupce rovnu hodnotě primárního klíče z jiné tabulky, na kterou referuje. Mezi ColumnConstrainty patří NotNull Constraint, zabraňující vkládání NULL hodnot do sloupce. Dále by do této skupiny měla patřit i CheckConstraint, avšak její struktura této entity nebyla zatím definovaná v rámci tohoto projektu.

Stejně jako aplikační metamodel je rozšířen o sadu možných změn aplikace, je i databázový metamodel rozšířen o sadu možných změn databáze. **Operation** je ekvivalent operacím typu ModelOperation, ale nad databázovou doménou. Podobně jako ModelOperation má vstupní a výstupní model, má i Operace vstupní a výstupní databázi. Hlavním rozdílem oproti operacím z aplikačního metamodelu je existence ComposedOperation, která sdružuje více databázových operací do jedné.

3.3 Evoluce modelů

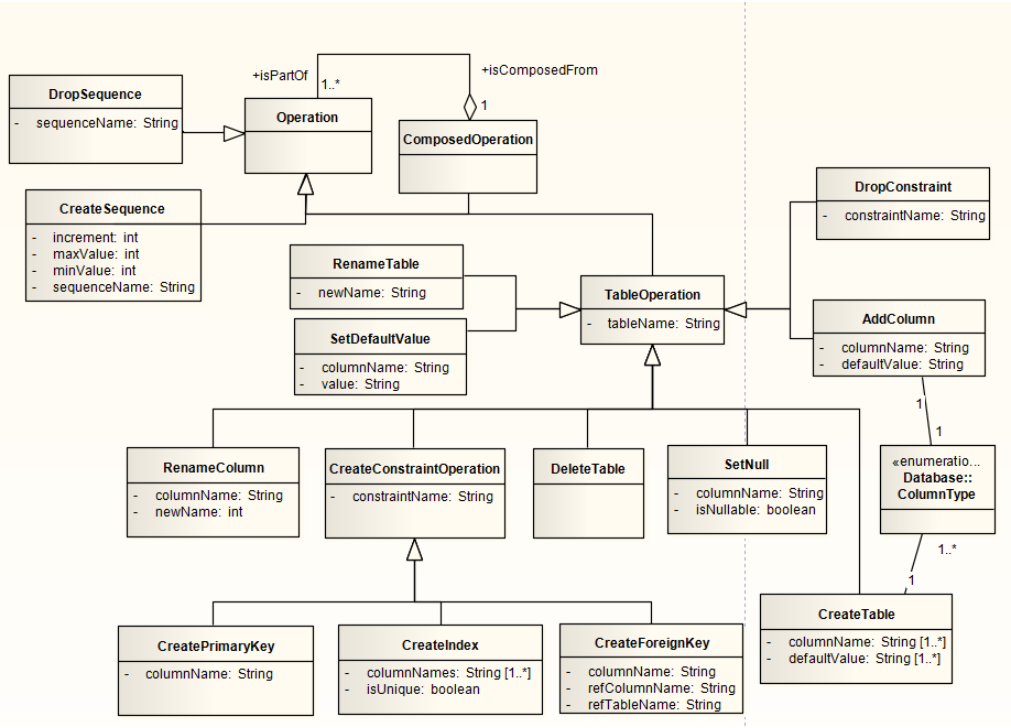
Jak bylo popsáno v kapitole ORM, objektový model aplikace a databázové schéma spolu úzce souvisí. Pro každou verzi aplikačního modelu nalezneme její obraz popsaný pomocí metamodelu databázového. Vývoj aplikačního modelu nazýváme slovem **evoluce**.



Obrázek 3.6: Metamodel databáze

V průběhu vývoje může vývojář přijít na to, že aplikace není kompletní či není korektní. V nějaké třídě může chybět atribut, je nutné ho doplnit. Dalšími možnými změnami aplikačního modelu je vytvoření nadtypu třídy, extrahování některých atributů třídy do předka... Abychom měli možnost pracovat při modelování i se změnami, museli jsme zahrnout operace do metamodelu. Cílem nebylo vytvoření velkého množství komplikovaných operací, bylo nutné definovat elementární operace. V metamodelu jsou elementární operace definované jako potomci ModelOperation viz obr. 3.3. V první fázi projektu byla transformace definovaná jako přechod mezi jedním Modelem na druhý, v metamodelu byly vstupní a výstupní atributy třídy Model. V průběhu běhu tohoto projektu se ukázalo, že manipulace s takovýmto metamodelem by byla těžkopádná, proto byly inputModel a outputModel odstraněny a nahrazeny kolekcí Modelů. Tato změna umožňuje zaznamenávat historii změn modelu. V metamodelu je definován element **ModelGeneration** jako uspořádaná kolekce po sobě jdoucích Modelů. Kromě prvního a posledního modelu je každý model vstupem jedné operaci a výstupem druhé. První model není výstupem žádné operace a poslední není vstupem, což vynucuje uspořádání operací.

Nemáme definovaný matematický jazyk pro mapování z aplikačního modelu do jiného aplikačního modelu, proto si analyzujeme operace pouze slovně a připíšeme jejich preconditions a postconditions. Vzhledem ke slovnímu popisu si jen uvědomme, že existence daného elementu v precondition je vázána na vstupní generaci, existence daného elementu v postcondition potom na generaci výstupní. Operace, které byly definovány s jejich vstupními



Obrázek 3.8: Potomci třídy operation kompletující databázový metamodel

a výstupními podmínkami:

Operace pro vytvoření třídy
vstupní podmínky: class se stejným jménem v systému nesmí existovat
výstupní podmínky: existence třídy s jedním sloupcem id typu String

Nastavení třídy jako abstraktní
vstupní podmínky: třída musí existovat
výstupní podmínky: nastavení abstrakce

Nastavení atributu třídy isEmbedded na danou hodnotu
vstupní podmínky: třída musí existovat
výstupní podmínky: nastavení správné hodnoty isEmbedded

Nastavení předka třídy
vstupní podmínky: třída musí existovat $\wedge$ musí existovat předek $\wedge$ předek nesmí být potomek třídy className
výstupní podmínky: nastavení předka

Nastavení třídy jako primitivní
vstupní podmínky: třída s className musí existovat

výstupní podmínky: nastavení vlastnosti setPrimitive na danou hodnotu

Přidání property do třídy
vstupní podmínky: musí existovat třída s name owningClassName ∧ nesmí v ní existovat daná property
výstupní podmínky: existence property s name name v třídě s name owningClassName

Změna typu property
vstupní podmínky: musí existovat třída s owningClassName ∧ musí existovat property s propertyName v rámci dané třídy ∧ musí existovat třída
s name type
výstupní podmínky: změna typu property

Nastavení ID vstupní podmínky: musí existovat třída s name owningClassName, musí existovat property s name name
výstupní podmínky: nastavení hodnoty isID

Přejmenování třídy
vstupní podmínky: musí existovat třída s name className ∧ třída s jménem name nesmí ještě existovat
výstupní podmínky: přejmenování třídy

Přejmenování property
vstupní podmínky: musí existovat třída s className ∧ property s daným jménem ještě nesmí existovat v rámci třídy className ∧ musí existovat třída
výstupní podmínky: změna name property na novou hodnotu

Smazání třídy
vstupní podmínky: musí existovat třída s className ∧ třída s className nesmí mít potomka ∧ třída nesmí mít žádnou vazbu na jiné třídy ∧
pokud je třída isPrimitive, nesmí existovat žádné Property jejího typu
výstupní podmínky: smazání třídy

Smazání property
vstupní podmínky: musí existovat třída s className ∧ musí existovat property v třídě className ∧ property nesmí být součástí asociace
výstupní podmínky: smazání property

Smazání asociace
vstupní podmínky: asociace musí exitovat
výstupní podmínky: smazání asociace

Nastavení atributu isOrdered
vstupní podmínky: třída s name owningClassName musí existovat ∧ property s name name

musí existovat v rámci dané třídy
výstupní podmínky: nastavení hodnoty isOrdered na správnou hodnotu

Nastavení hodnoty isUnique
vstupní podmínky: třída s name owningClassName musí existovat ∧ property s name name musí existovat v rámci třídy owningClassName
výstupní podmínky: je nastavena příslušná hodnota

Přidání asociace
vstupní podmínky: třída s name firstClassName musí existovat ∧ třída s name secondClassName musí existovat
výstupní podmínky: asociace je vytvořena

Některé z operací jsou typickými settery známými z objektově orientovaných programovacích jazyků. Naopak některé z těchto operací nemají za úkol jen měnit hodnotu, takže třeba operace remove jsou složitější než operace set. Tento fakt si můžeme ověřit analýzou problematických stavů, které mohou operace způsobit. Problém u odstraňovacích operací je ten, že musíme definovat, co se stane, jakmile bude například třída uprostřed hierarchie dědičnosti odstraněna. Třída by mohla být odstraněna, její sloupce přesunuty do tříd potomků, případně by mohlo být vytvořeno id, pokud je třída kořenem dědičné hierarchie. Situací, které nejsou jednoduché a bylo by nutné ošetřit speciálním způsobem je mnoho, každá z nich vyžaduje jiné řešení a tyto povinnosti bysme jako programátoři neměli nechat na samotném mapování.

Nejjednodušším způsobem, jak těmto problémům předejít, je zamezit takovému případu a nechat na člověku či programu, ať zajistí před použitím operace splnění potřebných preconditions například odstraněním třídy z dané hierarchie. Jednotlivé kroky evoluce symbolizované potomky ModelOperation mají své obrazy v Databázovém metamodelu, nicméně vzhledem k nedostačující specifikaci některých operací a nedořešení některých problémů nebyla specifikována zatím transformace ModelOperation => Operation tj. transformace operací z aplikačního modelu do modelu databázového.

Zástupce problémů ve specifikaci je nastavení třídy jako primitive, pokud má nějaké property. Nastavení setID na hodnotu false či vytvoření nové třídy může mít za následek nekonzistentní stav třídy - třída mimo hierarchii dědičnosti bez ID property. Problematické je i vytvoření ID property, protože všechny primitivní typy musí být součástí modelu, tudíž je otázkou, jak zvolit defaultní nastavení typu Property.

Kapitola 4

Transformace

Přechod od některých entit k ekvivalentům v databázi znamená občas jen změnu názvu (např. Class => Table, Property => Column), v některých případech v databázovém modelu vzniknou nové entity, které nemají v modelu aplikačním žádný vzor a jsou charakteristické jen pro databáze (např. ForeignKey, Indexy).

Například Class Person, která nemá přímou nadtřídu se do databáze namapuje jako Table Person a Property name se namapuje jako Column name. Naopak entita Index je v databázovém modelu vytvořena jako důsledek vytvoření primárního klíče tabulky.

Transformace aplikačního modelu můžeme popsat transformačními pravidly, která jsou zaznamenána v souborech InheritanceType_trid.pdf, mm2db-mapping-spec.pdf a vazby.pdf v složce dokumenty na přiloženém médiu.

V následujícím textu budem používat tyto pojmy:

E . . . Množina všech možných objektů vstupního modelu (instance tříd aplikačního metamodelu)

Libovolnou podmnožinu $\mathbb{M} \subset \mathbb{E}$ nazýváme *model*

$\mathbb{D}$ . . . Množina všech možných objektů výstupního modelu (instance tříd z metamodelu popisující db schéma)

Libovolnou podmnožinu $\mathbb{S} \subset \mathbb{D}$ nazýváme *schéma*

$\mathbb{P}$(X) ... potenční množina množiny X (power set, množina všech podmnožin)

Funkce tvaru $\pi : \mathbb{E} \to \mathbb{P}(\mathbb{S})$ nazýváme *mapovacími funkcemi*. Mapovací funkce zobrazuje element vstupního modelu na množinu elementů výstupního modelu a je základním prvkem při konstrukci složitějších transformací. Množinu všech takových funkcí značíme $\Pi = \mathbb{E} \to \mathbb{P}(\mathbb{S})$.

Množinu všech mapovacích pravidel značíme $\Omega = \Pi \times E \to \{0, 1\}$.

Uspořádanou dvojici $\omega = [\pi; g] : \omega \in \Omega$ nazýváme *mapovacím pravidlem*, funkci g pak *guardem* mapovacího pravidla. Guard má pro mapovací pravidlo význam podmínky, která musí být splněna, aby bylo pravidlo aplikováno, tedy aby byla užita mapovací funkce π. Lépe patrné je to z definice obrazu elementů níze.

Obraz elementu e $\in$ E pro pravidlo $\in \Omega = [\pi; g]$ definujeme jako

$$db(e, \omega) : \mathbb{E} \times \Omega \to \mathbb{P}(\mathbb{S}) = \begin{cases} g(e) = 0 \Rightarrow \emptyset \\ g(e) = 1 \Rightarrow \pi(e) \end{cases}$$

Libovolnou množinu mapovacích pravidel nazýváme mapováním (obvykle dále značena jako λ).
Množinu všech mapování označujeme jako $\Lambda = \mathbb{P}(\Omega))$. Mapování sdružuje jednoduché mapovací funkce/pravidla a umožňuje popsat složitější transformaci. Existence termínu mapování usnadňuje analýzu vlastnosti sady mapovacích pravidel jako jsou úplnost a nekonfliktnost. Úplnost sady mapovacích pravidel zajišťuje, že nenastane situace, která by již nebyla popsatelná jedním mapovacím pravidlem či nějakou jejich kombinací. Nekonfliktnost dvou mapovacích pravidel A a B nám zajišťuje, že průběh mapovacího pravidla A neovlivní průběh mapovacího pravidla B a naopak. Nekonfliktnost nám zajišťuje, že pokud v sekvenci můžeme libovolně zaměňovat pořadí provedení sady mapovacích pravidel.

Mírnýmm rozšířením definujme obraz elementu $e \in \mathbb{E}$ při mapování $\lambda \in \Lambda$ jako:

$$db(e, \lambda) : E \times \Lambda \to \mathbb{P}(D) = \bigcup_{p \in \lambda} (e, p).$$

Zcela analogicky definujme ještě *obraz modelu* $M \in \mathbb{P}(E)$ při mapování $\lambda \in \Lambda$ takto:

$$db(M, \lambda) : E \times \Lambda \to \mathbb{P}(D) = \bigcup_{w \in \lambda, einE} (e, \omega) = \bigcup_{einE} (e, \lambda)$$

Model je množina, jejíž prvek E bude pomocí mapovacího funkce transformován na množinu prvků *schématu*. Model reprezentuje v textu abstrakci instance aplikačního metamodelu a schéma potom instanci metamodelu databázového. Element ze vstupního modelu se po aplikaci mapovací funkce projeví ve schématu jako *obraz*. *Mapovací* pravidlo je uspořádaná dvojice mapovacího funkce a guardu. *Guard* je podmínka, testující, jestli se pro daný vstupní element E provede mapovací pravidlo. Při úspěšném splnění guardu bude výstupem mapovacího pravidla element 1 a mapovací funkce se uskuteční. Při neúspěšném splnění guardu bude výstupem mapovacího pravidla 0 a mapovací funkce se neuskuteční.

Smyslem mapovacích funkcí a pravidel je možnost na elementární úrovni popsat, které databázové objekty jsou třeba pro serializaci nějakého elementu ze vstupního aplikačního modelu. Různé mapovací funkce mohou vracet různé výstupní objekty a realizovat tak různá dílčí mapování.

Pro ilustraci tohoto postupu uvedu následující příklady:

poznámky: K[*Element*] ... kolekce elemenů typu E

*E ... tento symbol byl zaveden pro elementy, které jsou potřebné pro úspěšné mapování, nicméně nevzniká z vstupního elementu modelu

tečková notace je použita k přiřazení atributů jednotlivým elementům

class.precedessor(i), $i \in \mathbb{N}$ je třída nacházející se o i úrovní výše v stromu dědičnosti, tzn. dědí od ní třída, na které je testován guard
Matematicky se tento vztah dá vyjádřit pomocí rekurze:
class.precedessor(1) = class.parent
class.precedessor(i) = class.precedessor(i - 1).parent

root - index kořenu podstromu hierarchického stromu - nemá předka, root definuje hloubku třídy v podstromu (třída nemusí být list)
Matematicky vyjádřeno:
$root(class) \in \mathbb{N}$: class.precedessor(root(class)).parent = NULL

Slovem resolve je určeno hledání obrazu elementu daného typu, takže tato podmínka

$$g_4 : parent.resolve(Table) = true$$

zajišťuje namapování parenta třídy na element typu tabulka před provedením mapovací funkce nebo zapříčiní neprovedení mapofací funkce

- *Pravidlo 1*
  $\pi_0 : class \rightarrow \{K[column]\}$
  $g_0 : class.isPrimitive = true$
  $class \in Class, column \in Column$
  $\omega_1(\pi_0, g_0)$
  Pozn: Třída, která je primitivní se nenamapuje do databázového modelu na Tabulku. Tyto tridy se namapují na sloupce v databázi daného typu.

- *Pravidlo 2*
  $g_1 : class.isPrimitive = false$
  $g_2 : \exists!\ i \in \mathbb{N} : class.properties[i].isID = true$
  $g_3 : class.parent = NULL$
  $\omega_1(\pi_1, g_1 - g_3)\pi_1 : class \rightarrow \{table, K[Column], primaryKey\}$ *Pozn:* Platnost prvních dvou guardů - neprimitivnost a existence právě jednoho primárního klíče jsou nutné pro namapování jakékoliv třídy na tabulku, celé mapování namapuje třídu bez předka - tudíž třídu mimo hierarchii dědičnosti na nově vzniklou tabulku, množinu jejich sloupců a primární klíč

- *Pravidlo 3*
  $g_4 : class.parent.resolve(Table) = true$

$g_5 : class.inheritanceType = "\_\_not\_defined" \wedge$
$class.parent \neq NULL \, \forall j \in \mathbb{N} : j \leq root(class) : class.precedessor(j).inheritanceType =$

$"\_\_not\_defined"$
$g_6 : (class.parent \neq NULL) \wedge ((class.inheritanceType = class.parent.InheritanceType \wedge$

$class.inheritanceType = Joined) \ \vee \ class.inheritanceType = NULL$
$\wedge \ \exists i \in \mathbb{N} : \forall j \in \mathbb{N} : j < i : class.precedessor(j).inheritanceType = NULL$
$, class.precedessor(i).inheritanceType = Joined)$
$\omega_3(\pi_2, \ g_1 \wedge g_2 \wedge g_4 \wedge (g_5 \vee g_6))$
$\pi_2 : class \rightarrow \{table, \ K[column], \ *parentPrimaryKeyColumn,$
$foreignKey, \ primaryKey\}$
$class \in Class, \ column \in TableColumn, \ table \in Table, \ primaryKey \in PrimaryKey,$
$foreignKey \in ForeignKey, \ parentPrimaryKeyColumn \in TableColumn$
typ Joined vytvoří novou tabulku, sloupce a referuje na id sloupec

4.1 QVT

QVT je standard OMG, který definuje strukturu a chování svých realizací. Specifikaci QVT nalezneme na stránkách OMG viz [5]. QVT standard je možné ještě dělit na 3 podstandardy - QVT Operational, QVT Core a QVT Relations. Kromě částí standardu existují i jazyky, které nejsou implementované podle specifikace, ale svým charakterem se velmi blíží QVT. Takovýmto jazykem je například jazyk ATL (Atlas Transformation Language), který je dostupný jako plugin do IDE Eclipse.

Jazyk QVT (Query, view, transformation) patří do rodiny Domain specific languages - jazyků určených pro řešení specifických problémů na specifické doméně. Jeho primárním účelem je dotazování, zobrazování a transformace modelů, jak již napovídá název. Transformace psaná v QVT slouží k změně vstupního modelu do modelu výstupního. Každý model musí být popsán svým metamodelem. Vstupní model označujeme klíčovým slovem **in**, výstupní potom klíčovým slovem **out**. Kromě použití dvou různých formátů modelů je možné použít i tzv. **inout** model, tedy aplikovat transformaci na vstupní model, který je po ukončení transformace v novém stavu. Nevýhodou tohoto přístupu je nemožnost opakovat transformaci (za předpokladu, že její provedení modifikovalo model) za stejných podmínek.

Z hlediska zpracovávání příkazů rozdělujeme programovací jazyky na dva hlavní proudy - deklarativní a imperativní jazyky. Při programování v **deklarativním jazyce** se nestaráme o vlastní algoritmus, ale specifikujeme vztah mezi vstupem a požadovaným výstupem, tento přístup má tu výhodu, že je možné pomocí jedné definice transformovat model typu A na model typu B či reverzně transformovat model typu B na model typu A. Deklarativní programování závisí na interpretu jazyka, který provádí algoritmuckou část, při deklarativním programování definujeme množinu funkčních proměnných, nebývá při něm důležité pořadí provádění jednotlivých mapování. Dalšími typickými znaky jsou střídmé používání proměnných a řešení cyklů pomocí rekurze. Zástupci deklarativního programování se často řadí mezi funkcionální jazyky jako například programovací jazyk Haskell. QVT Relations a QVT Core patří do rodiny deklarativních jazyků.

Imperativní jazyky interpretují program jako popis algoritmu, kdy na sebe jednotlivé kroky algoritmu navazují, narozdíl od deklarativních jazyků známe jejich pořadí provádění. Ačkoliv i v imperativním jazyce můžeme napsat program pomocí rekurzivních metod, není výjimkou, že imperativní jazyky používají iterační konstrukce jako jsou cykly for, while, do-while. Imperativní jazyky používají větvení (příkazy switch, if, goto) a přiřazení. Oproti programům napsaným v deklarativním jazyce se v programech napsaných v jazycích imperativních vyskytují častěji chyby. Na druhé straně je pro člověka jednodušší napsat program v imperativním jazyce. Pod pojmem imerativní jazyk si můžeme představit libovolný z dnešních nejpoužívanějších objektově orientovaných jazyků jako je Java, C++, C#.

QVT Operational se řadí mezi imperativní jazyky, součástí QVT Operational jsou i konstrukce z jazyka OCL.

V rámci této práce byl použit Eclipse M2M Operational QVT. QVT operational má kromě implementace Eclipse M2M Operational QVT i implementace SmartQVT, Borland Together. Poslední jmenovaný projekt byl pozastaven a stal se součástí Eclipse M2M Operational QVT.

V kapitole analýza byly uvedeny dva příklady mapovacích pravidel.

Příklad 1 vypadá v QVT trochu jinak než mapovací pravidlo 1 :

```
mapping APP::reduced::Property::toPrimitive( ) : RDB::rdb::TableColumn
//query self.isPrimitive zajišťuje existenci primitivní třídy mezi
//všemi třídami s daným jménem, pro zajištění namapování na všechny
//sloupce je nutné použití //nějakého iteračního cyklu v programu
 when { self.isPrimitive() = true}
{
        name:= self.name;
        type:= self.type.name;
}
```

Příklad 2 převedený do QVT by vypadal takto:

```
//vstupní a vystupní entity pravidla jsou v hlavičce mapování
mapping APP::reduced::Class::toJoinedTable():RDB::rdb::Table
when{ ( self.hasJoinedInheritanceType() or
self.hasImplicitInheritanceType()) and
self.isGeneralClassSerializable() and self.parent <> null }
{
        //obraz třídy předka
        var precedessorTable:RDB::rdb::Table :=
            self.parent.resolveone(RDB::rdb::Table);

        //vytvoření cizího klíče referencujícího precedessorTable -
        //tabulku předka
        var FK:RDB::rdb::ForeignKey = object RDB::rdb::ForeignKey{
```

```
            name := "FK_" + precedessorTable.name;
            targetTable := precedessorTable;
        };
        result.constraints+=FK;

        //je nutné vytvořit sloupec id referencující předka
        var PKcol:RDB::rdb::TableColumn:= object RDB::rdb::TableColumn{
            name:= self.getRootClass().properties->selectOne(prop |
                prop.isID).name;
        };
        PKcol.type := self.getRootClass()->properties->selectOne(prop |
            prop.isID) .type.name;
        result.ownedColumns+= PKcol;
        var pk : RDB::rdb::PrimaryKey := object RDB::rdb::PrimaryKey{
            name:= "pk_"+ self.name;
            underlyingIndex := object RDB::rdb::Index{
                columns+=PKcol;
                name := "pk_index";
            }
        };

result.constraints+= pk;
FK.constrainedColumn := PKcol;
result.name := self.name;
self->properties->asOrderedSet()->xmap toColumns(result);
}
```

Ačkoliv mapovací pravidlo neříká, jak se má implementovat daná transformace vstupního modelu do výstupního, ukazuje vstupní elementy, výstupní a elementy, které jsou již vytvořeny a budou potřeba k úspěšnému namapování. Mapovací pravidlo je vlastně takový interface mapování.

Guardy mapování g_1, g_2, g_5, g_6 jsou zobrazeny v klauzuli when. Querry isGeneralClassSerializable sdružuje g_1 a g_2, g_5 a g_6 odpovídají hasImplicitInheritanceType() a isGeneralClassSerializable(). Query g_4 je kontrolované mimo vlastní transformaci.

Pokud zapřemýšlíme nad řádkem mapování

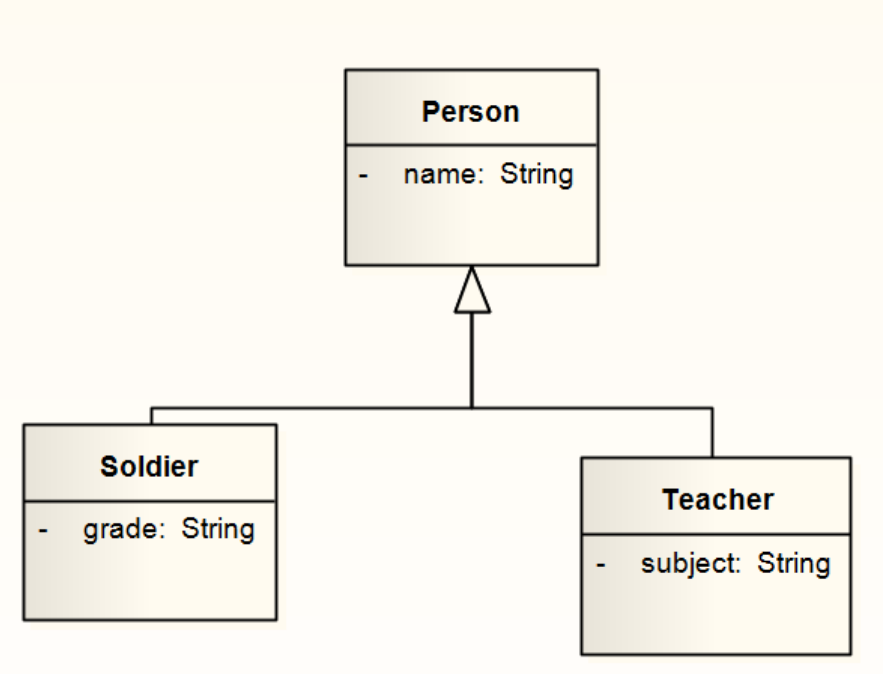
$$\pi_2 \ : \ class \ \rightarrow \quad \{ \ table, \ K[column], \ *parentPrimaryKeyColumn, \ foreignKey, \ primaryKey \ \}$$

tak si uvědomíme, že vstupní element class je v QVT jasně viditelný. Nicméně výstupní element QVT transformace je jen Table z package RDB::rdb. Tělo transformace nám ukazuje vznik zbylých výstupních instancí. var pk : RDB::rdb::PrimaryKey je deklarace proměnné, klíčové slovo **object** nás upozorňuje na vznik nové instance. Řádek obsahující klíčové slovo resolveOne potom ukazuje nalezení první instance typu Table, na kterou se namapovala pa-

rent instance mapovaného objektu.

Pro ilustraci těchto typů si představme jednoduchou hierarchii dědičnosti zobrazenou na obrázku 4.1. Třída Person má dva potomky, třídu Soldier a třídu Teacher. Třída person obsahuje atribut name, který je identifikátorem každé instance. Dále třída Soldier obsahuje atribut grade - nejvyšší dosaženou hodnost vojáka a třída Teacher obsahuje atribut subject reprezentující předmět, který vyučuje. Pomiňme nedokonalost modelované skutečnosti, fakt, že každý učitel vyučuje pouze jeden předmět není podstatný, tento příklad je pouze ilustrativní a pro tento účel postačující.

Kromě zmíněné transformace byl napsán také prototyp aplikující operace na danou generaci, a která je obsažena v souboru populate_generations.qvto.



Obrázek 4.2: Ukázková dědičné hierarchie tříd obsahující třídy Person, Soldier a Teacher

Kapitola 5

Realizace transformací

V rámci tohoto projektu byla vytvořena transformace realizující ORM mapování tříd a jednoduchých dědičných hierarchií. Jak již bylo naznačeno v sekci o metamodelech 2.1, transformace realizující mapování třídy bez potomků není složitá. Takové mapování vytvoří tabulku v modelu databáze se stejným jménem, dále vytvoří primaryKey a namapuje Property dané třídy na sloupce. Jakmile je třída umístěna do dědického stromu hierarchie, je situace složitější. Není jeden způsob uložení hierarchie do databáze, jsou tři - SingleTable, Joined, TablePerClass.

Nejjednodušším InheritanceType je TablePerClass, tento typ vybere všechny atributy z třídy a všech jejích předků a vytvoří pro mapovanou třídu novou tabulku. Výhodou tohoto typu je, že všechny instance konkrétní třídy nalezneme v jedné tabulce. Nevýhodou je potom hledání všech instancí nadtřídy zahrnující i její potomky. Toto mapování je výhodné zvláště, pokud používáme hierarchii, kde se dotazujeme na instance konkrétní či listové třídy (třídy, které v hierarchickém stromu nemají žádného potomka).

Druhým typem je typ SingleTable, který namapuje všechny třídy do jedné tabulky. Aby byly v této tabulce odlišitelné řádky reprezentující jednotlivé instance tříd, je nutné vytvořit tzv. diskriminační sloupec, který obsahuje řetězec identifikující typ. Výhodou tohoto typu je vytvoření jedné tabulky. Nevýhodou širokého hierarchického stromu je vznik velké tabulky naplňované řádky s mnoha hodnotami NULL.

Třetím typem je typ Joined, jenž je podobný typu TablePerClass. Stejně jako TablePerClass vytvoří novou tabulku pro každou třídu v dědické hierarchii. Nicméně, každá nově vytvořená tabulka bude obsahovat jen sloupce vytvořené mapováním atributů třídy, jež je mapovaná, nikoliv jejích předků. Navíc tato tabulka bude obsahovat id sloupec z s hodnotou obsaženou v tabulce namapované z třídy předka v hierarchii (třídy bez nadtřídy) a cizí klíč na tento sloupec. Tento typ je defaultní, mapování je uskutečněno za jeho pomoci při neuvedení žádného typu - v metamodelu je tato situace znázorněna pomocí Enumeration __not_defined.

Pro ilustraci těchto typů si představme jednoduchou hierarchii dědičnosti zobrazenou na obrázku 4.1. Třída Person má dva potomky, třídu Soldier a třídu Teacher. Třída person obsahuje atribut name, který je identifikátorem každé instance. Z pohledu databázového to není

úplně korektní, lepším řešením by bylo rozložení name na firstName a surName Dále třída Soldier obsahuje atribut grade - nejvyšší dosaženou hodnost vojáka a třída Teacher obsahuje atribut subject reprezentující předmět, který vyučuje. Pomiňme nedokonalost modelované skutečnosti, fakt, že každý učitel vyučuje pouze jeden předmět není podstatný, tento příklad je pouze ilustrativní a pro tento účel postačující. Představme si dále, že máme tři instance dat - pro každou třídu jednu. Pro třídu Soldier instanci s id Ján Matůška s atributem grade rovným svobodník. Pro třídu Teacher instanci s id Josef Lobotka s "Ekonomií"jako atribut subject. A nakonec pro třídu Person mějme instanci s id Martin Lukeš.

name	grade	subject	discriminator
Josef Lobotka	NULL	Ekonomie	Teacher
Ján Matůška	svobodník	NULL	Soldier
Martin Lukeš	NULL	NULL	Person

Tabuka Person

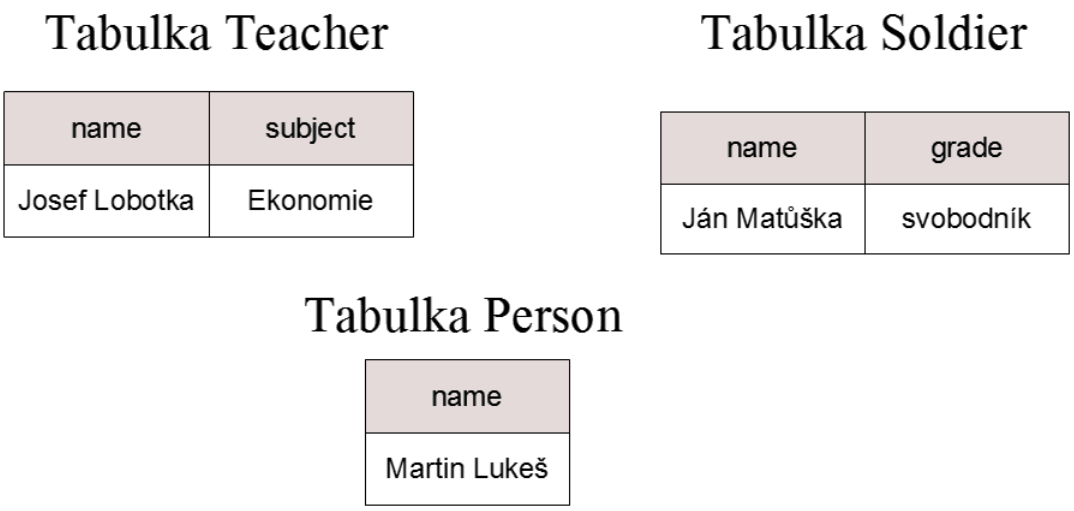
Obrázek 5.2: Výsledek mapování při použití InheritanceTypu SingleTable

Na obrázku 5.1 vidíme výsledek mapování při použití aplikace transformace na hierarchii s InheritanceTypem SingleTable. Všimněme si NULL hodnot. I takto jednoduchá tabulka má počet NULL hodnot vůči počtu instancí dat v poměru 1:1. Diskriminační sloupec je zde neodstranitelnou částí.

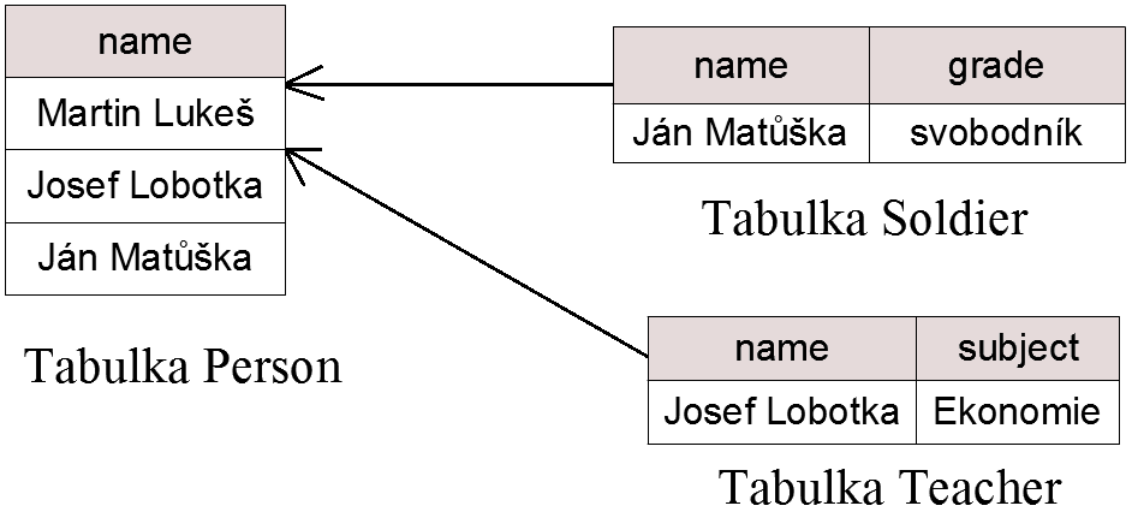
Obrázek 5.3 nám ukazuje výsledek při použití aplikace transformace na hierarchii s InheritanceTypem TablePerClass. Všimněme si počtu sloupců v jednotlivých třídách. V případě složitějších tříd a hlubší hierarchie bychom dostali opravdu široké tabulky - tabulky s mnoha a mnoha sloupci.

Poslední InheritanceType si můžeme prohlédnout na obrázku 5.5. Obsahuje stejné množství tabulek jako TablePerClass, ale všimněme si, že Joined má tabulky užší. Na obrázku jsou šipkami znázorněny cizí klíče na třídu Person. Vzhledem k používanosti není neobvyklé, že tento InheritanceType také vytváří diskriminační sloupec, čímž je zajištěn lepší výkon, nicméně narozdíl od SingleTable tento sloupec není nutný.

Realizaci celé transformace zapsanou v QVT si můžeme prohlédnout v souboru transform_OCL.qvto. V rámci QVT Operational je možné psát zažitým Java-like stylem, nicméně QVT Operational obsahuje mnohé konstrukce, které usnadňují psaní, čtení a snižují chybovost programů. Pro ilustraci stylů psaní můžete kód shlédnout kód transformace transform.qvto, která má téměř tu samou funkcionalitu jako transform_OCL.qvto, ale je psaná Java-like stylem.



Obrázek 5.4: Výsledek mapování při použití InheritanceTypu TablePerClass



Obrázek 5.6: Výsledek mapování při použití InheritanceTypu Joined

Kapitola 6

Test: ORM transformace dědičné hierarchie

Transformace transform_OCL.qvto, jež je realizací návrhu byla otestována následujícími způsoby:

1. Porovnání vlastností výstupního modelu transformace s vlastnostmi očekávaného výstupu.
2. Vygenerováních SQL modelů z výstupních modelů transformace
3. Spuštění vygenerovaných SQL skriptů v databázovém stroji

První test je proveditelný pouze za pomoci transformace. K druhému testu bylo nutné vytvořit transformaci sql-transform.qvto, která vygeneruje z modelu databáze model vytvořený podle metamodelu SQL.ecore. SQL.ecore je jednoduchý metamodel - obsahuje pouze kolekci elementů typu Statement. Každý Statement reprezentuje jeden SQL příkaz zakončený znakem ';', nicméně v programu není provedena validace Statementů. K realizaci posledního testu bylo nutné vytvořit třídu SAX.java, která po přeložení příkazem java SAX <input> <output> extrahuje z XMI souborů SQL kód. Test 3 byl proveden v databázi Derby.

Byly vytvořeny testovací soubory, které zastupují tradiční možná data
Soubory test_inheritance_types01.xmi, test_inheritance_types02.xmi představují reprezentanty dat namapovaných na prázdnou množinu - soubor s indexem 01 obsahuje prázdnou generaci, soubor s indexem 02 obsahuje jen primitivní typy. Na tato data byl aplikován pouze první test, jelikož má mít prázdný výstup.

Soubory test_inheritance_types1.xmi, test_inheritance_types2.xmi, test_inheritance_types3.xmi, test_inheritance_types4.xmi, test_inheritance_types5.xmi reprezentují běžná validní data. Na tyto soubory byly aplikovány všechny 3 testy.

Soubory fail_inheritance_types1.xmi, fail_inheritance_types2.xmi a fail_inheritance_types3.xmi obsahují nevalidní data. Jsou to soubory s více třídami se stejným jménem, třídy s Properties bez patřičných primitivních typů, soubory třídami bez id. Na tyto soubory spuštěna transformace, která má ohlásit za běhu chybu a nemá vytvořit výstup.

Mapování tříd s kolizními SQL názvy nebylo implementováno, takže testy pro tento case nejsou potřebné. Všechny testy dopadly úspěšně.

Kapitola 7

Závěr

7.1 Zhodnocení

Cíle této bakalářské práce byly:

- Popis databázového a aplikačního meta-modelu
- Rozbor vazeb mezi objektovým a databázovým světem
- Realizace ORM transformace

Databázového a aplikačního metamodelu byly popsány v kapitole Použité metamodely. Rozbor vazeb mezi objektovým a databázovým světem byl proveden v kapitole ORM Realizace. ORM transformace byla analyzována a realizována pomocí transformace transform_OCL.qvto a v kapitole Test je byla otestována její kvalita. Zadání práce se ukázalo příliš rozsáhlé, proto jsem se po konzultaci s vedoucím práce rozhodl zaměřit jen na uvedené části. Tato práce je dobrým výchozím materiálem - úvodem do problematiky a bude jistě využita pro zhotovení prací navazujících.

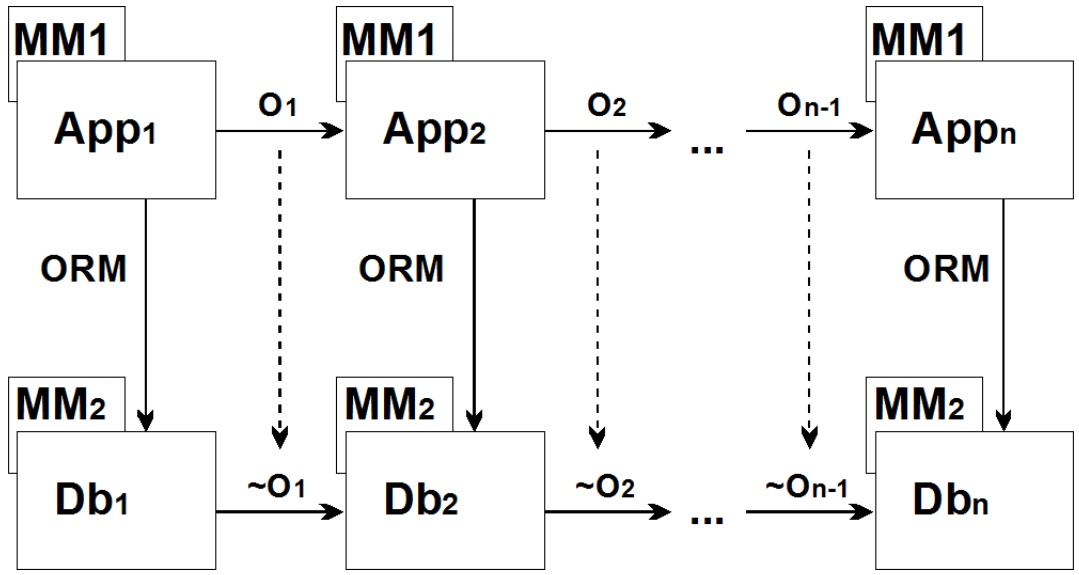
7.2 Budoucí projekty

Tato bakalářská práce byla zpracována v rámci projectu Migdb a vzhledem k k faktu, že byla první zpracovanou součástí v rámci Migdb, obsahuje jen úvod do problematiky.

Obrázek 7.1 ilustruje cíle projektu Migdb. Modely App_1, App_2,... App_n reprezentují jednotlivé generace aplikačních modelů. Modely Db_1, Db_2, ... Db_n reprezentují modely databáze. Jak již bylo řečeno v sekci o metamodelech, každý model musí být utvořen podle nějakého metamodelu. Pro aplikační modely je řídícím metamodelem MM1, pro databázové modely je řídícím metamodelem MM2.

Z aplikačních modelů můžeme získat příslušné databázové modely aplikováním ORM transformace na daný model. Mezi dvojicemi sousedních modelů je přechod definován operací O_n, která mění aplikační model App_n na aplikační model $App_n + 1$. Tyto operace jsou souástí metamodelu MM1, tudíž mají i svůj obraz v databázovém metamodelu. Čárkovaná šipka zobrazuje mapování operace O_n na skupinu operací v databázovém modelu O_n.

Mezi jednotlivými generacemi databáze dochází k provedení skupiny operací O_n, která představuje jednu a více databázových operací.



Obrázek 7.2: Obrázek ilustrující cíl projektu Migdb

Pro zisk modelu databáze po provedení změn $O_1, \ldots O_n$ existují dvě cesty. První cesta je aplikovat na vstupní model App_1 operace $O_1, \ldots O_n$, čímž je získán model App_n a na ten následně aplikovat ORM. Tato cesta je na obr 7.1 viditelná jako sekvence vodorovných šipek a nakonec svislá čárkovaná šipka vedoucí z App_n do Db_n.

Druhou cestou je transformovate pres ORM App_1 na Db_1, zisk databázových skupin operací $O_1, \ldots O_n$ jako obrazů operací $O_1, \ldots O_n$ a následné aplikování těchto změn na Db_1 vedoucí rovněž k zisku Db_n

Na úrovni změny modelů - tzv úrovni M1 zmiňované v kapitole 1.6.2 není rozdíl mezi těmito dvěma postupy. První postup vypadá jednodušeji, protože neobsahuje zisk skupin databázových operací $O_1, \ldots O_n$. Nicméně tento postup zanedbává existující data v databázové generaci Db_1 nebo očekává jejich nové vložení do databázové generace Db_n. Naopak druhý postup umožňuje udělat modifikace nejen na úrovni modelů M1, ale i na úrovni dat M0. Předpoklad je, že tento projekt bude využíván hlavně pro transformaci aplikačních modelů s existujícími daty v databázi, takže ztráta dat je nepřípustná a tím se stává druhá možnost jedinou správnou.

Operace v aplikačním i databázovém metamodelu jsou definované, existuje implementace operací v aplikačním modelu, ale zatím nebyla implementována ani definována transformace operací z aplikačního světa do databázového. Dále nebyly definovány změny provedené pomocí DbOperations v daných generacích DB. Dále jak si jistě pozorný čtenář mohl všimnout, v metamodelech jsou zmiňovány třídy, které nejsou v transformaci nijak použity. Tyto třídy jsou připraveny k použití v dalších navazujících pracích, ať už autora této bakalářské práce nebo jiných studentů.

Literatura

[1] Chris Daly. Emfatic, 2011. URL <http://www.eclipse.org/gmt/epsilon/doc/articles/emfatic/>.

[2] Asbjørn Danielsen. Datové modely, 2011. URL <http://www.fing.edu.uy/inco/grupos/csi/esp/Cursos/cursos_act/2000/DAP%_DisAvDB/documentacion/OO/Evol_DataModels.html>.

[3] Martin Fowler. *Destilované UML*. Grada Publishing, Havlíčkův Brod, 2009.

[4] Petr Klemšinský. Diplomová práce návrh systému pomocí mda, 2009. URL <http://www.scribd.com/doc/28689537/18/Metamodelovani-a-MOF>.

[5] OMG. Qvt specifikace, 2011. URL <http://www.omg.org/spec/QVT/1.1/PDF/>.

[6] Rudolf Pecinovský. *Návrhové vzory*. Computer Press a.s., Brno, 2007.

[7] Rudolf Pecinovský. *Myslíme objektově v jazyku Java*. Grada publishing, Praha, 2009.

[8] Přispěvatelé Wikipedie. Databáze, 2011. URL <http://en.wikipedia.org/wiki/Databases>.

[9] Přispěvatelé Wikipedie. Databázové modely, 2011. URL <http://en.wikipedia.org/wiki/Database_model>.

[10] Přispěvatelé wikipedie. Xmi, 2011. URL <http://en.wikipedia.org/wiki/XML_Metadata_Interchange>.

[11] Přispěvatelé wikipedie. Association, 2011. URL

Kapitola 8

Seznam použitých zkratek

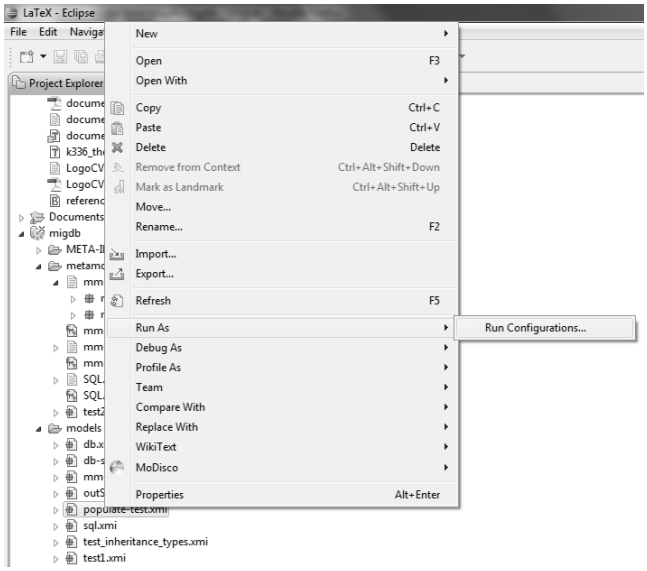
ORM object-relational mapping
XML extensible Markup Language
QVT query view transformation
DBMS database management system
ACID atomicity, consistency, isolation, durability
RDBMS relational database management system
OODBMS object oriented database management system
ORDBMS object relational database management system
SQL structured query language
JPA Java persistence API
MDA model driven architecture
MDE model driven engineering
MDD model driven development
OMG object management group
UML unified modeling language
CIM computation independent model
PIM platform independent model
PSM platform specific model
ISM implementation specific model
DTD document type definition
IDE integrated development environment
XMI XML metadata interchange
MOF Meta Object Facility
EMF Eclipse modeling framework
W3C World Wide Web Consorcium
OCL Object Constraint Language
php PHP: Hypertext Preprocessor
ATL Atlas transformation language

Kapitola 9

Instalační a uživatelská příručka

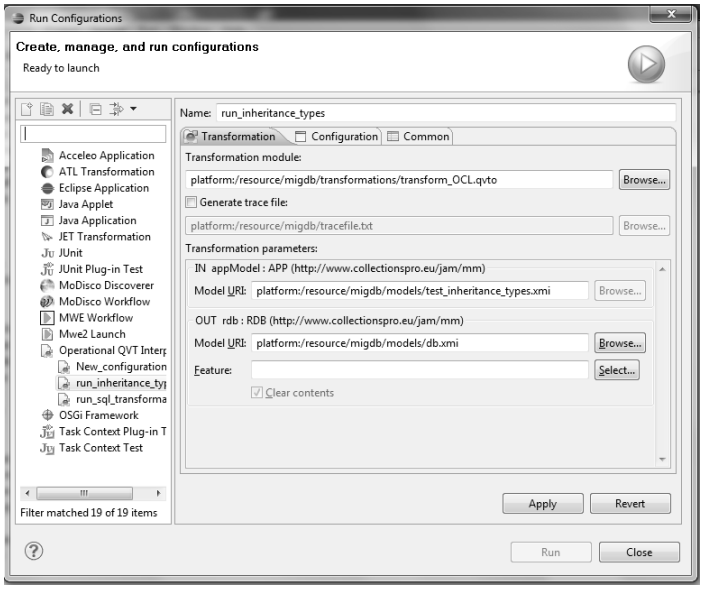
Na přiloženém médiu je umístěno IDE Eclipse Helios, které je plně funkční na systému Windows s nainstalovanými prostředky potřebnými pro 32 bit Javu. IDE se spouští pomocí souboru Eclipse.exe.

Pro spuštění transformací je nutné kliknout pravým tlačítkem na project, vybrat run as a následně run configuration viz obr 9.1



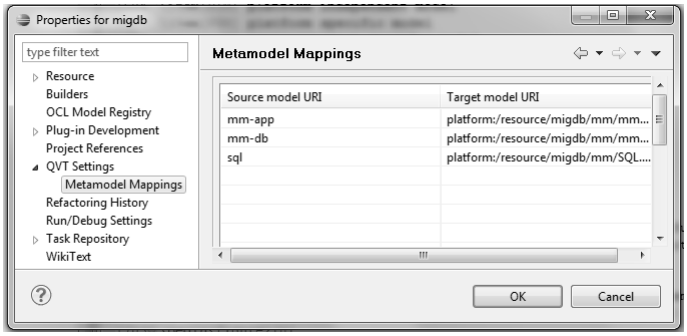
Obrázek 9.2: Run configuration

Otevře se dialogové okno 9.3. Pole transformation module upřesňuje, která transformace bude spuštěna, další pole jsou určena pro výběr cesty k vstupním a výstupním souborům. Pokud není možné spustit transformaci (tlačítko run je zašedlé a nedá se stisknout), neodpovídá vybraný model svému metamodelu. Nedoporučuji měnit strukturu souborů, nastal by problém s URI modelů a byla by nutná následná modifikace XMI. Otevřeme si properties projektu right-clickem na název projektu, výběrem položky properties. V dialogovém okně nalezneme QVT Settings, pod touto nabídkou je potom Metamodel-mappings a v tomto okně jsou zobrazené cesty k metamodelům,



Obrázek 9.4: Run configuration

je nutné nastavit cesty ke všem metamodelům určeným v hlavičce transformace. Jsou předpřipravené run konfigurace test_inheritance_types1 až 5 pro transformaci transform_OCL.qvto a test_sql1 až 5 pro sql-transform.qvto.



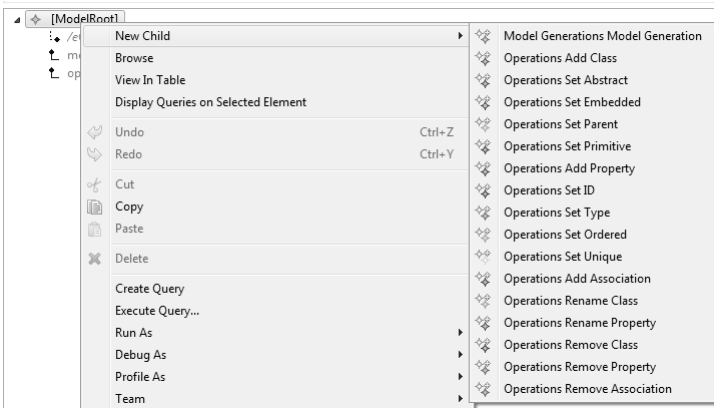
Obrázek 9.6: Run configuration

Problémem je znovuspouštění transformací. Výstupní soubory se před aplikací transformace nevyprazdňují a tudíž se duplikují výstupní modely. Nejjednodušším řešením je před každým spuštěním transformace smazat. V předpřipravených složkách jsou testy pro dané transformace, ale pokud je možné vytvořit si vlastní model a otestovat transformaci na něm. Vytvoření vlastního modelu se provede otevřením souboru s metamodelem, rightlick na ModelRoot a výběr nabídky CreateDynamic instance - viz 9.7. V případě, že vytvoříte potomky ModelRootu, nebude potom vytvořen validní model.

Nově vytvořený soubor musíme otevřít v Generic EMF editoru nebo v Modisco Model Browseru (rightclick na soubor a open with). Při editaci klikneme pravým tlačítkem

na element, jehož potomky chceme vytvářet a v pop-up okně vybereme danou entitu viz například 9.9 pro ModelRoot databáze.

Obrázek 9.8: Create Dynamic instance



Obrázek 9.10: New child

Kapitola 10

Obsah přiloženého CD