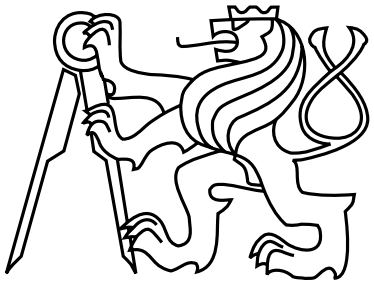
Zadání práce

Realizujte nástroj, který z popisu změny databázového schématu vygeneruje sadu SQL dotazů, které umožní migraci konkrétní databáze. Popište databázový meta-model, dále vytvořte model-to-text transformace pro generování SQL dotazů z tohoto meta-modelu a realizované operace. Před realizací se seznamte s nástrojem Eclipse modeling framework a jazyky Xtend 2, QVT, OCL a MWE.

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Modelem řízená evoluce databáze

Petr Tarant

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

17. května 2012

Poděkování

Děkuji panu Ing. Ondřeji Mackovi za trpělivost a spolupráci na projektu. Chtěl bych také poděkovat pánům Pavlu Moravcovi a Davidu Harmancovi, Ph.D., za jejich ochotu a cenné rady při spolupráci. V neposlední řadě bych chtěl poděkovat panu Jiřímu Ježkovi za skvělou spolupráci.

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 5. 4. 2012 .....

Abstract

Object-relational mapping is a common programming technique used to accelerate development of software applications. Our project is focusing on this problematic domain and trying to move it further towards automation. Our goal is to change the database schema corresponding with the changes made in application object model, without any data loss. The focus on data consistency during database changes is what makes our project unique among others. My part of work on the project focuses mainly on database model-to-model and model-to-text transformations. By deploying the project into practice, we expect increased efficiency of programmers, who will no longer have to deal with the database regeneration.

Abstrakt

Objektově-relační mapování je běžnou programovací technikou, která se používá k urychlení práce při vývoji aplikací. Náš projekt se zaměřuje právě na tuto problémovou doménu a snaží se ji posunout dále směrem k automatizaci. Naším cílem je změna databázového schématu na základě změn na aplikační úrovni, a to bez ztráty dat. Právě v uchování datové informace i přes změnu databáze se náš projekt odlišuje od konkurenčních řešení. Moje práce na projektu se soustředí hlavně na databázové model-to-model a model-to-text transformace. Nasazením projektu do praxe si slibujeme zvýšení efektivnosti práce programátorů, kteří se už nebudou muset zabývat přegenerováním databáze.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Projekt s názvem Migrace Databáze si stanovil cíl, vyřešit problematiku evolučních změn v životním cyklu aplikace. Konkrétně se jedná o řešení objektově-relačního mapování bez ztráty informací v relační databázi. Právě za uchováním informace se skrývá hlavní myšlenka celé práce. Je velkou výzvou vymyslet evoluční operace generující SQL dotazy pro databázi, které budou pokrývat základní množinu změn na aplikační a databázové úrovni tak, aby celý software zůstal konzistentní. Vyřešení tohoto problému by přineslo další úroveň automatizace do evolučního procesu mapování, který je jinak v dnešní době zcela v rukou programátorů.

Hlavní myšlenka celé práce je právě ve slově evoluce. Snažíme se naším evolučním procesem dovést aplikaci z počátečního stavu až do požadovaného výsledku. Myšlenka zanechání konzistence dat v databázi nás donutila k rozdělení evoluce na co nejelementárnější kroky. Pod pojmem krok je vhodné si představit jednu drobnou změnu napříč celým softwarem (od objektové úrovně až po SQL dotaz do databáze).

Tento evoluční proces jsme se kvůli potřebě vysoké abstrakce rozhodli aplikovat na vrstvě modelů definujících software. Máme model představující aplikaci a model představující databázi. Ona evoluční transformace pak pouze mění stavy těchto modelů, což má za následek znovu generování vlastního kódu aplikace a generování SQL dotazů do databáze.

V této práci budete seznámeni jak s implementací evoluce, tak se samotnou strukturou našeho frameworku. Dále bude blíže popsáno, jaké důsledky má snaha o zachování dat v konzistentním stavu a co vlastně znamená elementární krok v průběhu evoluce softwaru. Celý náš projekt je ve výsledku jen sadou komponent, které jsou ve vhodném pořadí spouštěny v pracovním toku aplikace, proto bude každá část programu definována jako jedna z komponent a v závěru textu bude znázorněno, jak spolu všechny tyto komponenty komunikují.

Cílem není popsat kompletní strukturu našeho frameworku, ale pouze její databázovou část a generátor SQL. Aplikační úroveň je zmiňována pouze okrajově, přestože je nezbytnou součástí databázové úrovně aplikace. Více informací o aplikačních modelech či objektově-relačních mapováních lze získat v bakalářské práci pana Jiřího Ježka [1].

1.1 Experimentální projekt

Zpočátku na tomto projektu, který je zadán externí firmou CollectionsPro, s.r.o., pracoval tým studentů, který se soustředil hlavně na nasazení implementační metodiky, adaptaci s technologiemi a vývojovým prostředím. Po půl roce převzal projekt náš tým a naší úlohou na projektu je již samotná implementace návrhového řešení.

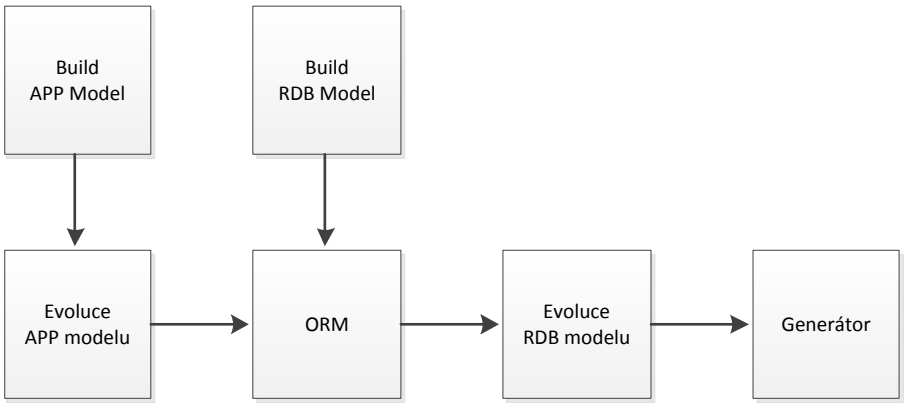
Vývoj probíhá v prostředí Eclipse, které podporuje všechny na projektu používané technologie. I z pohledu technologií a jazyků se jedná o experimentální projekt. Ještě přesně nevíme, jestli za nějaký čas nenarazíme na strop jazyka, nebo na nějakou vhodnější technologii, kterou bychom mohli nahradit tu stávající.

Nelze s jistotou konstatovat, v jaké části vývoje se nacházíme, nebo zda při vývoji směřujeme správným směrem. Jelikož je projekt experimentálního rázu, není možné říct, zda bude někdy dokončen, nebo alespoň v určitém stádiu schopen nasazení. Může se také ukázat, že náš způsob vývoje narazí na příliš velká omezení, která by bránila použitelnosti v praxi.

Kapitola 2

Evoluční framework

Framework se skládá ze čtyř základních komponent - Evoluce na úrovni aplikace a databáze, objektově-relační mapování a generátor SQL. Mým úkolem byla implementace databázové evoluční transformace a generátoru SQL. Pro představu jsou všechny komponenty a jejich spolupráce znázorněny na obrázku 2.1.



Obrázek 2.1: Uspořádání komponent ve frameworku

2.1 Model Driven Architecture

Sdružení Object Management Group (OMG) v roce 2001 standardizovala Model Driven Architecture (MDA)[1], jako jeden z možných způsobů vývoje software, který bude dovolovat snadnou výměnu použitých komponent a frameworků v systému.

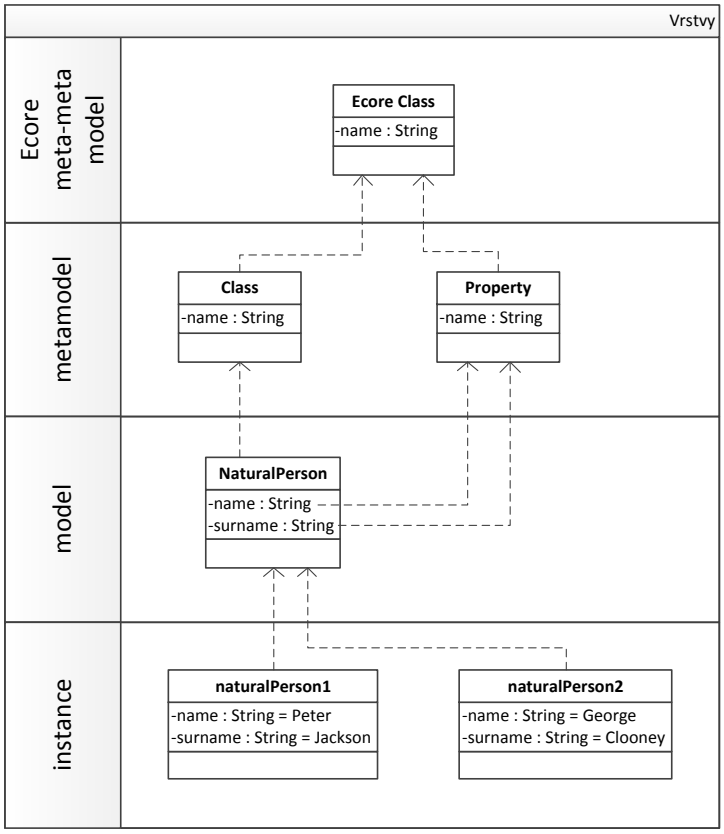
[1] MDA: http://www.omg.org/mda/

Základní myšlenku MDA dobře popsal P. Klemšinský [3] jako přesunutí těžiště vývoje, nacházejícího se převážně na úrovni kódu, do úrovně modelování. Výsledný zdrojový kód pak vzniká transformováním těchto modelů.

Model je popis systému (nebo jeho části) napsaný pomocí dobře definovaného jazyka [3]. Model nám dovoluje popsat strukturu nějakého problému a následně s ním pracovat jako s celkem. Je také důležité si uvědomit, jaký význam má model v aplikaci a umět ho rozlišit od jeho abstraktnějších verzí, meta-modelů.

Meta-model je také modelem v obecném slova smyslu. Pouze jeho funkce se drobně liší od modelu samotného. Meta-model má za úkol definovat, jakou strukturu a jaká pravidla smí mít modely v dané aplikaci. Jde tedy o ještě vyšší vrstvu abstrakce, která ale nemusí být nutně tou poslední.

Samotná struktura MDA se skládá z více vrstev, které jsou definovány či reprezentovány výše zmíněnými modely a meta-modely, mezi nimiž se přechází za pomocí transformací s jasně definovanými pravidly. Na obrázku 2.2 je znázorněna čtyřvrstvá architektura MDA v Eclipse.



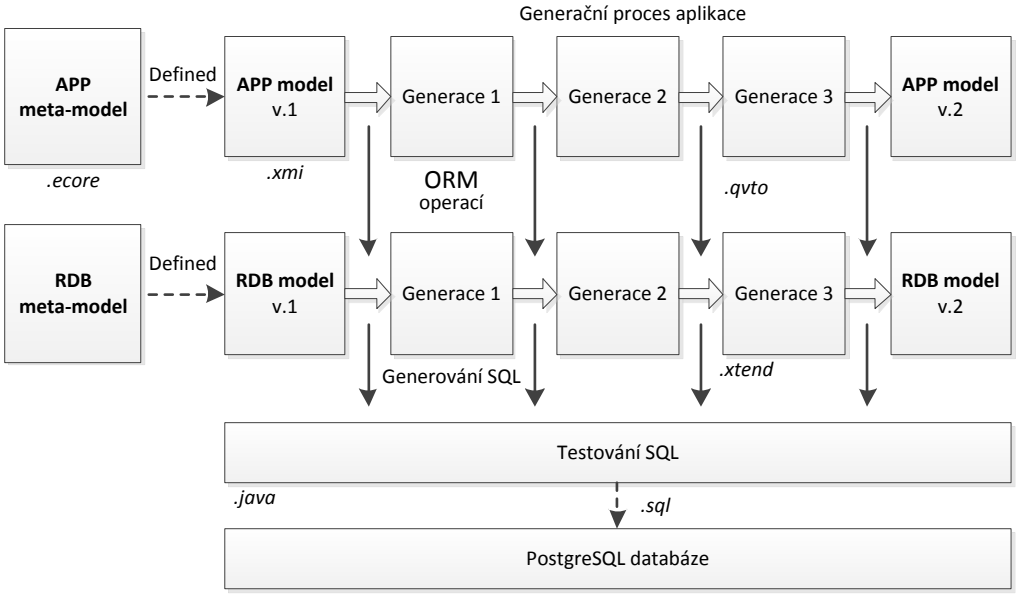
Obrázek 2.2: Čtyřvrstvá architektura MDA v prostředí Eclipse za pomoci technologie Ecore

Náš projekt je implementovaný metodikou Model Driven Development (MDD), díky níž jsme schopni dosáhnout vyšší abstrakce, což nás odstiňuje od konkrétních databází nebo programovacích jazyků.

Vývoj zaměřený na vyšší abstrakci je pro naši práci velmi vhodný, protože jsme schopni vytvořit silný aplikační a databázový meta-modelový základ, který bude schopen pojmout většinu ze základních vlastností programovacích jazyků a databází. Pro transformaci z Platform Specific Model (PSM) do Implementation Specific Model (ISM) jsem si kvůli ověření výsledků zvolili pravidla databází PostgreSQL a jejich syntaxi.

2.2 Struktura frameworku

Základními kameny frameworku jsou aplikační a databázová vrstva. Aplikační vrstvu zastupuje aplikační model definovaný aplikačním meta-modelem. Databázovou vrstvu zastupuje databázový model definovaný databázovým meta-modelem. Přechod mezi aplikační úrovní a databázovou úrovní zastává ORM. Mezi samotnou databází a databázovým modelem je vrstva generující SQL z operací, které jsou vykonávány nad databázovým modelem. Databáze má ještě jakýsi ochranný štít v podobě drobné vrstvy, která posílá SQL do databáze a zachytává případné odpovědi z databáze. Na obrázku 2.3 je celá tato struktura znázorněna v evolučním procesu.



Obrázek 2.3: Struktura frameworku

Kapitola 3

Objektově-relační mapování a instance

Objekt je programovací entitou, která je schopna jednoduše reflektovat svět okolo nás, a zároveň je lehce pochopitelná a zpracovatelná člověkem. Díky objektům a samozřejmě také vyšším programovacím jazykům jsme v dnešní době schopni vytvářet mnohem větší, sofistikovanější a chytřejší systémy než dříve.

Mohlo by se zdát, že program sestavený z objektů má jen samé výhody. Opak je však pravdou. Objekty nám sice dovolují abstraktněji přemýšlet, ale zároveň nás oddalují od platformy, se kterou pracujeme. Příkladem takové platformy může být relační databáze definovaná matematickým modelem. Rozdíl mezi matematickým světem a objektovým přístupem je právě oním oddálením od platformy. Oba světy si totiž zcela neodpovídají, a tak vzniká jakási neshoda, šedá zóna (Object-relational Impedance Mismatch[1]), jejímž řešením se stalo objektově-relační mapování (ORM) [8].

Objektově-relační mapování není novinkou na poli informatiky. V dnešní době existuje celá řada nástrojů a frameworků zajišťujících pohodlný přesun dat z objektové struktury aplikace do relační databáze. Mezi nejpoužívanější se řadí Hibernate [2] a OpenJPA [3] pro jazyk JAVA , LiteSQL [4] a ODB [5] pro C++, Zend Framework [6] pro PHP, ActiveRecord [7] pro Ruby on Rails a v neposlední řadě například Entity framework [8] od Microsoftu. Všechny tyto nástroje se zaměřují na stejný problém při mapování aplikačních modelů. Snaží se co nejvíce zautomatizovat vlastní proces tvorby aplikace. Ale co evoluce již zavedené aplikace?

S evolucí softwaru bojuje snad každá firma, která udržuje velké databáze. Celý problém s ORM totiž tkví v datech (instancích) samotných. Výše popsané nástroje sice umí vytvořit strukturu databáze podle aplikační předlohy, ale s již existujícími instancemi si neporadí. Pokud totiž databázi zaplníme daty a znovu budeme požadovat úpravu aplikace, o data s

[1] ORIM: http://www.agiledata.org/essays/impedanceMismatch.html
[2] Hibernate: http://www.hibernate.org/
[3] OpenJPA: http://openjpa.apache.org/
[4] LiteSQL: http://sourceforge.net/apps/trac/litesql/
[5] ODB: http://organizersdb.org/
[6] Zend Framework: http://framework.zend.com/
[7] ActiveRecord: http://ar.rubyonrails.org/
[8] Entity Framework: http://msdn.microsoft.com/en-us/library/aa697427(v=vs.80).aspx

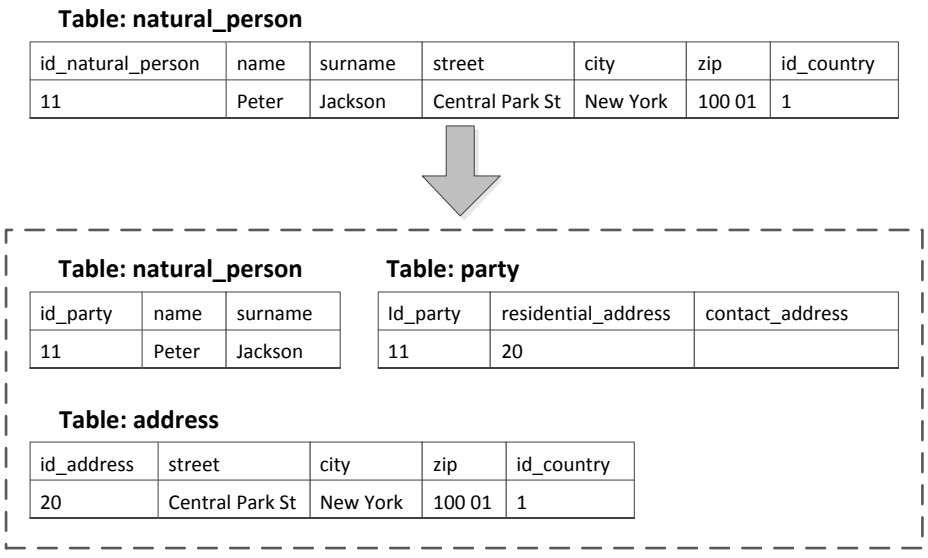
použitím těchto nástrojů přijdeme. Proto se firmy vydávají různými směry jako například vývojem vlastního zcela individuálního řešení pro migraci dat, které je vytvořené zcela pro účely firmy.

Cílem našeho projektu je tedy jinými slovy vytvoření takového frameworku, který bude schopen upravit databázové schéma podle změn objektové struktury, a to bez ztráty jakýchkoliv dat. Navíc si klademe za cíl obecnost nástroje tak, aby nebyl závislý ani na programovacím jazyku, ani na typu relační databáze.

3.1 Datová konzistence

Udržet data v konzistentním stavu je alfou a omegou tohoto projektu. Konzistentnost v relačních databázích označuje stav, kdy konkrétní data porušují pravidla a omezení, která jsou na ně kladena [2]. Kdy může nastat situace, že bude aplikace v nekonzistentním stavu? Nejjednodušším příkladem je smazání sloupce, na který v jiné tabulce ukazoval cizí klíč. Nekonzistentnost aplikace může nastat skoro po každé evoluční operaci, kterou nad modelem aplikujeme. Naším úkolem je zabránit takovým stavům a nedovolit průniku této změny do finálního stavu aplikace.

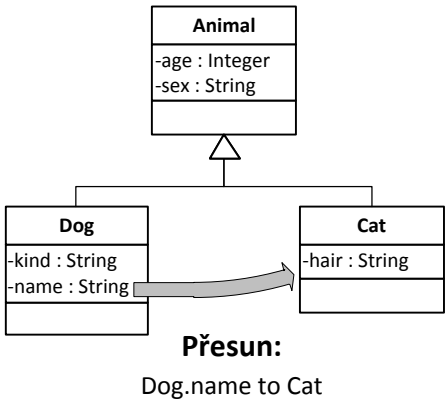
Naším cílem ale není jen zachování konzistence, ale také zachování informace, kterou data drží. Data se tedy mohou konzistentně změnit, ale musí držet stále stejnou informaci. Pro lepší ilustraci je vše znázorněno na obrázku 3.1.



Obrázek 3.1: Zachování datové informace - Informační hodnota dat o Petru Jacksovoni zůstala stejná i po transformačním rozdělení do tří tabulek.

Problém se ztrátou informace je zakořeněný mnohem výše v hierarchii programu. Kritická je totiž již chvíle, kdy v ORM převádíme aplikační změny v databázové. Aniž by se to zdálo na ORM vrstvě logické, již zde musíme myslet na instance a konat kroky k jejich zachování. V podstatě jediná vrstva zcela odstíněná od instančních problémů je vrstva aplikační. Všechny ostatní musí při jakékoliv akci brát ohled na konzistentnost dat a možnou ztrátu informace.

Představme si situaci jako na obrázku 3.2. Mějme dvě hierarchicky sourozenecké třídy s několika atributy. Atribut *name* chceme převést z třídy *Dog* do třídy *Cat*. Na objektové úrovni se změní pouze pozice atributu. Na databázové úrovni to znamená vytvoření nového sloupce v tabulce *cat*, převedení dat a následné smazání původního sloupce.



Obrázek 3.2: Ztráta datové informace při přetažení atributu do jiné třídy

Nyní se ale zaměřme přímo na instance. V tuto chvíli nepřemýšlejme nad jejich významem, ale zamysleme se nad jejich množstvím. Zásadní problém zde totiž nastává ve chvíli, kdy tabulka *cat* drží méně instancí než tabulka *dog*. Jelikož je obtížně proveditelné do ostatních sloupců třídy *cat* dogenerovat výchozí hodnoty, logicky bychom museli o přebývající data z tabulky *dog* přijít (museli bychom je zahodit). Přesně v tomto okamžiku dochází ke ztrátě informace.

ORM vrstva by tedy měla před spuštěním vlastní operace požádat databázovou vrstvu o kontrolu, zda je počet instancí shodný. Pokud shodný nebude, musí se dotázat přímo uživatele, zda mu nevadí případná ztráta informace.

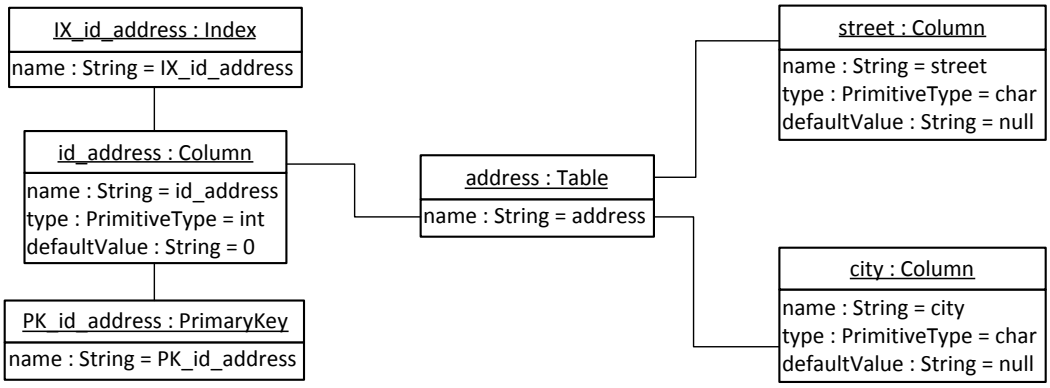
Ztráta informace není vždy takto jasně viditelná a lehce vyřešitelná. Problém ze ztrátou informace se může objevit i při nevyhovujících omezeních nad sloupci a v mnoha jiných případech.

Kapitola 4

Databázový meta-model

Nejdůležitějším prvkem databázové vrstvy je její meta-model. Databázový meta-model obsahuje šablonu entit vhodných pro obecný modelový popis aktuálního stavu databáze. Každý databázový prvek či omezení je v modelu reprezentován vlastním objektem. Mezi nejzákladnější patří *Schema*, *Table*, *Column*, *Sequence*, *Index* a *Constraint*.

Vše v meta-modelu je objekt s různými vlastnostmi. Pokud v modelu, který je definovaný tímto meta-modelem, chceme vytvořit tabulku, jež bude mít nějaké sloupce, je to vazba mezi množinou objektů typu *Column* a objektem typu *Table*. Na obrázku 4.1 je tato objektová struktura modelu vyobrazená názorněji. Tabulka *address* si drží tři sloupce *street*, *city* a *id_ address*. Sloupec *id_ address* je navíc primárním klíčem



Obrázek 4.1: Objektová představa databázového modelu.

4.1 Základní entity meta-modelu

Na obrázku 4.2 je vyobrazený databázový meta-model bez výpisu jednotlivých operací. Za všechny operace je v modelu pouze *ModelOperation*.

Table a *Column* jsou nejzákladnějšími a zároveň nejjednoduššími prvky meta-modelu. Tabulka (Table) se skládá z řádků a sloupců a slouží k přímému ukládání dat do databáze. Sloupec (Column) je základním prvkem databázové tabulky. Kolekce sloupců pak patří mezi hlavní vlastnost tabulky.

Schema v našem modelu znamená stejně jako v databázích, ve kterých se používá, pouze jmenný prostor pro dané databázové prvky (tabulky, indexy, sequence, ...). Pojem schéma se také používá ve smyslu definování struktury nebo schématu databáze.

Sequence se v databázích používá jako generátor posloupnosti čísel. Můžeme jej tedy využít pro generování hodnot primárních klíčů.

Běžné použití primárních klíčů je takové, že každá tabulka má svou vlastní sekvenci. Pokud chceme v databázi nasimulovat hierarchii, použijeme například vazbu tabulek skrze primární klíče. Celá hierarchie má tedy jen jednu sekvenci. V naší aplikaci jsme se vzhledem k obtížím s přečíslováváním primárních a cizích klíčů při modelových transformacích rozhodli pro jiný, ale také v praxi používaný konstrukt. V databázi je pouze jedna globální sekvence. Díky tomu máme zajištěnou unikátnost každého identifikátoru v celé aplikaci.

Index nám slouží k urychlení přístupu k požadovanému řádku tabulky. Indexace je oproti sekvenčnímu čtení velice dobrý optimalizační nástroj. V našem databázovém modelu jsou automaticky indexované všechny sloupce s omezením *PrimaryKey*.

Constraint (omezení) v našem meta-modelu dělíme na tabulkové *TableConstraints* a sloupcové *ColumnConstraints*. Tabulková omezení si drží ve vlastnostech přímo tabulka. Jedná se o unikátní index *UniqueIndex*, primární klíč *PrimaryKey* a cizí klíč *ForeignKey*. Sloupcové omezení je v našem meta-modelu pouze nenulové omezení *NotNullConstraint*. Důvodem pro rozdělení na tabulková a sloupcová omezení jsou údaje, které potřebujeme pro jejich vytvoření. Například unikátní index nelze vázat ke sloupci, protože unikátnost může definovat i kolekce sloupců.

Mezi další entity patří v databázovém meta-modelu *ModelRoot*, *ModelGeneration* a *ModelOperation*, které byly vytvořeny uměle pro potřeby projektu.

ModelRoot je základním kamenem celého meta-modelu. Drží všechny evoluční generace modelu a všechny operace, které můžeme nad modelem vykonávat.

ModelGeneration se zabývá evolucí modelu. Při aplikování jakékoliv změny nad modelem vzniká nová generace neboli nový stav tohoto modelu. Objekt typu *ModelGeneration* obsahuje údaje o tom, kolik má daný model v daném okamžiku tabulek, sloupců a omezení ve schématech.

ModelOperation je abstraktním předkem všech operací, které je možné nad modelem vykonat. Jelikož jsou operace nejdůležitější částí našeho meta-modelu a ve své podstatě jsou základem našeho "know how", věnuji jim celou následující podkapitolu.

Databázový meta-model si lze prohlédnout na přiloženém médiu *bp/mm-rdb.ecore*.

4.2 Databázové operace

Meta-model nám definuje strukturu všeho, co se smí vyskytovat v budoucím databázovém modelu. Proto je v něm třeba nadefinovat i strukturu operací, které budeme chtít v budoucnu aplikovat nad modelem.

Pokud se rozhodneme pro změnu aplikačního modelu (například přidat třídu), tato změna se skrze ORM zpropaguje do databázové vrstvy jako sada operací, které změnu provedou na databázové i datové úrovni. Právě tato sada vygenerovaných operací je nástrojem pro úpravu databázového modelu. Operace na databázové úrovni dělíme do několika skupin podle jejich použití.

Nejzákladnějšími operacemi jsou operace přidávající do modelu něco nového. Tyto operace mají prefix Add- a dovolují nám od základu sestavit celý model. Mezi tyto operace patří třeba *AddTable*, *AddColumn*, *AddIndex*, *AddSequence* atd.

Ve stádiu sestaveného modelu jsou k užitku editační operace s prefixy Rename- a Set-. Prefix Rename- značí přejmenování daného objektu v modelu. Prefix Set- zase nastavení určité vlastnosti objektu.

Poslední ze základní sady operací jsou operace s prefixem Remove-. Tyto operace odstraňují prvky z modelu. Operace Remove- mají nejpřísnější validační kontroly před samotným vykonáním.

Všechny předešlé operace mají za následek nejen vytvoření nové generace v evoluci modelu, ale také konkrétní změnu v modelu samém. Jelikož databázová úroveň má mimo jiné za úkol i udržení datové informace a konzistenci dat, obsahuje speciální skupinu operací, které sice model přivedou do nové generace, ale neprovedou žádnou modelovou úpravu. Tyto operace jsou totiž primárně zaměřené na data v databázi. Těmto operacím souhrnně říkáme datové a lze je dále dělit na dotazovací a editační.

Dotazovací operace slouží k vygenerování SQL dotazu, který ověří námi požadovaný stav dat. Například operace *HasNoInstances* se dotazuje, zda je tabulka záznamově prázdná.

Editační operace neupravují model, ale data samotná. Může se jednat například o generování hodnot ze sequence nebo všemožné modifikace příkazu INSERT.

Aby byl problém datových operací dostatečně nastíněn, popíšeme si zde jednoduchou transformační situaci. Pokud se například rozhodneme o přesunutí atributu z jedné třídy do druhé, v databázi to pro nás znamená v prvé řadě srovnání počtu instancí v obou tabulkách (prevence před ztrátou informace), dále vytvoření nového sloupce (*AddColumn*), pak nakopírování samotných dat (*InsertInstances*) a nakonec odstranění původního sloupce (*RemoveColumn*). Ověření a kopírování jsou v tomto jednoduchém případě datovými operacemi. Jejich činnost sice mění datový stav, ale nijak nemodifikuje stav modelu.

Soupis všech operací je v příloze A.

4.3 Struktura operací

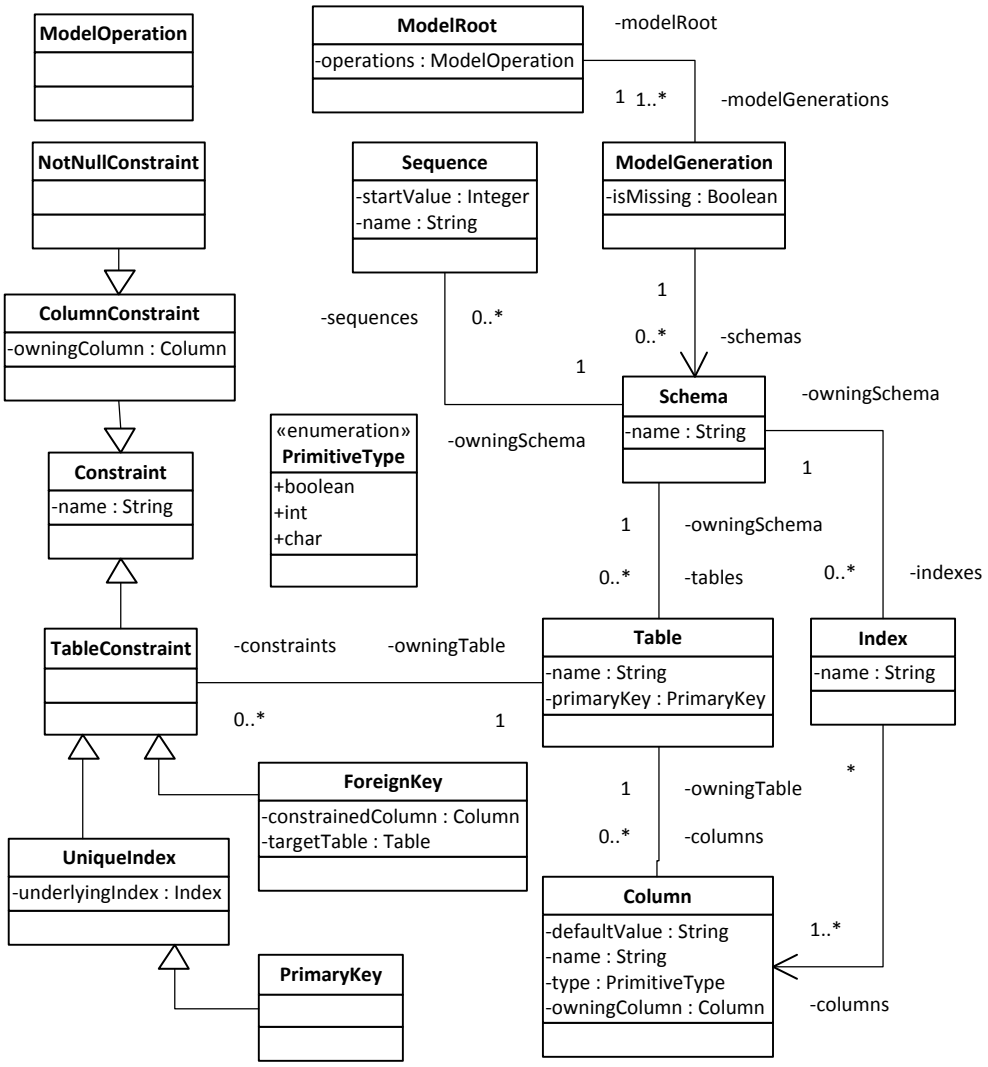
Nyní se již vzdalujeme od samotného meta-modelu. Meta-model nám u operací definoval jejich strukturu, tedy jaké informace potřebují k tomu, aby mohly být aplikovány nad modelem, ale nedefinoval strukturu toho, jak je používat. Operace *AddTable* například potřebuje mít na vstupu informaci o tom, v jakém schématu a s jakým jménem se má tabulka vytvořit. V meta-modelu už ale nezaznívá informace o tom, že se musí ověřit validita těchto informací vůči modelu a datům. Proto je důležité předběžně zde nastínit strukturu aplikování operací, i když probíhá v úplně jiné komponentě systému.

U každé operace je třeba znát aplikační informace. Tyto informace nejprve musí projít validací, tedy testem, zda žádným způsobem nepoškodí konzistentnost modelu (například vytvoření tabulky se stejným názvem, jaký již existuje). Validační testy na ověřování konzistence nejsou vždy na první pohled zřejmé. Například u operace *RemoveTable* se mimo jiné musí ověřovat neexistence vazeb na jiné tabulky, zda nad tabulkou nejsou aplikována nějaká omezení atd. Často je to právě ona validační část, která je na operacích nejsložitější.

4.4 Ecore a reprezentace meta-modelu v aplikaci

Ecore neboli Core EMF je součástí Eclipse Modeling Frameworku (EMF)[4], díky němuž jsme schopni v prostředí Eclipse pracovat s meta-modely. Ecore je velice intuitivní vizuální nástroj pro sestavení meta-modelů. Programátor se při práci s tímto nástrojem nemusí učit žádný nový jazyk, žádnou novou syntax. Jedná se o pouze vizualizované XMI[10]. Největší výhodou Ecore je propojení s ostatními nástroji při vývoji řízeném modelem v Eclipse. Nevýhodou nástroje je jeho jednoduchost. Pokud se budeme snažit o rychlou textovou editaci, budeme si muset soubor otevřít v XML editoru a nejprve se zorientovat ve velice nepřehledném XML kódu. Pro účely rychlé editace proto existuje celá řada jiných nástrojů.

Abychom mohli v rámci naší aplikace používat meta-model v jakékoliv komponentě či projektu, musíme ho nejprve exportovat do podoby, které bude rozumět celé vývojové prostředí. XML je sice pro stroj čitelné, ale není to vhodná reprezentace, pokud požadujeme pro práci s modelem objektový přístup. Z důvodu tohoto požadavku musí být meta-modely exportovány do jazyka JAVA, díky němuž nám vznikne běžná třídní struktura. Každá entita meta-modelu je pak zastoupena vlastní třídou, jejíž název je zakončen sufixem Impl- (mezi entity samozřejmě počítáme i jednotlivé operace).



Obrázek 4.2: Databázový meta-model

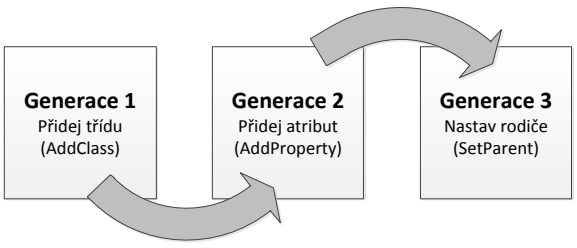
Kapitola 5

Evoluce aplikace

Evoluci, neboli vývoj či rozvoj softwaru, si pravděpodobně dokáže představit každý. S novými verzemi aplikace přibývá funkcionalita, ubývá chyb a narůstá skupina spokojených zákazníků. V dnešním objektovém světě dokonce není nikterak složité přidat do systému něco nového. Pokud je vše dobře navržené, přidání požadavků se promítne pouze na jednoduché úpravy aplikační vrstvy (přidání třídy, úprava hierarchie atd). Díky tomu je v dnešní době verzování softwaru zcela běžnou a velice rychlou záležitostí.

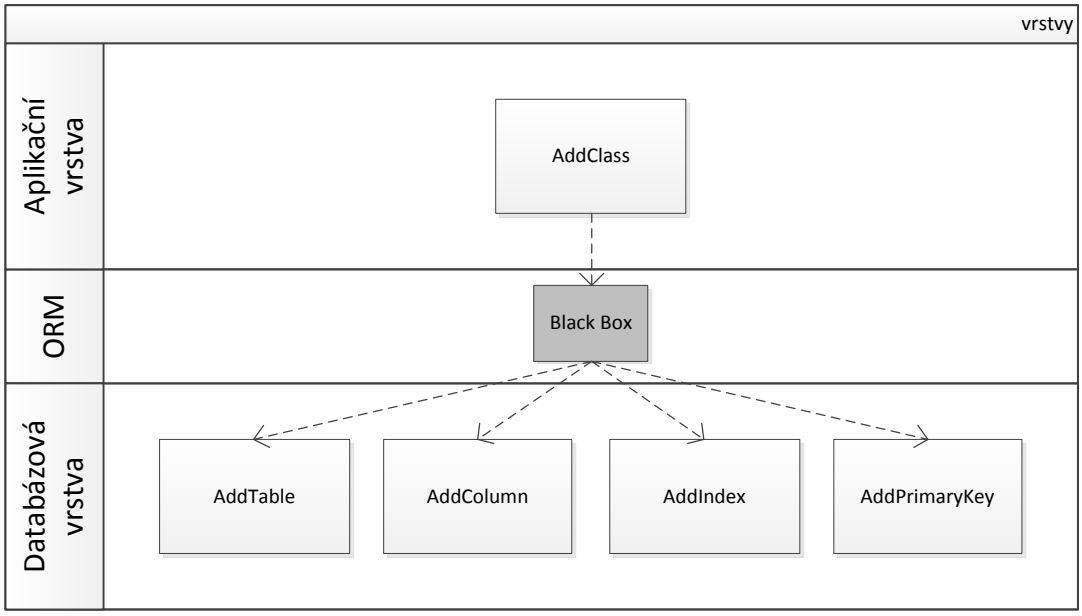
Jak již zaznělo v úvodní kapitole, největším problémem dnes není implementace nových požadavků, ale udržení informace mezi starou a novou verzí softwaru. Náš evoluční proces se snaží právě tento problém vyřešit.

V zásadě se nejedná o nic jiného než o Model-to-Model horizontální (zůstáváme na stejné vrstvě) transformace, kdy přecházíme z jednoho stavu modelu do druhého. Na aplikační vrstvě definované aplikačním modelem chceme provést určité změny. Aplikační meta-model [1] nám nabízí veliké množství operací, díky kterým jsme schopni model modifikovat. Sestavíme tedy sekvenci operací, jejíž cílem bude požadovaná změna. Jelikož je zachování informace velice citlivou záležitostí, musíme mít operace co nejelementárnějšího rázu. Každá tato elementární operace automaticky vytvoří novou generaci, tedy mezistupeň v evolučním procesu. Na obrázku 5.1 je vyobrazena představa generačního procesu.



Obrázek 5.1: Rozdělení evolučního procesu na generace

Tyto aplikační operace jsou skrze ORM přetransformovány na sadu databázových operací, které provedou stejný generační postup na databázové úrovni. Jelikož mapování operací z aplikační úrovně do databázové není 1:1, neodpovídají si ani počty generací na aplikační a databázové úrovni. Tato drobná nekonzistence však v žádném ohledu není na závadu. Na obrázku 5.2 je znázorněno mapování operací mezi aplikační a databázovou vrstvou, z níž je patrné, že jedna operace na aplikační úrovni může vyvolat celou sadu operací na úrovni databázové.



Obrázek 5.2: Mapování operací

5.1 QVT

Před bližším popisem první komponenty databázové vrstvy je třeba si přiblížit transformační jazyk QVT, ve kterém je tato komponenta napsána.

QVT (Query/View/Transformation) je imperativní transformační jazyk definovaný OMG. Primárním účelem jazyka QVT je práce s modely, která je klíčová pro architekturu řízenou modelem. Nejsilnější stránkou jazyka je podpora komponenty OCL, která dovoluje programátorům mnohem elegantnější a přirozenější práci s modely.

QVT se dělí do tří základních větví - Core, Relations, Operational. V našem projektu používáme QVT Operational, který slouží k mapování Model-To-Model (M2M). Více o QVT ve specifikaci jazyka [9].

5.2 OCL

Jazyk OCL [12] je čistě funkcionální specifikační jazyk určený k vyjádření invariantů. V jazyce OCL neexistuje žádná globální paměť, vyhodnocení výrazu je zcela bez vedlejších efektů. Stejná funkce se stejnými argumenty nám dává vždy stejný výsledek. Zápisy v OCL chápeme jako specifikaci pro programy, které budou odpovídající integritní omezení zajišťovat.

OCL je silně typovaný jazyk, tj. každý výraz v jazyce OCL má definován typ. To umožňuje silnou statickou typovou kontrolu zapsaných omezení při jejich interpretaci. OCL má předdefinovánu sadu primitivních typů. Jsou to zejména Integer, Boolean, String, Real, Unlimited Integer, ze kterých lze vytvářet složené typy jako rozmanité druhy kolekcí: množina (Set), multi-množina (Bag), obecná kolekce (Collection), posloupnost (Sequence) atd.

Asi nejsilnější zbraní OCL je práce s kolekcemi. OCL zavádí šipkovou a tečkovou notaci. Šipková je určena pro kolekce a tečková pro instance. V příkladu níže je nejprve nalezen potřebný sloupec metodou findColumn, která nám vrátí objekt typu Column. Každý sloupec si v naší aplikaci drží kolekci omezení (constraints), která jsou nad ním aplikována. V uvedeném dotazu se ptáme, zda je nad sloupcem aplikováno omezení s daným názvem (rName). Šipka zde znamená porovnání jména každého prvku kolekce.

```
self.findColumn(sName, tName, cName).constraints->exists(name = rName);
```

5.3 Komponenta - Databázová evoluční transformace

Tato komponenta je implementací databázové evoluční transformace, již z několika pohledů zde nastíněné.

Vstupem transformace je databázový model, k němu je připojena sekvence operací, které se nad modelem mají vykonat. V hlavičce transformace se získá kořenový element z modelu (*ModelRoot*) a kolekce operací. Dále se již pouze spouští jednotlivé operace v tom pořadí, v jakém byly připojeny k modelu.

Jak již bylo v kapitole 4.3, každá operace se skládá ze dvou částí - validační a mapovací. Nejprve musí proběhnout validace, která nás ujistí, že operace nedostane model do nekonzistentního stavu. Pokud operace projde validačním procesem, začíná mapovací část, která změnu provede na modelové úrovni. Pokud operace neprojde validací, celý evoluční proces je ukončen a provedené změny odstraněny. Model ani data se nijak nezmění. Na příkladu níže je ukázka kompletního mapování operace *RemoveTable*.

```
query RDB::RemoveTable::isValid(gen : RDB::ModelGeneration) : Boolean {
  return gen.isTableInSchema(self.owningSchemaName, self.name)
    and gen.findTable(self.schemaName, self.name).columns->size() = 0
    and gen.findTable(self.schemaName, self.name).constraints->size() = 0;
}
mapping RDB::RemoveTable::apply(inout gen : RDB::ModelGeneration) {
  var s : RDB::Schema := gen.findSchema(self.schemaName);
  s.tables:= s.tables->excluding(gen.findTable(self.schemaName, self.name));
}
```

Ve validační části je patrné, že vedle banalit (existence tabulky ve schématu dané generace) můžou být validační pravidla i méně intuitivní. Například pokud chceme odstranit tabulku z dané generace, musíme nejprve smazat všechny sloupce, které patří tabulce a zároveň všechna omezení nad tabulkou. Mapovací část operace už pouze vyřadí zcela prázdnou tabulku ze schématu dané generace.

Pokud bychom chtěli tabulku odstranit v reálném ostrém provozu, musíme před použitím operace *RemoveTable* vytvořit sekvenci operací, které zajistí splnění validačních omezení u *RemoveTable* (nejprve odstranění omezení, potom sloupců, ...).

Je důležité zdůraznit, že pokud tabulka neexistuje v daném schématu nyní, neznamená to, že v něm neexistovala někdy dříve, nebo nebude existovat někdy později. Tento paradox je velice zásadní. Jestliže například vytváříme sekvenci navzájem velice úzce provázaných operací, musíme myslet na to, že entity, se kterými v modelu budeme chtít pracovat, tam ještě nemusejí existovat (nebo už nemusejí existovat).

Datové operace zaměřené na ztrátu informace mají validační i mapovací část velice prostou. Často jde pouze o triviální kontroly, po kterých se zastavení celé evoluce ani neočekává. Síla datových operací se totiž projevuje na nižších vrstvách celého systému. Pokud například nesedí počty instancí, operace vygeneruje varovný signál, který nezastaví modelovou evoluci, ale zastaví celý proces generování SQL. Všechny vygenerované SQL příkazy jsou poté zahozeny.

Evoluční transformace je na přiloženém médiu *bp/populate_generations_rdb.qvto*.

Kapitola 6

Model-to-Text transformace

Model-to-Text vertikální transformace jsou změny probíhající mezi různými vrstvami v rámci MDA. Jinými slovy přecházíme z PSM do ISM, tedy ke konečnému generování kódu.

Každá operace, kterou provedeme na modelové úrovni, má svou šablonu na nejnižší generující vrstvě. Při namapování této operace se automaticky vygeneruje vhodné SQL, které reprezentuje modelovou operaci nad skutečnou databází.

Databázový meta-model je dostatečně obecný, aby byl schopný reprezentovat většinu z existujících relačních databází. Pro ověření myšlenky jsme se rozhodli pro databázi typu PostgreSQL a jejich syntaxi.

6.1 Xtend 2

Pro Model-to-Text (M2T) transformace existuje celá řada jazyků. Jazyky z rodiny Xtext [7] byly vybrány z důvodu jednoduchosti syntaxe, integrovanosti do prostředí Eclipse a zcela dostačujícím vlastnostem pro naše účely.

Xtend je staticky-typový funkcionální a objektově orientovaný jazyk postavený nad jazykem JAVA. Nesnaží se ho žádným způsobem nahradit, ale spíše vylepšit a odstínit od zbytečného psaní kódu. Syntaxe je tedy velice podobná jako v JAVA. Celý kód psaný v Xtend jazyku se pak překládá přímo do lehce čitelného JAVA zdrojového kódu. Nejsilnější stránkou Xtend jazyka je polymorfismus nad parametrem a obohacování řetězců (rich-string). Více o Xtend v [6].

Polymorfismus nad parametrem funguje obdobně jako přetěžování metod v JAVA, pouze s tím rozdílem, že předem nemusíme znát typ parametru. Příklad níže ukazuje situaci, kdy se dá polymorfismus nad parametrem elegantně využít.

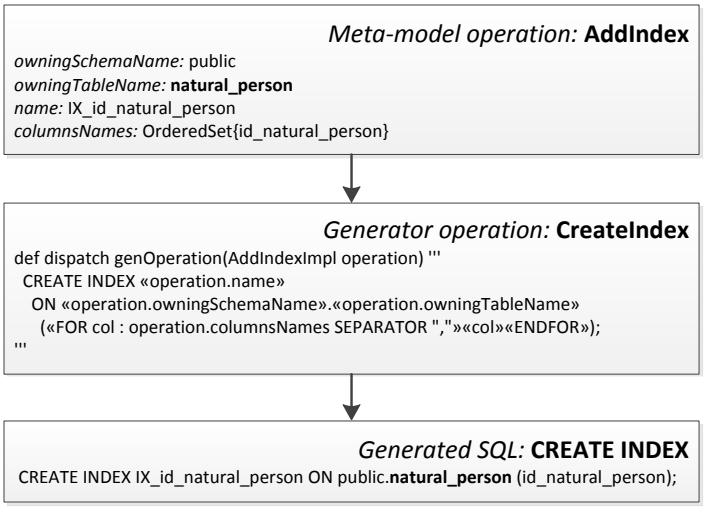
```
def toplevelGenerator(ArrayList<ModelOperationImpl> operations) {
  for (op : operations)
    op.genOperation
}

def dispatch genOperation(AddPrimaryKeyImpl operation) ''' KOD OPERACE '''
def dispatch genOperation(AddForeignKeyImpl operation) ''' KOD OPERACE '''
```

Metoda *toplevelGenerator* obsahuje cyklus, který prochází postupně všechny operace a spouští na nich metodu *genOperation*. Polymorfismus v JAVA se rozhoduje podle typu objektu, nad kterým metodu voláme. V jazyku Xtend se také rozhoduje podle typu objektu, ale v parametru metody. Pokud chceme v Xtendu začít pracovat s parametrovým polymorfismem, musíme do hlavičky metody napsat klíčové slovo dispatch.

Další obrovskou výhodou jazyka Xtend je obohacování řetězců. Obohacený řetězec začínáme a ukončujeme trojitým apostrofem. Kouzlo obohacování spočívá v tom, že přímo do řetězce můžeme psát cykly, nebo podmínky. Program se rozhodne, jak bude výsledný text vypadat až přímo ve chvíli, kdy ho bude generovat. Příklad níže ukazuje generování SQL pro vytvoření indexu v databázi.

```
/**
 * CREATE INDEX
 * To create a B-tree index on the column title in the table films:
 * >> CREATE INDEX title_idx ON films (title); <<
 * @param AddIndexImpl operation : operation of type AddIndexImpl
 */
def dispatch genOperation(AddIndexImpl operation) '''
  CREATE INDEX «operation.name»
    ON «operation.owningSchemaName».«operation.owningTableName»
      («FOR col : operation.columnsNames SEPARATOR ","»«col»«ENDFOR»);
'''
```

For cyklem zde vypíšeme všechny sloupečky, které budou součástí indexu. Na obrázku níže 6.1 jsou znázorněny různé podoby téže operace na všech vrstvách.



Obrázek 6.1: Různé podoby téže operace skrze vrstvy aplikace

6.2 Komponenta - Generátor SQL

Generátor SQL je implementací PSM-to-ISM transformace. Na vstupu má modelové informace a na výstupu textovou podobu operace ve formě SQL příkazu. Generátor je tedy pouze mostem mezi modelem a databází, proto je nežádoucí, aby obsahoval nějakou vlastní logiku, která by nebyla definovaná z vyšší vrstvy. Všechny potřebné informace musí generátor dostat z modelu na vstupu a podle nich generovat SQL. Například i přesto, že máme v systému pouze jednu globální sekvenci, není možné po generátoru požadovat, aby si dohledal její název a aplikoval jej nad sloupcem. V tomto případě musí být součástí vstupu do generátoru i informace o názvu sekvence.

Komponenta převezme finální verzi modelu po evoluci s připojenou sadou již vykonaných operací. Každá operace má definovanou šablonu, podle které se vygeneruje SQL. Jediné, co v šabloně schází, jsou konkrétní údaje z operace (například název nové tabulky). Modelové informace se do obohaceného řetězce vkládají za pomoci takzvaných francouzských uvozovek. Práce s modelovými informacemi je patrná na příkladu níže, kde vytváříme novou tabulku.

```
/**
 * CREATE TABLE
 * So to create a table in the new schema, use:
 * >> CREATE TABLE myschema.mytable (...); <<
 * @param AddTableImpl operation : operation of type AddTableImpl
 */
def dispatch genOperation(AddTableImpl operation) '''
  CREATE TABLE «operation.owningSchemaName».«operation.name» ();
'''
```

Vstupem operace je objekt typu *AddTableImpl*, což je JAVA třída reprezentující modelovou operaci.

Jakmile vyplníme obohacený řetězec, za pomoci běžné JAVA třídy File vytvoříme soubor a text do něj zapíšeme. Soubory pojmenováváme číselně a inkrementálním způsobem. Aby však nedošlo ke špatnému řazení (1, 11, 12, 2, ...), startovní hodnotou je 100 (100, 101, ...). První vygenerované SQL se pak jmenuje "100.SQL".

Generátor nevytváří pouze .SQL soubory. Pokud nastane situace, kdy potřebujeme nejprve otestovat, zda je stav instancí v databázi vhodný pro vykonání samotného příkazu, vygeneruje se nejprve .q soubor. Po vykonání takovéhoto dotazu očekáváme, na rozdíl od .SQL souborů, odpověď od databáze v podobě hodnot TRUE nebo FALSE. Pokud nám databáze na daný dotaz vrátí hodnotu FALSE, celý proces posílání SQL se ukončí a všechny dosavadní změny v DB nebudou uloženy. Pokud dostaneme TRUE, je vše podle pravidel a může se s databází komunikovat dál.

Na příkladu níže je šablona pro operaci *CopyInstances*, která má za úkol zkopírovat instance z jedné tabulky do druhé. Základní problém spočívá v tom, že je třeba, aby byl v obou tabulkách stejný počet instancí (nemusíme pak řešit výchozí hodnoty sloupců, nebo zahazování dat). Operace se uživatele ptá, zda si přeje být striktní (strict) či tolerantní (tolerant). Striktní stav znamená, že se operace vykoná pouze pod podmínkou, že mají tabulky shodný stav instancí. Tolerantní stav nic netestuje, protože má dovoleno zahazovat přebytečná data.

```
/**
 * COPY INSTANCES
 * This operation copy data from one column to another.
 * Target and source column can be in the same table.
 * @param CopyInstancesImpl operation : operation of type CopyInstancesImpl
 *
 */
def dispatch genOperation(CopyInstancesImpl operation){
  if(operation.type.toString().equals("strict")){
    generateFile(operation.getFileName(".q"), this.isSameTableSize(...);
  }
  return '''UPDATE «operation.owningSchemaName».«operation.targetTableName»
    SET «operation.targetColumnName» = (SELECT «operation.sourceColumnName»
    FROM «operation.owningSchemaName».«operation.owningTableName»);''';
}
```

Stav tolerance a striktnosti se tedy liší pouze v tom, zda se vykoná test na počet instancí v tabulkách.

Generátor SQL je na přiloženém médiu *bp/Generator.xtend*.

Kapitola 7

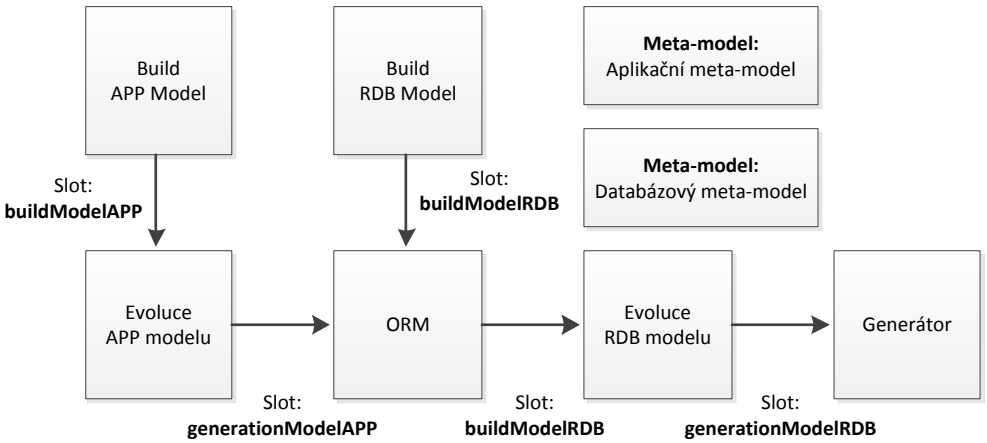
Pracovní tok frameworku

Pracovní tok (workflow) nám spojuje všechny komponenty do jednoho uceleného organismu. Komponenty jsou řazeny sériově (za sebou) - výstup jedné komponenty je vstupem pro další. Sériovost spojení komponent zcela neodpovídá původní myšlence, kdy se vykoná jedna operace skrze všechny vrstvy až do databáze a následně další. Díky sériovosti vše probíhá blokově (nejprve aplikační evoluce, pak ORM, ...).

Komponenty spolu komunikují skrze sloty, do kterých ukládají modely, které jsou definovány naším aplikačním nebo databázovým meta-modelem.

Jak již bylo řečeno, celá naše aplikace se skládá z volně obměnitelných komponent, psaných v různých jazycích a sloužících k různým účelům (generující, transformující, modelující). Tyto komponenty skládáme v pracovním toku v námi zvoleném pořadí.

Pracovní tok naší aplikace je pro lepší představu na obrázku níže 7.1.



Obrázek 7.1: Struktura pracovního toku

První komponentou toku je transformace sestavující (nebo upravující již existující) aplikační model. Po vytvoření aplikačního modelu se spustí další komponenta, která pouze vytvoří základní stav databázového modelu (Základní stav obsahuje pouze entity *ModelRoot*, *ModelGeneration*, *Schema* a *Sequence*).

Po vytvoření aplikačního a databázového modelu se spustí komponenta objektově-relačního mapování, která ze sady aplikačních operací vytvoří operace databázové a připojí je k databázovému modelu. Takto upravený databázový model se předá komponentě pro databázovou evoluční transformaci. Poslední hlavní komponentou je pak generátor, který z operací v databázovém modelu vygeneruje SQL dotazy.

Pracovní toky jsou k nahlédnutí na přiloženém médiu *bp/build.mwe2*, *bp/first.mwe2* a *bp/second.mwe2*.

7.1 MWE2 - The Modeling Workflow Engine 2

Jazyk MWE2 je deklarativní, externě nastavitelný nástroj, který slouží k popisu libovolné objektové struktury. Díky jednoduché syntaxi je MWE2 velice silným nástrojem při vytváření pracovních toků a testování modelových aplikací.

Pracovní tok MWE2 je obvykle složený z několika komponent, které na sebe nějakým způsobem vzájemně působí. K předdefinovaným komponentám MWE2 patří například čtení modelu, práce s transformacemi, práce se souborovým systémem (čištění složek, vytváření či odstraňování souborů). Více o MWE2 na [5].

Pokud základní komponenty MWE2 nestačí, lze si naprogramovat své vlastní komponenty v jazyce JAVA. V našem projektu využíváme vedle předem nadefinovaných i komponenty naprogramované společností CollectionsPro, s.r.o.

Pokud chceme v pracovním toku pracovat s meta-modely, musíme nejprve načíst JAVA balíček meta-modelu do beany. Uložení aplikačního a databázového meta-modelu je znázorněno na příkladu níže.

```
/* Define of APP meta-model */
  bean=org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri=".."
    registerGeneratedEPackage = "mm.app.AppPackage"
  }
/* Define of RDB meta-model */
  bean=org.eclipse.emf.mwe.utils.StandaloneSetup {
    platformUri=".."
    registerGeneratedEPackage = "mm.rdb.RdbPackage"
  }
```

Jednou z nejdůležitějších vlastností MWE2 jsou již zmiňované sloty. Sloty nám slouží jako přepravky pro modely mezi jednotlivými komponentami. Na příkladu níže je transformační komponenta pro databázovou evoluci.

```
component = QVTOExecutor {
  inputSlot = "buildModelRDB"
  transformationFile = "../populate_generations_rdb.qvto"
  outputSlot = "generationModelRDB"
}
```

Transformace si na vstup vezme model ze slotu input a stav modelu po evoluci uloží na output slot. Sloty mohou být nakombinované i jako vstupně-výstupní.

7.2 Ostatní komponenty aplikace

Náš pracovní tok obsahuje i sadu pomocných komponent, které nám zjednodušují rutinní práci (přeposílání SQL do databáze, čištění adresářů, atd).

DirectoryCleaner a *Writer* jsou již integrované komponenty v MWE2. *DirecotryCleaner* je komponenta, která nám vyčistí složku, do které se budou později generovat SQL soubory. *Writer* slouží k zapisování modelů ze slotu do XMI souborů.

CommandLineExec je komponenta z knihovny společnosti CollectionsPro, s.r.o. Zastává pozici komunikátoru mezi systémovým příkazovým řádkem a pracovním tokem. Za pomoci této komponenty spouštíme naší poslední vrstvu aplikace pojmenovanou Komunikátor s databází, která posílá vygenerované SQL do databáze a kontroluje návratové hodnoty.

7.2.1 Komunikátor s databází

Komunikátor sice není komponentou v pravém slova smyslu, ale je nedílnou součástí komponenty *CommandLineExec*, bez které bychom ho nemohli propojit s celým naším systémem.

Komunikátor seřadí všechna vygenerovaná SQL podle pořadí (pořadí je definováno názvem souboru 100.SQL, 101.SQL, ...) a postupně je posílá do databáze. Pokud jediné SQL z celé řady selže, v databázi neproběhne jediná změna a vše je zahozeno.

Tato drobná aplikace má ještě jednu velice důležitou vlastnost. Při odeslání .q souboru kontroluje odpověď a v případě neúspěchu zastaví celý proces odesílání SQL souborů.

Tato nejnižší ale zároveň nejdůležitější komunikační a testovací vrstva je psaná v jazyce Ruby. Za její vytvoření vděčíme panu Ing. Ondřeji Mackovi, který nám tím umožnil plně testovat hotovou část aplikace.

7.3 Testování pracovním tokem

Testy je třeba provádět nad všemi komponentami na všech vrstvách aplikace. Musíme testovat správnost nově navržené meta-modelové operace, korektnost průběhu validace a mapování při evolučních transformacích, generování nových operací v ORM, správnost vygenerovaných SQL a v neposlední řadě stav instancí v databázi.

Díky jazyku MWE2 můžeme testovat všechny komponenty na jednom místě a plně automatizovat náš testovací proces.

7.3.1 Test evoluční databázové transformace

V evoluční transformaci je třeba testovat jak samotnou validační část operací, tak stav modelu po vykonání operace. Na příkladu níže je test stavu modelu po evoluci, který zjišťuje, zda mají třídy na aplikační úrovni správný počet ID sloupců.

```
if(self.testID()) then {
  log("Model invalid - uncorrect ID count in classes");
  validity := false;
}endif;

query app::ModelRoot::testID() : Boolean {
  var c : Sequence(Class) :=  self.modelGenerations->classes[Class];
  return not c->forAll(cls|cls.assertIDCondition());
}

query app::Class::assertID() : Boolean {
  if((self.properties->select(isID)->size() = 1 and self.parent = null)
   or (self.parent <> null
         and self.properties->select(isID)->size() = 0))then{
    return true;
  }endif;
  log("class " + self.name + " has incorrect ID properties);
  return false;
}
```

Pokud chceme testovat validační pravidla operací, vytvoříme si modelovou situaci a ověříme, že validační pravidlo operace zastavilo evoluční transformaci.

7.3.2 Test generátoru SQL a stavu instancí

Stěží by šlo definovat nějaké obecné pravidlo, které by ověřilo, zda jsou instance v databázi ve stavu, který jsme požadovali. Proto je důležité, aby naše testovací simulace obsahovala co nejvhodnější data a co nejpřesnější model, který by byl připraven přesně pro daný problém.

Pokud tedy chceme provést instanční test, nejprve si navrhneme základní testovací model a změnu, kterou na něm budeme chtít provádět. Model sestavíme na aplikační úrovni a skrze pracovní tok naší aplikace necháme vygenerovat tabulkovou strukturu, kterou pošleme do databáze v podobě SQL dotazů. Databázi zaplníme daty, které si předem připravíme. V tuto chvíli máme vytvořený výchozí stav, který je připraven na naši testovací transformaci. Transformaci pustíme nad výchozím stavem.

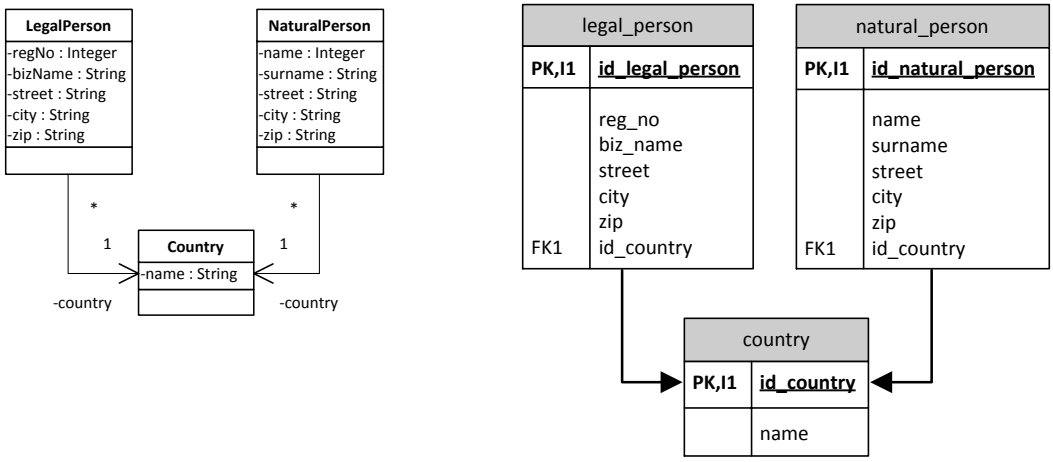
Kontrolu konzistence instancí provedeme porovnáním mezi očekávaným stavem a výstupním stavem transformace.

Jelikož by bylo náročné stále dokola vymýšlet výchozí model a instance, kterými jej zaplnit, vytvořili jsme si jeden výchozí model, který bude obsahovat většinu jevů, které v aplikaci mohou nastat. Nad tímto modelem pak spouštíme testované transformace.

Kapitola 8

Ukázkový příklad

Představme si v aplikaci situaci jako na obrázku 8.1. Zde vidíme třídu *LegalPerson* reprezentující právnické osoby a *NaturalPerson*, která představuje fyzické osoby. Třída *Country* je jakýmsi výčtem všech zemí.



Obrázek 8.1: Výchozí situace v aplikaci

Největší nevýhodou této struktury je duplikace kódu (city, street, zip a country jsou v obou třídách). Dále je problém s ukládací logikou dat. Jelikož jsou informace o adresách uloženy na dvou místech, bude v budoucnu nesmírně zatěžující prohledávat adresní prostor. Z těchto nevýhod vyplývá, že bude nutné model nějak transformovat. Z důvodu lepší názornosti bude transformace rozdělena na dvě části, kdy v každé proběhne jedno vylepšení.

8.1 Sestavení výchozího modelu aplikace

Abychom mohli začít model zdokonalovat, musíme ho nejprve postavit. K tomuto účelu využijeme pracovní tok, který jako vstupní aplikační model dostane sadu operací, které model postaví. Pro představu je část kódu pro sestavení modelu na příkladu uvedeném níže.

```
model.operations += addClass("Country", false,  APP::InheritanceType::joined);
model.operations += addProperty("Country", "name", "String");

model.operations += addClass("LegalPerson", false,  APP::InheritanceType::joined);
model.operations += addProperty("LegalPerson", "regNo", "Integer");
model.operations += addProperty("LegalPerson", "bizName", "String");
model.operations += addProperty("LegalPerson", "street", "String");
model.operations += addProperty("LegalPerson", "city", "String");
model.operations += addProperty("LegalPerson", "zip", "String");
model.operations += addProperty("LegalPerson", "country", "Country");
```

Po vytvoření aplikačního modelu se vytvoří i model databázový reprezentující skutečnou databázi, která se sestavila za pomoci vygenerovaných SQL. Do vytvořené databáze uložíme testovací instance. Například v tabulce *naturalperson* máme uloženy záznamy jako v tabulce 8.1.

id	name	surname	street	city	zip	country
11	"Peter"	"Jackson"	"Central Park St"	"New York"	"100 01"	1
12	"George"	"Clooney"	"S Orange Ave"	"Orlando"	"320 24"	2
13	"Jack"	"Daniels"	"Down St"	"Dublin"	"456 01"	5
14	"Thomas"	"White"	"R 32th St"	"Washington"	"983 43"	6
15	"Mel"	"Gibson"	"Up St"	"New Jersey"	"843 89"	8

Tabulka 8.1: Záznamy v tabulce naturalperson ve výchozím stavu

Podívejme se například na Petra Jacksona s identifikátorem 11. Vše, co o něm potřebujeme vědět, je zde v jednom řádku (jméno, příjmení, ulice,..). Po každé fázi transformace se podíváme, jak se Petrova data pohybují a tříští mezi různé tabulky a sloupce.

Všechna vygenerovaná SQL pro sestavení modelů jsou v přiloženém médiu v projektu *migdb.run /output_ sql_ build*. Aplikační i databázový model jsou k nahlédnutí v *migdb.run /output_ xmi_ build*.

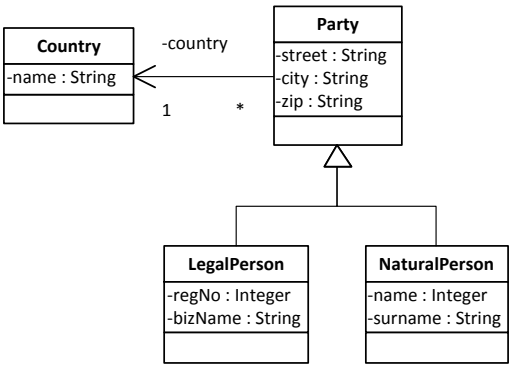
8.2 První transformace - Sestavení hierarchie

Nejprve se pokusíme zbavit duplikace kódu a nelogického ukládání adres na dvě místa. Jedno z řešení je vytvoření společného předka *Party* pro třídy *NaturalPerson* a *LegalPerson*, do kterého se přesunou adresní informace.

Vstupem do pracovního toku je v tuto chvíli výchozí stav modelu a sada operací, které budeme nad modelem chtít vykonávat. Volání operací probíhá obdobně jako u sestavování modelu, viz příklad níže.

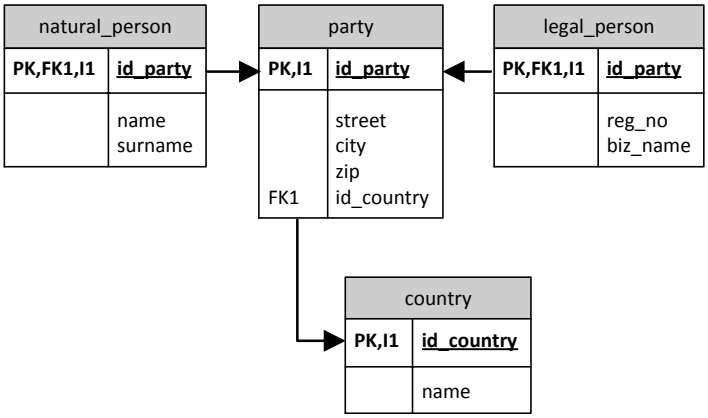
```
model.operations += addClass("Party", false, InheritanceType::joined);
model.operations += addProperty("Party", "street", "String");
model.operations += addProperty("Party", "city", "String");
model.operations += addProperty("Party", "zip", "String");
model.operations += addProperty("Party", "country", "Country");
model.operations += setParent("LegalPerson", "Party",
                     OrderedSet{"street", "city", "zip", "country"});
model.operations += setParent("NaturalPerson", "Party",
                     OrderedSet{"street", "city", "zip", "country"});
```

Aplikační model po takto vykonané transformaci je na obrázku 8.2.



Obrázek 8.2: Aplikační model po první transformaci

Databázový model po transformaci vypadá jako na obrázku 8.3.



Obrázek 8.3: Databázový model po první transformaci

Nyní se podíváme, jak se přemístila data Petra Jacksona. Petrova data se rozdělila do dvou tabulek. V tabulce *Party* 8.2 jsou nyní uložena jeho adresní data a v tabulce *naturalperson* 8.3 zůstalo jeho jméno a příjmení.

Hierarchická vazba mezi tabulkami vznikla skrze primární klíče. To znamená, že instance má v rámci celé hierarchie stejný identifikátor ve všech tabulkách (Petr má indikátor 11).

id	street	city	zip	country
11	"Central Park St"	"New York"	"100 01"	1
12	"S Orange Ave"	"Orlando"	"320 24"	2
13	"Down St"	"Dublin"	"456 01"	5
14	"R 32th St"	"Washington"	"983 43"	6
15	"Up St"	"New Jersey"	"843 89"	8
16	"Nice Street"	"Redmond"	"757 65"	10
17	"E 12th St"	"Sillicon Valley"	"346 09"	8

Tabulka 8.2: Záznamy v tabulce party

id	name	surname
11	"Peter"	"Jackson"
12	"George"	"Clooney"
13	"Jack"	"Daniels"
14	"Thomas"	"White"
15	"Mel"	"Gibson"

Tabulka 8.3: Záznamy v tabulce naturalperson po první transformaci

Všechna vygenerovaná SQL pro první transformaci modelů jsou v přiloženém médiu v projektu *migdb.run /output_sql_first*. Aplikační i databázový model jsou k nahlédnutí v *migdb.run /output_xmi_first*.

8.3 Druhá transformace - Extrahování třídy

V tomto okamžiku je model zcela vyhovující aktuálním požadavkům, problém by ale nastal, pokud bychom chtěli u jedné firmy uvést dvě adresy. Představme si, že budeme chtít mít u každé firmy kontaktní a fakturační adresu.

Vstupem do pracovního toku je v tuto chvíli stav modelu po první transformaci a sada nových operací, které nám zajistí možnost mít dvě adresy u jedné firmy. Aplikační a databázový model po druhé transformaci vypadá jako na obrázku 8.4.

Data Petra Jacksona se nám rozdělila do tabulek *address* 8.4, *party* 8.5 a *legalperson* 8.6. V tabulce *legalperson* máme stále jméno a příjmení. V tabulce *party* máme ukazatele na fakturační a kontaktní adresy. Při transformaci se všechny existující adresy automaticky změnily na obytné. Záznam o Petrově adrese se přesunul do tabulky *address*.

Po dvou transformačních procesech se změnil aplikační model do požadované podoby a zároveň se neztratila žádná data. Konzistence modelů i databáze zůstala zachována stejně jako informace obsažená v záznamech.

id	street	city	zip	country
18	"Nice Street"	"Redmond"	"757 65"	10
19	"E 12th St"	"Sillicon Valley"	"346 09"	8
20	"Central Park St"	"New York"	"100 01"	1
21	"S Orange Ave"	"Orlando"	"320 24"	2
22	"Down St"	"Dublin"	"456 01"	5
23	"R 32th St"	"Washington"	"983 43"	6
24	"Up St"	"New Jersey"	"843 89"	8

Tabulka 8.4: Záznamy v tabulce address

id	residentialaddress	contactaddress
11	20	
12	21	
13	22	
14	23	
15	24	
16	18	
17	19	

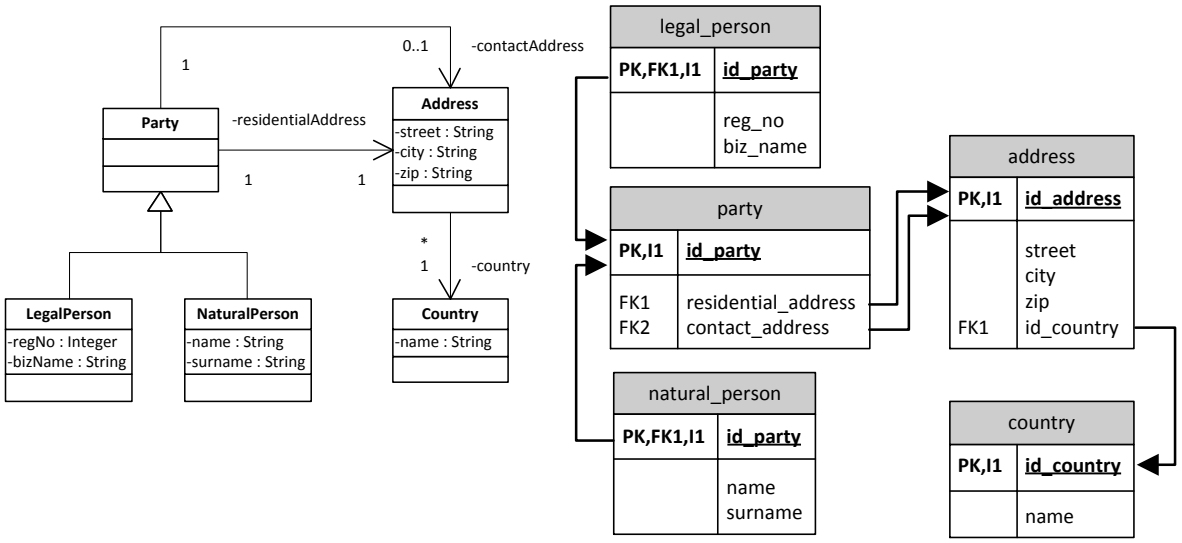
Tabulka 8.5: Záznamy v tabulce party po druhé transformaci

id	name	surname
11	"Peter"	"Jackson"
12	"George"	"Clooney"
13	"Jack"	"Daniels"
14	"Thomas"	"White"
15	"Mel"	"Gibson"

Tabulka 8.6: Záznamy v tabulce naturalperson po druhé transformaci

Při samotných transformacích je také velice zajímavé sledovat vlastní generování databázových operací z operací aplikačních. Tento problém však spadá do problematiky ORM, o které píše ve své práci Jiří Ježek.

Všechna vygenerovaná SQL pro druhou transformaci jsou v přiloženém médiu v projektu *migdb.run /output_ sql_ second*. Aplikační i databázový model jsou k nahlédnutí v *migdb.run /output_ xmi_ second*.



Obrázek 8.4: Aplikační model po druhé transformaci

Kapitola 9

Shrnutí

9.1 Zhodnocení nástrojů

Aplikace je vyvíjená v IDE Eclipse Modeling Tools verze Indigo, do kterého byly doinstalovány potřebné moduly a podpora pro využívané jazyky. Celý proces vývoje musel probíhat v prostředí Linux, protože MWE2 neumí pracovat s adresářovou strukturou Windows.

Vývoj řízený modelem s sebou nese celou řadu problémů (začínali jsme s prakticky nulovou zkušeností s tímto vývojem). V prvních chvílích bylo obtížné pracovat s celou řadou nových nástrojů, jazyků, metodik a v neposlední řadě s operačním systémem Linux. Překlenout toto stádium a začít přemýšlet nad samotným problémem nám trvalo velice dlouho.

Velikou nevýhodou vývoje řízeného modelem je celková nevyzrálost. Snad všechny nástroje (samotný Eclipse nevyjímaje) jsou ještě ve vývoji. Komunita kolem programovacích jazyků a technologií je velice malá, mnohdy žádná. Často je velký problém nalézt i takzvaný "hello world"tutoriál, který by programátorům pomohl s prvními kroky v novém terénu.

Několik příkladů: V jazyce QVT, aniž by to bylo někde uvedeno, z nějakého zvláštního důvodu nefunguje import QVT souborů, které neleží ve stejné složce jako QVT, do kterého je chceme importovat. V jazyce Xtend sice funguje vše, co je naimplementováno, bohužel až do veze 2.0 například nebyly zprovozněny atributy třídy, což pro jazyk, postavený nad jazykem JAVA a vytvořený k účelu jazyk JAVA zpříjemnit, není dobrá vlastnost.

Samozřejmě, že vývoj řízený modelem stojí za překlenutí prvotních problémů. Nejúžasnější vlastností je automatické generování tisíců řádků kódu z pouhé hrstky informací, které programátor vytvoří. Odpadá tím zdlouhavé psaní samotného kódu a eliminují se tak i zbytečné chyby. Dokonce i výše haněné jazyky QVT a Xtend velice zjednodušují práci a usnadňují programátorům práci s modely.

9.2 Zhodnocení frameworku

Po přibližně roce a půl vývoje jsme dosáhli prvních výsledků v podobě jednoduchých transformací napříč celou aplikací. Je to pouze malý úspěch ve srovnání s tím, co nás ještě čeká, ale na druhou stranu je to zásadní mezník ve vývoji. V tuto chvíli už nemusíme přemýšlet,

zda je náš způsob vývoje nevhodný, nebo vybrané technologie nedostačující. Aplikace sice umí pracovat pouze ve velice laboratorních podmínkách, ale na druhou stranu naším úkolem je už jen rozšiřovat funkcionalitu za dveře laboratoře.

Tím, že na aplikaci pracovalo v průběhu vývoje hodně studentů, kteří až do nedávna netušili, jakým směrem se pořádně mají vydat, obsahuje aplikace spoustu již nepoužívaných metod, souborů, tříd či projektů, které by si zasloužilo jednou provždy smazat. Ze stejného důvodu by si aplikace zasloužila i pořádný refaktoring kódu a meta-modelů.

V tuto chvíli jsem schopni skoro stoprocentně model aplikace od nuly sestavit a zbourat. Je jasné, že toto nebyl primární úkol ani cíl, je ale patrné, že nám všechny komponenty spolupracují přesně tak, jak by měly. Samotná evoluce je již po malých krůčcích také možná, testováním jsem ale přišli na to, že často preferujeme spíše komplexnější větší operace, které vytvoří kontext mezi oněmi malými krůčky. Naším dalším úkolem tedy bude vytvoření množiny komplexních operací, které by mohly být používány v praxi.

Jelikož je náš nápad na rozdělení migrace databáze na drobná evoluční stádia unikátní, byli jsme pozváni, abychom ve složení Pavel Moravec, David Harmanec, Petr Tarant a Jiří Ježek prezentovali naši myšlenku na konferenci Code Generation 2012 [11], která se konala letos v anglickém Cambridge. Reakce na náš nápad i samotnou přednášku byly pozitivní a myslím, že mohu říci, že nám všichni kolegové drží palce, ať dorazíme ke zdárnému cíli.

9.3 Budoucí rozvoj

Budoucí směr na vývoji projektu jsem již naznačil dříve. Nyní je velice důležité vytvořit takovou množinu operací, aby stačily k běžným úpravám aplikace a byly schopny vyhovět našemu cíli, aby se udržela co největší konzistence modelu a zachovalo se přesně tolik informací, kolik si uživatel přál.

Mezi menší, ale rozhodně velice potřebné úkoly pak patří refaktoring, vytvoření precizního testovacího systému a v neposlední řadě vytvoření kvalitního výchozího modelu, který by byl schopen pojmout vše, co potřebujeme do budoucna vyvíjet v operacích.

Kapitola 10

Závěr

Mým cílem na projektu byla práce na databázové vrstvě (databázový meta-model, komponenta evoluční databázové evoluční transformace a komponenta generátoru SQL) a vytvoření pracovního toku celého frameworku. Databázový meta-model, který je stěžejní pro mou část práce, je popsaný ve třetí kapitole. Databázová evoluční transformace, která je hybnou silou celého projektu, je vysvětlena i s potřebnými jazyky v kapitole čtvrté. Generování SQL souborů společně s úvodem do jazyka Xtend 2 je v kapitole páté. Jako poslední úkol jsem měl vytvořit pojidlo mezi všemi komponentami. Tím je vytvoření pracovního toku za pomoci jazyka MWE2, ten je vysvětlen v šesté kapitole, kde je i nastíněna spolupráce všech komponent projektu. V závěru práce je ukázka našeho frameworku v činnosti.

Za poslední půlrok jsem velice pokročil v práci na všech komponentách našeho frameworku. Podařilo se mi propojit a zprovoznit všechny komponenty, vytvořit funkční databázové operace skrze celý pracovní tok a v neposlední řadě pracovat i se samotnými instancemi v databázi. Díky tomu, že jsem propojil vše skrze celý framework, jsem nyní schopen zcela komplexního testování na všech vrstvách.

Literatura

[1] JEžEK, J. Modelem řízená evoluce objektů, 2012. Bakalářská práce.

[2] KATEřINA, H. *Hodnocení indexace, kvalita a konzistence indexace* [online]. 2006. [cit. 15. 4. 2012]. Dostupné z: <http://www.phil.muni.cz/kivi/clanky.php?cl=67>.

[3] KLEMšINSKý, P. *Diplomová práce Návrh systému pomocí MDA* [online]. 2009. [cit. 15. 4. 2012]. Dostupné z: <http://www.scribd.com/doc/28689537/18/Metamodelovani-a-MOF>.

[4] KOMUNITA. *Eclipse Modeling Framework Project (EMF)* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/modeling/emf/>.

[5] KOMUNITA. *MWE2 - The Modeling Workflow Engine 2* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/Xtext/documentation/2_0_0/118-mwe-in-depth.php>.

[6] KOMUNITA. *Xtend - Language made for JAVA developers* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/xtend/#intro>.

[7] KOMUNITA. *Xtext - Language development made easy* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.eclipse.org/Xtext/>.

[8] MERUNKA, V. *Objektové modelování*. Apla, 1th edition, 2008.

[9] OMG. *Meta Object Facility (MOF) 2.0 Query View Transformation Specification* [online]. 2011. [cit. 15. 4. 2012]. Dostupné z: <http://www.omg.org/spec/QVT/1.1/PDF/>.

[10] OMG. *MOF 2 XMI Mapping* [online]. 2011. [cit. 15. 4. 2012]. Dostupné z: <http://www.omg.org/spec/XMI/>.

[11] PAVEL MORAVEC, P. T. J. J. D. H. *A practical approach to dealing with evolving models and persisted data* [online]. 2012. [cit. 15. 4. 2012]. Dostupné z: <http://www.codegeneration.net/cg2012/sessioninfo.php?session=37>.

[12] RICHTA, K. *Jazyk OCL a modelem řízený vývoj* [online]. 2010. [cit. 15. 4. 2012]. Dostupné z: <https://www.ksi.mff.cuni.cz/~richta/publications/Richta-MD-2010.pdf>.

Příloha A

Databázové operace

A.1 Operace přidání

AddSequence Vytvoření nové sekvence

AddNotNullConstraint Vytvoření nového nenulového omezení

AddPrimaryKey Vytvoření nového primárního klíče

AddForeignKey Vytvoření nového cizího klíče

AddUniqueIndex Vytvoření nového unikátního indexu

AddIndex Vytvoření nového indexu

AddColumn Vytvoření nového sloupce

AddTable Vytvoření nové tabulky

AddSchema Vytvoření nového schématu

A.2 Operace mazání

RemoveTable Smazání tabulky

RemoveColumn Smazání sloupce

RemoveIndex Smazání indexu

RemovetableConstraint Smazání tabulkového omezení

RemoveColumnConstraint Smazání sloupcového omezení

RemoveDefaultValue Smazání výchozí hodnoty

RemoveSequence Smazání sekvence

A.3 Operace nastavení

RenameTable Přejmenování tabulky

RenameColumn Přejmenování sloupce

SetColumnDefaultValue Nastavení nové výchozí hodnoty pro sloupec

SetColumnDataType Nastavení datového typu sloupce

A.4 Datové operace

GenerateSequenceNumbers Generuje nová sekvenční čísla do sloupce

AddInstances Kopíruje instance v rámci hierarchie

HasNoInstances Kontroluje, zda má tabulka nějaké instance

HasNoOwnInstances Kontroluje, zda má tabulka nějaké vlastní instance

CopyInstances Kopíruje instance s nastavením striktnosti

InsertInstances Kopíruje instance ze zdrojových sloupců do cílových (nehierarchické)

Příloha B

Seznam použitých zkratek

ORM Object-relational mapping

OMG Object management group

MDA Model driven architecture

MDD Model driven development

CIM Computation independent model

PIM Platform independent model

PSM Platform specific model

ISM Implementation specific model

EMF Eclipse modeling framework

Ecore Core EMF

XML Extensible Markup Language

QVT Query, View, Transformation

M2M Model-To-Model

OCL Object Constraint Language

M2T Model-To-Text

SQL structured query language

IDE integrated development environment

XMI XML metadata interchange

⋮

Příloha C

Instalační a uživatelská příručka

Před vlastní prací s naší aplikací je třeba si připravit pracovní prostředí. Je nutné, aby byla aplikace testována pouze v operačních systémech Linux, nebo iOS. Eclipse pro svůj chod potřebuje v systému podporu jazyka JAVA. Dále je třeba mít nainstalovanou podporu Ruby[1], pod kterou nám běží naše komunikující aplikace. Jako databázový systém jsme zvolili PostgreSQL[2], kterému jsme také přizpůsobili syntaxi generovaných SQL příkazů, proto je nutné mít nainstalovaný právě tento databázový systém.

Přiložené médium obsahuje IDE Eclipse Modeling Tools verzi Indigo, která je již plně nakonfigurovaná pro všechny technologie. Stačí tedy pouze zkopírovat Eclipse z disku do počítače a spustit pomocí souboru Eclipse.exe.

Na médiu je také celý program, který sestává z několika projektů, které je v prostředí Eclipse třeba importovat (*File− >Import− >Existing Projects into Workspace*).

Po uložení všech projektů do prostředí Eclipse už stačí pouze spustit jakýkoliv pracovní tok v projektu *migdb.run*, který je spouštěčem všech komponent.

Kompletní informace o instalaci a nastavení jednotlivých komponent jsou na Gitu projektu Migrace databáze[3]. Na stránkách projektu jsou k náhledu zápisy ze schůzí, uzavřené a probíhající issues, naše milestones a wiki stránka.

Další informace k ukázkovému příkladu z poslední kapitoly jsou na Git page projektu[4].

[1] Ruby: http://www.ruby-lang.org/en/
[2] ProstgreSQL: http://www.postgresql.org/
[3] https://github.com/migdb/migdb
[4] http://migdb.github.com/migdb/

Příloha D

Obsah přiloženého CD

eclipse Nakonfigurované vývojové prostředí Eclipse Modeling Tools

migdb Náš framework

bp Vybrané soubory z mé práce

text Zdrojové kódy k textu bakalářské práce