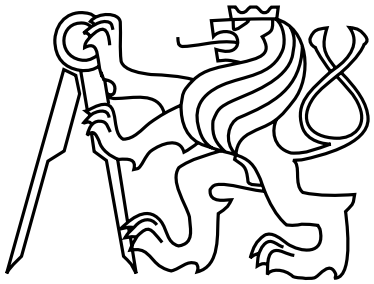
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

Modelem řízená evoluce objektů

Jiří Ježek

Vedoucí práce: Ing. Ondřej Macek

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

24. května 2012

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Ondřeji Mackovi a pánům Pavlu Moravcovi a Davidu Harmancovi, Ph.D. za jejich cenné rady a čas, který mi věnovali. Nakonec děkuji kolegovi Petru Tarantovi 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 24. 5. 2012 .....

Abstract

Object-relational mapping is used to speed up the development of software applications. With regards to their evolution, it can become ineffective and the whole process need be executed manually. Our project, which consists in an automated evolution of database schemes focuses on solving this problem and a related data migration in virtue of changes of an object layer. My duty is especially the design and implementation of an application meta-model and a related evolutional transformation, followed by a definition of operations designed for this layer and finally a implementation an object-relational mapping of these operations from the object layer to the database scheme. From the whole project, which my work is part of, we primarily promise a more effective database migration than any which is provided by our competitors.

Abstrakt

Objektově-relační mapování se využívá k urychlení vývoje softwarových aplikací. Přistoupíme-li k jejich evoluci, stává se neúčinným a celý proces je nutné vykonat ručně. Náš projekt se pak zaměřuje na řešení tohoto problému, které spočívá v automatizované evoluci databázových schémat a s tím související migraci dat na základě změn objektové vrstvy. Mojí povinností je zejména návrh a implementace meta-modelu objektové vrstvy a s ním spjaté evoluční transformace. Dále pak definice operací pro tuto vrstvu určených a konečně implementace objektově-relačního mapování operací z objektové vrstvy na databázové schéma. Od celého projektu, kterého je má práce nedílnou součástí, si slibujeme především efektivnější migraci databáze, než kterou nabízejí konkurenční řešení.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Pro urychlení vývoje softwarových aplikací se v praxi běžně využívá výhod objektově-relačního mapování (ORM). Ovšem nástroje, které tuto techniku ovládají, vykonají svou práci především ve fázi tvorby, neboť na obtížnější evoluci již nestačí. Evoluční proces pak musí probíhat zcela manuálně, neboť nesmíme opomenout migraci dat, která se stále ukazuje jako největší problém.

Prostřednictvím našeho projektu se snažíme vyvinout framework, který si se zmíněnou evolucí aplikace hravě poradí. Nesoustředíme se ale pouze na evoluci databázových schémat, dbáme totiž také na korektní přenos instancí a posléze se pokoušíme celý proces automatizovat. V našem případě to znamená, že změnu zaznamenáváme již na objektové vrstvě, odkud ji propagujeme skrz databázové schéma až do databáze samotné.

Celý vývoj je řízený modelem, s čímž souvisí i několik nových technologií a jazyků, které jsme se museli časem naučit používat. Také by se slušelo poukázat na experimentální povahu a řekněme vědecký potenciál projektu, který jsme si také ověřili na mezinárodní konferenci Code Generation [2], která se letos v březnu konala v britském Cambridge a které jsme mohli být součástí.

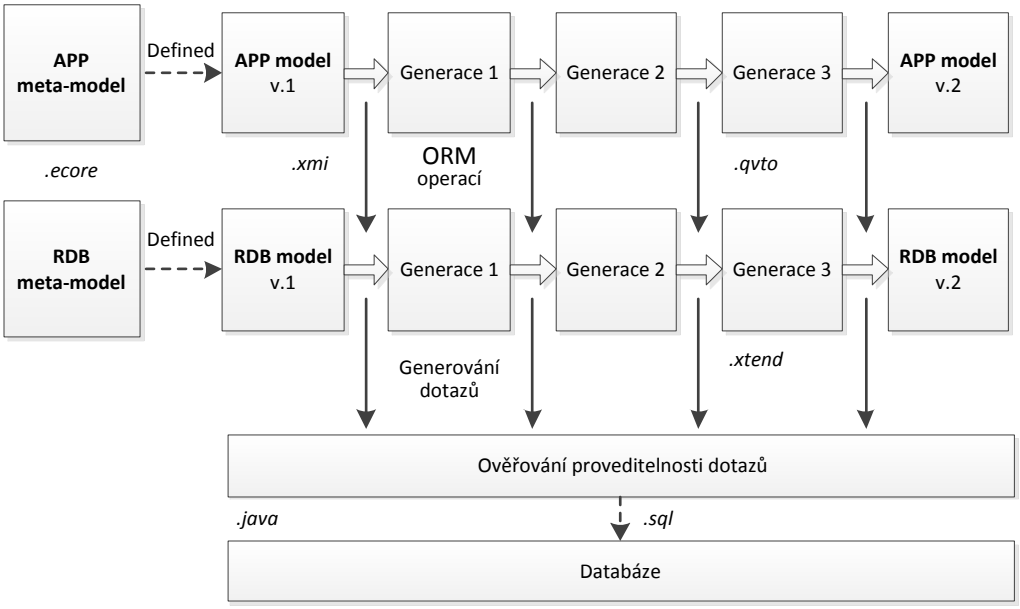
Dále je třeba zmínit, že projekt není individuální záležitostí, ale podílí se na něm hned několik členů. Vezmeme-li v potaz pouze ty v současnosti aktivní, autor této práce je jedním z nich. Druhým členem je pak Petr Tarant [12], který má na starosti především implementaci meta-modelu a příslušné evoluční transformace databázové vrstvy a generování dotazů do databáze. Mým úkolem je pak realizace meta-modelu a obdobné evoluční transformace objektové vrstvy a konečně ORM operací pro databázové schéma.

Text je členěn následovně. V kapitole 4 si popíšeme stěžejní vlastnosti již zmíněného meta-modelu objektové vrstvy. Dále pak v kapitole 5 nastíníme hlavní princip operací a jejich využití. Na evoluční transformace se blíže podíváme v kapitole 6 v sekci 6.1 a diskutované ORM operací pak v téže kapitole v sekci 6.2. Ještě předtím se ovšem ve dvou úvodních kapitolách zaměříme na podrobný popis problému a jeho navrhované řešení.

Kapitola 2

Popis problému

Framework je rozdělen do tří základních vrstev, jak je patrné z obrázku 2.1. První vrstva se nachází na úrovni entit, popisuje tedy objektovou strukturu dané aplikace. Zbylé dvě vrstvy jsou reprezentovány samotnou databází. První z nich znázorňuje databázové schéma, které odpovídá objektové vrstvě výše a popisuje strukturu databáze, druhá vrstva pak skýtá úložiště jednotlivým instancím.



Obrázek 2.1: Struktura frameworku.

Životní cyklus 1 popisuje v prvních třech krocích tvorbu a využití aplikace. Z praxe ovšem víme, že požadavky na aplikaci se mění již během vývoje či jsou vyžadovány zásahy do již hotové verze za účelem rozšíření funkcionality. Tak či tak je obvykle nutné přistoupit k evoluci jak samotných entit, tak databázového schématu a posléze i dílčích instancí.

Algoritmus 1 Životní cyklus tvorby, využití a evoluce běžné aplikace.

1: Definice entit na objektové vrstvě
2: Tvorba databázového schématu dle daných entit
3: Ukládání instancí do databáze
4: **if** Vyžádána evoluce **then**
5: Redefinice entit na objektové vrstvě
6: Opětovná tvorba databázového schématu dle redefinovaných entit
7: Sběr potřebných informací pro migraci dat a příprava migračního skriptu
8: Ověření proveditelnosti migračního skriptu
9: Migrace dat
10: **end if**

Evoluční proces reprezentují kroky 5-9 životního cyklu 1. Tento proces je vyvolán již při změně objektové vrstvy a následně je propagován přes databázové schéma až do databáze s uloženými daty. Evoluce tedy neovlivní pouze každou dílčí vrstvu, ale může mít také vliv na samotné instance uložené v databázi. V takovém případě je nutná jejich změna a následná migrace z původního databázového schématu do nového.

Migrace dat je popsána v krocích 7-9. První dva kroky, tedy sběr informací a ověřování proveditelnosti, musí být provedeny manuálně a mají vliv na následnou proveditelnost celé migrace a konzistenci databáze po jejím vykonání. Absencí těchto dvou kroků se mohou vyskytnout problémy jako jsou např. kolize jmen, chybějící klíče či ztráta některých informací. Třetí krok pak představuje samotnou migraci.

Tyto stěžejní kroky musí být vykonány manuálně, byť existuje bezpočet nástrojů, které zvládají ORM jako například populární Hibernate [8]. Soustředí se ovšem pouze na vývoj aplikace, ale její evoluci již nezvládají. Jsou sice schopny vygenerovat nové databázové schéma, ale nedokážou zajistit migraci samotných dat. Na vině je zejména skutečnost, že si tyto nástroje buď vůbec, nebo pouze omezeně udržují vztah mezi jednotlivými verzemi modelů. Více si o těchto nástrojích povíme v kapitole 8.

Je zřejmé, že největší díl práce je koncentrován do třech posledních kroků, které musí být vykonány ručně. Za pozornost také stojí fakt, že evoluční změny definujeme zbytečně hned dvakrát. Poprvé v kroku pět kvůli entitám, podruhé pak v kroku sedm kvůli instancím.

Kapitola 3

Navrhované řešení

Až doposud bylo nutné dílčí kroky migrace definovat a posléze vykonávat zcela ručně, což bylo nejen velmi časově náročné, ale také to mělo za následek jak ohrožení konzistence modelů, tak samotných instancí. Z těchto důvodů se snažíme tyto neduhy eliminovat. Naším cílem není pouze snaha automatizovat celý proces, ale také zajistit, aby evoluci entit a stačilo popsat pouze jednou, ne dvakrát, a předejít tak inkonzistenci modelů či ztrátě cenných dat po ukončení evolučního procesu.

Algoritmus 2 Životní cyklus sjednocující tvorbu a evoluci běžné aplikace.

1: **if** Vyžádána evoluce **then**
2: Sběr potřebných informací pro migraci dat a příprava sady operací
3: Ověření proveditelnosti sady operací
4: Vykonání sady operací
5: **end if**

Námi navrhované řešení si poradí se všemi těmito problémy. Jeho hlavní devízou je sada unikátních operací, pomocí kterých se dá popsat jakákoliv myslitelná změna objektové vrstvy (např. přidání nové třídy či odstranění nevhodného atributu). Tato změna je poté propagována z objektové vrstvy přes databázové schéma až do samotné databáze. Z tohoto důvodu musí operace již na počátku disponovat všemi potřebnými informacemi pro každou z vrstev.

Toto řešení nám umožňuje zachytit drtivou většinu případných problémů již v jejich zárodku (viz krok 3 životního cyklu 2), ještě předtím než vůbec začne samotná evoluce. Minoritní díl problémů je pak většinou zaznamenán až v samotné databázi. Poněvadž se sada dotazů směřujících do databáze chová po vzoru transakcí, není ani tento stav nezvratný. Další výhodou je, že vše nutné stačí definovat na jednom jediném místě a to v kroku 2. Ve finále je celý proces automatizovaný, čímž předcházíme nežádané inkonzistenci jednotlivých vrstev či ztrátě dat.

3.1 Pracovní tok

Proces propagace změny objektové vrstvy do databáze je realizován pracovním tokem, který v daném pořadí spouští příslušné nástroje nad vstupními modely, dohlíží na jejich

úspěšnost a přivádí tak evoluci do zdárného konce. Vyskytne-li se někde nějaký problém, celý proces se přeruší a modely včetně dat v databázi se vrátí do původního stavu.

Nejprve je spuštěna transformace na objektové vrstvě, která se pokusí vykonat danou sadu operací nad vstupním modelem. Je-li úspěšně vykonána, je tato sada namapována na odpovídající sadu pro databázové schéma, kde se spustí obdobná transformace jako na objektové vrstvě. Nedojde-li k neúspěchu ani na této vrstvě, jsou všechny operace převedeny do podoby dotazů a poslány do databáze k provedení. Na vstup každé z dílčích transformací (dvě na každé vrstvě, další mezi objektovou a databázovou vrstvou a konečně poslední pro generaci dotazů do databáze) je nutné zavést příslušný model, se kterým má pracovat. Jejím výstupem je pak tentýž model včetně změn, nebo sada dotazů v případě generátoru.

Podrobněji si pracovní tok popíšeme ještě v úvodu kapitoly 6.

3.2 Motivační příklad

Představme si situaci, ve které se na objektové vrstvě nachází dvojice tříd se několika společnými atributy reprezentujícími kontaktní údaje. V průběhu vývoje se ukáže, že by bylo záhodno zmíněné společné atributy vyextrahovat do společného předka. Samo o sobě se nejedná o nic netriviálního, ovšem po vytvoření nového databázového schématu, musíme příslušné sloupce z původních tříd sjednotit ve společném předkovi. To už pouze pomocí objektově-relačního mapování nezvládneme. Našemu frameworku ovšem stačí vykonat jediný řádek kódu s definicí operace s příslušnými parametry.

```
model.operations += extractParent({"NaturalPerson", "LegalPerson"},
                                  {"street", "city", "country", "zip"},
                                  "Party");
```

Po spuštění transformace se framework už sám postará o ověření proveditelnosti operace a o případnou evoluci vrstev a migraci dat.

Kapitola 4

Aplikační meta-model

Abychom se mohli vypořádat se změnou objektové vrstvy, je nutné tuto vrstvu nejdříve nějakým způsobem reprezentovat. Díky Model Driven Architecture (MDA) [9], které využíváme, nám pak stačí jeden meta-model pro objektovou vrstvu (aplikační) a druhý pro databázové schéma (databázový). K tomu nám slouží Eclipse Modeling Framework (EMF) [4], který nám poskytuje patřičné zázemí pro tvorbu meta-modelů.

Konkrétně si pak vystačíme s jedním s dvojice značkovacích jazyků. Buď Ecore [5], který je schopen vizuální reprezentace, která nám umožňuje si celý meta-model doslova naklikat, nebo jeho člověku srozumitelnější obdoba Emfatic [6]. Tyto jazyky jsou navzájem konvertibilní, takže není problém, kdykoliv přejít z jednoho na druhý.

Ukázková implementace entity zastupující standardní třídu (v modelu se pak nachází ještě třídy primitivní a vložené) v jazyku Ecore.

```
<eClassifiers
    xsi:type="ecore:EClass"
    name="StandardClass"
    eSuperTypes="#//APP/GeneralClass">
    <eStructuralFeatures xsi:type="ecore:EReference"
        name="parent"
        eType="#//APP/StandardClass" />
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="isAbstract"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"
        defaultValueLiteral="false" />
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="inheritanceType"
        lowerBound = 1
        eType="#//app/InheritanceType"
        defaultValueLiteral=0 />
</eClassifiers>
```

Odpovídající úsek kódu tentokrát v jazyku Emfatic.

```
class StandardClass extends GeneralClass {
    ref StandardClass parent;
    attr boolean isAbstract = false;
    attr InheritanceType[1] inheritanceType = 0;
}
```

Ze zápisu by mělo být patrné, co je jeho účelem. Řádky tohoto kódu definují entitu *StandardClass*, o které si ještě více povíme v následující sekci.

4.1 Popis dílčích komponent aplikačního meta-modelu

Aplikační meta-model slouží jako formální předpis pro tvorbu konkrétních aplikačních modelů. Definuje dílčí komponenty, ze kterých se tyto konkrétní modely mohou skládat, a také zahrnuje sadu operací, které posléze mohou být nad těmito modely vykonávány. Jak již bylo zmíněno, aplikační meta-model jsme přebírali v docela jiné podobě, než v které se nachází dnes. V tomto směru došlo k jeho rozšíření, zároveň ovšem i k zefektivnění. Další změny nejsou vyloučeny.

4.1.1 Balíček mm

Hlavní balíček *mm* (meta-model) obsahuje balíček *app* (application meta-model), v němž jsou uloženy jednotlivé třídy aplikačního meta-modelu, a dále pak balíček *ops* (operations), který pro změnu obsahuje definice jednotlivých operací, které mohou být vykonány nad konkrétním aplikačním modelem.

4.1.2 Balíček app

Základním kamenem každého modelu je kořen (*ModelRoot*), který uchovává dvojici veledůležitých kolekcí. První z nich (*generations : ModelGeneration*) zahrnuje generace, kterými model prošel a kde první je ta nejmladší a poslední ta nejstarší. Druhá kolekce (*operations : ModelOperation*) pak představuje sadu operací, které mají být nad modelem vykonány.

ModelGeneration Zastupuje konkrétní generaci, nebo-li verzi, modelu, na nějž si také uchovává zpětnou vazbu (*modelRoot : ModelRoot*). Dále si drží množinu všech entit (*entities : ModelEntity*), ze kterých se příslušná verze modelu skládá.

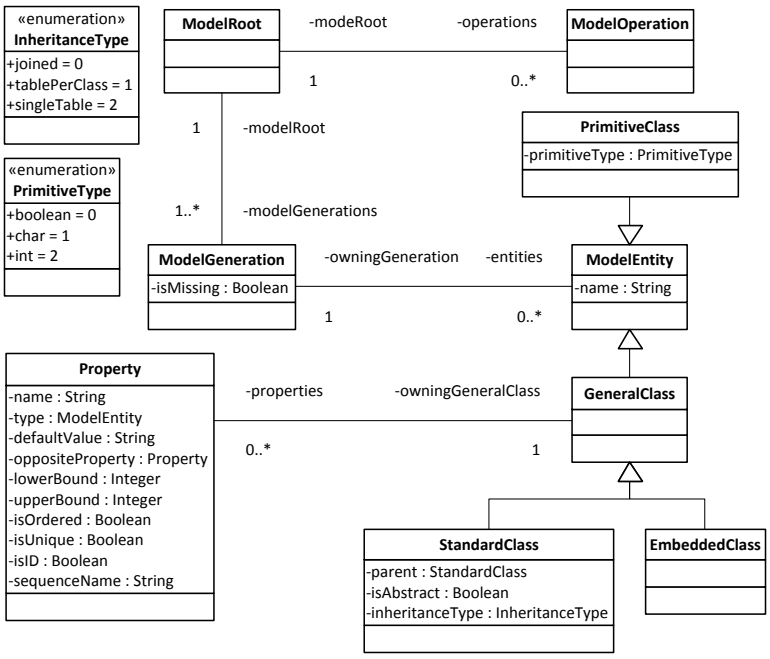
ModelEntity (abstraktní) Nejobecnější člen třídní hierarchie, který si uchovává jednak zpětnou vazbu na generaci (*owningGeneration : ModelGeneration*), do které náleží, za druhé pak svůj název (*name : EString*).

PrimitiveClass (dědí z ModelEntity) Reprezentuje primitivní datové typy (*primitiveType : PrimitiveType*), které se mohou objevit nad databází.

GeneralClass (dědí z ModelEntity) (abstraktní) Obsahuje kolekci atributů (*properties : Property*), které její potomky odlišují od primitivních tříd.

EmbeddedClass (dědí z GeneralClass) Obsahuje pouze atributy primitivních datových typů a nedisponuje vymoženostmi standardních tříd, jako jsou např. dědičnost či abstrakce. Slouží především k zpřehlednění objektové vrstvy a drží si logicky příbuzné atributy, se kterými je zacházeno jako s jedním. Na databázové úrovni se pak celá třída mapuje do tabulky, která odpovídá třídě s příslušnou referencí a tato reference zaniká. Nevzniká tak nová tabulka, jak by tomu bylo v případě standardní třídy.

StandardClass (dědí z GeneralClass) Může obsahovat jak atributy primitivních datových typů, tak reference na jiné standardní třídy. Také může být součástí hierarchie, jejíž typ lze určit parametrem *inheritanceType : InheritanceType*, standardních tříd (*parent : StandardClass*) či disponovat vlastností abstrakce (*isAbstract : EBoolean*).



Obrázek 4.1: UML diagram aplikačního meta-modelu.

Property Představuje atribut obecné třídy. Uchovává si jak zpětnou vazbu na třídu (*owningGeneralClass : GeneralClass*), které náleží, tak svůj název (*name : EString*) a datový typ (*type : ModelEntity*). Atribut nemůže nabývat daného ať už primitivního či komplexního datového typu, neexistuje-li v modelu příslušná třída tento typ reprezentující. Buď tedy primitivní třída, jejíž parametr *primitiveType* odpovídá požadovanému datovému typu, nebo

obecná třída, jejíž název se shoduje s požadovaným datovým typem. Dále je možné určit, půjde-li o povinný atribut (*lowerBound : EInt*) nebo kolekci (*upperBound : EInt*) a to ať už setříděnou (*isOrdered : EBoolean*) či unikátní (*isUnique : EBoolean*). Také lze nastavit vazbu 1..1 (*oppositeProperty : Property*) či příznak identifikátoru (*isId : EBoolean*). Nakonec je možné přiřadit atributu vlastní sekvenci (*sequenceName : EString*).

PrimitiveType (výčet) Jedná se o výčet všech primitivních typů (boolean, char, int) dostupných primitivním třídám (*PrimitiveClass*).

InheritanceType (výčet) Obsahuje trojici možných hierarchických uspořádání, které se projeví při mapování do databáze. Jmenovitě jde o strategie *joined*, *tablePerClass* a *singleTable*. Současná implementace pokrývá první z nich, která se vyznačuje vlastní tabulkou pro každou třídu. Samotná tabulka pak obsahuje pouze atributy dané třídy.

4.1.3 Balíček ops

Každá z operací se vyznačuje tím, že dědí ze společného předka *ModelOperation*, který si pouze uchovává zpětnou vazbu na model (*modelRoot : ModelRoot*), kterému náleží. Základní operace se dělí do tří základních rodin dle svých prefixů: *Add*, *Set* nebo *Rename* a *Remove*. Následuje stručný popis zmiňovaných kategorií. Kompletní výčet operací je pak k nalezení v příloze A.

Operace s prefixem Add Jedná se o operace, které slouží k tvorbě objektů, ať už to jsou nejrůznější třídy či atributy. Pomocí těchto operací je možné postavit jakoukoliv myslitelnou objektovou strukturu. **Patří sem:** *AddPrimitiveClass*, *AddEmbeddedClass*, *AddStandradClass*, *AddProperty* a další.

Operace s prefixem Set nebo Rename Operace, které mají na svědomí změny již vytvořených objektů a jejich vlastností. Škála zahrnuje změny od datových typů, tedy vazeb mezi třídami, přes úpravy hierarchií až po triviality jako přejmenování dané entity. Na jejich proveditelnost jsou obvykle kladeny vysoké nároky. **Patří sem:** *SetParent*, *SetAbstract*, *SetType*, ale také *RenameEntity*, *RenameProperty* a další.

Operace s prefixem Remove Sada operací významově opačných k operacím s prefixem *Add*. Mají na starosti odstraňování tříd, vzájemných vazeb a atributů. Na jejich proveditelnost jsou obvykle kladeny vysoké nároky. **Patří sem:** *RemoveEntity* a další.

Komplexní operace Sofistikovanější operace, které slouží k vykonávání netriviálních změn objektové vrstvy, jako jsou např. slučování tříd, přesuny atributů či extrakce společných předků. Tyto operace mohou v podstatě plnit jakékoliv zadání, které si uživatel usmyslí. Jejich aplikací jsme se zabývali v motivačním příkladě. **Patří sem:** *ExtractParent*, *ExtractClass*, *MoveProperty* a další.

Meta-model je k nalezení na přiloženém médiu v *bp\mm-app.ecore*.

Kapitola 5

Operace a jejich přínos

V předchozí kapitole jsme si stručně popsali jednotlivé kategorie operací rozlišitelných dle jejich prefixů. Ať už se tedy jedná o operace k tvorbě, odstraňování či pouhopouhé modifikaci entit, bez výjimky každá změna objektové vrstvy, ať už strukturální či sémantická, je realizovatelná pomocí jedné či více těchto atomických operací, které lze posléze skládat do obsáhlejších celků a docílit tak požadované modifikace o to efektivněji. Ovšem je nutné dbát na sémantiku dílčích operací a ověřit jejich případnou proveditelnost napříč všemi vrstvami. V následující sekci si proto více přiblížíme nástrahy spjaté s verifikací operací a jak se jim vyvarovat.

5.1 Testování proveditelnosti operací

Existují celkem tři základní sady ověřovacích pravidel. V první řadě je nutné, aby operace vůbec dávaly smysl samy o sobě a též vůči danému modelu. Dále je potřeba zajistit konzistenci modelu po vykonání operace. Nakonec není žádoucí jakákoliv neočekávaná ztráta dat. Proveditelnost operací se musí ověřit ještě před zahájením samotného evolučního procesu. Testování probíhá na každé dílčí vrstvě, a proto je nutné celou evoluci simulovat.

Ověřovací pravidla pro každou operaci jsou pevně dána a tedy i neměnná. Za všech okolností tak zajistíme konzistenci modelů po evoluci.

Za prvé je tedy uživatel povinen zajistit správnou sémantiku operací na objektové vrstvě, předpokládá-li se, že je tato operace navržena vyhovujícím způsobem, tj. poskytuje veškeré potřebné informace pro každou ze tří vrstev. Při mapování operace z objektové vrstvy na operaci či sadu operací pro databázové schéma je její sémantika zajištěna tímto mapováním. Obdobně platí, že sémantika dotazů směřujících do databáze je zajištěna jejich generováním. Sémantika operace musí též odpovídat příslušnému modelu v momentě volání, tj. nemůže například manipulovat s entitami či tabulkami, které v modelu neexistují. V našem motivačním příkladě musí tedy třídy *NaturalPerson* a *LegalPerson* nejen existovat, ale také obě dvě obsahovat čtveřici atributů stejného datového typu *street*, *city*, *country* a *zip*.

Za druhé je nutné, aby se model nacházel v konzistentním stavu po vykonání příslušné operace, tj. není např. možné narušit unikátnost názvů entit či tabulek. Co se motivačního příkladu týče, není například možné, aby na objektové vrstvě již existovala třída *Party*.

Za třetí a nakonec je potřeba ověřit proveditelnost operací z instančního hlediska, tj. zkontrolovat, nakládá-li operace s příslušnými daty, tak jak bylo požadováno. Operace nesmí žádným způsobem manipulovat s daty, které nespadají do její kompetencí. V některých případech toto chování definuje uživatel při volání operace, kdy se může rozhodnout, jak bude s instancemi nakládáno ve sporných případech, tj. když je např. snaha spojit dvě tabulky s rozdílným počtem záznamů do jedné. Tuto situaci si nejlépe můžeme představit na příkladu sloučení dvojice tříd, kde každá obsahuje jiný počet instancí. V takovém případě se musíme ještě před zahájením evolučního procesu rozhodnout, jak s instancemi naložíme, zda-li doplníme prázdné buňky implicitními hodnotami, odstraníme přebytečné záznamy, nebo vůbec evoluci nedovolíme.

Nesporným přínosem tohoto řešení je automatizace ověřovacích pravidel napříč vrstvami. Neboť znalost, zda-li je operace vůbec proveditelná či neztrácí data, je nezbytně nutná pro její vykonání.

5.2 Komplexní operace

Základní sada atomických operací slouží k vykonávání triviálních změn objektové struktury aplikace, jako jsou například přidání nové třídy, změna názvu či datového typu atributu, či odstranění asociační vazby mezi třídami. Můžeme směle prohlásit, že tyto operace odpovídají DDL[1] dotazům na úrovni databázového schématu.

Řádného využití a zefektivnění celého evolučního procesu ale tyto operace dosáhnou až při jejich sloučení do komplexnějších celků, pro jejichž úspěch je nutné využít jak DDL, tak DML [2] dotazů (např. operace *ExtractParent* z motivačního příkladu, která nejen vytvoří novou třídu s příslušnými atributy, ale také se musí postarat o přenos kontaktních údajů z jedné tabulky do druhé). Jediným omezením, se kterým se tu můžeme setkat, je nutnost dodržet pořadí operací a výstup každé operace musí být v souladu se vstupem operace následující v sekvenci. Toto chování by se dalo připodobnit k databázovým transakcím.

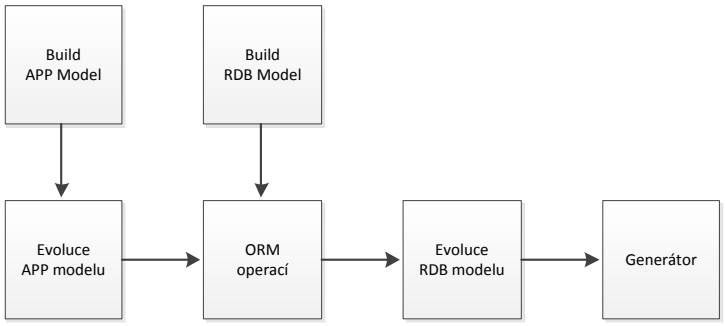
Výhodou komplexních operací je, že uživateli stačí vykonat méně práce při definování požadované změny aplikace. Hlavním smyslem je ale pak především skutečnost, že jedině komplexní operace dokážou udržet kontext celé evoluční změny. Selže-li tedy byť jediná operace v sekvenci, je po vzoru transakcí zahozena celá transformace. Toto chování je nesmírně důležité, neboť komplexní operace popisují tak netriviální změny aplikační struktury, že vykonání jen několika operací v sekvenci může uvést model do inkonzistentního stavu.

[1] **DDL** Data Definition Language
[2] **DML** Data Manipulation Language

Kapitola 6

Evoluce aplikace

Každá dílčí změna ať už objektové vrstvy nebo databázového schématu je reprezentována odpovídající operací. Úspěšným vykonáním této operace příslušný model prochází změnou a říkáme, že dochází k přechodu z jedné generace do druhé. S každou vykonanou operací tedy vzniká i jakási pomyslná generace modelu. V praxi se pak tento proces spouští aplikováním konkrétního transformačního souboru nad daným modelem.



Obrázek 6.1: Realizace pracovního toku.

V našem případě existují dva takové transformační soubory napsané v jazyku QVT [13]. První má za úkol realizovat evoluci objektové vrstvy, druhý pak databázového schématu. Jejich vstupem je příslušný model vrstvy popsaný značkovacím jazykem XMI [14] a výstupem tentýž model po evoluci. Transformace v tomto modelu najde množinu operací, které mají být na model aplikovány a začne ji vykonávat. Nejdříve ověří, zda-li je operace vůbec proveditelná a posléze je vykonána. Takto pokračuje, dokud není množina prázdná. Následně je celá množina s využitím objektově-relačního mapování operací – též napsaného v jazyku QVT – namapována na množinu operací pro databázové schéma, kde se spustí obdobná transformace a opět se pokusí vykonat beze zbytku všechny operace. Tyto jsou posléze převedeny na SQL dotazy prostřednictvím generátoru vyhotoveného pomocí Xtext [15] technologií, které se

pošlou do databáze. Proběhne-li všechno v pořádku, můžeme prohlásit, že aplikace postoupila do nové generace, v opačném případě se celá sada změn v databázi odroluje a simulované modely se zahodí. Tento proces se nazývá pracovní tok a je realizován v jazyku MWE2 [10]. Pracovní tok vystihuje obrázek 6.1.

6.1 Evoluční transformace objektové vrstvy

Následující popis bude zaměřen na objektovou vrstvu, neboť především ta je náplní mé práce. Ovšem nutno podotknout, že transformace pro databázové schéma je obdobná.

Ještě před spuštěním samotné evoluční transformace je nutné v modelu, který do transformace bude vstupovat, definovat operaci či sadu operací, která má být nad modelem vykonána. Dosáhneme toho tak, že do diskutovaného modelu přidáme pro každou požadovanou operaci nový objekt typu *ModelOperation* a vyplníme jeho příslušné parametry. V *build* souboru napsaného v jazyku QVT by taková definice mohla vypadat následovně.

```
model.operations += addStandardClass("Party");
```

V budoucnu by mělo být možné do konzole zapsat pouze příkaz jednoduchý příkaz, jak bylo naznačeno v motivačním příkladu v úvodu této práce v kapitole 3.

Stěžejní částí transformačního souboru je procedura *main()*, která kromě zpracování kořene a příslušné generace vstupního modelu má též na starosti samotnou aplikaci operací na diskutovaný model. V prvním kroku algoritmu 3 vidíme započetí cyklu, který čítá jednu operaci za druhou. Ve svém těle (kroky 2-9) pak nejprve oznámí, že přistupuje k operací a následně vstoupí do podmínky (krok 3), ve které se ověří proveditelnost operace. Je-li operace proveditelná, vykonají se kroky 4 a 5. První z nich je opět výpis a druhý se postará o aplikaci samotné operace na model v dané generaci. Takhle se pokračuje, až už nezbývá žádná operace k vykonání. Není-li operace proveditelná, opět se to oznámí výpisem na terminál a celý cyklus se zastaví, tj. další operace v sekvenci se již nevykonávají. Simulace je tak zahozena a uživatel upozorněn na problém.

Algoritmus 3 Hlavní logika evoluční transformace.

```
1:  for all Operace ze vstupní množiny do
2:      Výpis operace na terminál
3:      if Operace proveditelná then
4:          Výpis o proveditelnosti operace
5:          Aplikace operace na model objektové vrstvy
6:      else
7:          Výpis o neproveditelnosti operace
8:          Ukončení cyklu
9:      end if
10: end for
```

Implementace každé operace se tedy skládá ze dvou částí: ověření proveditelnosti a samotného vykonání. Ukažme si konkrétní implementaci z motivačního příkladu, kde potřebujeme vytvořit novou třídu.

```
query AddStandardClass::isValid(gen:ModelGeneration):Boolean {
     return not gen.isEntityInGeneration(self.name)
            and gen.isPrimitiveClassInGeneration("Integer");
}

mapping AddStandardClass::apply(inout gen:ModelGeneration) {
    gen.entities += gen.AddStandardClass(self);
}
```

První dotaz (query) představuje dvojici ověřovacích pravidel. Nejdříve se ptáme, jestli už náhodou třída se stejným názvem v modelu neexistuje, následně si potřebujeme ověřit, že existuje primitivní třída *Integer*, kterou budeme potřebovat pro vytvoření povinného identifikátoru celočíselného typu. Jsou-li obě dvě podmínky splněny, vrátí dotaz hodnotu *true*.

V druhém případě se již nejedná o dotaz, ale o mapování (mapping), jehož parametrem je generace s prefixem *inout*, který zajišťuje, že daná generace je jednak vstupem, ale v případně pozměněném stavu také výstupem. Posléze je do její množiny entit přidána nová standardní třída s příslušným názvem a identifikátorem.

Popsali jsme si způsob, jak definovat operace ve vstupním modelu a následně, jak jsou tyto operace nad daným modelem vykonány prostřednictvím evoluční transformace. Po úspěšném vykonání operace není objekt, kterým byla ve vstupním modelu reprezentována, odstraněn, ale zůstává, aby se dalo na transformaci navázat a požadované změny bylo možné prostřednictvím objektově-relačního mapování propagovat do nižších vrstev.

Transformace je k nalezení na přiloženém médiu v *bp\populate_generations_app.qvto*.

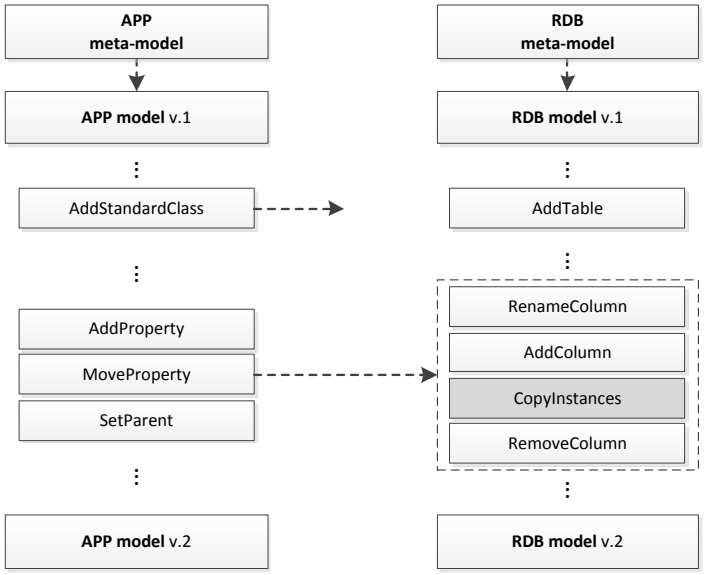
6.2 Objektově-relační mapování operací

Odečítat změny z nové generace modelu po úspěšné evoluční transformaci by bylo obtížně implementovatelné. Proto volíme cestu schůdnější a to sice objektově-relační mapování jednotlivých operací (ORMO), které se v návaznosti na úspěšnou evoluci objektové vrstvy postará o namapování jednotlivých operací do modelu reprezentujícího databázové schéma. Celá myšlenka by měla být patrná z obrázku 6.2.

Každá operace na objektové vrstvě má odpovídající obraz na databázové úrovni. Ovšem ne vždy jde o jednu jedinou operaci, obvykle se jedná o celou sadu operací, která musí být vykonána po vzoru transakcí. Dojde-li tedy na této vrstvě k neúspěchu, musíme zrušit i změnu na objektové úrovni, z tohoto důvodu musíme celou evoluci nejdříve simulovat.

Po úspěšném vykonání evoluční transformace objektové vrstvy zůstanou ve výstupním modelu objekty zastřešující operace, které byly na tuto vrstvu aplikovány. ORMO transformace pak v metodě *main* obsahuje cyklus nepodobný tomu z evoluční transformace (viz algoritmus 4), který čítá jednotlivé operace a uvnitř svého těla je mapuje (krok 3) do výstupního modelu databázového schématu. V této fázi samozřejmě nemusíme ověřovat proveditelnost operací, neboť se je nechystáme vykonávat, ale pouze je převádíme na jinou množinu.

Na základě typu operace, která je právě na řadě, zvolí transformace odpovídající implementaci. Samotné namapováni si ukážeme na příkladu s vytvořením standardní třídy.



Obrázek 6.2: Obejktově-relační mapování operací mezi vrstvami.

```
mapping AddStandardClass::toRdb(appGen:ModelGeneration, rdbRoot:ModelRoot) {
    rdbRoot.operations += addTable(self.name);
    rdbRoot.operations += addIdColumn(self.name);
    rdbRoot.operations += addIndex(self.name);
    rdbRoot.operations += addPrimaryKey(self.name);
}
```

Algoritmus 4 Hlavní logika ORMO transformace.

1: **for all** Operace ze vstupní množiny **do**
2: Výpis operace na terminál
3: Namapování operace na sadu operací pro databázové schéma
4: **end for**

Jak je z příkladu patrné, z jediné operace dostáváme rovnou čtveřici. První dvě pro vytvoření samotné tabulky a identifikačního sloupce, nad kterým další dvě operace v pořadí vytvoří index a příslušné integritní omezení primárního klíče. Tím získáme platnou tabulku, které je plně v souladu s naším modelem.

Podobně namapujeme všechny operace ze vstupní množiny, čímž získáme sadu operací pro databázové schéma. Tyto operace jsou posléze reprezentovány jako nové objekty typu *ModelOperation* ve výstupním modelu.

Nyní by mělo být zcela zřejmé, jak propagovat změny objektové vrstvy na databázové schéma, udržet jejich kontext a nedovolit modelům, aby upadly do inkonzistentního stavu.

ORM nástroj je k nalezení na přiloženém médiu v *bp\orm_operations.qvto*.

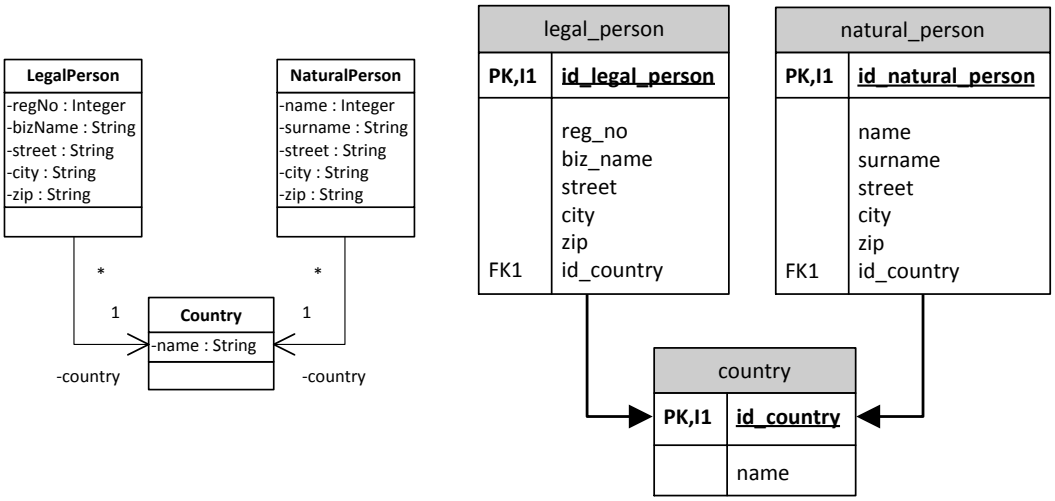
Kapitola 7

Motivační příklad

V této kapitole si podrobně popíšeme motivační příklad z úvodu této práce a přesvědčíme se, jak jednoduše a zároveň efektivně se dá náš nástroj použít.

7.1 Počáteční stav

Představme si situaci, ve které řešíme následující problém. Dejme tomu, že udržujeme netriviální aplikaci, která disponuje celkem bohatou objektovou vrstvou, tomu odpovídající databází napěchovanou nejrůznějšími daty a kdesi v útrobách se skví dvojice odlišných entit reprezentující právní subjekty, tedy fyzické a právnické osoby.



Obrázek 7.1: Obejktová vrstva a databázové schéma v počátečním stavu.

Jak je patrné z obrázku 7.1, každá ze dvou tříd obsahuje něco málo informací, které entity vzájemně odlišují, ale také množinou atributů, ve kterých se entity shodují a které by

id	name	surname	street	city	country	zip
1	Luca	Woods	4886 Deer Haven Drive	Greenville	SC	29615
2	Jack	Stark	4372 Michael Street	Houston	TX	77002
3	Ann	McCormick	2364 Libby Street	Culver City	CA	90232

Tabulka 7.1: Tabulka natural_person reprezentující fyzické osoby.

id	reg_no	bis_name	street	city	country	zip
4	390-274	External Paradigm	1790 Wayside Lane	Oakland	CA	94606
5	780-625	Slow Cyber Fast	3387 Haymond Road	Glide	OR	97443
6	973-176	Commervial Design	4076 Nash Street	Southfield	MI	48076

Tabulka 7.2: Tabulka legal_person reprezentující právnické osoby.

se daly obecně označit jako kontaktní údaje (street, city, country a zip). Tento nešikovaný návrh má za následek nejen duplikaci kódu jako takového, případnou redundanci dat, ale také roztroušení souvisejících informací do několika různých tabulek.

Jako efektivní řešení by se mohla ukázat extrakce společného předka, který by obsahoval právě ony shodné atributy. Ovšem takto netriviální změnu objektové vrstvy doprovází generování nového databázového schématu a posléze ruční migrace dat.

K využití frameworku je nejdříve nutné vytvoří příslušný objekt typu *ModelOperation* ve vstupním modelu reprezentujícím objektovou vrstvu. Do *build* souboru v jazyku QVT bude tedy třeba připsat následující řádek, čímž je příslušná operace definována a připravena k použití.

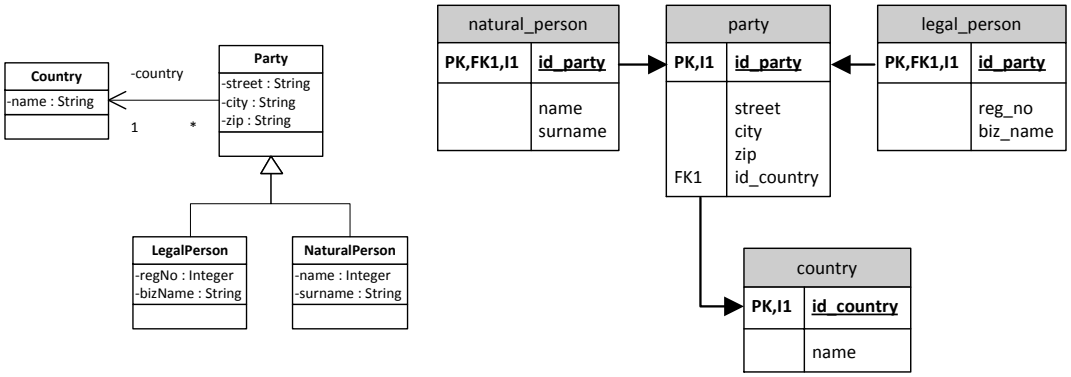
```
model.operations += extractParent({"NaturalPerson", "LegalPerson"},
                                  {"street", "city", "country", "zip"},
                                  "Party");
```

Objektová vrstva a databázové schéma v počátečním stavu je k nahlédnutí na obrázku 7.1. V tabulkách 7.1 a 7.2 je k vidění několik záznamů, na nichž si názorně ukážeme, jak je nakládáno s instancemi. Zaměřme se například na Jacka Starka z Hustonu v tabulce 7.1.

7.2 Extrakce společného předka

Operace *ExtractParent* je operací komplexní a skládá se ze čtveřice operací atomických (O1, O2, O3 a O4). Dvojice operací O3 a O4 je obzvlášť kritická a musí být vykonána v rámci společného kontextu, jinak by se model mohl dostat do inkonzistentního stavu, jak si ukážeme dále.

Neboť na objektové úrovni nemusíme dbát na jednotlivé instance, jsou následující změny velice triviální. Nejprve vytvoříme novou třídu (O1) s odpovídajícím názvem a přidáme do ní množinu požadovaných atributů (O2), čímž získáme kýženého předka. Následně potřebujeme odstranit společné atributy (O4) z dvojice původních tříd, ovšem nesmíme zapomenout, že na



Obrázek 7.2: Obejktová vrstva a databázové schéma po extrakci společného předka.

nižších vrstvách budeme potřebovat zkopírovat data v těchto atributech uložených do nové tabulky. Proto nejdříve vykonáme operaci, která prováže třídy dědičnou vazbou (O3) s právě vytvořenou třídou a následně se postará o smazání daných atributů (O4).

Poté co je operace úspěšně vykonána na objektové vrstvě, namapuje se na příslušnou sadu operací pro databázové schéma. Nejprve vytvoříme tabulku s odpovídajícím názvem a přiřadíme jí primární klíč (odpovídá O1). Následně do ní přidáme množinu požadovaných sloupců (odpovídá O2). Neboť v databázovém schématu je dědičnost realizována pomocí identifikátorů a práce s instancemi není stále dostupná, nemá O3 na schématu příslušný obraz. Tato informace se pouze přepošle dál v podobě prázdné operace, aby bylo zřejmé, v jaký moment je nutné zkopírovat záznamy, než budou nenávratně ztraceny. Nakonec se odstraní sloupce z dvojice původních tabulek (odpovídá O4).

Poslední a nejdůležitější vrstvu reprezentuje samotná databáze. Dotazy poslané do databáze odpovídají operacím na databázovém schématu s tím rozdílem, že je vykonána kritická operace odpovídající O3, která zkopíruje jednotlivé instance do rodičovské tabulky.

Povšimněme si, jak se změnily dílčí vrstvy 7.2 a posléze tabulky 7.4 a 7.5. Všechny kontaktní údaje se totiž přesunuly do tabulky 7.3, kde také pod identifikátorem č. 2 najdeme odpovídající adresu Jacka Starka.

Co když bychom ale rádi uchovávali více jak jednu adresu pro každý právní subjekt? Nabízí se přesunutí kontaktních údajů do nové třídy. A v tabulce 7.3 pak můžeme vytvořit tolik referencí, kolik bude potřeba. Opět nám stačí jeden jediný řádek kódu.

```
model.operations += extractClass("Party",
                                 {"street", "city", "country", "zip"},
                                 "Address");
```

id	street	city	country	zip
1	4886 Deer Haven Drive	Greenville	SC	29615
2	4372 Michael Street	Houston	TX	77002
3	2364 Libby Street	Culver City	CA	90232
4	1790 Wayside Lane	Oakland	CA	94606
5	3387 Haymond Road	Glide	OR	97443
6	4076 Nash Street	Southfield	MI	48076

Tabulka 7.3: Tabulka party reprezentující právní subjekty.

id	name	surname
1	Luca	Woods
2	Jack	Stark
3	Ann	McCormick

Tabulka 7.4: Tabulka natural_person po extrakci společného předka.

7.3 Extrakce kontaktních údajů do nové třídy

Operace *ExtractClass* je taktéž operací komplexní a skládá se ze čtveřice operací atomických (O1, O2, O3 a O4).

Opět nemusíme dbát na jednotlivé instance, proto nám stačí vytvořit novou třídu (O1) s odpovídajícím názvem a příslušnými atributy (O2). Nakonec v původní třídě vytvoříme sloupec (O3), který bude sloužit jako reference na novou třídu s kontaktními údaji, a odstraníme již nepotřebné atributy (O4). Odstranění atributů musí být opět vykonáno jako poslední, referenční sloupec bude totiž potřeba při kopírování instancí mezi tabulkami.

Vytvoření nové tabulky (odpovídá O1) s daným názvem, primárním klíčem a příslušnými sloupci (odpovídá O2) je triviální. Přesun jednotlivých instanci je už ovšem složitější. Nejprve musíme vytvořit sloupec v původní tabulce (odpovídá O3) a následně si připravit (ještě před zkopírováním jednotlivých záznamů) operaci pro vygenerování nových identifikátorů pomocí sekvence[1] (O3a). Toto pořadí volíme proto, abychom zajistili správné spárování příslušných dat. Nyní můžeme též připravit operaci pro zkopírování instancí do nové tabulky (O3b), kde její identifikační sloupec bude odpovídat námi právě vytvořenému (a později identifikátory naplněnému) sloupci referenčnímu. Nakonec odstraníme příslušné atributy (odpovídá O4) z původní tabulky a omezíme referenční sloupec cizím klíčem.

Následně vše ve formě odpovídajících dotazů pošleme do databáze, včetně kritických operací pro generování sekvenčních čísel a kopírování instancí.

Ovšem aby změny odpovídaly schématům na obrázku 7.3, je nutné ještě přejmenovat referenční atribut s implicitním názvem *address* v tabulce 7.6 na námi zvolený název *residentialAddress* a vytvořit další atribut *contactAddress* pro reprezentaci kontaktních adres, což se dá uskutečnit pomocí dvojice triviálních operací, jejichž podrobné vykonáváni si zde již nebudeme popisovat. Definice těchto operací by pak mohla vypadat nějak takto.

[1] Pro zjednodušení uvažujme globální sekvenci.

id	reg_no	bis_name
4	390-274	External Paradigm
5	780-625	Slow Cyber Fast
6	973-176	Commervial Design

Tabulka 7.5: Tabulka legal_person po extrakci společného předka.

id	residential_address	contact_address
1	7	
2	8	
3	9	
4	10	
5	11	
6	12	

Tabulka 7.6: Tabulka party reprezentující právní subjekty po extrakci kontaktních údajů.

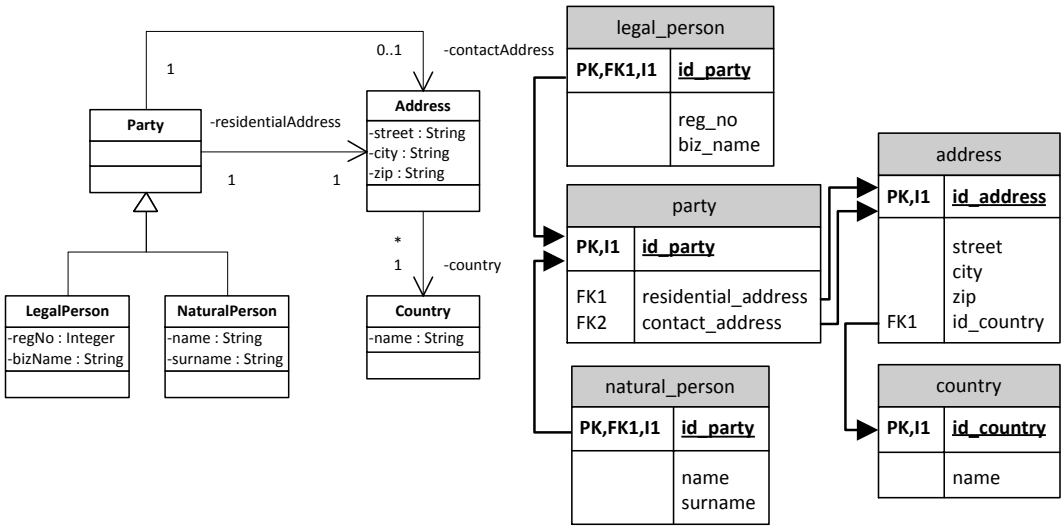
```
model.operations += renameProperty("address",
                                   "residentialAddress");

model.operations += addProperty("Party",
                                "contactAddress",
                                "Address");
```

Nakonec se podívejme, jak vypadá instance Jacka Starka, která je stále k nalezení v tabulce 7.4, která se druhou transformací nezměnila. V tabulce 7.6 vidíme pod odpovídajícím identifikátorem č. 2 kontaktní adresu, která ještě nemohla být vyplněna, a adresu bydliště, která odkazuje na identifikátor č. 8 do tabulky 7.7 na příslušné bydliště v Hustonu.

id	street	city	country	zip
7	4886 Deer Haven Drive	Greenville	SC	29615
8	4372 Michael Street	Houston	TX	77002
9	2364 Libby Street	Culver City	CA	90232
10	1790 Wayside Lane	Oakland	CA	94606
11	3387 Haymond Road	Glide	OR	97443
12	4076 Nash Street	Southfield	MI	48076

Tabulka 7.7: Tabulka address reprezentující adresy.



Obrázek 7.3: Obejktová vrstva a databázové schéma po extrakci kontaktních údajů.

Kapitola 8

Příbuzné projekty

Následující výčet si nebere za cíl být kompletní, nicméně zahrnuje práce, které se našeho projektu bezprostředně týkají a v jistých ohledech nám slouží jako zdroj inspirace. V současné době samozřejmě nemůžeme žádnému z uvedených projektů přímo konkurovat. Následující text je rozdělen do tématicky zaměřených odstavců.

8.1 Objektově-relační mapování

Objektově-relační mapování je ve světě orientovaném na objekty známou záležitostí. Existuje nespočet nástrojů, které se tímto problémem zabývají a dokonce ho i úspěšně řeší. Z mnoha zmiňme třeba populární Hibernate [8]. Ačkoliv si tyto nástroje hravě poradí s ORM, na evoluci samotnou již většinou nestačí. Pochopitelně tu máme i kategorii nástrojů, které si s evolucí více či méně poradí. Patří mezi ně například ADO.NET Entity Framework [7], který této vlastnosti nabyl ve své poslední verzi, nebo velice mocný ActiveRecord [1], který implementuje stejnojmenný návrhový vzor a který je integrován do populárního frameworku pro tvorbu webových aplikací Ruby on Rails. Zvládá migraci dat na základě změny samotných entit. Dalo by se tedy říct, že ActiveRecord řeší právě náš problém. Ovšem není tomu tak zcela úplně. Náš nástroj by měl být absolutně nezávislý na platformě a měl by disponovat větší vyjadřovací schopností než zmiňovaný ActiveRecord, který si neporadí s více komplexnějšími strukturami objektových či databázových modelů.

8.2 Evoluce databázových schémat

Z onoho nepřeberného množství projektů, které se tímto tématem zabývají, vyzdvihněme solidní PRISM [11]. Tento nástroj poskytuje aparát, jak si bez větších potíží poradit s evolucí složitých databázových schémat a posléze nesnadnou migrací samotných dat. Jedná se o další z řady nástrojů, které nabízejí řešení nám velice blízké. Nicméně PRISM je především zaměřen na administrátory databází. V takovém případě se stačí zaměřit pouze na databázovou vrstvu. Ovšem naše řešení cílí na softwarové vývojáře a nesměřuje tedy pouze k vrstvě databázové, ale také k té objektové.

8.3 Evoluce meta-modelů

V této oblasti vyniká nástroj COPE [3] vyvinutý Markusem Herrmannsdörferem pro využití v prostředí EMF, který se specializuje na migraci modelů v závislosti na změnách v meta-modelu. Stěžejní myšlenka je velice podobná té naší. Herrmannsdörfer se jako my snaží definovat každou dílčí změnu samostatnou operací a posléze je vykonávat v dávce. Dále také například uchovává historii změn, díky čemuž může plynule přecházet mezi jednotlivými verzemi. Náš framework touto dovedností sice nevládne, nicméně generační proces, jak již bylo zmiňováno dříve, je navržen a do budoucna se počítá s jeho rozšířením. Byť se tedy COPE nezaměřuje přímo na naši oblast zájmu, jedná se o mimořádně inspirativní počin.

8.4 Vyhodnocení

Vysoká konkurenceschopnost technologií a nástrojů diskutovaných v této kapitole rozhodně není důvodem k úprku. Naopak tento stále se rozvíjející svět nabízí velice inspirativní a motivační zázemí. Věříme, že framework, o kterém pojednává tato práce, se těmto nástrojům v budoucnu vyrovná a v jistých ohledech je i překoná.

Kapitola 9

Shrnutí

Pojďme si na samý závěr shrnout dosavadní práci, kterou jsme na projektu vykonali, a okomentovat její současný stav a vizi, které bychom rádi v budoucnu dosáhli (viz sekce 9.2 a 9.3). Na úvod ovšem ještě v sekci 9.1 stručně zhodnotíme používané technologie a nástroje.

9.1 Používané technologie

S příchodem k projektu jsme se museli popasovat s nepřebernou škálou pro nás neznámých technologií. Vůbec celá myšlenka modelem řízené architektury byla pro nás novinkou. Dalo by se ale říct, že jsme si na ni rychle zvykli a zalíbila se nám.

Výhody a vyjadřovací schopnost jazyku QVT chytře využívajícího sílu deklarativního jazyku OCL[1] nám také rychle přešla do krve. Jiná situace byla okolo dvojice konvertabilních jazyků Ecore a Emfatic. První z nich nám okamžitě imponoval svou vizuální variantou, v té není ale možné používat komentáře. V tomto ohledu jsme tudíž stále nerozhodní. S meta-modely v těchto jazycích realizovanými se také pojí jedna nepříjemná vlastnost a to sice, že aby jich mohl využívat pracovní tok, je nutné je generovat do tzv. pluginů. Tento proces je zdlouhavý a bohužel je vyžadován s každou změnou daného meta-modelu a těch je ve fázi vývoje samozřejmě nemálo. Petr ve své práci ještě využívá technologie Xtext pro generování dotazů a MWE2 pro realizaci pracovního toku. Nevýhodou frameworku, který nám pro práci s pracovním tokem byl dodán zadavatelem projektu, je skutečnost, že si nerozumí s operačním systémem Windows.

Nakonec si ještě připomeňme, že využíváme vývojového prostředí Eclipse, pro nějž jsou všechny nástroje určeny a se kterým jsme spokojeni.

9.2 Aktuální stav

V současné době disponujeme dvojicí meta-modelů popisujících objektovou a databázovou vrstvu. Dále pak dvojicí evolučních transformací pro každou ze dvou vrstev a dvěma mapovacími nástroji, kde první funguje na principu objektově-relačního mapování a realizuje mapování operací z objektové vrstvy na databázovou a druhý slouží jako generátor dotazů do

[1] OCL. <**http://www.omg.org/spec/OCL/2.0/**>

databáze. Dále se nám podařilo realizovat pracovní tok aplikace, který je schopen spouštět jednotlivé transformační nástroje a ověřovat jejich úspěšnost. Efektivnost frameworku je patrná z kapitoly 7.

Co se meta-modelu objektové vrstvy týče, k mé nelibosti ho bylo třeba spíše rozšiřovat. Ukázalo se totiž jako naprosto nezbytné pozměnit celou hierarchii tříd (*PrimitiveClass*, *StandardClass* a *EmbeddedClass*), neboť původní návrh s jediným zástupcem (*Class*) nedokázal efektivně zohlednit vlastnosti jednotlivých typů tříd. Další úpravy se pro změnu dotkly dílčích vlastností jednotlivých entit, kde většinou došlo k jejich zanedbání či přesunu a které nemá smysl na tomto místě zmiňovat. Největší část změn se ale odehrála mezi operacemi, kde bylo nutné nově definovat celou škálu od těch nejjednodušších až po ty složitější.

Následně přišla na řadu evoluční transformace, kde jsme se zprvu potýkali se samotným jazykem QVT. Snažili jsme se spolu s implementací na objektové vrstvě vytvořit i odpovídající implementaci na vrstvě databázové. Tento postup se ale brzy ukázal jako nevyhovující. Neboť nástroj pro ORM operací vyžadoval mírně jiný přístup. Dovolím si připomenout, že ORM operací do databázové vrstvy bylo mým posledním úkolem v rámci této práce. Vycházel jsem z objektové vrstvy, kde jsem se snažil pro každou operaci definovat odpovídající množinu pro databázové schéma. Tímto postupem jsem realizoval mapování celé škály atomických a několika komplexnějších operací.

Sada operací, na nichž stojí hlavní myšlenka celého projektu, je zatím v plenkách. Nicméně základní jádro existuje a jedná se o množinu operací, se kterými je možné stavět a bourat jednodušší modely.

Vzhledem k experimentálnosti projektu a neokoukanosti používaných technologií nejsou tyto výstupy zcela dokonalé. Naopak bude nutné zapracovat na jejich zlepšení. Dokonce se můžeš stát, že některé ze zde zmiňovaných postupů, se ukáží jako nevhodné a bude nutné se vydat jinou cestou.

Také existují jistá omezení platformy, přičemž některá jsou úmyslná (meta-model objektové vrstvy v zásadě vyhovuje jazyku JAVA[2], nedokázal by ale například pokrýt všechny potřeby jazyka C++[3], neboť nedisponuje schopností vícenásobné dědičnosti) a jiná prozatimní (momentálně jsme se přizpůbsobili databázovému systému PostgreSQL[4]).

9.3 Vize

Z důvodu seznamování se s vývojovým prostředím, platformou, architekturou i samotnými jazyky je zde mnoho prostoru pro zlepšení. V nejbližších dnech bude tedy nuté podniknout zejména rozsáhlou refaktorizaci kódu všech zmíněných transformací.

Co se vývoje týče jsme stále na začátku a ještě chvíli tam pobudeme. Následující cíle spočívají v rozšíření sady dostupných operací a možnosti spouštět je konzolovým příkazem nad jakoukoliv databází. Ovlivněn již zmiňovaným projektem COPE bych další možné kroky spatřoval v posunu realizace navrženého systému generací. Uchovávání historie a dílčích verzí modelů by nám usnadnilo i samotný vývoj.

[2] JAVA. <**http://www.oracle.com/us/technologies/java/overview/index.html**>
[3] C++. <**http://www.cplusplus.com/**>
[4] PostgreSQL. <**http://www.postgresql.org/**>

Kapitola 10

Závěr

Mým úkolem na tomto projektu byl návrh a následná implementace meta-modelu objektové vrstvy a příslušné evoluční transformace. K tomu se přímo váže definice základní sady operací pro tuto vrstvu určených a konečně realizace objektově-relačního mapování těchto operací pro databázové schéma. Odhlédneme-li od nutnosti refaktorizace a dalších úprav kódu a vezmeme-li v potaz demonstraci efektivnosti řešení v sekci s motivačním příkladem a následnou pozitivní odezvu na mezinárodní konferenci Code Generation z letošního března, můžeme směle prohlásit, že zadání této práce se mi podařilo zdárně naplnit. Vzhledem k tomu, že je jednáno o pilotním nasazení, hodlám se projektu věnovat i nadále a úspěšně naplnit naše další vize.

Literatura

[1] HANSSON, D. H. a J. KEMPER. *Project Info* [online]. 2009 [cit. 2012-05-23].

[2] MORAVEC, P., D. HARMANEC, P. TARANT a J. JEŽEK. *A practical approach to dealing with evolving models and persisted data* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.codegeneration.net/cg2012/sessioninfo.php?session=37>.

[3] HERRMANNSDÖRFER, M. *COPE* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://cope.in.tum.de/>.

[4] ECLIPSE FOUNDATION. *Modeling Framework* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.eclipse.org/modeling/emf/>.

[5] ECLIPSE FOUNDATION. *Ecore* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://wiki.eclipse.org/ecore>.

[6] ECLIPSE FOUNDATION. *Emfatic* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://wiki.eclipse.org/emfatic>.

[7] MICROSOFT. *ADO.NET Entity Framework* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://msdn.microsoft.com/en-us/library/bb399572.aspx>.

[8] JBOSS COMMUNITY. *Hibernate* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.hibernate.org/>.

[9] OMG. *Model Driven Architecture* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.omg.org/mda/>.

[10] ECLIPSE FOUNDATION. *MWE2* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://wiki.eclipse.org/modeling_workflow_engine_(mwe)>.

[11] KOLEKTIV. *PRISM* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://yellowstone.cs.ucla.edu/schema-evolution/>.

[12] TARANT, P. *Modelem řízená evoluce databáze*, 2012. Bakalářská práce.

[13] OMG. *QVT* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.omg.org/spec/qvt/1.1/>.

[14] OMG. *XMI* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.omg.org/spec/xmi/2.4.1/>.

[15] ECLIPSE FOUNDATION. *Xtext* [online]. 2012 [cit. 2012-05-23]. Dostupné z: <http://www.eclipse.org/xtext>.

Příloha A

Operace objektové vrstvy

A.1 Operace pro tvorbu entit

AddPrimitiveClass Přidá primitivní třídu reprezentující primitivní datový typ.

AddEmbeddedClass Přidá vloženou třídu bez atributů.

AddStandardClass Přidá standardní třídu s identifikátorem.

AddProperty Přídá atribut daných vlastností do obecné třídy.

A.2 Operace pro modifikaci entit

RenameEntity Přejmenuje danou entitu.

SetParent Nastaví rodiče, který je standardní třídou, standardní třídě.

SetAbstract Nastaví abstraktnost standardní třídě.

RenameProperty Přejmenuje daný atribut.

SetType Nastaví datový typ atributu.

SetOpposite Nastaví vazbu 1..1 mezi atributy

SetBoundries Nastaví dolní a horní hranici atributu.

SetOrdered Nastaví setříděnost kolekce.

SetUnique Nastaví unikátnost kolekce.

A.3 Operace pro odebrání entit

RemoveEntity Odstraní danou entitu.

RemoveProperty Odstraní daný atribut.

A.4 Komplexní operace

ExtractParent Extrahuje společného předka z množiny daných tříd.

ExtractClass Extrahuje množinu atributů z dané třídy.

Příloha B

Seznam použitých zkratek

DDL Data Definition Language

DML Data Manipulation Language

EMF Eclipse Modeling Framework

MDA Model Driven Architecture

MWE2 Modeling Workflow Engine 2

OCL Object Constraint Language

ORM Object-Relational Mapping

QVT Query/View/Transformation

SQL Structured Query Language

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. Eclipse pro svůj chod potřebuje v systému podporu jazyka JAVA. Dále je třeba mít nainstalovanou podporu jazyka Ruby[1], pod kterým nám běží naše aplikace pro komunikaci s databází. Jako databázový systém jsme zvolili PostgreSQL, kterému jsme také přizpůsobili syntaxi generovaných SQL příkazů, proto je doporučeno mít nainstalovaný právě tento databázový systém.

Přiložené médium obsahuje vývojové prostředí 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 projekt, který sestává z několika menších projektů, které je do prostředí Eclipse třeba importovat (*File→Import→Existing Projects into Workspace*).

Po uložení všech projektů do prostředí Eclipse už stačí pouze spustit příslušný 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[2]. Na stránkách projektu jsou k nahlédnutí zápisy ze schůzí, uzavřené a probíhající úkoly, milníky a wiki stránky. Také zde najdete další informace k motivačnímu příkladu[3].

[1] Ruby. <**http://www.ruby-lang.org/en/**>
[2] Dostupné z: <**https://github.com/migdb/migdb**>
[3] Dostupné z: <**http://migdb.github.com/migdb/**>

Příloha D

Obsah přiloženého CD

eclipse Nakonfigurované vývojové prostředí Eclipse Modeling Tools

migdb Projekt s naším frameworkem

bp Soubory, na kterých jsem pracoval

text Text a zdrojový kód mé bakalářské práce