# Analýza

## Úvodní shrnutí

Projekt migrace databáze se bude zabývat, jak již název napovídá, zjednodušením přechodu z jednoho databázového modelu na jiný, modifikovaný, model. Tento projekt se snaží automatizovat přechod na nový databázový model a tím urychlit celkový vývoj aplikace a redukovat množství kódu, který musí být psán ručně.

Projekt je zadán společností CollectionsPro, která postrádá modul, který se stará o aktualizování databázového modelu a přesunu již uložených dat v databázi. Model zadavatele je tvořen ve frameworku EMF (Eclipse Modeling Framework).

Náš software bude navazovat na aplikaci zadavatele. Náš projekt dostane data ve formátu EMF (reprezentovaného XML popisem) a sadu změn v domluveném formátu. Náš úkol začíná v té chvíli, kdy zadavatel vydá novou verzi své aplikace. V té chvíli bude vygenerován nový model aplikace a sada SQL příkazů, které přesunou původní data do nové datové struktury.



Obrázek - Vrstvy abstrakce vyvíjené aplikace

## Požadavky

1. Funkční požadavky
   1. Generování migračních SQL scriptů
   2. Přesunutí dat do nové databázové struktury
   3. Spolupráce s EMF (Eclipse Modeling Framework)
2. Systémové požadavky
   1. Databázový stroj PostgreSQL

## Use-case

### DDL operace

### https://lh4.googleusercontent.com/qqYeolO6DrYFsXFZX9So_tTGIl1Dfc4Kgjk_kAchvZ606pZqVUQmNEYmsdJZny0yYK0KLiFoq_6_KrEUny8OLXnBf3BlI16yg6qnFHvog0attFzgzQ

Obrázek - Model DDL operací, které bude naše aplikace zahrnovat

### Přejmenovat sloupec

Tato operace přejmenuje sloupec v databázi. Tato operace potřebuje znát tabulku, v které se sloupec nachází, sloupec tabulky, který chceme přejmenovat, a nové jméno sloupce.

### Přejmenovat tabulku

Tato operace přejmenuje v databázi tabulku. Ke svému spřávnému fungování potřebuje znát tabulku, kterou chceme přejmenovat, a nové jméno tabulky.

### Přidat sloupec

Tato operace přidá v databázi k tabulce sloupec. Operace musí znát tabulku, ke které má sloupec přidat, jméno nového sloupce a integritní omezení (constraints), které musí daný sloupec splňovat.

### Smazat sloupec

Tato operace smaže z databáze sloupec tabulky. Operace potřebuje znát konkrétní tabulku, z které chceme odstranit sloupec, a odstraňovaný sloupec.

### Smazat tabulku

Tato operace smaže z databáze tabulku. Ke svému správnému fungování musíme operaci odkázat na tabulku, kterou chceme smazat.

### Vytvořit tabulku

Tato operace vytvoří v databázi tabulku. Ke svému správnému fungování potřebuje následující informace. Operace musí znát jméno vytvářené tabulky, seznam sloupců, které se budou v tabulce nacházet, seznam integritních omezení (constraints) a vygenerovaný primární klíč.

### Změnit datový typ sloupce

Tato operace změní datový typ sloupce v databázi. Operace potřebuje znát tabulku, v které se daný sloupec nachází, konkrétní sloupce, kterému chceme změnit datový typ, a datový typ, na který chceme sloupec převést.

### Realizace scénářů

### https://lh4.googleusercontent.com/GvX5XEOI3TBip0454d_IbkAiQm7nZFqgTuz_AIDnDkMexHsk93gUOaphGGIHRWdX7LcAxeS2j90rhCPGOpzuAMTG98o4rAnOKk2TUQ8Q0znT-Ebd8g

Obrázek - Přehled jenotlivých scénářů

### Přidat předka a sourozence

*Scénář 1*

K samostatné existující třídě přidat předka a sourozence a (nyní) společné atributy převést do předka.

Scénář akcí

1) Vytvořit tabulku předka.

2) Vztvořit tabulku sourozence.

3) Nalézt společné atributy (sloupce) sourozenců.

4) Vytvořit v předkovi atributy (sloupce) odpovídající společným atributům (sloupcům).

5) Přesunout data ze sourozenců do rodiče.

6) Smazat společné atributy (sloupce) ze sourozenců.

### Přesunout atributu do předka

*Scénář 2*

V existující hierarchii tříd, přenést některé atributy ze sourozenců na rodiče s tím, že ve stávajícím modelu pouze někteří "sourozenci" mají odpovídající atribut a nemusí být přesně stejného typu (měl by být ale na rodičovský typ převoditelný). Případně opačná úloha "rozpuštění" atributu do části potomků.

    Scénář akcí

1) Nalezení atributů (sloupců), které chci přenést do předka

2) Vytvoření odpovídajících atributů (sloupců) v předkovi

3) Zjištění, zda si datové typy v předkovi a rodičovi odpovídají <<extend>>         přetypování dat v potomkovi na datový typ předka

4) Přesun dat z potomků na předka

5) Smazání přesunutých atributů (sloupců) z potomků

### Rozpustit atributy do potomků

*Scénář 3*

V existující hierarchii tříd, přenést některé atributy ze sourozenců na rodiče s tím, že ve stávajícím modelu pouze někteří "sourozenci" mají odpovídající atribut a nemusí být přesně stejného typu (měl by být ale na rodičovský typ převoditelný). Případně opačná úloha "rozpuštění" atributu do části potomků.

Scénář akcí

1) Zjištění atributů (sloupců), které chci rozpustit

2) Vytvoření atributů (sloupců) v potomcích do, kterých chci data přesunout

3) Přesun dat z atributů (sloupců) rodiče do potomků <<extend>> přetypování     atributů (sloupců) v potomcích

4) Smazání atributu (sloupce) v rodiči

## DDL operace

(Data Definition Language)

|  |  |  |  |
| --- | --- | --- | --- |
| operace | metameta model | meta model | data |
| table.add | jméno tabulky, sloupce, primary key, sequence, constrain | class Table {  row A;  } | CREATE TABLE název\_tabulky (název\_sloupce datový\_typ,... ); |
| table.rename | tabulka, nové jméno | class RenamedTab {  row A;  } | ALTER TABLE název\_tabulky RENAME nový\_název\_tabulky ; |
| table.del | tabulka |  | DROP TABLE název\_tabulky; |
| column.add | tabulka, jméno sloupce, constrains sloupce | class Table {  row A;  row B;  } | ALTER TABLE název\_tabulky ADD COLUMN název\_sloupce datovy\_typ; |
| column.rename | tabulka, sloupec, nové jméno sloupce | class Table {  row A;  row Renamed;  } | ALTER TABLE název\_tabulky MODIFY název\_sloupce nové\_nastavení; |
| column.typechange | tabulka, sloupec, nový datový typ sloupce |  | ALTER TABLE  název\_tabulky  CHANGE název\_sloupce datový\_typ; |
| column.del | tabulka, sloupec | class Table {  row A;  } | ALTER TABLE název\_tabulky DROP název\_sloupce; |

## Datové typy

přehled datových typů v PostgreSQL: <http://www.postgresql.org/docs/7.4/interactive/datatype.html>

|  |  |  |
| --- | --- | --- |
| Vstup | Model | PostgreSQL |
| string | EString | text |
| int | EInt | integer |

## Emfatic

Emfatic je jazyk speciálně navržený pro reprezentaci EMF Ecore modelu v textové formě. Plugin do eclipse umožňuje automatický přechod mezi emfaticem a Ecore modelem. Syntaxe emfaticu je poměrně jednoduchá a svým zápisem se velice podobá programovacímu jazyku Java.

Mezi základní konstrukty emfaticu patří balíčky (package), třídy (classes), datové typy (Data Types), výčtové typy (enum) a mapy (Map Entries). Význam těchto konstruktů je stejný jako v jazyce java. Dále třída může obsahovat následující klíčová slova attr (atribut primitivního typu), op (operace - odpovídá metoďě v jazyku java), ref (reference - odkaz na jinou třídu), val (reference - třída vlastní další třídu). Další konstrukty, které se v emfaticu používají jsou například OCL notace, které model omezují, nebo jsou pomocí OCL generována celá těla metod.

Zákazník používá emfatic pro popis metamodelu, z kterého následně generuje jednotlivé java třídy.

## Query/View/Transformation - QVT

QVT je transformační jazyk, který se zabývá transformacemi na úrovni meta-modelů. V naší aplikaci využíváme QVT Operational. Jedná se o imperativní implementaci QVT, která je nativně podporovaná vývojovým prostředím Eclipse.

Transformace QVT probíhá následujícím způsobem. Na vstupu transformace jsou dva, nebo i více, meta-modelů. Vždy je alespoň jeden označen jako vstupní meta-model, a jiný z meta-modelů je označen jako výstupní meta-model. Podle těchto meta-modelů se bude provádět nadefinovaná transformace. K samotné transformaci se používá několik základních konstruktů: mapping, query a helper. Tyto konstrukty mění strukturu modelu (transformují ji), dle námi zadaných podmínek. Daly by se přirovnat k metodám, či funkcím v jiných programovacích jazycích. Jelikož se jedná o imperativní jazyk, můžeme uvnitř transformačních funkcí deklarovat proměnné, rozhodovat se pomocí konstruktů if then else, využívat cyklu when a mnoho dalších podobných konstruktů, které známe z jiných programovacích jazyků.

Podrobnou specifikaci tohoto jazyka naleznete v následujících dokumentech:

vezre 1.0: <http://www.omg.org/spec/QVT/1.0/PDF/>

verze 1.1: <http://www.omg.org/spec/QVT/1.1/Beta2/PDF> - draft

## Poznámky k QVTo

V průběhu našeho studování specifikace QVTo jsme si dělali poznámky, abychom se lépe orientovali v dané problematice a byli schopni předat naše zkušenosti mezi sebou či studentům pokračujícím v projektu. Varování - tyto poznámky obsahují náš subjektivní pohled, nezaručujeme jejich naprostou správnost. Autoři poznámek předpokládají čtenářovu základní znalost programovacího jazyka Java, proto se na něj budou odkazovat. Klíčová slova budou zvýrazněna tučně.

### Deklarace proměnných

* deklarace má tvar **var**     <identifikator> : Typ

***př.     var*** *a: String*

### Komentáře

* Komentáře se dělají jako v Javě
* existují     dva typy:
  + Jednořádkové

***př.*** *//Toto je jednořádkový komentář*

* + Obecné (víceřádkové - řádkově neomezené)

***př.*** */\*Toto je*

*dvouřádkový komentář \*/*

### Podmínka If then else

* Podmínka má tvar:

**if** ( condition ) **then**     {  
            */\*statement1 \*/*  
} **else** {  
             */\* statement2 \*/*  
} **endif;**

* závorky ani **else** větev nejsou potřebné, dají se vynechat(závorky značí jen blok, podobně jako v Javě a jiných jazycích), středník za endif ani ostatní části vynechatelné nejsou

***př.* *if****(0 = 0)* ***then******log****("Common")* ***else*** *l****og*** *("Miracle")* ***endif;***

### Tabulka základních operátorů

|  |  |
| --- | --- |
| Popis | Operátor |
| Konec výrazu | ; |
| Přiřazení  Objectu nebo Setu | := |
| Přidání položky do Setu | += |
| Operátor je rovno | ' = ' |
| Operátor není rovno | <> |
| Operátory přístupu | . -> |

* Z tabulky jsou     zajímavé obzvlášť operátory     **+=** a **<> ,** které nejsou v  programovacích jazycích obvyklé
* S operátorem :=je možné použít tzv. *conditional expression* k zkrácení zápisu. Tento operátor slouží k přiřazení hodnoty proměnné

**př.     var** a: String  
    **var** b: Integer;  
    a:=( **if** (b **=** b ) **then** “Logické” **else** “Zázrak“ )

* Operátor = slouží k přiřazení reference nebo k porovnání
* Operátor přístupu **.** sloužík přístupu k jednotlivým proměnným
* Operátor přístupu -> se používá k přístupu k položkám a metodám kolekcí

### Switch

* příkaz **switch** se používá k řízení toku podobně jako v Javě, nicméně nepoužívá jeden výraz, ale vždy za klíčovým slovem **case** následuje podmínka, klíčové slovo **else** je použito, pro případy, kdy není splněna žádná z předchozích podmínek

***př.* switch** {

**case**    (condition1)     */\* Statement1 \*/*  
            **case**    (condition2)     */\* Statement2 \*/*  
            **else**     */\* Statement3\*/*  
        }

### Self

* klíčové slovo     **self** je     equivalentní k this v Javě – získáváte pomocí něj přístup k objektu, nad kterým pracujete ( jeho metodám, atributům)

**př.** self.name :=””

### Result

* Klíčové     slovo **result** pracuje     podobně jako Self s atributy a metodami, ale nepracuje s vstupním objektem, ale výstupním, lze ho použít v **mapping**u

### cyklus while

* cyklus while se používá k opakovanému provádění těla cyklu, stejně jako v Javě

**př. while**    (condition){

            /\*statement \*/  
        }

### Deklarace transformace

* skládá se z jména, vstupního a výstupního metamodelu
* může být poděděna pomocí klíčového slova **extends**, v tom případě potomek dědí všechna mapování a dotazy, které může předefinovat
* klíčové     slovo **access**     má podobnou funkci, ale narozdíl od klíčového slova **extends** není možné cokoliv předefinovat, celá transformace je použita jako celek
* klíčová     slova **new** a **transform** slouží k instanciaci přijaté transformace

### Modeltype definition

* definice typu modelu - reference na modeltype nebo je možné vložit celou definici (inline definice)
* může referencovat na lokální file*(př.     1)* nebo je definována pomocí reference na package namespace URI *(př. 2)*
* local specific reference - v Eclipse se to dělá     prefixací “platform:/resource/“, za kterou následuje relativní cesta k souboru v workspacu

**př. 1**     **modeltype** MM1 **uses** “platform:/resource/MM1toMM2/transforms/MM1.ecore“

**př. 2    modeltype** MM1 **uses** “http://mm1/1.0“

### Helper

* Operace, která vykonává výpočet na jednom nebo více objektech (parametrech) a tvoří výsledek. Tělo helperu je uspořádaný seznam výrazů, které jsou vykonány v     řadě po sobě (v sekvenci). Helper může jako vedlejší efekt modifikovat parametry.
* Pozn: Autoři textu nevyužívali helpery a zadavatel vyslovil podezření na nefunkčnost helperů v současné verzi QVTo, předcházející definice je vytažena z QVTo Specifikace

### Query

* Query je helper bez vedlejších efektů, tzn nemění vstupní „objekt“

    **př. query** APP::reduced::Property::isID():Boolean{  
               if(self.serialization.isID = true) then {  
                   return true;  
               }endif;  
               return false;

}

### when

* podmínka následující po klíčovém slově **when** je nazývána pre-condition nebo též guard.
* k provedení daného mapování musí být tato precondition splněna
* **tvar:** ***mapping*** *MM1::Model::toModel() : MM2::Model*

            **when** {**self.**Name.startWith(“M”);}  
                { //konkrétní mapping }

### Disjuncts

* seřazený seznam mapování.
* je zavolané první mapování, jehož **guard** (typ a podmínka uvozená klíčovým slovem **when**)     je platný
* pokud není platný žádný guard je vrácena hodnota *null*
* pomocí disjuncts lze     nahradit nemožnost přetížení mapování

**př.     mapping** UML::Feature::convertFeature () : JAVA::Element

**disjuncts** convertAttribute, convertOperation, convertConstructor() {}

**mapping** UML::Attribute::convertAttribute : JAVA::Field { name := self.name; }

**mapping** UML::Operation::convertConstructor : JAVA::Constructor **when** {

            self.name = self.namespace.name;} { name := self.name; }

**mapping** UML::Operation::convertOperation : JAVA::Constructor **when** {

            self.name <> self.namespace.name;} { name := self.name; }

### Main funkce

* účel funkce main() je nastavit proměnné prostředí a zavolat první mapování

### log(“message”)

* vypisuje zprávu do konzole

### Assert

* má tři levely: warning,     error a fatal
* Při nesplnění assertu levelu fatal transformace skončí
* **Je tvaru**: **assert** **level** (condition) with log(“message”)
* **Pozn.** V příkladě od zadavatele je assert bez levelu, nejspíš je implicitní level     warning nebo error

***př.* assert** **warning** (self.x > 2) **with log**(“Hodnota x je menší než 2 a je rovna:” + self.x)

### Map vs xmap

* **map**     - v případě, že se neprovede mapování, vrátí null
* **xmap**     – v případě, že se nepovede mapování, vyvolá výjimku

### Dictionary

* Kolekce (container) =Javovská Map - uskladňující data uspořádaná podle klíče
* úplný popis operací viz 8.3.7 v specifikaci QVT
* operace:
  + Dictionary( KeyT , T ) :: get ( k : KeyT ) : T
  + Dictionary( KeyT , T ) :: hasKey ( k : KeyT ) : Boolean
  + Dictionary( KeyT , T ) :: put ( k : KeyT , v : T ) : Void
  + Dictionary ( KeyT , T ) :: size ( ) : Integer
  + Dictionary(KeyT,T)::values() : List(T)
  + Dictionary(KeyT,T)::keys() : List(KeyT)
  + Dictionary(KeyT,T)::isEmpty() : Boolean

***př. var*** *x:****Dict****(String,Actor);* // Dictionary Itemů typu Actor s klíčem String

### ForEach

* Iterátor nad kolekcí, provede tělo pro **všechny** prvky, pro něž je zadaná podmínka platná

**př.** self.allSubobjectsOfType(Cifx::Update).oclAsType(Cifx::Update) ->  
            forEach(Upd) {  
                resets += Upd.QueryFromUpdate();  
            };  
    //self.allObjectsOfType(Cifx::Update).oclAsType(Cifx::Update) - všechny podpoložky         //typu Cifx:: Update přetypuje na Cifx::Update  
    //každou z těchto položek přidá do kolekce resets

### ForOne

* Iterátor nad kolekcí, provede tělo pro **první**     prvek, pro který je zadaná podmínka platná, pro další prvky     již ne
* **forEach** i **forOne** jsou popsány v specifikaci v sekci **8.2.2.6**

***př.*** *self.allSubobjectsOfType(Cifx::Update).oclAsType(Cifx::Update) ->*  
*forOne(Upd) {*  
*resets += Upd.QueryFromUpdate();*  
*};*  
*//self.allObjectsOfType(Cifx::Update).oclAsType(Cifx::Update) - všechny podpoložky         //typu Cifx:: Update přetypuje na Cifx::Update*  
*//první položku typu Cifx::Update přidá do kolekce resets*

## Zhodnocení - kritika naší transformace

    Naše transformace byla vytvořena s malou zkušeností a znalostí v QVTo světě, proto jsme ne úplně vždy využívali ty nejlepší konstrukce, ale spíše ty, co nás napadli jako první. Tato část dokumentu popisuje, co se nám nyní zdá jako problematické či špatně naimplementované a v některých případech naznačuje, jak by se to dalo opravit.

Porovnávání pomocí for-each a rovnosti jména není ten nejlepší nápad. Zadavatel práce navrhoval použití konstrukce *(select): c->select( v : T | b(v)     ):: Collection(T*)

***viz.***<http://www.csci.csusb.edu/dick/samples/ocl.html>

*Znalost OCL je při používání QVTo transformací ceněná schopnost. Proto je  dobré se OCL naučit v případě, že budete QVTo a nejspíše i jiné transformační jazyky používat. Například v námi vytvořené transformaci nejsou zapsány některé přiřazení, ale ve výsledku jsou reference např na owningSchema z Tabulky. Toto zajišťuje právě OCL.*

V transformaci není použito mapování MtoN, protože není dovymyšleno, jak vytvořit mezitabulku.

FK referencuje sloupec z cizí tabulky, toto zatím také nebylo řešeno

PK dědí od UniqueIndex, takže obsahuje UnderlyingIndex typu Index a má odkazovat na sloupce tabulky, kterých se týká, toto také zatím chybí v transformaci.

Opravovat constraints po namapování na Tabulky z RDB není nejšťastnější (ale byla to nejjednodušší cesta, která se naskytla)

Dále nám zadavatel doporučil nastudovat konstrukty a některé specifické situace v jím zaslaném prikladě QVTo transformace:

* mapping
* when (omezujici podminky pro mapping)
* inherits (dedicnost - i nasobna)
* disjuncts (deklarativni vyber variant mapovani objektu)
* explicitni instanciace objektu - konstrukt object x : Type { }
* init sekce mappingu a co se stane, když v rámci ní zapíšete do proměnné result
* abstract mapping
* chápat a odlišovat map vs. Xmap

Dále nám také dal radu, abysme používaly helpery a query operace - nejspíše kvůli reusability napsaného kódu.