Migrace databáze (MigDB)

Předmět: Y36SI3 Realizace programových systémů

Řešitelé:

Tomáš Herout - *vedoucí týmu*
Martin Lukeš
Zdeněk Pecka
Michal Líška

Obsah

Licence (MIT Licence)

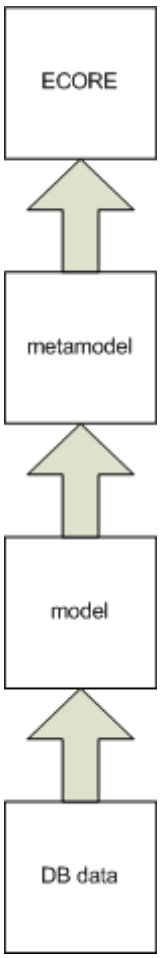
Analýza

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.



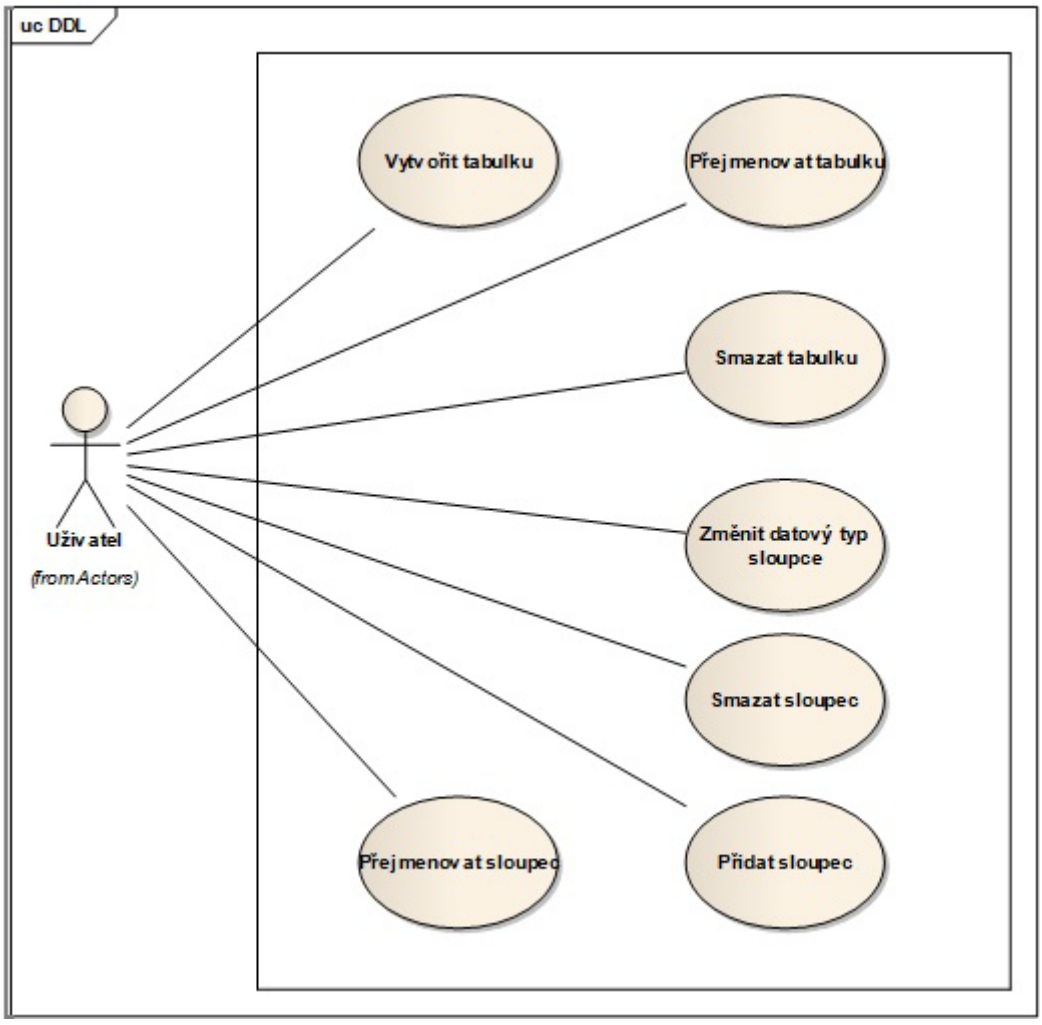
Obrázek 1 - Vrstvy abstrakce vyvíjené aplikace

Požadavky

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

Use-case

DDL operace



Obrázek 2 - 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.

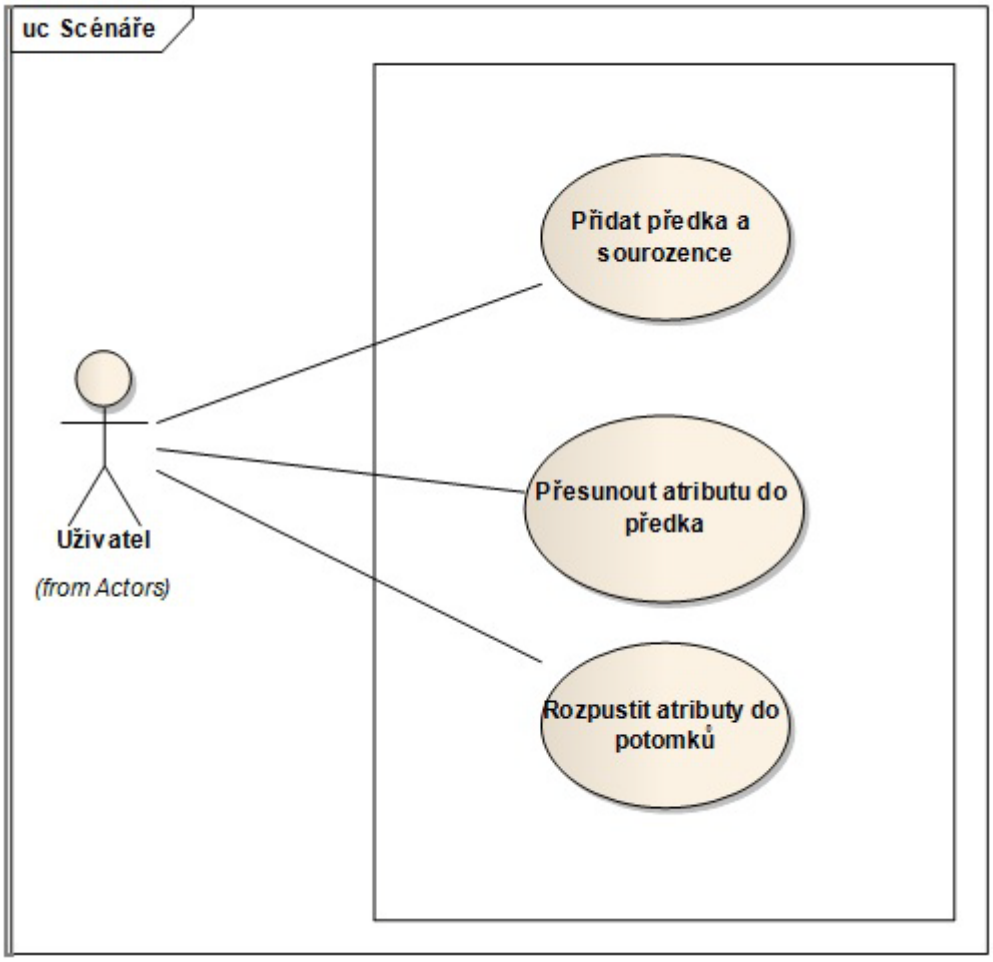
Vytvořit tabulku

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



Obrázek 3 - 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 metoděv 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 **.** slouží k 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:

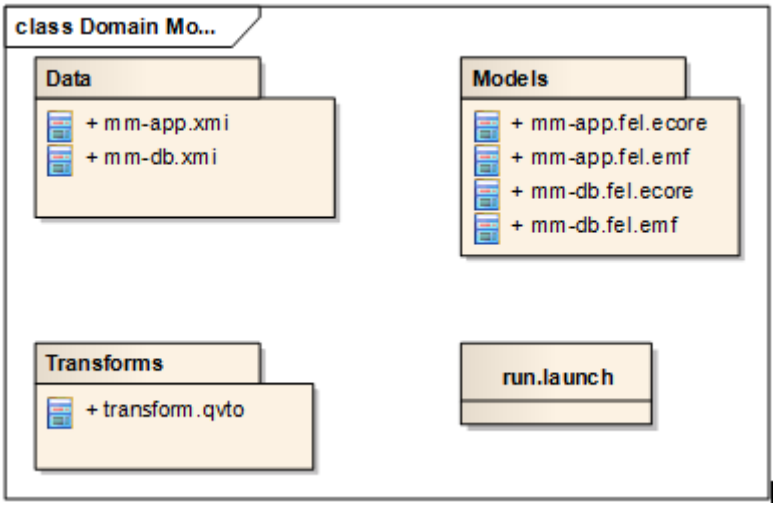
- mapping
- when (omezujici podminky pro mapping)
- inherits (dedicnost - i nasobna)
- disjuncts (deklarativni vyber variant mapovani objektu)
- explicitni instanciace objektu - konstrukt object x : Type { }
- init sekce mappingu a co se stane, když v rámci ní zapíšete do proměnné result
- abstract mapping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

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

Návrh

Struktura aplikace

Na následujícím obrázku můžete vidět jednotlivé stavební části námi navrhované aplikace.



Obrázek 4 - Zde jsou znázorněny jednotlivé adresáře a soubory naší aplikace

Adresář Data

Adresář data obsahuje konkrétní modely dat. Konkrétními modely rozumíme takové modely, které odpovídají své specifikaci, neboli jejich struktura je popsaná metamodely z adresáře models. V našem případě se v tomto adresáři nacházejí dva soubory. Jedná se konkrétně o soubory mm-app.xmi a mm-db.xmi. Tyto soubory reprezentují konkrétní data, s kterými navržená transformace pracuje.

Soubor mm-app.xmi

Tento soubor je vstupní soubor pro transformace, který obsahuje konkrétní data (třídy, atributy, jednotlivé vztahy mezi třídami), kterými je struktura modelu tvořena. Nad těmito daty jsou prováděny transformace, které data přetvářejí do struktury popsané jiným metamodelem.

Soubor mm-db.xmi

Jedná se o výstupní soubor, který je vytvořen v průběhu transformací. Obsahuje data, která byla popsána v souboru mm-app.xmi, ovšem tyto data odpovídají jiné struktuře, která je popsaná příslušným metamodelem.

Adresář Models

Adresář models obsahuje metamodely, podle kterých se budou provádět jednotlivé transformace. Každý metamodel se zde nachází ve dvou formách. První formou je zápis metamodelu v jazyku Emfatic (soubor s příponou .emf), který je pro člověka velice pěkně čitelný. Druhý soubor je vygenerovaný z Emfatic souboru a reprezentuje zápis v jazyku Ecore (přípona .ecore). Ecore metamodel je použit jako vstupní i výstupní metamodel pro transformace. Je tomu především proto, že zápis v Ecore je dobře strukturovaný a tudíž pársovatelný počítačem. Metamodely udávají strukturu, kterou konkrétní modely musí splňovat.

Soubory mm-app.fel.ecore a mm.app.fel.emf

V těchto souborech se nachází metamodely, podle kterých se budou jednotlivé transformace provádět. Metamodely vlastně předepisují konstrukty a omezení, které budou muset konkrétní modely splňovat. V těchto metamodelech se předepisuje struktura pro aplikační modely (vstupní modely do transformace). Definují se zde třídy, atributy, operace.

Soubory mm-db.fel.ecore a mm.db.fel.emf

Tyto soubory popisují metamodel pro výstupní data z transformace. Tento metamodel se zabývá definicí modelů na úrovni databáze. Je zde specifikováno, jak má vypadat tabulka, sloupec, primary key, forein key, indexy, sekvence, constrains a jsou zde také popsány základní DDL operace, jako vytvoř tabulku, přejmenuj sloupec atd.

Adresář Transforms

Tento adresář obsahuje soubory, které jsou zodpovědné za samotnou transformaci. V našem případě se zde nachází pouze jeden soubor a to transform.qvto

Soubor transform.gvto

Tento soubor se zabývá konkrétní transformací vstupního modelu na výstupní. V našem případě má na vstupu model, který odpovídá metamodelu mm-app.fel.ecore, jehož data převádí na výstupní model, který odpovídá modelu mm-db.fel.ecore. Jinými slovy, převádí strukturu, která je popsaná pomocí tříd, atributů, atd. na strukturu, která odpovídá databázovému světu, tedy tabulky, sloupce atd.

Soubor run.launch

Tento soubor spouští celou transformaci. Jsou zde například uvedené informace o umístění transformačních skriptů, vstupních i výstupních modelů a spoustu dalších inicializací.

Technologie

Použité technologie do značné míry zastřešuje EMF, **neboli Eclipse Modeling Framework**. Jedná se o souhrn modelovacích nástrojů pro vývojové prostředí Eclipse. Eclipse i celý koncept EMF jsme použili z důvodu kompatibility s aplikací, do které bude náš modul umístěn.

Používali jsme především jazyky, které popisují modely. Jedná se zejména o **Ecore**, který je založen na XML, a **Emfatic**, který je velice přehledný a svým zápisem se podobá programovacímu jazyku Java. Mezi těmito modely se dá jednoduše přecházet (z jednoho zápisu se dá vygenerovat zápis druhý). V jazyku Emfatic jsme modely vytvářeli, následně jsme vygenerovali jemu odpovídající model v Ecore a ten jsme dále programově zpracovávali.

Dále jsme používali transformační jazyk. Naše volba padla na transformační jazyk **QVT Operational**. Tento jazyk jsme zvolili kvůli jeho snadné integraci do Eclipse, ale především proto, že zadavatel má s tímto jazykem bohaté zkušenosti. Jazyk QVT nám umožní převést datovou strukturu odpovídající modelu A (původní) na strukturu odpovídající modelu B (nový model) pomocí definované transformace.

Upozornění na budoucí místa interakce/interface

EMF (Emfatic)

Ve formátu EMF nám bude poskytnuta struktura aplikace a sada změn, které se objevily při přechodu na novou verzi.

Postgres SQL

Relační databáze, nad kterou budou prováděny změny v modelu.

MigDB - testy

Testování je důležitou, dalo by se říci až nedílnou součástí, každého úspěšného projektu. Přispívá ke zkvalitnění vyvíjené aplikace a ověřuje, zda byly splněné všechny požadavky zákazníka na systém. Zákazník také často požaduje stoprocentní otestování dodávané aplikace. Tento požadavek se v reálném vývoji aplikací nedá splnit, ale můžeme se mu přiblížit tím, že budeme postupovat podle předem naplánovaných procesů (viz. dokument plánování testů) a hlavně nebudeme testování odkládat až na konec projektu. Je dokázáno, že čím dřív se chyba opraví, tím má menší ekonomické dopady na celý projekt.

Jelikož jsme aplikaci vyvíjeli v iteracích, objeví se zde popis všech testů, které jsme v jednotlivých fázích projektu prováděli.

Testy prvního prototypu

První prototyp, který byl podroben testům, se zabýval modifikací vytvořeného modelu v ecore a generováním SQL příkazů.

Vzhledem k použitému programovacímu jazyku - Java, byl jako framework pro testování zvolen **jUnit**. Tento framework se opírá o tzv. *jednotkové testy*, které lze chápat jako testování jednotlivých částí. Ideální stav je vše pokryté testy (tzv. code coverage) a pro produkční nasazení musí být všechny testy úspěšné.

Testované třídy

následuje seznam testovaných tříd, popis metodiky a úspěšnost

migdb.sql.DialectSQLTest

Tento test si klade za cíl testování správné implementace rozhraní SQLDialect. Cílem je tedy ověřit, že všechny podporované dialekty jsou popsány. V současné době je podporován pouze PostgreSQL, nicméně při přidání dalšího dialektu je při současném návrhu testu snadněji zaručeno, že dojde k otestování všech nezbytných údajů - test se provádí nezávisle na konkrétním dialektu.

Test pokrývá jedinou metodu, kterou výše uvedené rozhraní definuje.

Test je úspěšný.

migdb.ecore.EcoreUtilTest

Během tohoto testu se provádí v současnosti všechny metody třídy EcoreUtil, které mění model. Jedná se o přidání/přejmenování/smazání třídy/atributu (celkem tedy 6 metod). Během testu je postupně vytvořena třída, přidán atribut, přejmenována třída, přejmenován atribut, smazán atribut, smazána třída.

Po každé operaci se testuje, zdali operace byla provedena úspěšně (přibylo co mělo, zmizelo, co mělo).

Test pokrývá veškeré metody, které modifikují model.

Test je úspěšný.

Testy druhého prototypu

V této části vývoje jsme se oprostili od myšlenky vyvíjet aplikaci v programovacím jazyku Java a přistoupili jsme na transformační jazyk. Jako transformační jazyk jsme zvolili QVT Operational, který je součástí EMF (Eclipse modeling framework) ve vývojovém prostředí Eclipse.

Pro testování našich transformací jsme zvolili následující postup. K provádění transformace a k jejímu následnému testování máme k dispozici několik modelů. Prvním z modelů je vstupní model. Vstupní model vyjadřuje počáteční stav, z kterého vycházíme při inicializaci transformací. Druhým modelem je výstupní model. Výstupní model reprezentuje model, který chceme dostat po provedení všech transformací. Dále máme k dispozici soupis změn, které popisují cestu, jak se ze vstupního modelu vytváří výstupní model. Z těchto souborů se dají vytvořit testovací data, která budeme předkládat testované transformaci.

Jednotkové testování (Unit-testing) provádíme tím způsobem, že testujeme zvlášť každý mapping, query i helper, zda dělají to, co od nich očekáváme.

Testování u zákazníka

Před předáním produktu zákazníkovi je plánováno testování v produkčním prostředí.

Infrastruktura

Tento dokument popisuje infrastukturu, kterou jsme k řešení projektu využívali. U jednotlivých nástrojů je popsán účel, ke kterému se konkrétní nástroj používal, a nechybí ani zhodnocení vhodnosti jednotlivých nástrojů.

Sdílení kódu a dokumentů

Pro sdílení kódů a dokumentů jsme používali tři nezávislé systémy. Hlavním důvodem proč jsme se tak rozhodli, byl fakt, že pracujeme na projektu s externím zadavatelem, který nemá přístup do "fakultního" repozitáře. Proto jsme zřídili externí repozitář, který slouží ke sdílení informací mezi řešitelským týmem a zadavatelem projektu.

Rabbit.felk.cvut.cz

Jedná se o SVN repozitář, který je pro projekt založen automaticky. Používání tohoto repozitáře je povinné. Tento repozitář slouží především pro komunikaci týmu s cvičícím. Do tohoto repozitáře jsme ukládali všechny finální verze dokumentů a kódů.

Zhodnocení:

Systém je vhodný pro komunikaci týmu a cvičícího, především proto, že cvičící má veškeré informace od všech svých studentů na jednom místě.

Náš tým měl s tímto systémem drobné potíže. Při nahrávání větších souborů, se někdy nahrávaný soubor poškodil a opravit ho zabralo poměrně dost práce. Také jsme měli problémy se zadáváním úkolů přes "tickets", které ze začátku nefungovalo.

GIThub.com

Jedná se o GIT repozitář, který jsme si v průběhu projektu sami založili. Tento repozitář byl využíván především pro komunikaci se zákazníkem.

Zhodnocení:

Tento repozitář se nám velice osvědčil a můžeme ho jen vřele doporučit.

GoogleDocs

GoogleDocs jsme používali pro sdílení dokumentů. Zde měl každý člen týmu k dispozici vždy aktuální verzi dokumentu a viděl změny, které se v dokumentu udály.

Zhodnocení:

Tato webová aplikace se nám velice osvědčila. Především z toho důvodu, že každý člen týmu měl vždy k dispozici aktuální verzi dokumentu, a na jednom souboru mohlo pracovat více lidí najednou aniž by docházelo ke konfliktu.

Vývojové prostředí

Při vývoji aplikace jsme používali dvě vývojová prostředí.

NetBeans IDE

Toto vývojové prostředí jsme používali v první části projektu k psaní Java kódu a jeho testování.

Zhodnocení:

Toto vývojové prostředí se ukázalo jako ne příliš ideální pro vývoj našeho projektu, především kvůli infrastruktuře zadavatele.

Eclipse

Toto vývojové prostředí jsme využívali v druhé části projektu. Spolupráce s ním byla jedním ze systémových požadavků od zadavatele na výsledný software. Integrace do již existující části aplikace, která Eclipse využívá, plně vysvětluje zásadní význam tohoto požadavku.

Zhodnocení:

Pro náš projekt se Eclipse jevil jako ideální vývojové prostředí. Eclipse poskytuje řadu plug-inů, které nám vývoj velice usnadnily.

Komunikace týmu

Komunikace v týmu je velice důležitá věc a my jsme ji dosahovali následujícími nástroji.

GoogleGroups

GoogleGroups jsme používali jako mailling list.

Zhodnocení:

Pro tento druh aplikace je tato aplikace k nezaplacení. Každý člen týmu ví, co se v projektu děje. Při posílání emailu všem členům týmu si stačí pamatovat pouze jednu adresu.

Jabber

Jabber jsme používali pro rychlou komunikaci nebo řešení závažných problémů.

Zhodnocení:

Jabber se nám osvědčil hlavně v případech, kdy se potřebovalo něco rychle vyřešit, nebo k ujasňování detailů práce.

ICQ

ICQ jsme využívali stejným způsobem jako Jabber, tedy k rychlé komunikaci a řešení problémů.

Zhodnocení:

Přínos ICQ je v podstatě stejný jako u Jabberu. Jediným důvodem, proč jsme využívali dvě podobné služby, jsou preference jednotlivých členů týmu.

MigDB - rešerše

O projektu

Projekt migrace databáze (dále jen MigDB) si klade za cíl vyřešit problematiku převodu databázového schématu na nově definovaný při zachování dat.

Domovská stránka projektu je https://rabbit.felk.cvut.cz/trac/migdb.

Technologie

Během vývoje aplikace jsme se setkali s celou řadou softwarových technologií souvisejících s modelováním dat, které lze rozdělit do několika oblastí:

1. Eclipse Modeling Framework
2. jazyky popisující modely
3. transformační jazyky
4. třídy (API) pro manipulaci s modely

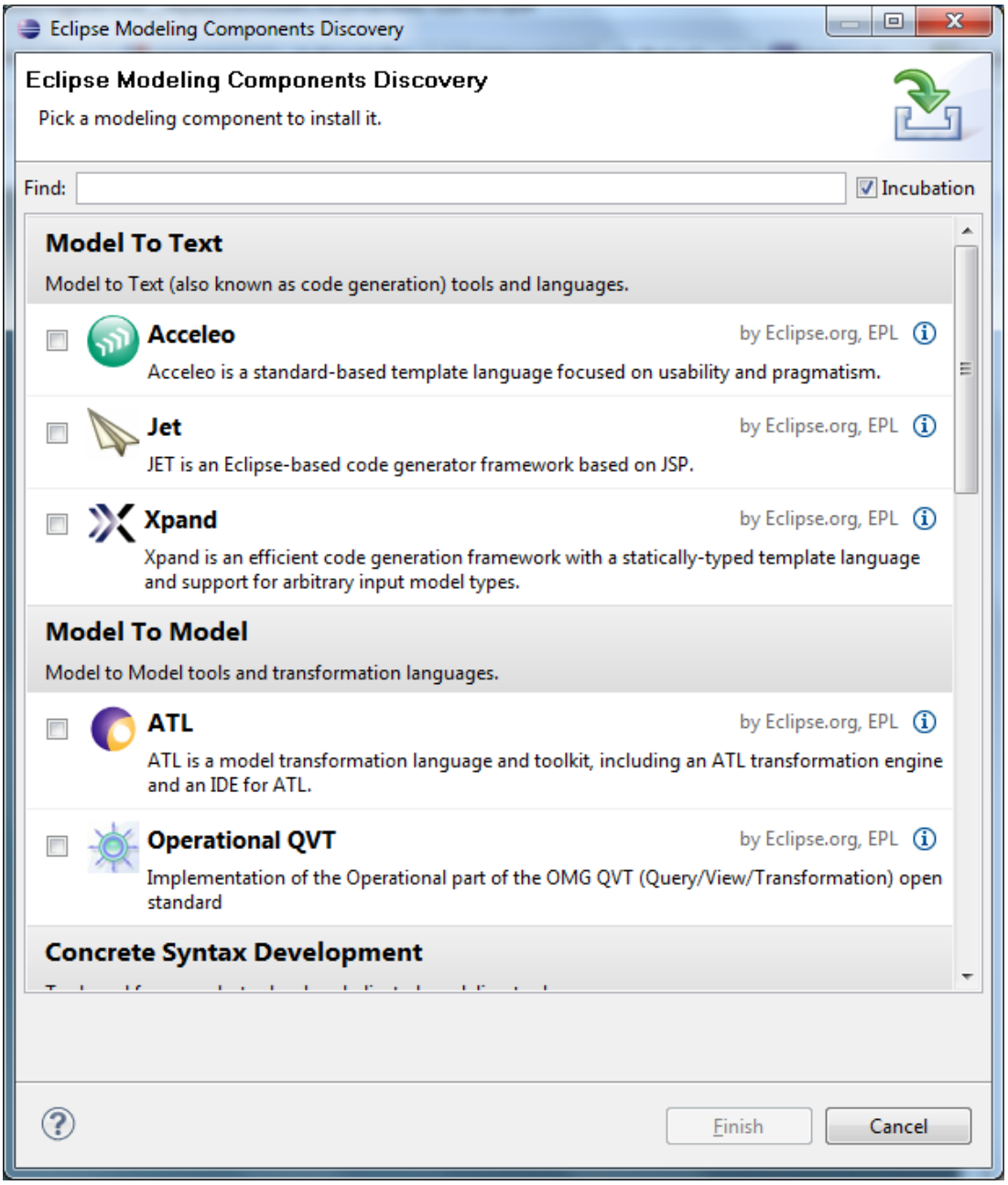
Následuje podrobnější popis jednotlivých oblastí.

Eclipse Modeling Framework

Veškeré, níže popsané věci, lze snadno používat v rozhraní Eclipse - podpora modelů, transformačních jazyků a prací s nimi v prostředí Eclipse je cílem projektu **Eclipse Modeling Framework** (EMF) - http://www.eclipse.org/modeling/emf/

Není-li zapotřebí konkrétních verzí jednotlivých komponent EMF, pak nejsnažší cestou ke zprovoznění všech potřebných věci je stáhnout z http://www.eclipse.org/downloads/ vydání **Eclipse Modeling Tools (includes Incubating components)**.

Toto sestavní již zahrnuje podporu Ecore a většinu ostatních komponent lze snadno přidat. Po spuštění je v menu pod položkou Help volba Install Modeling Components:



Obrázek 5 - rozšíření pro Eclipse

Pro náš projekt je nejduležitější **Operational QVT**, další co stojí za vyzkoušení je např. ATL a Acceleo.

Instalace požadovaných komponent probíhá snadno - v instalátoru stačí odsouhlasit výběr, později licenci, počkat až dojde ke stažení požadovaných pluginů a nakonec restartovat Eclipse.

Výjimkou je podpora Emfatic modelů - musíme zvolit Help > Install New Software..., tlačítkem Add přidat reposirotory http://scharf.gr/eclipse/emfatic/update (název např. Emfatic) a pod Uncategorized zvolit **Emfatic (Incubation)**. Zbytek instalace je identický s předchozím způsobem.

Jazyky popisující modely

Pro nás klíčovým formátem je **Ecore** (používá se přípona .ecore). Ecore v datové podobě odpovídá XMI (XML Metadata Interchange), díky XML je formát čitelný i v případě nedisponování

vhodným software pro čtení tohoto souboru. Větší přednost spočívá především ve snadném strojovém zpracování (čtení/vytváření/modifikace). Za tímto účelem lze využít již hotových nástrojů, které jsou uvedeny níže. Bohužel formát není úplně vhodný pro ruční psaní - režie (poměr obsah vs. forma) je poměrně vysoká.

Ukázka Ecore modelu v datové podobě:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="Families">
  <eClassifiers xsi:type="ecore:EClass" name="Family">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="lastName" ordered="false"
        unique="false" lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="father" ordered="false"
        lowerBound="1" eType="ecore:EClass test.ecore#/0/Member" containment="true"
        eOpposite="test.ecore#/0/Member/familyFather"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="mother" ordered="false"
        lowerBound="1" eType="ecore:EClass test.ecore#/0/Member" containment="true"
        eOpposite="test.ecore#/0/Member/familyMother"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="sons" ordered="false" upperBound="-1"
        eType="ecore:EClass test.ecore#/0/Member" containment="true" eOpposite="test.ecore#/0/Member/familySon"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="daughters" ordered="false"
        upperBound="-1" eType="ecore:EClass test.ecore#/0/Member" containment="true"
        eOpposite="test.ecore#/0/Member/familyDaughter"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Member">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="firstName" ordered="false"
        unique="false" lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="familyFather" ordered="false"
        eType="ecore:EClass test.ecore#/0/Family" eOpposite="test.ecore#/0/Family/father"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="familyMother" ordered="false"
        eType="ecore:EClass test.ecore#/0/Family" eOpposite="test.ecore#/0/Family/mother"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="familySon" ordered="false"
        eType="ecore:EClass test.ecore#/0/Family" eOpposite="test.ecore#/0/Family/sons"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="familyDaughter" ordered="false"
        eType="ecore:EClass test.ecore#/0/Family" eOpposite="test.ecore#/0/Family/daughters"/>
  </eClassifiers>
</ecore:EPackage>
```

Obrázek 6 - Ecore model

Dalším formátem je **Emfatic** (.emf). Přednostní Emfaticu oproti Ecore je právě zaměření na jeho plain-textovou podobu:

```
        @OCL(inv= "self.owningTable = self.underlyingIndex.indexedTable")
class UniqueIndex extends TableConstraint {
        ref Index[1] underlyingIndex;
}
```

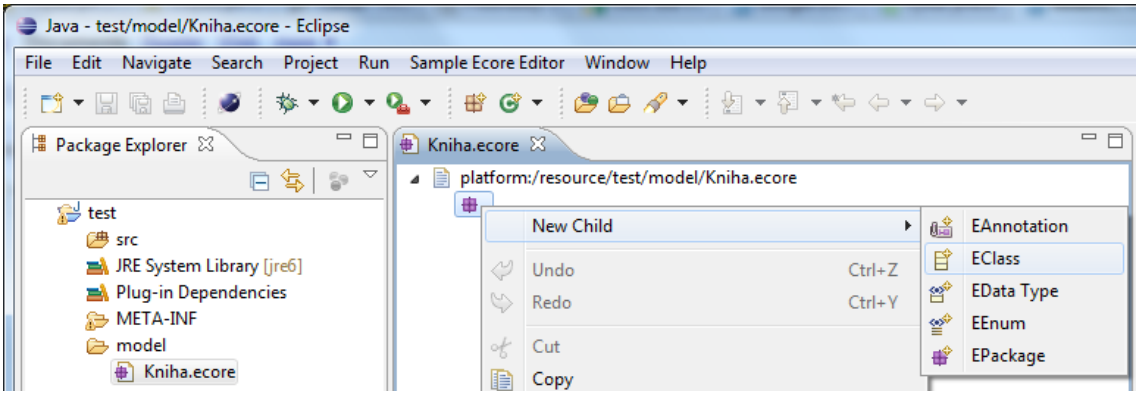
Struktura jazyka je evidentně snadněji ručně zapisovatelné, než u Ecore - zápis je blízký psaní kódu z vyšších programovacích jazyků - např. Java, podobnost s anotacemi, deklarací třídy, zápis dědičnosti, polí nelze přehlédnout. Emfatic je syntaxí blízký metamodelovacímu jazyku KM3 (**Kernel Meta Meta Model**), nejedná se však o ekvivalenty.

Podpora v Eclipse

Pro vyzkoušení vytvoříme nový projekt přes File > New > Project … > Model to Model Transformations > Operational QVT Project, po zvolení názvu projektu je projekt vytvořený.

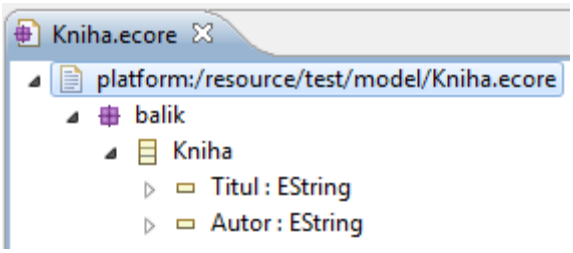
rada pro Eclipse nováčky - zavřete záložku Welcome …

Nyní vytvoříme nový Ecore model, např. na složce model v našem projektu klikneme pravým tlačítkem myši, zvolíme New > Other … > Eclipse Modeling Framework > Ecore Model. Zvolíme název (s koncovkou .ecore) - např. Kniha.ecore



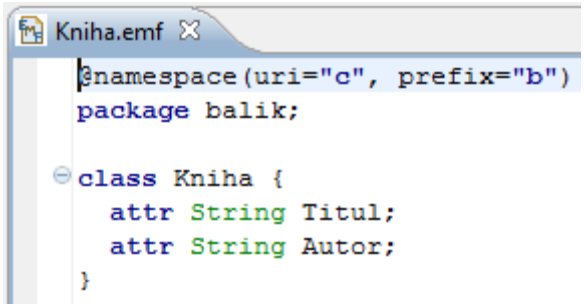
Obrázek 7 - Náhled na Ecore

V tomto vizuálním editoru můžeme snadno upravovat Ecore model. Pro demonstraci nejprve v Properties natavíme Name, Ns Prefix, Ns URI (libovolné názvy) a dále si vytvoříme EClass, v podokně Properties nastavíme Name na Kniha a na této EClass podobnými způsobem provedem New Child > EAttribute (2x). V jejich Properties nastavíme jednomu Name na Titul, druhému na Autor a oběma a EType na EString. Dostaneme tak takovýto model:



Obrázek 8 - Vyzualizované Ecore

Tento model můžem v případě potřeby převést na Emfatic - v podokně Package Explorer kliknout pravým tlačítkem myši na Kniha.ecore a vybrat **Generate Emfatic Source**:



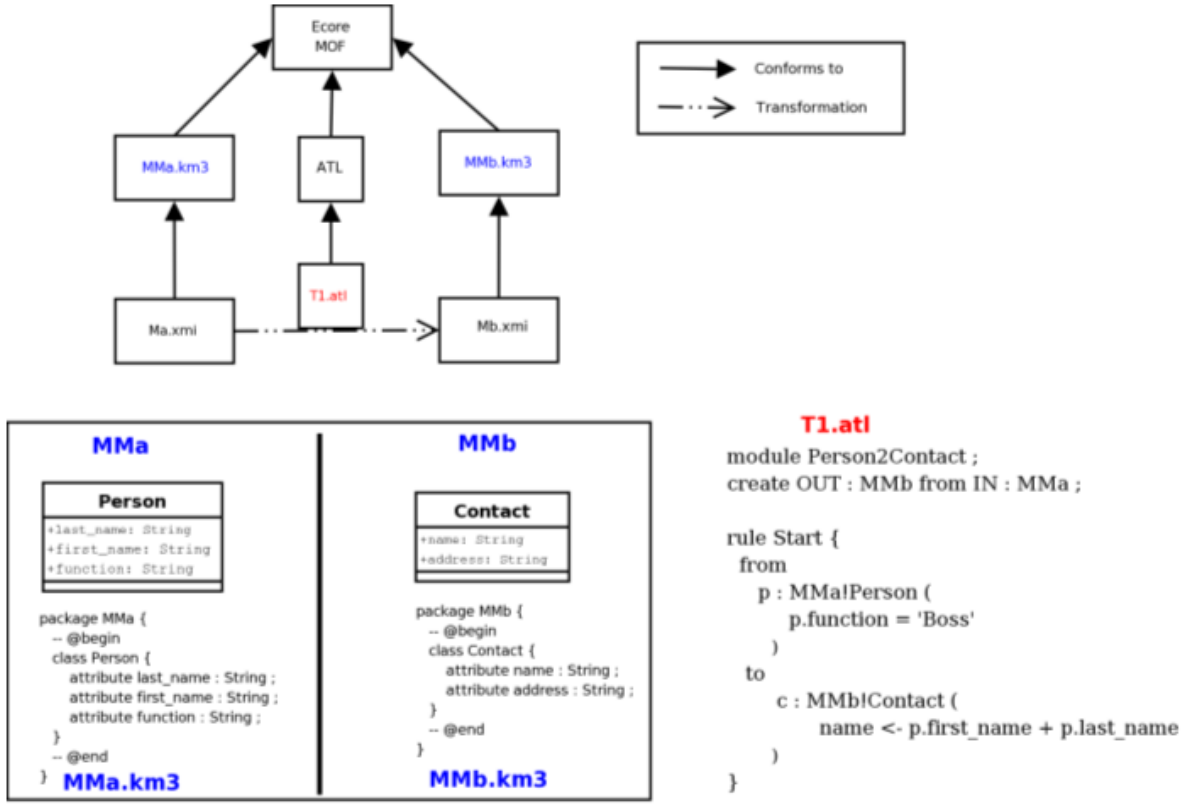
Obrázek 9 - Zápis v Emfaticu

Konverze je možná i opačným směrem, na Emfatic modelu zvolit **Generate Ecore Model**.

Instance namodelovaných dat můžeme podržet v XMI souborech. Takovýto XMI soubor snadno vytvoříme přes Ecore model - na vybrané EClass v Ecore editoru v menu vyvolaném přes pravé tlačítko zvolíme **Create Dynamic Instance...** - tímto způsobem můžeme XMI upravovat podobně jako Ecore model s tím rozdílem, že data v XMI odpovídají našemu modelu.

Transformační jazyky

Pro ty, kteří s transformačními jazyky nemají žádnou zkušenost je lze poměrně snadno přiblížit tímto schémetem:



Obrázek 10 - Znázornění transformace

Diagram zobrazuje Ecore MOF (Meta-Object Facility), v tomto případě můžeme uvažovat za výkonnou část, dále pak metamodel A (zdrojový) a metamodel B (cílový). Metamodely jsou zapsány ve formátu KM3. Transformace je zapsaná v jazyce ATLAS.

Cílem této transformace je vzít data popsané metamodelem A (osoba má atributy křestní jméno, příjmení a funkci) do formátu popsaného metamodelem B (kontakt má jméno a adresu). Má-li osoba funkci “Boss”, pak se vytvoří kontakt, kde jméno vznikne zřetězením křestního jména a příjmení.

Transformační jazyk tedy popisuje způsob převodu dat, které jsou popsány metamodely.

Existuje široká škála transformačních jazyků: ATL, Beanbag, GReAT, Kermeta, M2M, Mia-TL, MOF, MOLA, MT, QVT, SiTra, Stratego/XT, Tefkat, VIATRA. Rozděleny mohou být do 2 hlavních kategorií - imperativní a deklarativní.

Imperativní jazyky

- program v imperativním jazyku je popis algoritmu
- jednotlivé kroky programu (algoritmu) na sebe navazují, je důležité jejich pořadí
- používají cykly for, while, do-while
- používají větvení (switch, if, goto)
- používají přiřazení
- vyšší chybovost
- vyšší optimalita

Deklarativní programování

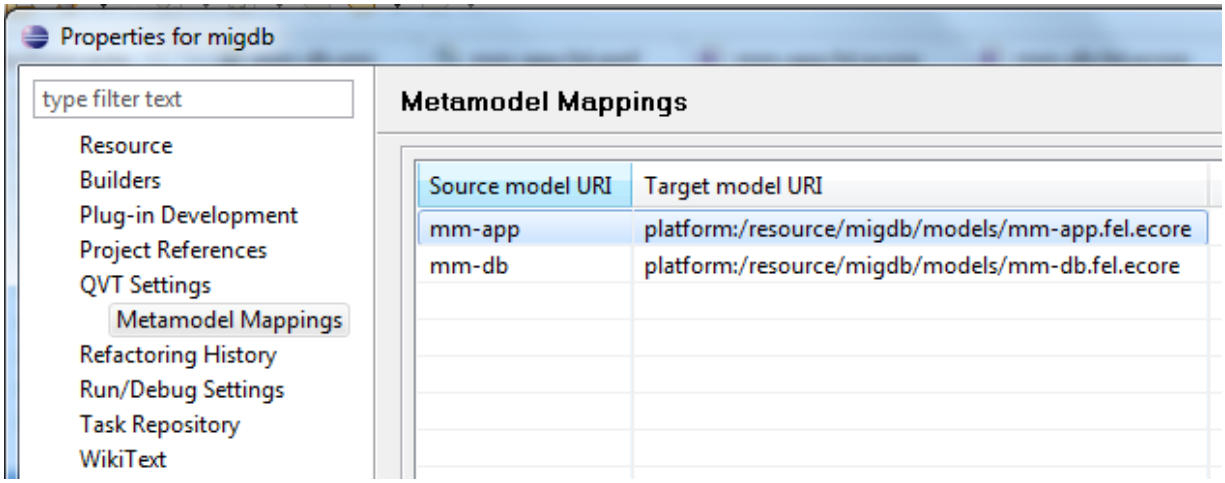
- nestará se o algoritmus, specifikuje požadovaný cíl
- závisí na interpretu jazyka, který provádí algoritmickou část
- definuje se množina funkčních závislostí nebo pravidel
- nebývá důležité pořadí jednotlívých pravidel
- střídmé využívání proměnných
- cykly řešeny rekurzí

QVT operational transformace (QVTo) v Eclipse

Na začátku QVTo transformace se nachází definice modelů:

```
modeltype APP uses 'mm-app';
modeltype RDB uses 'mm-db';
```

aby Eclipse vědělo, o které modely se jedná, je zapotřebí toto nastavit v Project > Properties a v otevřeném okně zvolit QVT Settings a Metamodel Mappings nastavit tak, aby mm-app ukazoval na Ecore soubor:



Obrázek 11 - Nastavení metamodelů

Transformace se v našem projektu spustí přes soubor run.launch > (pravoklik) > Run As > 1 run

Třídy (API) pro manipulaci s modely

Doposud popsané věci jsou užitečné do doby, kdy modelů využíváte např. k dokumentaci projektu či např. transformaci dat. V momentě, kdy potřebuje samotný model zpracovávat v aplikaci, je výhodné využít hotového aplikačního rozhraní (API).

Kód projektu Eclipse Modeling Framework je šířen pod svobodnou licencí s otevřeným zdrojovým kódem – Java

V našem projektu potřebuje pracovat s Ecore modelem, k tomu stačí využít balíčků:

- `org.eclipse.emf.common`
- `org.eclipse.emf.ecore.xmi`
- `org.eclipse.emf.ecore`

Práce nad modelem pak připomíná práci s Document Object Model (DOM) nad XML dokumentem - pomocí gettrů získávat jména tříd, atributů a datové typy. Pomocí setterů je měnit a do kolekcí přidávat nové třídy, atributy atd.

Zdroje a odkazy

http://eclipse.org/emf/
http://wiki.eclipse.org/Emfatic
http://en.wikipedia.org/wiki/ATLAS_Transformation_Language

Zhodnocení projektu

Tento projekt je poněkud odlišného rázu než většina ostatních. Projekt nebyl omezen časem pouze jednoho semestru, ale byl od začátku plánován jako dlouhodobější spolupráce mezi fakultou a zadavatelem. Z toho také plynou priority pro realizátorský tým. Naším hlavním úkolem nebylo projekt kompletně dokončit (nebylo by to v našich silách - nestačil by nám přídělený časový horizont), ale vypracovat základní analýzu, prozkoumat různé technologie, které by se daly v budoucnu použít, a hlavně předat námi nabyté zkušenosti týmu, který bude v projektu pokračovat. Myslím, že tento záměr se nám podařilo splnit. Dokázali jsme se přenést přes počáteční obtíže při pochopení hlavního záměru práce, který se týden co týden drobně, ale někdy i významně měnil, zmapovali jsme problémy, které nastávaly při interakci jednotlivých technologií, a také jsme implementovali základní funkčnosti systému.

Při zaškolování nového týmu se ukázalo, že námi vypracované dokumenty zkrátí potřebnou dobu na pochopení problému na přijatelnou dobu. To samé můžeme tvrdit o době potřebné pro přípravu vývojového prostředí. Po přečtení dokumentu rešerše byl schopen nový tým instalovat všechny potřebné rozšíření pro systém během několika hodin, kdežto nám to trvalo několik dní.

RACI matice

Jméno	Datum zahájení	Datum ukončení	Responsible
INFRASTRUKTURA	24.9.10	23.10.10	
Swinpro/Rabbit	24.9.10	25.9.10	Tomáš
Google groups + docs	24.9.10	25.9.10	Tomáš
Vytvoření adresářové struktury na SVN	24.9.10	25.9.10	Tomáš
Oprava SVN	22.10.10	23.10.10	Tomáš
Rozchození GITu	19.10.10	20.10.10	všichni
SCHŮZKY	29.9.10	16.12.10	
1. schůzka	29.9.10	30.9.10	Zdeněk
2. schůzka	5.10.10	6.10.10	Tomáš, Michal
3. schůzka	8.10.10	9.10.10	Tomáš, Michal
4. schůzka	12.10.10	13.10.10	Zdeněk, Michal
5. schůzka	19.10.10	20.10.10	Zdeněk, Michal
6. schůzka	2.11.10	3.11.10	Tomáš, Zdeněk
7. schůzka	12.11.10	13.11.10	Tomáš, Zdeněk
8. schůzka	18.11.10	19.11.10	Tomáš, Zdeněk
9. schůzka	3.12.10	4.12.10	Tomáš, Zdeněk
10. schůzka	8.12.10	9.12.10	MigDB + new gen.
11. schůzka	15.12.10	16.12.10	MigDB + new gen.
VZDĚLÁVÁNÍ	4.10.10	18.11.10	
ATL	4.10.10	5.10.10	všichni
Ecore	11.10.10	12.10.10	všichni
Operational QVT	5.11.10	6.11.10	všichni
Acceleo	2.11.10	3.11.10	všichni
Analýza JAM transformace	17.11.10	18.11.10	všichni
IMPLEMENTACE	11.10.10	3.12.10	
modifikace zadavatelovo modelu	25.10.10	26.10.10	Zdeněk
transformace app2db	22.11.10	23.11.10	Tomáš
primary key	29.11.10	30.11.10	Martin
foreign key	2.12.10	3.12.10	Martin
jUnit testy	25.10.10	26.10.10	Tomáš
Java	11.10.10	3.11.10	
modifikace Ecore	11.10.10	12.10.10	Tomáš
licence ve zdrojovém kódu	22.10.10	23.10.10	Tomáš
parsování vstupního XML	1.11.10	2.11.10	Tomáš
generování SQL	2.11.10	3.11.10	Tomáš, Martin
SI2	4.10.10	9.10.10	
Akceptační testy	8.10.10	9.10.10	Martin
Analýza rizik	4.10.10	5.10.10	Martin
SI3	5.10.10	22.12.10	
Analýza	11.10.10	2.11.10	
Use Cases	11.10.10	12.10.10	Zdeněk
Imperativní vs. Deklartivní přístup	11.10.10	12.10.10	Martin
DDL operace	18.10.10	19.10.10	Tomáš, Zdeněk, Martin
Doplnění na aktuální stav	25.10.10	26.10.10	Tomáš, Zdeněk
Definice formátu změn	1.11.10	2.11.10	Tomáš
Návrh	5.10.10	6.10.10	Zdeněk
Plán projektu	19.10.10	20.10.10	Zdeněk
Přerozdělení bodů	5.10.10	22.12.10	
1. přerozdělení bodů	5.10.10	6.10.10	Tomáš
2. přerozdělení bodů	26.10.10	27.10.10	Tomáš
3. přerozdělení bodů	16.11.10	17.11.10	Tomáš
4. přerozdělení bodů	7.12.10	8.12.10	Tomáš
5. přerozdělení bodů	21.12.10	22.12.10	Tomáš
Rešerše	25.10.10	26.10.10	Tomáš
Testy	25.10.10	26.10.10	Tomáš
Infrastruktura	16.11.10	17.11.10	Zdeněk
CVIČENÍ	22.9.10	16.12.10	
1. cvičení	22.9.10	23.9.10	všichni
2. cvičení	29.9.10	30.9.10	všichni
3. cvičení	6.10.10	7.10.10	všichni
4. cvičení	13.10.10	14.10.10	všichni
5. cvičení	20.10.10	21.10.10	všichni
6. cvičení	27.10.10	28.10.10	Tomáš, Zdeněk, Martin
7. cvičení	3.11.10	4.11.10	Tomáš, Zdeněk, Martin
8. cvičení	10.11.10	11.11.10	všichni
9. cvičení (10. týden)	24.11.10	25.11.10	Tomáš, Zdeněk, Martin
10. cvičení (11. týden)	1.12.10	2.12.10	Tomáš, Zdeněk, Martin
11. cvičení (12. týden)	8.12.10	9.12.10	Tomáš, Zdeněk, Martin, new gen.
12. cvičení (13. týden)	15.12.10	16.12.10	Tomáš, Zdeněk, Martin, new gen.

Obrázek 12 - RACI matice

Hodnocení členů týmu

Tomáš Herout

Klíčová pozitiva našeho projektu jsou především neobvyklé téma (měli jsme možnost pracovat s prostředky, se kterými bychom se pravděpodobně jinak nemuseli setkat) a práce na produktu, který je zamýšlený jako reálný pro produkční nasazení.

Velkým přinosem byl sám zadavatel, který je v dané problematice velice dobře orientovaný a během naší etapy byl větším přínosem nám, než my jemu.

Dalšími přínosy jsou možnost pracovat v týmu a činnost neorientovaná pouze na psaní kódu, ale i na celou řadu dalších souvisejících věcí.

Věříme, že naše práce bude dále rozvíjena a výsledkem předčí očekávání :-)

Martin Lukeš

Tento projekt pro mě nebyl tolik velkým přínosem jako pro ostatní členy, ale ukázal mi i některé technologie, o kterých bych se jinak ve škole nedozvěděl. Valná část projektu sestávala z komunikace s zadavatelem, kteréžto jsem se neúčastnil. Tím se pro mě projekt posunul mnohem blíže k ostatním školním projektům, v kterých se člověk učí „jen“ používat nové technologie.

Myslím si, že vzhledem k poměru času stráveného na projektu a reálnému výstupu, pro nás bude velkým zadostiučiněním až uvidíme výsledky nového týmu tzv. Next Generation.

Zdeněk Pecka

Myslím si, že tento projekt pro mě byl velikým přínosem. Nejenom, že jsem se setkal s novými technologiemi, ke kterým bych se asi jen tak nedostal, ale především si cením spolupráce s reálnou firmou na skutečném problému. Mohl jsem si na vlastní kůži vyzkoušet jednání se zákazníkem a řešení různých problémů, které ve „školním“ projektu nenastávají příliš často. I přes počáteční obtíže, kdy se téměř každý týden měnily požadavky (spíš jsme se se zákazníkem ze začátku příliš nepochopili), dokázali jsme se nakonec sejít „na stejné vlně“ a implementovali základní funkčnosti vyvíjeného systému.

Myslím si, že náš tým spolupracoval velice dobře a hlavně v důležitých chvílích jsme dokázali táhnout za jeden provaz.