VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Klientská část objektově orientovaného databázového systému

2002 Lumír Návrat

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

datum: 7.5.2002 podpis:

Abstrakt

S rozvojem informačních technologií a častějším používáním objektových technologií se dosavadní ukládání dat do klasických relačních databází stává problémem. Tato práce si klade za cíl navrhnout a implementovat uživatelské rozhraní objektové databáze JUMBO. Rozhraní by mělo umožňovat prohlížení a manipulaci s daty již vytvořených aplikací. Aplikace je naprogramována v jazyce JAVA a pro komunikaci se serverem používá protokol SOAP.

Klíčová slova JUMBO, JAVA, JIVE, objektová databáze, ODMG, SOAP, uživatelské rozhraní, klient

Abstrakt

Expansion of the information technologies and more often using object technologies present saving data into classical Relational Database state problem. This thesis behind aim propose and implement user interface for the object database JUMBO. Interface would have enable browsing and manipulation data which are already created by the application. Application was written in the language JAVA and use for the communication with a server protocol SOAP.

Keywords client, JUMBO, JAVA, JIVE, object database, ODMG, SOAP, user interface

Seznam použitých zkratek

BLOB Binary Large Object

CD Compact Disc

DTD Document Type Declaration

GMT Greenwich Mean Time

HTTP Hypertext Transfer Protocol

ODL Object Definition Language

ODMG Object Database Management Group

OID Object Identifier

OIF Object Interchange Format

OO SŘBD Objektově orientovaný SŘBD

OMG Object Management Group

OQL Object Query Language

RPC Remote Procedure Call

SMTP Simple Mail Transfer Protocol

SQL Structured Query Language

SŘBD Systém Řízení Báze Dat

URI Uniform Resource Identifier

XML Extended Markup Language

Obsah

1. Úvod

S rozvojem výpočetní techniky se stále zvyšuje potřeba uchovávání dat a jejich následného zpracovávání. K oborům, kterým do této doby stačily nynější databáze založené na relačním datovém modelu, se přidávají další obory působící v oblastech jako je např. CAD, CASE, GIS a další. Tyto systémy mají jedno společné. Potřebují uchovávat velké objemy dat. Proto začala být potřeba po novém typu SŘBD, který by byl založen na úspěchu současných systémů, ale s lepší podporou správy informací. Stručné informace o dalších databázových technologiích jako jsou digitální knihovny, historické databázové systémy je možné nalézt v [4].

Kapitola 2 se zabývá technologiemi, které se vážou k tématu této diplomové práce. Nejprve se velmi stručně pokusím nastínit rozdíl mezi SŘBD založených na záznamových modelech a SŘBD založených na objektových modelech. Následně se přesunu k popisu standardu ODMG, který je standardem pro objektové systémy. Pak bude následovat stručný popis objektové databáze JUMBO a na závěr kapitoly se zmíním o komunikačním protokolu SOAP, který jsem použil pro komunikaci mezi serverem databáze JUMBO a mým klientem.

Kapitola 3 se nejdříve zabývá popisem aktuálního stavu systému. Po tomto seznámení následuje analýza řešení Jumbo klienta. Zbytek kapitoly je věnován návrhu a popisu řešení navrženého klienta.

2. Použité technologie

2.1 Systémy řízení báze dat

Tyto systémy umožňují definování datových struktur a datových souborů. Řeší ukládání dat na médiu počítače, umožňují manipulovat s daty a formátovat vstupní a výstupní informace. Na tyto systémy se zaměřím z pohledu datových modelů, které implementují.

2.1.1 Logické datové modely

Modely založené na záznamech

Do této kategorie můžeme zařadit dva modely: síťový model a relační model. Rozdíl mezi těmito modely je ve způsobu ukládání dat. Síťový model vytváří seznamovou strukturu se záhlavím zatímco relační model uchovává data ve formě tabulek.

Identifikace entit v relačních modelech

Přirozený klíč např. jméno, rodné číslo, ...

- nelze zajistit absolutní jednoznačnost.
- nelze zajistit existenci.
- systémy není možné spojovat pokud identifikátor ztratí jednoznačnost
- identifikátor nelze jednoduše měnit.

Přidělený identifikátor člověkem nebo počítačem

- nutnost generování identifikátoru
- vytvořený klíč nemá sémantickou souvislost s entitou. Je pouze další informací v entitě.

Nevýhody záznamových modelů

předpokládá se, že data lze popsat jednoduchými typy. Většinou číselné typy a řetězce. Rovněž často chybí podpora pro strukturované typy a typ BLOB je pouze částečné řešení.

chybí jim datové struktury, které přesněji odrážejí strukturu informací v reálném světě

předpokládá se velký počet relativně malých záznamů, oproti CAD aplikacím, které mají mnoho relací s málo záznamy.

vyjadřují spíše strukturu dat, než jejich sémantiku, obtížné ukládání aplikační kódu do databáze

2.1.2 Objektově relační model

Tento model se snaží propojit vlastnosti klasického relačního modelu s objektovými principy. Tento model je implementován v SŘBD Oracle 9i. V rámci tohoto modelu je možné pracovat se strukturovanými položkami (tabulky jsou chápány jako hodnoty). Tyto modely jsou velice populární z důvodu možnosti využití stávající relační technologie, kterou pak jen rozšíří o objektové vlastnosti.

2.1.3 Objektový model

Tento model je založen na úspěchu současných objektových jazyků a technologií. Model vychází z požadavku potřeby modelování následujících bodů

informace o entitách reálného světa - velmi často není možné tuto entitu popsat jediným záznamem tvořeným čísly a řetězci

vlastnosti entit - entity nejsou pouze textové a numerické a nemusejí být ani jednotypové. Jedna vlastnost může mít více typů (např. plátce daně je jak fyzická, tak i právnická osoba)

vztahy mezi entitami - potřeba řešit modelování obecných vztahů typu M:N

operace nad entitami - entity reálného světa mají své chování, které je potřeba uchovávat s atributy

Identita objektů

Objekt je jednoznačně identifikovatelný během celé doby své existence tzv. OID. Toto OID je generované systémem a není ho možné během celé doby změnit. Rovněž není viditelné jak pro programátora tak ani pro koncového uživatele.

Problémy objektových SŘBD

I přes již relativně dlouhý vývoj objektových databází, mají tyto databázové systémy stále problémy se prosadit. To je dáno několika faktory. Firmy velmi investovaly do již existujících relačních technologií a s tím souvisí problém migrace dat mezi těmito systémy. Dalším problémem je doposud chybějící jednotná standardizace pojmu „Objektově orientovaný datový model“. Dnešní systémy založené na záznamových modelech mají, díky své dlouhé existenci, již implementovány velmi efektivní techniky, které je pro OO SŘBD nutné teprve vyvinout. Objektově orientovaný model je složitější a proto i jeho realizace je náročnější. Rovněž uživatelská rozhraní OO SŘBD zatím nedosahují kvalit současných relačních SŘBD.

2.2 Standard ODMG a jeho objektový model

2.2.1 Historie

Historie objektových databází se začíná odvíjet již v 70. letech minulého století, kdy vznikl standard CODASYL pro síťové databáze. Tento standard sice nebyl objektový ale je možné v něm nalézt některé náznaky budoucích objektových databází. První výzkumy přímo v oblasti OO SŘBD pak začínají počátkem 80. let. Na konci těchto let vznikají první komerční systémy (např. ObjectDesign, Versant, $\boldsymbol{O}_2$, Objectivity). Organizaci ODMG založil v roce 1991 Rick Cattell. O rok později vzniká standard SQL 92 pro relační databáze a začíná vývoj na standardu SQL3 pro objektové databáze. V následujícím roce je zveřejněn první standard ODMG 93. Ten byl v roce 1997 upraven na ODMG 2 a v roce 1999 byla zveřejněna zatím poslední verze tohoto standardu pod názvem ODMG 3 ([2]).

Části standardu

Object model — vychází z objektového modelu organizace OMG a definuje sémantiku, která pak může být přímo použita v OO SŘBD. Dále popisuje, jak jsou objekty identifikovány a jak mohou vstupovat do vztahů.

Object Specification Languages

Object Definition Language — používá se pro definici objektů a jejich typů v rámci objektového modelu

Object Interchange Format — používá se pro popis způsobu, jak data ukládat nebo číst ze souboru případně množiny souborů

Object Query Language — deklarativní (neprocedurální) jazyk pro vyhledávání a modifikaci objektů v OO SŘBD. Vychází z relačního standardu SQL, ale je rozšířen o další užitečné vlastnosti použitelné v objektových databázích

Language Bindings — definuje propojení objektového modelu s programovacími jazyky C++, JAVA, SmallTalk

2.2.2 Základní prvky objektového modelu

objekt má unikátní identifikátor (OID)

literál narozdíl od objektu nemá vlastní identitu a je definován jen svou hodnotou

objekty a literály se kategorizují podle svých *typů*. Typ definuje prvky se stejnými vlastnostmi a stejným chováním

stav objektu je definován hodnotami svých vlastností. tj. *atributů* a *vztahů*.

chování objektů je definováno sadou *operací*, které lze nad objektem provést. Operace mají vstupní a výstupní parametry a mohou vracet hodnotu

databáze ukládá objekty a umožňuje jejich sdílení více uživateli a aplikacemi. Je založena na schématu definovaném pomocí jazyka ODL a obsahuje instance typů definovaných schématem.

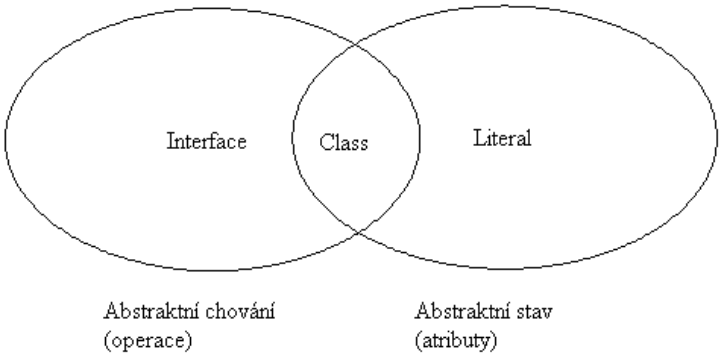
2.2.3 Specifikace a implementace typů

Specifikace typu

Specifikace typu je implementačně nezávislá a je abstraktním popisem operací, výjimek a vlastností, které bude moci vidět uživatel tohoto typu. *Interface (rozhraní)* definuje pouze abstraktní chování typu. *Literál* pro změnu definuje pouze abstraktní stav nebo hodnotu). *Class (třída)* je průnikem předchozích dvou pojmů a umožňuje definovat jak abstraktní chování, tak i stav (obr. 2.1).

Implementace typu

Implementace typu se skládá z *reprezentace* a množiny *metod*, přičemž není viditelná pro uživatele. Reprezentace je datová struktura, která se definuje na základě mapování do jazyka. Metody jsou pak těla operací definovaných ve specifikaci typu. Tyto metody definují veřejně viditelné



Obrázek 2.1: Specifikace typu

chování objektového typu. Rovněž mohou existovat metody, které nejsou přímou součástí specifikace typu. Typ může mít více než jednu implementaci, ale může být použita pouze jedna v daném programu (např. jedna pro C++, druhá pro JAVu atd.)

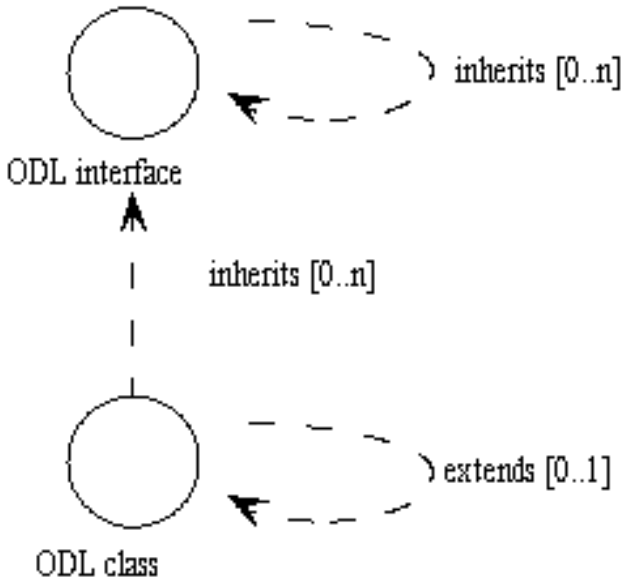
2.2.4 Podtypy a dědičnost chování

Stejně jako v jiných objektových modelech, tak i ODMG objektový model obsahuje vazbu mezi typem a podtypem založenou na dědičnosti. Tomuto vztahu se někdy říká *is-a* vztah. Pro tento vztah bylo definováno, že instance podtypu je zároveň instancí nadtypu a dále každý objekt má právě jeden nejvíce specifický typ, který popisuje všechny atributy a operace objektu. Model povoluje vícenásobnou dědičnost pouze pro chování objektu. Jak je vidět na obrázku 2.2 tato vícenásobná dědičnost není možná mezi třídami. ODL třídy jsou v cílovém implementačním jazyku mapovány na třídy, které mohou být přímo instanciované, zatímco interface je typ, který přímo instanciovat nelze.

Dalším vztahem ke vztahu is-a je vztah *extends*. Tento vztah definuje dědičnost mezi jednotlivými objektovými typy. Tento vztah povoluje pouze jednoduchou dědičnost mezi dvěmi třídami, kdy děděná třída dědí všechny vlastnosti a chování ze své nadtřídy.

2.2.5 Extenty

Extent typu je množina všech instancí daného typu v OO SŘBD. Jestliže objekt je instance typu ***A*** potom bude členem extentu typu ***A***. Pokud bude typ ***A*** podtypem typu ***B***, pak extent typu ***A*** bude podmnožinou extentu typu ***B***.



Obrázek 2.2: Vztah třída-interface

SŘBD může tento extent udržovat automaticky, tj. může přidávat do množiny nově vytvořené instance nebo z této množiny tyto instance odebírat, pokud byly smazány. Rovněž je možné vytvářet a spravovat indexy pro zrychlení přístupu k extentu.

2.2.6 Objekty

Vytváření objektů

Pro vytváření objektů se využívá návrhový vzor „Abstract Factory“, který dává možnost metodou `new()` implementovat v daném cílovém implementačním jazyku kód starající se o vytvoření objektu.

```
interface ObjectFactory {
    Object  new();
};
```

Z následujícího rozhraní dědí všechny objekty v modelu a je tak implicitním rozhraním pro všechny objekty.

```
interface Object {
    enum Lock_Type(read,write,upgrade);
    void lock(in Lock_Type mode) raises(LockNotGranted);
    boolean try_lock(Lock_Type mode);
    boolean same_as(in Object anObject);
    Object copy();
    void delete();
};
```

Identifikátory objektů (OID)

Všechny objekty mají svůj identifikátor, kterým jsou od sebe vždy vzájemně rozlišitelné. Abychom mohli porovnat tyto objekty slouží k tomu metoda `same_as()` výše uvedeného rozhraní `Object`. Identifikátor objektu se během celé životnosti objektu nemění. Tato podmínka platí i pokud se úplně změní obsah jeho atributů a vztahů. Identifikátor se dále používá pro odkazy mezi objekty. Generování identifikátoru je zajišťováno prostředky SŘBD a aplikace jej může jen využívat. Zde je vidět zásadní rozdíl mezi primárním klíčem v relačních SŘBD a OID objektu.

Jména objektů

Objektům lze přiřadit několik jmen, které jsou aliasy pro identifikátory objektů. Tyto aliasy jsou vstupními body pro navigaci v databázi a jsou unikátní v rámci celé databáze. Aliasy lze chápat jako globální jméno proměnné v programovacích jazycích. Nejsou definovány v rozhraní typu a nemají ani žádný vztah k hodnotám atributů objektu.

Doba života objektů

Definuje dobu, po kterou je objektům přidělena paměť. Objekty můžeme rozdělit do dvou skupin.

Tranzientí objekty – existují pouze po dobu činnosti aplikace. Např. lokální proměnné v metodách nebo globální proměnné procesu. Přidělování paměti řídí run-time programovacího jazyka.

Perzistentní objekty – existují i po ukončení procesu, který jej vytvořil. Přidělování paměti řídí run-time systému SŘBD.

Doba života objektu není závislá na typu objektu a tak je možné, aby jeden typ měl současně jak tranzientní tak i perzistentní instance. Tato doba pak může být definovaná již při vytvoření objektu nebo dosažitelností objektu.

Druhy objektů

atomické objekty jsou definované uživatelem a standard nedefinuje žádné zabudované atomické objekty

kolekce obsahují instance atomického objektu, jiné kolekce nebo typu literálu. Všechny prvky kolekce musí být téhož typu.

- `Set<t>` – neuspořádaná množina prvků, které se nesmí v kolekci opakovat
- `Bag<t>` – neuspořádaná množina prvků, které se mohou v kolekci opakovat
- `List<t>` – uspořádaná množina prvků, které se mohou v kolekci opakovat
- `Array<t>` – uspořádaná množina prvků, která mění dynamicky velikost. K jednotlivým prvkům kolekce je možné přistupovat na základě znalosti jejich umístění v kolekci
- `Dictionary<t>` – neuspořádaná množina páru klíč-hodnota, přičemž se nesmí klíče opakovat

strukturované typy

- `Date` – datum
- `Interval` – časový interval, který je možné používat s objekty `Time` a `Timestamp`. Interval se vytváří metodou `subtract_time` definovanou v rozhraní `Time`.
- `Time` – reprezentuje světový čas v dané časové zóně. Interně se čas ukládá v časové zóně GMT.
- `Timestamp` – zapouzdřuje objekty `Date` a `Time`

2.2.7 Atributy

Hodnotou atributu je vždy literál, nebo OID. Atribut je abstraktním stavem instance a nemusí tak vždy odpovídat datové struktuře (např. atribut `age` v následujícím příkladu může být implementován metodami, které zpracují atribut `birthdate`). Jelikož atribut nemá vlastní identitu nemohou atributy mezi sebou definovat vztahy.

```
interface i_Person {
    attribute short age;
}
```

```
class Person: i_Person {
   attribute Date birthdate;
   attribute string name;
   attribute enum gender {male, female};
   attribute Address home_adress;
   attribute set<Phone_no> phones;
   attribute Department dept;
}
```

2.2.8 Vztahy

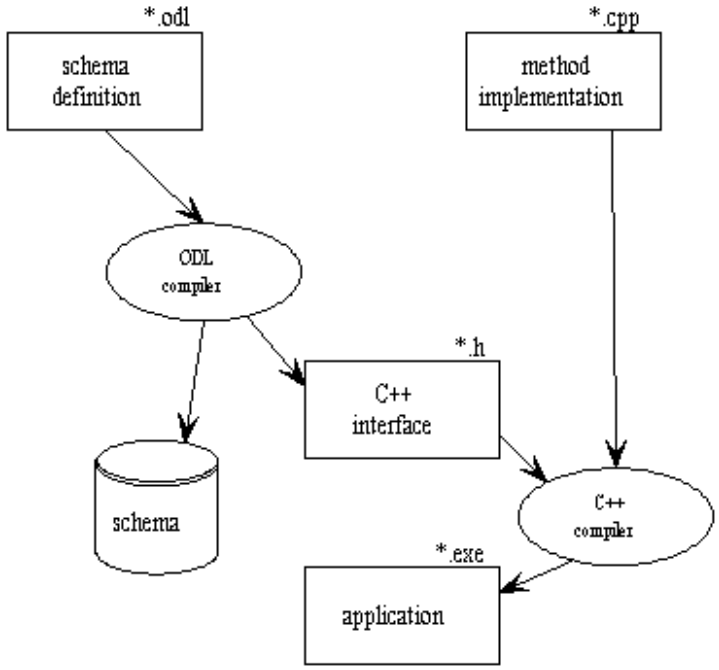
Vztahy se definují pouze mezi typy a mohou být pouze binární tj. 1:1, 1:N a M:N. Tyto vztahy jsou vždy obousměrné a SŘBD je zodpovědný za dodržování referenční integrity těchto vztahů. To znamená v našem příkladě, že v případě smazání instance typu `Professor` musí dojít ke smazání všech referencí ve všech instancích typu `Course`. Atributy typu Object, nebo kolekce objektů tak nezajišťují referenční integritu. Následující příklad ukazuje způsob zápisu vztahů. Je uveden zápis pro vztah 1:N, kdy profesor učí několik kurzů a kurz je učen pouze jedním učitelem. Klíčové slovo *inverse* identifikuje název vztahu na druhé straně vazby.

```
class Professor {
    ...
    relationship set<Course> teaches
        inverse Course::is_taught_by;
    ...
};

class Course {
    ...
    relationship Profesor is_taught_by
        inverse Professor::teaches;
    ...
};
```

2.2.9 Operace

Vedle atributů a vztahů, které definují stav objektu je potřeba definovat chování objektu. Toto chování je specifikováno množinou *signatur operací*. Tato signatura definuje jméno operace, jména a typy argumentů a jména generovaných výjimek. Operace je možné definovat pouze pro jediný typ, ale je možné operace přetěžovat, takže může existovat v jiném typu operace stejného jména. Při volání operace daného jména se provádí výběr na základě nejvíce specifického typu.



Obrázek 2.3: Vytváření aplikace dle standardu ODMG3.0

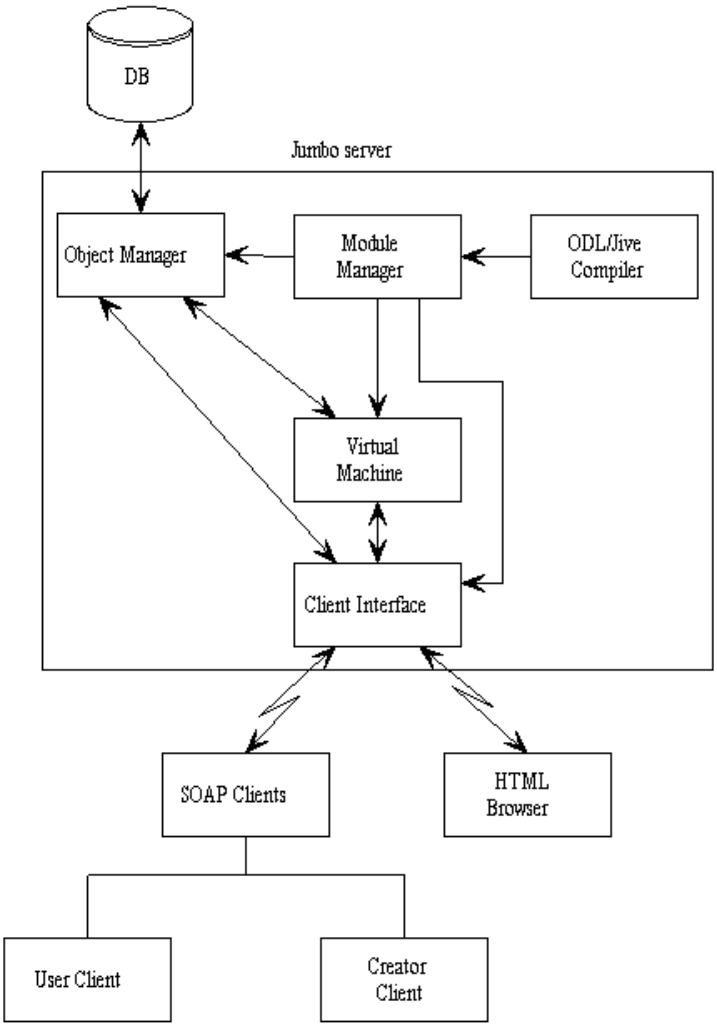
2.3 Databázový systém JUMBO

Jedná se o experimentální objektově orientovaný databázový server, který vychází ze standardu ODMG, přičemž standard rozšiřuje o podporu rolí a procesů. Systém vznikl na výsledcích výzkumného projektu „Vícetypové objekty v objektovém datovém modelu.“ Tento projekt probíhal v rozmezí let 1998 a 1999 na Vysokém učení technickém v Brně.

Struktura systému je znázorněna na obrázku 2.4. Objekty v systému jsou specifikovány prostřednictvím rozhraní v jazyce ODL. Samotná implementace operací je zajišťována jazykem JIVE, který vychází z jazyka JAVA, který rozšiřuje o programové konstrukce pro transparentní manipulaci s tranzientními a perzistentními objekty.

Modul je základním prvkem aplikace. Přeložením jeho definiční částí napsané v jazyce ODL a příslušné implementace v jazyce Jive dostaneme binární modul. Tento binární modul můžeme pak v případě požadavku nahrát do paměti databázového serveru a spouštět přeložené operace ve virtuálním stroji jazyka Jive (JiveVM).

Databázový server umožňuje komunikovat s klienty pomocí dvou protokolů. Implementačně starším je protokol HTTP, který se používá pouze pro experimentování a testování serveru.



Obrázek 2.4: Struktura databázového serveru

V rámci této diplomové práce vznikl uživatelský klient a Petr Šula ve své diplomové práci pracoval na programátorském klientovi, který umožňuje navrhovat schémata modulů v jazyce ODL. Oba klienti komunikují se serverem protokolem SOAP.

Systém po spuštění nejprve inicializuje překladač a databázové jádro. Následně je vytvořen popis metadat (modul ODLMetaObjects) ten je vložen do systému. Takto je možné reprezentovat databázové schéma, včetně schématu metadat popisem sama sebe. Největší výhodou tohoto postupu je možnost implementace mnohem speciálnějších operací databázového schématu přímo v jazyce Jive bez nutnosti cokoli programovat v hostujícím jazyce systému C++. Sa-

mozřejmě je možné implementovat operace v nativním jazyce (např. z důvodu požadavku efektivity nízkoúrovňových operací).

Role

Jak již bylo uvedeno, Jumbo používá k popisu rozhraní objektů jazyk ODL. Tento jazyk dále rozšiřuje o koncept rolí. Způsob zápisu je uveden v následujícím příkladu. Role jsou v run-timové hierarchii reprezentovány hlavním objektem, který tvoří kořen stromu objektů. Role fungují v podstatě jako rozšíření stromu dědičnosti, vytvořeného během překladu, během běhu systému. Role dědí operace a vlastnosti svého základního objektu. Proto pro každý přístup k tomuto objektu skrz roli je potřeba udržovat skrytou referenci z role na tento objekt. Komunikace se samotným objektem může pak probíhat nepřímo přes jakoukoliv roli, která reprezentuje objekt. Adresovaná role buď zpracuje požadavek sama, nebo případně jej předá rodičovskému objektu.

Role jsou tak schopné nahradit většinu vlastností vícenásobné dědičnosti, která není podporována v standardu ODMG v3.0. Navíc je možné role přidávat a odebírat dynamicky během životního cyklu objektu a tak lze velmi jednoduše modelovat životní cyklus objektů v reálném světě.

```
class Student roleof Osoba (extent studenti) {
    relationship Skola     skola inverse studenti;
    attribute    long      rocnik;
    attribute    TypStudia studium;
};
```

Dalším rozšířením jsou *kvalifikované role*. Ty vycházejí z požadavku modelování situací kdy jeden objekt hraje tutéž roli vícekrát (např. osoba studuje více škol). V takovém případě je nutné jednoznačně určit o kterou roli se jedná. Toto rozšíření zatím není v jazyce Jive implementováno.

Jazyk Jive

Standard ODMG nedefinuje vlastní jazyk pro manipulaci s daty, ale jen specifikuje mapování do jazyků C++, Java a Smalltalk. Typické schéma implementace aplikace je pak znázorněno na obrázku 2.3. I když tento přístup je vcelku přijatelný, nenaplňoval plně principy ortogonality perzistentních jazyků [3]. Z tohoto důvodu byl zvolen zcela odlišný přístup a byl vytvořen speciální manipulační jazyk Jive, který tyto principy splňuje v souladu se standardem ODMG. Jazyk Jive je tedy ortogonálním rozšířením jazyka Java s následujícími přidanými vlastnostmi.

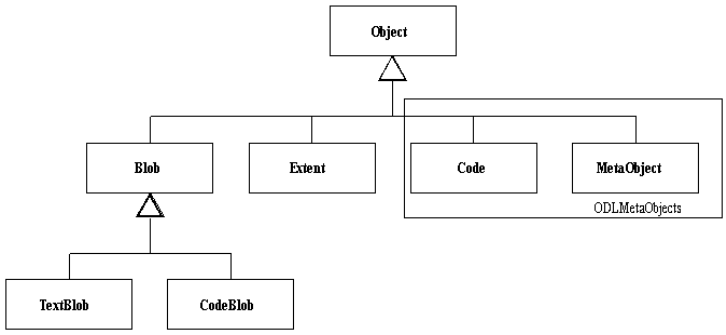
- Jive je pevně svázaný se standardní definicí jazyka ODL, takže není zde žádný problém s impledancí typu [3]. Jive podporuje všechny typy a konstrukce jazyka ODL včetně vztahů mezi objekty.
- Jive obsahuje podporu pro ODMG 3.0 OQL, což zajišťuje plnou kontrolu dotazů již během překladu.
- Virtuální stroj JiveVM přímo podporuje instrukce orientované na specifické databázové konstrukce (např. dotazy, datové operátory).

2.3.1 Struktura objektů

Systémové a metaobjekty databáze JUMBO jsou tvořeny dvěmi moduly. Modulem `ODL` a modulem `ODLMetaObjects`. Jak již bylo uvedeno tyto moduly jsou vytvářeny okamžitě po inicializaci systému a díky tomu je možné k nim přistupovat stejně jako k jiným objektům v databázi. Systém zatím plně neimplementuje standard ODMG a to hlavně v oblasti transakcí, řízení společného přístupu a jmenného prostoru typů. Poslední vlastnost se projevuje tím, že veškeré objekty a typy jsou definované a viditelné v rámci celého modulu.

Modul ODL

Tento modul obsahuje typy `Object`, `Blob`, `Extent`, `CodeBlob` a `TextBlob`. Jednotlivé závislosti tříd je možné vidět na obrázku 2.5.



Obrázek 2.5: Schéma tříd v modulu ODL

Object typ který implicitně implementují všechny systémové i uživatelské typy.

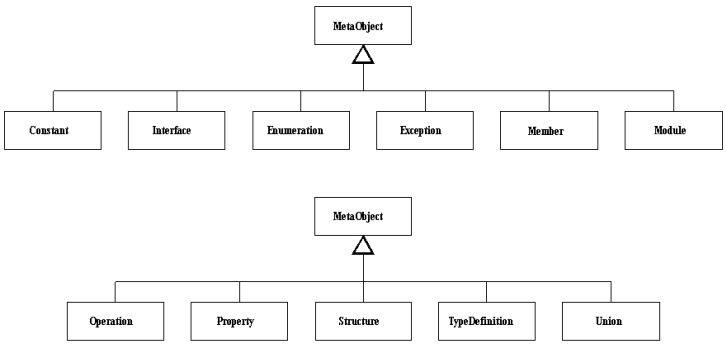
Blob, CodeBlob, TextBlob typy slouží k uložení binárních dat do databáze.

Extent typ pro ukládání extentů. (viz 2.2.5)

Modul ODLMetaObjects

Tento druhý systémový modul definuje metaobjekty standardu ODMG a vytváří tak databázové schéma systému.

MetaObject Obecný metaobjekt, který poskytuje společné vlastnosti všem metaobjektům. Jedná se o atributy `name` a `comment`.

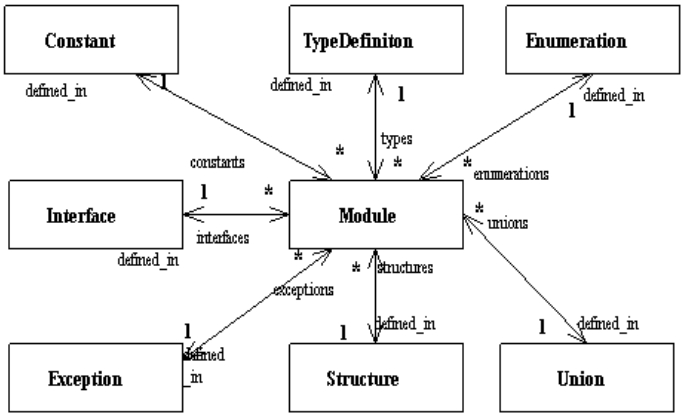


Obrázek 2.6: Schéma děděných tříd z MetaObject

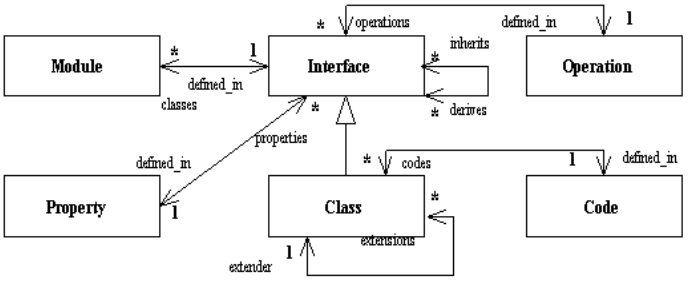
Module výchozí bod pro popis metaschématu dané aplikace. Vše co popisuje data je viditelné pouze v rámci daného modulu. Modul má atribut `sourceFile`, který obsahuje jméno souboru s ODL popisem.

Interface jeden z nejdůležitějších typů. Definuje abstraktní chování aplikačních objektů

Class rozšiřuje `Interface` o abstraktní stav objektů. Jedním z atributů je `is_role`, který určuje, že instance třídy je rolí. Dalším atributem je `extents`, který obsahuje referenci na extent třídy. Atribut `source` pak obsahuje referenci na Blob objekt obsahující zdrojový kód třídy.



Obrázek 2.7: Schéma vazeb typu Module



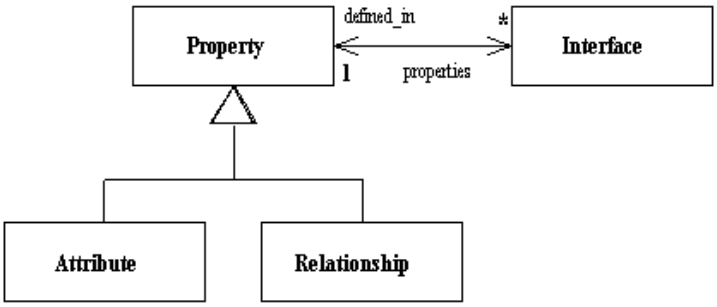
Obrázek 2.8: Schéma vazeb typů Interface a Class

Property abstraktní typ, který zastřešuje typy `Attribute` a `Relationship`. Obsahuje atribut `type`, který definuje typ vlastnosti.

Attribute vlastnost, která uchovává informace o stavu. Má atribut `read_only`, který určuje, že neexistuje možnost změnit hodnotu atributu.

Relationship vlastnost, která realizuje obousměrnou vazbu mezi typy. Má atribut `traversal`, který obsahuje jméno vazby u druhého objektu.

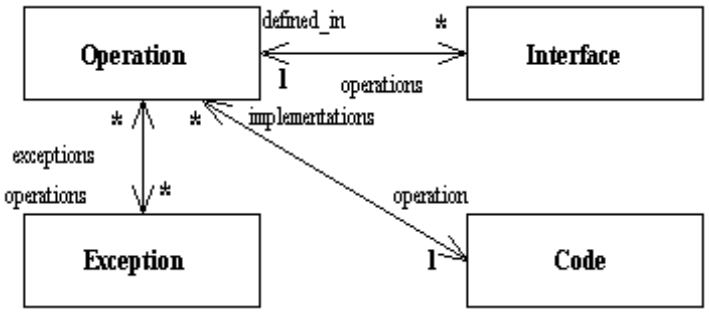
Operation typ modeluje podporu chování aplikačních objektů.



Obrázek 2.9: Schéma vazeb typů Property,Attribute,Relationship

Exception typ popisuje výjimky, které mohou operace volat.

Code typ popisuje binární kód operací. Vlastní kód v jazyce Jive je v atributu `code`. Další atributy `stack_size`, `var_size` a `is_native` blíže určují vlastnosti operace v JiveVM.



Obrázek 2.10: Schéma vazeb mezi typy Operation, Exception a Code

Constant zajišťuje popis konstanty v modulu pomocí atributů `the_value` a `type`.

TypeDefinition tento typ má zajistit podporu aliasu daného typu.

Enumeration je strukturovaný typ, který realizuje výčtový typ. Prvky výčtu jsou uložené v atributu `consts`. Prvkem mohou být jen literály typu `string`.

Strukturované typy jedná se o typy `Union`, `Structure` a s tím související typ `Member`.

Ukázka definice modulu pro systém JUMBO

```
module Pokus {

  class Kontakt (extent kontakty) {
      attribute  string adresa;
  };

  class Osoba extends Kontakt (extent osoby) {
      attribute  string jmeno;
      attribute  string adresa;
      attribute  char   znak;
      string kdo_jsi();
      Student do_skoly(in Skola skola);
      Zamestnanec do_prace(in Firma firma, in long plat);
  };

  class Firma extends Kontakt (extent firmy) {
      attribute    string           nazev;
      attribute    string           adresa;
      attribute    unsigned long    ico;
      attribute    set<Kontakt>     kontakty;
      relationship set<Zamestnanec> zamestnanci inverse firma;
      void pridej_kontakt(in Kontakt x);
  };

  class Skola extends Firma (extent skoly) {
      relationship set<Student> studenti inverse skola;
  };

  class Zamestnanec roleof Osoba (extent zamestnanci) {
      relationship Firma firma inverse zamestnanci;
      attribute    long  plat;
  };

  enum TypStudia {
      bc,mgr,dr
  };
```

```
  class Student roleof Osoba (extent studenti) {
      relationship Skola     skola inverse studenti;
      attribute    long      rocnik;
      attribute    TypStudia studium;
  };

} // module pokus
```

Ukázka implementace funkcí předchozího modulu

```
implementation module Pokus {

  class Firma {

    void pridej_kontakt(in Kontakt x) {
      kontakty.insert_element(x);
    } // pridej_kontakt
  }; // Firma

  class Osoba {

    string kdo_jsi() {
      return "Osoba: " + jmeno;
    } // kdo_jsi

    Student do_skoly(in Skola skola) {
      var Student student = newRole Student;
      student.skola = skola;
      student.rocnik = 1;
      return student;
    } // do_skoly

    Zamestnanec do_prace(in Firma firma, in long plat) {
      var Zamestnanec zam = newRole Zamestnanec;
      zam.firma = firma;
      zam.plat = plat;
      return zam;
    } // do_prace
  }; // Osoba
```

```
  class Zamestnanec {

    string kdo_jsi() {
      return "Zamestnanec: " + jmeno + " - " + firma.nazev;
    } // kdo_jsi
  }; // Zamestnanec

  class Student {

    string kdo_jsi() {
      return "Student: " + jmeno + " - "
             + skola.nazev + ", rocnik " + rocnik;
    } // kdo_jsi
  }; // Student
} // Pokus
```

2.4 Protokol SOAP (Simple Object Access Protocol)

2.4.1 Úvod

SOAP je protokol poskytující jednoduchý mechanismus pro výměnu informací v decentralizovaném, distribuovaném prostředí. Protokol používá XML a sám o sobě nedefinuje žádnou aplikační sémantiku. Poskytuje jen modulární model pro zapouzdření dat a mechanismus pro kódování těchto dat v rámci modulu. To umožňuje protokolu SOAP být používán v různých systémech od systému pro zasílání zpráv až po RPC. Protokol může být rovněž použitý v kombinaci s jinými protokoly (např. HTTP, SMTP). Tento protokol byl standardizován a je udržován organizací W3C. Samotný vznik protokolu byl podmíněn potřebou jednoduchého protokolu, který by byl nezávislý na platformě jak systémové, tak i programové. Na samotném vzniku protokolu se tak podílely firmy jako je IBM či Microsoft. V dnešní době s rozvojem internetu a webových služeb se tento protokol stává populárnějším a k tomu přispívá i množství implementací pod různými programovacími jazyky.

Protokol SOAP se skládá z následujících částí:

SOAP Envelope – (viz 2.4.3) definuje aplikační rámec vyjadřující co je obsahem zprávy, kdo by ji měl zpracovat a zda je obsah povinný nebo volitelný

SOAP Encoding rules – definují pravidla pro serializaci[1] aplikačních datových typů a umožnit tak jejich přenos.

SOAP RPC – definuje pravidla, které lze použít během vzdáleného volání funkcí a získávání odpovědí na ně.

Protokol byl navržen jako jednoduchý a rozšiřitelný. To má za následek chybějící podporu některých vlastností tradičních systémů pro předávání zpráv a distribuovaných systémů.

Následující dva příklady ukazují, jak může vypadat požadavek a odpověd v tomto protokolu.

SOAP požadavek v protokolu HTTP

```
POST /StockQuote HTTP/1.1
Host:
www.stockquoteserver.com
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP odpověd na požadavek v protokolu HTTP

```
HTTP/1.1 200 OK
Content-Type: text/xml;
charset="utf-8"
Content-Length: nnnn
```

[1] Serializace je mechanismus převodu instancí objektů do binární podoby vhodné pro přenos.

```
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <m:GetLastTradePriceResponse xmlns:m="Some-URI">
      <Price>34.5</Price>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.4.2 Model výměny zpráv

Zprávy protokolu SOAP jsou v podstatě jednosměrné ve směru od odesílatele k příjemci. Často jsou ale implementovány podle návrhového vzoru Požadavek/Odpověd. Samotné implementace tohoto protokolu mohou být optimalizovány pro konkrétní charakteristiky komunikačních sítí. Tedy např. pokud zprávu posíláme v rámci požadavku protokolu HTTP, odpověd přijde stejným protokolem s použitím stejného spojení. Nehledě na protokol, kterým jsou tyto zprávy přenášeny, je možné zprávy přeposílat z jednoho příjemce k druhému s postupným zpracováním až ke konečnému příjemci.

Aplikace, která přijímá zprávy protokolu musí provést následující operace v pořadí v jakém jsou napsány.

1. Identifikace všech částí SOAP zprávy, určených pro aplikaci
2. Ověření všech povinných částí předchozího bodu, které jsou podporovány aplikací pro tuto zprávu a podle toho ji zpracovat. Jestliže tomu tak není, pak tuto zprávu ignorovat. Je možné ignorovat nepovinné části kroku 1 bez ovlivnění výsledku zpracování
3. Pokud aplikace není posledním příjemcem zprávy, odstraní před přeposláním všechny části identifikované v kroku 1.

Jak už jsem se zmínil protokol SOAP je aplikací XML. Z tohoto důvodu by aplikace tohoto protokolu měla obsahovat příslušné jmenné prostory všech elementů a atributů ve zprávách, které generuje. Rovněž aplikace musí být schopná tyto jmenné prostory zpracovat v přijatých zprávách. Pokud jmenný prostor není korektní musí aplikace tyto zprávy zrušit. Zprávy bez jmenného prostoru je možné chápat jako zprávy s korektním jmenným prostorem tohoto protokolu.

Protokol definuje následující dva jmenné prostory:

SOAP Envelope – „http://schemas.xmlsoap.org/soap/envelope/“

SOAP Serialization – „http://schemas.xmlsoap.org/soap/encoding/“

Zprávy rovněž nesmí obsahovat DTD a procesní instrukce protokolu XML.

2.4.3 Envelope

SOAP zprávy se skládají z povinné části *SOAP Envelope*, volitelné *SOAP Header* a povinné části *SOAP Body*. Envelope je nejvyšší element dokumentu reprezentující zprávu. Header je všeobecný mechanismus pro přidávání vlastností do zprávy v decentralizovaném chování bez předchozí domluvy komunikačních stran. SOAP definuje několik málo atributů, které lze použít k indikaci kdo má s danou vlastností pracovat a zda je volitelná nebo povinná. Body je pak kontejner pro povinné informace určené koncovému příjemci zprávy. Standard definuje jeden element pro Body a to element *Fault*, který se používá pro předávání chyb.

Pravidla pro vytváření zprávy

1. Envelope

   jméno elementu je „Envelope“

   element musí být součástí zprávy

   element může obsahovat deklaraci jmenných prostorů právě tak jako další atributy nebo podelementy. Pokud jsou tyto atributy nebo elementy přítomné, pak musí být specifikované jménem jmenného prostoru a musí být umístěné ze elementem Body

2. Header

   jméno elementu je „Header“

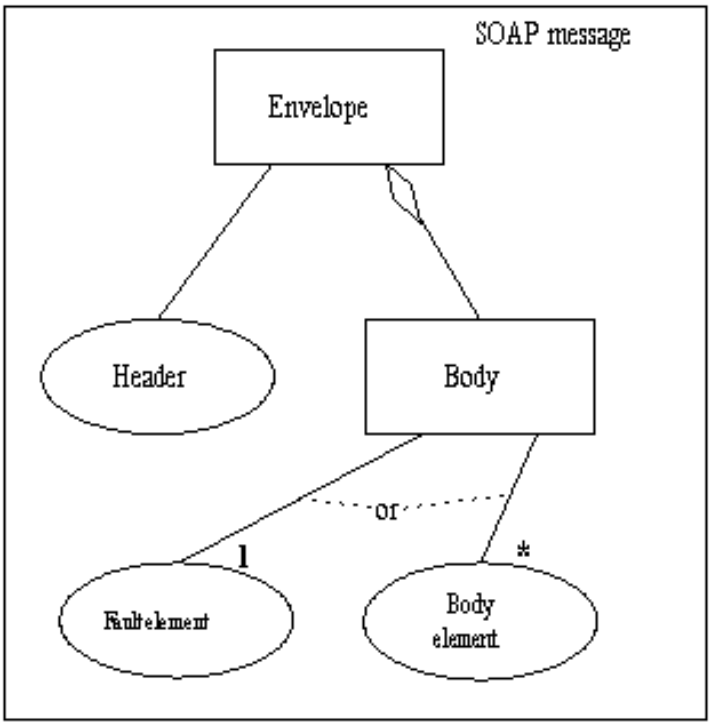
   element může být součástí zprávy. Pokud je součástí, pak musí být prvním podelementem v elementu Envelope.

   element může obsahovat množinu podelementů. Všechny tyto podelementy musí být rovněž specifikované jménem jmenného prostoru.

3. Body

   jméno elementu je „Body“

   element musí být součástí zprávy a musí být buď prvním podelementem elementu Envelope, nebo musí následovat bezprostředně za podelementem Header, pokud je přítomen.



Obrázek 2.11: Struktura zprávy protokolu SOAP

Atribut encodingStyle

Tento globální atribut je možné použít k indikaci serializačních pravidel použitých ve zprávě. Pokud je v elementu tento atribut uveden, pak se vztahuje jen na obsah elementu a všechny jeho podelementy. Standard nedefinuje implicitní pravidlo. Hodnotou atributu je seznam jednoho nebo více URI definujících serializační pravidla. Pravidla se uvádějí v pořadí od nejvíce specifického po nejméně.

Standard definuje vlastní serializační pravidlo, které je definované v 5. kapitole standardu [1] a jeho URI je: „http://schemas.xmlsoap.org/soap/encoding“. URI, které začínají tímto řetězcem pak indikují kompatibilitu s tímto pravidlem a poskytují jeho zpřesnění. Pokud bude hodnota atributu `""`, pak nebudou žádné požadavky na elementy.

Podpora verzí

Tento protokol nedefinuje tradiční model pro správu verzí založených na hlavních a vedlejších číslech verzí. Zprávy místo toho musí obsahovat element, který je asociovaný se jmenným pros-

torem definovaným tímto standardem („http://schemas.xmlsoap.org/soap/envelope“). Pokud aplikace přijme Envelope element s jiným jmenným prostorem, musí tuto zprávu zahodit a vrátit chybu verzí.

2.4.4 Header

Mechanismus hlaviček umožňuje flexibilně rozšiřovat zprávu o informace bez nutnosti předchozí znalosti komunikačního partnera. Typickým příkladem pro použití hlaviček je autentifikace a řízení transakcí.

Header element je ve zprávě umístěn jako první podelement v elementu Envelope. Pravidla pro položky hlavičky jsou následující:

1. položka hlavičky je identifikována plným jménem elementu, který se skládá z URI jmenného prostoru a lokálního jména. Všechny podelementy pak musí být rozlišitelné tímto jmenným prostorem.
2. atribut `encodingStyle` může být použit pro indikaci pravidel pro serializaci na položku hlavičky
3. atributy `mustUnderstand` a `actor` mohou být použité pro indikaci způsobu zpracování položky a kým má být zpracována.

Atributy hlavičky určují, jak se má příjemce zprávy zachovat. Aplikace, která posílá zprávu by měla používat atributy následujícího podelementu elementu hlavičky. Následně příjemce zprávy musí ignorovat všechny atributy, které nejsou k tomuto podlementu vztaženy.

Ukázka hlavičky

```
<SOAP-ENV:Header>
  <t:Transaction
    xlmns:t=,,some-URI‘‘ SOAP-ENV:mustUnderstand=,,1‘‘>
      5
  </t:Transaction>
</SOAP-ENV:Header>
```

Atribut actor

Během putování zprávy od vysílatele k cílovému příjemci může zpráva procházet různými mezi-aplikacemi, přičemž tyto aplikace jsou zodpovědné za přeposlaní dalšímu příjemci. Soap zpráva tak může obsahovat části, které nejsou určené jen cílovému příjemci, ale mohou být jen pro některé aplikace na její cestě.

Tento atribut slouží k identifikaci příjemce zprávy. Identifikace je opět založena na URI řetězci, a každý příjemce musí nastavit novou hlavičku před odesláním zprávy. K identifikaci prvního příjemce slouží řetězec „http://schemas.xmlsoap.org/soap/actor/next“. Jestliže tento atribut chybí, potom je příjemcem rovnou cílová aplikace.

Atribut mustUnderstand

Tento atribut indikuje, zda položka hlavičky je povinná nebo volitelná z hlediska zpracování příjemcem. Příjemce je definován atributem `actor`. Atribut může nabývat hodnot „1“ nebo „0“. Nepřítomnost atributu je chápána jako hodnota „0“.

Pokud je hodnota atributu „1“, pak se musí příjemce položky buď podřídit sémantice a zpracovat zprávu korektně dle dané sémantiky nebo vyvolat chybu zpracování. Tím zajišťujeme robustnost změn sémantiky, jelikož nejsou ignorovány nesprávně rozpoznané zprávy.

2.4.5 Body

Tento element slouží k předávání povinných informací cílovému příjemci zprávy. Nejčastějším použitím elementu je vkládání RPC volaných metod a předávání chyb. Element musí být buď prvním podelementem elementu Envelope, nebo pokud element Envelope obsahuje element Header, pak musí následovat hned za tímto podlementem. Všechny položky elementu Body jsou chápány jako nezávislé. Standard definuje jednu položku Fault 2.4.6,která je určena pro přenos chyb.

Pravidla pro položky jsou následující:

1. položka musí být identifikována plným jménem, které se skládá z URI jmenného prostoru a vlastního jména. Podelementy mohou být identifikovány plným jménem.
2. atribut `encodingStyle` může být použitý pro indikaci způsobu kódování dané položky.

Přestože elementy Header a Body jsou definovány jako nezávislé elementy, existuje mezi nimi vztah. Tento vztah je možné chápat následovně: Položka Body je ekvivalentní položce elementu Header určené pro defaultního příjemce s atributem mustUnderstand nastaveným na hodnotu „1“. Defaultní příjemce je indikován, ale není použit atribut `actor`.

2.4.6 Fault

Tento element se používá k přenosu chyb a/nebo stavových informací zprávy. Pokud je tento element přítomný, pak musí být umístěn v elementu Body. V celé zprávě může být tento element jen jednou.

Element definuje následující podelementy:

faultcode — obsahuje číslo chyby, které je možné použít k vyhodnocení chyby. Standard definuje několik základních kódů pro protokol SOAP. Kódy chyby se generují podobně jako v protokolu HTTP. Rozdílem je, že jednotlivé části kódu se oddělují tečkou a jedná se o textové řetězce (např: `Client.Authentication`).

Základní kódy chyb jsou: `VersionMismatch`, `MustUnderstand`, `Client` a `Server`. Podrobnosti a významy kódů lze nalézt v [1] (kapitola 4.4.1).

faultstring — obsahuje text chybové zprávy, který je možné zobrazit uživateli. Musí být součástí chybové zprávy a měl by vysvětlit o jakou chyby se jedná.

faultactor — identifikuje kdo danou chybu vyvolal. Je podobný atributu `actor`, ale narozdíl od něj indikuje zdroj chyby ve formě URI. Aplikace, která není cílovým příjemcem musí tento element vkládat do zprávy. Cílová aplikace může použít tento element k explicitní indikaci chyby v kombinaci s elementem detail.

detail — je zodpovědný za přenos aplikačních informací o dané chybě vztahujících se k elementu Body. Musí být součástí zprávy tehdy pokud element Body nemůže být úspěšně zpracován. Detailní informace o chybě v položce elementu Header musí být přenášeny rovněž v položkách elementu Header.

Nepřítomnost tohoto elementu lze využít pro rozlišení, zda element má být nebo nemá být zpracováván v okamžiku chyby.

2.4.7 SOAP v HTTP

I když je možné SOAP protokol použít s různými protokoly, jeho nejčastější použití je s protokolem HTTP a jeho metodou POST. Protokol nemění sémantiku protokolu HTTP jen jej rozšiřuje o svou. SOAP zachovává styl zpráv založených na modelu požadavek/odpověd, kdy SOAP požadavek je odesílan jako HTTP požadavek a naopak SOAP odpověď je zasílána jako HTTP odpověď. Typ obsahu HTTP zprávy nesoucí SOAP zprávu musí mít hodnotu „text/xml“

SOAP protokol rozšiřuje hlavičku HTTP požadavku o pole SOAPAction, které může identifikovat obsah a umožňuje tak různým serverům nebo firewallům příslušně filtrovat HTTP zprávy.

SOAP odpověď dodržuje sémantiku HTTP stavových kódů pro komunikaci, takže např. pokud se posílá zpět výsledek správně zpracované zprávy zasílá se kód 2xx. Naopak pokud došlo k chybě během zpracování, musí se zaslat kód 500 „Internal Server Error“ a tělo SOAP zprávy musí obsahovat element Fault.

2.4.8 SOAP a RPC

Jednou z výhod protokolu SOAP je jeho schopnost zapouzdřovat volání RPC funkcí do jazyka XML. RPC volání je možné mapovat do HTTP požadavku a odpověď na volání RPC funkce pak do HTTP odpovědi. Protokol HTTP, ale není jediný protokol a může být použitý i jakýkoliv jiný protokol, který umí přenášet zprávy protokolu SOAP.

Pro volání funkcí je nutné poskytnout následující informace:

URI cílového objektu

jméno metody

volitelně signaturu metody

parametry metody

volitelně data v hlavičce

RPC volání i odpověd jsou přenášeny v elementu Body a s použitím následující reprezentace:

volání metody je modelováno jako struktura

volání metody je chápáno jako jednoduchá struktura obsahující elementy pro každý vstupní nebo vstupně/výstupní parametr. Struktura je zároveň pojmenována a otypována stejně jako jméno metody.

každý vstupní nebo vstupně/výstupní parametr je zobrazován jako element s jménem a typem odpovídající jménu a typu parametru. Vše ve stejném pořadí jako v signatuře metody.

odpověď metody je modelována jako struktura

odpověd je chápána jako jednoduchá struktura obsahující elementy s návratovou hodnotou pro každý výstupní nebo vstupně/výstupní parametr. První element obsahuje návratovou hodnotu funkce a další pak jednotlivé parametry ve stejném pořadí jako v signatuře typu.

Každý element má stejné jméno a typ jako příslušný parametr. Jméno návratové hodnoty není podstatné, stejně tak jako jméno struktury. Přesto se doporučuje dodržovat konvenci, která pojmenovává strukturu jménem metody s přidaným řetězcem „Response“.

přenos chyby v metodě je přenášen pomocí elementu Fault. Pokud přenosový protokol přidává další pravidla pro identifikaci chyby, je nutné je rovněž uvést.

Chybějící parametry může metoda buď zpracovat, nebo vrátit chyby.

3. JUMBO klient

3.1 Analýza současného stavu

Jak již bylo uvedeno v pasáži věnované systému JUMBO, jedná se o experimentální databázi. Zatím existoval pouze server se kterým bylo možné pracovat pomocí prohlížeče HTML stránek. Toto řešení ale neposkytuje ve své podobě dostatečný komfort pro práci uživatele. Samotný server je bohužel stále nedokončený a některé funkce vznikaly souběžně až s touto diplomovou prací a tak nebylo možné je plně otestovat.

3.2 Specifikace požadavků

procházení mezi objekty na serveru v rámci daného zobrazování

vytváření instancí objektů serveru

editace a mazání atributů a vztahů

volání operací nad objekty

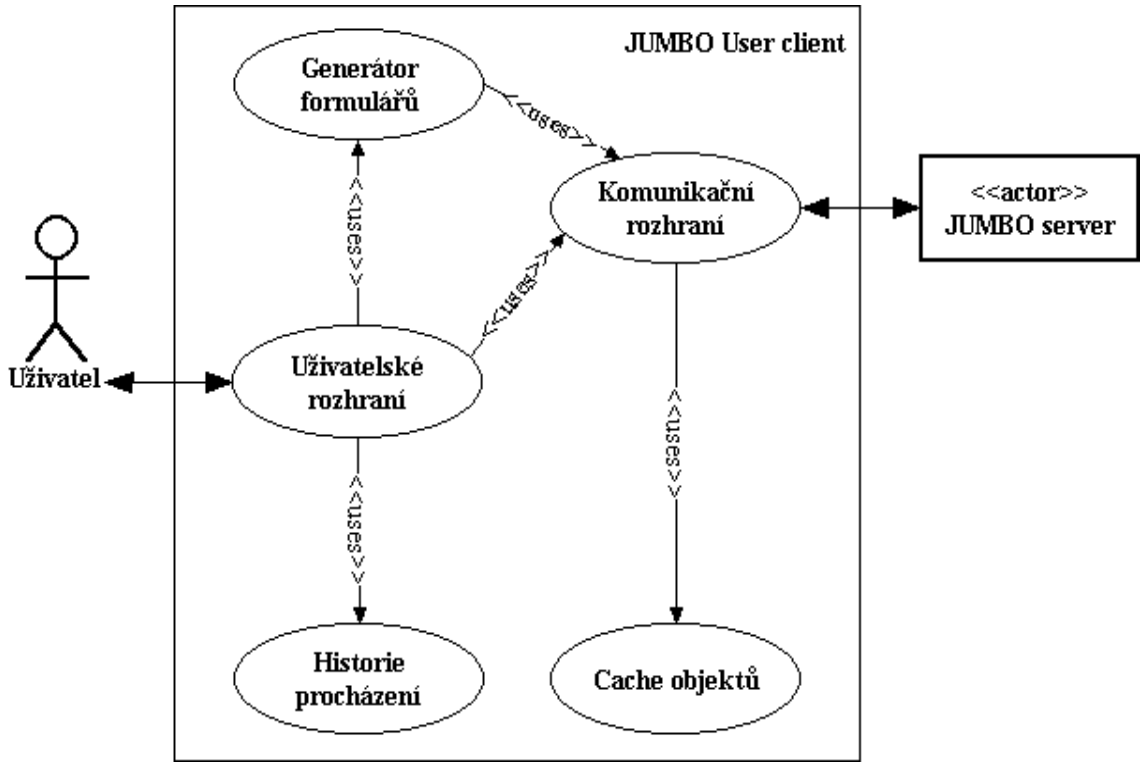
řešení cachování objektů serveru na straně klienta

aplikaci implementovat tak, aby umožňovala více způsobů zobrazení objektů

3.3 Analýza

3.3.1 Generátor formulářů

Jedním ze základních nedostatků nynější verze klienta je jeho uživatelská nepřívětivost v oblasti zobrazování objektů. Aplikace by proto měla být v tomto ohledu variabilní a je ji proto nutné navrhnout tak, aby umožňovala zobrazovat objekty různými způsoby. Tuto schopnost zajišťuje „Generátor formulářů“, který je možné vidět na obrázku 3.1. Tento generátor poskytuje rozhraní pro samotné uživatelské rozhraní, které na základě dat z generátoru a samotných dat aplikace nabídne tento formulář uživateli.



Obrázek 3.1: Navržené schéma aplikace

3.3.2 Komunikace se serverem

Další požadavek, který bylo nutné řešit byla samotná komunikace mezi klientem a serverem. Tato komunikace je zajišťována komunikačním rozhraním. Toto rozhraní rozhoduje, zda se mají použít již lokálně uložené objekty, nebo se mají tyto objekty načíst z databáze. Rozhodnutí používat cachování objektů na lokální straně vycházelo z požadavku, který předpokládá, že během procházení daty se většina objektů nebude měnit. Toto pravidlo platí hlavně pro metadata jednotlivých modulů. Samotný server pro potřebu cachování poskytuje pro každý objekt časové razítko, které informuje o čase poslední změny objektu. Na obrázku 2.4 je možné vidět, že tento klient bude využívat protokol SOAP. Server v rámci tohoto protokolu poskytuje následující metody. Pojmy `attribute` a `parameter` vychází z pojmu protokolu SOAP, kdy atribut je chápán jako atribut elementu a parametr je chápán jako podelement elementu s metodou.

metoda getClass

string name – jméno třídy ve formátu: modul$třída

Metoda vrátí třídu uloženou v databázi na základě jejího jména

metoda GetObject

attribute long oid – oid objektu

attribute string path – umístění objektu v databázi

return object – data objektu

Metoda může dostat jako parameter jeden ze dvou možných. Nikdy ne oba zároveň. Na základě daného parametru následně vyhledá v databázi příslušný objekt a vrátí ho.

metoda GetObjectHeader

attribute long oid – oid objektu

return object – hlavička objektu

Metoda vrátí pouze hlavičku objektu skládající se z jeho jména, oid, typu a časového razítka

metoda DeleteObject

attribute long oid – oid objektu

Metoda zajistí vymazání objektu s daným oid z databáze.

metoda EditObject

attribute long oid – oid objektu

parameter object – změněné atributy/vztahy objektu

Metoda zajistí změnu hodnot daného objektu. Data jsou zasílány v podobě dvojice jména atributu/vztahu a jeho nové hodnoty jako podelementy parametru object. Pokud se mění kolekce pak se zasílají pouze změny v rámci kolekce. Znak „+“ znamená přidání prvku do kolekce a znak „-“ odstranění následujícího prvku z kolekce.

metoda CreateObject

attribute string class – jméno třídy, kterou chceme instanciovat

return long oid – oid nově vytvořené instance daného objektu

Metoda vytvoří novou instanci dané třídy. Jméno třídy se zadává v podobě modul$třída.

metoda InvokeMethod

attribute string _op – jméno operace, kterou chceme vykonat

attribute long _this – oid objektu, který danou operaci volá

parameter parameters – parametry dané operace

return result – výsledek volání operace

Metoda zajišťuje volání operací objektů v databázi. Výsledkem operace je následně buď literál, nebo oid objektu, který obsahuje výsledek volané operace.

3.3.3 Cache

Jak již bylo zmíněno, objekty bude potřeba na straně klienta dočasně ukládat. Algoritmus cachování vychází z možností serveru a kopíruje princip cachování protokolu HTTP. Komunikační manažer nejprve otestuje existenci objektu v cache. Pokud tento objekt neexistuje, pak požádá server rovnou o celý objekt. Pokud je tento objekt již v lokální cache, tak se žádá po serveru jenom hlavička objektu. Následně se porovnají časové razítka. Pokud je časové razítko ze serveru novější než razítko objektu v cache, zajistí se opětovné načtení objektu ze serveru.

3.3.4 Historie

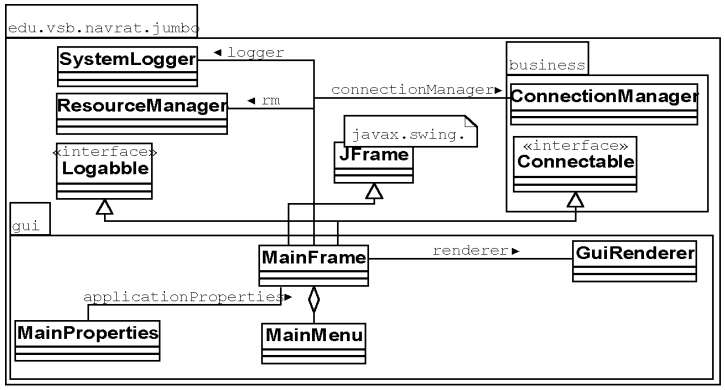
V databázi mohou existovat objekty, které se skládají jen z atributů a tak není možné z tohoto objektu pokračovat na jiný objekt. Tato skutečnost znamená, že se nedostaneme ani na předchozí objekt, ze kterého jsme přišli. Z tohoto důvodu je nutné v aplikaci implementovat historii procházení, která tento nedostatek řeší.

3.4 Návrh a implementace

Přestože samotný server je napsaný v jazyce C++, díky protokolu SOAP mohla být implementace klienta vytvořena v jazyce JAVA. Pro protokol SOAP byla zvolena implementace eSOAP firmy Embedding.Net, která je dostupná jak pro C++, tak i JAVu. Okno aplikace bylo rozděleno na 2 důležité části. V horní části hned pod menu je panel s nabídkou operací, které lze v aplikaci provádět. Pod tímto panelem je plocha pro umístění formulářů daného generátoru. Klient dodržuje zásady pro psaní vícejazykových programů a tak je možné klienta používat ve více jazycích. Konkrétně se jedná o češtinu a angličtinu. Pro ostatní jazyky lze využít nastavení dle aktuálního jazyka systému. Uvádím zde jenom nejdůležitější třídy rozhraní. Podrobný popis všech tříd a metod lze nalézt na přiloženém médiu, kde se nachází i tato aplikace.

3.4.1 Rámec aplikace

Celá aplikace je spojena pomocí třídy `MainFrame`. Tato třída zajišťuje propojení mezi jednotlivými částmi aplikace a je odpovědná za rozmístění jednotlivých grafických komponent v aplikaci. Jedná se o aplikační menu, panel s operacemi nad objekty a stavový řádek. Dalším úkolem této třídy je vytvoření instance komunikačního rozhraní a podpory pro logování událostí v aplikaci. Následující diagram tříd zobrazuje vazby mezi třídami aplikace. Třída `MainMenu` implementuje menu aplikace, které kromě nastavení parametrů poskytuje dva vstupní body k objektům v databázi. Jedná se o seznam extentů a seznam modulů.



Obrázek 3.2: Diagram tříd rámce aplikace

3.4.2 Generátor formulářů

V balíčku `edu.vsb.navrat.jumbo.gui` lze nalézt dále uvedené třídy a rozhraní generátoru.

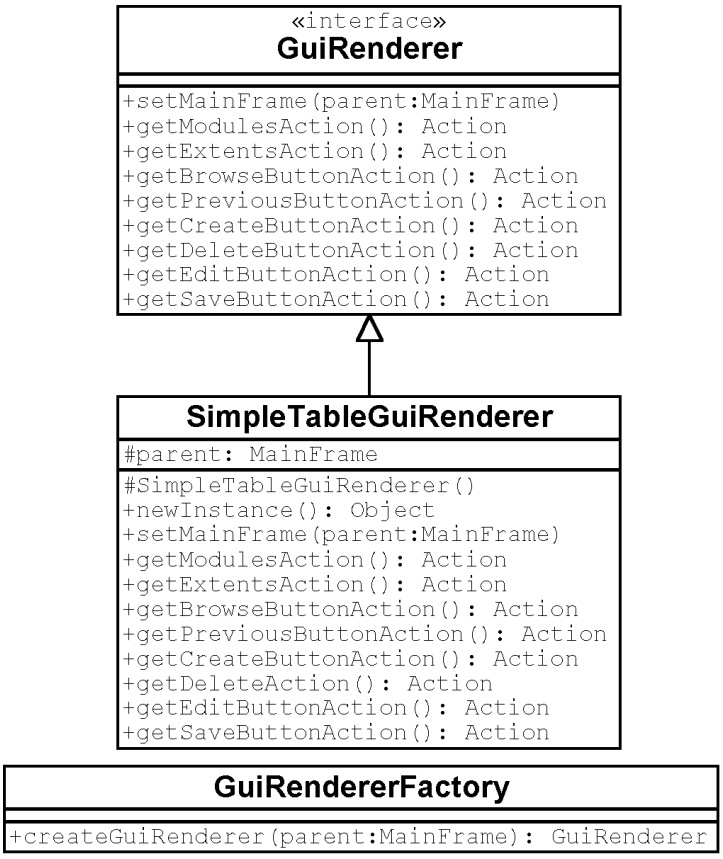
Během implementace aplikace, bylo nutné řešit problém návaznosti této diplomové práce a diplomové práce Ladislava Pavlici. Obsahem jeho diplomové práce je knihovna grafických uživatelských prvků. Použitím těchto prvků bude možné vytvářet „pěkné“ formuláře. Tyto prvky na základě typu jednotlivých objektů a typu atributů budou volit vhodnou grafickou reprezentaci dat. Například pokud objekt bude obsahovat výčtový typ o velikosti tří možných hodnot nabídne skupinu přepínačů, kdežto pro výčtový typ o větší velikosti nabídne již seznam s povoleným výběrem jedné hodnoty.

Generování těchto formulářů následně zajistí příslušné generátory. Na základě stavu rozpracovanosti jednotlivých prací byl implementován základní generátor, který zobrazuje jak atributy, tak i vztahy objektu jako položky tabulky. Samotné operace objektu jsou pak zobrazovány pod touto tabulkou jako tlačítka. Jelikož zatím neexistuje jiná implementace tohoto generátoru, není uživateli nabídnuta možnost interaktivně si tyto generátory měnit. Přesto již s touto možností klient počítá a generátor je možné nastavit editací konfiguračního souboru klienta.

interface GuiRenderer

Tento interface popisuje rozhraní, které musí jednotlivé implementace generátoru implementovat. Interface vznikl na základě implementace nynějšího generátoru a je možné, že toto rozhraní se mírně pozmění v případě pokusu začlenění generátoru používajícího knihovnu grafických prvků. Metody tohoto rozhraní vrací jednak konkrétní implementace daných akcí, které může klient provádět a dále odkaz na konkrétní panel s formulářem. Základní filozofie vychází z principu, kdy jsou jednotlivé operace s objekty přizpůsobené na míru danému generátoru formuláře.

`setMainFrame()`	nastaví odkaz na aplikační rámec aplikace
`getModulesAction()`	implementace akce zajistí načtení extentu se seznamem všech modulů instalovaných v databázi a jeho zobrazení
`getExtentsAction()`	implementace akce zajistí načtení seznamu všech extentů v databázi a zobrazí tento seznam
`getBrowseAction()`	implementace zajistí zobrazení dalšího požadovaného objektu
`getPreviousButtonAction()`	implementace zajistí přechod na předchozí objekt v historii
`getCreateButtonAction()`	implementace zajistí vytvoření nového objektu
`getDeleteButtonAction()`	implementace zajistí smazání zobrazeného objektu z databáze



Obrázek 3.3: Diagram tříd generátoru

`getEditButtonAction()`	implementace zajistí přepnutí do editačního režimu objektu, případně jeho zrušení
`getSaveButtonAction()`	implementace zajistí uložení změněných hodnot zobrazeného objektu
`getViewPanel`	implementace vrátí aktuální instanci panelu se zobrazeným objektem

class GuiRendererFactory

Tato třída na základě nastaveného parametru (viz 3.4.8) aplikace vytvoří instanci konkrétního generátoru formulářů. Instance je následně předána aplikačnímu rámci.

`createJumboGuiRenderer()`	instancializuje konkrétní implementaci generátoru

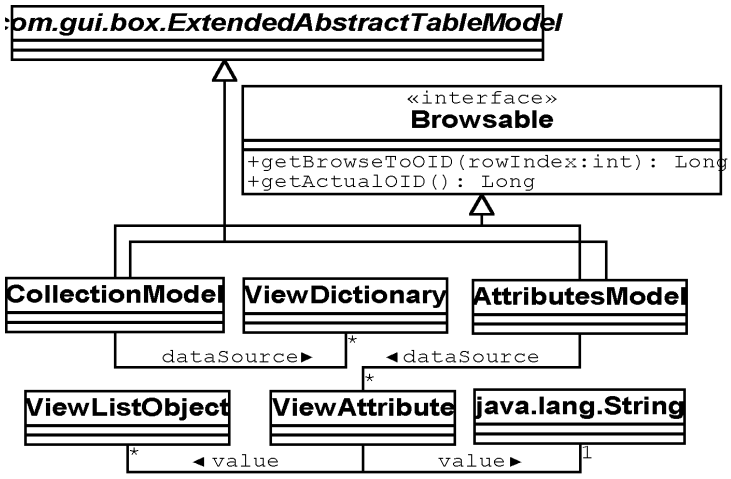
class SimpleTableGuiRenderer

Zatím jediná implementace generátoru formulářů

`newInstance()`	vytvoří a vrátí instanci této třídy

3.4.3 Generátor formulářů - SimpleTableGuiRenderer

Tato zatím jediná implementace generátoru obsahuje dvě základní varianty zobrazení. První varianta se používá pro zobrazení strukturovaného typu `Dictionary` a je použita pro zobrazení seznamu extentů. Druhá varianta je následně použita pro všechny ostatní objekty. Tyto dvě varianty řešení představují třídy `CollectionModel` a `AttributesModel`.



Obrázek 3.4: Diagram tříd modelů pro třídu JTable

interface Browsable

U obou modelů bylo potřebné dozvědět se, který objekt je aktuální a který objekt chceme případně dále zobrazit. Tuto vlastnost zajišťuje implementace tohoto rozhraní.

`getActualOID()`	metoda vrátí oid objektu, na kterém se momentálně nacházíme
`getBrowseToOID()`	metoda vrátí oid objektu na který odkazuje vybraný řádek tabulky

class AttributesModel

Třída implementuje model tabulky a definuje vzhled této tabulky. Model je schopen rozhodnout na základě znalostí hodnoty atributu, zda má zobrazit jednoduchý řetězec nebo kombobox s více hodnotami. Model rovněž podporuje editační režim. Editace primitivních typů se provádí přímo v tabulce. Typy vycházející z databázového typu Blob a veškeré reference a kolekce se následně editují v jiných dialozích a model zajišťuje jen aktualizaci výsledku editace.

class CollectionModel

Druhý z modelů pro tabulku zobrazující objekty je určen pro zobrazení položek kolekce typu `Dictionary`. Tento model je určen pouze pro zobrazení dat a nepovoluje žádnou editaci.

class ViewAttribute

Třída je výsledkem transformace atributů a vztahů do grafických objektů. Uchovává informace o daném atributu nebo vztahu a tvoří jeden řádek v tabulce s modelem třídy `AttributesModel`. Instance této třídy si uchovávají informace o tom, zda byly modifikovány což umožňuje ukládat jen ty atributy, které se skutečně změnily.

class ViewDictionary

Třída uchovává informace o položce kolekce a tvoří jeden řádek v tabulce s modelem třídy `CollectionModel`.

class ViewListObject

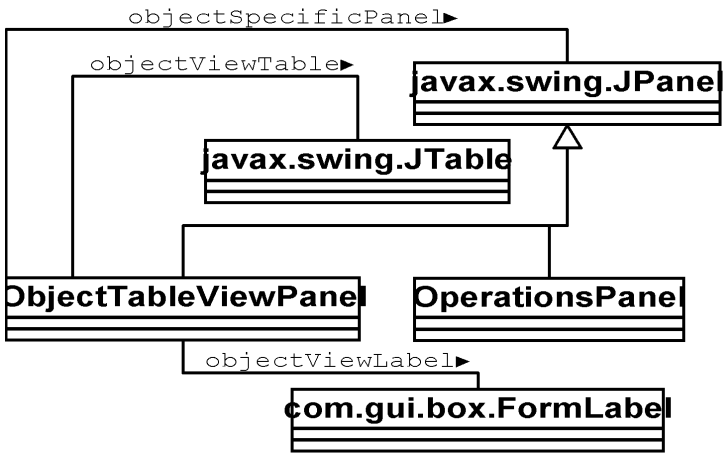
Instance této třídy mohou obsahovat libovolný objekt a uchovávají si indikaci zda je daný objekt vybraný. Využívá se jako položka komboboxů v tabulce a v editaci kolekcí a referencí.

BlobDialog

Dialog poskytuje schopnosti editace objektů odvozených od typu `Blob`.

ListEditDialog

Dialog umožňuje editovat u objektu vztahy a atributy, které jsou typu kolekce. Pokud je znám pro typ položky extent, je obsah tohoto extentu uživateli nabídnut. V ostatních případech je možné novou hodnotu vložit přímým zadáním OID (v případě typů).



Obrázek 3.5: Diagram tříd panelů generátoru

class ObjectTableViewPanel

Třída má na starost samotné zobrazení požadovaného objektu a rozmístění jednotlivých panelů. Po této inicializaci vzhledu se pak již stará o změnu stavu některých tlačítek pro práci s objekty.

class OperationsPanel

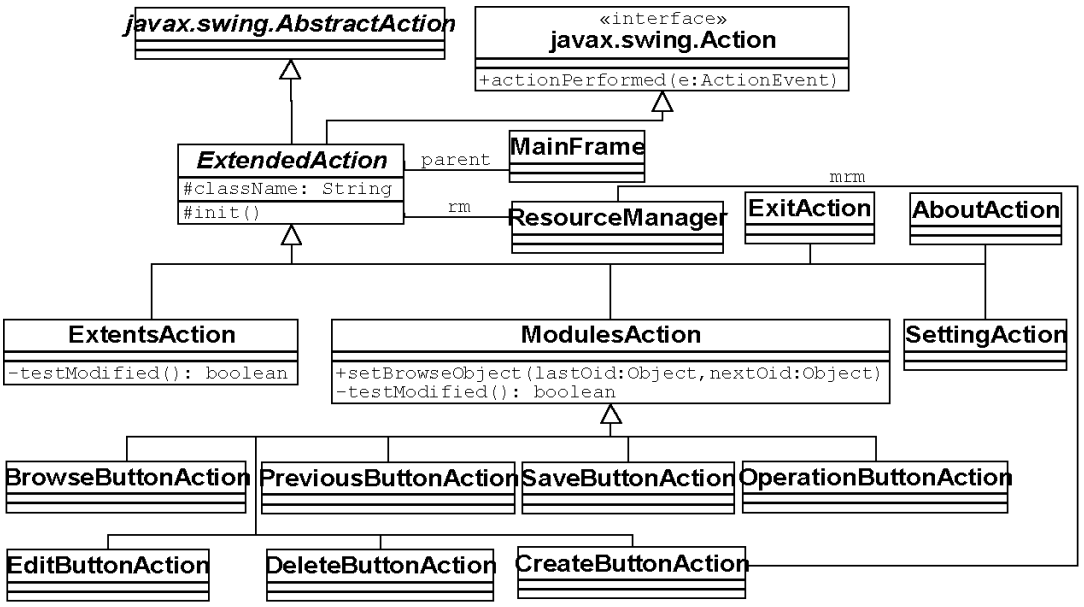
Třída vytváří v aplikaci panel obsahující tlačítka všech operací, které lze vykonat nad daným objektem. Instance této třídy je přiřazena proměnné `objectSpecificPanel` ve výše uvedené třídě.

class ParametersInputDialog

Třída implementuje dialog pro zadávání parametrů operací. Dialog zobrazuje vždy zatím místo skutečného názvu parametru slovo „param“ následující jeho pořadovým číslem. Je to způsobeno tím, že server neuchovává po kompilaci metod tyto parametry, ale jen signaturu typů, ze které klient pro tento dialog generuje tyto parametry.

3.4.4 Akce klienta

Veškeré akce klienta jsou v balíčku `edu.vsb.navrat.jumbo.gui.action`.



Obrázek 3.6: Diagram tříd akcí

class ExtendedAction

Tato abstraktní třída je společným předkem všech tříd. Pro jednotlivé akce nastavuje aplikační konstanty na základě aktuálního jazyka aplikace. Nastavuje se klávesová zkratka pro danou akci, jméno akce, znak pro přístup přes klávesu Alt a kontextová nápověda pro akci.

`init()`	metoda inicializuje konkrétní konstanty

Akce rozhraní

Jedná se o obecné akce, které nemají vztah k databázi.

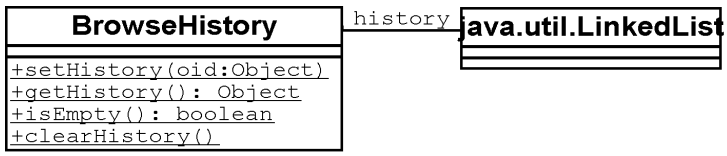
`AboutAction`	akce zobrazí informační dialog o aplikaci
`ExitAction`	akce ukončí aplikaci a uloží parametry aplikace do souboru
`SettingAction`	akce zobrazí dialog pro nastavení parametrů aplikace

Databázové akce

Tyto akce pracují s objekty databáze. Jedná se o podporu čtyř základních databázových operací, o procházení databází a volání operace nad daným objektem.

`ExtentsAction`	akce zobrazí seznam se všemi extenty databáze
`ModulesAction`	společná implementace pro další akce. Zajišťuje načtení objektu z databáze
`BrowseButtonAction`	akce na základě režimu práce zajistí přechod na objekt na který ukazuje aktuální řádek tabulky, nebo pokud je v editačním režimu umožní editaci některých atributů a vztahů
`CreateButtonAction`	akce vytvoří novou instanci právě zobrazeného objektu a přejde na něj
`DeleteButtonAction`	akce smaže z databáze právě zobrazený objekt a pokusí se vrátit na předchozí objekt v historii
`EditButtonAction`	akce zajišťuje přepínání do editačního režimu objektu a jeho stornování pokud změny nechceme uložit
`PreviousButtonAction`	akce umožňuje vrátit se v historii procházení na předchozí zobrazený objekt
`SaveButtonAction`	akce uloží provedené změny v objektu do databáze

3.4.5 Historie



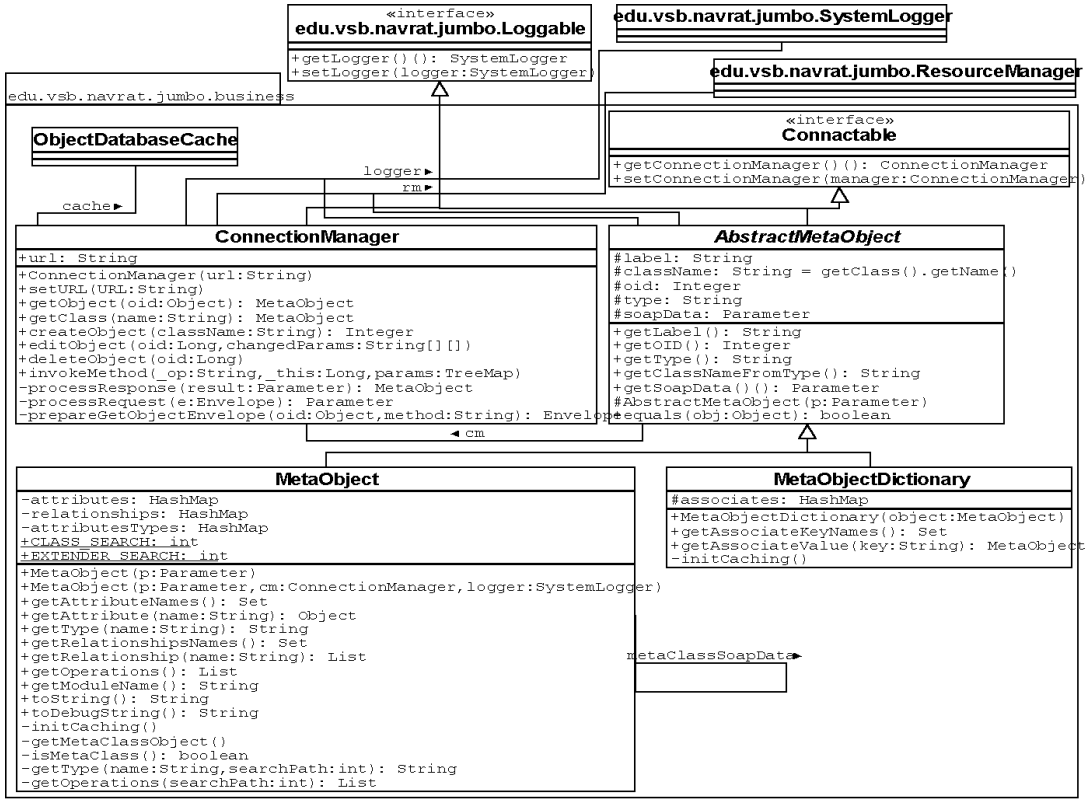
Obrázek 3.7: Třídní diagram historie procházení

class BrowseHistory

Tato třída implementuje historii procházení. Historie funguje na principu zásobníku, kdy při přechodu na další objekt je do historie vloženo oid aktuálního objektu. Následně pokud přecházíme zpět je nejvýše uložené oid použito a je z vrcholu odstraněno. Pokud historie na vrcholu obsahuje hodnotu oid odpovídající seznamu všech extentů, pak je celá historie vymazána. Toto omezení bylo dodáno z implementačních důvodů, kdy se nepodařilo vyřešit přepínání modelů z režimu zobrazení objektů do režimu zobrazení extentů.

3.4.6 Komunikační rozhraní

Komunikační rozhraní zajišťuje veškerou komunikaci se serverem. Komunikace se serverem je bezestavová, což znamená, že s každým požadavkem se připojujeme znovu k serveru a po obdržení odpovědi je toto spojení ukončeno. Třídy tohoto rozhraní se nacházejí v balíčku `edu.vsb.navrat.jumbo.business`. Klient pro svou práci nepotřebuje mít objekty dle standardu ODMG. Třída `AbstractMetaObject` a z ní zděděné třídy tak slouží pouze jako adaptér načtených dat protokolu SOAP. Tyto data je možné pouze číst. Jejich modifikace probíhá nezávisle na těchto objektech. Objekt se aktualizuje po editaci opětovným načtením ze serveru.



Obrázek 3.8: Diagram tříd komunikačního rozhraní

interface Connectable

Rozhraní umožňuje propagovat v rámci aplikace referenci na vytvořené komunikační rozhraní klienta.

class ConnectionManager

Samotné komunikační rozhraní. Třída vytváří zprávy pro protokol SOAP a následně zpracovává odpovědi na tyto zprávy.

`setURL()`	metoda nastaví adresu, na které běží databázový server
`getObject()`	vrátí načtený objekt. Tento objekt se vrátí buď z lokální cache, nebo načtený ze serveru
`getClass()`	vrátí načtenou třídu ze serveru, nebo z lokální cache
`createClass()`	vytvoří novou instanci daného typu na serveru a vrátí její oid
`editObject()`	uloží provedené změny v objektu na server
`deleteObject()`	volá metodu pro smazání objektu z databáze
`invokeMethod()`	zajišťuje spuštění metody objektu ve virtuálním stroji JiveVM na serveru. Po vykonání operace vrací její výsledek

class AbstractMetaObject

Tato abstraktní třída implementuje společné metody dvou následujících tříd.

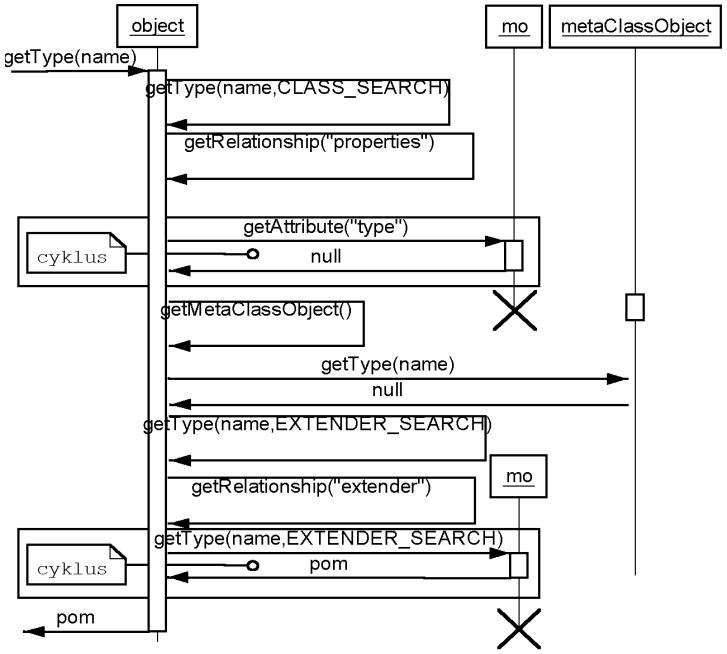
`getOID()`	vrátí oid objektu pod kterým je v databázi uloženo
`getType()`	vrátí typ objektu ve formátu vhodném pro zobrazení
`getLabel()`	vrátí jméno objektu.
`getClassNameFromType()`	vrátí plné jméno třídy ve tvaru `<modul>$<typ>`. Tento formát se používá při komunikaci se serverem.

class MetaObject

Třída uchovává načtený objekt z protokolu SOAP, který obaluje svými metodami. Objekt je nejprve předzpracován, kdy se jednotlivé atributy, vztahy a typy atributů uloží z původní struktury do slovníku parametrů. Hodnoty jednotlivých atributů, vztahů a typu atributů jsou následně zpracovány z těchto parametrů až při požadavku na daný atribut, vztah či typ atributu. Kolekce a vztahy jsou vráceny jako seznam hodnot a binární data jsou zpětně dekódovány z formátu BASE64 do čitelné podoby. Obdobně pokud je atribut typu `Type` je ze signatury převeden zpět do textové podoby.

Vyhledávání typů – objekty v databázi jsou uložené ve stromové struktuře. Při hledání typu atributu je nutné velmi často tento strom procházet. Typ atributu se nejprve vyhledává přímo v aktuálním objektu a dále ve všech objektech, které jsou k objektu ve vztahu „properties“. Pokud nebyl žádný takový objekt nalezen pokračuje se v metaobjektu. Zde se celá situace opakuje až do okamžiku, kdy se typ `Class` ptá sám sebe na typ některého atributu. Pokud jej nezná, vyhledávání pokračuje z původního objektu přes objekty, které jsou k objektu ve vztahu „extender“. Obrázek 3.9 zobrazuje nejhorší variantu vyhledávání.

Načtení operací – Podobně jako u hledání typu pro konkrétní atribut se postupuje při načítání seznamu operací, které lze nad objektem spustit. Načítají se nejprve všechny operace metatřídy pro daný objekt (objekty ve vztahu „operations“) a následně rekurzivně všechny operace objektů ze kterých metatřída dědí (vztah „extender“).



Obrázek 3.9: Hledání typu atributu nebo vztahu

`initCaching()`	metoda převede parametry ze seznamu do struktury slovníku
`getAttributeNames()`	metoda vrátí seznam jmen všech atributů
`getAttribute()`	metoda vrátí hodnotu atributu.
`getRelationshipsNames()`	metoda vrátí seznam jmen všech vztahů
`getRelationship()`	metoda vrátí seznam objektů se kterými je objekt v tomto vztahu
`getType()`	veřejně viditelná verze metody spouští proces vyhledávání typu. Samotné vyhledávání probíhá v neveřejné variantě této metody
`getOperations()`	veřejně viditelná verze metody spouští proces získání seznamu operací nad objektem. Načítání samotných operací probíhá v neveřejné variantě této metody.
`getModuleName()`	metoda vrátí modul ve kterém byl objekt definován
`getMetaClassObject()`	metoda načte objekt, který reprezentuje metatřídu objektu
`isMetaClass()`	zajišťuje konec vyhledávání typu v okamžiku, kdy je objekt totožný s objektem reprezentující typ `Class` v systémovém modulu `ODLMetaObjects`

class MetaObjectDictionary

Tato třída reprezentuje strukturovaný typ `Dictionary`. Vnitřní implementace je podobná jako u předchozí třídy. Rozdíl je pouze v názvech metod a parametrů.

`initCaching()`	metoda vytvoří ze seznamu parametrů opět jejich slovník
`getAssociateKeyNames()`	vrátí seznam všech klíčů slovníku
`getAssociateValues()`	vrátí hodnotu pro daný klíč

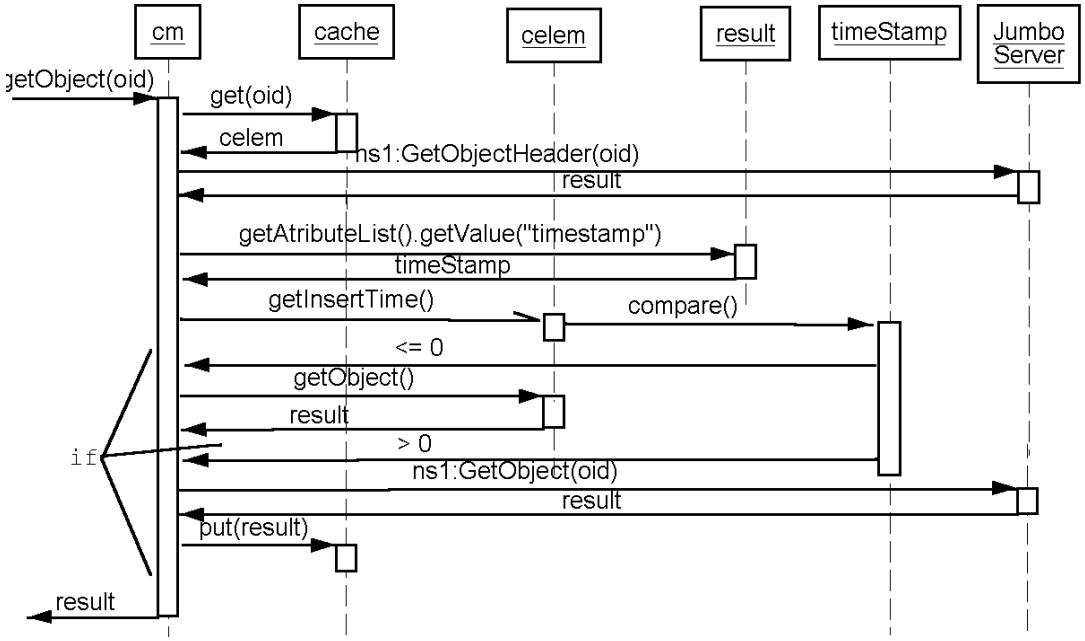
class JumboTypesResolver

V databázi jsou typy jednotlivých objektů a atributů uložené jen jako signatury. Běžnému uživateli ale nic neřekne, že např. vztah typu `{SLODLMetaObjects$Property` je vlastně kolekce typu `Set`, jejíž položky jsou reference na typ `Property` definovaném v modulu `ODLMetaObjects`. Proto je nutné tuto signaturu převést zpět do původního popisu. Tuto funkci zajišťuje tato třída.

`typeResolver()`	překládá signaturu typu na její textový popis

3.4.7 Cache

Komunikace po sítí je většinou časově náročná. U protokolu SOAP je nutné navíc připočítat čas potřebný na režii zpracování dat, jelikož daní za platformovou nezávislost protokolu je velké množství režijních dat ve zprávě. Algoritmus ukládání dat u klienta popsaný v analýze je implementován třídou `ConnectionManager`. Následující obrázek zobrazuje graficky tento algoritmus.



Obrázek 3.10: Algoritmus cachování objektů

class ObjectDatabaseCache a class ObjectDatabaseCache.CacheElement

Třída funguje jako úložiště načtených objektů ze serveru. Umožňuje vyhledávat objekty podle jejich oid a nebo podle jména objektu. Tato schopnost je zajištěna dvěmi slovníky, které si uchovávají jako hodnotu odkaz na stejnou položku cache. Vnitřní třída `CacheElement` představuje položku v cache. Tato položka si uchovává kromě objektu taky jeho oid, jméno a timestamp. Uložení obou klíčů v položce je nutné v okamžiku mazání klíče.

3.4.8 Nastavení aplikace

Klient umožňuje uživateli nastavovat několik vlastností. Tyto vlastnosti se ukládají do konfiguračního souboru, který je hledán v adresáři definovaném proměnnou `JUMBO_HOME`. Pokud je zde nenalezne, pak se pokusí jej vyhledat v domovském adresáři uživatele. Soubor obsahuje níže uvedené parametry. Hodnoty těchto parametrů v případě, že není nalezen soubor lze najít na přiložením CD v popisu třídy MainProperties.

`edu.vsb.navrat.jumbo.gui.MainFrame.host`
: adresa stroje (včetně identifikace `http://`), kde běží server

`edu.vsb.navrat.jumbo.gui.MainFrame.port`
: port na kterém běží server

`edu.vsb.navrat.jumbo.gui.MainFrame.router`
: název routeru pro RPC komunikaci protokolu SOAP. Nezadává se interaktivně v aplikaci.

`edu.vsb.navrat.jumbo.gui.MainFrame.localeID`
: pořadové číslo jazyka prostředí. 0 - systémový jazyk, 1 - česky, 2 - anglicky

`edu.vsb.navrat.jumbo.gui.MainFrame.ui`
: Třída nastavující vzhled prostředí aplikace (Look & Feel)

`edu.vsb.navrat.jumbo.gui.GuiRendererFactory.renderer`
: Konkrétní implementace generátoru formulářů. Nezadává se interaktivně v aplikaci.

class MainProperties

Třída zajišťuje podporu pro práci s parametry aplikace. Zajišťuje jejich načtení po startu aplikace, jejich uložení při ukončení aplikace a jejich změny během práce.

class SettingDialog

Dialog pro nastavení některých parametrů aplikace. Jedná se o adresu stroje, vzhled klienta a použitý jazyk klienta.

3.4.9 Podpůrné třídy

Třídy popsané v této části lze nalézt v balíčku `edu.vsb.navrat.jumbo`. Jedná se o pomocné třídy zajišťující logování událostí a vícejazykovost aplikace.

interface Logable

Interface umožňuje třídám, které ho implementují propagovat odkaz na třídu zajišťující logování událostí a chyb v systému.

class SystemLogger

Třída, která zajišťuje zápis událostí v aplikaci do souboru. Události se zapisují do souboru `jumbo_client.log`. Kromě toho aplikace rovněž přesměrovává do souboru standardní výstup a standardní chybový výstup. Tyto soubory mají příponu `out` a `err`.

ResourceManager

Třída spravuje řetězcové zdroje použité v aplikaci a poskytuje podporu pro vícejazykovost klienta.

4. Závěr

4.1 Zhodnocení

Cílem práce bylo vytvoření nového uživatelského klienta pro objektový systém Jumbo. Během tvorby této aplikace jsem se seznámil s technologiemi, které mají velkou šanci v budoucnu ovlivnit informační technologie. Objektové databázové systémy v nejbližších letech sice nedobudou převahy na trhu s databázovými technologiemi, ale pokud se podaří zvládnout nevýhody těchto systémů mohou velice dobře konkurovat dnešním systémům. Narozdíl od těchto systémů se protokol SOAP již začíná vyskytovat na internetu a intranetu mnohem více. Nejčastěji se o něm mluví ve spojitosti s webovými službami. Do jaké míry bude tato oblast úspěšná ukáže čas.

Velice kladně hodnotím spolupráci s vedoucím diplomové práce, který je autorem systému Jumbo. Nesmím zapomenout ani na Petra Šulu a Ladislava Pavlicu, jejichž diplomové práce se také týkaly systému Jumbo.

4.2 Další vývoj

Aplikace v nynější verzi se podobá trochu své starší předchůdkyni. Aby tato aplikace byla uživatelsky „přítulná“ bude nutné vytvořit pro tohoto klienta další generátor formulářů, který již bude používat výsledek diplomové práce Ladislava Pavlici. Do oblasti úprav stavajícího řešení je možné zahrnout změnu formátu zasílaných dat. Jedná se hlavně o kolekce, které se zatím posílají v protokolu SOAP jako jeden řetězec oddělený mezerami. Pro tyto kolekce je logičtější použít typ `SOAP-ENC:Array`.

Bylo by vhodné upravit databázové schéma systému Jumbo tak, aby uživatel při volbě parametrů operací viděl skutečné názvy těchto parametrů a ne jen slovo jméno `param0` pro první parametr. Rovněž bylo by vhodné vyřešit podporu pro OQL. V nynější podobě lze nabídnout uživateli v nabídkách pouze objekty, které jsou v extentu.

Do oblasti velkých změn je možné zařadit používání rozhraní Reflection v Javě. Použitím tohoto rozhraní bychom mohli reprezentovat aplikační typy přímo jako třídy. To znamená, že např. při požadavku na hodnotu atributu `adresa` bychom nepoužili volání `getAttribute("adresa")`, ale přímo volání `getAdresa()`.

Literatura

[1] Box Don et al. *SOAP: Simple Object Access Protocol*. World Wide Web Consortium, 2000. `http://www.w3.org/TR/SOAP/`.

[2] Cattel R.G.G. et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco, 1999. ISBN 1-55860-647-4.

[3] Beneš M. Data types in persistent object systems. *In: Transaction of the VŠB - Technical Univerzity*, Vol I.(1/2001): str. 4–10, 2001.

[4] Beneš M. Databázové a informační systémy 3. [sylaby k přednáškám], VŠB – Technická Univerzita Ostrava, 2001.

A. Uživatelská příručka

A.1 Požadavky na systém

Aplikace pro svůj běh potřebuje mít nainstalován Java Runtime Environment ve verzi 1.3 a vyšší. Další požadavky na systém vyplývají z podmínek pro provoz tohoto prostředí.

A.2 Instalace a spuštění

Klienta není potřeba složitě instalovat. Stačí rozbalit soubor `jumbo-client.zip` do libovolného adresáře a na tento adresář nastavit systémovou proměnnou `JUMBO_HOME`. Klient se následně spouští dávkou `client.bat` (resp. `client.sh` v Linuxu).

A.3 Nastavení aplikace

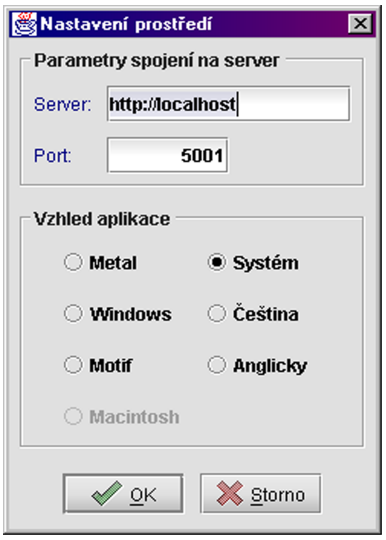
Po spuštění je potřeba nastavit adresu stroje na kterém běží server. Toto nastavení lze provést v dialogu který se nachází pod volbou *Nastavení* v menu *Soubor*. Dialog můžete vidět na obrázku A.1. V tomto dialogu si můžete rovněž nastavit vzhled klienta a jazyk, kterým bude komunikovat. Změna jazyka se projeví až po novém spuštění aplikace.

A.4 Práce s databází

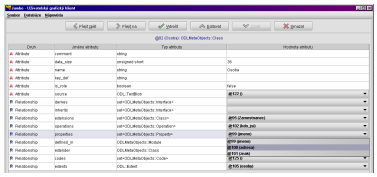
Po spuštění klienta je viditelná pouze čistá pracovní plocha. Práci s databází lze začít jedním ze dvou vstupních bodů. Oba body jsou v menu *Databáze*. Volbou *Extenty* se dostaneme na seznam všech extentů v databázi. Volba *Moduly* následně zobrazí extent, který obsahuje seznam se všemi instalovanými moduly.

A.4.1 Procházení mezi objekty

Na obrázku A.2 je vidět, že objekty jsou zobrazeny v tabulce. Pokud některý atribut nabývá více hodnot, nebo se jedná o vztah jsou hodnoty zobrazeny pomocí komboboxu. Vidět je pouze



Obrázek A.1: Dialog nastavení



Obrázek A.2: Zobrazený objekt

aktuální hodnota na kterou je možné přejít. Změna této hodnoty se provádí výběrem v tomto komboboxu. Pro přechod se následně použije tlačítka Přejít na. Pokud se chceme z objektu vrátit na předchozí objekt, můžeme k tomu použít tlačítko Přejít zpět.

A.4.2 Vytváření objektů

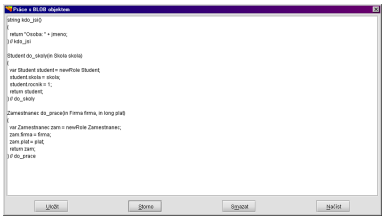
Pod nabídkou tlačítek s operací se nachází popis aktuálně zobrazeného objektu. Je zde zobrazeno oid objektu, jeho jméno (pokud nějaké má) a typ. Pokud je objekt typu ODLMetaObjects::Class, lze vytvořit novou instanci tohoto objektu. To provedeme tlačítkem Vytvořit. Nově vytvořená instance se následně zobrazí.

A.4.3 Editace objektů

Během editace se částečně mění funkce tlačítka Přejít na. Jeho použití neprovede přechod na následující objekt, ale umožní editovat atributy vycházející z typu Blob, atributy typu kolekce a v neposlední řadě vztahy. Do režimu editace je možné se přepnout tlačítkem Editovat. Stejné tlačítko pak slouží i pro zrušení editace bez uložení změn. Provedené změny je možné uložit tlačítkem Uložit. Pokud by jsme se chtěli v režimu editace vrátit na předchozí objekt a již jsme provedli v aktuálním objektu změny, pak je zobrazen potvrzovací dialog. Editace atributů, které jsou primitivních typů je možné provádět přímo v tabulce kliknutím na hodnotu příslušného atributu. V následujících odstavcích je popsána editace jiných typů.

Editace binarních textových dat

Editace těchto dat se provádí v dialogu, který je možné vidět na obrázku A.3. V tomto dialogu je možné vytvořit nový Blob objekt, upravovat existující, případně tento objekt vymazat z databáze. Pro usnadnění editace je možné do pracovní plochy nahrát obsah souboru.

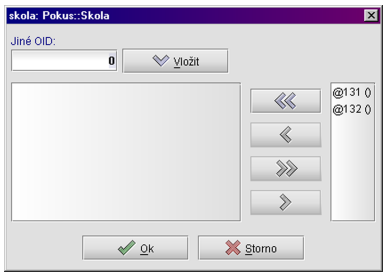


Obrázek A.3: Dialog pro editaci objektu typu Blob

Editace kolekcí a vztahů

Jak pro atributy typu kolekce, tak i pro vztahy se používá totožný dialog. Tento dialog je možné vidět na obrázku A.4. Dialog se skládá ze dvou seznamů, v levém seznamu jsou vidět aktuálně vybrané hodnoty kolekce a vpravo jsou hodnoty, které lze vložit do kolekce. Pokud zde nějaká hodnota chybí, můžeme ji zadat v poli nad seznamem vybraných hodnot. Pokud editujeme kolekci obsahující referenci, pak zadáváme oid objektu. V opačném případě přímo hodnotu, kterou chceme vložit do seznamu. Pokud typ položky kolekce obsahuje v databázi extent, jsou hodnoty tohoho extentu nabídnuty v pravém seznamu.

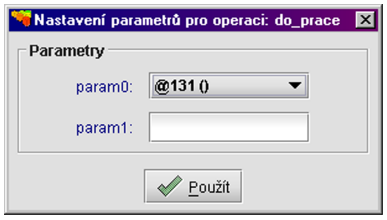
Mezi těmito dvěmi seznamy se nachází panel s tlačítky pro přesun položek těchto seznamů mezi sebou. Tyto přesuny jsou hlídány z pohledu typu kolekce, takže není možné vložit do kolekce typu `Set` dvakrát stejný objekt. Rovněž pokud se jedná jen o vztah není možné, aby vybraných hodnotách bylo více než jedna položka.



Obrázek A.4: Dialog pro editaci kolekcí a vztahů

A.4.4 Volání operací

Panel pod tabulkou je vyhrazen pro tlačítka jednotlivých operací, které lze nad objektem provádět. Pokud operace potřebuje zadat parametry, je zobrazen dialog, který toto zadání umožňuje. Pokud některý z paramterů je typu mající v databázi extent, jsou hodnoty tohoto extenu nabídnuty parametru. V opačném případě je možné zadat přímo oid objektu. Po tomto zadání je operace spuštěna. Pokud operace vrací referenci na objekt databáze, je tento objekt rovnou zobrazen. V ostatních případech je zobrazena zpráva s výsledkem volání operace. Dialog pro zadavání parametrů je možné vidět na obrázku A.5.



Obrázek A.5: Dialog pro zadávání parametrů operací

A.4.5 Mazání objektů

Objekty je možné z databáze vymazat tlačítkem Smazat. Před smazáním je provedena ještě kontrolní otázka. Zatím neexistují pro mazání žádné pravidla (a to ani na serveru). Proto při mazání buďte velmi opatrní.

B. Obsah CD

Adresář	**Popis**
`/jumbo-client.zip`	Zabalená binární verze klienta
`/Ant/`	Obdoba utility make pro kompilaci
`/Java/`	Instalační soubory Java2 SDK v 1.2
`/Jumbo_server/`	Aktuální verze systému JUMBO. Spouští se dávkou `runserver.bat` v adresáři `db`
`/Jumbo/src`	Zdrojové soubory aplikace
`/Jumbo/doc/api`	Popis tříd pomocí JAVADoc
`/Jumbo/doc/dipl-tex`	Texty diplomové práce