Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Tlustý klient pro správu báze znalosti a jednoduché dotazovaní expertního systému

Plzeň, 2012 Jaroslav Kohout

Prázdná strana pro zadání

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne

Jaroslav Kohout

Abstrakt

The aim of the thesis named *Rich Client for Knowledge Base Maintenance and Simple Expert System Questioning* was to create a rich client which will be able to manage a knowledge base of expert system *MATEX*, which is developed for the needs of the Department of Machine Design, Faculty of Mechanical Engineering, University of West Bohemia.

The main aims of the thesis include also a suitable database structure design which will be able to store the heterogeneous data in organized hierarchical dependending structures (trees). This thesis follows in the original project, that can be called *TREX*, *Treeface* or *Treebase*.

The structure of the documment is opened by a chapter, which deals with analysis of the original application and its architecture. Later on the assignment is set in detail. The chapter of the main analysis deals with possible problems and its solutions. The next part of the thesis occupies especially with the implementation of the database and the application. The document is summarized at the end.

Poděkování

Nejprve bych rád poděkoval vedoucímu mé diplomové práce, panu Ing. Kamilu Ekšteinovi, Ph.D., za umožnění zůčastnit se takového projektu a za jeho morální podporu a pochopení při plnění tohoto úkolu. Dále bych rád poděkoval všem osobám, které mě, byť i nevědomky, podporovali a motivovali.

Bez této podpory bych zřejmě práci nikdy zdárně nedokončil.

Obsah

Kapitola 1

Úvod

Zadáním diplomové práce bylo vytvořit tlustého klienta, který bude umožňovat správu báze znalostí expertního systému *MATEX* (Materiál Explorer), vyvíjeného pro potřeby Katedry konstruování strojů FST ZČU. Zároveň bude tento klient podporovat jednoduché vyhledávání, v takové bázi znalosti, prostřednictvím dotazů.

Mezi hlavní cíle práce patří také navržení vhodné databázové struktury (v relačním databázovém systému), která bude schopna ukládat heterogenní data, tj. data libovolného formátu a obsahu, v organizovaných hierarchických závislostních strukturách (stromech).

Tato práce navazuje na původní projekt, který lze rovněž prezentovat jmény *TREX*, *Treeface* či *Treebase*. Protože tedy jde o pokračování již existujícího projektu, je potřeba čtenáře nejprve seznámit s původním stavem. Tento stav je detailně analyzován jak z pohledu implementace, tak i z funkčního pohledu.

Práce se tedy nejprve zaobírá detailním rozborem původního stavu projektu, kde budou diskutovány použitá řešení a naznačovány návrhy na možná vylepšení a to jak funkčnosti, tak implementace.

Dokument pokračuje detailním zadáním, které obsahuje zejména požadavky na implementované funkce programu, který bude rozhraním, viditelným a obsluhovaným uživateli vzniknuvšího systému.

Rozsáhlou kapitolou dokumentu je samotná analýza možných řešení v dané problematice. Podstatou této kapitoly je tedy najít a identifikovat veškeré možné překážky související s návrhem a implementací zadané aplikace. Zároveň s označením problému jsou pak uvažována různá řešení a závěry, které jsou na danou otázku odpovědí. Součástí těchto odstavců jsou taktéž

závěry, které byly učiněny, a kterými se řídil samotný návrh a implementace programu. Mezi diskutované otázky patří např. určení vhodného objektově-relačního mapování, způsob uložení heterogenních dat či určení optimálního způsobu sledování změn v relační databázi.

Druhá polovina dokumentu je zejména věnována implementaci vznikající aplikace. Nejprve je tedy rozepsána implementační stránka databáze, tedy výčet a charakteristika databázových objektů. Následuje popis implementace klientského programu.

Závěr práce je pak věnován zhodnocení vykonaných úkolů a určení dalších možností rozvoje vzniklé aplikace. Rovněž jsou v závěru vyřčeny mé osobní pocity a poznatky, které souvisely s touto prací.

Kapitola 2

Analýza původní aplikace

V této kapitole je úkolem seznámit čtenáře s původním stavem projektu, na který navazuje tato práce.

Původní projekt sestává z návrhu a implementace databáze v relačním databázovém systému *MySQL* a částečně implementovaného tlustého klienta, vytvořeného pomocí vývojového prostředí *Borland Delphi 7*.

Před zahájením jakékoliv analýzy a návrhu nové verze systému je potřeba provést detailní studii, co a jak již původní program a databáze provádí. Původní aplikace poslouží jako inspirace a měřítko nově vytvořeného díla.

2.1 Funkčnost původní aplikace

Jako první jsem podrobil kontrole původní klientskou aplikaci. Snažil jsem se otestovat a zjistit veškerou (i částečně) implementovanou funkčnost a nyní se zjištěné informace pokusím rozepsat v následujících odstavcích.

2.1.1 Definice typů

Aplikace pravděpodobně měla umožnit vytvoření libovolného množství vlastních, uživateli definovaných, datových typů jako např. délka, šířka či barva. Každý uzel datového stromu má přiřazený právě takový typ, čímž je určen význam uložené hodnoty. V původní verzi klientského programu nebyla funkčnost správy takových typů implementována, nicméně bylo možné je zobrazit a používat pro vytváření a editování uzlů stromu.

2.1.2 Vlastní datový strom

Původní klient dovoloval svému uživateli kompletní správu datového stromu tj. správu uložených a přidávání nových uzlů. Ke každému uzlu šlo rovněž připojit libovolné množství doprovodných textů.

2.1.3 Doprovodné texty

Jak již bylo napsáno v předchozím odstavci, klient umožnil svému uživateli spravovat texty přiložené k uzlům datového stromu. Pokud uživatel vytvářel nový, nebo editoval stávající text, využíval k tomu určeného okna, které zprostředkovalo i základní vizuální formátování konkrétního textu. I z pouhé aplikace šlo vypozorovat, že k formátování se používá minimální množiny *HTML*[1] tagů.

2.1.4 Přiložené soubory

Při procházení původní aplikace jsem narazil na ovládací prvek, který pravděpodobně měl sloužit k nahrání souboru k právě otevřenému uzlu. Bohužel tato funkce pravděpodobně nebyla implementována, protože při pokusu o její vyvolání došlo k zobrazení chybové zprávy. Ovšem i samotná chybová zpráva poskytla cennou informaci, a to že mělo být užito *SFTP*[2] serveru pro uložení těchto souborů.

2.1.5 Uložené připojovací informace

Již původní verze programu nevyžadovala po uživateli zadání připojovacích informací při každém novém připojení k databázi, ale využívala speciálního souboru uloženého ve složce s aplikací, kde byly zapsány *IP adresy*[3] databázových serverů, ke kterým byla možnost se připojit. Na druhou stranu zde nebyla implementována žádná správa tohoto souboru a dokonce nebylo možné ani zadat připojovací informace ručně.

[1] HyperText Markup Language je značkovacím jazykem hypertextu.

[2] SSH File Transfer Protocol je protokol pro bezpečný přenos souborů počítačovou sítí.

[3] IP adresa je číslo jednoznačně identifikující síťové rozhraní v počítačové síti, která používá IP protokol.

2.2 Původní struktura databáze

Následovalo studium původní struktury databáze, kde bylo potřeba určit případné nedostatky nebo naopak inspirativní postřehy. Tato analýza tvoří stavební kámen nově navržené struktury databáze. Rovněž databáze poskytla lepší obraz toho, co ještě bylo v plánu implementovat do klientské aplikace. Model původní podoby databáze je zobrazen v příloze B.

2.2.1 Tabulka definovaných typů

Jak již bylo napsáno v kapitole 2.1.1, aplikace využívala nadefinovaných datových typů. Tyto typy byly uloženy v jediné databázové tabulce `types`.

Sloupec	**Typ**	**Popis**
`name`	`varchar(255)`	Zkrácený identifikační název typu
`published`	`text`	Publikovaný název datového typu
`contains`	`text`	Sloupec popisující dovolené vazby jinými typy
`unit`	`varchar(63)`	Jednotky typu
`base_type`	`varchar(15)`	Textová reprezentace základního (skutečného) typu
`updated_by`	`varchar(63)`	Autor záznamu
`updated_on`	`timestamp`	Čas vzniku záznamu

Tabulka 2.1: Sloupce původní tabulky types

Struktura této entity je naznačena v tabulce 2.1. Jedním z klíčových sloupců byl `base_type`, který obsahoval textovou reprezentaci skutečného datového typu např. `String` či `Float`. Dalším, ne úplně typickým polem, byl sloupec `contains`, který speciálním textovým zápisem určoval závislosti na jiných, v této tabulce definovaných, typech. Pole `updated_by` a `updated_on` tvoří *auditorské informace* záznamu, tj. uživatelské jméno autora aktuální podoby záznamu a s tím související časová značka.

Vhodným postupem k vylepšení této tabulky by mohlo být nahrazení sloupce s textovým zápisem základního typu za tabulku, která bude obsahovat veškeré podporované základní typy dat databáze.

Dále by bylo vhodné odstranit sloupec se zápisem vazeb na další typy a jeho nahrazení tabulkou, která bude popisovat možné vztahy definovaných typů prostřednictvím cizího klíče.

2.2.2 Tabulka uzlů stromu

Tabulka s uzly datového stromu měla tradiční strukturu, která se běžně používá při tvorbě stromových hierarchií.

Sloupec	**Typ**	**Popis**
`id`	`int(10)`	Identifikace záznamu uzlu
`parent`	`int(10)`	Odkaz na nadřazený uzel
`type`	`varchar(255)`	Odkaz na definovaný typ dat tabulky `types`
`value`	`varchar(255)`	Vlastní hodnota uzlu
`updated_by`	`varchar(63)`	Autor záznamu
`updated_on`	`timestamp`	Čas vzniku záznamu

Tabulka 2.2: Sloupce původní tabulky nodes

Stromová hierarchie je zde tvořena pomocí prvních dvou sloupců. Prvním je sloupec identifikující konkrétní uzel. Druhý sloupec pak obsahuje identifikaci uzlu, pod který ve stromu spadá. I zde, stejně jako u tabulky typů, jsou pole s auditorskými informacemi. Veškeré sloupce jsou zapsány v tabulce 2.2.

U této tabulky nebyl nalezen žádný nedostatek, který by bylo potřeba odstranit.

2.2.3 Tabulka textů

Databáze obsahuje tabulku `texts` pro uložení textů, jejíž struktura odpovídá tabulce 2.3.

Jedinou změnou, která by se dala uvážit, je nahrazení sloupce s pozicí jménem textu, které by bylo unikátní v rámci jediného uzlu stromu. Tato změna je spíše kosmetického charakteru a neodstraňuje tedy žádný nedostatek.

Sloupec	Typ	Popis
id	int(10)	Identifikace záznamu textu
node	int(10)	Odkaz na uzel (tabulka nodes), jemuž text náleží
position	int(11)	Pořadové číslo resp. identifikace textu v rámci uzlu
published	text	Vlastní text
updated_by	varchar(63)	Autor záznamu
updated_on	timestamp	Čas vzniku záznamu

Tabulka 2.3: Sloupce původní tabulky texts

2.2.4 Tabulka odkazů na soubory

Jak již bylo naznačeno při studiu původní klientské aplikace, každý uzel databáze může mít přiloženo libovolné množství souborů, resp. příloh. V původní verzi databáze byl uložen pouze odkaz na patřičný soubor, jehož fyzickým uložištěm byl souborový server (konkrétně SFTP). Tabulka links, jejíž struktura je znázorněna tabulkou 2.4, byla odpovědná právě za uložení odkazů.

Sloupec	Typ	Popis
id	int(10)	Identifikace záznamu odkazu na soubor
node	int(10)	Odkaz na uzel (tabulka nodes), jemuž text náleží
filename	varchar(255)	Odkaz se jménem souboru
published	text	Popis souboru
minetype	varchar(255)	Typ obsahu souboru
updated_by	varchar(63)	Autor záznamu
updated_on	timestamp	Čas vzniku záznamu

Tabulka 2.4: Sloupce původní tabulky links

Zde stojí za zvážení, zda nevyužít tzv. *BLOBů*[4], které jsou součástí

[4]Binary Large OBject (BLOB) je datový typ blíže nespecifikovaných binárních dat

většiny dnešních databázových systémů a přináší výhodu společné správy souborů a dat v databázi, například automatické odstranění obsahu souboru (BLOBu) z databáze při mazání záznamu o souboru z tabulky.

2.2.5 Tabulka změn

Původní podoba databáze zahrnuje tabulku `changelog`, ve které měly být ukládány zprávy o změnách, jež vnikají používáním databáze. Struktura záznamu resp. zprávy o vzniklé události odpovídá zobrazení tabulky 2.5.

Sloupec	**Typ**	**Popis**
`id`	`int(10)`	Identifikace záznamu změny
`author`	`varchar(255)`	Uživatel odpovědný za změnu
`change_desc`	`varchar(255)`	Vlastní hodnota změny (zpráva)
`change_date`	`timestamp`	Čas změny

Tabulka 2.5: Sloupce původní tabulky changelog

V původní verzi aplikace tato funkce zřejmě nebyla implementována, protože při jejím testování nedocházelo ke generování zpráv do zmíněné tabulky. Ale z povahy programu, jak jsem jej mohl prostudovat, lze odvodit, že tyto zprávy bude generovat a ukládat klientská aplikace.

Zde se nabízí několik alternativ. Například přenesení řešení generování zpráv o změnách do databáze v podobě tzv. *triggerů*[5] nebo vytvoření záložních tabulek. Problematice sledování změn v databázi se budu podrobně věnovat v kapitole 4.5.

2.2.6 Tabulka parametrů

Tabulka `parameters` je databázovou (centrální) obdobou konfiguračního souboru, do kterého se ukládají obecné parametry aplikace. Struktura odpovídá tabulce 2.6.

Zde stojí za zvážení, zda i taková tabulka musí obsahovat auditorské informace. Dle mého názoru není zde potřeba tyto informace ukládat, protože

v databázi.

[5]Trigger definuje činnosti, které se mají provést jako reakce na událost nad databázovou tabulkou.

Sloupec	Typ	Popis
id	varchar(255)	Název parametru
value	varchar(255)	Vlastní hodnota parametru
updated_by	varchar(63)	Autor záznamu
updated_on	timestamp	Čas vzniku záznamu

Tabulka 2.6: Sloupce původní tabulky parameters

změn v této tabulce bude naprosté minimum, či dokonce nemusí ke změnám docházet vůbec.

2.2.7 Tabulka uživatelů

Seznam uživatelů tvoří databázová tabulka `users`, jejíž záznam odpovídá struktuře tabulky 2.7. Jak je vidět, obsahuje informace jako uživatelské jméno, heslo, celé jméno uživatele, e-mailová adresa a příznak uživatelského oprávnění definujícího přístupová práva k jednotlivým funkcím aplikace. Tento způsob správy uživatelů kompletně obstarává klientská aplikace. Dokonce i přístupová práva jsou plně v kompetenci klienta a jeho implementace.

Sloupec	Typ	Popis
username	varchar(255)	Uživatelské jméno
password	varchar(255)	Heslo uživatele
name	varchar(255)	Skutečné jméno uživatele
email	varchar(255)	E-mailová adresa
auth_level	char(1)	Příznak oprávnění

Tabulka 2.7: Sloupce původní tabulky users

Alternativním a pravděpodobně lepším řešením by bylo využít správy uživatelů, které poskytuje přímo databázový systém a to včetně definování oprávnění. Jediným problémem se může zdát nemožnost uložení dodatečných informací, jako např. celé jméno nebo e-mailová adresa.

Zde se nabízí řešení v podobě vytvoření tabulky s dodatečnými informacemi, jejíž primárním klíčem bude přihlašovací jméno daného databázového

uživatele. Alternativním řešením pak mohou být komentáře, které lze vytvářet k libovolným objektům databáze, a to včetně uživatelů.

2.3 Další zjištění

Původní aplikace obsahovala uživatelské rozhraní typu *MDI*[6], které umožnilo uživateli připojit více databází naráz v jedné instanci aplikace. Každé z oken sdružených v hlavním okně odpovídá jedné připojené databázi.

Architektura původního programu neobsahovala žádné *ORM*[7] v pravém slova smyslu. Obsluhu databáze, resp. manipulaci s daty zde obstarávala programová jednotka obsahující sadu k tomu určených funkcí.

Jak již bylo naznačeno výše, aplikace využívala pro uložení souborů SFTP serveru a v databázi byl uchováván pouze odkaz na takto uložený soubor.

[6]Multiple Document Interface je rozhraní skládající se z několika formulářů, které se nacházejí uvnitř hlavního okna aplikace.

[7]Objektově relační mapování (z angl. Object-Relational Mapping) je programovací technika zajišťující konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem.

Kapitola 3

Požadavky na aplikaci

V tuto chvíli by se nabízelo stanovit požadavky na konečný produkt této práce, tj. na konečnou podobu vyvíjené aplikace. Požadavky budou vycházet z původního systému, zadání práce a samozřejmě z požadavků, které byly domluveny ústně s vedoucím práce.

3.1 Funkční požadavky

Nejprve tedy soupis veškerých požadavků na funkčnost aplikace.

3.1.1 Správa typů

Nově vytvořená aplikace umožní svým uživatelům definovat vlastní datové typy. Každý z takto definovaných typů sestává z vlastního jména, základního datového typu a případně jednotek. Samozřejmostí je možnost upravit již existující typ či jej úplně odstranit.

Ukázkou takového datového typu může být např. typ **délka** se základním typem **celé číslo** (`Integer`) a jednotkou **mm** (milimetr).

3.1.2 Definování struktury

Program umožní uživatelům definovat předepsanou strukturu, resp. posloupnost uložených datových typů. Tyto posloupnosti budou navíc specifikovat množství výskytů jednotlivých typů v konkrétní větvi vlastního datového stromu.

Demonstrací definice posloupnosti datových typů nechť je např. uzel typu **rozměry** může obsahovat právě jeden uzel typu **šířka**, rovněž právě jeden uzel typu **délka** a maximálně jeden uzel typu **výška**.

Definované posloupnosti bude samozřejmě dovoleno editovat či mazat.

3.1.3 Správa uzlů

Uživatelé aplikace mohou prohlížet datový strom ve srozumitelném formátu. Uzly lze libovolně přidávat, upravovat i odebírat. Každý uzel je určen svou polohou ve stromu, přiřazeným datovým typem a svou vlastní hodnotou.

Příkladem může být uzel datového typu délka v milimetrech a hodnotou 1, tj. **délka = 1 mm**.

3.1.4 Doprovodné texty

Každému uzlu umožní aplikace přidat doprovodný text, u kterého lze provádět alespoň základní formátování jako styl písma, apod. Program dovolí texty libovolně vytvářet, upravovat i mazat.

3.1.5 Přiložené soubory

Ke každému uzlu datového stromu umožní program připojit (nahrát) soubor libovolného obsahu. Takové soubory bude dovoleno opakovaně stáhnout a samozřejmě je půjde libovolně vytvářet, nahrazovat či mazat.

3.1.6 Kontrola uzlů

Klientská aplikace bude, ať už aktivně či pasivně, nabádat uživatele k zadávaní pouze odpovídajících hodnot (hodnot jednotlivých uzlů) a k dodržení definované struktury (posloupnost typů).

Situace, kdy program bude schopen zjistit nevyhovující chování uživatele, je např. nastavení hodnoty uzlu, který je typu délka (celé číslo), na hodnotu, která nevyjadřuje celé číslo.

3.1.7 Vyhledávání uzlů

Program poskytne uživateli možnost vyhledávání, resp. filtrování uzlů datového stromu pomocí jednoduchého dotazovacího jazyka. Příkladem či in-

spirací takového jazyka může být *XPath*[1] nebo libovolný jiný či vlastní.

Aplikace umožní dotazy charakteru např.: „Zobraz všechny materiály, které mají červenou barvu a délku větší než 1 mm!“.

3.1.8 Sledování změn

Vytvořená aplikace dovolí uživatelům sledovat změny provedené v databázi. Tím jsou myšleny změny prováděné jak uživatelem samotným, tak i ostatními uživateli, kteří přistupují k datům.

Tato funkce umožní zjistit uživatele, který je odpovědný za konkrétní změny v uložených datech.

3.1.9 Správa uživatelů

Program zprostředkuje správu uživatelů, kteří mají přístup k uloženým datům. Součástí správy uživatelů bude definování oprávnění, kterými budou jednotliví uživatelé disponovat. Příkladem mohou být uživatelé, kteří mohou pouze prohlížet nebo uživatelé s možností změnit uložená data, apod.

3.1.10 Uložení připojovacích informací

Klientská aplikace dovolí uživatelům ukládat připojovací informace pro jejich příští použití. Rovněž bude umožněno tyto informace spravovat, tj. přidávat, editovat či mazat.

Program by měl mít snahu vyžadovat minimální množství zadáváných informací k připojení od uživatele.

3.2 Ostatní požadavky

Zbývá doplnit požadavky na aplikaci, které se nevážou na hlavní funkčnost, resp. účel programu.

[1] XML Path Language je počítačovým jazykem umožňující adresování částí XML dokumentu.

3.2.1 Požadavky na rozhraní

Rozhraní programu, které bude zajišťovat interakci s uživatelem, bude plně grafické, intuitivní a bude dodržovat standardní zvyklosti programu s uživatelským rozhraním.

Aplikace, stejně jako původní program, bude postavena na rozhraní typu MDI, které dovolí pracovat s více uložišti v jedné spuštěné instanci programu.

3.2.2 Požadavky na výkon

Klientská aplikace bude poskytovat dobrou odezvu rozhraní a svou povahou bude hardwarově nenáročná, aby ji šlo provozovat i na slabších strojích.

3.3 Motivace

Projekt vznikl jako systém pro správu báze znalostí expertního systému MATEX (Material Explorer), vyvíjeného pro potřeby Katedry konstruování strojů FST ZČU.

Vzniklá aplikace tak umožní pracovníkům výše zmíněné katedry kvalitně uložit informace o materiálech do jimi definovaného katalogu, který vytvoří právě hierarchickou stromovou strukturu.

Kapitola 4

Analýza problému

V této kapitole budou analyzovány klíčové problémy, s nimiž je vytvoření zadané aplikace spjato. Pro každý takový problém zde budou uvedeny alternativy, kterými jej lze zdárně vyřešit.

4.1 Objektově-relační mapování

Nejjednodušší charakteristika pojmu objektově-relačního mapování říká, že jde o techniku konverze dat mezi relačním databázovým systémem a objektově orientovaným programovacím jazykem. Tato technika vznikla kvůli rozdílné reprezentaci entity v relační databázi, kde je představovaná konkrétním záznamem, a její reprezentaci v objektově orientovaném jazyce, kde jde o instanci třídy (objekt). Důležitou funkcí ORM je persistence dat, což zjednodušeně znamená, že je zde snaha o synchronizaci objektů v operační paměti se záznamy v relační databázi.

Prvním úkolem tedy bylo se rozhodnout, zda použít existující řešení ORM (např. *Hibernate*) nebo zda si napsat vlastní mapování, které lépe vystihne potřeby a požadavky dané problematiky.

4.1.1 Existující mapování

Použití existujícího mapování má nepochybně mnoho kladných stránek. U některých z nich odpadá dokonce nutnost znalostí databázové problematiky či dotazovacího jazyka *SQL*[1]. Umožňují tedy přímé používaní tzv. *Bussiness*

[1]Structured Query Language je dotazovací jazyk pro manipulaci s databázovými daty.

objektů[2].

Nevýhodou použití tzv. hotového řešení může být jeho závislost na konkrétní technologii, užší možnosti jeho přizpůsobení konkrétní problematice či dokonce zvýšené nároky na hardware, jež mohou být cenou za univerzálnost takového řešení.

Hibernate

Hibernate je v jazyce *Java* napsaný *framework*[3], který zprostředkovává mapování objektů (právě jazyka Java) na relační databáze. Protože využívá *JDBC*[4], není vázaný na konkrétní databázovou technologii.

Hibernate nevyužívá jazyka SQL, ale vlastního řešení, tzv. jazyka *Hibernate Query Language* (zkráceně *HQL*). Tento jazyk rozumí pojmům objektově orientovaného programování jako je např. dědičnost či polymorfismus. Pro namapování lze využít buď *mapovacích souborů*, což jsou specifické XML soubory, nebo tzv. *anotací*, které se vepisují přímo do kódu třídy.

Hibernate přináší tradiční výhody hotových řešení a některá další, např. opravdová databázová nezávislost v podobě JDBC. Na druhou stranu je jeho použití doprovázeno nutností použití technologií z okruhu jazyka Java, které svou hardwarovou náročností, zejména pak paměťovou, nejsou vhodné pro výpočetní techniku staršího data výroby.

InstantObjects

InstantObjects je řešením objektově-relačního mapování pro prostředí Delphi. Jeho použití je možné i v dnes již neexistující linuxové verzi *Kylix* či ve svobodné alternativě *Free Pascal*, resp. *Lazarus*.

Mapování umožňuje uložení objektů do relační databáze či do prostých XML souborů. InstantObjects podporuje většinu nejznámějších databázových technologií jako např. *Firebird*, *Microsoft SQL Server*, *IBM DB2*, *Oracle*, *InterBase*, MySQL či *PostgreSQL*.

V současné době je dostupná verze 2.0, která spatřila světlo světa již v roce 2006, a přestože na oficiálním webu je předznamenán příchod verze 3.0, lze se domnívat, že projekt byl pravděpodobně pozastaven.

[2]Objekty reálného světa, se kterými software operuje (např. zákazník, objednávka).

[3]Framework je podpůrný softwarový prostředek pro vývoj obsahující např. knihovny.

[4]Java Database Connectivity definuje jednotné rozhraní pro přístup k relačním databázím v jazyce Java.

tiOPF

Objektově-relační mapování tiOPF je volně dostupný framework s otevřeným zdrojovým kódem pro Delphi či Free Pascal. Tento nástroj podporuje v současné době databázové technologie Interbase, Firebird, Oracle, *Microsoft Access*, Microsoft SQL Server, XML soubory, CSV soubory, aj.

ODB

ODB je volně šiřitelným multi-platformním objektově-relačním mapováním pro programovací jazyk *C++*, které automaticky generuje kód pro práci s databázemi. V současné době podporuje databázové technologie MySQL, Oracle, PostgreSQL, *SQLite* a Microsoft SQL Server.

Database Template Library

Database Template Library, či zkráceně *DTL*, je knihovna, která má za cíl využívat množinu záznamů *ODBC*[5] jako standardní kontejner (viz *STL*[6]) jazyka C++.

Knihovna podporuje databázové ovladače ODBC verze 3.0. Oficiálně jsou pak podporovány databáze Oracle, Microsoft SQL Server, Microsoft Access či MySQL. Neoficiálně pak podporuje PostgreSQL, *Sybase* a IBM DB2.

SOCI

SOCI je knihovnou programovacího jazyka C++ pro přístup k databázím, které podporují dotazovací jazyk SQL. Snaží se o co nejintuitivnější použití SQL dotazů v podobě tzv. *embedded* (tj. vložených) zápisů. Dále knihovna obsahuje podporu práce s objekty, tj. vlastní objektově-relační mapování a také zprostředkovává práci se záznamy pomocí STL knihovny.

SOCI podporuje databázové technologie MySQL, ODBC, Oracle, PostgreSQL či SQLite3.

4.1.2 Vytvoření vlastního mapování

Vytvoření vlastního mapování je druhou z cest, kterou lze následovat při tvorbě zadaného systému. Na rozdíl od hotových ORM je zde nutnost vy-

[5]Open Database Connectivity je standardizované rozhraní pro přístup k databázím.
[6]Standard Template Library je softwarová knihovna jazyka C++.

naložit značně vyšší úsilí pro jeho vytvoření. Pomineme-li ale tuto nevýhodu, lze říci, že dále už přináší toto řešení veskrze jen pozitiva.

Na rozdíl od tzv. hotových řešení zde dává vlastní mapování naprostou svobodu a možnost vytvořit si ORM přesně podle požadavků programu. Nejinak je tomu při výběru databázové technologie a návrhu její struktury.

Abych ale řešení v podobě vlastního mapování jen nechválil, může být jeho překážkou vazba na konkrétní databázovou technologii. Nelze předpokládat, že vzniklé řešení by bylo ihned schopné pracovat s více relačními databázemi, jak je tomu u hotových řešení. Je tedy nutné zvolit cílovou databázi o to pečlivěji. Částečným řešením této nevýhody může být buď vytvoření jednotného rozhraní jako vrstvy mezi databází a vlastním ORM, či použití ODBC.

Důležité je ale podotknout, zda takové zvýšené úsilí bude ve výsledku uplatněno tj. je-li schopnost práce s více databázemi žádoucí. Rovněž vzniknuvší vrstva může přinášet vedlejší náklady v podobě vyšší hardwarové náročnosti a nebo v potlačení žádoucích specifických vlastností konkrétní databázové technologie.

4.1.3 Volba mapování

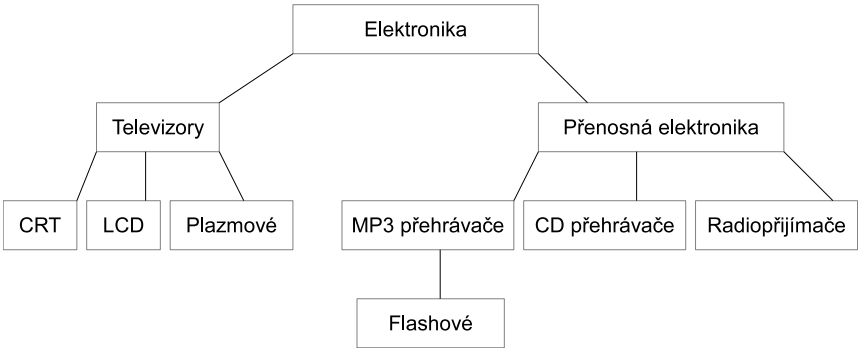
Přestože hotová řešení nabízejí značnou úsporu času potřebného k vytvoření výsledné aplikace, došlo nakonec k rozhodnutí vytvořit si vlastní objektově-relační mapování, které půjde snáze přizpůsobit dané problematice.

Existující mapování rovněž přinášela velkou míru nejistoty, kdy nelze dopředu určit, zda zvolené řešení opravdu vyhoví všem požadavkům. Snahou tedy bylo zabránit zjištění, že hotové řešení nevyhovuje až v průběhu implementace.

4.2 Manipulace se stromem

Jednou z klíčových otázek bylo určení vhodného způsobu uložení stromové struktury do relační databáze. Zvolené uložení musí rovněž umožnit snadné a rychlé získání dat z takové hierarchie.

Pro demonstrování následujících technik užijeme problematiku kategorizace zboží v internetovém obchodě. Zboží prodávané v obchodě patří do kategorií, jež tvoří hierarchickou strukturu (strom). Ukázkou takového stromu kategorií zboží je obrázek 4.1.



Obrázek 4.1: Strom kategorií zboží

Oba ze zde uvedených způsobů pak budou pracovat s takto uloženou strukturou dat.

4.2.1 Základní řešení

Základní řešení pro vytvoření hierarchické struktury (stromu) je realizace prostřednictvím vazby tabulky na sebe samu. Přesněji řečeno tabulka obsahuje primární klíč, který jednoznačně identifikuje jednotlivé záznamy (uzly) a také cizí klíč, který se odkazuje na právě zmíněný primární, čímž identifikuje svůj nadřazený uzel (záznam).

Dále je potřeba v takové tabulce určit způsob, jakým lze identifikovat kořenový prvek, resp. kořenový uzel. Pro tento účel existují dvě následující možnosti.

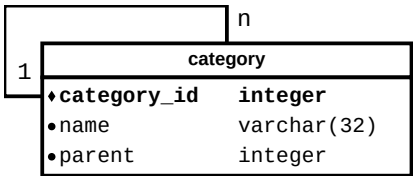
1. **Kořenový uzel nemá nastaven nadřazený.**

   To znamená, že kořenový uzel obsahuje ve svém záznamu prázdnou hodnotu, tzv. `NULL`, ve sloupci, který odkazuje na nadřazený uzel (záznam).

2. **Kořenový uzel odkazuje sám na sebe.**

   Takový záznam obsahuje ve sloupci, odkazujícím se na nadřazený záznam, odkaz na sebe sama, tj. vlastní identifikátor. Co do implementace lze tento způsob označit za jednodušší.

Vytvoříme tedy tabulku odpovídající struktury, jež nám umožní uložit kategorie jak byly zobrazeny na obrázku 4.1. Vzniknuvší tabulka bude mít tedy strukturu odpovídající modelu na obrázku 4.2.



Obrázek 4.2: ERA diagram tabulky kategorií

Máme-li tabulku vytvořenou, naplníme ji požadovanými daty, podle obrázku 4.1. Výsledný obsah tabulky je pak naznačen v tabulce 4.1.

category_id	**name**	**parent**
1	Elektronika	
2	Televizory	1
3	CRT	2
4	LCD	2
5	Plazmové	2
6	Přenosná elektronika	1
7	MP3 přehrávače	6
8	Flashové	7
9	CD přehrávače	6
10	Radiopřijímače	6

Tabulka 4.1: Obsah tabulky kategorií

Nyní již nic nebrání možnosti demonstrovat typické úkoly nad takovou tabulkou. Prvním problémem bude zobrazení kořenových prvků, resp. kořenových kategorií zboží. Jakým způsobem toto provést, ukazuje výpis 4.1.

Dále může být žádoucí zjistit přímé podkategorie jedné konkrétní. Opět postačí standardní výběrový dotaz, jak je demonstrováno výpisem 4.2.

Dobrou ukázkou může být rovněž dotaz, který zjistí, zda jednotlivé kategorie neobsahují (či obsahují) podkategorie, tj. zda jsou listy stromu (viz výpis 4.3). Zde už bylo nutné přistoupit k tzv. *korelovanému* (tj. vnořenému)

Výpis kódu 4.1: Výběr kořenových prvků

```
SELECT * FROM category WHERE parent IS NULL;

 category_id |    name     | parent
-------------+-------------+--------
           1 | Elektronika |
(1 řádka)
```

Výpis kódu 4.2: Výběr podkategorií

```
SELECT * FROM category WHERE parent = 2;

 category_id |   name    | parent
-------------+-----------+--------
           3 | CRT       |      2
           4 | LCD       |      2
           5 | Plazmové  |      2
(3 řádky)
```

dotazu. Obecné doporučení zde říká, že je dobré se takovým dotazům (tam kde to lze) vyhnout, pro jejich zvýšenou náročnost.

Složitějším příkladem pak je zobrazení větve od konkrétního uzlu (kategorie), tj. zobrazení všech podkategorií a to včetně nepřímých.

Za předpokladu, že počet úrovní není dopředu určen si již nevystačíme s konvenčním výběrovým dotazem, ale ke slovu se zde dostávají tzv. *rekurzivní dotazy*, které umožní rekurzivní průchod.

Naštěstí většina dnešních databázových systémů podporuje takové dotazy. Nejčastější je tato funkčnost implementována pomocí tzv. *CTE*[7], ale třeba Oracle má vlastní řešení. Demonstrování užití CTE pro zobrazení větve je patrné z výpisu 4.4.

Další ukázkou, která by bez CTE byla jen těžko realizovatelná, je získaní posloupnosti kategorií ke kořeni resp., získání cesty (viz výpis 4.5).

Tyto demonstrační příklady byly provedeny na databázi PostgreSQL pomocí jeho implementace CTE. Podobnou, pravděpodobně dokonce shodnou strukturu používá i Firebird.

Aby zde nedošlo k chybnému závěru, že bez rekurzivních dotazů nelze takové problémy řešit, považuji za nutné uvést ve stručnosti možnou alter-

[7]Common Table Expressions je pojmenovaná dočasná množina záznamů.

Výpis kódu 4.3: Zobrazení kategorií s příznakem listu

```
SELECT c.*, NOT EXISTS (
  SELECT * FROM category cc WHERE cc.parent = c.category_id
) AS is_leaf FROM category c;

 category_id |          name          | parent | is_leaf
-------------+------------------------+--------+---------
           1 | Elektronika            |        | f
           2 | Televizory             |      1 | f
           3 | CRT                    |      2 | t
           4 | LCD                    |      2 | t
           5 | Plazmové               |      2 | t
           6 | Přenosná elektronika   |      1 | f
           7 | MP3 přehrávače         |      6 | f
           8 | Flashové               |      7 | t
           9 | CD přehrávače          |      6 | t
          10 | Radio přijímače        |      6 | t
(10 řádek)
```

nativu k rekurzivním dotazům.

Rekurzivní průchod takovým stromem lze rovněž realizovat i pomocí bloků jazyka *PL/SQL*[8]. Vytvoření takových bloků ve formě uložených procedur resp. funkcí ale není nijak triviální záležitostí. Rovněž výkonnostní hledisko zde zavádí drobnou míru nejistoty takového řešení.

4.2.2 Alternativní řešení

Není-li k dispozici CTE či podobný nástroj a zároveň není žádoucí programování složitých funkcí jazyka PL/SQL, jež by nepřítomnost podpory rekurzivních dotazů suplovaly, je možné upravit databázové tabulky tak, aby šlo využít prostých výběrových dotazů.

Řešení, které zde budu označovat jako alternativní, vychází z výše uvedeného a rozšiřuje jej. Při studiu tohoto řešení jsem narazil na označení této techniky jako *Nested Set Model* (model vnořených množin). Myšlenkou je zde neuvažovat ani tak o stromu samotném, jako spíše o právě zmíněných množinách.

[8]Procedural Language/Structured Query Language je procedurální nadstavbou jazyka SQL.

Výpis kódu 4.4: Zobrazení větve kategorií

```
WITH RECURSIVE r AS (
    SELECT * FROM category WHERE category_id = 6
  UNION
    SELECT c.* FROM category c, r
    WHERE c.parent = r.category_id
)
SELECT * FROM r;

 category_id |          name          | parent
-------------+------------------------+--------
           6 | Přenosná elektronika |      1
           7 | MP3 přehrávače       |      6
           9 | CD přehrávače        |      6
          10 | Radio přijímače      |      6
           8 | Flashové             |      7
(5 řádek)
```

Podstatou řešení je přidání dvou sloupců do tabulek, obsahujících strom. Tyto sloupce nazvěme pro naše účely levou resp. pravou mezí (rovněž by šlo užít začátek resp. konec). Tyto meze pak tvoří hranici větve stromu, či, jak se zde hodí, hranici množiny. Hranice, resp. meze, které ji tvoří jsou sloupce s celočíselnou hodnotou. Protože se tento model špatně vysvětluje, doporučuji naznačení na obrázku 4.3, kde jsou také znázorněny meze jednotlivých množin (větví).

Stanovení mezí podléhá filosofii, kdy každý uzel má svou levou mez vždy o alespoň jedničku větší než ji má jeho nadřazený uzel. Pravou mez má pak vždy alespoň o jedničku nižší než pravá mez jeho nadřazeného uzlu. Listy mající společný nadřazený uzel dělí jeho mez rovnoměrně mezi sebe. Kořenový uzel, je-li jediný, obsáhne ve svých mezích veškeré podřízené uzly, tzn. obsahuje veškeré podmnožiny. Naopak jde-li o uzel, který je listem, tvoří jeho meze množinu o velikosti právě jedna.

Nejprve je tedy potřeba upravit databázovou tabulku pro tento způsob práce. Model takové tabulky vyjde z původního a přidá dva sloupce s levou a pravou mezí. Diagram vzniklé tabulky bude odpovídat obrázku 4.4.

Protože nejde o triviální problém, může pomoci v pochopení této problematiky tabulka 4.2, která demonstruje již naplněnou databázovou tabulku se shodnými daty jako v předchozí kapitole, tj. dle obrázku 4.1.

Upravená tabulka kategorií odpovídá tedy obrázku 4.4. Naplníme-li ji

Výpis kódu 4.5: Zobrazení cesty

```
WITH RECURSIVE r AS (
    SELECT * FROM category WHERE category_id = 8
  UNION
    SELECT c.* FROM category c, r
    WHERE c.category_id = r.parent
)
SELECT * FROM r;

 category_id |          name          | parent
-------------+------------------------+--------
           8 | Flashové               |      7
           7 | MP3 přehrávače         |      6
           6 | Přenosná elektronika   |      1
           1 | Elektronika            |
(4 řádky)
```

shodnými daty jako v předchozím případě, bude její obsah odpovídat tabulce 4.2.

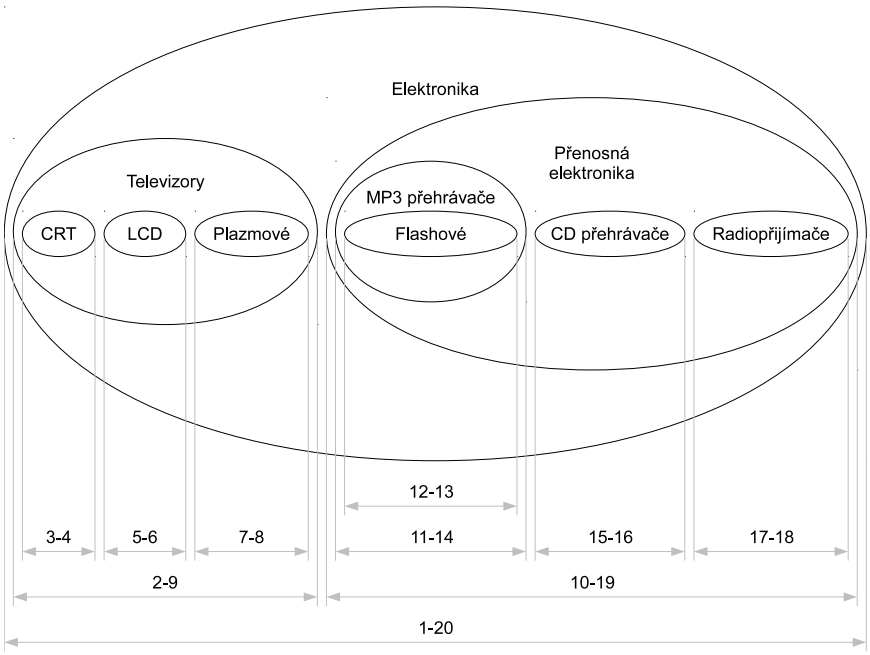
To, jakým způsobem nám toto řešení umožní zbavit se závislosti na rekurzivních dotazech, naznačují modifikované verze předešlých dotazů. Zobrazení větve kategorií bez použití CTE je demonstrováno výpisem 4.6.

Výpis kódu 4.6: Zobrazení větve bez CTE

```
SELECT c1.* FROM category c1, category c2
WHERE c2.category_id = 6
AND c1.lft >= c2.lft AND c1.rgt <= c2.rgt;

 category_id |          name          | parent | lft | rgt
-------------+------------------------+--------+-----+-----
           6 | Přenosná elektronika   |      1 |  10 |  19
           7 | MP3 přehrávače         |      6 |  11 |  14
           8 | Flashové               |      7 |  12 |  13
           9 | CD přehrávače          |      6 |  15 |  16
          10 | Radio přijímače        |      6 |  17 |  18
(5 řádek)
```

Dalším příkladem, kde jsme museli využít rekurzivního dotazu, bylo zobrazení cesty ke kořenové kategorii. V upravené tabulce již není rekurzí potřeba, jak demonstruje výpis 4.7.



Obrázek 4.3: Množinové znázornění kategorií

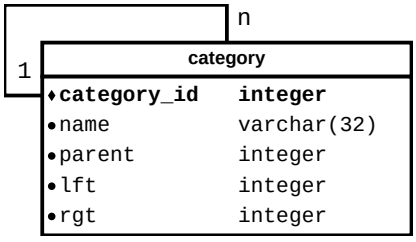
I v případě zjištění, zda je kategorie listem stromu, resp. zda obsahuje podřazené uzly, dojde k výraznému usnadnění, a to v podobě odpadnuvšího vnořeného dotazu. Výsledný dotaz a odpovídající výsledek jsou demonstrovány výpisem 4.8.

Na první pohled se může zdát, že tento přístup je jistou výhrou, ale i zde jsou nějaké ta „proti“. Největším problémem je nutná a nákladná rekonstrukce stromu při změně, např. při přidání listu musí dojít k přepočítání mezí pro všechny následující větve a jejich uzly. Pokud ale databáze nepodporuje rekurzivní dotazy či jiné nástroje, je toto řešení dobrou cestou.

Pokud databáze podporuje triggery a jazyk PL/SQL či jinou alternativu, lze tento přepočet řešit přímo na této úrovni, čímž odpadá veškerá starost klientské aplikace o konzistenci množin.

4.2.3 Volba přístupu ke stromu

V případě uložení stromu do databáze došlo k upřednostnění základního uložení a použití s tím souvisejícího databázového systému s podporou hierar-



Obrázek 4.4: ERA diagram upravené tabulky kategorií

category_id	name	parent	lft	rgt
1	Elektronika		1	20
2	Televizory	1	2	9
3	CRT	2	3	4
4	LCD	2	5	6
5	Plazmové	2	7	8
6	Přenosná elektronika	1	10	19
7	MP3 přehrávače	6	11	14
8	Flashové	7	12	13
9	CD přehrávače	6	15	26
10	Radiopřijímače	6	17	18

Tabulka 4.2: Obsah tabulky kategorií

chií, resp. rekurzivních dotazů. Odpadá zde nutnost rekonstrukčních operací při změně stromu.

4.3 Heterogenní data

Druhým problémem, se kterým bylo potřeba se vypořádat, bylo samotné uložení heterogenních dat. Protože není dopředu jasné, zda bude uloženo číslo, datum či prostý text, je nutné zvolit univerzální typ, ze kterého pak můžeme zpětně vytvořit specifickou informaci.

Rekonstrukci, resp. přetypování hodnoty lze nejlépe provést přímo v klientské aplikaci. Databáze nedovolují přetypování již při výběru dat z databáze. Přesněji řečeno, nelze získat více řádků, kde každý ze záznamů má

Výpis kódu 4.7: Zobrazení cesty bez CTE

```
SELECT c1.* FROM category c1, category c2
WHERE c2.category_id = 8
AND c1.lft <= c2.lft AND c1.rgt >= c2.rgt;

 category_id |          name          | parent | lft | rgt
-------------+------------------------+--------+-----+-----
           8 | Flashové               |      7 |  12 |  13
           7 | MP3 přehrávače         |      6 |  11 |  14
           6 | Přenosná elektronika   |      1 |  10 |  19
           1 | Elektronika            |        |   1 |  20
(4 řádky)
```

v rámci jednotlivých sloupců různé typy hodnot.

Při pokusu takto přetypovávat hodnoty dojde k chybě a dotaz se nevykoná. Tuto skutečnost považuji jako dostatečný důkaz toho, že nelze převést hodnoty na odpovídající typy již na úrovni databáze.

Jako univerzální datový typ poslouží nejlépe délkou neomezený `text`, který případná specifika skutečných typů nelimituje.

Pokud je potřeba, aby databáze rozuměla vloženým hodnotám, je nutné vytvořit vazbu na konkrétní datový typ databáze. Této vazby lze pak využít při ověření, zda je zadaná hodnota v souladu s konkrétním datovým typem.

Pro vytvoření vazby hodnoty se skutečným datovým typem lze použít například systémových tabulek databázového systému či použití sloupce, který popisuje pouze datové typy podporované databází. Zde závisí na konkrétním *SŘBD*[9] a jeho prostředcích, které je schopen v dané problematice poskytnout. Příkladem může být datový typ `regtype` systému PostgreSQL, kterým lze vyjádřit pouze podporované datové typy tohoto SŘBD, jako jsou např. `varchar`, `integer` či `boolean`.

Porozumění heterogenním datům na úrovni databáze má své opodstatnění, a to hlavně s ohledem na fakt, že vytvářená aplikace umožní vyhledávání v datovém stromu uzlů.

Pokud bychom se chovali ke všem hodnotám tak, jak jsou fyzicky uloženy, tj. jako k prostému textu, není možné využít operátorů větší, menší, apod. u hodnot, které reprezentují číslo či datum.

Alternativně lze uvážit situaci, kdy by vyhledání bylo provedeno klient-

[9]Systém Řízení Báze Dat je korektní označení databázových systémů.

Výpis kódu 4.8: Příznak listu bez korelovaného dotazu

```
SELECT category_id, name, parent,
  (lft + 1 = rgt) AS is_leaf FROM category;

 category_id |         name          | parent | is_leaf
-------------+-----------------------+--------+---------
           1 | Elektronika           |        | f
           2 | Televizory            |      1 | f
           3 | CRT                   |      2 | t
           4 | LCD                   |      2 | t
           5 | Plazmové              |      2 | t
           6 | Přenosná elektronika  |      1 | f
           7 | MP3 přehrávače        |      6 | f
           8 | Flashové              |      7 | t
           9 | CD přehrávače         |      6 | t
          10 | Radio přijímače       |      6 | t
(10 řádek)
```

Výpis kódu 4.9: Více typů hodnot v rámci jednoho sloupce

```
SELECT '123'::text
UNION
SELECT '123'::integer;

ERROR:  UNION types text and integer cannot be matched
```

skou částí aplikace, kde by si program přetypování vyřešil sám, bez potřeby, aby databáze rozuměla uloženým datům. Nelze ale opomenout důležitý fakt, že by takové vyhledávání vyžadovalo načtení celého datového stromu před samotným filtrováním, což je ale při možnosti značně zaplněné databáze velice nešikovné řešení.

4.3.1 Oddělení serveru a klienta

Otázka dobrého oddělení klienta a databáze byla již naznačena v kapitole 4.3. Jedním z důvodů přenášení funkčnosti směrem na databázový server může být např. výkonové hledisko, a to jak v podobě přenesení náročných operací na databázový server nebo omezení přenosu dat mezi klientem a databázovým úložištěm.

Důsledkem je odlehčení klientské aplikace. Příkladem toho může být prosté nastavení kaskádního mazání závislých záznamů. Díky tomu nemusí program řešit odstranění závislých záznamů, ale pouze vykoná požadované mazání a závislé mazání již obstará SŘBD samotný.

Zanesení funkčnosti do databáze zároveň usnadňuje případné vytvoření alternativního klienta.

Protože sympatizuji s přesunem některých funkcí směrem na databázový server, bude i výsledná aplikace podléhat této filosofii. Proto je důležité vybrat vhodnou databázovou technologii, která bude podporovat prostředky, umožňující tento postoj realizovat.

Databázový systém tedy musí podporovat uložené funkce a procedury, nejlépe pak spolu s podporou jazyka PL/SQL. S tím rovněž souvisí podpora triggerů, která je dnes již takřka samozřejmostí v databázových systémech. Neméně důležitou pak shledávám podporu tzv. *pohledů*, které umožní uložit komplexní výběrové dotazy na straně SŘBD. Pro zobrazení takového pohledu pak stačí minimální vyběrový dotaz na straně klientského programu.

4.4 Multiuživatelská databáze

V databázích, do kterých přistupuje naráz více uživatelů, přichází nutnost zavést určitá opatření, aby nedocházelo k nekonzistenci dat či vzájemnému přepisování záznamů uživateli.

Protože klientský program nejprve načte a pak zobrazuje data, mohou tato data být již v momentě jejich zobrazení uživateli neaktuální. Obecně se tedy řeší v takových databázích zejména problémy aktuálnosti dat a také otázka výhradního přístupu k datům (zejména pak při zápisu).

Jelikož klientská aplikace má načtena data ve své paměti, představme si situaci, kdy již nějakou dobu načtená data chceme upravit a uložit do databáze. Nyní je ale možné, že načtená data, která chceme upravovat již nejsou aktuální či dokonce byl tento záznam odstraněn z databáze úplně. Klientský program, který je multiuživatelským databázím uzpůsoben, by měl takovou kontrolu vždy provést.

Nyní se nabízí umožnit uživateli výhradní přístup k upravovanému záznamu. Většina současných databází obsahuje prostředky pro uzamčení záznamu nebo k uzamčení celé tabulky.

Zde nastává problém určit způsob zamykání. Uzamčení celé tabulky je pro programátora samozřejmě jednodušší a z hlediska dat eliminuje více rizik

než uzamčení jednotlivých záznamů. Stinnou stránkou zamykání celých tabulek je omezování práce ostatních uživatelů databáze, kteří se ani nemusejí pohybovat na stejných záznamech, jako uživatel, který tabulku uzamkl. Zamykání jednotlivých záznamů je tedy ohleduplnější vůči ostatním uživatelům databáze, kteří nejsou tolik blokováni.

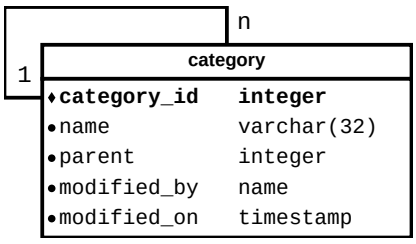
Tím tedy vzniká další požadavek na vybíraný SŘBD. Databázový systém musí podporovat nějakou formu zamykání na úrovni záznamů.

4.5 Sledování změn databáze

Nyní se zaměříme na problematiku monitorování změn prováděných v databázi. Zde existuje několik technik, které se v praxi používají. Úkolem je vybrat tu nejvhodnější pro účel vytvářené aplikace.

4.5.1 Auditorské záznamy

Pod pojmem auditorských záznamů se skrývá technika, která přidává k tabulkám, kde chceme alespoň elementárně sledovat změny, dva sloupce. První ze sloupců obsahuje uživatelské jméno autora změny záznamu, tj. uživatele, který záznam vložil či upravil (sledování mazání tato technika implicitně nepodporuje). Druhý sloupec pak obsahuje informaci o tom, kdy k této změně došlo. Ukázka takto upravené tabulky je vidět na obrázku 4.5.



Obrázek 4.5: Model tabulky kategorií s auditorskými záznamy

Tato technika je základem pro sledování změn, resp. pro určení osob odpovědných za aktuální podobu databáze. Technika sama o sobě nepodporuje sledování akcí mazání, což může být nevýhodou. Naopak výhodou je, že každá tabulka bude rozšířena právě o dva sloupce a nevzniká zde žádná další

tabulka. Tento přístup nenarušuje rovnoměrnost zaplnění tabulek databáze od původního návrhu její struktury.

Auditorské záznamy tvoří ideální základ pro některou z následujících technik, a to z důvodu, že není potřeba zaznamenávat i samotné vložení záznamu do databáze kvůli nalezení osoby odpovědné za vznik takového záznamu. Rovněž dojde k odstranění nutnosti duplicitního záznamu v záložních tabulkách, právě kvůli uložení informací o autorovi záznamu a času, kdy záznam vznikl. To jsou nesporné výhody v podobě redukce dat v databázi bez ztráty informací.

4.5.2 Vytvoření záložních tabulek

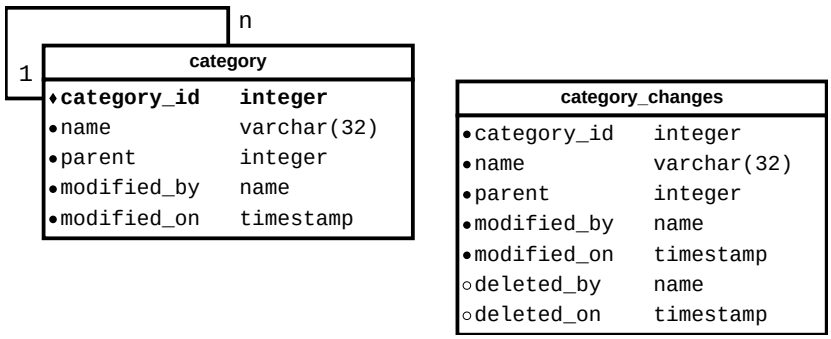
Pod tímto pojmem se skrývá technika, kdy každá z námi sledovaných tabulek (primárních) obsahuje svou kopii (sekundární), která je ještě rozšířená o další sloupce popisující, co se s daným záznamem stalo.

Výsledkem jsou tedy primární tabulky, které obsahují aktuální data, se kterými se manipuluje a tabulky sekundární, které obsahují historii změn jednotlivých záznamů. Tuto techniku je vhodné kombinovat s auditorskými záznamy v primárních tabulkách, stejně jako v ukázce na obrázku 4.6.

Pro udržení linie sledování změn konkrétního záznamu je nutné dodržet zásadu neměnit hodnotu primárního klíče. Jeho případná změna by pak byla zodpovědná za nemožnost přístupu k historii takového záznamu a uložené změny v sekundární tabulce byly bezcenné. Dokonce lze říci, že taková změna by způsobila nekonzistenci dat, protože sekundární tabulka obsahuje historii záznamu, který není přítomen v tabulce primární a zároveň nebylo zaznamenáno odstranění takového záznamu (v sekundární).

Tato technika má nesporně mnoho kladů. Prvním je, že opět zde nedochází k porušení rovnoměrnosti zaplnění tabulek oproti původnímu návrhu struktury databáze. Další výhodou je kompletní přehled změn jednotlivých záznamů.

Potíž ovšem nastává tehdy, chceme-li tyto změny zobrazit. Pokud, a je to nanejvýš pravděpodobné, návrh tabulek obsahuje závislosti na jiných tabulkách ve struktuře (a ty jsou rovněž zálohované), může taková rekonstrukce podoby záznamu být značně složitá právě proto, že sekundární tabulky nemohou obsahovat primární klíč u změněných záznamů. Případné napojení záznamů (viz `JOIN`) zde podléhá jak číslu záznamu, resp. hodnotě primárního klíče, který byl platný v primární tabulce, tak časovým razítkům popisujícím, od kdy byl záznam platný či kdy byl smazán (informace v pouze v sekundární



Obrázek 4.6: Model tabulky kategorií se zálohou vč. auditorských záznamů

tabulce).

Další nespornou nevýhodou může být velikost takové databáze. Vzniklá historie záznamů může představovat velké nároky na prostor. Stojí tedy za zvážení, zda je potřeba takto komplexní řešení.

Rovněž lze zmínit nevýhodu v podobě nutnosti změnit strukturu sekundární tabulky při změně primární. Na druhou stranu ke změnám struktury tabulky v průběhu životního cyklu aplikace asi dochází jen minimálně, a proto to nelze považovat za zásadní překážku v implementaci této techniky.

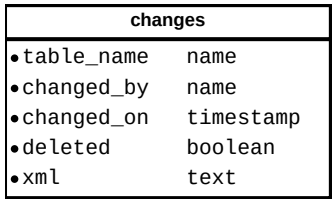
4.5.3 Jediná tabulka změn

Nyní přejdeme k technikám, které jsou postaveny na jediné tabulce, která monitoruje změny. Výhodou takového přístupu může být přehlednost struktury databáze.

Tuto techniku lze dále ještě rozlišit na dva možné způsoby, jak ji uchopit.

1. **Záloha celého záznamu v jediném poli záložní tabulky.**

   Principem této techniky je transformovat původní záznam, resp. jeho zálohu do formátu, který jej umožní uložit do jediného pole tabulky. Ideálním kandidátem může být formát XML, kterým lze záznam definovat včetně pojmenování sloupců jednotlivých hodnot. Ukázka takové tabulky je na obrázku 4.7. Obsah je pak demonstrován v tabulce 4.3. Alternativou může být příkaz `ROW`, který podporuje např. databáze PostgreSQL. Ten převede jediný záznam do textové podoby, které databáze rozumí a dokáže ji případně rekonstruovat na původní záznam.



Obrázek 4.7: Model záložní tabulky s XML

table_name	changed_by	changed_on	deleted	xml
category	user	2012-05-01 11:22:33	false	<category> <category_id>2</category_id> <name>Televizory</name> <parent>1</parent> </category>
category	user	2012-05-01 11:33:55	true	<category> <category_id>2</category_id> <name>Televize</name> <parent>1</parent> </category>

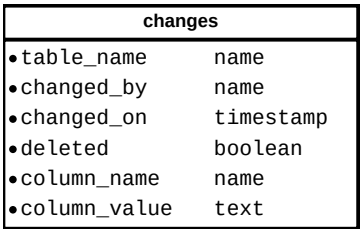
Tabulka 4.3: Ukázka zálohy pomocí XML

Nevýhodou těchto technik je opět rekonstrukce záznamu. Tím ale nemyslím problém napojení na jiné tabulky (ten samozřejmě přetrvává), ale záznam samotný. Jeho transformování z XML či prostého textu (viz příkaz `ROW`) přináší značnou režii.

2. **Ukládání záznamu po jednotlivých sloupcích.**

   Tato technika zavádí o něco komplexnější podobu zálohovací tabulky, jako například na obrázku 4.8. Principem je zde uložení informací původního záznamu v páru jméno sloupce a hodnota takového sloupce. Obsah je demonstrován v tabulce 4.4.

   Jinak řečeno, tento způsob vytváří v záložní tabulce pro každý sloupec zálohovaného záznamu jeden řádek. Tento postup pak může značně zvýšit tempo růstu záložní tabulky.



Obrázek 4.8: Model záložní tabulky - záloha po sloupcích

table_name	changed_by	changed_on	deleted	column_name	column_value
category	user	2012-05-01 11:22:33	false	category_id	1
category	user	2012-05-01 11:22:33	false	name	Televizory
category	user	2012-05-01 11:22:33	false	parent	2
category	user	2012-05-01 11:33:55	true	category_id	1
category	user	2012-05-01 11:33:55	true	name	Televize
category	user	2012-05-01 11:33:55	true	parent	2

Tabulka 4.4: Ukázka zálohy jednotlivých sloupců

Obecné nevýhody takových řešení jsou problematické rekonstrukce záznamů samotných, což je cena za uložení do společné tabulky.

4.5.4 Vytváření zpráv o změnách

Tento způsob je založen na generování zpráv nebo generování takových záznamů, které umožní vytvoření zpráv na úrovni aplikace. Stejně jako u předešlého způsobu, i zde se uvažuje jediná tabulka.

Databáze server, či v horším případě klientská aplikace, vygeneruje na základě změn v databázi vždy záznam (či zprávu), který bude popisovat právě uplatněnou změnu. V kombinaci s auditorskými záznamy stačí sledovat pouze aktualizace či odstranění záznamů.

Tato technika je relativně nenáročná na implementaci, ale poskytuje dostatek informací k naznačení, o jakou změnu šlo a zároveň nalezení osoby za změnu odpovědné, což je shodou okolností i účelem sledování změn ve vytvářené aplikaci.

4.5.5 Temporální databáze

Extrémním řešením problematiky sledování změn, ale zároveň nejúčinnějším, jsou tzv. *temporální databáze*[10]. Tyto databáze obsahují prostředky, které dovolují prohlížet stav databáze v různých časových bodech. Většinou jde o databáze, které jsou ve své podstatě navržené jako temporální. Rovněž některá komerční řešení konvenčních databází obsahují podporu pro temporální databáze, např. Oracle či IBM DB2.

V době, kdy jsem ještě neměl dobrou představu, jakou hloubku sledování změn zvolit ve vytvářené aplikaci, pokusil jsem se chování temporálních databází simulovat v konvenčním databázovém systému, a to v PostgreSQL.

Princip byl takový, že primární klíč (každé) tabulky bude složen jak z identifikace konkrétního záznamu (typicky identifikační číslo), tak bude současně doplněn časovou značkou, od kdy je záznam platný. Alternativou k časové znače by rovněž mohlo být číslo verze záznamu, které by s jeho aktualizováním rostlo. Je jasné, že takové tabulky je potřeba doplnit ještě o další sloupce, které ponesou další potřebné časové značky či příznaky (např. příznak smazání záznamu).

Databáze by za pomoci triggerů pracovala tak, že pokud by byl záznam upraven, došlo by zároveň znovu k vložení staré podoby záznamu. Toto řešení slibovalo tu výhodu, že veškerá data, a to jak aktivní, tak zálohy, budou uložena v jediné tabulce. Jinak řečeno, nedošlo by k navýšení počtu tabulek oproti původnímu návrhu databázové struktury.

I zde ale dochází ke značným problémům a nárokům na získání uložených dat, které by pravděpodobně musel řešit klientský program samotný. Za druhou nevýhodu lze považovat samotné složení primárního klíče a s tím související cizí klíče. Totiž každá aktualizace záznamu v jedné tabulce měla za následek aktualizaci záznamu v další tabulce, se kterou byla spojena, čímž tedy došlo i zde k aktualizaci stávajícího záznamu a vložení původního. To nastávalo díky kaskádní aktualizaci časové značky, která byla součástí primárního, resp. cizího klíče.

4.5.6 Další možnosti

Pro úplnost je potřeba uvést, že existují a rovněž byly uvažovány i další varianty. Například nevyužití auditorských záznamů v primárních tabulkách. Auditorské záznamy pak byly obsaženy (či suplovány) pouze v tabulkách

[10]Temporální databáze zohledňuje časové vlastnosti ukládaných dat.

záložních (chcete-li sekundárních). To by vyžadovalo vytvoření záložního záznamu i při vložení (do primární tabulky). Následné získání auditorských záznamů by vždy vyžadovalo nalezení odpovídajícího záznamu v tabulce sekundární. Z této situace pak plyne redundance dat v podobě přítomnosti nově vloženého záznamu jak v primární tabulce, tak uložení jeho podoby do tabulky záložní.

Další alternativa byla taková, že záložní tabulka bude obsahovat jak záznam původní, tak záznam nový, což se tedy týká zejména situace, kdy je záznam v primární tabulce aktualizován. Tuto alternativu lze pak uvážit zejména v případě jedné záložní tabulky, a to jak v podobě zálohy po jednotlivých sloupcích záznamu tak při využití jednotného uložení (viz XML, apod.).

Uložení původního i aktualizovaného najednou lze uvážit také v situaci, kdy jsou zaznamenávány pouze zprávy o změnách. To může uživatelům aplikace pomoci v lepší orientaci ve změnách databáze.

4.5.7 Volba způsobu sledování

Pro sledování změn, které probíhají v databázi bylo upřednostněno jednodušší řešení v podobě generování záznamů, ze kterých klientská aplikace sestaví uživateli srozumitelnou zprávu, popisující provedenou změnu.

Tuto techniku lze zajistit pomocí databázových triggerů, díky čemuž tuto činnost obstará SŘBD samotný a nepřenáší odpovědnost na klientskou část aplikace.

4.6 Výběr databázového systému

Jednou z klíčových otázek analýzy bylo určení vhodného databázového systému. Výběr lze rozdělit do dvou kategorií. První jsou komerční řešení a druhá pak alternativní, volně dostupná řešení.

Požadavky na SŘBD pro tento projekt jsou zejména výkonnost a podpora rekurzivních dotazů. Jak rovněž vyplývá z přecházejících kapitol, je žádoucí aby vybraný databázový nástroj podporoval jazyk PL/SQL, triggery a pohledy. Dále je kladen důraz na podporu BLOBů. Databázový systém by měl rovněž poskytnout prostředky pro výhradní přístup k záznamů, tj. nástroje pro jejich uzamčení.

4.6.1 Komerční řešení

Komerční řešení je vždy problematické, protože přináší, většinou nemalé, finanční náklady do projektu. Tento faktor může být drobnou překážkou, ale to především jen v projektech, které opět svým vytvořením a nasazením finanční prostředky opět přinesou a hlavně zmíněné náklady převýší.

Z tohoto důvodu byla komerční řešení nepřímo označena jako nevhodná či tomuto projektu nedostupná. I přesto považuji za dobré uvést zde některá z takových řešení a alespoň zjistit, zda by jejich případné využití mělo nějaký přínos.

Microsoft SQL Server

SQL server od společnosti Microsoft už ze své podstaty odkazuje na technologie právě této firmy, ale aktivně nejsou vyžadovány s vyjímkou operačního systému. Pro svůj běh požaduje databázový server systém z rodiny Microsoft Windows.

Vazba na technologie společnosti Microsoft může být výhodou v podobě možnosti propojení s jinými technologiemi právě této společnosti. Nevýhodou této vazby je pak zejména nutnost pořízení operačního systému, který je podporován.

Microsoft SQL Server obsahuje podporu procházení tabulek s hierarchickou strukturou pomocí CTE. Přítomna je rovněž podpora BLOBů do velikosti 2 GB. To jsou ale vlastnosti, kterými disponují i nekomerční řešení, a proto lze říci, že použití tohoto systému nepřináší žádnou přidanou hodnotu.

Oracle

Oracle je dnes jedním z nejrozšířenějších databázových řešení, využívaných v komerčním prostředí. Na rozdíl od konkurenčního Microsoft SQL Serveru není platformně závislý. Je tedy multiplatformní.

Oracle rovněž podporuje práci se stromovými strukturami, ale na rozdíl od ostatních databázových systémů obsahuje pro tuto činnost vlastní konstrukci `START WITH ... CONNECT BY`, která je ale svým principem podobná CTE. Přítomna je rovněž podpora datového typu BLOB, který zde umožňuje uložení až 4 GB dat.

Interbase

Interbase je multiplatformní databáze, vyvíjena původně společností Borland, dnes již společností Embarcadero. Jde o relační databázi, která svým původem odkazuje na technologie jako Delphi či *C++ Builder*. Na základech této databáze vznikla volně dostupná alternativa Firebird.

Databáze podporuje BLOBy, ale přítomnost podpory rekurzivních dotazů se mi nepodařilo zjistit. Pravděpodobně je tedy nepodporuje.

4.6.2 Volně dostupná řešení

Cílovou skupinou tedy byla volně dostupná či šiřitelná řešení, tedy databázové systémy, které nenesou břemeno v podobě finančních nákladů na jejich pořízení.

Firebird

Jak již bylo naznačeno výše, Firebird vznikl z databáze Interbase, jejíž zdrojové kódy byly kolem roku 2000 dočasně uvolněny. Po znovu-uzavření projektu Interbase jdou každý svou cestou.

Firebird nabízí podporu BLOBů stejným způsobem jako jeho předloha tedy Interbase. Na rozdíl od něj ale obsahuje podporu rekurzivních dotazů v podobě CTE.

Jako dnes již většina databázových systémů podporuje pohledy, uložené procedury i triggery. Pro svou společnou historii s Interbase má tato databáze dodnes blízko k technologiím společnosti Borland, dnes již Embarcadero, jako jsou např. vývojové prostředí Delphi. Uzamykání záznamů je zde rovněž podporováno.

MySQL

MySQL je velmi oblíbený databázový systémem mezi vývojáři webů, zejména je pak často kombinovaný se skriptovacím jazykem *PHP*.

MySQL podporuje BLOBy do velikosti 4 GB. Dále umožňuje vytvářet uložené procedury a funkce v jazyce PL/SQL. Rovněž přítomnost triggerů a podpora uzamykání záznamů je samozřejmostí.

V posledních verzích tohoto SŘBD přibyla i možnost vytvářet pohledy. Na rozdíl od konkurenčních řešení ale neobsahuje žádné prostředky pro rekurzivní průchod hierarchickými strukturami.

PostgreSQL

PostgreSQL je volně šiřitelný multiplatformní SŘBD se otevřeným zdrojovým kódem vycházející z projektu Ingres.

PostgreSQL nabízí podporu BLOBů a to dvěma způsoby. Jako BLOB lze využít datový typ `bytea`, který limituje obsah na 1 GB. Tento datový typ lze pak zcela standardně použít jako sloupec libovolné tabulky. Druhým způsobem je pak použití specifických funkcí, které uloží BLOB do systémových tabulek. Na takto uložený obsah se pak lze odkazovat prostřednictvím `oid`, které mu bylo systémem přiřazeno. Při uložení obsahu v systémových tabulkách je velikostní limit posunut na 2 GB.

Podpora triggerů, uložených procedur a funkcí je v tomto systému samozřejmostí. Pro jejich implementaci lze využit jazyka PL/SQL nebo přímo programovacího jazyka C, což ale není úplně triviální. PostgreSQL samozřejmě také dovoluje vytváření pohledů a uzamykání záznamů. Databázový systém rovněž implementuje CTE pro rekurzivní průchod hierarchickými strukturami.

4.6.3 Volba SŘBD

Přestože jsou uvážené SŘBD takřka rovnocenné, tabulka 4.5 jasně ukazuje, že databázový systém MySQL nenabízí podporu rekurzivních dotazů. Proto jej lze vyloučit z konečného výběru.

Funkce	**Firebird**	**MySQL**	**PostgreSQL**
BLOB	2 GB	4 GB	1 GB/2 GB
Rekurzivní dotaz	CTE	-	CTE
Pohledy	ano	ano	ano
Uložené procedury/funkce	ano	ano	ano
Triggery	ano	ano	ano
Zamykání záznamů	ano	ano	ano

Tabulka 4.5: Přímé srovnání databází

Do finále výběru databázového systému se tedy dostaly jak PostgreSQL tak Firebird. Oba systémy v dané problematice obsahují obdobné prostředky a proto nelze jednoznačně označit ten či onen za vhodnější.

Z tohoto důvodu nakonec o volbě SŘBD rozhodly mé osobní sympatie a zkušenosti s databázovým systémem PostgreSQL. Zejména znalost tohoto systému vč. jeho systémových tabulek může být v průběhu vývoje aplikace neopomenutelnou výhodou.

Na druhou stranu by případná volba Firebirdu, z hlediska kvality SŘBD, nebyla o nic horším řešením.

4.7 Vývojový nástroj

Další otázkou byla volba vhodného nástroje pro implementaci klientského programu, který bude viditelnou částí projektu pro běžného uživatele. Protože požadavky aplikace vyžadují grafické uživatelské rozhraní, je dobré zohlednit možnosti daného nástroje resp. jazyka v této oblasti.

Volba nástroje rovněž musí uvažovat výkonnost výsledné aplikace a hardwarové nároky na počítače, na kterých bude aplikace provozována. Takové stroje nejsou vždy úplně moderní.

4.7.1 Uvažované nástroje

Zde uvedený seznam nástrojů pro implementaci klientské části aplikace uvádí pouze prostředky, které by dle mého názoru mohli nabídnout zajímavé vlastnosti. Zároveň jde o nástroje, které nepředstavují finanční náklady na jejich pořízení. Nejde tedy o seznam všech řešení, které jsou dnes dostupná. Existence dobře použitelného *IDE*[11] je rovněž jedním z rozhodujících faktorů.

Microsoft Visual Studio

IDE Visual Studio představuje samo o sobě možnost výběru z více programovacích jazyků, jmenovitě tedy výběr z C++, C# nebo Basic. V tomto případě jsem ale uvažoval zejména jazyk C++, jehož překladač od společnosti Microsoft je považován za jeden z nejlepších. Visual Studio je volně dostupné v tzv. *Express* edici. Ostatní edice jsou již placené.

Nevýhodou tohoto nástroje je vazba na technologie společnosti Microsoft a na operační systémy Microsoft Windows.

[11] Integrated Development Environment je anglickým označením pro vývojové prostředí.

C++ a Qt

Kombinace programovacího jazyka C++ a grafického frameworku Qt poskytuje kvalitní základ pro vytvoření multiplatformních aplikací s kvalitním *GUI*[12]. Framework Qt je dnes vlastněn a vyvíjen společností Nokia, která jej využívá i pro vytváření aplikací pro mobilní zařízení.

Spolu s frameworkem Qt je vyvíjeno i kvalitní IDE *QtCreator*. Dříve frameworku využívala i společnost Borland v produktech Delphi a Kylix, kde Qt tvořilo základ multiplatformní knihovny komponent *CLX*.

Java

Java je pro výslednou aplikaci asi nejméně vhodným prostředkem a to zejména proto, že jde o interpretované prostředí, se kterým souvisí zvýšená hardwarová náročnost.

Vytvoření GUI v Javě není zcela triviálním úkolem, a protože je požadováno relativně komplexní rozhraní, mohlo by použití tohoto nástroje být velmi problematické.

Na rozdíl o ostatních nástrojů, nabízí Java pomocí JDBC asi nejlepší databázovou konektivitu. Použití JDBC by pak umožnilo vzniku univerzálního ORM bez závislosti na konkrétní databázové technologie.

Delphi (Lazarus)

Delphi, dříve vyvíjené společností Borland, jsou velmi dobrým nástrojem pro vytvoření aplikací s rozsáhlým grafickým uživatelským rozhraním. Dnes jsou Delphi vyvíjeny společností Embarcadero.

Stejně jako v době největší slávy tohoto IDE, i dnes nabízí Delphi jeden z neefektivnějších způsobů, jak rychle a kvalitně vytvořit aplikaci s GUI. Rovněž přítomný překladač programovacího jazyka *Object Pascalu* produkuje výkonný strojový kód. V Delphi lze dnes vytvářet zejména pro platformu Microsoft Windows, a to jak ve 32-bitové, tak v 64-bitové verzi. V nedávné době přibyla podpora operačních systémů společnosti Apple, tj. *iOS* a *Mac OS X*. Podpora Linuxu pak byla přislíbena do jedné z příštích verzí.

Nevýhodou Delphi v dnešní době je to, že není k dispozici žádná volně dostupná verze tohoto produktu, což může být rovněž problémem při snaze více zpopularizovat toto IDE.

[12]Graphic User Interface je anglický výraz pro grafické uživatelské rozhraní.

Nechceme-li investovat do vývojového prostředí Delphi, přichází možnost volby na IDE Lazarus, které je volně šiřitelnou alternativou Delphi. Lazarus je multiplatformní vývojové prostředí využívající překladače jazyka Free Pascal, který je obdobou překladače Object Pascalu v Delphi.

4.7.2 Volba nástroje

Nakonec jsem zvolil, pro někoho možná rozporuplné, řešení v podobě IDE Delphi, konkrétně *Turbo Delphi 2006*, které bylo do nedávna volně dostupné při zaregistrování. Efektivita tohoto nástroje při tvorbě s GUI je dodnes nezpochybnitelná. Volba byla podpořena skutečností, že i původní projekt byl vytvořen pomocí tohoto IDE a v neposlední řadě byly brány v potaz mé sympatie a zkušenosti s tímto nástrojem.

Protože je velice pravděpodobné, že tento projekt bude potřeba nějakým způsobem udržovat či rozšiřovat, nemůže tedy výběr Delphi představovat následné náklady. V takovém případě existuje již zmíněné řešení v podobě volně šiřitelného IDE Lazarus, do kterého lze projekt s minimálním úsilím převést. Alternativně lze využít licencí na Delphi 2007, které katedra doposud vlastní.

Kapitola 5

Implementace databáze

Předmětem této kapitoly je použití zjištěných poznatků a závěrů v návrhu a implementaci databáze. Pro úplnost bych rád uvedl, že pro navrženou strukturu bude používán název *TreeBase*, který vychází ze jmen spojovaných s původním projektem.

Veškeré objekty budou umístěny ve vlastním databázovém schématu `tb`. To samozřejmě neplatí pro skupiny a uživatele, kteří (alespoň v PostgreSQL) nenáleží konkrétní databázi, ale jsou platné pro celý databázový server. Model vzniklé databáze je zobrazen v příloze A.

5.1 Tabulky databáze

Následující odstavce popisují vzniklé databázové tabulky, jejich atributy a také význam, který představují v navržené databázové struktuře.

5.1.1 Tabulka base_types

Prvním objektem databáze je tabulka, která bude obsahovat soupis podporovaných základních typů. Ve své podstatě jde o entitu, která vytvoří pomyslný most mezi uvažovanými základními datovými typy v navrhovaném programu a skutečnými datovými typy databázového systému PostgreSQL. Tak vznikne pevně definovaná množina základních datových typů, ze které nelze v aplikaci vybočit. Tato množina bude neměnná, a to alespoň v rámci životního cyklu jedné verze aplikace.

Sloupec	Typ	Popis
base_type_id	int2	Identifikační číslo základního typu dat
base_type_name	varchar(32)	Název základního datového typu
help	varchar(256)	Uživateli srozumitelná nápověda pro odpovídající hodnotu
pg_type	regtype	Typ SŘBD PostgreSQL

Tabulka 5.1: Sloupce tabulky base_types

Vzniklá vazba se skutečnými datovými typy SŘBD vytvoří uživateli seznam srozumitelně nazvaných základních datových typů. Tím bude umožněno manipulovat v aplikaci se základním datovým typem např. datum, ale jeho význam na úrovni databáze bude popsán datovým typem `date`. Díky tomu SŘBD pozná, jakým skutečným datovým typem případnou hodnotu vyjádřit pro další operace.

Strukturu databázové tabulky `base_types` prezentuje tabulka 5.1. Klíčový sloupec zde je zejména `pg_type`, jež je speciálního typu `regtype`, kterým lze popisovat pouze platné typy v PostgreSQL.

5.1.2 Tabulka user_types

Další entitou databáze je tabulka uživateli definovaných datových typů. Obsah této tabulky bude již libovolně měněn, a proto tabulka obsahuje sloupce pro auditorské informace. Odkazem do tabulky základních typů je určeno, jaký formát hodnoty daný uživatelský typ podporuje.

Soupis sloupců této entity je uveden v tabulce 5.2. Na první pohled je tedy patrné (pole `base_type`), že uživatelem definovaný typ vychází z některého ze základních. Tím je opět tvořena vazba na reálný typ PostgreSQL. Tabulka rovněž prvně zavádí auditorské informace a to v podobě sloupců `modified_by` a `modified_on`.

5.1.3 Tabulka relations

Tabulka `relations`, jejíž struktura je popsána tabulkou 5.3, slouží k definování povolených posloupností uživateli definovaných datových typů tj. popisuje předepsanou strukturu datového stromu. Tato struktura ale není

Sloupec	Typ	Popis
`user_type_id`	`serial`	Identifikační číslo uživatelem definovaného typu dat
`user_type_name`	`varchar(64)`	Pojmenování uživatelem definovaného typu dat
`base_type`	`int2`	Odkaz na použitý základní typ
`units`	`varchar(16)`	Nastavené jednotky
`modified_by`	`name`	Uživatelské jméno autora současné podoby záznamu
`modified_on`	`timestamp`	Časová značka vzniku aktuální podoby záznamu

Tabulka 5.2: Sloupce tabulky user_types

aktivně vynucována v podobě např. triggerů. Důvod, proč tomu tak je, je zřejmý. Pokud bychom vynucovali strukturu, byla by následná úprava takového předpisu téměř nemožná, protože stávající podoba stromu by již nevyhovovala definované. Toto byl důvod, proč je tato předepsaná struktura vynucována pouze pasivně.

Sloupce `relation_id` a `relation_parent` tvoří hierarchii (strom) podle dle kapitoly 4.2. Kořenový vztah je zde definován tak, že jsou uvedená pole shodná, tj. záznam odkazuje sám na sebe. Klíčovým polem je zde `user_type`, které vazbou s entitou `user_types` vytváří ze záznamů této tabulky strukturu uživatelských typů.

Se vzniklou hierarchií definovaných typů souvisí sloupce `min_count` a `max_count`. Ty určují kolik poduzlů uživatelského typu, jež je určen aktuálním záznamem vztahu, může mít uzel, který je typu (a umístění) reprezentovaného záznamem nadřazeného vztahu (viz `relation_parent`). Tedy například uzel **rozměr** musí obsahovat právě jeden uzel typu **délka** a **šířka**, ale maximálně jeden typu **výška**.

Sloupce `min_text_count` a `max_text_count` definují počet textů, kterými může odpovídající uzel disponovat. Obdobně pak pole `min_file_count` a `max_file_count` určují toto omezení pro soubory.

Zajímavostí tabulky jsou tzv. *počítaná pole*, která obsahují dopředu vypočítané hodnoty jež souvisí se záznamem samotným. Takové informace se obvykle počítají přímo ve výběrovém dotazu (nebo pohledu) a neukládají

Sloupec	Typ	Popis
relation_id	serial	Identifikační číslo vztahu
relation_parent	int4	Odkaz na nadřazený vztah
user_type	int4	Odkaz na uživatelem definovaný datový typ
min_count	int4	Minimální počet výskytů typu
max_count	int4	Maximální počet výskytů typu
min_text_count	int4	Minimální počet textů
max_text_count	int4	Maximální počet textů
min_file_count	int4	Minimální počet souborů
max_file_count	int4	Maximální počet souborů
has_sub_relations	boolean	Příznak zda má vztah podřízené vztahy (není listem)
typed_path	text	Typová cesta odpovídající poloze vztahu
modified_by	name	Uživatelské jméno autora záznamu vztahu
modified_on	timestamp	Časová značka vzniku aktuální podoby vztahu

Tabulka 5.3: Sloupce tabulky relations

se v tabulkách. Na druhou stranu předpočítání některých informací již při vložení nebo úpravě záznamu může citelně snížit nároky na tyto informace při jejich výběru (viz `SELECT`). A to je důvod vzniku sloupců `has_sub_relations` a `typed_path`.

Druhý zmíněny sloupec je jedním z nejdůležitějších polí tabulky. Obsahuje tzv. *typovou cestu*, tedy textem reprezentovanou posloupnost identifikačních čísel (oddělených tečkou) definovaných typů, které konkrétní vztah odpovídá. Tento sloupec má rovněž příznak `UNIQUE`, čímž je zaručena unikátnost definice vztahu. Toto pole hraje zásadní roli při ověření struktury samotného datového stromu.

5.1.4 Tabulka nodes

Entita `nodes` (viz tabulka 5.4) je tabulkou obsahující samotné uzly datového stromu. Jde tedy o první databázovou tabulku, jež obsahuje skutečné informace, kvůli kterým aplikace vznikla.

Sloupec	Typ	Popis
`node_id`	`serial`	Identifikační číslo uzlu
`node_parent`	`int4`	Odkaz na nadřazený uzel stromu
`user_type`	`int4`	Odkaz na uživatelem definovaný typ
`node_value`	`text`	Vlastní hodnota uzlu
`has_sub_nodes`	`boolean`	Příznak zda uzel obsahuje podřízené uzly (není list)
`path`	`text`	Uzlová cesta
`typed_path`	`text`	Typová cesta
`modified_by`	`name`	Uživatelské jméno autora aktuálního záznamu
`modified_on`	`timestamp`	Časová značka vzniku aktuální podoby záznamu

Tabulka 5.4: Sloupce tabulky nodes

Shodně s tabulkou `relations`, i v této je vytvořena hierarchie dvojicí sloupců, a tedy `node_id` a `node_parent`. Následně je uzel definován zejména typem (pole `user_type`) a svou vlastní hodnotou (sloupec `node_value`).

Rovněž i zde jsou počítaná pole. Typová cesta v tomto případě představuje posloupnost typů, resp. typů uzlů, jež leží v cestě od kořene stromu k danému uzlu. Typová cesta slouží jak ke kontrole struktury, tak je jí užito při vyhledávání (filtrování).

Dalším klíčovým polem, které přispívá svou přítomností k implementaci filtrování je sloupec `path`. Jde tedy o tzv. *uzlovou cestu*, která je zde obdobou typové, s tím rozdílem, že jde tentokrát o posloupnost identifikačních čísel samotných uzlů. Postranním efektem tohoto sloupce je možnost urychlení při manipulaci s větvemi. Stačí vyjít ze skutečnosti, že všechny uzly jedné větve mají společný prefix právě uzlové cesty.

5.1.5 Tabulka texts

Tabulka `texts`, jež obsahuje sloupce popsané tabulkou 5.5, ukládá doprovodné texty k jednotlivým uzlům datového stromu. Příslušnost textu k uzlu definuje sloupec `node`.

Sloupec	Typ	Popis
`text_id`	`serial`	Identifikační číslo textu
`text_name`	`varchar(64)`	Název doprovodného textu
`node`	`int4`	Odkaz na příslušný uzel
`text_value`	`text`	Vlastní text
`modified_by`	`name`	Uživatelské jméno autora záznamu textu
`modified_on`	`timestamp`	Časová značka vzniku aktuální podoby záznamu

Tabulka 5.5: Sloupce tabulky texts

5.1.6 Tabulka files

Entita, ukládající přiložené soubory, je reprezentována tabulkou `files` a svou strukturou (viz tabulka 5.6) je velice podobná tabulce doprovodných textů. Důležitým polem této tabulky je sloupec `pg_lagreobject`, pojmenovaný podle systémové tabulky, která obsahuje uložené BLOBy.

5.1.7 Tabulka log

Databázová tabulka nazvaná `log` shromažďuje záznamy o jednotlivých změnách v uložených datech. Jde o jakousi „černou skříňku“, která obsahuje informace, jež popisují, co se stalo s uloženými daty. Každá ze sledovaných tabulek (tj. tabulky uživatelských typů, vztahů, uzlů datového stromu, doprovodných textů a uložených souborů) obsahuje odpovídající trigger, který v reakci na změnu generuje záznam o události právě do této tabulky.

Tabulka je tvořena ze sloupců, uvedených v tabulce 5.7. Klíčovými poli, které obsahují (uživateli srozumitelné) informace popisující změnu, jsou sloupce `before_rec`, `before_path`, `after_rec` a `after_path`. Zda zaznamenaná

Sloupec	Typ	Popis
file_id	serial	Identifikační číslo záznamu souboru
file_name	varchar(64)	Název (fyzický) přiloženého souboru
node	int4	Odkaz na příslušný uzel
description	text	Popis souboru
pg_largeobject	oid	Identifikace uloženého BLOBu
modified_by	name	Uživatelské jméno autora aktuálního záznamu
modified_on	timestamp	Časová značka vzniku aktuální podoby záznamu

Tabulka 5.6: Sloupce tabulky files

událost popisuje aktualizaci záznamu nebo jeho odstranění, lze určit z pole `after_rec`. Je-li prázdné (tedy `NULL`), jde o operaci mazání.

Uvedené sloupce pak nabízejí dostatečnou podporu pro získaní představy o provedené změně a zároveň tvoří dobrý základ pro generování zpráv o změnách na úrovni klientské aplikace.

5.1.8 Tabulka params

Poslední entitou navržené databázové struktury je tabulka `params`, která slouží jen jako uložiště případných parametrů a jejich hodnot, jež upravují chování databáze samotné.

Obsah této tabulky (viz tabulka 5.8) je tedy vždy tvořen párem názvu a hodnoty parametru. Pro univerzální uložení hodnoty parametru je využito datového typu `text`.

5.2 Funkce, procedury a triggery

Dalšími databázovými objekty jsou uložené funkce, procedury a triggery (vč. jejich funkcí), které rozšiřují základní funkčnost vytvořené databáze z pouhého uložiště informací na tzv. *aktivní databázi*.

Sloupec	Typ	Popis
time_of_change	serial	Časová značka vzniku události
author	name	Uživatel odpovědný za vzniklou událost
table_name	int4	Název tabulky, s níž změna souvisí
before_rec	text	Textová podoba záznamu před proběhnuvší události (povinné)
before_path	text	Textová reprezentace související cesty záznamu před události
after_rec	text	Textová reprezentace záznamu po změně
after_path	text	Textová reprezentace související cesty záznamu po změně

Tabulka 5.7: Sloupce tabulky log

Sloupec	Typ	Popis
param_name	varchar(64)	Název parametru
param_value	text	Vlastní hodnota parametru

Tabulka 5.8: Sloupce tabulky params

5.2.1 Trigger log_clean

Funkce triggeru `log_clean()` udržuje požadovanou délku historie změn. Za její spuštění je odpovědný trigger `log_clean_trg` tabulky `log` a to v době `AFTER INSERT`.

Po jejím spuštění dojde k načtení parametru `log_max_days` z tabulky `params`, který udává maximální stáří záznamu o změně v řádu dnů. Následně dochází k mazání všech záznamů (této tabulky), které jsou starší než je dovoleno.

Funkce je spouštěna v režimu `SECURITY DEFINER`, což znamená, že je spuštěna s právy svého autora. Tím odpadá nutnost komplikovaného definování přístupových práv této tabulce.

5.2.2 Trigger user_types_check

Funkce triggeru `user_types_check()` nastavuje odpovídající auditorské informace vkládaným a zejména aktualizovaným záznamům tabulky s uživatelskými typy. Vykonání funkce obstará trigger `user_types_check_trg` zmíněné tabulky při `BEFORE INSERT OR UPDATE`.

5.2.3 Funkce get_user_type_caption

Funkce `get_user_type_caption()`, definovaná dle výpisu 5.1, generuje a vrací textový popisek, reprezentující uživatelem definovaný datový typ. Parametry představují jméno uživatelského typu dat a případné jednotky.

Výpis kódu 5.1: Deklarace funkce get_user_type_caption

```
CREATE OR REPLACE FUNCTION tb.get_user_type_caption(
    varchar(64), varchar(16)) RETURNS text AS ...
```

5.2.4 Trigger user_types_log

Funkce triggeru `user_types_log()` sleduje změny v tabulce uživatelských typů a generuje záznamy o takových událostech. Původní podoba typu (a změněný v případě aktualizace) je reprezentována pomocí popisku, generovaného výše zmíněnou funkcí `get_user_type_caption()` (viz kapitola 5.2.3).

O vykonání funkce se stará trigger `user_types_log_trg` patřící tabulce `user_types` v situaci `AFTER UPDATE OR DELETE`. Spuštění je opět provedeno s právy autora (viz `SECURITY DEFINER`).

5.2.5 Trigger relations_check

Funkce triggeru `relations_check()`, za jejíž spuštění je odpovědný trigger `relations_check_trg`, nastavuje hodnoty auditorských a počítaných (viz `has_sub_relations` a `typed_path`) polí, jejichž význam je popsán v kapitole 5.1.3. Vykonání je iniciováno při `BEFORE INSERT OR UPDATE` tabulky `relations`.

Funkce rovněž provádí detekci případné vznikající smyčky. Při nalezení smyčky vyvolá chybu.

5.2.6 Trigger relations_apply

Funkce triggeru `relations_apply()` reaguje na změny tabulky vztahů při `AFTER INSERT OR UPDATE OR DELETE` a provádí aktualizaci počítaných polí záznamů, které souvisejí s provedenou změnou. Je-li např. vztahu, který je listem, přidán podvztah, musí být upraven jeho příznak `has_sub_relations`. Podobně při změně uživatelského typu některého ze vztahů, musí dojít k úpravě typové cesty všech zbývajících záznamů vztahů v dané větvi stromu. Trigger `relations_apply_trg` provádí spuštění této funkce.

5.2.7 Trigger relations_log

Funkce triggeru `relations_log()` sleduje proběhnuvší změny tabulky se vztahy a generuje záznamy o takových událostech do tabulky `log`. Vztah (jeho změna) je popsán funkcí `get_user_type_caption()` a to vč. cesty (viz pole `before_path` a `after_path` tabulky `log`).

Shodně jako všechny funkce, jejiž jméno končí „`_log`“, běží s příznakem `SECURITY DEFINER`. Za spuštění odpovídá trigger `relations_log_trg` při `AFTER UPDATE OR DELETE`.

5.2.8 Trigger nodes_check

Funkce triggeru `nodes_check()`, spouštěná triggerem `nodes_check_trg`, nastavuje hodnoty auditorských a počítaných polí tabulky `nodes`. Shodně jako `relations_check()` (viz kapitola 5.2.5) rovněž detekuje vznikající smyčky. Funkce je vykonána při `BEFORE INSERT OR UPDATE`.

5.2.9 Trigger nodes_apply

Funkce triggeru `nodes_apply()` reaguje na (proběhlé) změny tabulky `nodes` a upravuje hodnoty počítaných polí souvisejících záznamů, podobně jak je tomu u funkce `relations_apply()` (viz kapitola 5.2.6).

Za spuštění při `AFTER INSERT OR UPDATE OR DELETE` zodpovídá trigger `nodes_apply_trg`.

5.2.10 Funkce is_node_value_valid

Funkce `is_node_value_valid()`, deklarovaná dle výpisu 5.2, kontroluje zda je zadaná hodnota v souladu s konkrétním datovým typem PostgreSQL.

Přijímanými parametry jsou vlastní hodnota uzlu a platný datový typ. Funkce vracím zda je hodnota vyhovující.

Výpis kódu 5.2: Deklarace funkce is_node_value_valid

```
CREATE OR REPLACE FUNCTION tb.is_node_value_valid(
    text, regtype) RETURNS boolean AS ...
```

5.2.11 Funkce get_node_caption

Funkce `get_node_caption()`, jejíž deklarace odpovídá výpisu 5.3, generuje a vrací textovou reprezentaci uzlu. Parametry, které přijímá jsou: platný typ PostgreSQL, jméno uživatelem definovaného typu, případné jednotky a vlastní hodnota uzlu.

Výpis kódu 5.3: Deklarace funkce get_node_caption

```
CREATE OR REPLACE FUNCTION tb.get_node_caption(regtype,
    varchar(64), varchar(16), text) RETURNS text AS ...
```

5.2.12 Trigger nodes_log

Funkce triggeru `nodes_log()`, spuštěna triggerem `nodes_log_trg`, monitoruje změny tabulky `nodes` a generuje o nich záznamy. Funkce je prováděna při `AFTER UPDATE OR DELETE` s příznakem `SECURITY DEFINER`.

Vzniklý záznam v tabulce `log` reprezentuje uzel popiskem, generovaným funkcí `get_node_caption()` (viz kapitola 5.2.11).

5.2.13 Trigger texts_check

Funkce triggeru `texts_check()`, spouštěná triggerem `texts_check_trg` tabulky `texts` při `BEFORE INSERT OR UPDATE`, nastavuje auditorské informace záznamu doprovodného textu.

5.2.14 Trigger texts_log

Funkce triggeru `texts_log()` sleduje změny tabulky `texts` a generuje záznamy o takových událostech (změnách). Doprovodný text, uložený v záznamu o změně, je reprezentován svým názvem a cestou k uzlu, kterému náležel (či náleží). Pro sestavení takové cesty je použito funkce `get_node_caption()` (viz kapitola 5.2.11). Funkce je spouštěna triggrem `texts_log_trg` při `AFTER UPDATE OR DELETE` s příznakem `SECURITY DEFINER`.

5.2.15 Trigger files_check

Funkce triggeru `files_check()` nastavuje hodnoty auditorských polí záznamu tabulky s přiloženými soubory. Spuštění provádí trigger `files_check_trg` v době `BEFORE INSERT OR UPDATE`.

5.2.16 Trigger files_apply

Funkce triggeru `files_apply()`, spouštěná triggerem `files_apply_trg` tabulky `files` při `AFTER INSERT OR UPDATE`, upravuje práva vzniknuvšího BLOBu. Dojde k nastavení práva číst, tj. `SELECT` skupině uživatelů, kteří mají přístup pro čtení této databáze (viz kapitola 5.3).

5.2.17 Trigger files_clean

Funkce triggeru `files_clean()` odstraní nevyužívané BLOBy z databáze. Při odstranění záznamu souboru nebo při aktualizaci v podobě nahrazení BLOBu jiným, dojde k uvolnění nevyužívaného obsahu pomocí funkce k tomu určené `lo_unlink()`.

Spuštění funkce vykoná trigger `files_clean_trg` tabulky s přiloženými soubory v době `AFTER UPDATE OR DELETE` a příznakem `SECURITY DEFINER`.

5.2.18 Trigger files_log

Funkce triggeru `files_log()` je odpovědná za sledování změn tabulky souborů a generování záznamů o takových událostech. Jde o obdobu funkce `texts_log()` (viz kapitola 5.2.14).

5.3 Uživatelé a skupiny

Pro administraci uživatelů a jejich práv jsem zvolil vytvoření několika skupin, které představují jakési šablony oprávnění, jež jsou jednotlivými volbami oprávnění. Tento postup usnadňuje následné udílení přístupů, resp. oprávnění jednotlivým uživatelům. Stačí tedy, aby se uživatel stal členem příslušné skupiny. V PostgreSQL je rozdíl mezi definicí skupiny a uživatele pouze v příznaku `LOGIN`, který přísluší pouze uživatelům.

Pro realizaci tohoto je zapotřebí, aby uživatelé měli příznak `INHERIT`, což umožní dědit práva skupin, jichž je uživatel členem (pouze přímým). K nastavení členství resp. jeho zrušení slouží příkazy `GRANT` resp. `REVOKE` (viz výpis 5.4).

Výpis kódu 5.4: Nastavení resp. zrušení členství

```
GRANT group_name TO user_name;
REVOKE group_name FROM user_name;
```

Řešení pomocí skupin přináší nespornou výhodu a zpřehledňuje tak udělování oprávnění jednotlivým uživatelům. Stačí tedy důsledně definovat práva vytvořeným skupinám a následně již jen udělovat uživatelům členství v těchto skupinách.

5.3.1 Skupina tb_can_read

Tato skupina představuje uživatele, kteří smí pouze číst uložená data. Neposkytuje tedy žádné možnosti úprav. Zároveň tato skupina nemá možnost prohlížet změny, tedy oprávnění `SELECT` v tabulce `log`. Právo nahlížení do historie změn databáze představuje jiná skupina. Skupina má nadefinována práva číst veškeré databázové tabulky či definované pohledy.

5.3.2 Skupina tb_can_edit_data

Členové této skupiny mohou přidávat a měnit uložená data. Jde tedy o oprávnění `INSERT` a `UPDATE` pro tabulky obsahující vlastní uložená data, nikoliv však definice. Jmenovitě jsou to tabulky `nodes`, `texts` a `files`. Zároveň jsou této skupině udělena práva `USAGE` nad sekvencemi zmíněných tabulek.

5.3.3 Skupina tb_can_edit_def

Skupina je obdobou `tb_can_edit_data` (viz kapitola 5.3.2) pro databázové tabulky `user_types` a `relations`.

5.3.4 Skupina tb_can_del_data

Skupina nazvaná `tb_can_del_data` sdružuje uživatele, kteří mohou mazat uložená data. Jde tedy o udělení oprávnění `DELETE` pro tabulky `nodes`, `texts` a `files`.

5.3.5 Skupina tb_can_del_def

Jde o obdobu skupiny `tb_can_del_data` (viz kapitola 5.3.4) pro tabulky `user_types` a `relations`.

5.3.6 Skupina tb_can_log

Skupina `tb_can_log` zprostředkovává svým členům oprávnění `SELECT` pro tabulku `log`. Členové skupiny mohou nahlížet do historie změn uložených dat.

5.3.7 Uživatel tb_admin

Součástí vytvořené struktury databáze je rovněž uživatel `tb_admin`, který představuje účet administrátora. Zároveň je vlastníkem vytvořené struktury tabulek a odpovědný za vytvoření skupin uživatelů.

5.4 Definované pohledy

Následující pohledy vznikly kvůli zjednodušení výběrových dotazů v klientské části programu. Bylo snahou minimalizovat počet sloupů jednotlivých pohledů, kvůli redukci přenášených dat.

5.4.1 Pohled users_viw

Pohled `users_viw` (viz tabulka 5.9) zobrazuje seznam veškerých uživatelů databázového serveru PostgreSQL. Výběr tvoří pouze uživatelé s příznakem

`LOGIN`, tedy ti, kteří se mohou přihlásit (není skupinou viz kapitola 5.3).

Sloupec	Typ	Popis
`rolname`	`name`	Obsahuje uživatelské jméno databázového uživatele
`description`	`text`	Komentář databázového objektu uživatele. Obsahuje dodatečné informace o uživateli
`tb_can_read`	`boolean`	Příznak členství ve skupině uživatelů, kteří smí číst
`tb_can_edit_data`	`boolean`	Příznak členství ve skupině uživatelů, kteří mohou upravovat data
`tb_can_edit_def`	`boolean`	Příznak členství ve skupině uživatelů, kteří mohou upravovat definice
`tb_can_del_data`	`boolean`	Příznak členství ve skupině uživatelů, kteří mohou mazat data
`tb_can_del_def`	`boolean`	Příznak členství ve skupině uživatelů, kteří mohou mazat definice
`tb_can_log`	`boolean`	Příznak členství ve skupině uživatelů, kteří mají přístup k historii změn

Tabulka 5.9: Sloupce pohledu user_viw

Tento pohled je jediným, který má přístupové právo `SELECT` nastaveno pseudo-skupině `public`, která v SŘBD PostgreSQL reprezentuje všechny uživatele. Jinými slovy, je zobrazení tohoto pohledu povoleno všem uživatelům.

5.4.2 Pohled base_types_viw

Pohled `base_types_viw` zobrazuje seznam podporovaných základních typů v minimálním zobrazení (viz tabulka 5.10) požadovaném klientskou aplikací.

Sloupec	Typ	Popis
base_type_id	int2	Identifikační číslo základního datového typu
base_type_name	varchar(32)	Název základního typu dat

Tabulka 5.10: Sloupce pohledu base_types_viw

5.4.3 Pohled user_types_viw

Pohled user_types_viw zobrazuje seznam uživatelských datových typů včetně podpůrných sloupců z tabulky základních typů. Jednotlivá pole tohoto pohledu popisuje tabulka 5.11.

Sloupec	Typ	Popis
user_type_id	int4	Identifikační číslo uživatelem definovaného datového typu
user_type_name	varchar(64)	Název uživatelského typu dat
base_type	int2	Odkaz do tabulky základních datových typů
units	varchar(16)	Nastavené jednotky
modified_by	name	Uživatelské jméno autora aktuální podoby záznamu
modified_on	timestamp	Datum vzniku aktuální podoby záznamu
base_type_name	varchar(32)	Název odpovídajícího základního typu
help	varchar(256)	Nápověda pro formát hodnoty
pg_type	regtype	Datový typ PostgreSQL
pg_type_oid	oid	Datový typ PostgreSQL reprezentovaný oid

Tabulka 5.11: Sloupce pohledu user_types_viw

5.4.4 Pohled relations_viw

Pohled `relations_viw` zobrazuje (dle tabulky 5.12) definovanou strukturu vztahů uživatelských datových typů.

Sloupec	Typ	Popis
`relation_id`	`int4`	Identifikační číslo vztahu
`relation_parent`	`int4`	Nadřazený vztah
`user_type`	`int4`	Odkaz na uživatelský datový typ
`min_count`	`int4`	Minimální výskyt typu
`max_count`	`int4`	Maximální výskyt typu
`min_text_count`	`int4`	Minimální počet vlastněných doprovodných textů
`max_text_count`	`int4`	Maximální počet vlastněných doprovodných textů
`min_file_count`	`int4`	Minimální počet přiložených souborů
`max_file_count`	`int4`	Maximální počet přiložených souborů
`modified_by`	`name`	Uživatelské jméno autora současné podoby záznamu
`modified_on`	`timestamp`	Časová značka vzniku aktuální podoby záznamu
`has_sub_relations`	`boolean`	Příznak existence podvztahů, tj. vztah není listem
`user_type_name`	`varchar(64)`	Název datového typu definovaného uživatelem
`units`	`varchar(16)`	Nastavené jednotky

Tabulka 5.12: Sloupce pohledu relations_viw

5.4.5 Pohled nodes_viw

Základním pohledem, používaným klientskou aplikací při zobrazování datového stromu, je `nodes_viw`. Zobrazuje veškeré uložené uzly. Výčet sloupců tohoto pohledu obsahuje tabulka 5.13.

Sloupec	Typ	Popis
node_id	int4	Identifikační číslo uzlu
node_parent	int4	Nadřazený uzel
user_type	int4	Odkaz na uživatelem definovaný typ dat
node_value	text	Vlastní hodnota uzlu
modified_by	name	Uživatelské jméno autora současné podoby záznamu
modified_on	timestamp	Časová značka vzniku aktuální podoby záznamu
has_sub_nodes	boolean	Příznak zda má vztah podřízené uzly (není list)
base_type_name	varchar(32)	Název základního datového typu
pg_type_oid	oid	Datový typ PostgreSQL reprezentovaný oid
user_type_name	varchar(64)	Název datového typu definovaného uživatelem
units	varchar(16)	Nastavené jednotky

Tabulka 5.13: Sloupce pohledu nodes_viw

5.4.6 Pohled nodes_filter_viw

Pohled nodes_filter_viw (viz tabulka 5.14) je rozšířenou verzí nodes_viw, která zavádí sloupce pro podporu filtrování. Zejména jde pak o pole s uzlovou a typovou cestou.

5.4.7 Pohled files_viw

Pohled files_viw rozšiřuje pouhé zobrazení tabulky files o odhad velikosti souvisejícího BLOBu. Pro zobrazení odhadu velikosti je využíváno systémové tabulky pg_largeobject, která je odpovědná právě za uložení BLOBů. Protože pro přímý přístup k systémovým tabulkám je potřeba uživatele s příznakem SUPERUSER, bylo vytvoření takového pohledu pro získání a zobrazení těchto informací běžným uživatelům nezbytné. Seznam sloupců pohledu obsahuje tabulka 5.15.

Sloupec	Typ	Popis
node_id	int4	Identifikační číslo uzlu
node_parent	int4	Odkaz k nadřazenému uzlu
user_type	int4	Odkaz na uživatelský typ dat
node_value	text	Vlastní hodnota uzlu
modified_by	name	Uživatelské jméno autora současné podoby záznamu
modified_on	timestamp	Časová značka vzniku aktuální podoby záznamu
has_sub_nodes	boolean	Příznak zda má vztah podřízené uzly
path	text	Uzlová cesta
typed_path	text	Typová cesta uzlu
is_node_value_valid	boolean	Obsahuje příznak souladu hodnoty s datovým typem (viz funkce is_node_value_valid())
base_type_name	varchar(32)	Jména základního datového typu
pg_type_oid	oid	Datový typ PostgreSQL reprezentovaný oid
user_type_name	varchar(64)	Název datového typu definovaného uživatelem
units	varchar(16)	Nastavené jednotky

Tabulka 5.14: Sloupce pohledu nodes_filter_viw

5.4.8 Pohled nodes_valid_viw

Pohled nodes_valid_viw rozšiřuje nodes_viw o pole, která obsahují příznaky korektnosti daných uzlů (viz tabulka 5.16). Vytvoření tohoto pohledu bylo nutné, protože jeho implicitní použití je příliš náročné. Proto došlo k oddělení standardního zobrazení stromu, v podobě nodes_viw, od zobrazení s kontrolními informacemi.

Tento pohled pro prováděnou kontrolu obsahuje celou řadu vnořených tj. korelovaných dotazů, což značně zvyšuje nároky na jeho vykonání a prodlužuje tak i odezvu aplikace samotné. Proto je tedy možnost zobrazování takových informací přímo v datovém stromu klientského programu volitelná.

Sloupec	Typ	Popis
file_id	int4	Identifikační číslo souboru
file_name	varchar(64)	Fyzický název souboru
node	int4	Odkaz k uzlu, jemuž soubor náleží
description	text	Popis souboru
pg_largeobject	oid	Odkaz na uložený BLOB v podobě oid
modified_by	name	Uživatelské jméno autora současné podoby záznamu
modified_on	timestamp	Časová značka vzniku aktuální podoby záznamu
page_count	boolean	Počet stránek zabraných BLOBem. Velikost jedné stránky odpovídá LOBLKSIZE, typicky 2 kB

Tabulka 5.15: Sloupce pohledu files_viw

Sloupec	Typ	Popis
node_id	int4	Identifikační číslo uzlu
node_parent	int4	Nadřazený uzel
user_type	int4	Odkaz k uživatelskému typu dat
node_value	text	Vlastní hodnota uzlu
modified_by	name	Uživatelské jméno autora podoby záznamu
modified_on	timestamp	Časová značka vzniku aktuální podoby záznamu
has_sub_nodes	boolean	Příznak zda má vztah podřízené uzly
base_type_name	varchar(32)	Jména základního datového typu
pg_type_oid	oid	Datový typ PostgreSQL reprezentovaný oid
user_type_name	varchar(64)	Název datového typu definovaného uživatelem
units	varchar(16)	Nastavené jednotky
path	text	Uzlová cesta
is_node_valid	boolean	Příznak zda je uzel v pořádku
is_branch_valid	boolean	Příznak zda jsou veškeré podřízené uzly (větev) v pořádku

Tabulka 5.16: Sloupce pohledu nodes_valid_viw

Kapitola 6

Implementace programu

Klientský program byl tedy implementován v IDE Delphi pro jeho podporu vývoje aplikací s GUI. Uživatelské rozhraní bylo programováno pomocí tzv. *akcí*, které představují dobrý prostředek pro tvorbu komplexního rozhraní s uživateli. Akce umožňují programátorovi snáze sdružit úsek kódu s více prvky rozhraní.

Implementovaný program se nazývá *Tr:Ex* (zkrácení slov Tree Explorer). Tento název rovněž odkazuje na jméno původní klientské aplikace. Ukázka výsledného programu je zobrazena v příloze C.

6.1 Jednotky

Jednotky, či angl. *units* jsou základním stavebním kamenem programů vytvářených v IDE Delphi. Zde uvedené jednotky představují jakési knihovny, které byly vytvořeny pro účel vznikajícího programu.

6.1.1 Jednotka PgAPI

Tato jednotka obsahuje základní funkce pro manipulaci s PostgreSQL. Jde funkce knihovny *libpq*, jež je oficiální knihovnou tohoto SŘBD pro tvorbu klientských aplikací. Jednotka `PgAPI` definuje pouze prototypy těchto funkcí a také základní datové typy rozhraní. Deklarace funkcí jsou napojeny na knihovnu *libpq.dll*, která obsahuje jejich implementaci.

Detailní informace o tomto rozhraní lze nalézt v oficiální dokumentaci SŘBD PostgreSQL [1].

6.1.2 Jednotka Config

Programová jednotka `Config` obsahuje prostředky pro načtení a uložení informací k připojení do souboru. Seznam obsažených tříd zobrazuje tabulka 6.1.

Třída	Popis
`TCfgConnection`	Položka připojení
`TCfgConnectionList`	Seznam položek připojení

Tabulka 6.1: Třídy jednotky Config

Třída TCfgConnection

Třída `TCfgConnection` je reprezentací jedné položky uloženého připojení. Každé uložené připojení je definováno svým vlastním jménem, adresou nebo jménem serveru PostgreSQL, číslem portu a jménem databáze, ke které se bude připojovat a jež má požadovanou strukturu.

Třída TCfgConnectionList

Třída `TCfgConnectionList` představuje seznam, obsahující jednotlivé položky připojení. Jde tedy o kontejner objektů třídy `TCfgConnection`. Třída obsahuje prostředky pro načtení a uložení takového seznamu do souborů INI.

Pro potřeby a podporu GUI obsahuje procedury `Notice` a `Noticed`. První jmenovaná nastavuje příznak `FWasChanged`, kterým lze informovat o jakékoliv proběhnuvší změně v seznamu připojení. Za nastavení příznaku je odpovědné např. samotné načtení položek ze souboru či změna položky v okně se správou těchto připojení. Je-li změna zaznamenána a reflektována (nabídka hlavního okna), voláním `Noticed` dojde k vymazání příznaku změny, resp. nastavení jej na `False`.

6.1.3 Jednotka TreeBase

Nejrozsáhlejší jednotka celé aplikace, nazvaná shodně s navrženou databází, tedy `TreeBase`, obsahuje zejména implementaci vlastního ORM. Výčet jednotkou obsažených tříd zobrazuje tabulka 6.2.

Třída	Popis
`TTBDatabase`	Třída představuje databázi TreeBase
`TTBFilter`	Třída zpracovávající filtrovací dotazy a následné generování odpovídajícího SQL
`TCustomTB`	Základní třída celého ORM. Jde o abstrakci libovolného záznamu
`TCustomTBAudit`	Třída je abstrakcí libovolného záznamu s auditorskými informacemi
`TTBBaseType`	Třída představující základní typ dat
`TTBUserType`	Třída popisující uživatelem definovaný typ dat
`TTBRelation`	Třída reprezentující vztah
`TTBNode`	Třída vlastního uzlu datového stromu
`TTBText`	Třída představující doprovodný text
`TTBFile`	Třída představující přiložený soubor
`TTBValidation`	Třída provádějící kontrolu uzlu a její vyhodnocení
`TTBUser`	Třída představující uživatele PostgreSQL
`TTBLog`	Třída popisující jediný záznam změny

Tabulka 6.2: Třídy jednotky TreeBase

Třída TTBDatabase

Instance třídy `TTBDatabase` představuje objekt připojené databáze TreeBase. Obsahuje metody pro manipulaci s připojením (tedy pro připojení resp. odpojení), dále poskytuje operace pro získání dat z databáze např. kořenové uzly stromu či seznam definovaných typů.

Jde o základní třídu navrženého ORM, a každý objekt, reprezentující už libovolný záznam, se na instanci připojení odkazuje. Pro ostatní třídy ORM je tedy přístupovým bodem k databázi.

Třída TTBFilter

Třída `TTBFilter` v sobě zapouzdřuje prostředky pro manipulaci s filtrováním resp. vyhledáváním. Zjednodušeně řečeno, tato třída přijímá dotaz filtrovacího jazyka (viz kapitola 6.4), ověří a transformuje jej na SQL dotaz, který je již přímo aplikovatelný na připojenou databázi. Výsledkem je tedy SQL dotaz, jež zobrazí (pouze) požadované uzly (popsané filtrovacím dotazem).

Třída TCustomTB

Třída `TCustomTB`, jak již název vypovídá, je základní abstrakcí libovolného záznamu v databázi. Třída již obsahuje generickou implementaci operací se záznamem, tj. operace `SELECT`, `INSERT`, `UPDATE` a `DELETE`. Protože implementace koncových tříd se liší zejména v SQL dotazech, musí každý potomek této třídy implementovat abstraktní funkce `GetSelectSQL`, `GetInsertSQL`, `GetUpdateSQL` a `GetDeleteSQL` podle konkrétních požadavků.

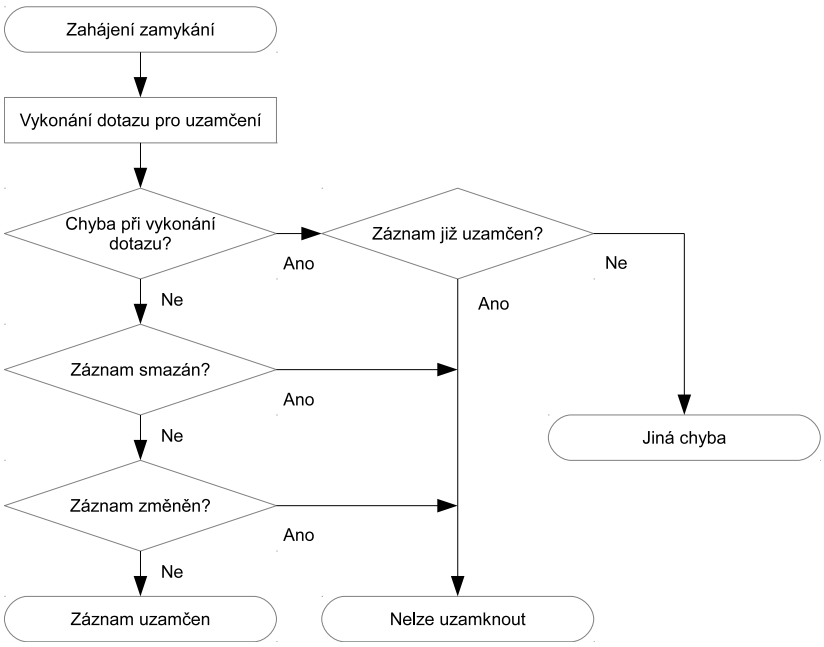
Hlavním účelem této abstrakce bylo minimalizovat opakování kódu, např. pro vykonání dotazu, který by byl v jednotlivých třídách takřka totožný. Další výhodu abstrakce představují tzv. *reference na třídu*, což je specifická konstrukce jazyka Object Pascal, která dovoluje odkazovat se na libovolnou třídu dědící od té, jejíž referenci použijeme. Tohoto využívá funkce `GetObjectsByQuery` třídy `TTBDatabase`, která může díky této skutečnosti vrátit libovolný seznam objektů zadané třídy.

Třída TCustomTBAudit

Další abstrakce, která je potomkem třídy `TCustomTB` a dále ji rozšiřuje o podporu auditorských informací. I zde stála za vznikem třídy snaha o minimalizaci kódu.

Dále třída zavádí abstrakce pro uzamykání databázových záznamů. Pružnost tohoto řešení představuje zejména funkce `GetLockSQL`, která svým potomkům ukládá povinnost definovat SQL dotaz, pro uzamčení odpovídajícího záznamu.

Protože PostgreSQL zamyká záznamy způsobem výběru se speciálním příznakem (příznak `FOR UPDATE`), je tento záznam zároveň vrácen ve výsledku dotazu. Z hlediska multiuživatelských databází je pak vhodné zkontrolovat, zda již nebyl záznam změněn (či vymazán). Kontrolu případné změny obstará funkce `WasChanged`. Pro upřesnění je ještě potřeba uvést, že používaná syntaxe dotazu po uzamčení ještě přidává `NOWAIT`, která vyvolá chybu je-li



Obrázek 6.1: Proces zamčení záznamu

záznam již uzamčen jiným uživatelem. Pokud by nebylo užito tohoto parametru, došlo by k neomezenému čekání, něž bude opět záznam odemčen. Schéma průběhu uzamykání záznamu je zobrazeno na obrázku 6.1.

Poslední prostředek, jež kompletuje abstrakci uzamykání záznamů, je funkce `GetLockDeleteSQL`, která vrací dotaz pro uzamčení před samotným smazáním záznamu a to včetně veškerých závislých.

Třída TTBBaseType

Jde o první třídu, které je přímo využíváno v programu, tj. třídu, která představuje záznam konkrétní databázové tabulky. Objekt této třídy reprezentuje základní typ, který je databází podporován. Třída `TTBBaseType` je potomkem `TCustomTB`.

Protože tabulka základních typů je (minimálně ze strany klienta) neměnná, nejsou funkce `GetInsertSQL`, `GetUpdateSQL` a `GetDeleteSQL` řádně implementovány.

Třída TTBUserType

Třída `TTBUserType`, jež je potomkem `TCustomTBAudit`, představuje uživatelem definovaný datový typ. Objekt této třídy dovoluje vykonat kontrolu hodnoty, zda je v souladu s uživatelským typem (a s tím souvisejícím typem PostgreSQL), který je instancí reprezentován. K tomu slouží funkce `Validate`, která je klientským protějškem databázové funkce `is_node_value_valid()`.

Třída TTBRelation

Objekt třídy `TTBRelation`, jež je potomkem `TCustomTBAudit`, představuje vztah uživatelského datového typu resp. popis jeho povoleného výskytu v datovém stromu.

Aby mohl být vztah nějak reprezentován v GUI, zavádí třída funkcí `GetCaption`, která sestaví popisek uživatelem definovaného typu, na který se vztah odkazuje. Funkce je obdobou `get_user_type_caption()` (viz kapitola 5.2.3).

Třída TTBNode

Třída `TTBNode`, která je potomkem `TCustomTBAudit`, představuje samotný uzel datového stromu. Pro prezentování uzlu zavádí třída funkci `GetCaption`, jež sestaví popisek odpovídající databázové funkci `get_node_caption()`.

Třída TTBText

Třída `TTBText`, jež je potomkem `TCustomTBAudit`, představuje doprovodný text k uzlu. Pro uložení formátovaného textu užívá program *RTF*[1] a s tím související komponenty `RichEdit`.

Kvůli přítomnosti mnoha specifických formátovacích značek v tomto formátu, nelze provést jeho uložení do databáze přímo. Nejprve musí být zdrojový text RTF formátu zpracován funkcí jednotky `PgAPI` (viz kapitola 6.1.1) `PQescapeStringConn` a až následně uložen. Bez této úpravy by dotaz uložení skončil s chybou.

[1] Rich Text Format je společností Microsoft vyvinutý formát souboru pro uložení formátovaného textu.

Třída TTBFile

Třída `TTBFile`, která je potomkem `TCustomTBAudit`, představuje přiložený soubor. Obsahuje prostředky pro nahrání (obsahu) souboru na server (procedura `LoadFromFile`) resp. jeho stažení (procedura `SaveToFile`). Zmíněné metody akceptují jako parametr proceduru odpovídající výpisu 6.1, pomocí které lze sledovat průběh přenosu souboru a případně jej i ukončit.

Výpis kódu 6.1: Typ procedury pro sledování přenosu

```
TTBFileEvent = procedure (Current, Size: Integer;
    var Cancel: Boolean) of object;
```

Třída TTBValidation

Třída `TTBValidation` představuje prostředek pro kontrolu uzlů, zda vyhovují jak hodnotou, tak i umístěním. Rovněž kontroluje počty textů a souborů. Výsledek kontroly je třídou uložen a lze jej následně publikovat pomocí k tomu určených funkcí.

Tyto funkce využívá rám `ShowValidation` (viz kapitola 6.2) a generuje uživateli srozumitelné zprávy o proběhlé kontrole.

Třída TTBUser

Třída `TTBUser`, která je potomkem `TCustomTB`, představuje uživatele databázového serveru PostgreSQL. Třída poskytuje prostředky pro nastavení hesla či určení členství resp. oprávnění. Pro uložení doprovodných informací jako je např. celé jméno nebo e-mailová adresa využívá třída možnosti vytvořit komentář nad uživatelem (příkaz `COMMENT ON ROLE ...`) v SŘBD PostgreSQL.

Třída TTBLog

Třída `TTBLog`, která je potomkem `TCustomTB`, představuje záznam jedné změny v databázi. Záznamy, reprezentované třídou `TTBLog`, převádí rám `ShowLog` (viz kapitola 6.2) na uživateli srozumitelné zápisy.

6.2 Rámy

Rámy představují možnost jak vytvářet souvislé části GUI (podobné formuláři). Rámů lze užít jak kvůli jejich znovupoužitelnosti, tak kvůli snaze o rozložení rozsáhlých a nepřehledných oken na menší a lépe čitelné jednotky, tedy rámy. Jde o alternativu k vizuální dědičnosti formulářů.

Příkladem dobrého užití rámů jsou okna, které obsahují více karet či záložek (komponenta `PageControl` nebo `TabControl`). Pomocí rámů lze navrhnout a implementovat každou kartu zvlášť a následně jí umístit do cílového formuláře. Umístěný rám ve formuláři není třídou, ale instancí rámu. Proto je možné znovupoužití rámu i s drobnými obměnami. Rovněž lze již implementované obsluhy uvnitř rámu upravit či úplně předefinovat, podobně jako je tomu u dědičnosti.

Rám	**Popis**
`ManageTypes`	Správa uživatelských typů
`ManageRelations`	Správa struktury/vztahů uživatelských typů
`ManageUsers`	Správa uživatelů.
`ShowLog`	Rám zobrazující historii změn databáze
`ManageNodes`	Nejdůležitější rám aplikace, obsahuje správu datového stromu
`ManageTexts`	Správa doprovodných textů
`ManageFiles`	Správa přiložených souborů
`ShowValidation`	Rám sloužící ke kontrole uzlu

Tabulka 6.3: Rámy aplikace

Protože rámy aplikace jsou jen vizuální, implementačně nezajímavou vrstvou, postačí toto téma shrnout odkazem na zdrojové kódy programu (viz příloha D) a seznamem rámů v tabulce 6.3, kde je ke každé položce uvedena i stručná charakteristika.

6.3 Formuláře

Největšími celky uživatelského rozhraní v Delphi jsou jednotlivá okna resp. formuláře. Jejich výčet, tedy těch vytvořených pro účel aplikace, je prezen-

tován v tabulce 6.4.

Formulář	Popis
`Main`	Hlavní formulář aplikace
`Child`	Okno s jednou připojenou databází
`NewConnection`	Dialogové okno pro nové připojení
`Login`	Dialogové okno pro přihlášení
`ManageConnections`	Okno se správou uložených připojení
`Filter`	Dialogové okno pro nastavení výrazu filtru
`Progress`	Dialogové okno zobrazující průběh časově náročné operace

Tabulka 6.4: Formuláře aplikace

Podobně jako rámy, ani formuláře nejsou implementačně zajímavé a proto odkazuji rovnou na zdrojové kódy programu (viz příloha D).

Za zmínku stojí pouze okno Progress, které zobrazuje průběh časově náročné operace a umožňuje uživateli případné přerušení této akce. Toto okno při svém zobrazení zahájí onen náročný proces (přenos souborů), ale v nově vytvořeném, separátním vlákně.

6.4 Vyhledávání

Vyhledávání, filtrování či dotazování bylo jednou ze stěžejních kapitol implementace. Nejprve bylo potřebné navrhnout nějaký jednoduchý dotazovací jazyk, který by pokryl veškeré potřeby aplikace. Příklad jsem si tedy vzal z již v zadání zmíněného XPath.

Za zpracování dotazu, který je v souladu s navrženým jazykem (viz výpis 6.2) je odpovědná třída `TTBFilter` (viz kapitola 6.1.3). V této kapitole se nebudu zabývat vlastním parsováním dotazu, ale principem transformace dotazu filtrovacího jazyka do SQL. Pro lepší pochopení bude vysvětlení převodu demonstrováno na příkladu. Užitý demonstrační výraz odpovídá výpisu 6.3.

Zadaný výraz (výpis 6.3) tedy požaduje zobrazení všech kořenových uzlů typu `1`. Toť veškerá **základní část** dotazu. Vybrané uzly dále podléhají třem podmínkám, které jsou navzájem v konjunkci. **První** podmínka požaduje, aby vybrané uzly měly jinou hodnotu než „A“. **Druhá** podmínka říká, že

Výpis kódu 6.2: Navržený dotazovací jazyk

```
QUERY       ::=   PATH [ CONDITIONS ]
PATH        ::=   STEP { STEP }
STEP        ::=   DELIMITER <user_type>
DELIMITER   ::=   '/' [ '/' ]
CONDITIONS  ::=   { '[' CONDITION ']' }
CONDITION   ::=   PARAMETER OPERATION '"' <value> '"'
PARAMETER   ::=   '.' | ( [ '//' ] <user_type> [ PATH ] )
OPERATION   ::=   '=' | '<>' | '<' | '>' | '<=' | '>='
```

Výpis kódu 6.3: Dotaz na filtrování dat

```
/1[.<>"A"][3/5<"3"][//2="zelená"]
```

vybrané uzly musí zároveň obsahovat poduzel typu `3`, a ten musí obsahovat poduzel typu `5`, jehož hodnota je menší než „3“. **Třetí** (a poslední) podmínka říká, že vybrané uzly budou mít poduzel v libovolné úrovni větve, který bude mít hodnotu „zelená“.

1. Nejprve dojde ke zpracování tzv. *základní části* dotazu, která říká co hledáme. Jde tedy o zpracování symbolu `PATH` v pravidle `QUERY` (viz výpis 6.2), tj. „`/1`“.

Výraz	**Popis**
`a/b`	Uzel typu `b` bezprostředně následuje `a`
`a//b`	Uzel typu `b` je v libovolné úrovni větve počínající uzlem `a`

Tabulka 6.5: Oddělovače cesty

2. Ze základu je pak stanovena typová cesta. Přesněji tedy dochází k sestavení regulárního výrazu, který popisuje typovou cestu pouze vyhovujících uzlů. Důvod užití regulárního výrazu plyne z tabulky 6.5 (viz sekvence „`//`“).

3. Z tohoto základu lze stanovit první (jedinou, pokud nejsou zadány podmínky) část SQL dotazu. Tato část výběrového dotazu zobrazí

veškeré uzly, jejichž typová cesta odpovídá sestavenému regulárnímu výrazu (viz výpis 6.4).

4. Nyní je potřeba zpracovat zadané podmínky.

   (a) Zpracování pravidla `PARAMETER` je obdobné jako při zpracování základu. Hlavní rozdíl představuje možnost zadání „.“, což znamená dotázání se přímo na hodnoty uzlů vybraných základní částí SQL dotazu. V takovém případě se použije výraz typové cesty základu dotazu.

   (b) Při zpracování podmínek (vlastně i při zpracování základu) probíhá kontrola, zda je ve výrazu užito platného, uživatelem definovaného typu. Zpracování podmínky dále přináší povinnost zkontrolovat, zda zadaná hodnota v dotazu odpovídá cílovému typu. Pokud by tomu tak nebylo, přetypování ve výsledném SQL by způsobilo chybu. Poslední kontrola ověří, zda lze užít zadané operace v kombinaci s cílovým typem. Nelze například použít operace „větší/menší“ pro typ textového řetězce.

   (c) Nyní přichází na řadu samotná konstrukce zbytku SQL dotazu. Každá zadaná podmínka je v SQL reprezentován zápisem, který začíná „`AND EXISTS(...`“. Aby byla vytvořena vazba mezi uzly, které byly vybrány základem a podmínkami, musí mít každý uzel vybraný podmínkou prefix uzlové cesty shodný s uzlem, který byl vybrán základem.

   (d) Každý uzel vybraný podmínkou musí mít typovou cestu, která vyhovuje výrazu, jež byl pro tuto podmínku sestaven.

   (e) Dále musí mít každý uzel, vybíraný podmínkou, hodnotu v souladu s datovým typem, kterým je sám určen. Uzly s nevyhovující hodnotou nejsou do výběru zahrnuty, aby se předešlo chybě při vykonání dotazu SQL.

   (f) Nakonec je tedy sestavena část, kdy dochází k vlastnímu srovnání hodnoty uzlu a hodnoty zadané podmínkou filtrovacího dotazu. Samozřejmě dle zadané operace.

Výpis 6.4 představuje výsledné SQL, které odpovídá filtračnímu dotazu z výpisu 6.3. Zároveň je patrné využití pohledu `nodes_filter_viw` (viz kapitola 5.4.6).

Výpis kódu 6.4: Výsledný SQL dotaz

```
-- Základní část
SELECT *
FROM tb.nodes_filter_viw m
WHERE m.typed_path ~ E'^1$'
-- První podmínka
AND EXISTS (
  SELECT *
  FROM tb.nodes_filter_viw s
  WHERE s.path LIKE m.path || '%'
  AND s.typed_path ~ E'^1$'
  AND CASE WHEN s.is_node_value_valid
    THEN s.node_value::text NOT ILIKE 'A'::text
    ELSE false END)
-- Druhá podmínka
AND EXISTS (
  SELECT *
  FROM tb.nodes_filter_viw s
  WHERE s.path LIKE m.path || '%'
  AND s.typed_path ~ E'^1\.3\.5$'
  AND CASE WHEN s.is_node_value_valid
    THEN s.node_value::bigint < '3'::bigint
    ELSE false END)
-- Třetí podmínka
AND EXISTS (
  SELECT *
  FROM tb.nodes_filter_viw s
  WHERE s.path LIKE m.path || '%'
  AND s.typed_path ~ E'^1\.(\\d{1,}\.)*2$'
  AND CASE WHEN s.is_node_value_valid
    THEN s.node_value::text ILIKE 'zelená'::text
    ELSE false END)
```

Kapitola 7

Závěr

Závěrem dokumentu je potřeba shrnout čeho se podařilo v rámci diplomové práce dosáhnout, nastínit možné další kroky vývoje aplikace a subjektivně zhodnotit práci samotnou.

7.1 Dosažené výsledky

V rámci zadání se podařilo vytvořit aplikaci s relativně komplexním GUI. Byla navržena nová struktura databáze, která klade důraz na přesun funkcí směrem k SŘBD. Tím došlo k odlehčení klientského programu a poklesu nároků na výpočetní techniku pro provozování klienta. Rovněž bylo vytvořeno nové, vlastní řešení objektově-relačního mapování, které přímo využívá funkcí rozhraní pro PostgreSQL. Tím byla eliminována případná režie (nebo překážky) v podobě další mezivrstvy pro manipulaci s databázovým serverem. Při implementaci byl kladen důraz na výkonnost celého řešení. Příkladem může být minimalizace počtu vykonávaných dotazů a počtu přenášených sloupců v jejich výsledcích.

Program jsem testoval na obdobných datech, jaký mi byla zprostředkována z původní databáze (materiálů) programu. Bohužel šlo jen o minimální, ale naštěstí však reprezentativní množství dat, ze kterého jsem byl schopen určit požadavky na ukládaná data. Podle těchto zjištění jsem si pak data libovolně replikoval, abych získal jejich větší vzorek. Díky tomu jsem mohl aplikaci lépe odladit a přizpůsobit. Program samozřejmě není limitován pouze ukládáním informací o materiálech. Protože lze v programu definovat libovolné typy (uzlů), lze do takové databáze ukládat téměř libovolné informace.

7.2 Další vývoj

Kudy by se tedy mohl ubírat další vývoj. Nejprve je asi potřeba nechat nějakou dobu aplikaci fungovat a dlouhodobě sbírat případné nedostatky či návrhy na rozšíření.

Pokud bych ale mohl bez tohoto dlohodobého testování navrhovat možné úpravy projektu, zvážil bych další přesun funkcí směrem k databázovému systému. V průběhu implementace jsem se několikrát zabýval myšlenkou vytvoření aplikačního serveru. Tím by bylo možné implementování tzv. *tenkého klienta*. Případný následný vývoj alternativního klienta by byl rovněž snažší. Na druhou stranu vývoj aplikačního serveru není zcela triviální. To už je ale na zvážení pokračovatele tohoto projektu.

Mezi konkrétní kroky k vylepšení či dalšímu vývoji aplikace bych nakonec zařadil například přesun zpracování filtrovacího jazyka do SŘBD. PostgreSQL nabízí možnost implementovat uložené funkce v jazyce C. Implementace této činnosti pomocí PL/SQL by byla pravděpodobně nevhodná.

Právě přenos většího množství funkcí směrem k databázovému serveru a jejich implementace v jazyce C by mohl být vhodným krokem v dalším vývoji. Zároveň by tím došlo k částečnému suplování aplikačního serveru.

7.3 Zhodnocení

Práce mě naučila mnohému. V době jejího dokončování jsem nabyl dojmu, že jsem se naučil rychleji programovat. Při implementaci vlastního programu se mě příliš neosvědčila návrhová část. Přestože jsem návrhu programu věnoval větší množství času, následné implementace častokrát končily ve slepé uličce. Naopak způsob, který nakonec vedl k úspěšnému implementování programu, bylo prvotní naprogramování aplikace bez většího návrhu a za cenu nekvalitního a opakujícího se kódu. Až následně jsem prováděl návrh konečného návrhu, kdy už jsem dopředu věděl, kterými vlastnostmi musí tento návrh disponovat. Potom již bylo implementování konečné aplikace snažší a mohl jsem se více soustředit na kvalitu kódu výsledné aplikace a na dodržení dobrých programátorských zásad a návyků.

Přestože tento způsob práce není zcela obvyklý a odporuje mnohému co jsem se v průběhu studia učil, vedl právě tento postup ke zdárnému vytvoření programu.

Úplným závěrem bych rád vyjádřil vděčnost zadanému tématu, jež mě

umožnilo vyzkoušet si techniky, které jsem znal pouze ze skript. Při vytváření programu jsem měl možnost zabývat se problematikou multiuživatelských databází, s čímž souvisí pojmy jako transakce, izolace a uzamykání.

Za získané zkušenosti jsem velice rád a i přes nemalý stres, který jsem při plnění zadání prožíval, byla pro mě tato práce cenným přínosem.

Přehled zkratek

BLOB	Binary Large OBject (BLOB) je datový typ blíže nespecifikovaných binárních dat v databázi.
CSV	Comma-Separated Values je souborový formát popisující tabulková data.
CTE	Common Table Expressions je pojmenovaná dočasná množina záznamů.
GUI	Graphic User Interface je anglický výraz pro grafické uživatelské rozhraní.
HQL	Hibernate Query Language je objektově orientovaná analogie k SQL frameworku Hibernate.
HTML	HyperText Markup Language je značkovacím jazykem hypertextu.
IDE	Integrated Development Environment je anglickým označením pro vývojové prostředí.
INI	INI je standardní formát konfiguračních souborů.
IP	Internet Protocol je protokol používaný v počítačových sítích.
JDBC	Java Database Connectivity je jednotné rozhraní pro přístup k relačním databázím v jazyce Java.
MDI	Multiple Document Interface je rozhraní skládající z několika formulářů, které se nacházejí uvnitř hlavního okna aplikace.

ODBC	Open Database Connectivity je standardizované rozhraní pro přístup k databázím.
ORM	Objektově relační mapování (z angl. Object-Relational Mapping) je programovací technika zajišťující konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem.
PL/SQL	Procedural Language/Structured Query Language je procedurální nadstavbou jazyka SQL.
RTF	Rich Text Format je společností Microsoft vyvynutý formát souboru pro uložení formátovaného textu.
SFTP	SSH File Transfer Protocol je protokol pro bezpečný přenos souborů počítačovou sítí.
SQL	Structured Query Language je dotazovací jazyk pro manipulaci s databázovými daty.
SSH	Secure SHell je zabezpečený komunikační protokol v počítačových sítích.
STL	Standard Template Library je softwarová knihovna jazyka C++.
SŘBD	Systém Řízení Báze Dat je korektní označení databázových systémů.
XML	Extensible Markup Language je standardizovaný značkovací jazyk.

Literatura

[1] The PostgreSQL Global Development Group: *PostgreSQL 9.1.3 Documentation*. PostgreSQL [online], 2011. [cit. 1.5.2012].
Dostupné z: http://www.postgresql.org/docs/9.1/interactive/index.html.

[2] Cantú, Marco: *Myslíme v jazyku Delphi 6, díl 1.*
Grada Publishing a.s., 2002. ISBN 80-247-0334-3.

[3] Cantú, Marco: *Myslíme v jazyku Delphi 6, díl 2.*
Grada Publishing a.s., 2002. ISBN 80-247-0335-1.

[4] Písek, Slavoj: *Delphi - Začínáme programovat.*
Grada Publishing a.s., 2002. ISBN 80-247-0547-8.

[5] Přispěvatelé Wikipedie: *Objektově relační mapování*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 3.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Objektově_relační_mapování.

[6] Přispěvatelé Wikipedie: *Hibernate*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 3.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Hibernate.

[7] Ethea S.r.l: *InstantObjects: Open Source Object Persistent Framework*. Home ETHEA [online]. [cit. 4.5.2012].
Dostupné z: http://www.ethea.it/eng_tecnologia_instantobjects.asp.

[8] tiOPF Project Team: *A quick guide to tiOPF*. tiOPF Home Page [online]. [cit. 4.5.2012].
Dostupné z: http://tiopf.sourceforge.net/Doc/overview/index.shtml.

[9] Code Synthesis Tools CC: *C++ Object Persistence with ODB*. Code Synthesis [online], 2012. [cit. 4.5.2012].
Dostupné z: http://tiopf.sourceforge.net/Doc/overview/index.shtml.

[10] Michael Gradman and Corwin Joy: *Frequently Asked Questions about DTL*. Database Template Library Programmer's Guide [online]. [cit. 4.5.2012].
Dostupné z: http://dtemplatelib.sourceforge.net/FAQ.htm.

[11] SOCI People: *SOCI - The C++ Database Access Library*. SOCI [online]. [cit. 4.5.2012].
Dostupné z: http://soci.sourceforge.net/.

[12] Mike Hillyer: *Managing Hierarchical Data in MySQL*. MySQL Developer Zone [online], 2010. [cit. 1.4.2012].
Dostupné z: http://web.archive.org/web/20110606032941/
http://dev.mysql.com/tech-resources/articles/hierarchical-data.html.

[13] Scott Mitchell: *Maintaining a Log of Database Changes - Part 1*. 4GuysFromRolla.com [online], 2007. [cit. 4.4.2012].
Dostupné z: http://www.4guysfromrolla.com/webtech/041807-1.shtml.

[14] Přispěvatelé Wikipedie: *Microsoft SQL Server*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 5.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Microsoft_SQL_Server.

[15] Přispěvatelé Wikipedie: *Oracle*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 5.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Oracle.

[16] Přispěvatelé Wikipedie: *Interbase*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 5.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/InterBase.

[17] Přispěvatelé Wikipedie: *Firebird*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 5.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Firebird.

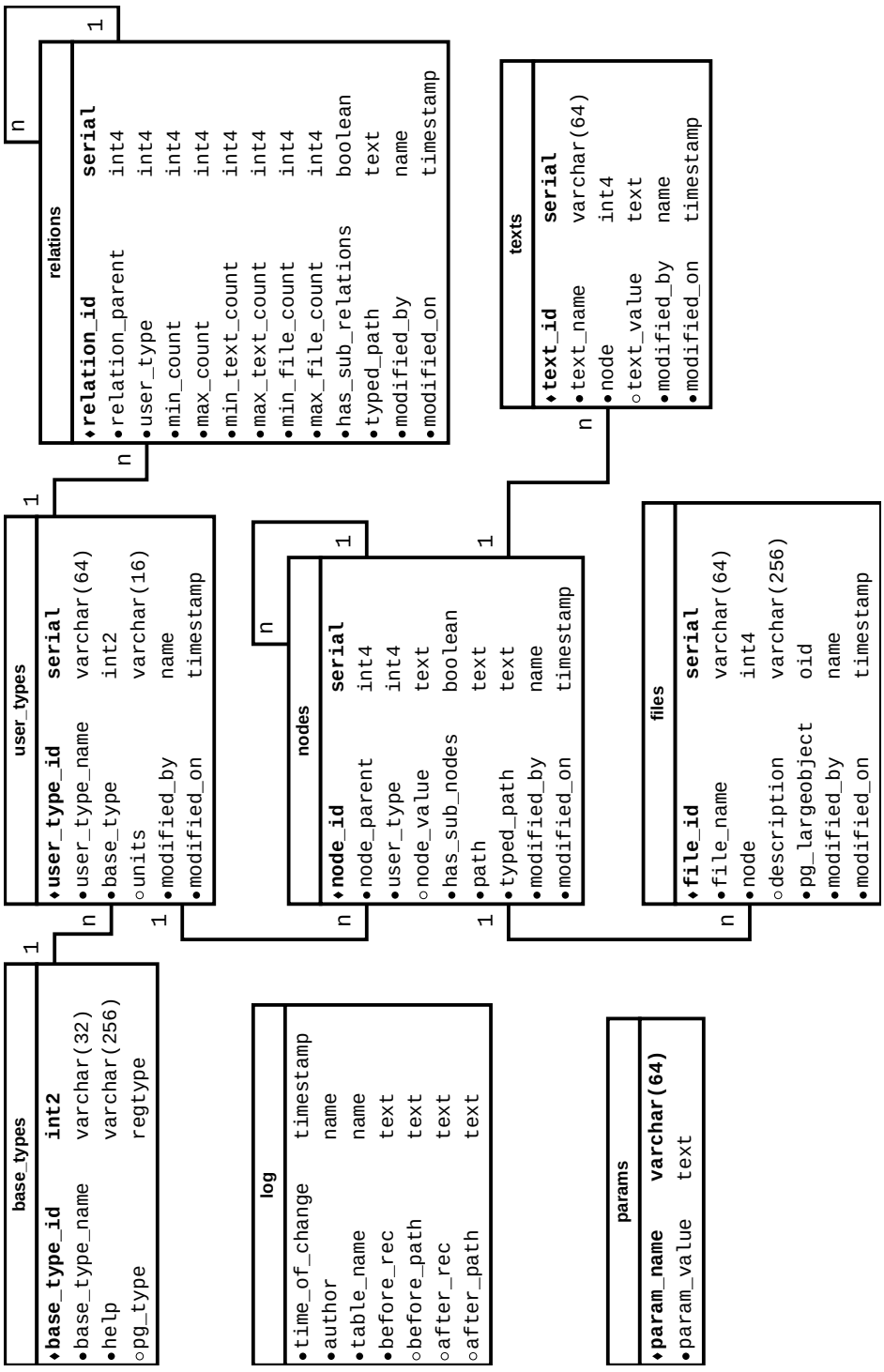
[18] Přispěvatelé Wikipedie: *MySQL*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 5.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/MySQL.

[19] Přispěvatelé Wikipedie: *Qt*. Wikipedie: Otevřená encyklopedie [online], 2012. [cit. 7.5.2012].
Dostupné z: http://cs.wikipedia.org/wiki/Qt_(knihovna).

[20] FindTheBest: *Firebird vs MySQL vs PostgreSQL*
FindTheBest.com [online]. [cit. 5.5.2012].
Dostupné z: http://database-management-systems.findthebest.com/compare/13-30-43/Firebird-vs-MySQL-vs-PostgreSQL.

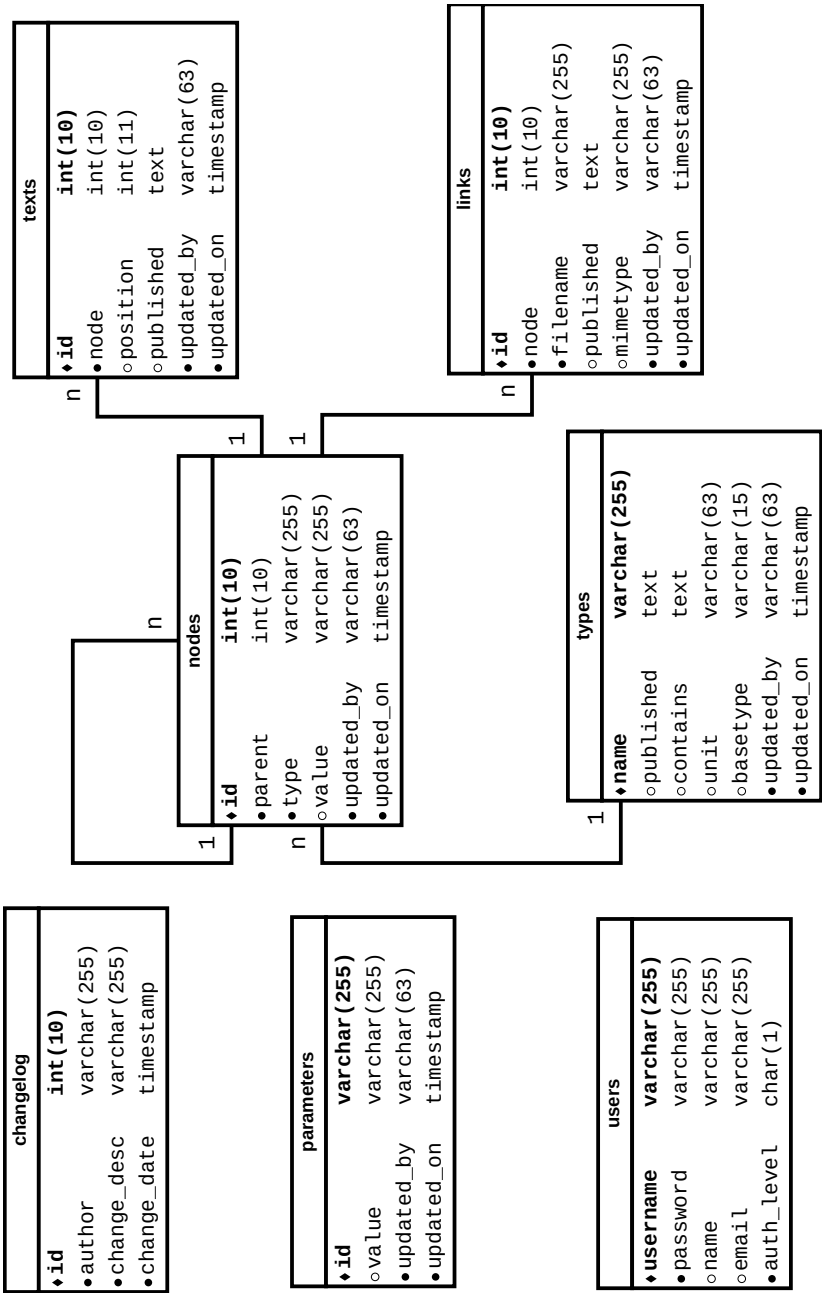
Příloha A

Nový model databáze


base_types
base_type_id int2
base_type_name varchar(32)
help varchar(256)
pg_type regtype
user_types
user_type_id serial
user_type_name varchar(64)
base_type int2
units varchar(16)
modified_by name
modified_on timestamp
relations
relation_id serial
relation_parent int4
user_type int4
min_count int4
max_count int4
min_text_count int4
max_text_count int4
min_file_count int4
max_file_count int4
has_sub_relations boolean
typed_path text
modified_by name
modified_on timestamp
nodes
node_id serial
node_parent int4
user_type int4
node_value text
has_sub_nodes boolean
path text
typed_path text
modified_by name
modified_on timestamp
texts
text_id serial
text_name varchar(64)
node int4
text_value text
modified_by name
modified_on timestamp
files
file_id serial
file_name varchar(64)
node int4
description varchar(256)
pg_largeobject oid
modified_by name
modified_on timestamp
log
time_of_change timestamp
author name
table_name name
before_rec text
before_path text
after_rec text
after_path text
params
param_name varchar(64)
param_value text
1
n

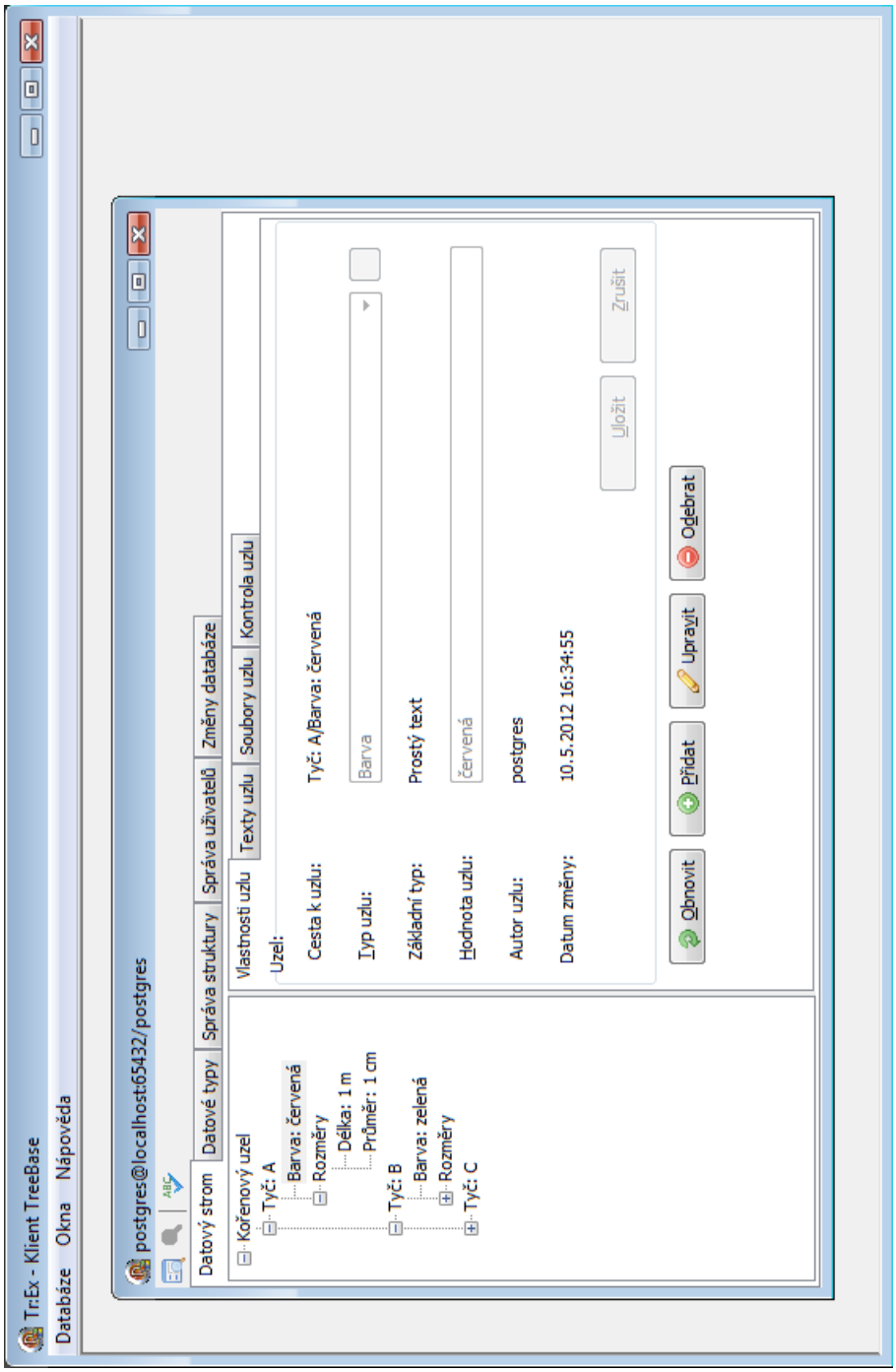
Příloha B

Model původní databáze


texts
♦id int(10)
•node int(10)
◦position int(11)
◦published text
•updated_by varchar(63)
•updated_on timestamp
links
♦id int(10)
•node int(10)
•filename varchar(255)
◦published text
◦mimetype varchar(255)
•updated_by varchar(63)
•updated_on timestamp
nodes
♦id int(10)
•parent int(10)
•type varchar(255)
◦value varchar(255)
•updated_by varchar(63)
•updated_on timestamp
types
♦name varchar(255)
◦published text
◦contains text
◦unit varchar(63)
◦basetype varchar(15)
•updated_by varchar(63)
•updated_on timestamp
changelog
♦id int(10)
•author varchar(255)
•change_desc varchar(255)
•change_date timestamp
parameters
♦id varchar(255)
◦value varchar(255)
•updated_by varchar(63)
•updated_on timestamp
users
♦username varchar(255)
•password varchar(255)
◦name varchar(255)
◦email varchar(255)
•auth_level char(1)
n
1
n
1
n
1
n
1
n
1

Příloha C

Vzhled programu


TrEx - Klient TreeBase
Databáze Okna Nápověda
postgres@localhost:65432/postgres
Datový strom Datové typy Správa struktury Správa uživatelů Změny databáze
Kořenový uzel
Tyč: A
Barva: červená
Rozměry
Délka: 1 m
Průměr: 1 cm
Tyč: B
Barva: zelená
Rozměry
Tyč: C
Vlastnosti uzlu Texty uzlu Soubory uzlu Kontrola uzlu
Uzel:
Cesta k uzlu: Tyč: A/Barva: červená
Typ uzlu: Barva
Základní typ: Prostý text
Hodnota uzlu: červená
Autor uzlu: postgres
Datum změny: 10.5.2012 16:34:55
Uložit
Zrušit
Obnovit
Přidat
Upravit
Odebrat

Příloha D

CD

Součástí práce je také přiložený CD-ROM s vytvořenou aplikací a jejími zdrojovými kódy.