Abstrakt

Předmětem této práce je zpracování návrhu centralizovaného databázového informačního systému založeného na klient-server architektuře pro správu klientů, členů, předplatitelů, sponzorů, dobrovolníků a jiných druhů subjektů. U subjektů se mají zanalyzovat a zalgoritmizovat datové toky jako jsou jejich platby, rozesílky, komunikace, účast na akcích apod. Projekt je zpracováván pro neziskovou organizaci sídlící v Liberci s názvem "Společnost přátel přírody". Tato společnost za pomoci mnoha subjektů obnovuje přirozené lesy v Jizerských horách a provozuje ekologické výukové programy. Pro kontrolu došlých plateb, celkového plánování a evidenci rozesílek potřebují robustní serverovou část a rychlou a přehlednou klientskou část databázového informačního systému. Návrh databáze v Case Studiu 2 pro databázový server FireBird 1.5 je poté doplněn GUI front-endem běžícím na klientském PC programovaném v prostředí Borland Delphi 6 a komunikujícím s databází po síti protokolem TCP/IP na portu 3050.

Klíčová slova: DBIS, databáze, informační systém, Case Studio, Firebird

Abstract

Main goal of this project is to process the proposal of centralized database information system, which is based on client-server architecture. It should be used for managing of clients, members, subscribers, sponsors, volunteers and the other kinds of subjects. There should be analysis and algorithmisation of data flows such as payments, distributors, communications, participation in action etc. Project has been processed for nonprofit organization resident in Liberec with name of „Společnost přátel přírody“ (Company of nature friends). This company is renewing natural forests in Jizerské Mountains and does ecological education.

For controlling of payments, planning and distributions record they need robust server part and quick and well arranged client part of database information system. Proposal of database in Case Studio 2 for database server Firebird 1.5 is then completed with GUI front-end, which is running on client PC. Client program is made in Borland Delphi 6 and it communicates with database on the server by the protocol TCP/IP at the port 3050.

Keywords: DBIS, database, information system, Case studio, FireBird

OBSAH:

Slovník pojmů:

Atribut

datová složka relace nebo entity, vazby. Musíme definovat jeho typ (nebo doménu) a můžeme také definovat samostatný constraint. Při převodu do konkrétní relační databáze se z atributu stává sloupec tabulky.

Batch (dávka)

krátký program, který může být zpracováván kdykoliv je k tomu vhodná příležitost. Je tvořen jediným souborem. Připravené dávky se pak řadí do front k dávkovému zpracování (batch procesing)

CASE

Computer Aided System Engineering - jedná se o komplexní programové systémy, vytvořené pro podporu práce týmu projektantů v jednotlivých fázích procesu vývoje IS

Connectionstring

Řetězec, který určuje připojení k databázi – musí obsahovat místo uložení databáze, port popř. název služby. Pro každý databázový systém se používá jiný connectionstring. viz www.connectinstrings.com

Constraint

specifikace vazební podmínky pro zajištění integrity

Databázová relace

konstrukt relačního databázového modelu, relace je vybavena pomocnou strukturou, které se říká schéma relace; schéma relace se skládá ze jména relace a jmen atributů a domén

DFD

diagram datových toků - představuje stručný způsob zachycení toků dat (informací) v systému; důležitou metodu pro porozumění funkcím systému a pro jejich vyjádření při komunikaci v týmu; umožňuje popsat vstup; vnitřní proces transformace (zpracování) dat v systému a výstup dat; má stromovou strukturu a rozlišuje několik úrovní popisu; technika rozvoje shora – dolů umožňuje měnit úrovně popisu a zobrazování podrobností.

Doména

definiční množina pro daný atribut, která se definuje příslušným datovým typem a popřípadě také constraintem

Entita

je zobrazení prvků z reálného světa (lidé, auto,...), který je popsán svými atributy-vlastnostmi. Jedná se o množinu objektů stejného typu. Při převodu na konkrétní relační DB se z entity stává tabulka, čímž se z ERM dostáváme k DRM.

ERD

Entity relationship diagrams - popisují **statickou** logickou strukturu databáze; existuje několik notací (OMG, ORM, UML, IDEF1X, Crow's Foot, Bachman, Chen, James Martin) těchto schémat

Foreign klíč

je sloupec databázové tabulky, který odkazuje na jiný sloupec (jiné tabulky). Hodnoty takového sloupce musí být shodné s některou z hodnot v sloupci, ke

kterému je klíčem. Vytváří se tak reference – odkaz. Podmínka shody se kontroluje při všech operacích nad databází, což se označuje jako referenční integrita.

JEDI

mezinárodní komunita vývojářů v Delphi snažící se svými komponentovými balíky zdokonalit vývoj v Delphi

Kardinalita

neboli vazebnost určuje počet výskytů jedné entity k druhé entitě. Např. Na faktuře může být jenom jeden dodavatel, ale dodavatel může mít několik faktur: Kardinalita je 1:N

Minispecifikace

je popis procesu na nejnižší úrovni hierarchického rozkladu. Upřesňuje, jaká je logika procesu. Zatímco DFD definuje strukturu procesů, minispecifikace musí vyjadřovat **postup**, jak jsou datové toky, které do procesu vstupují transformovány na výstupní datové toky.

Normalizace databáze

je proces, který postupnou dekompozicí původní relace vede k vytvoření množiny relací, u kterých: jsou eliminovány určité typy redundance, je zjednodušená kontrola integrity, nedochází k anomáliím při údržbě dat.

Perzistentní objekt

Objekt, který existuje i poté, co zanikl objekt (program), který jej vytvořil.

Popisná entita

není schopna samostatné existence, je svázána hierarchicky se silnou, nebo vazební entitou (položka faktury, kusovníky). Nadřazená entita je označována jako Master entita, podřízená se nazývá Detail entita.

Primární klíč

je pole nebo kombinace polí, jednoznačně identifikující každý záznam v databázové tabulce. Žádné pole, které je součástí primárního klíče nesmí obsahovat hodnotu NULL. Každá tabulka může mít definovaný pouze jeden primární klíč.

Silná entita

je entita, která charakterizuje reálný objekt schopný samostatné existence (zákazník, čtenář, kniha)

Validovat

ověřit správnost dat, např. vkládaných uživatelem

Vazební entita

vyjadřuje činnost, do které vstupují silné entity a většinou je podložena dokladem (fakturuje, dodává, objednává, půjčuje)

Úvod

Cílem toho bakalářského projektu je nalézt a zpracovat vhodné řešení databázového informačního systému šitého „na míru“ neziskové organizaci Společnost přítel přírody působící v Liberci a jeho okolí. Návrh databáze má suplovat nejběžnější administrativní úkony prováděné ve společnosti a tyto pak nabízet uživatelsky přívětivou formou v uživatelském rozhraní představovaném desktopovou aplikací. Kromě toho má nabízet možnost zobrazovat statistická data a umožnit exporty do aplikací MS Office a další funkce popsané v hlavní části projektu. Více o konkrétních pochodech ve společnosti píši v samostatné kapitole. Pochopení principů manipulace s daty v zákazníkově firmě a vztahů mezi jednotlivými pochody je nadmíru důležité u návrhu každého informačního systému, nejen databázového. V případě databází však toto platí dvojnásob, jelikož např. na rozdíl od dopravního systému, kde entity a relace tvoří většinou evidentní a jasné objekty (auto, semafor, silnice, čidlo) není určení těchto objektů u databázového systému natolik zřetelné. Málokterá společnost má své vnitřní úkony postavené natolik algoritmicky, aby se daly jednoduše předložit počítači. Většinou je nutné s novým databázovým informačním systémem některé kroky i algoritmizovat v reálné společnosti, na což si fungující společnost a její zaměstnanci těžko zvykají. Při návrhu databázového systému je také potřeba zohlednit mnoho výjimečných stavů, jelikož nelze jednoduše počítači říci: „Jednou výsledek počítej takto a podruhé si přeji, abys jej vypočítal jinak.“. Lze mu však říci: „V případě této situace proveď tuto akci a v odlišné definované situaci jinou“. Je také potřeba určit, co má databázový server provést automaticky bez vědomí obsluhy (uživatele klientského software) a k čemu potřebuje potvrzení, čímž dostane uživatel právo VETA nad algoritmicky definovanými postupy na serveru. V podstatě se dá shrnout výše zmíněné tak, že nejlepší databázový systém by byli schopni pro danou firmu navrhnout pouze dlouhodobě v ní pracující zaměstnanci, kteří úkony znají a z praxe vědí, jaké případy mohou a jaké nemohou nastat. Nejlépe určí identifikační klíče entit a vytvoří jasné relace.

Ovšem tito zaměstnanci bohužel nemají (většinou) natolik dobrou znalost programování a systémových návrhů, chybí jim důležitá systematičnost práce a zadání požadovaného informačního systému se pak stává nepřehledné, chaotické, zbytečně nafouklé a nejednoznačné. Proto je nutná úzká spolupráce návrháře databáze a zákazníka v době návrhu databáze. Návrhář má za úkol jednotlivé úkony zpřehlednit, co nejvíce zjednodušit a také myslet na budoucí možné změny v databázi. Což je, jak je jistě vidět, v praxi největší problém při návrhu databáze. Jakmile je jasná struktura, není v podstatě již překážka. Ovšem nalézt tuto funkčnost tak, aby zákazníkovi a počítači vyhovovala je v podstatě vizitkou a rukopisem návrháře databáze. Zákazník také většinou tuto fázi tvorby databázového informačního systému nevítá, nechce se mu platit čas programátorů, když se vlastně ještě nic neprogramuje.

Vrátím se zpět k mému návrhu databáze a konkrétní postupy komunikace a návrhu databáze pro neziskovou organizaci SPP popíši až v samostatné části.

Velice zjednodušený schématický popis navrhovaného řešení by zněl takto: Uživatel se s pomocí naprogramované **klientské aplikace**, instalované **na svém počítači**, **připojí k databázi ležící na serveru**. Databázový systém, v našem případě

Firebird 1.5, přijímá a správně reaguje na požadavky klienta. Návrh databáze má zajistit, aby byla za pomocí serverových funkcí zajištěna integrita dat. Nesmí tedy být možné smazat subjekt, je-li v databázi uložena například platba příslušející tomuto subjektu. Při změně primárního klíče záznamu entity „Akce“ se **musí zároveň kaskádně změnit** všechny podřazené záznamy odkazující se na tento primární klíč. Klientská aplikace musí nabízet jednoduché rozhraní pro důležité zálohování databáze. V praxi si to většinou správce IT podniku zajišťuje v pravidelném intervalu nějakým batch programem sám. Pravidlo z praxe však říká: „Záloh není nikdy dost.“.

Hlavní rysy mé klient-server aplikace a databázového systému se dají popsat stručně takto:

Databázový server – musí být dostatečně robustní a stabilní, musí zajišťovat správnost dat, řešit přístup více uživatelů najednou, poskytovat prostředky pro jednoduchou zálohu, obnovu a opravu dat a zajistit rychlé třídění a vyhledávání v datech.

Klientská aplikace – musí zprostředkovat příjemné GUI (Graphical User Interface) uživateli a uživatelskou formou nabídnout jinak vnitřně složité dotazy na server. Také nelze veškerou validaci přenechat na databázovém systému, tedy musí validovat vkládaná a editovaná data.

Protokol – standardizovaný komunikační protokol pro komunikaci mezi klienty a serverem.

1. Historie, současnost a budoucnost databázových informačních systémů

Problémem statistických úřadů bylo vždy zpracování a vyhledávání ve velkém množství dat. První původ databázových systémů se datuje do roku 1880, kdy si jej vynutila dnes již běžná praxe – sčítání lidu. Tehdy sčítání lidu ve Spojených státech trvalo 7 let. Mělo-li být to v roce 1890 rychlejší, bylo zapotřebí vynalézt nějaký automatický mechanizmus. Vynalezl jej němec Herman Hollerith (praotec společnosti IBM) a jednalo se o děrnoštítkový stroj. Bylo to vůbec první využití děrných štítků coby nosičů dat, ne programu. Zkrátil tak sčítání lidu na dobu šesti týdnů.

Za několik málo desetiletí pak vznikalo a vzniká několik dalších databázových modelů (ve smyslu abstraktním) v pořadí tak, jak jsou zde uvedeny:

1.1 Model správy souborů - FMS (File Management System)

Rozmach se přikládá době kolem roku 1950. Jedná se o velice primitivní databázový model, postavený na principu binárního souboru uloženého na disku, děrném štítku či magnetické pásce, mající v sobě uložené všechny záznamy. Výhoda tohoto řešení spočívala v jednoduchosti jeho pochopení. Bylo však nutné znát dobře strukturu ukládaných dat (u ostatních systémů nás vlastně nezajímá, kde jsou fyzicky data uložena), systém nezaručoval integritu a validaci. Jediné možné vyhledávání – sekvenční – bylo pro velké soubory zdlouhavé. Problémy nastávaly při požadavku čtení nebo zápisu z/do souboru více uživateli. Nebyl to tedy vlastně ani pravý databázový systém v jeho významu, jelikož databázový systém právě tyto věci zajišťuje. Bez nich se jedná pouze o jakousi elektronickou kartotéku.

1.2 Hierarchický model

Nejznámější implementací tohoto modelu byl systém IMS vyvinutý v šedesátých letech společnostmi IBM a NAA, a to pro realizaci skladového hospodářství v rámci projektu Apollo.

Tento model je analogií stromu z reálného světa – jako hlavní uzel považujeme jakýsi „kořen stromu“. Strom má pak různé větve (představující větve) a listy (představující záznamy). Programátor nemusí vědět, kde jsou data fyzicky uložena. Další výhodou je velice rychlý zástřel na data, známe-li číslo větve, listu. Vztahy typu 1:M.

1.3 Síťový model

Tento model se datuje k roku 1971. Funguje na principu ukazatelů vyjadřujících vztahy mezi jednotlivými databázovými položkami. Jelikož tyto ukazatele mohou být lineární nebo cyklické, lze s tímto modelem definovat i vztahy typu N:M, i když při velké DB to znamená vysokou režii. Model se neujal natolik, aby šlo uvést nějaké známé komerční aplikace na něm postavené, jelikož v tu dobu již vznikal relační databázový model. Nevýhodou síťového, stejně jako hierarchického modelu je obtížná a někdy i nemožná modifikace struktury databáze, nedostatečná dynamičnost.

1.4 Relační model

Tento dnes nejrozšířenější databázový model byl vynalezen již roku 1969. Doktor E. F. Codd tehdy definoval matematický aparát odvozený z teorie relační algebry. Relace jsou matematicky zjednodušeně podmnožinami kartézského součinu entit. Relační model zavádí pojmy jako je entita a relace, se kterými definuje vazby a data. Jako první model vůbec se stará také o integritu dat a tento pojem zavádí za pomocí použití triggerů, uložených procedur atd. . Tento a další pomocný aparát relačního modelu byl doplněn později.

S relačním modelem lze navrhovat téměř jakékoliv vztahy typů M:N, 1:N atd. Ne vždy však zcela vyhovuje potřebám aplikace. Např. na modelování datových skladů se raději používají multidimenzionální databáze. Více o relačních databázích uvedu v celé práci, jelikož je to hlavní stavební kámen celého projektu.

1.5 Post-relační model

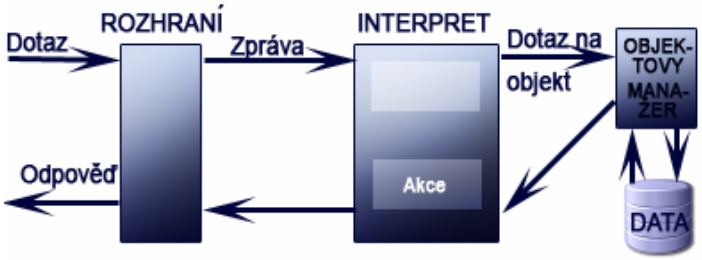
Dnešním světem hýbe objektové programování. Výhoda dívat se na kusy kódu jako na objekty z reálného světa je zřetelná. Není nutné psát znova stejné části zdrojového kódu díky dědičnosti a polymorfizmu. Je možné vyspělým způsobem pracovat s instancemi objektu a vlákny je výkonnostně upravovat na míru procesoru. Tyto výhody nemohl ignorovat svět databází. Byl tedy vynalezen objektový model, o kterém se rozepíši více v následujícím odstavci. Tento model je však velice mladý a problematika stability a integrity dat není jasně definována pro všechny případy. Kvůli velice dlouhé praxi s používám relačních databází (cca 40 let) je trendem přechod na objektový model pomocí tzv. post-relačních databází, spojujících výhody a zkušenosti z relačního modelu a výhody modelu objektového. V blízké budoucnosti můžeme očekávat, že se s tímto typem modelu uvidíme častěji.

1.6 Objektový model

Výzkum nad OODBMS(object-oriented-data-base-management-system) začal již na přelomu 60. a 70.let. Zatímco u předchozích modelů většinou nový nahradil ten starý, u tohoto modelu se počítá s paralelním vývojem spolu s relačními databázemi. Relační databáze jsou totiž jednoduché, standardizované, existují pro ně rozšířené a oblíbené dotazovací jazyky (SQL, QBE). Avšak mají i nevýhody vyplývající z jejich jednoduchosti – malou modelovací sílu. Znají pouze jednoduché datové typy a nelze do nich ukládat např. pole objektů. To mají umět objektově modelované databáze. Je vidět, že třídění a vyhledávání v takových datech nebude tak přehledné a snadné, jako je tomu u relačních databází. Ty totiž stále vynikají při třídění velkého množství jednoduchých dat. Pracujeme-li však se složitými datovými modely je vhodné použít objektový model.

Návrh objektově modelované databáze je vhodný zejména díky konzistentnosti s objektovými programovacími jazyky. Jednou vytvořená OODBMS je totiž poté většinou celého aplikačního programu, který dokáže jednoduše reagovat na akce v databázi. To značně zrychluje vývoj podnikových aplikací. Obdobou primárního klíče je u objektových databází tzv. OID – objektová identita. Není však odvozována od atributů objektů (u relačních = CONSTRAINT), ale je systémově generována jakmile je perzistentní objekt zapsán do databáze. OID je neměnný (narozdíl od relačních DB, kdy se na změnu reaguje akcemi typu CASCADE, RESTRICT apod.) a samozřejmě unikátní. Další výhodou oproti relační databázi je verzování každého objektu. Je tedy

zpětně možné sledovat jeho vývoj v čase. Také umí reagovat obsáhlým systémem na podmínky a akce – ne tedy pouze BEFORE a AFTER INSERT,UPDATE, DELETE, jako to umí relační databáze. Objektové databáze se již dočkali svých prvních standardů jako je např. ODMG (Object Database Management Group), má již standardizovánu obdobu SQL, a to OQL (Object Query Language). Jako příklady objektově orientovaných databází lze uvést: Objectivity, Versant, POET nebo CACHÉ.



Obr.1 - Architektura OODBMS

1.6.1 Příklad:

Zajímavý určitě je i náhled toho, jak vypadá taková práce s objekty v DB. Následuje příklad – rozdíl ve vkládání záznamů do tabulek (objektů) ROSTLINA a ZAHON z aplikace psané v Delphi pro relační model a v Javě pro objektový model:

relační model:

```
//předpoklad= vytvořený objekt IBSQL, mající správně nastavené
                    //připojení k DB
IBSQL.SQL.Text:="INSERT INTO ROSTLINA
                              (ID, DRUHJM, RODJM)VALUES(1,\„sněženka\",\"podsněžník\")";
IBSQL.ExecQuery; database.Commit;
//vynecháme inserty pelyňku a sedmikrásky pro zkrácení délky příkladu
IBSQL.SQL.Text:="INSERT INTO ZAHON(CISLO, M2)VALUES(1,25)";
IBSQL.ExecQuery; database.Commit;
//do relační tabulky představující záznamy typu M:N vložíme údaje o tom, jaký záhon má
//jaké květiny:
IBSQL.SQL.Text:="INSERT INTO ZAH_ROSTL(CISLO, ID_ROSTL)VALUES(1,1)";
IBSQL.ExecQuery; database.Commit;
IBSQL.SQL.Text:="INSERT INTO ZAH_ROSTL(CISLO, ID_ROSTL)VALUES(1,2)";
IBSQL.ExecQuery; database.Commit;
//atd.
```

objektový model:

```
Rostlina snezenka = new Rostlina("sněženka", "podsněžník");
Rostlina pelynek = new Rostlina("pelyněk", "černobýl");//nevynechávám –komentář zabere
Rostlina sedmikraska = new Rostlina("sedmikráska", null);// víc místa
Zahon zahon1 = new Zahon(1, 25);
zahon1.PridejRostlinu(snezenka);
zahon1.PridejRostlinu(pelynek);
zahon1.PridejRostlinu(sedmikraska);
db.set(zahon1); //uložení do DB
```

Uvedla jsem jen ty nejznámější modely v historii i současnosti. Vyvíjeny byly samozřejmě i další principy, které se ovšem neujaly tolik jako ty výše zmiňované.

2. Seznámení s datovými toky ve společnosti přátel přírody – pohled zákazníka

2.1 K čemu je informační systém zapotřebí

To, že se nezisková organizace SPP rozhodla hledat možnost vytvoření na míru šitého informačního systému je dáno nedostatkem podobných reálných aplikací na trhu. Ty nepočítají s jednotlivými výjimkami v chodu takovéto specifické společnosti. Při evidenci plateb u běžných firem totiž jednoduše platba přichází od fyzické či právnické osoby platebním příkazem, fakturou či hotově, je zdaněna a je s ní nakládáno dle zákona, tedy algoritmicky. To však není případ SPP. Jejich klienti nejsou totiž pouze plátci. Společnost rozlišuje několik typů subjektů: členové, sponzoři, dobrovolníci, předplatitelé jejich časopisu, brigádníky a spolupracovníky. S těmito lidmi vedou specifické komunikace, rozesílají časopisy a jiné materiály, docházejí jim od nich platby s určitými specifickými symboly, dle kterých poté tvoří statistické údaje a plány na příští roky. Některé subjekty dostávají časopis zdarma, jiné si jej musí předplatit a další jej dostanou v rámci jiné než předplatitelské platby jako bonus. Někteří lidé pomáhají prací – dobrovolníci, někteří penězi a někteří svým know how.

Spolu s SPP jsme vytvořili následující zadání tak, jak si myslí vedení společnosti, že fungují. Je v něm mnoho nejasných relací, které jsme museli později upravovat a definovat výjimky. V každém slově textu musí analytik sledovat podtext ve smyslu „Co – kdo – při jaké situaci – kam, popř. jak“.

2.2 Databázový model – subjekty a toky dat

2.2.1 Členové SPP

2.2.1.1 Kdo je členem

Členem občanského sdružení SPP se stane každý člověk (popř. i rodina či školní třída), který o to projeví zájem vyplněním a podepsáním členského listu a zaplacením ročního členského příspěvku ve výši minimálně 365 Kč ročně. Tento příspěvek může zaplatit jednorázově nebo v několika splátkách v průběhu roku. Část členů platí svůj členský příspěvek i formou trvalého bankovního příkazu s měsíční či jinou frekvencí.

2.2.1.2 Jak a kdy se členy komunikuje

V okamžiku, kdy se někdo stane členem SPP, tak mu je zaslán poštou dopis s poděkováním a základními informacemi o sdružení SPP. Dále je mu v průběhu roku zaslán v ideálním případě čtyřikrát časopis Lomikámen a zpravodaj pro členy Vnitrokámen. (Lomikámen a Vnitrokámen vycházejí současně). Z logiky věci by vyplývalo, že tato rozesílka je prováděna čtvrtletně vždy po třech měsících. Skutečnost je taková, že výtisk časopisu a následná rozesílka bývá dost nepravidelná. Někdy vyjde po třech měsících, někdy po čtyřech a někdy i ve zcela jiném intervalu.

2.2.1.3 Propadnutí členství

Členství propadne ve chvíli, kdy uplyne 1 rok od okamžiku, kdy člen zaplatil svůj poslední jednorázový členský příspěvek. V případě trvalého bankovního příkazu je určení propadnutí členství trochu složitější a někdy se musí řešit individuálně případ od

případu. Obecně platí, že propadne měsíc po ukončení měsíčního trvalého příkazu, tři měsíce po ukončení čtvrtletního trvalého příkazu apod.

2.2.1.4 Upomínky členů

První upomínkový dopis je rozesílán společně s rozesílkou časopisu Lomikámen a zpravodaje Vnitrokámen. Tato upomínku je rozeslána všem lidem, kterým propadlo členství od poslední rozesílky Lomikamene a Vnitrokamene.

Druhý upomínkový dopis je rozesílán opět s rozesílkou časopisu Lomikámen a zpravodaje Vnitrokámen všem lidem, kteří byli při předešlé rozesílce upomínáni poprvé a kteří ani po této upomínce nezaslali svůj další členský příspěvek.

Posledním typem upomínky je telefonát členům, na které máme telefonní spojení s dotazem, zda chtějí ve svém členství pokračovat nebo zda si přejí, aby byli jejich kontaktní údaje z databáze vymazány.

2.2.1.5 Reagující a nereagující členové upomenutí

Pokud upomínaný člen zašle svůj další členský příspěvek, pak ho opět zařazujeme mezi členy a komunikujeme s ním dále jako se členem. Pokud upomínaný člen ani po druhé upomínce (popřípadě telefonátu) svůj členský příspěvek nezašle, potom ho uložíme do složky archiv a dále ho již neupomínáme.

2.2.2 Dárci SPP

2.2.2.1 Kdo je dárcem SPP

Dárcem SPP je každý člověk, který věnuje občanskému sdružení finanční dar, ale zároveň nepodepíše členský list. Mezi těmito lidmi jsou například dárci na konkrétní strom, lidé, kteří přispěli v rámci veřejné sbírky na Nový prales apod.

2.2.2.2 Jak a kdy se s dárci komunikuje

Dárci je poslán ve chvíli, kdy pošle svůj dar, děkovný dopis nebo e-mail s otázkou, zda by chtěl od nás v budoucnu dostávat náš zpravodaj Vnitrokámen. Pokud odpoví ano, je mu posílán čtyřikrát ročně. (nepravidelně – vždy, když vyjde).

Jestliže dárce odpoví, že Vnitrokámen nechce, tak mu další korespondenci není prakticky rozesílána s výjimkou mimořádných fundraisingových akcí, kdy jsou osloveni s žádostí o mimořádný dar i některé „nekomunikující“ dárci.

2.2.2.3 Souběh dárce a člen

Jestliže některý ze členů přispěje kromě placení svého členského příspěvku například na strom či na Nový prales, komunikujeme s ním jako s každým jiným členem a je mu posílán i nadále Lomikámen a Vnitrokámen. Žádný druhý Vnitrokámen mu již posílán není. Komunikace členská je tedy z tohoto pohledu nadřazená komunikaci dárcovské.

2.2.3 Dobrovolníci SPP

Dobrovolníci SPP jsou lidé, kteří projevili zájem pomoci sdružení SPP svojí prací, za kterou dostanou obvykle stravu a v případě práce v terénu i ubytování. V zásadě rozlišujeme dva druhy dobrovolníků.

2.2.3.1 Zájemci o dobrovolnictví

Lidé, kteří dají SPP buď elektronickou nebo jinou cestou vědět, že by nám v budoucnu rádi dobrovolně pomohli a zúčastnili se našich akcí pro dobrovolníky. Jedná se o lidi, kteří nebyli v minulosti ještě dobrovolníky. Pokud se zájemci o dobrovolnictví nezúčastní během dvou let od okamžiku, co se přihlásili, žádné akce pro dobrovolníky, tak jsou přesunuti do „meziarchivu dobrovolníků“, odkud je koordinátor po zvážení všech okolností může mechanicky přesunout zpět mezi aktivní dobrovolníky nebo do archivu dobrovolníků. Pokud se během této doby některé z akcí pro dobrovolníky SPP zúčastní, pak se z nich rázem stávají „aktivní dobrovolníci“.

2.2.3.2 Aktivní dobrovolníci

Jedná se o lidi, kteří se zúčastnili alespoň jedné z dobrovolnických akcí SPP. Ve skupině aktivních dobrovolníků zůstávají po dobu tří let od okamžiku, kdy se zúčastnili poslední akce pro dobrovolníky. Pokud se pak žádné dobrovolnické akce po dobu 3 let nezúčastní, jsou přesunuti do „meziarchivu dobrovolníků“,viz odstavec výše.

2.2.3.3 Komunikace se zájemci o dobrovolnictví a s aktivními dobrovolníky

Všem zájemcům o dobrovolnictví i aktivním dobrovolníkům je posílána jednou až dvakrát ročně nabídka dobrovolnických akcí na daný rok s prosbou, aby si v ní zaškrtli akce, kterých se chtějí zúčastnit. Následně jsou jim před akcí, o kterou mají zájem, zaslány podrobnosti o dané akci. Před každou akcí tedy SPP dělá malou rozesílku lidem, o kterých ví, že by se jí rádi zúčastnili. Všem zájemcům o dobrovolnictví a všem aktivním dobrovolníkům, kteří o to požádají, je posílán zpravodaj Vnitrokámen.

2.2.3.4 Komunikace s dobrovolníky přesunutými do archivu

Dobrovolníkům přesunutým do meziarchivu dobrovolníků a do archivu dobrovolníků není posílán zpravodaj Vnitrokámen. Není jim zasílána ani žádná „papírová“ korespondence poštou. Pokud ale mají e-mail je jim zasílána i nadále nabídka dobrovolnických akcí. Pokud, se některé z nich zúčastní, stávají se z nich opět dobrovolníci aktivní.

Pokud dobrovolník nedá na sebe e-mailovou adresu, tak s ním komunikace přesunem do archivu končí. V okamžiku přesunu do meziarchivu dobrovolníků nebo do archivu dobrovolníků přestává dobrovolník dostávat Vnitrokámen.

2.2.3.5 Dobrovolníci a databáze SPP

Údaje o dobrovolnících v databázi SPP mají sloužit k tomu:

- aby při náhledu na kartu určitého člena či dárce bylo ihned poznat, zda je či není tento člověk zároveň dobrovolníkem. Dále musí být databáze SPP schopna ukázat ve kterých letech se daný dobrovolník zúčastnil alespoň jedné z dobrovolnických akcí.

- aby mohla z údajů v ní obsažených být správně zaslána rozesílka zpravodaje Vnitrokámen. To znamená, že Vnitrokámen smí odejít pouze zájemcům o dobrovolnictví a aktivním dobrovolníkům, kteří o to požádají. Zároveň nesmí dojít k tomu, že někdo dostane Vnitrokámen poštou několikrát. Vzhledem k tomu, že dochází k průnikům jednotlivých skupin v databázi a jeden dotyčný člověk může být např. zároveň aktivní dobrovolník, dárce a ještě člen, tak se může stát, že dostane 1 Vnitrokámen jako dobrovolník, 1 jako dárce a 1 společně s Lomikamenem jako člen.

2.2.3.6 Hierarchie rozesílek

Vzhledem k tomu, že je zpravodaj Vnitrokámen rozesílán členům, některým dárcům i některým dobrovolníkům a tyto skupiny se vzájemně překrývají, jsou pro tyto rozesílky stanovené následující priority:

Obr. 2 - priority pro rozesílky

1) Nejprve odejdou zpravodaje Vnitrokámen spolu s časopisem Lomikámen všem členům (to jest i všem členům, kteří jsou zároveň dobrovolníky či dárci) → „modrá rozesílka“

2) Poté jsou obesláni všichni dobrovolníci (s vyjímkou již obeslaných dobrovolníků – členů) → „zelená rozesílka“

3) Nakonec jsou obesláni dárci, kterým ještě nebyl Vnitrokámen poslán z jiného titulu (člena či dobrovolníka) → „červená rozesílka“

3. Datové toky ve Společnosti přátel přírody analyticky

3.1 Úvod – hlavní problémy analýzy u SPP

Největším problémem jsem shledala to, jak společnost postupně dodávala nové a nové požadavky na databázi. Nejednalo se pouze o přidání polí a funkcí. Některé nové informace přišly až natolik pozdě, že bylo třeba přehodnotit primární klíče některých entit. Taková zdánlivá maličkost však dokáže návrhem databáze pěkně zatřást, a tak jsem byla nucena znovu navrhnout nové relace a jejich referenční integritu. Taková situace např. nastala u pole specifický symbol.

V době, kdy jsme spolu s SPP definovali strukturu databáze mi existenci a zavedené používání specifických symbolů pro jednotlivé akce, rozesílky a platby pověření zaměstnanci SPP nevědomky zatajili a já pro každou tuto entitu vytvořila speciální vlastní primární klíč ID. Celkový počet entit poté, než jsem tato id sjednotila na základě nových poznatků do jediného specifického symbolu (a ty jsou nyní definovány

v samostatném číselníku) překračoval 10. Tím, že jsem použila nový způsob se počet entit snížil na 5, návrh struktury se zpřehlednil a více otevřel budoucím modifikacím.

Mnoho požadavků mělo za následek přidání celé nové entity, tedy návrh databáze více a více bobtnal. Zejména se jednalo o entity definující relace typu M:N. Návrh databáze má nyní po mnoha redukcích s sjednocení celkem 32 entit. Je velmi pravděpodobné, že se v průběhu života společnosti přátel přírody toto číslo ještě zvedne.

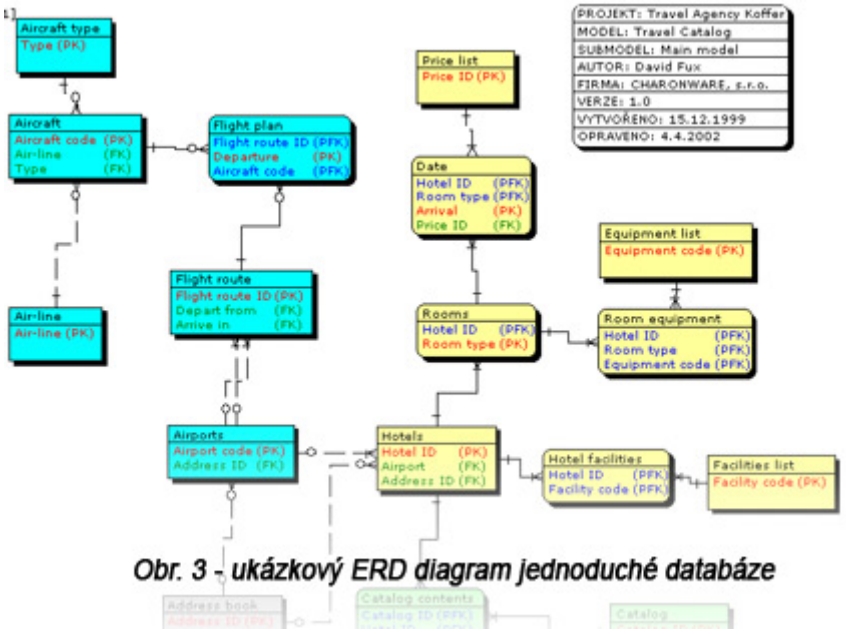
Dalším problémem bylo např. vysvětlit lepší řešení požadavku na bezpečnost ukládaných dat. SPP se bála, aby o svá data nepřišla a požadovala, aby všechna data byla ukládána 2x. Což je samozřejmě nesmysl – v relačních databázích a databázích obecně se snažíme duplicit vyvarovat. Ochranu dat pak zajistíme jejich častým zálohováním – archivováním.

Zajímavé byly i požadavky typu „Vždy, všechno a všude jde vybrat, označit a vytisknout.“ a jiné.

3.2 ERD diagramy a jejich aplikace na informační systém SPP

Entitně relační diagramy jsou rozšířeným nástrojem pro definici relačních databází. Díky grafickému zobrazení takových diagramů je návrh databáze více názorný, analytik udělá méně chyb a vývoj je rychlejší. ERD diagram pracuje s relačními databázemi obecně – definuje struktury entita, relace, atribut, doména, primární klíč, foreign klíč, constraint. Co jednotlivé struktury znamenají si může čtenář přečíst v slovníku pojmů. Nás totiž již zajímá praktická aplikace těchto struktur na databázi SPP.

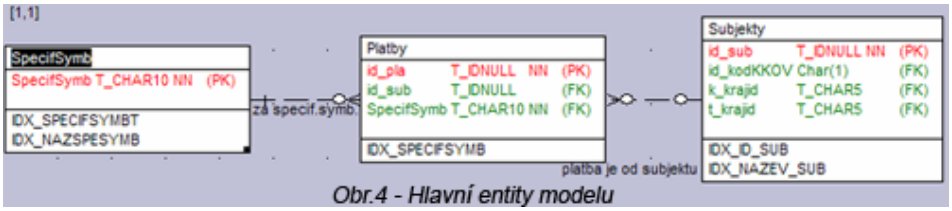
ERD diagram jsem vytvářela ve velmi oblíbeném programu CASE Studio 2.22. Tento program je dobrý v mnoha ohledech, od možnosti výběru všech známých databází, pro které je model navrhován přes přehlednou grafickou editaci a vývoj k speciálním nástrojům jako je Reverse engineering – z hotové databáze vytvoří její grafickou podobu. Mimo ERD diagramů lze v Case Studiu navrhovat také DFD diagramy, o kterých je pojednáno v samostatné kapitole. Náhled, jak vypadá ERD diagram vytvořený společností RKSoft jako sample k programu Case Studio je uveden pro ilustraci zde:



Obr. 3 - ukázkový ERD diagram jednoduché databáze

3.3 Entity – definice a nalezení entit pro databázi SPP

Dostáváme se k jedné ze dvou nejobsáhlejších kapitol – konkrétnímu návrhu databáze. U popisu definovaných entit se zmíním také o relacích mezi nimi, bude – li to k pochopení důležité. Konkrétní informace o relacích jako je referenční integrita apod. bych si ráda nechala až do další kapitoly. V některých případech však není možné výklad dělit.



Obr.4 - Hlavní entity modelu

3.3.1 Centrální hlavní entity a příslušné relace

Jádrem struktury jsou tři hlavní entity: <Subjekty>, <Platby> a <SpecifSymb>.

3.3.1.1 Entita SUBJEKTY

Občanské sdružení Společnost přátel přírody je založeno především na komunikaci s fyzickými osobami. Fyzická osoba bývá často členem sdružení (ročně přispívá 365Kč), dobrovolníkem (pomáhá prací na víkendových akcích) nebo brigádníkem apod. Kromě fyzických osob, společnost komunikuje s firmami ohledně sponzorských darů, spolupráci nebo čistě jako se zákazníkem (výstavba oplocenek, prodej stromků ze školek).

Obr.5 - Náhled entity Subjekty
(podrobné informace jsou v příloze)

Dále se SPP zabývá ekologickou výchovou - zajišťuje ji ve školách, zájmových kroužcích, rodinách a různých zařízeních.

Všechny je výhodné evidovat pod názvem <Subjekty>. Teprve atribut <typ_sub> rozhodne, která z podskupin je určitý záznam. Dle toho pak například školy nebudou mít vyplněné rodné číslo (NULL), nebo osoba IČ.

Důležitým atributem této entity je <prefix_id_spp> tzv. kód karty. Společnost tak určuje, je- li kód karty roven:

BE (číselně 1) - subjekt běžný - subjekt, který se nějak aktivně účastní dění ve společnosti a není tzv. "plonkový". To znamená, že má alespoň jeden výskyt v entitě dary, akce_subjekty, platby apod.

PR (2) - k probuzení - plonkový subjekt

PU (3) - půjčená - půjčená karta z databáze jiných společností - nutno chovat se opatrně

V zadávacím formuláři Subjekty nebude možné vložit subjekt s jiným kódem karty než je PR, PU. O přesun do BE se automaticky postará uložená procedura spouštěná triggerem při vkládání a editaci entit představujících nějakou aktivní účast

subjektu. Triggery se v příloze nazývají <AKCE_UCASTNICI_AIUD>, <KOMUNIKACE_AIUD> a také velice rozsáhlý trigger <AIUD_PLATBY>, kterému je věnována celá poslední kapitola.

Z dalších atributů zmíním pouze ty, které mi připadají důležité. Plný výpis atributů je možné nalézt v příloze. Entita <Subjekty> má 46 atributů. Atribut <neEmail>- slouží k evidenci zákazu posílání hromadných e-mailů danému subjektu, z důvodu jeho přání. <ID_KKOV> je cizím klíčem k entitě <KKOV>. Tato entita obsahuje při instalaci předvyplněné údaje ze statistického úřadu o dosaženém vzdělání osoby. Nemá samozřejmě význam jinde než při <typ_sub>='O' - osoba proto může být NULL.

Ještě se zmíním o primárním klíči této entity - <id_sub>. Jedná se o desetimístné číslo, které není pouze interní. Slouží k identifikaci subjektu i pro SPP. Toto ID vyplňují subjekty při platbách složenkou apod. Není možné jej však z programu vkládat ručně - program automaticky generuje ID inkrementací zapomocí funkčnosti FireBirdu. Ten má totiž možnost definovat takzvané generátory, které si pamatují poslední vkládané číslo. Jedná se o něco, jako je pro mikroprocesor čítač.

Cizím klíčem jsou ještě položky ukazující do entity <KRAJ> (kontaktní bydliště a trvalé bydliště). Mohou a nemusí být vyplněny. Jsou - li však vyplněny, musí existovat také v entitě <KRAJ>. Více o tom při popisu relací (parcialita, kardinalita).

Jelikož <Subjekty> jsou entita, ve které se bude jednoznačně nejčastěji vyhledávat a třídit a lze poměrně dobře určit dle jakých polí, vytvořila jsem dva indexy do entity. Index je struktura, která by se neměla využít zbytečně z důvodu času CPU. Indexy jsou vlastně ukazatele na konkrétní záznamy v tabulce. Jejich vytvoření a mazání znamená značnou režii. Naopak pro vyhledávání se dají použít velice dobře, jelikož hledání v indexovaných sloupcích je velice rychlé. Já jsem zvolila za indexy pole <NAZEV> a <ID_SUB>. Více indexu by již neznamenalo výkonnostní zlepšení, ba naopak. Indexy mají jména:<IDX_ID_SUB> a <IDX_NAZEV_SUB> a jsou oba řazeny vzestupně (ASCENDING). Při náhledu do databáze nad těmito atributy by měl databázový server upřednostnit použití indexů a výsledky vrátit rychleji, je-li dotaz podán ve stejném (ascending) řazení. Tím, že nemáme definované překrývající se indexy, bude výběr naprosto jednoznačný. Problém nastává např. v případě kdybychom měli jeden index pro skupinu atributů, např. <ID_SUB> a <PREFIX_ID_SPP> a také samostatný <INDEX> pro <ID_SUB>. V tom případě by mohlo vést k nejednoznačnosti, který index bude vlastně použit. Databázové servery dneška by však tyto nejasnosti měly umět řešit i za cenu, že budeme definovat, který index má optimalizátor použít.

Na náhledu je vidět další prvek používaný v ERD diagramech – domény. Jedná se o definiční množinu pro daný atribut, která se definuje příslušným datovým typem a popřípadě také constraintem. Můžeme využít ty, které zná, či (podporuje-li to databáze) si můžeme definovat vlastní. FireBird 1.5 domény podporuje, dokonce se v něm říká doménám stejně. Výhodou domén je např., že si mohu deklarovat typ a jeho typovou kontrolu např. `Create Domain T_pohlavi As Char(4) Check ((VALUE IS NULL) OR (VALUE IN('žena','muž')));` a tuto doménu pak kdekoliv použít. Budu-li chtít např. změnit kontrolu na `CHECK IN (VALUE IS NULL) OR (VALUE IN('žena','muž','více osob'))` mohu změnit pouze jedno místo a následně se mi změna projeví všude, kde je definován atribut s doménou <T_POHLAVI>.

3.3.1.2 Entita Platby

Platby jsou stěžejní entita databáze. Jedná se o evidenci plateb, a to jak kladných, tak záporných. Je - li <castka> kladná, jedná se o platbu subjektu s <id_sub> pro společnost. Je-li záporná, jedná se o platbu společnosti subjektu.

Subjekt však nemusí být vždy vyplněn.

Je-li <id_sub> NULL, jedná se o tzv. "neidentifikovatelnou platbu". To v praxi znamená, že na účet došla platba bez specifického či variabilního symbolu. Není možné původce platby dopátrat. Chceme ji však mít evidovánu. Do pole identifikace1 a 2 se zapíšou jakékoliv známky, dle nichž bude možné subjekt určit v budoucnosti. Např. dojde přihláška subjektu ke členství, kde bude zapsáno číslo účtu subjektu. Tento subjekt uložíme do databáze a nevíme, že nám již od něj platba před týdnem došla. Je-li toto číslo účtu uvedeno jako identifikace, je možné algoritmem platbu najít a přiřadit. O to se postará program.

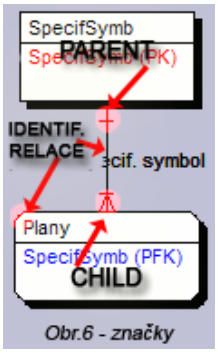
Stěžejní pole <SpecifSymb> je v poštovní terminologii synonymem pro specifický symbol. Dle něj je programově možné dopátrat platbu za členství, předplatné apod. či za jakékoliv evidované specifické symboly. Jelikož si společnost tyto specifické symboly sama vytváří, může filtrovat libovolné platby za jakoukoliv variantu platby.

Jako indexované pole jsem zvolila právě tuto položku, jelikož dle něj bude tříděno nejčastěji.

3.3.1.3 Relace fk_SubjektyPPlatbyCH

Díky tomu, že je výklad této relace zařazen jako první, je zde také možnost nalézt stručný výklad o jiných možných vlastnostech relací než právě těch, které jsou zvoleny pro tuto relaci. Při výkladu dalších relací již budeme znát význam jednotlivých možných nastavení a pouze si zdůvodníme, proč byla použita právě ta.

Jelikož relace bývají často definicí cizího klíče FK – foreign key (někdy PFK - primary foreign key) bývají pojmenovávány tak, že začínají zkratkou FK, následuje podtržítko. Pak je již způsob pojmenovávání relací individuální. Má vystihovat charakter vazby. Já jsem zvolila způsob jmenování relací dle masky: fk_ %parenttablename%P%childtablename%CH. Je pak jasné, která entita je závislá jak na které. Problém v nejednoznačnosti může nastat, je-li takových relací mezi dvěma entitami více. To se stalo např. u entit <Subjekty> a <Kraj>. V subjektech je totiž možné definovat jak kraj kontaktní, tak i trvalé bydliště. V takových případech volím masku individuálně např. fk_Kon %parenttablename%P%childtablename%CH. Je vidět, že měli – li by názvy vystihovat skutečně podstatu celé relace, jejich název by byl neúměrně dlouhý. V modelu je možné setkat se s přehlednějším názvem „platba je od subjektu“ avšak do DB se generuje jako fk_SubjektyPPlatbyCH.

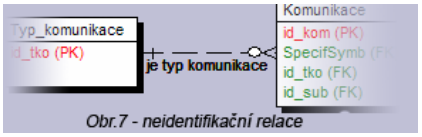


Obr.6 - značky

Možné typy relací jsou identifikační, neidentifikační, M:N, Self-relace a informativní relace.

Identifikační relace – je relace kdy primární klíč parent entity migruje do primárního klíče child entity, kde je pouze poté možno

jednoznačně identifikovat záznam entity. Child entita je pak zcela závislá na Parent entitě. Relaci značíme plnou čárou a child entitu se zaoblenými rohy, viz. obr.6.



Obr.7 - neidentifikační relace

Neidentifikační relace – Stejně jako u identifikační, ovšem vzniklý klíč není součástí primární klíče. Jedná se pouze o klíč cizí. Značí se čárkovanou čarou child entita nemá zaoblené rohy.

M:N – je v podstatě N identifikační relací z N parent entit vedoucí do jedné child entity, kde primární klíč entity je tvořen N PFK parent entit.

Self – relace – relace ukazující z jednoho záznamu na jiný záznam téže entity. Např. v entitě subjekty by se takto mohl zvolit atribut „manžel/manželka“, ve které by bylo uloženo <id_sub> jiného příslušného subjektu. Tento typ relace není v mém modelu použit, nebudu se o něm zmiňovat více.

Informativní – je speciální typ relace, který do databáze nic negeneruje a slouží k zpřehlednění modelu.

Mezi entitami <Subjekty> a <Platby> je relace typu „neidentifikační“. To znamená, že primární klíč parenta tvoří pouze cizí klíč u child entity. V entitě PLATBY vznikne pole <id_sub>, které je označeno FK. Viz. obr.4. Způsob propojení je přes primární klíč. Pozn. Další dva možné způsoby propojování entit jsou skrze UNIQUE atributy a skrze alternativní klíče. MS Access a podobné databázové platformy alternativní klíče nepodporují.

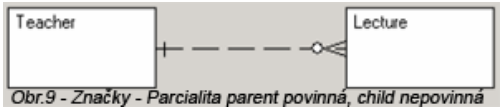
Do databáze je generován následující kód:

```
Alter Table Platby add Constraint fk_SubjektyPPlatbyCH Foreign Key (id_sub) references Subjekty (id_sub);
```

Parcialita parent resp. child = volitelnost relace vůči parent entitě resp. child entitě. Nejlépe vysvětlím – li to na 4 možných případech.

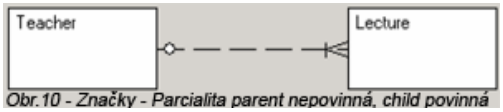
Obr.8 - Značky - Parcialita parent povinná, child povinná

Obr8. znamená, že učitel musí mít žáka a žák musí mít učitele.



Obr.9 - Značky - Parcialita parent povinná, child nepovinná

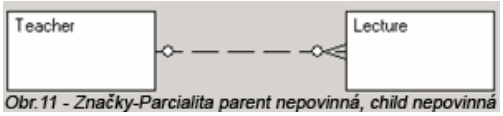
Obr 9. znamená, že učitel může mít žáka a žák musí mít učitele. Viz. Obr.4 – relace mezi SpecifSymb a Platby – Platba musí mít specifický symbol, tudíž FK SpecifSymb v entitě PLATBY je NN – NOT NULL



Obr.10 - Značky - Parcialita parent nepovinná, child povinná

Obr.10 říká, že učitel musí mít žáka a žák může mít učitele.

Obr.11 říká, že učitel může mít žáka a žák může mít učitele. Případ relace fk_SubjektyPPlatbyCH.



Obr.11 - Značky-Parcialita parent nepovinná, child nepovinná

Parcialita nemá význam při výsledném generování skriptu. Je vhodná pro určení struktury modelu. Výjimku tvoří nastavení NOT NULL pro cizí klíče. Primární klíč je totiž vždy definován jako NUT NULL, ale cizí klíče nemusí být vždy not null. Proto, je-li parcialita povinná, vytvoří se cizí klíč s NOT NULL atributem, při nepovinné může být ve sloupci uložené NULL. Case Studio tuto synchronizaci umožňuje zapnout. V mém modelu byla používána. Nemá vliv na již vytvořené relace, pouze na nové.

V relaci je použita parcialita child nepovinná, parcialita parent také nepovinná. Tedy SUBJEKT může mít PLATBU a PLATBA může, ale nemusí mít subjekt. V tom případě se z hlediska SPP jedná o „neidentifikovatelnou platbu“. FK <id_sub> není tedy v entitě <PLATBY> definován jako NOT NULL.

Kardinalita – vyjadřuje mocnost vztahu mezi výskyty entit. Jednoduše řečeno – kolik maximálně potomků může mít rodič a naopak. Vygeneruje se TRIGGER, který toto kontroluje, většinou BEFORE INSERT. Kardinalita může být 1:1, 1:N nebo N:M, kde N můžeme stanovit konkrétním číslem a toto číslo je pak v triggeru testováno. Velice názorná je grafická reprezentace:1:1 =  1:N=  . Jelikož SUBJEKT může mít více než jednu PLATBU, jedná se o kardinalitu 1:N, přičemž nechceme omezovat počet plateb konkrétním číslem. Jedna platba nemůže mít více plátců – subjektů.

Referenční integrita – pro relaci můžeme deklarovat referenční integritu určující reakce při akcích INSERT, UPDATE, DELETE parent nebo child entity. Jedná se o velice důležité nastavení, díky nimž je možné vůbec udržet data ve správném a chtěném stavu, i když v klientském programu na to myslet nebudeme. Generovat lze buď deklarativně nebo pomocí triggerů. „Deklarativně „ vytvoří SQL ve stylu:

```
Alter Table Platby add Foreign Key (id_sub) references Subjekty (id_sub) on update
cascade on delete restrict ;
```

Kdežto trigger ve stylu:

```
Create Trigger tu_Platby for Platby before update as declare variable numrows integer;
begin  /* Restrict child Platby, when reference to parent Subjekty is updated */
  if (new.id_sub is not null) then begin
        select count(*)
        from Subjekty
        where new.id_sub = Subjekty.id_sub
        into :numrows;
        if (numrows = 0) then exception moje_ceska_vyjimka;
 end
end
```

Tedy trigger zabere více místa, naopak ale můžeme přidat další funkcionalitu a volat konkrétní výjimky, které můžou být napsány v češtině. Kdybychom měli vícejazyčný program, dá se trigger větvit a zobrazit hlášku v uživatelově jazyce.

Referenční integrita je definovaná pro akce:

1. **Parent update** – *co se děje v případě kdy měníme PK záznamu v parent entitě*
    - Nastavíme-li ***NONE***, žádná akce se neprovede.
    - Při nastavení ***RESTRICT*** se při změně PK parent entity kontroluje, zda již není v DB uložen záznam s tímto PK. Je-li uložen, změna se neprovede a systém vyvolá výjimku. **To je chování, které více vhodné při pokusu o mazání – parent delete.** Pak nepovolím smazat subjekt, kterému je přiřazena platba. Nepovolit také změnu (parent update) ID subjetku není vhodné.
    - Vhodnější je použít nastavení ***CASCADE***. Tehdy se všechny FK<id_sub> změní kaskádně spolu se změnou PK v entitě SUBJEKTY.
    - Možnosti ***SET NULL a SET DEFAULT*** vyvolají akce, které dle názvu očekáváme.

2. **Parent delete** – *má stejné akce se stejným významem jako parent update*.
3. **Child insert** – *co se děje v případě kdy přidáváme záznam do child entity*
    - Možnosti jsou pouze dvě. Význam možnosti ***NONE*** je zřejmý.
    - Druhá je ***RESTRICT***, kdy nepovolíme vložit záznam do child entity v případě, že nemá odpovídající záznam v parent entitě. Jedná se o velice častý případ a je to případ i relace `fk_SubjektyPPlatbyCH` .

      Jakmile vkládám PLATBU a zadám <id_sub>, toto id_sub musí v DB existovat. Nemůžu přiřadit platbu neexistujícímu subjektu. Můžu však říci, že tato platba je neidentifikovatelná, tedy <id_sub> je null.
4. **Child update** – *co se děje v případě, že měním cizí klíč záznamu v child entitě.* Možnosti jsou stejné jako Child insert.

Výsledkem tohoto nastavení je následující skript. Udělám tak výjimku mému tvrzení, že skripty budou přiloženy pouze v příloze, převážně na CD. Myslím si totiž, že pro názornost je důležité alespoň jeden skript deklarující referenční integritu a pravidla relace uvést.

```
Create Domain T_IDNULL As Integer;
Create Table Subjekty  (
       id_sub T_IDNULL NOT NULL,
....... );
Alter Table Subjekty add Constraint pk_Subjekty Primary Key (id_sub);
Alter Table Subjekty add UNIQUE (id_spp);
Create Table Platby  (
       id_pla T_IDNULL NOT NULL,
       id_sub T_IDNULL,
.... );
Alter Table Platby add Constraint pk_Platby Primary Key (id_pla);

Alter Table Platby add Constraint fk_SubjektyPPlatbyCH Foreign Key (id_sub) references
Subjekty (id_sub) on update cascade ;
Create Exception except_del_p 'Záznam má podřízené záznamy! Nemohu je smazat!';
Create Exception except_ins_ch ' Nadřazený záznam neexistuje! Nemohu vložit záznam!';
Create Exception except_upd_ch 'Nadřazený záznam neexistuje! Nemohu změnit záznam!';

Create Trigger ti_Platby for Platby before insert as declare variable numrows integer;
begin  /* kontrola existence subjektu při vkládání do DB, není-li id_sub null */
  if (new.id_sub is not null) then   begin
              select count(*) from Subjekty where new.id_sub = Subjekty.id_sub
              into :numrows;
              if (numrows = 0) then exception except_ins_ch;
 end
end
^
Create Trigger tu_Platby for Platby before update
as declare variable numrows integer;
begin  /* kontrola existence subjektu při update záznamu v DB, není-li id_sub null */
  if (new.id_sub is not null) then begin
       select count(*) from Subjekty        where new.id_sub = Subjekty.id_sub
       into :numrows;
       if (numrows = 0) then exception except_upd_ch;
  end
 end
```

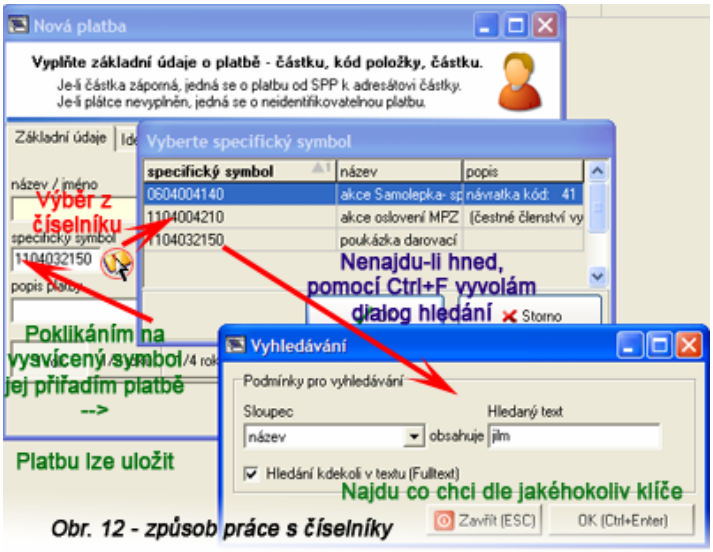
```
^
Create Trigger td_Subjekty for Subjekty before delete
as declare variable numrows integer;
begin /* Kontrola existence podřízených plateb, je-li subjekt mazán */
       select count(*) from Platby where Platby.id_sub = old.id_sub into :numrows;
       if (numrows > 0) then exception except_del_p;
end
^
```

Tedy na vše byly vygenerovány triggery kromě případu parent update, kdy stačí deklarativně podřízené záznamy změnit v kaskádě.

3.3.1.4 Entita SpecifSymb

Zahrnutí tohoto statického číselníku do databáze bylo dáno tím, že jsem zjistila jak si zaměstnanci SPP pamatují všechny ty kódy a symboly, se kterými jim platby docházejí.

SPP měla ve Word dokumentech na svém serveru tyto položky chaoticky rozepsané spolu s vysvětlivkami. Když pak po roce přišel požadavek zjišťovat údaje za určitý specifický symbol, následovalo dlouhé hledání v nepřehledných souborech. Je pravda, že systém, jakým jsou tyto symboly přiřazovány usnadňuje jejich hledání (např. první čtyři čísla určují datum akce), ale stále to ještě jaksi nebylo ono. Proto jsem zařadila entitu znázorňující jakýsi číselník specifických symbolů. Symboly, které bude potřeba vyhledat a vložit do určitého pole v klientské aplikaci bude možné nalézt přehledně v číselníku a poklikáním záznam automaticky vložit. Zaměstnanec SPP si již nebude muset pamatovat přehršle 10ti místných čísel. Vize byla asi taková (viz. obr.12).



Obr. 12 - způsob práce s číselníky

Entita tedy nemá žádnou zvláštnost, definuje tři hlavní (a několik dalších) atributů – SpecifSymb, Nazev a Popis.

3.3.1.5 Relace fk_SpecifSymbPPlatbyCH

Nebudu již zabíhat do přílišných teorií. S odkazem na předchozí relaci pouze vysvětlím, proč jsem volila ty určené vlastnosti.

Tato relace popisuje vztah mezi entitou <Platby> a <SpecifSymb>. **Každá platba musí mít specifický symbol** (i za předpokladu, že bychom v entitě SpecifSymb určili jeden <SpecifSymb> s názvem „neznámý“ apod.). To definuje **parcialitu parent povinnou, child nepovinnou**. Dále tak deklarujeme referenční integritu na **child insert** a **update** na „**RESTRICT**“. **Jedna platba musí mít jeden specifický symbol**, naopak **jeden specifický symbol může** být přiřazen **více platbám**. To definuje kardinalitu **1:N**. Relace je **neidentifikační**, vytváří v entitě <PLATBY> cizí klíč FK <SpecifSymb> NOT NULL, který není UNIQUE. Referenční integrita pro parent akce je dána v mém modelu

velice častým: parent update – cascade, parent delete – restrict. Všechny RESTRICT akce generuji zapomocí triggerů, cascade deklarativně, jelikož, zakazuji-li něco, chci zobrazit hlášku, proč tak činím.

3.3.1.6 Entita Plany

Tato entita nebyla určena jako hlavní avšak díky úzké návaznosti ne entitu SpecifSymb ji sem zařazuji.

Společnost přátel přírody má na každou akci, rozesílku, komunikaci určitý rozpočet a plán. Je velmi těžké ovšem určit, zda skutečně akce vynese tolik, jelikož nastávají komplikace. Příklad z praxe: při zjištění, že na určitém místě, kde se měli sázet stromky se nachází pod 20cm skála a práce trvají déle, zaměstnancům se zaplatí za delší dobu práce více peněz a celý zisk se tak snižuje. Je proto potřeba sledovat jaký plán byl dobře navržen, kde zisky byly menší než se předpokládalo, kde byly ztráty. Jako **referenční entita pro tyto algoritmy** programu je entita plány. Po dohodnutí plánu na celý rok uživatel programu vyplní tuto entitu a pak již bude sledovat, jak se jeho plány v průběhu roku naplňují.

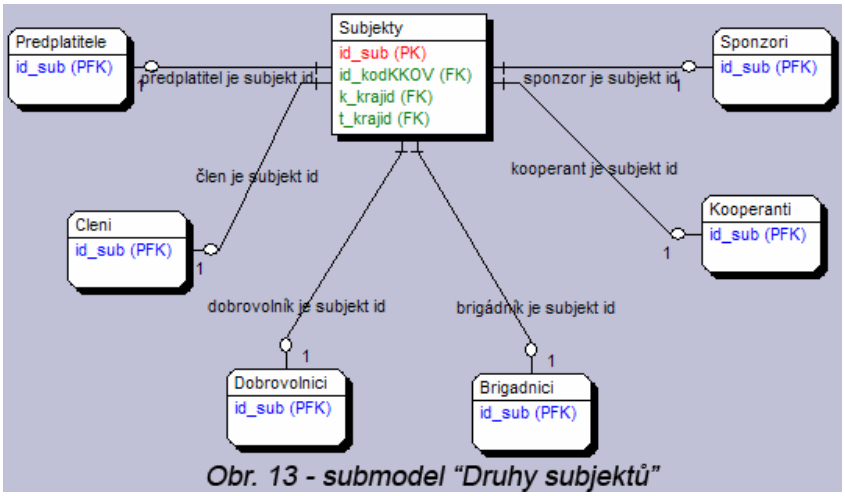
Jelikož akce, rozesílky i komunikace jsou definované třeba i stejným specifickým symbolem, je výhodné tvořit plány na určité konkrétní specifické symboly. Aby algoritmus ihned věděl jaký plán použít, **je primárním klíčem Specifický symbol** a jedná se o **identifikační relaci 1:N**.

3.3.2 Submodely - obecně

Jelikož v návrhu databáze je celkem 32 entit a 37 relací, nebudu popisovat v tomto textu všechny. V příloze je možné nalézt výpis všech relací, entit, atributů i domén apod. i se stručnými popisky, proč bylo zvoleno to či ono.

3.3.3 Submodel Druhy subjektů

Databázový model rozlišuje 6 druhů subjektů, se kterými je nějak zacházeno. Toto zacházení je odlišné, proto nešlo vytvořit pouze jednu entitu s atributem určujícím druh tohoto subjektu. Pozor na to, abychom si nyní nezaměnili typ subjektu – osoba, firma, škola, zařízení a druh subjektu – člen, sponzor, ...



Obr. 13 - submodel "Druhy subjektů"

Hlavní rozlišení spočívá v návaznosti jednotlivých druhů subjektů na entity PLATBY, DARY, Rozesilka_Adresati atd.

3.3.3.1 Členi

Jakýkoliv subjekt může být členem, naleznou- li se v entitě <platby> příspěvky s specifickým symbolem platby za členství. Je tedy vkládán automaticky algoritmem v klientském programu. Ten si zjistí, zda právě vkládaná platba s definovaným plátcem je členská z entity <VAR$PROGRAM>. Zjistí-li, že ano, provede insert daného subjektu do entity <CLENI>, nebo UPDATE, je-li v ní již (potom opraví hodnotu <pravoplatnyDo>, atd.) Algoritmus programu musí rozeznat, zda je subjekt členem, platí - li ročně, nebo trvalým příkazem měsíčně 30Kč. Jedná se o velice individuální rozlišování, proto je k částce Kč zaveden pro případ členských příspěvků **doplňující údaj „na počet dní“**. Tento údaj udává, jak dlouho bude subjekt od vložení platby právoplatným členem. Zaměstnanec SPP má tak vždy právo veta, zadat členství i člověku, který platí trvalým příkazem 30Kč měsíčně, ať má měsíc 30, 31, 28 či 29 dní. Dle původního plánu „ člen je ten, kdo platí 1Kč za 1 den “ by totiž vždy tento člověk poslední den v měsíci „spadl“ do 1. upomínkářů, což je nechtěný výjimečný stav. Někteří členi také dobrovolně přispívají i více (1000,- za rok) a neznamená to, že by si zaplatili členství na tři roky dopředu. Zaměstnanec SPP tedy zadá : částka = 1000,-, na počet dní = „365“. To že je člen nadmíru aktivní se projeví v atributu „aktivita“, který definuje dvou písmenné stavy typu „MČ“-malý člen, „NČ“ – normální člen, „VČ“ – VIP člen atd.

Právo veta oproti algoritmu zařazování člena do stavů „právoplatný, upomínka 1,...“ má mít vždy uživatel programu. Ten teprve přesouvá člena do jednotlivých stavů - právoplatný, v 1. upomínce, v 2. upomínce, spící, archivní. Program **mu pouze nabízí**, jak by dle daných pravidel společnosti sám člena zařadil, **dle tzv. přehledů adeptů**.

V entitě PLATBY představují členi zisk, tzn. pole "castkaKc" je kladná. Rozeznání členské platby zaručí pole "specifický symbol" končící na 10. Toto si však můžou SPP změnit v sekci „konstanty - členství“, kde se definuje také doba, za kterou se členi nabízí v přehledech adeptů. Tyto konstanty jsou uloženy již při generování skriptu databáze do entity VAR$PROGRAM ve smyslu klíč = hodnota.

3.3.3.2 Relace druhů subjektů s entitou Subjekty

Relace mezi entitou <Subjekty> a <Cleni> **je identifikační**, přidává do entity **UNIQUE primární klíč <id_sub>,** tedy nemůže se stát, že jeden Subjekt bude v entitě Cleni víckrát. V případě identifikační relace nelze definovat parent parcialitu jinak než na **povinnou**. Parcialita child je nepovinná z důvodu toho, že ne každý SUBJEKT musí být i členem. **Kardinalitu pak definujeme 1:1**, jelikož jeden člen nemůže být složen z více subjektů a naopak jeden subjekt nemůže mít více jak jeden záznam o členství. Entity propojuji primárním klíčem, jiné UNIQUE pole zde ani není definováno. Referenční integrita je jako většinou **parent update cascade, jinde RESTRICT**. Všechny relace mezi entitou SUBJEKTY a druhy subjektů jsou nastaveny stejně. Nebudu je tedy dále opakovat.

3.3.3.3 <Predplatitele> časopisu Lomikámen

Předplatitel časopisu Lomikámen je člověk, který má v entitě <PLATBY> odpovídající příspěvek za předplatné. Je tedy vkládán automaticky uloženou procedurou na serveru. Ten si zjistí, zda právě vkládaná platba s definovaným plátcem je předplatitelská z entity <VAR$PROGRAM>. Zjistí-li, že ano, provede insert daného subjektu do entity <PREDPLATITLE>, nebo UPDATE, je-li v ní již (potom opraví hodnotu zbyvaRoz). Předplatitelské platby mají tu zvláštnost, že se nevkládají „na počet dní“, nýbrž „na počet rozesílek“ a zaměstnanec SPP k tomu použije jiný formulář

vkládání platby. Zda již dostal rozesílku Lomikamene a kolik jich již obdržel zjišťuje uložená procedura na serveru spouštěná triggerem po insert, update, delete v entitě ROZESILKA_ADRESATI. Rozesílka Lomikamene je typ rozesílky, který mí ID = 1 a je šablonou vkládanou při generování DB, tedy nelze ji z programu měnit ani smazat.

Stejně jako u členů má uživatel programu právo veta nad rozhodnutím algoritmu a může i nemusí dát na jeho doporučení. To se projeví v políčku stav.

Členové dostávají časopis, který si předplatitelé platí, zdarma. Nejsou uvedeni v této entitě. Výběr adresátu rozesílky časopisu bude prováděn v několika krocích, kde členství má nejvyšší prioritu, předbíhá jej již pouze entita SubNeRozesilky, kde můžeme deklarovat, že typ rozesílky časopis LOMIKAMEN nechce některý subjekt individuálně odebírat (což nemá smysl u předplatitelů, ale u členů, kteří jej dostávají zdarma, ano).

3.3.3.4 <Dobrovolnici>

Dobrovolník je člověk, který se účastní akcí typu „dobrovolnická“ (<TYP_AKCE> <ID>= 3). Je proto plně svázán s entitou "akce". Tento člověk nemá nárok na honorář, pomáhá společnosti svojí nezpoplatněnou prací. Nemá tedy žádnou souvislost s entitou "platby". Dobrovolníka vkládá, edituje i maže manuálně uživatel programu. V tomto smyslu neexistuje žádná automatizace. Zařazování do „stavů“ dobrovolníka – aktivní, meziarchiv a archiv je také manuální, jelikož práce s dobrovolníky je velice individuální. Jediná automatizace je v programu po zmáčknutí tlačítka, se dosadí do entity <DOBROVOLNICI> všichni, kteří mají záznam v entitě <AKCE_UCASTNICI>, kde typ_akce je 3. O to se stará uložená procedura na serveru, která však není spouštěna triggerem, ale uživatelem po zmáčknutí tlačítka.

Zajímavostí je např. možnost definovat <typ_stravy> dobrovolníka a poté zjistit, kolik jezdí na akce vegetariánů, diabetiků, standardně se stravujících apod. a dle toho připravovat seznamy k nákupu potravin na tu kterou akci.

Příznak <exported> se nastavuje poté, co je dobrovolník po vložení do databáze mnou navrhované vyexportován do jejich Excelovské tabulky. Jedná se o nečisté řešení – vést tato data 2x, ale je to požadavek SPP a mají již zavedené některé postupy práce s dobrovolníky právě v MS Excel.

3.3.3.5 <Brigadnici>

Brigádník je člověk, který se účastní akcí a je za svoji práci finančně ohodnocen. Má souvislost s entitami <akce> i <platby>. Není však vkládán automaticky, ale ručně. Společnost přátel přírody chce u dobrovolníků i brigádníků sledovat vždy jiné atributy, proto jsou entity rozděleny.

3.3.3.6 <Kooperanti>

Subjekty (firmy i fyzické osoby), které se zavázali společnosti pomáhat předáváním know-how, půjčováním nářadí apod. Mají nárok na honorář za svoji práci, nebo nemusí požadovat nic. Tato entita je spíše evidenční, pro oslovování subjektů při potřebě konkrétní věci. Nemusí i může mít návaznost na entitu <platby>. Každopádně není nijak automatizována.

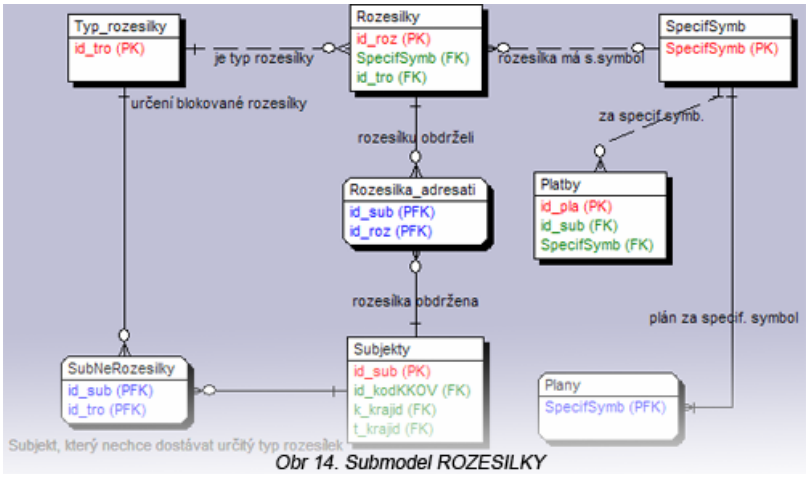
3.3.3.7 <Sponzori>

Přispívají dary (věcmi i službami) bez nároku na honorář. Společnost jim na svých tiskovinách vyjadřuje dík pomocí reklamy "Děkujeme sponzorům:".

Jsou svázáni s entitou <DARY>. Jedná - li se o finanční dar, který je třeba evidovat a zahrnout do příjmů společnosti, pak mají závislost i na entitě <PLATBY>. Tam se záznam dostane definovaným triggerem na AFTER INSERT do entity DARY. V něm se zkontroluje, zda dar má zaškrtlé pole, že je finanční a překopíruje se do entity <PLATBY> s příslušnou částkou v atributu <hodnotaKc>. Za typ platby dosadíme vždy HO – hotově a <ID_SUB> dosadíme id příslušného sponzora. Specifický symbol pro vkládaný DAR budou muset uživatelé vyplnit v dialogovém okně. Bude tedy potřeba zpřístupnit i rychlé vytvoření nového Specifického symbolu z tohoto místa. Do entity <SPONZORI> se subjekt může dostat buď manuálním vložením či po vložení záznamu do entity <DAR>, nebyl-li subjekt v entitě <SPONZORI> nalezen. Zde se nabízí použití triggeru na akci AFTER INSERT v entitě <DARY>.

3.3.4 Submodel Rozesílky

Ze tří druhů pohybů a manipulace se subjekty a platbami jsou rozesílky nejsložitější. Rozepíši se tedy o principech více a v textu o pohybech „Akce“ a „Komunikace“ se již budu více odkazovat na tuto kapitolu.



Obr 14. Submodel ROZESILKY

3.3.4.1 Entita <Rozesilky>

Entita <Rozesilky> má za úkol evidovat všechny rozesílky, které byly od začátku používání databáze zasílány.

Převážné množství komunikace SPP se subjekty je zajišťováno formou poštovních rozesílek. Množství poslaných dopisů je skutečně obrovské - ať jsou to poděkování za členské příspěvky, rozesílky časopisu Lomikámen předplatitelům a členům, informační letáky a podobně.

Pomocí atributů <datumCas> a <typ rozesílky> můžou uživatelé programu snadno vidět, zda již byli rozeslány 1. upomínky za nezaplacení členského příspěvku za první čtvrtletí (zajistí program). Je možné zjistit i kdo konkrétně rozesílku obdržel, a to pomocí entity <Rozesilka_adresati>.

Entita definuje atributy <nazev> a <popis>, které mají za úkol zpřehlednit uživateli informace o konkrétní rozesílce. Dále je možné **(ale nepovinné)** definovat Specifický symbol, tedy odkaz na entitu SpecifSymb, jejíž popis byl dán v předchozí kapitole. Relace s entitou SpecifSymb **je neidentifikační**, jelikož v entitě rozesílky může být více rozesílek pro jeden specifický symbol. **Parcialita parent je nepovinná,** jelikož můžeme vložit SpecifSymb NULL. **Parcialita child je také nepovinná** – specifický symbol může existovat, i když k němu není přiřazena rozesílka. FK <SpecifSymb> v entitě Rozesilky není definováno jako NOT NULL. Kardinalita je pak dána jako M:N.

Referenční integrita je **update CASCADE – delete RESTRICT**, z již mnohokrát vysvětleného důvodu. Zopakuji ještě, že **child insert a child update je RESTRICT**, jelikož si nepřejeme vložit neexistující SpecifSymb.,

3.3.4.2 Entita Typ_rozesilky

Jedná se o jednoduchý číselník typů rozesílek. Zatímco <SpecifSymb> určuje konkrétní poslanou rozesílku – např. adresná rozesílka časopisu dne 22.4.2006 <Typ_rozesilky> určuje obecně druh rozesílky.

Proč jsem nezkombinovala <typ_rozesilky> a specifický symbol? Specifický symbol je definován jako 10 čísel: AABBCCCCDD, kde C mají za význam právě určení typu rozesílek. Tedy filtr specifického symbolu dává typ rozesílky. Nabízí se definovat primární klíč jako řetězec o délce 10 ve stylu XXXX0001XX. Pak filtr XXXX000110 by určoval rozesílku za členství apod.

Je ale vidět, že bychom pak nebyli schopni říci, že tato členská rozesílka znamená rozesílku za 2.upomínku. V tomto směru nefunguje filtr specifického symbolu jako primární klíč.

Navíc při požadavku na správný chod programu nemůžeme dopustit, aby uživatel vkládal jiné, další kódy např. pro rozesílku členství 1.up. Pak by uživateli program nereagoval na všechny kódy správně (reagoval by pouze na ten, se kterým by programátor počítal). Program potřebuje alespoň nějaké stabilní body. Proto je tento číselník dodáván s atributem <jeSablona>, která určuje, zda byl daný záznam vložen při instalaci kvůli požadavku na správný běh programu nebo byl zadán uživatelem a ten jej může libovolně měnit. Statické záznamy pro program budou znamenat vodítko k informacím, zda daný subjekt již obdržel upomínku za členství, poděkování za dar apod.

Z výše uvedených důvodů jsem zvolila primárním klíčem atribut <id_tro>, přičemž několik typů rozesílek bude neměnných a program s nimi bude moci pracovat. Celkem jich je 10 a jsou ukázány v tabulce tab.1.

ID_TRO	NAZEV
0	rozesílka bez určení typu
1	časopis Lomikámen
2	časopis Vnitrokámen
3	poděkování - noví členi
4	poděkování - velký dar
5	poděkování - velká platba
6	poděkování - dar
7	poděkování - platba
8	členská upomínka č.1
9	členská upomínka č.2

tab.1. – typy rozesílek, šablony

S entitou **<ROZESILKY> je svázána neidentifikační relací**, **parcialitou parent povinnou** (rozesilka musí mít vždy typ, i kdyby se mělo jednat o typ s <id_tro>=0 viz. tab.1.), **child nepovinnou** (typ_rozesilky nemusí být v entitě <Rozesilky> zadán). To definuje kardinalitu 1:N. Referenční integrita je parent CASCADE – RESTRICT, child RESTRICT– RESTRICT.

3.3.4.3 Relace M:N mezi entitou Subjekty a Rozesilky definující entitu Rozesilka_adresati

Subjekty, kterým byla zaslána konkrétní rozesílka.

Uživatel programu samozřejmě nebude klikat jednotlivé subjekty, které mají rozesílku obdržet. Program jim v případě zmíněné rozesílky 1. upomínkářů nabídne dle entity <cleni>" a atributu <stav> subjekty, které ji mají obdržet. V případě předplatného a jiných případů podobně.

Primární klíč entity <Rozesilka_adresati> je tvořen **dvěma identifikačními relacemi vytvářejícími jeden primární klíč**: `Alter Table Rozesilka_adresati add Constraint pk_Rozesilka_adresati Primary Key (id_sub,id_roz);`

Parciality child jsem u obou relací zvolila jako **nepovinné**, jelikož Subjekty i rozesílky můžou existovat dokud jim nebyly child.

3.3.4.4 Relace M:N mezi entitou <Subjekty> a <Typ_Rozesilky> definující entitu <SubNeRozesilky>

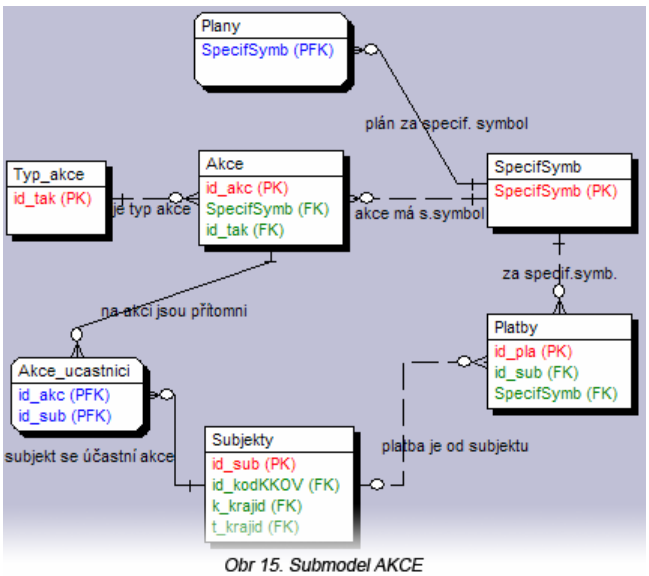
Tato entita definuje výjimečné stavy při rozesílkách, kdy nechceme, aby určitý subjekt dostal rozesílku určitého typu, jelikož si o to sám požádal. Implicitně tedy každý dostává všechny rozesílky, na které má právo. Zaměstnanec SPP však díky této entitě má možnost definovat výjimky.

Server bude kontrolovat požadovanou funkčnost pomocí triggeru na BEFORE INSERT entity <ROZESILKA_ADRESATI>. Najde-li, že rozesílku dostává člověk, jenž patří do tabulky <SubNeRozesilky>, automaticky jej vypustí z vkládání. Program by měl umět včas ukázat subjekty, které budou vypuštěny při vkládání.

Stejně jako v předchozím případě jsou obě relace identifikační a nastavení mají stejná.

3.3.5 Submodel Akce

Tento submodel definuje hlavní entitu <akce>. Typ akce je definován entitou <TYP_AKCE>, velmi podobně, jako tomu bylo u rozesílek. Relace mezi těmito entitami má stejnou definici, jako f případě relace fk_Typ_RozesilkyPRozesilkyCH



Obr 15. Submodel AKCE

3.3.5.1 Entita <Akce_ucastnici>

Podobně jako definuje M:N relaci v submodelu rozesílky entita <rozesilka_adresati>, najdeme v submodelu akce entitu <akce_ucastnici>, definovanou velmi podobně avšak mezi entitami <AKCE> a <SUBJEKTY>.

Do entity se zadávají konkrétní subjekty, které se akce zúčastnily. Jedná-li se např. o akci pro dobrovolníky a zjistí se, že vybraný subjekt není v entitě <dobrovolnici>, nabídne přiřazení subjektů do entity <dobrovolnici>. Proto zde má opět velký význam entita <Typ_akce>, ve které některé záznamy budou statické určené pro správný běh programu.

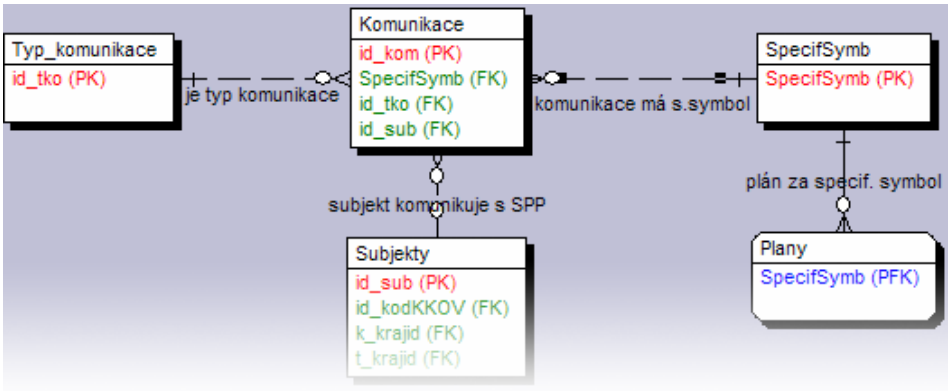
3.3.5.2 Entita <Typ_akce>

Je analogií entity <typ_rozesilky> s těmito standardními záznamy:

Typ akce	Popis akce
fundraisingová	pak SPP jde o dále nespecifikované zvýšení zisku - psaní projektů, žádosti EU atd.
sponzorská	jde o nové sponzory, akci pro stávající sponzory
předplatitelská	podobně jako členská, ovšem s číslem 20.
členská	speciální typ akce, její specifický symbol v současné době vždy končí na "10"). Proto bude program nabízet tento typ vždy, najde-li, že specifický symbol končí na 10. Jiná akce nemůže mít typ "členská". předplatitelská akce - podobně jako členská, ovšem s číslem 20.
dobrovolnická	sázení stromů, stavba oplocenek, akce pro získání nových dobrovolníků

tab.2. – typy akcí

3.3.6 Submodel komunikace



Obr.16. Submodel KOMUNIKACE

Je nejjednodušší z tří submodelů určujících pohyby v DB, jelikož nedefinuje žádnou relaci M:N. Entita <Komunikace> je svázána neidentifikující relací přímo s entitou <Subjekty>. Nelze tedy definovat hromadnou komunikaci (což je záměr). Pro hromadné komunikace se již předpokládá jejich zařazení mezi akce nebo rozesílky.

3.3.6.1 Entita <Komunikace>

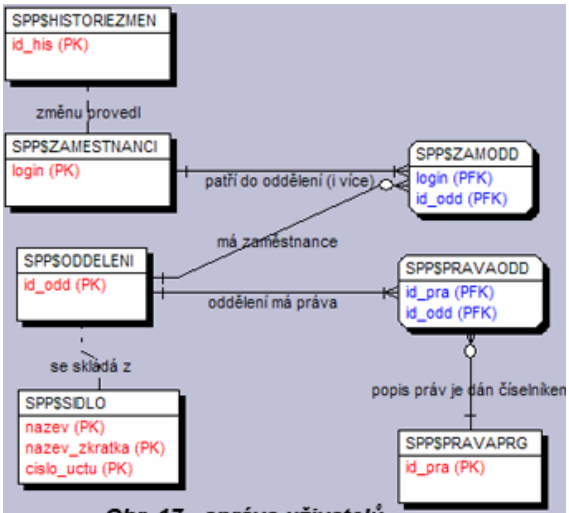
Evidence kdo s kým kdy komunikoval.

Jedná se o komunikaci mezi zaměstnancem a jedním subjektem z číselníku subjektů (pro komunikaci s více subjekty najednou se předpokládá spíše rozesílka dopisem). Komunikace ovšem nemusí být pouze s subjektem z číselníku. Je-li id_sub NULL, komunikace je označena za neidentifikovatelnou. V praxi to znamená, že došel dopis, ze kterého není možné určit odesílatele a přesto bych jej ráda uložila. Odesílatel se totiž třeba po týdnu sám telefonicky ozve a já pak již můžu označit konkrétní subjekt z číselníku subjektů.

Atribut <zodpovědný> nemá závislost na entitě <SPP$zamestnanci> z důvodu toho, že komunikující osoba např. SPP opustí, jeho záznam se vymaže a já chci nadále sledovat, s kým kdy v historii komunikoval.

Pole forma udává formu komunikace. Zda se jednalo o telefonát, e-mail, dopis, osobní návštěvu zaměstnance, osobní návštěvu subjektu a jiné.

3.3.7 Systémové entity



Obr. 17 - správa uživatelů

Slouží k uložení informací pro správný a bezpečný běh programu.

Dvě informativní entity <SPP$SIDLO> a <SPP$HistorieZmen> jsou doplněny pěti entitami zajišťujícími ukládání přihlašovacích údajů. Program se musí postarat, aby zejména heslo a jiné uživateli nepřístupné informace byly ukládány šifrovaně.

Relace submodelu jsou dvojího druhu: identifikační – pro entity k přihlašování a informativní pro konstanty a sledování změn. **Informativní relaci** mezi entitou <SPP$historiezmen> a <SPP$ zaměstnanci> jsem zvolila z důvodu toho, že chceme ukládat i historii akcí uživatelů programu – zaměstnanců, kteří již byly z DB smazáni. Nepotřebujeme tedy dodržet žádnou referenční integritu, a proto do databáze negenerujeme žádný skript s tímto související.

3.3.7.1 Entita <SPP$PRAVAPRG>

Tato entita je čistě statický programový číselník obsahující kódy a popisy možných přístupů uživatele skrze program do databáze. **Definuje tedy moduly aplikace a také nakládání s moduly.** Jedná se převážně o akce SELECT, INSERT, UPDATE, DELETE nad jednotlivými moduly programu. Díky právu SELECT, získá uživatel (patřící do oddělení) právo k modulu a může z něj číst. Má-li oddělení právo vkládat data do určitého modulu – např. jen subjekty a ne platby, musí mít v modulu subjekty práva na insert apod.

Nikdo do této entity nemůže zapisovat, je dána tvůrcem programu.

Atribut <sekce> označuje modul, o který se jedná. např. "cleni". Atribut <sekce_popis> se bude zobrazovat správci zaměstnanců při přidělování práv. např. "Členi"

Atribut <akce> udává jednu ze čtyř akcí, které je možné v dané sekci provést: "select, insert, update, delete". Nic nebrání dodat i další možné akce, např. „přesun členů do 2.upomínky“ apod.

Atribut <akc_popis> tyto programové kódy popisuje správci uživatelů "Prohlížení, Vkládání, Editace, Mazání" user-friendly.

Nabízí se tuto entitu dodávat celou šifrovanou, aby nebyly uživateli viditelné vnitřní konstanty programu.

3.3.7.2 Entita <SPP$ODDELENI> a entita <SPP$PRAVAODD>

Entita <SPP$ODDELEN>I definuje všechna oddělení, která má SPP. Každé oddělení má své ID, název, popis činnosti (generuje se např. při hromadné komunikaci), popř.telefon. Je svázána s entitou <SPP$PRAVAPRG> přes **kardinalitu typu M:N,** tedy **definuje další entitu** <SPP$PRAVAODD>. V této entitě správce uživatelů zvolí správná práva pro každé oddělení. <SPP$PRAVAODD> je tedy spojena dvěma identifikačními relacemi s entitami <SPP$PRAVAPRG> a <SPP$ODDELENI>.

Zajímavostí z pohledu celého návrhu DB je zde volba **parciality child povinné** v relaci <fk_SPP$PravaOddCHSPP$OddeleniP>, **jelikož nepředpokládáme existenci oddělení, které nemá jakákoliv práva**. S entitou <SPP$ODDELENI> je spojena informativní relací také entita <SPP$Sidlo>. Tato entita je vcelku nezajímavá, pouze definuje konstanty jako je IČ, telefon, email apod. pro hromadnou komunikaci.

3.3.7.3 Entita <SPP$Zamestnanci> a entita <SPP$ZamOdd>

Zaměstnanci společnosti jsou zároveň i uživatelé programu.

Zde v entitě <SPP$ZAMESTNANCI> jsou uloženy jejich (šifrované) přihlašovací údaje. O šifrování se stará program při ukládání dat. Entita <SPP$ZamOdd> definuje příslušnost zaměstnance do oddělení. Spojení s entitou <SPP$Oddeleni> jsem zvolila skrze **kardinalitu typu M:N** kvůli faktu, že málokdy má jeden člověk přesně jednu funkci v této malé společnosti. Je tedy možné přiřadit jednoho zaměstnance do více oddělení. Tím mu také přidělíme více práv.

Relace s entitou <SPP$ZamOdd> **je identifikační,** a tak jako v bodě 4.3.7.2 také zde nepředpokládáme existenci zaměstnance nepatřícího do žádného oddělení, tedy **parcialita child je povinná**. Jak již bylo psáno, tento údaj nemá stejně pro generování skriptu vliv.

Zajímavostí zde je **definice referenční integrity**, kdy jsem opustila klasický model *parent CASCADE-RESTRICT, child RESTRICT-RESTRICT*. Jak jsem napsala, nepředpokládáme existenci zaměstnance, který nepatří do žádného oddělení. Tedy na Parent delete není vhodné přiřadit akci RESTRICT, jelikož by nám při mazání program stále říkal, že má podřízené záznamy (v entitě <SPP$ZamOdd>). Proto je vhodné zde přiřadit spíše akci **CASCADE**, která způsobí, že při mazání zaměstnance se **rovnou smaže jeho příslušnost v odděleních**. Proti nechtěnému smazání tady musí přidat ochranu hlavně program, který se výstražnou hláškou zeptá správce uživatelů, zda chce opravdu uživatele smazat. Zbytek pak už bude na uvážení uživatele programu.

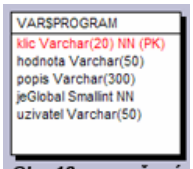
Po vytvoření databáze bude implicitně vložen jeden „admin“ s heslem „sppdb“, které si později správce změní. V programu nebo triggerem bude ošetřeno, že uživatel s loginem „admin“ nepůjde smazat a nelze mu vzít práva.

3.3.7.4 Entita <SPP$HistorieZmen>

Jedná se o jakýsi log akcí, který bude přístupný jen uživatelům, majícím k němu práva. Zapisovat se do něj budou hlavně akce typu “Uživatel Jarda dne 17.3.2006 smazal Subjekt ID 346778 – Franta Vomáčka”.

3.3.8 Entita proměnných

Název entity <VAR$PROGRAM> udává že se jedná o konstanty programu, které bude nutné držet v paměti. Např. poslední záloha DB = datum a kam byla provedena apod. nebo konstanty určující poplatek za členství, za předplatné. Ukládat se budou ve stylu <klic>=<hodnota>, kde atribut <klic> je primárním klíčem entity.



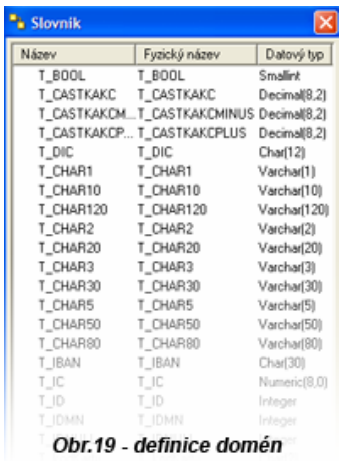
Obr. 18 - proměnné programu

3.4 Způsob modelování databáze, prostředky, CASE Studio

Samotnému modelování předcházelo dlouhodobé sbírání poznatků o chodu společnosti. Když bylo sebráno dostatečné množství dat, po krátkém seznámení s programem Case Studio, na který má TUL licenci jsem v něm mohla začít tvořit model. Jedná se o velice šikovný nástroj, který dokáže okamžitě reagovat na změny návrhu. Stačí v modelu změnit typ relace a výsledný generovaný kód vypadá úplně jinak. Návrhář samozřejmě není zbaven povinnosti znát význam jednotlivých generovaných příkazů a dokázat je upravit.

Na tomto místě ještě doplním nějaké další nastavení, které bylo zapotřebí udělat a z jistých důvodů jsem je nepřiřadila k popisu entit a relací.

3.4.1 Domény



Obr.19 - definice domén

V Case studiu lze definovat vlastní domény. Case Studio jim říká „slovníkové typy“. Tak jako se entitám po vyexportování do relační databáze již říká „tabulky“, v některých databázových serverech se říká doménám „datové typy“, jiné (FireBird) si ponechávají název „domény“. Doménou definujeme typ atributu (a popř. constraint – viz 5.3.1.1) přičemž doméně lze přímo přiřadit kontrolu typu check, default hodnotu, či NOT NULL/NULL.

Pro můj model jsem vytvořila celkem 27 slovníkových typů, které se po sléze generují do databáze jako domény.

Popíšu například nejjasnější doménu typu <T_BOOL>. K jejímu vytvoření mě vedl fakt, že v databázovém serveru FireBird chybí doména BOOLEAN. Vytvořila jsem tedy doménu s kontrolou check na hodnoty 0 a 1: CHECK VALUE IN(0, 1).

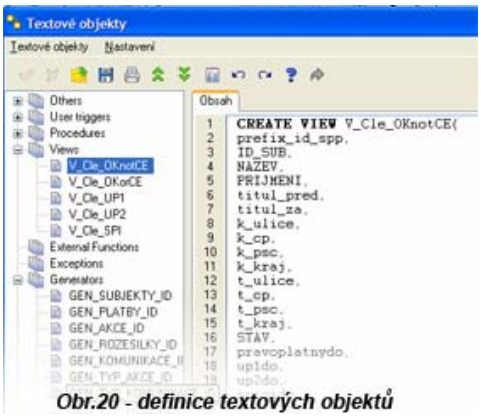
Další doména např. <T_CASTKAMINUS> musí vždy obsahovat zápornou hodnotu typu Currency. Přidáváme tedy check VALUE < 0. Jelikož firebird nemá doménu Currency, definujeme tuto doménu jako potomka domény Decimal délky 8 a plovoucí čárce v 2.desetinném místě.

Dále již domény a jejich určení vypisovat nebudu. Je snadné je pochopit z přílohy na CD.

3.4.2 Textové objekty

V Case Studiu nelze zadávat pokročilou funkčnost v uživatelsky přívětivém prostředí. Pro případ definice implicitních triggerů, generátorů, procedur apod. **vlastní textové zadávací prostředí**. Jelikož některé prvky potřebuji mít v databázi při každém

jejím vytvoření, je nutné je sem vložit, aby byly generovány spolu se skriptem. Jedná se také o implicitní inserty.



Obr.20 - definice textových objektů

3.4.2.1 Generátory, uživatelské triggery a uložené procedury

Jelikož databáze Firebird nemá jako např.MySQL constraint typu autoincrement, lze tuto funkčnost definovat pomocí generátorů. Ke každému automaticky inkrementovanému atributu (většinou se jmenuje ID), je třeba vytvořit generátor. Demonstrujme například na generátoru pro entitu <Platby>:

```
CREATE GENERATOR GEN_PLATBY_ID;
SET GENERATOR GEN_PLATBY_ID TO 0;
```

Tím jsme nastavili počáteční hodnotu generátoru. Nyní je pro autoinkrementaci nutné vygenerovat trigger na akci BEFORE INSERT:

```
CREATE TRIGGER PLATBY_BI FOR PLATBY
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID_PLA IS NULL) THEN
    NEW.ID_PLA = GEN_ID(GEN_PLATBY_ID,1);
END
^
```

Čím zvýšíme vždy po akci INSERT triggerem generátor o 1. Kdybychom chtěli o více, pozměníme skript na: `GEN_ID(GEN_PLATBY_ID,N);`

Dále je vhodné moci z programu zvyšovat aktuální hodnotu generátoru a mít v některých případech ještě před uložením záznamu informaci o jeho budoucím ID. V tom případě nepoužíváme zmiňovaný trigger. Vytvoříme proceduru serveru, kterou pak budeme z programu volat:

```
CREATE PROCEDURE SP_GEN_PLATBY_ID
RETURNS (ID INTEGER)
AS
BEGIN
  ID = GEN_ID(GEN_PLATBY_ID, 1);
  SUSPEND;
END
^
```

Pro ukázku postačí tento příklad. Všechny procedury, triggery a generátory je možné vidět v příloze na CD.

3.4.2.2 Views

V textových objektech je možno definovat views. Jedná se o náhledy do databáze pomocí složitých SQL dotazů, které se pro program tváří jako by to byla tabulka. Lze tedy dle sloupců třídit apod. Toho já ve složitých přehledech využiji. Příklad jednoho VIEW, které definuje náhled na členy, kteří jsou v první upomínce vypadá asi takto:

```
CREATE VIEW V_Cle_UP1(
prefix_id_spp,
```

```
ID_SUB,
NAZEV,
PRIJMENI,
titul_pred,
titul_za,
k_ulice,k_cp,k_psc,k_kraj,t_ulice,t_cp,t_psc,t_kraj,
STAV,pravoplatnydo,up1do,up2do,spicido,aktivita,cestny
)
AS
SELECT s.prefix_id_spp,S.ID_SUB,s.NAZEV, s.PRIJMENI, s.titul_pred, s.titul_za,
s.k_ulice, s.k_cp, s.k_psc, k.nazev, s.t_ulice, s.t_cp, s.t_psc, t.nazev,
C.STAV,c.pravoplatnydo, c.up1do, c.up2do, c.spicido, c.aktivita, c.cestny
FROM Cleni C
left join Subjekty S on S.ID_SUB = C.ID_SUB
left join Kraj k on k.nuts3 = s.k_krajid
left join Kraj t on t.nuts3 = s.t_krajid
WHERE C.STAV='UP1' ;
```

Většinou bývají SQL dotazy pro VIEWS ještě složitější. Pro tento SQL příkaz nemá ani význam VIEW vytvářet.

3.4.2.3 Ostatní

V sekci textových objektů s názvem „ostatní“ vkládám převážně skripty INSERT vkládající do databáze implicitní hodnoty, které v ní na počátku nesmí chybět. Jedná se zejména o hodnoty číselníků jako např. číselník PSČ – ten obsahuje 15482 záznamů a jejich vkládání trvá skoro půl minuty!

Některé entity, které v DB existují nebyly v textu popsány. Jedná se zejména o různé číselníky vydávané českým statistickým úřadem, majícím za úkol zpříjemnit co nejvíce uživateli programu práci. Na plný popis databáze odkazuji na přílohu na CD.

3.4.2.4 Výsledný model

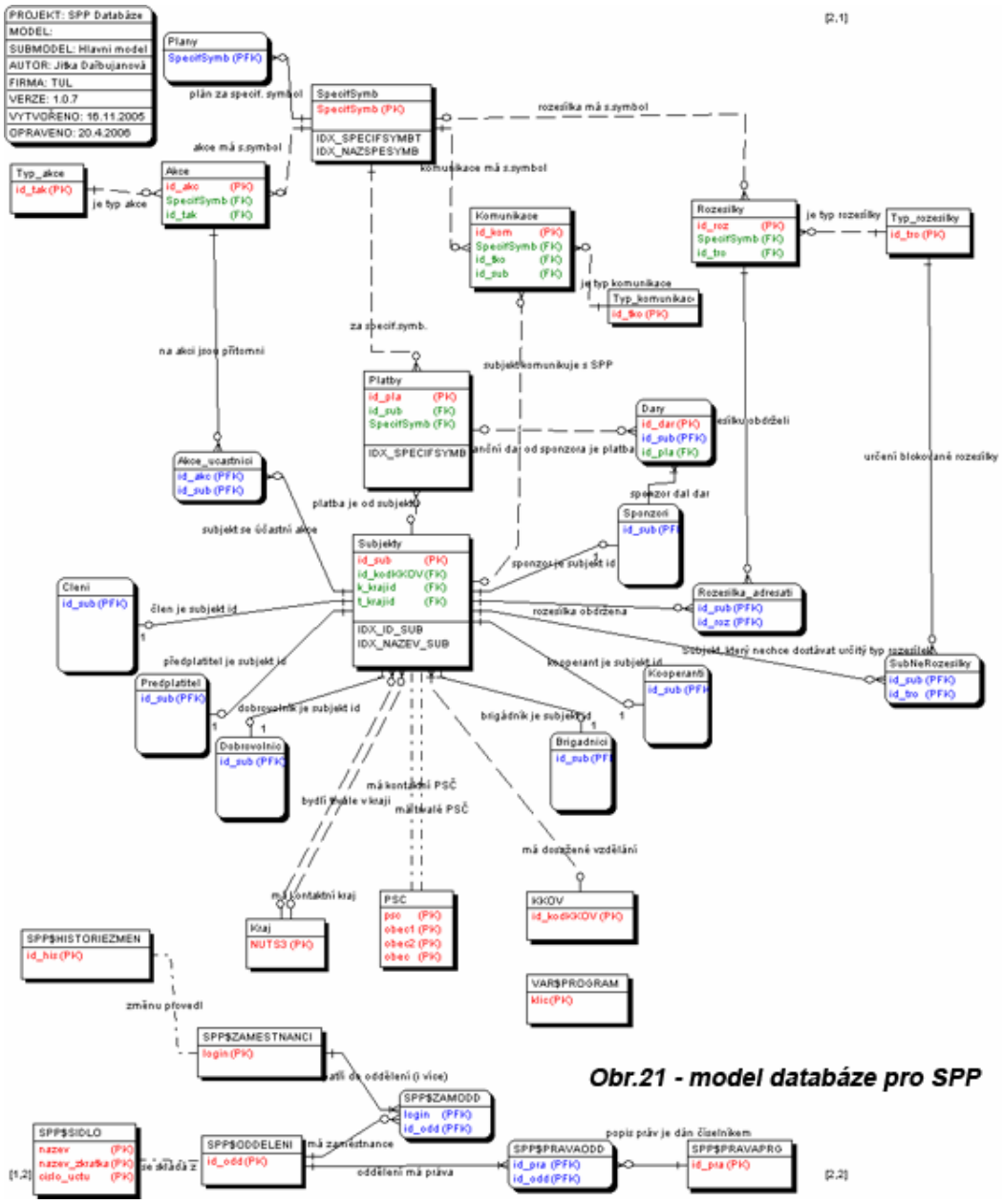
Několik statistických údajů doplním obrázkem 21 – náhledem na celistvý model databáze. Celý dobře čitelný obrázek je možné nalézt v elektronické podobě na přiloženém CD.

Počet slovníkových typů 27
Počet entit 32
Počet atributů 203
Počet relací 37
Počet indexů 5
Počet alternativních klíčů 0

tab.3 – statistika databáze

Výsledný skript má přes 18000 řádků, je to však dáno hlavně zmiňovanými inserty do entity PSC, která je skutečně veliká. Na skript potom zůstává **celkem 2500 řádků,** což je také úctyhodné číslo. Měl-li by si návrhář skript psát ručně, určitě by udělal více chyb, než když k tomu použije nějaký CASE.

Následuje náhled na výsledný model databáze, jednotlivé atributy jsou skryty pro zpřehlednění:



3.5 DFD diagramy – obecně, aplikace na vybrané procesy

Data flow diagram poskytuje kontrolní pohled na funkční model informačního systému. Zahrnuje v sobě funkcionalitu datových úložišť, tak algoritmů-procesů, které má zajistit program. Vytvoříme-li si nejdříve model funkčnosti aplikace, můžeme dle něj poté postupovat sekvenčně při psaní programového kódu. Jedná se však o zdlouhavou proceduru bez konkrétního výsledku (kromě jistoty dodržení správných pochodů), a tedy není v praxi využíván tolik, jako ER diagramy.

Funkční analýza se skládá z:

- Identifikace systémových funkcí
- Identifikace událostí
- Definice transakcí
- Popis transakcí

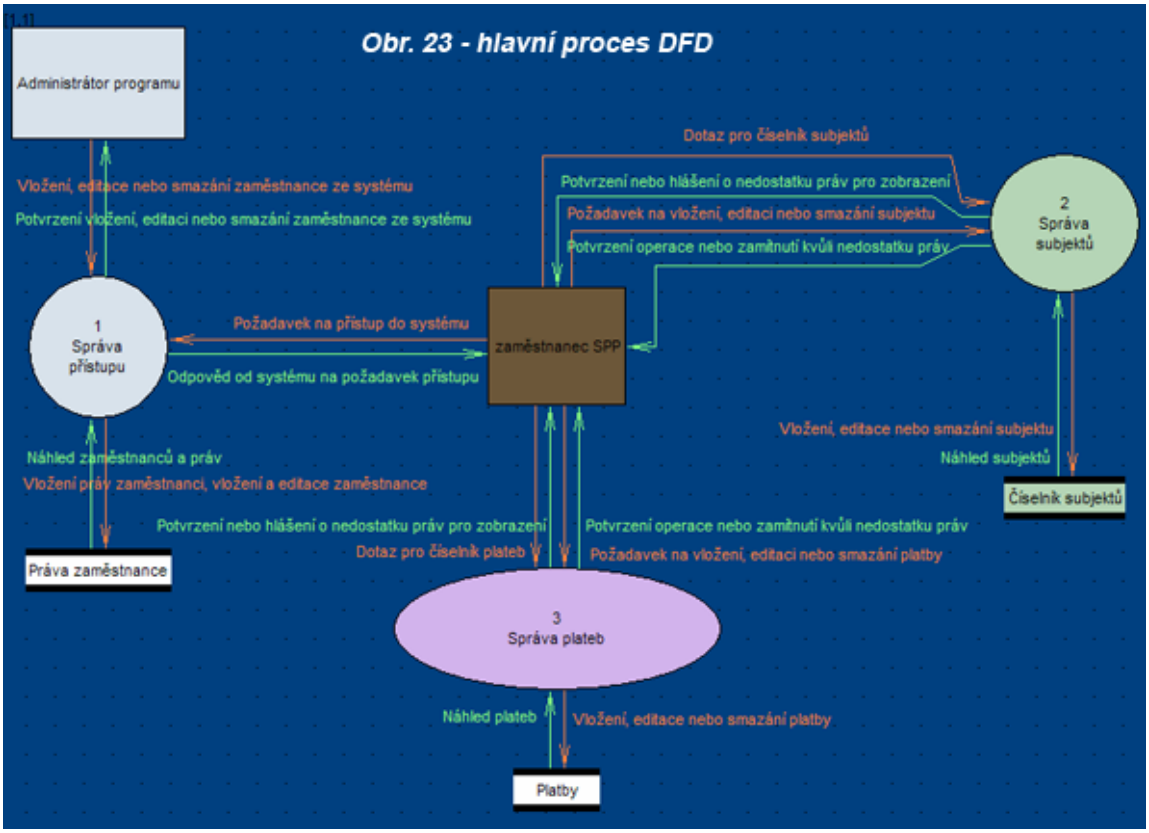
Obr.22 - stavební prvky DF diagramů

3.5.1 funkční analýza vybrané části aplikace

Mým úkolem nebylo vytvořit funkční model všech datových toků. Jelikož jich má aplikace velice hodně, při realizace takové analýzy by nezbyl čas na samotné programování aplikace.

Proto jsem si vybrala realizace funkčního modelu na dvou jeho částech – na správě uživatelů a na manipulaci se subjekty.

3.5.1.1 Hlavní proces

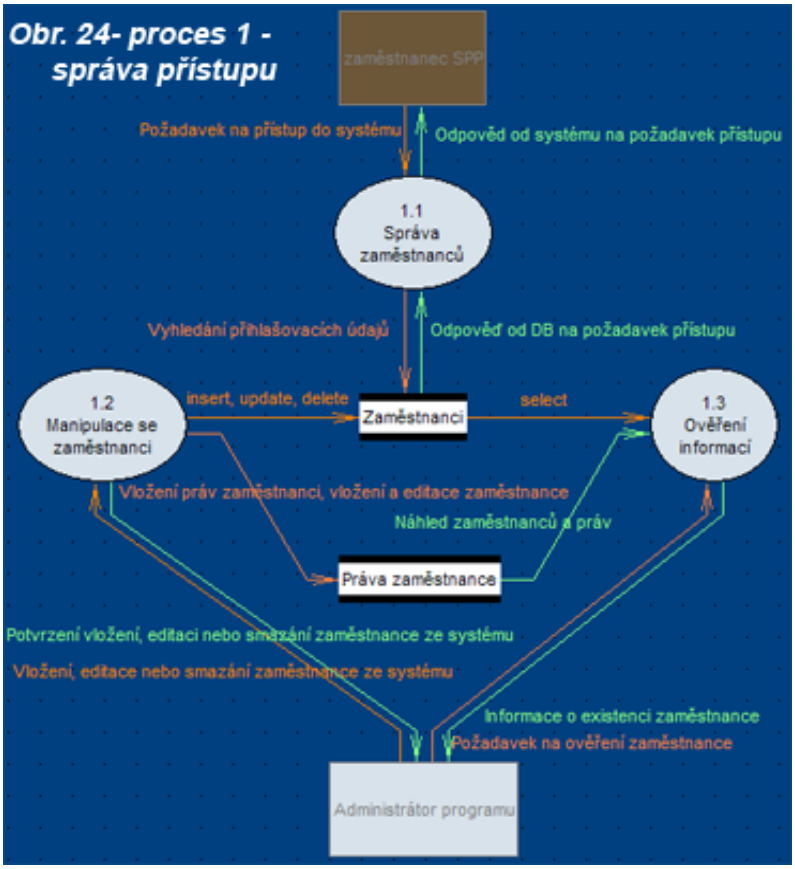


V hlavním procesu existují dva typy terminátorů – zaměstnanec SPP a administrátor programu – správce zaměstnanců.

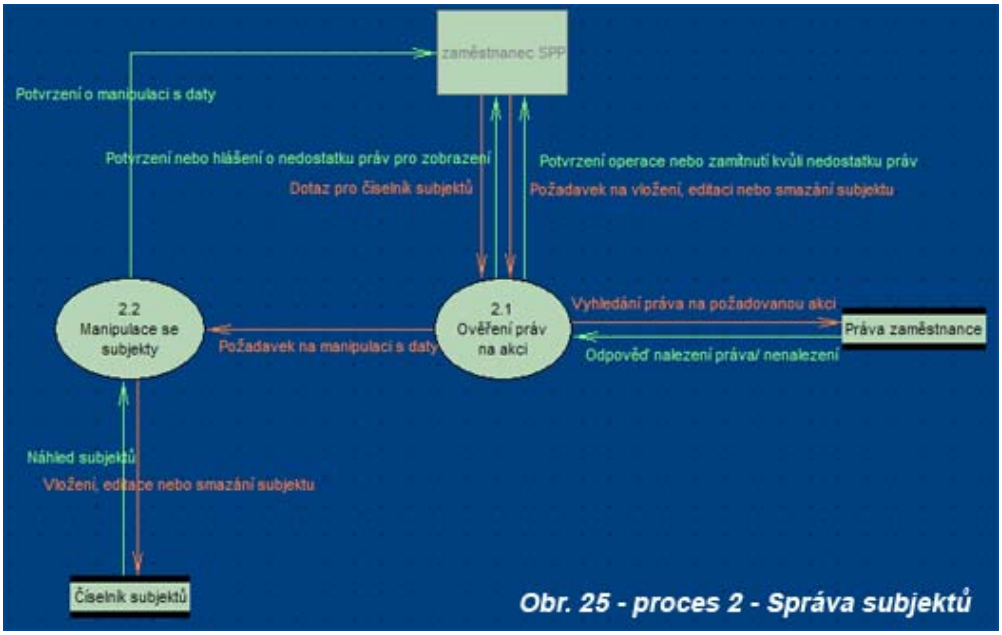
Administrátor programu skrze proces správa přístupu přiděluje práva zaměstnancům. Proces 1 není minispecifikací, a tedy je popsán konkrétně v bodě 4.5.1.2.

Detail procesu Správa subjektů je zobrazen v kapitole 4.5.1.3.

3.5.1.2 Proces Správa přístupu



3.5.1.3 Proces Správa subjektů



3.5.1.4 Proces Správa plateb

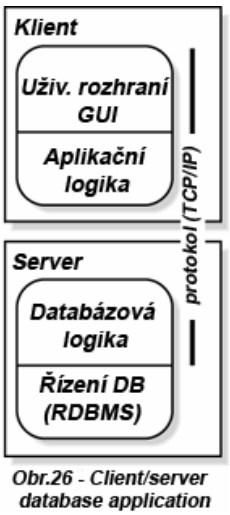
Díky podobnosti chodu procesů s procesem 2 jsem zvolila tento proces jako minispecifikaci, tedy nemá žádné detaily.

4. Problematika aplikací klient-server

Architektura mnou navrhnutého informačního systému je 3-vrstvá. To znamená že v sobě zahrnuje následující vrstvy:

1. presentační (interakce s uživatelem),
2. funkční (vlastní aplikace, bezpečnost, propojení se světem, kontrola...),
3. datová (vlastní data).

Jedná se o všeobecný standard, kterého jsem se držela.

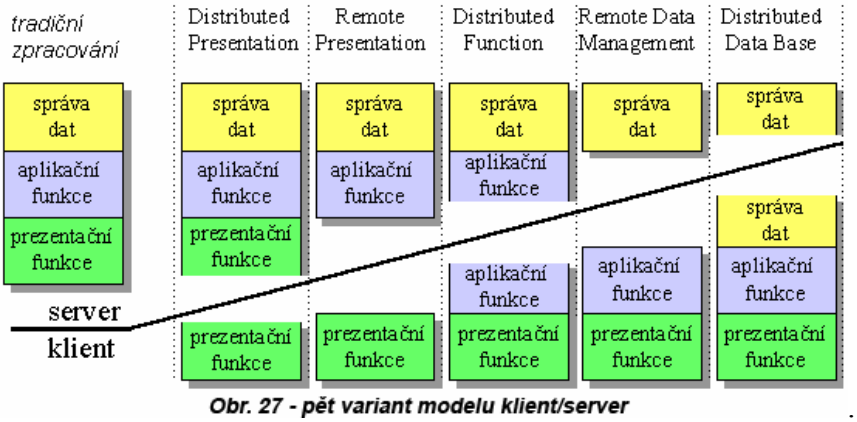


Obr.26 - Client/server database application

Klient-server je architektura, ve které jednotliví klienti komunikují vždy s centrálním serverem či servery, prostřednictvím kterého i komunikují s jinými klienty, pokud je to potřeba. Jedná se o pravý opak peer-to-peer architektury.

Program typu ***server*** obstarává samotnou službu (např. vyhledání v databázi), program typu ***klient*** zprostředkovává styk uživatele se serverem. Komunikace mezi klientem a serverem se děje podle přesných formálních pravidel, kterým se říká ***protokol***. O protokolech a způsobu připojení k serveru více v kapitole 5.1.

Klient-server aplikací je 5 variant. Ty určují, jaké množství úloh bude ležet na klientovi, jaké na serveru viz. obr.27



Obr. 27 - pět variant modelu klient/server

Moje aplikace klient – server je ve variantě Distributed Function, což znamená, že o část aplikační logiky se dělí klient se serverem. Většinu provádí klient, ale server dokáže přímo interagovat s akcemi uživatele typu neoprávněné mazání apod. chybovou hláškou pocházející ze strany serveru. Automaticky triggerem reaguje na vkládání určitých specifických plateb a generuje do tabulek (např. Členi, předplatitelé) nové záznamy.

4.1 Server

Jako serverovou část aplikace jsem vybrala **databázový server FireBird 1.5**. Firebird je open source relační databáze nabízející mnoho vlastností ANSI SQL-92 která pracuje na Linuxu, Windows, a řadě Unixových platforem. Je vyvíjena v C a v C++. Server SPP poběží na MS Windows XP.

K připojení k serveru budeme používat **protokol TCP/IP**. Jiné protokoly, které jsou FireBirdem podporovány jsou:

1. TCP/IP (celosvětový standard)

2. SPX - užíván Novellem
3. NetBEUI – není vyloženě síťový protokol, je to pomalý málo zabezpečený protokol Windows

Standardní **portem** pro DB server Firebird je **port 3050.**

Způsob připojení je dvojí: buď skrz **alias** – pak má connectionstring podobu textu zapsaného v systémovém souboru aliases.conf, nebo **přímou cestou k databázi** ve tvaru: SERVERNAME:d:\databases\db.fdb. Z toho je vidět, že soubor databáze (přípona .fdb nebo .gdb) musí ležet na lokálním disku serveru. K připojení používám TCP/IP protokolu. To by vzbuzovalo dojem, že v connectionstringu mohu zadat místo SERVERNAME jeho IP adresu, tak jak je to běžné, např.: 234.2.23.41:d:\databases\db.fdb. Tato funkcionalita však není ve FireBirdu přímo podporována, chceme-li ji využít, je potřeba užít jinou vrstvu např. WinSock2. V naší aplikaci si postačíme s vložením jména serveru.

K přihlášení je nutné znát přístupové údaje serveru. Standardně existuje v DB uživatel `SYSDBA`. Přihlášení je pak možné provést např. `CONNECT 'D:\DB.FDB' USER 'SYSDBA' PASSWORD 'masterkey';`

Přihlašování k databázi bude věcí klienta, proto bude popsáno v příslušné kapitole.

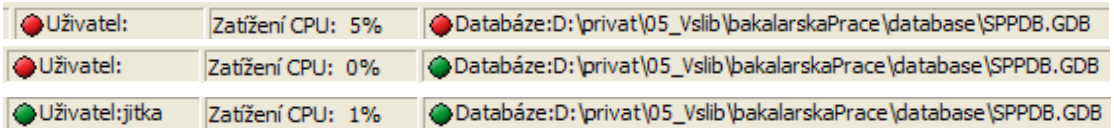
4.2 (tlustý) klient

Klientem bude aplikace s psaná v Delphi 6. Její přípona je standardní .exe. Klient se postará o presentační a valnou část aplikační vrstvy. Hlavním úkolem je připojit se k databázi a poskytnout správci systému možnost změny přihlašovacích údajů z programu. Přihlašovací údaje budou muset být uloženy samozřejmě na klientském počítači. Pro tento případ jsem vytvořila inicializační XML soubor, ze kterého bude aplikace číst přihlašovací údaje a který bude dodáván spolu s aplikací. Struktura XML bude vypadat následovně:

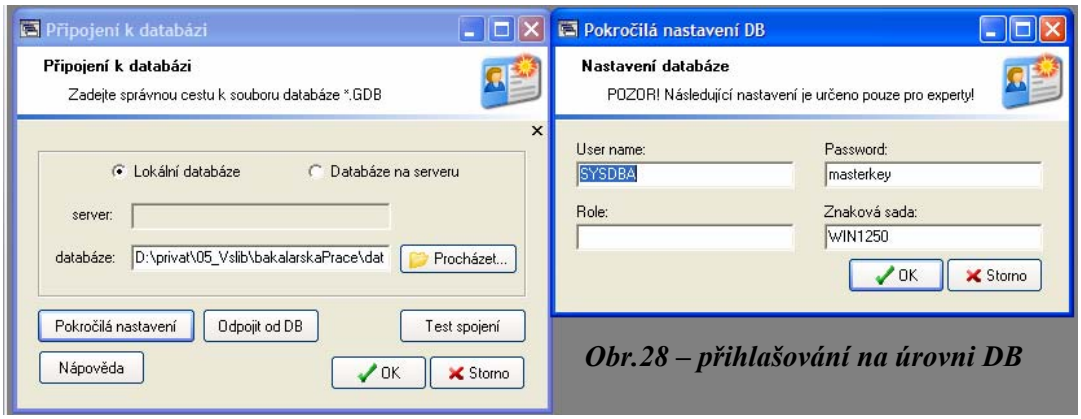
```
<?xml version="1.0" encoding="iso-8859-1"?>
<SPPDB>
  <Database>
    <user_name>SYSDBA</user_name>
    <password>masterkey</password>
    <sql_role_name/>
    <lc_ctype>WIN1250</lc_ctype>
    <connection_string>D:\bakalarskaPrace\database\SPPDB.GDB</connection_string>
  </Database>
  <User>
    <login>jitka</login>
    <password/>
  </User>
</SPPDB>
```

Jak je vidět, nalezneme zde dvojí přihlašovací údaje. Přihlašování do databáze jsem mohla řešit dvojím způsobem. Buď ponechat funkcionalitu na DB serveru a mít kromě uživatele SYSDBA další USER_NAMEs reprezentující jména zaměstnanců. Jistě by to bylo méně práce a přístup k datům by byl určen jednoznačně. Na druhou stranu bych neměla možnost přiřazovat k uživatelům data, která potřebuji, vkládat je do oddělení a přiřazovat jim práva do jednotlivých modulů. V tomto ohledu by byla manipulace s daty značně ztížená. Zvolila jsem tedy cestu vytvořit si správu uživatelů

sama. Program se bude k databázi připojovat standardním způsobem pomocí jednotného loginu SYSDBA – masterkey a bude v dolní liště indikovat stav připojení k DB. Zde po připojení k DB bude mít uživatel možnost se "přihlásit" a tím se mu zpřístupní "jeho" moduly. Toto přihlášení bude v dolní liště – statusbaru programu indikováno samostatně.



Přihlašovací okno (přihlášení na úrovni DB) vypadá následovně:



Obr.28 – přihlašování na úrovni DB

5. Návrh vybraných částí klientské aplikace, GUI

Dostáváme se k závěrečné části tvorby databázového informačního systému – programování aplikace. Na jejích bedrech leží veškerá verifikace vkládaných dat, příjemné uživatelské rozhraní, algoritmy vyhledávající v DB dle požadovaných souvislostí. Jedná se o poslední avšak **nejdelší část vývoje informačního systému**. Vlastně není poslední. Poslední by mělo tvoři **důkladné testování software**, které odhalí všechny chyby programu (všechny se neodhalí nikdy).

Snažila jsem se pokrýt většinu z požadavků na klientskou část databázového informačního systému. Myslím že se najde pár maličkostí, které jsem v uvedeném čase nestihla zprovoznit, ovšem většina požadované funkcionality je v programu zahrnuta.

K programování funkčnosti nestačilo pouze kopírovat funkčnost z DFD diagramů apod. Aby byl program uživatelsky přívětivý, byla jsem nucena vytvořit i vlastní balík komponent, který je v programu ve velké míře využíván. Delphi poskytuje totiž základní sadu komponent. Chcete-li od komponent však určité specifické chování, nastavení v Object inspectoru zpravidla nestačí. Více o tomto napíšu v kapitole 6.1.

Kromě svých a standardních komponent ještě používám komponenty JVCL, které jsou distribuovány open-source komunitou, která si říká Delphi JEDI. Mnohé z tohoto obrovského balíku jsou však nepoužitelné. U mnohých pěkně vypadajících komponent jsem narazila na chyby (mnohdy těžko odhalitelné), a tak jsem ve výsledku mnohdy použila hůře vypadající komponenty standardní.

Programování kódu mi zabralo nejvíce času. **Rozhodla jsem se však příliš stručný popis jednotlivých oken a funkcí vynechat.** Název bakalářské práce a zadání totiž udává jako hlavní část mé práce modelování databáze a realizaci jen určité části. Ponechám tedy kód k nahlédnutí na přiloženém CD. Popíšu jen základní koncept návrhu klientské části, popř. některý formulář vyzdvihnu. **Celkový počet formulářů a unit v projektu je 57.**

5.1 Programování vlastního balíku komponent v Delphi

Hned první ranou bylo, když jsem zjistila, že chování databázových komponent není uživatelsky přívětivé. Standardní TDBGrid neobsahuje žádnou funkcionalitu třídění, řazení, hledání v záznamech, dokonce neumí ani zarovnat sloupce dle šířky obrazovky.

Klasický TForm zase nejde nastavit tak, aby reagoval na stisk ESCAPE tak, že se uzavře. Je to drobnost, kterou jsou však uživatelé zvyklí používat a měli-li by ke každému zavírání potřebovat myš, bylo by to pro ně přinejmenším nepohodlné. A dodávat tuto funkčnost na každém formuláři na událost OnKeyPress mi přišlo zbytečné a neefektivní.

Další komponenta, která neměla funkčnost, kterou jsem požadovala byl TDBEdit. Líbila se mi komponenta TLabeledEdit, která již nad sebou obsahovala popisový label pole. Ovšem ta neumí pracovat s databází. Chtěla jsem, aby komponenta TDBEdit sama vygenerovala svůj popisek dle dat z datasource.

Proto jsem se rozhodla vytvořit svůj vlastní balík komponent jdDBD6p.dpk.

5.1.1 TJDDBGrid

TJDDBGrid je troufám si říct nejkomplexnější komponenta použitá v aplikaci. Její úkol je číst správně data z přiřazeného Datasource a ty uživateli přívětivě poskytnout v tabulkové formě. Podědila jsem ji z předka TDBGrid. Má tedy veškerou jeho funkcionalitu plus některou navíc.

Procedura TjdDBGrid.FixColumnsWidth dokáže přiřadit jednotlivým columns šířku dle titulku nebo dle nastavené width, ovšem rozšíří sloupce, nebo je zúží, pokud přesahují, na šíři obrazovky. Rozhodla jsem se z důvodu omezeného rozsahu bakalářské práce jednotlivé kódy ponechat k nahlédnutí pouze v elektronické podobě na přiloženém CD. Popíšu jen hlavní problémové body. V případě této procedury se jednalo hlavně o to, vyřadit sloupce, které nejsou Visible, tj. jsou momentálně uživateli skryty.

Dále bylo nutné zajistit funkcionalitu kliknutí na titulek Gridu a **dle kliknutého sloupce setřídit data v přidruženém datasetu**. TDBGrid však v sobě nemá ani takovou funkcionalitu, jakou je **změna kurzoru nad titulkem**. Bylo tedy nutné přetížit metodu OnMouseMove přidat do ní tuto funkčnost. Na proceduru TjdDBGrid.TitleClick(pColumn: TColumn); se provádí kontrola, zda je dataset aktivní a **spustí se procedura SortByColumn.** Tato procedura je složitá a vyžádala si mnoho oprav a testování. Procedura k přidruženému datasetu přidává **klauzuli ORDER BY** a název tříděných sloupců. V mém JDDBGridu lze třídit dle mnoha sloupců. Konstantou jsem ale zvolila maximum tři. Můžeme tak třídit dle sloupce příjmení a jsou-li shodná, pak ještě dle jména a je-li náhodou i toto shodné, pak např. dle rodného čísla. Problémů v tomto algoritmu bylo mnoho. Např. existují pole, které nejsou přímo odrazem sloupce v databázi, jsou takzvaně **Calculated** – tzn. jsou počítané na straně klienta, ale zobrazují se v Gridu i jinde, jako by to byly data z databáze. Taková data samozřejmě nejde vložit do klauzule ORDER BY. **Bylo nutné je v algoritmu oddělit a umožnit třídění alespoň v individuálních případech, kde jsem nastavila sloupce pro třídění.**

Nastala také situace, kdy jsem mezi s sebou při testování sloupce prohodila a třídění nefungovalo správně. Bylo potřeba přemísťování sloupců zakázat. To však TDBGrid neumí implicitně. Bylo třeba použít hack s přetypováním na TStringGrid.

Další záludností bylo vytvořit algoritmus na **třídění vzestupně a sestupně** – tedy zjistit, zda již v klauzuli toto pole je a je-li, přiřadit mu na správném místě DESC. Což

může být problém vezmeme-li si, že jednotlivá calculated pole jsou složena i z více polí třídění, tedy pro Calculated pole znamená „t_obec, t_ulice, t_cp“ kliknutí na jeden jediný sloupec „trvalá adresa“. Nicméně i to je v algoritmu vyřešeno.

Aktuální třídění je **potřeba také uživateli zobrazovat**. Bylo tedy potřebovat na tříděném sloupci zobrazit šipku dolů či nahoru, dle toho jak je tříděn a dle priority třídění také zobrazit číslo pořadí třídění. To mě donutilo vytvořit si soubor RES, ve kterém byly zakompilovány zdroje – obrázky BMP pomocí kompilátoru brcc32.exe. Zdrojový soubor s příponou .rc s kódem:

```
ARROWAZ Bitmap "arrowup.bmp"
ARROWZA Bitmap "arrowdown.bmp"
```

se překompiluje v Command line příkazem `brcc32 img.rc -32` , kde 32 určuje kompilaci pro 32bitové Windows a není nutné jej psát, je to default.

Pak v aplikaci z tohoto vkládaného souboru pomocí direktivy `{$R img.res}` získáme data do ImageListu, ze kterého jej v případě potřeby na OnDraw získáváme:

```
//vytvoř obrazky do title sort
FDrawArrowImgL := TimageList.Create(self);
vImg := Tbitmap.Create;
vImg.LoadFromResourceName(Hinstance, ,ARROWAZ‘);
FdrawArrowImgL.Width := vImg.Width;
FdrawArrowImgL.Height := vImg.Height;
FdrawArrowImgL.AddMasked(vImg, clFuchsia);
vImg.LoadFromResourceName(Hinstance, ,ARROWZA‘);
FdrawArrowImgL.AddMasked(vImg, clFuchsia);
vImg.Free;
```

Malá pomocná funkce zajišťuje **sběr všech viditelných titulků columns** – hodí se např pro jednoduché přidání názvů sloupců do ComboBoxu : `function TJDDBGrid.GetTitleStrings: TStrings;`

Dále jsem zařadila implicitně do **TDBGridu funkcionalitu vyhledávání.** Klávesové zkratky pro start hledání jsou přístupné, je-li Grid Focused a jejich význam je následující:

Start hledání: Ctrl+F, další hledání F3.

Na KeyDown se kontroluje stisk těchto kláves a je-li to možné, je zahájeno příslušné hledání. Nalezený záznam je zobrazen na obrazovce a jeho řádek je vysvícen. Hledá se dle sloupce zadaného v dialogu, který se objeví ihned po stisku CTRL+F.

Obr.29 – vyhledávání na úrovni TJDDBGridu

Vyhledávat se může buď na začátku slova nebo fulltextově. Celý kód, který není ani v tomto případě krátký je opět možno jej nalézt na přiloženém CD.

5.1.2 TJDDBEdit

Pro zrychlení práce jsem si vytvořila odvozenou komponentu TJDDBEdit, které jsem ovšem rovnou přiřadila její TLabel. Je to jakýsi kompromis mezi TDBEditem a TLabeledEditem. Výhodou je to, že je-li Datasource tohoto editu aktivní, automaticky se do labelu dosadí text z „DisplayLabel“ property datasetu. Není třeba tedy popisovat každý label v aplikaci vícekrát.

5.1.3 TJDEdit

V některých případech jsem potřebovala přiřadit do editu property ModalResult, kterou má standardně TButton. Z tohoto důvodu jsem odvodila tuto komponentu z komponenty TEdit a novou vlastnost doprogramovala.

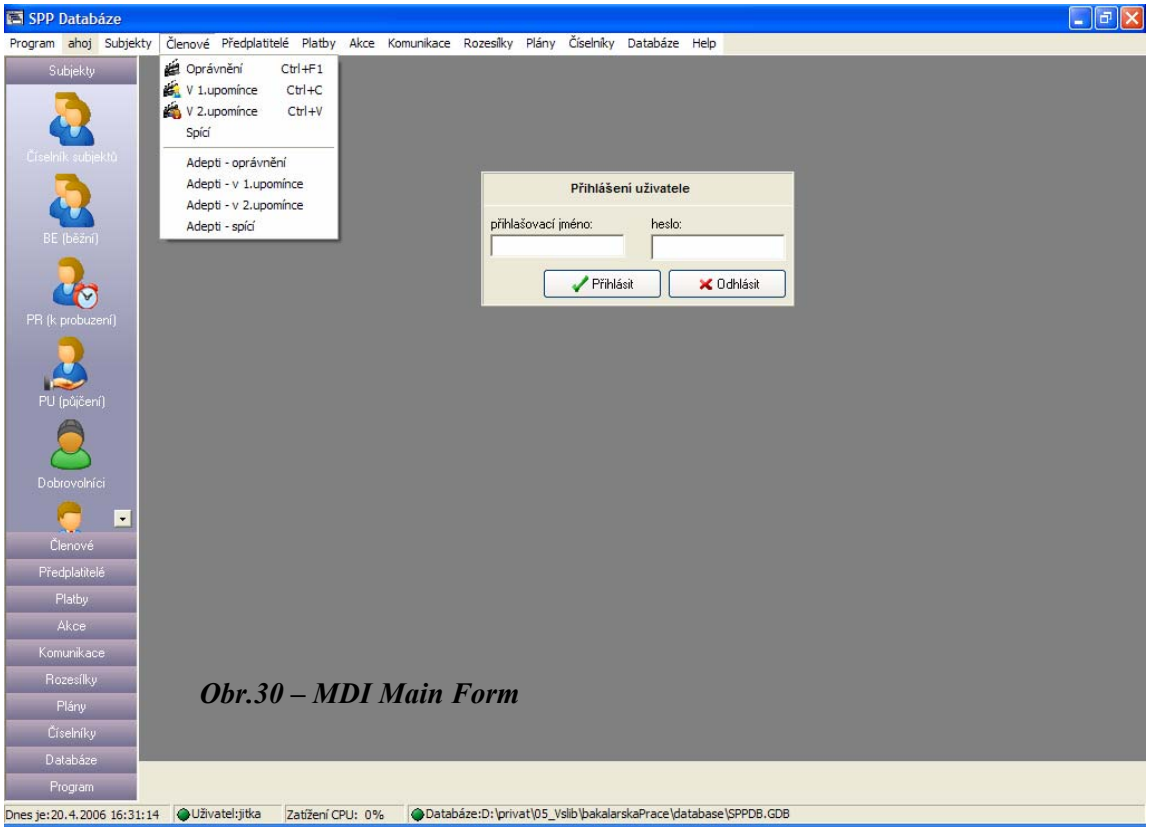
5.1.4 Ostatní komponenty

Na kód s popiskem ostatních komponent odkážu čtenáře na přiložené CD. Jedná se zejména o komponenty JDForm a JDMaskEdit. Celý balík komponent lze nalézt pod názvem : jdDBD6p.dpk. Při instalaci do Delphi se tyto komponenty přidají jako samostatná záložka s pojmenováním „JD – databázové komponenty“.

5.2 SDI form – hlavní část pro uživatele

SDI - Single Document Interface, neboli rozhraní pro jeden dokument. Taková aplikace umožňuje otevření (dockování) jednoho okna v daném čase.

Hlavní okno aplikace vypadá takto:



Obr.30 – MDI Main Form

V levé liště jsou zobrazeny jednotlivé moduly a jsou uživateli přístupné (enabled) pouze pokud má na ně práva.

Horní menu má všechny hlavní funkce s programem zahrnuty v sobě a je zahrnuta funkčnost přidání (merge) menu dle aktuálního okna – tedy menu specifické pouze pro dané okno.

Spodní lišta statusbar je aktivní a po kliknutí na databázi nebo uživatele se zobrazí dialog přihlašovacích údajů k databázi. Navíc v threadu informuje uživatele o aktuálním využití CPU. Jedná se o informativní údaj, při kterém uživatel vidí, že při načítání velkého množství záznamů z DB program nepřestal reagovat. Vlevo je zobrazen systémový datum a čas.

Hlavní šedý prostor je určen pro dockování formů. Vyloučila jsem aplikaci, kdy přes sebe „visí“ dvacet oken, které zabírají mnoha místa v paměti. Údaje, které bude potřebovat uživatel vidět najednou musí vidět ihned, a to v nadockovaném formu.

5.3 Základní struktura programu v Delphi – shared knihovny, dockovací formy, modální okna, dialogy

5.3.1 Unity

V mém projektu je zahrnuta jedna knihovna s názvem uShared, která v sobě obsahuje všechny funkce a procedury společné v celém programu. Jedná se zejména o

- funkce SQL: `GetSQLString, GetSQLInteger, RefreshDataset`
- funkce pro práci s XML: GetAppXMLValue, SetAppXMLValue
- šifrování dat: GetEncrypted, GetDecrypted
- zjištění práva uživatele na určitou akci: GetHasAccess
- získávání dat z čísleníků: GetSubjekt(...),GetAkce(),GetPSCObec()...
- validace dat: IsValidRc, IsValidMail
- odesílání emailu: SendMailTo
- otevírání www prohlížeče: OpenWWW(www: string)
- export do MS Excel
- a další...

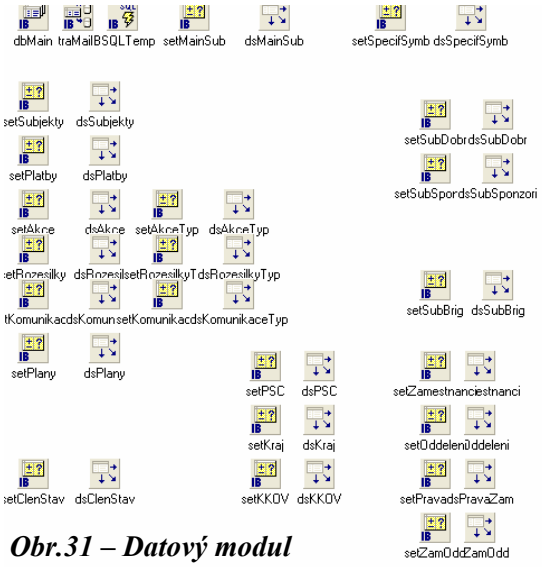
Unita uCPUUsage; vracející hodnotu zatížení procesoru byla převzata z internetu.

Součástí je RES file XP2.res, jelikož aplikace psané v Delphi6 mají po kompilaci vzhled oken z roku 1995. Tedy pro zobrazení hezkých oken a komponent vkládáme tento manifest.

Unita uConstants obsahuje konstanty společné celému programu.

5.3.2 Datový modul dmSPPdb

Datový modul dmSPPdb je hlavní úložiště společných datasetů a datasourců. Zde se deklarují SQL příkazy a generují sloupce(fields) a jejich uživatelské popisky. Jedná se po hlavním formuláři o nejkomplexnější unitu programu. Zde se také počítají zmiňovaná CalcFields na příslušné akce datasetů.



Obr.31 – Datový modul

5.3.3 Dockované formy

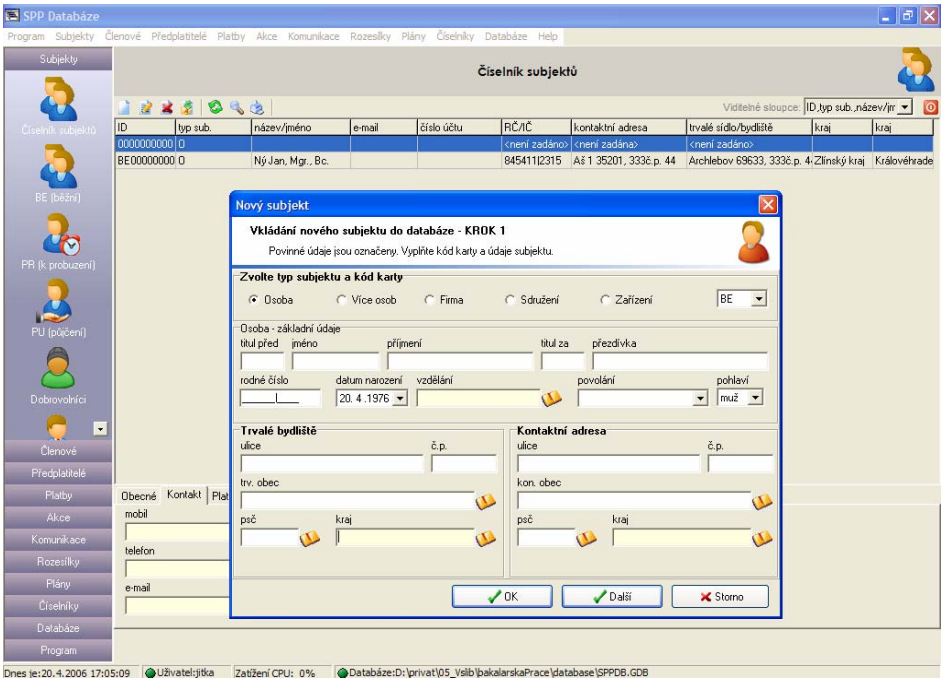
Zvolila jsem pravidlo, že veškeré dockované formy jsou pojmenovány s prefixem „fd“. Na každou akci otevření daného formuláře se po jeho vytvoření v paměti volá procedura p_openDockForm(frdPlany), která zajistí dockování formu v pracovním prostoru Main formu. V proceduře je hlavní tento kód:

```
for i := 0 to pnlWorkArea.DockClientCount - 1 do
  (pnlWorkArea.DockClients[i] as TJDForm).Close;

dockForm.ManualDock(frmMain.pnlWorkArea, nil, alNone);
dockForm.Show;
```

Nejedná se o celý výpis procedury. Dockovací TJDForm musí být pro dockování připraven nastavením: align= alClient a UseDockManager na true.

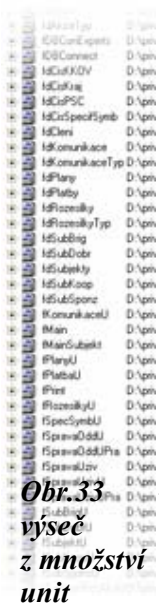
Po dockování vypadá program takto:



Obr.32 – Mainform s dockovaným TJDFormem a modálním oknem

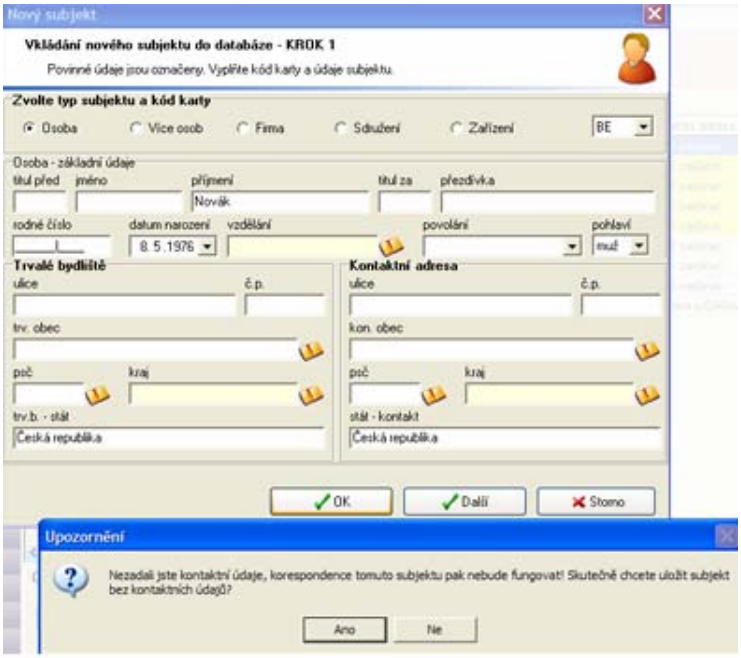
Z tohoto hlavního formu, který většinou obsahuje TJDDBGrid s náhledem dat, která jsou uložena v datasetu je možné otevírat další, vždy modální TJDFormy.

5.3.4 Ostatní modální okna nebo dialogy



Obr.33 výseč z množství unit

Modální akce jsou vyvolány určitou akcí uživatele. Tyto akce jsou definovány v ActionManageru každého „nadřazeného“ JDFormu a přiřazeny některým komponentám – ToolButtony v lištách, TMenuItem a v menu. Menu pro jednotlivé nadřazené formy je „mergováno“ (vsunováno) do hlavního menu, a tak není přepsáno. Spustí-li uživatel tuto akci, vyvolá se velice často instrukce „ShowModal“. Tato instrukce má návratovou hodnotu – s čím uživatel formulář opustil. Zmáčne-li tlačítko OK,navrátí se instrukce s TModalResult=mrOk, jinak např mrCancel a jiné, které si v aplikaci dle potřeby nastavuji. TJDForm má tu výhodu, že na tlačítko ESCAPE se vždy vrátí z instrukce ShowModal – s ModalResult=mrCancel. Před voláním této akce je často zapotřebí definovat parametry, se kterými je modální okno otevíráno. Jelikož používám otevírání modálního okna pro vkládání záznamu i pro jeho editaci (tedy slouží více akcím) je zpravidla nutné nastavit DataSet do některé z těchto akcí a edity formuláře zinicializovat dle dané situace. Neexistuje např.komponenta TDBDateTime, která by uživateli dovolovala výběr data z kalendáře a zároveň byla propojena s datasetem. Několik takových komponent jsem však našla v projektu open-source komunity JEDI a jejich komponenty jsem použila. Nebylo to však tak jednoznačné, jak se mohlo zprvu zdát, jelikož mnoho komponent, které jsem vyzkoušela bylo chybových. Odladila jsem tedy pro svou aplikaci jen několik málo z nich. V případech, kdy nešlo použít standardní databázové komponenty nebo funkční komponenty JEDI bylo zapotřebí použít komponenty nedatabázové a ty vždy správně plnit dle uživatelovy akce (vkládání, editace).



Obr. 34 - Modální okno pro „INSERT“ s dialogovým dotazem

Při zavírání formuláře s ModalResult mrOK bylo zapotřebí provést správnou validaci dat, aby uživatele neděsily chybové hlášky přímo ze serveru, které pro něj nejsou zrovna „user-friendly“. Ovšem i v případě neošetřené výjimky by měla být databáze navržena tak, že se žádnému klientu nepodaří do ní vložit nesprávná data.

Často jsou modální okna použita pro výběr záznamu z číselníku.

Z dialogových oken jsou v aplikaci použity nejčastěji ta, která slouží k potvrzení uživatelovy kritické akce (mazání, obnova ze zálohy apod.) nebo výběru souboru určitého typu (např. soubor databáze GDB).

5.3.4.1 Subjekty a Drag&Drop

Vždy jsem přemýšlela, kde by bylo vhodné a možné zahrnout funkci drag&drop. Vyplynulo prozatím jediné místo, kde je tato funkcionalita podpořena, a to na modálním okně Nový subjekt/editace subjektu při možnosti převzít adresu trvalé bydliště do adresy kontaktního a naopak.

Obr. 35 - Drag&Drop na modálním formuláři Subjekt

Drag&drop je funkční při nastavení DragMode na dmAutomatic a zároveň definici dvou (a jelikož máme možnost i na druhou stranu tak cca čtyř) procedur.

```
procedure TfrmSubjektU.boxPrechBydlisteDragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  with dmSPP.setSubjekty do
    begin
      FieldByName('K_ULICE').AsString := edtTBulice.Text;
      FieldByName('K_CP').AsString := edtTBcp.Text;
      ...
    end;
end;

procedure TfrmSubjektU.boxPrechBydlisteDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source = boxTrvBydliste then    Accept := True else Accept := False;
end;
```

5.4 Přihlašování uživatelů, správa uživatelů

Při instalaci programu je v DB v tabulce (již nemluvím o entitách) <SPP$ZAMESTNANCI> první uživatel s loginem „admin“. Jeho implicitní heslo je „sppdb“. Toto heslo by mělo být okamžitě změněno, a to poklikáním na statusbar v sekci „uživatel:admin“. Zde se objeví dialogové okno, které požaduje zadání starého i nového hesla. Heslo pak uloží opět v kryptované podobě do databáze.

Uživatel admin nejde smazat. Ošetřeno je to jak v programu, který tak ukazuje přívětivě „user-friendly“ hlášku, ale také na úrovni databáze triggerem s příslušnou výjimkou, aby nemohlo dojít k smazání kvůli nějaké neošetřené chybě. Pak by již nebylo možné z programu přidávat nové uživatele a přiřazovat jim práva.

```
CREATE EXCEPTION EXCEPT_DEL_ADMIN 'Nelze smazat administrátora!!!';
```

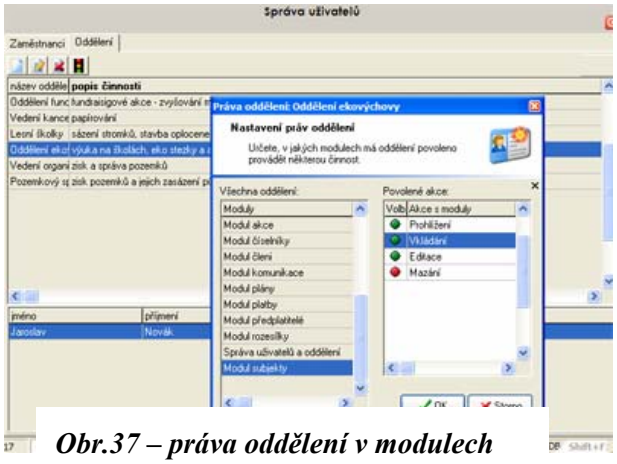
```
CREATE TRIGGER SPP$ZAMESTNANCI_BD0 FOR SPP$ZAMESTNANCI
ACTIVE BEFORE DELETE POSITION 0
AS
begin
  if (old.login='admin') then exception except_del_admin;
end
```

Na počátku tedy nemá nikdo práva na nic, neexistují jiní uživatelé nežli uživatel „admin“. Tento poprvé rozdělí práva a vytvoří jim na základě jejich žádostí konta. Kopírujeme popis funkčnosti na DFD diagramu v kapitole 5.5.1.2.

Administrátor programu může přiřadit libovolnému uživateli právo na správu uživatelů. Tento pak má všechny výhody administrátora kromě jediné – jde smazat. Výhodou i nebezpečím správce uživatelů je to, že správce uživatelů může měnit libovolně uživatelům hesla. Je to zapotřebí, pokud některý zaměstnanec své heslo zapomene a je třeba vygenerovat nové.

Obr.36– práva uživatele = patří do oddělení

Přiřazení práv se děje na formuláři s názvem: „Správa uživatelů“, který je přístupný pouze tehdy, má-li přihlášený uživatel na správu práva. Zde uživateli přiřazujeme, do jakého patří oddělení. Každé oddělení má poté nějaká práva.



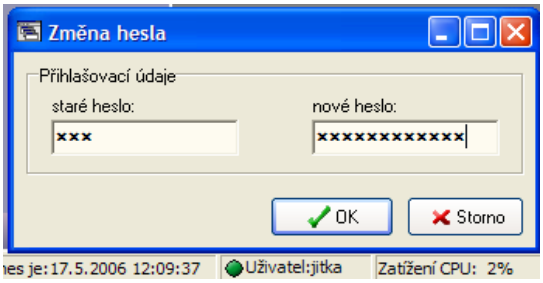
Obr.37 – práva oddělení v modulech

Editace oddělení je umístěna hned na vedlejší záložce, aby správce uživatelů nemusel moc klikat. Zde vytváří, maže a edituje oddělení, kterým pak pomocí „semaforu“ přiděluje práva. V gridu se mu zobrazuje buď červená „dioda“ či zelená „dioda“ pro indikaci, zda oddělení má práva na zvolenou akci.

Oddělení na obrázku nemůže mazat subjekty v modulu subjekty. Ostatní akce jsou mu povoleny.

5.4.1 Změna hesla uživatele

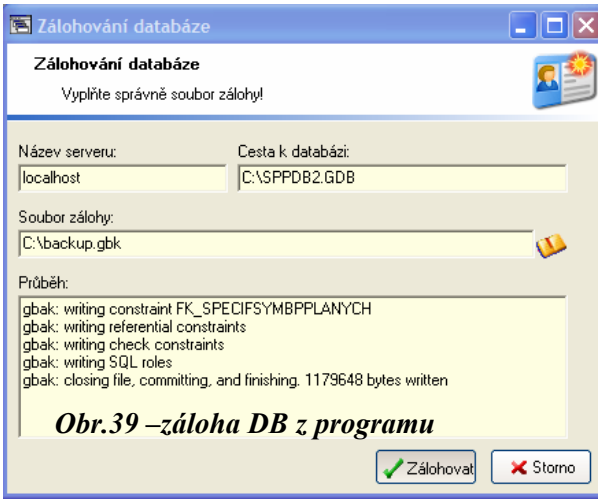
Každý uživatel si může libovolně měnit heslo poklepáním na StatusBar a vyvoláním dialogového okna. Musí správně zadat staré a nové heslo, které je před uložením šifrováno.



Obr.38 – změna hesla uživatele

5.5 Záloha a obnova databáze

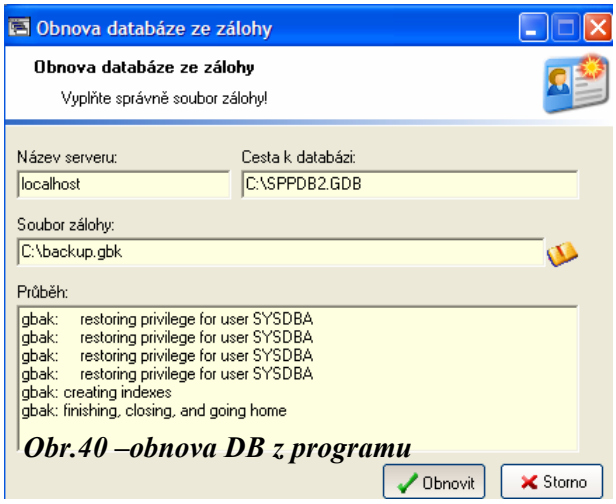
Databázový systém Firebird přišel s možností zálohy databáze při plném provozu serveru. Při obnově DB je nutné, aby byl k DB připojen max. 1 uživatel.



Obr.39 –záloha DB z programu

Pro zálohu a obnovu databáze FireBird byla vytvořena command line utilita GBAK. Tak je možné spustit zálohu DB z příkazového řádku. To pro uživatele znamená znalost spouštěcích příkazů apod. Proto byly použity instalované komponenty Delphi IBBackupService a IBRestoreService, které využívají interně tuto utilitu k vytvoření GUI možnosti zálohy a obnovy databáze.

Je zřejmý dotaz typu: proč je nutné zálohovat databázi s použitím gbak, a ne pouze soubor .gdb či .fbd zkopírovat na místo zálohy? Odpověď je zde: GBAK neprovádí pouze čistou kopii databáze. Tato utilita také databáze optimalizuje tím, že provádí Garbage Collection na vypršelé záznamy a vyvažuje indexy. Také obnovuje místo, které zabraly smazané záznamy a redukuje velikost celé databáze. Data, která v ní zůstala, „stěsní“. Nabízí možnost změny velikosti stránky a rozdělení databáze na několik souborů.



Obr.40 –obnova DB z programu

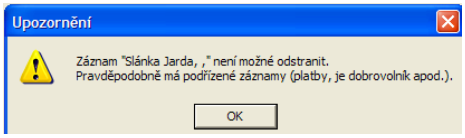
V programu probíhá záloha takto: Uživatel vybere místo zálohy databáze na svém lokálním počítači a zálohu pak spustí. Hláška indikuje, že záloha a obnova proběhla úspěšně či zobrazí chybu s jejím popisem.

5.6 Vkládání, editace, mazání záznamů

Pro vkládání a editaci záznamů je vždy vytvořen speciální formulář. Popíšu tyto akce na vkládání/editaci záznamu v entitě <Platby>. V hlavním okně číselníku plateb jsou v toolbaru či v MenuBaru přístupné akce INSERT – UPDATE – DELETE. Akce delete nevolá žádný formulář, pouze právě aktivní řádek vymaže z DB. Poté udělá Commit, je-li tato akce úspěšná. Není-li, provede Rollback a zobrazí např. hlášku ze serveru, že nelze smazat záznam s podřízenými záznamy.

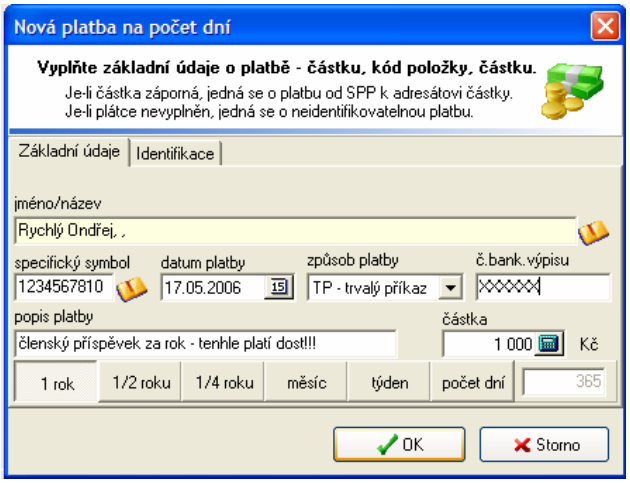
Obr.41 – tlačítka pro insert, update, delete, refresh a hledání, export do MS Excel

Po zmáčknutí na insert nebo update se vyvolá ShowModal příslušného TJDFormu.



Obr.42 – záznam má podřízené záznamy

Zde uživatel vyplní všechny důležité údaje. Ty, které nesmí být NULL, nebo mají jiná omezení se na událost OnCloseQuery formuláře kontrolují a také se nastavují popř. nějaké doplňující údaje (vložil, datum a čas vložení apod.). OnCloseQuery nastává vždy, když se zavírá formulář a tím, že mu přidáme CanClose = false se nakonec nezavře. To využívám při zjištění chybně zadaných dat. Pozor, abychom kontrolu prováděli pouze tehdy, je-li ModalResult = mrOK.



Obr.43 –formulář pro insert a update

Takže máme validní data, můžeme provést uložení. To provedeme tím, že datasetu pošleme příkaz Post. Při spuštění akce insert byl dataset uveden do stavu Insert, při akci update byl uveden do stavu Edit. Akcí Post odesíláme data s tím, že je ještě nutné příkazem Commit potvrdit transakci. Toto je zjednodušený popis toho, jaký je postup u ukládání záznamů. Každý jednotlivý formulář má svá specifika.

5.6.1 Konkrétněji popsané vkládání plateb a následné kontroly v DB – triggery, uložené procedury

Po uložení záznamu do tabulky <PLATBY> se spouští trigger AUID_platby.

```
CREATE TRIGGER AIUD_PLATBY FOR PLATBY
ACTIVE AFTER INSERT OR UPDATE OR DELETE POSITION 0
AS
    declare variable SpecSymbFiltrClensky varchar(10);
    declare variable SpecSymbFiltrPredpl varchar(10);
begin
/* INIT */
 select hodnota_s from var$program where klic = 'SpecSymbFiltrClensky'
 into :SpecSymbFiltrClensky;
select hodnota_s from var$program where klic = 'SpecSymbFiltrPredpl'
 into :SpecSymbFiltrPredpl;
 if ((SpecSymbFiltrClensky = '') or (SpecSymbFiltrClensky is null)) then exception
except_nenikonstclenstvi  ;
 if ((SpecSymbFiltrPredpl = '') or (SpecSymbFiltrPredpl is null) ) then exception
except_nenikonstpredpl  ;
/* INSERT */
 if ((new.id_sub is not null) and (inserting)) then
 begin
  execute procedure sp_fixkodkarty(new.id_sub);
   -- platba za členství od plátce pro SPP--
   if ((new.specifsymb like SpecSymbFiltrClensky) and (new.castka > 0)) then
    execute procedure SP_FIXCLENSTVI(new.id_sub);
    --předplatné od plátce pro SPP--
   if ((new.specifsymb like SpecSymbFiltrPredpl) and (new.castka > 0)) then
    execute procedure SP_FIXPREDPLATNE(new.id_sub);
 end
```

```
/* UPDATE - podobně */
...
/* DELETE - podobně */
end
^
```

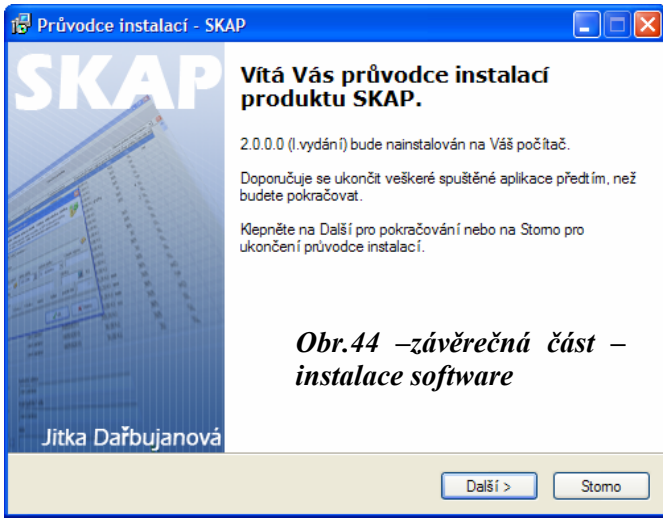
Tento trigger zjišťuje, zda vkládaná platba odpovídá platbě za členství či předplatné a volá příslušné uložené procedury <SP_FIXCLENSTVI> a <SP_FIXPREDPLATNE>, které mění záznam s <ID_SUB>, který je její vstupní proměnnou v tabulkách <CLENI> a <PREDPLATNE> dle zadaných pravidel. V každém případě však opraví kód karty subjektu voláním <SP_FIXKODKARTY>.

Výše jmenované uložené procedury nejsou krátké, jsou dobře popsané v elektronické podobě na CD.

6. Instalace software

Za pomocí freewarové utility InnoSetup jsem vytvořila instalační balík, který uživatele vede při instalaci vytvořeného programu a zkopíruje příslušné soubory do určeného místa.

Možnost zápisu do registrů a jiné možnosti nevyužívám ani zde, ani v aplikaci. Tak poběží software i uživatelům, kteří mají na svých Windows omezená práva.



Obr.44 –závěrečná část – instalace software

Po instalaci se program zařadí do cesty : `START > PROGRAMY > Jitka Dařbujanová SW > SKAP` a je také vidět v `Ovládací panely > Přidat/Odebrat programy`.

Zároveň je vygenerován také UNIINSTALL program.

Instalace serveru bude provedena ze standardních instalačních souborů a použijeme verzi Super Server, která je určena pro OS Windows XP.

7. Závěr

Celková práce mi zabrala skutečně čas od doby zadání až těsně do doby před odevzdáním. Návrh databáze byl dlouhodobější záležitost, programování a hlavně opravování chyb také. Mnoho věcí jsem bohužel ani nestačila v textu vyzdvihnout, jelikož se přímo netýkají zadání bakalářské práce, ale pro SPP je tato funkcionalita velice důležitá. Zajímavé by určitě bylo vysvětlit jakým způsobem jsem řešila export do MS Excel, jaký je konkrétní postup zadávání rozesílek a veškerá zjednodušení, která jsem zde zaměstnancům SPP naprogramovala. Mohla jsem konkrétněji popsat instalační skript na SW, který generuje soubor setup.exe. To však není předmětem zadání.

Práce na návrhu databázového informačního systému byla pro mě velice přínosná, jelikož přesně touto cestou bych se chtěla ubírat v budoucím životě. Zvláště detailní seznámení s prostředkem na vývoj databázového systému Case Studio byl pro mě jasným obohacením. Mnoho mi dalo si myšlenky napsat, jak který postup má fungovat a při čtení tohoto textu jsem vždy objevila nové relace nebo možná zjednodušení.

Programování v Delphi byla spíše rutinní a zdlouhavá práce, ovšem některé části byly také zajímavé. Např. manipulace s XML dokumenty apod.

Celkově si myslím, že kdyby si takto malá společnost měla nechat dělat takový software od soukromé firmy, nebyla by schopna náklady na jeho vývoj unést. Tipuji takový software na několik desítek tisíc korun. Měli by k tomu však samozřejmě veškerou systémovou podporu.

Na druhou stranu v dnešní počítačové době je potřeba manipulaci s ohromnými množstvími dat nějak automatizovat. Nelze již vše zařizovat ručně, když už i bankovní výpisy docházejí v el. podobě. Počty záznamů v databázi se předpokládají zatím do cca 2500 v entitě <Subjekty> a s ohledem na to bude počet v entitě <platby > např. 10 000. Doufám tedy, že moje několikaměsíční práce nezmizí po odevzdání ze světa a bude dlouhou dobu sloužit dobré věci.

Na závěr už jen vyslovím přání, ať Společnosti přátel přírody software dobře slouží a ať nám jejich bohulibá činnost přináší více nových původních dřevin do našeho okolí – Jizerských hor.