Univerzita Karlova v Praze
Filozofická fakulta
Katedra Logiky

Diplomová práce

Robert Polák

Posouzení výhod a nevýhod OOP databází vzhledem k relačním databázím

Comparison of advantages and disadvantages of OOP databases against relational databases

Vedoucí práce: Doc. PhDr. Petr Jirků, CSc.
Konzultant práce: Ing. Pavel Stěhule

2009

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a že jsem uvedl všechny použité prameny a literaturu.

V Praze 28.4.2009

Robert Polák

Poděkování

Na tomto místě bych rád poděkoval Doc. Jirků za trpělivost s vedením mé práce. Zvláštní poděkování patří Ing Stěhulemu za velmi četné konzultace nejen k systému Garuda. Díky patří i společnosti ILIETHIS ! s.r.o. za poskytnutí zdrojových kódů a dokumentace k systému Garuda pro účely této práce. V neposlední řadě patří mé díky ženě a dětem za to, že mi poskytly klid, zázemí a motivaci pro psaní práce kdykoliv jsem to potřeboval.

Abstrakt

Existují různé přístupy k modelování bází dat. V této práci se snažím identifikovat silná a slabá místa aktuálně používaných databázových technologii. Ať už to jsou klasické relační databáze, tak databáze, které používají objektově orientované principy, či přístupy jak zkombinovat svět objektů se světem tabulek relačních databází. Poslední část je věnována návrhu metodiky pro vývoj informačních systémů pomocí technologie, kombinující objektový a relační přístup tak, že realizuje objektový přístup k datům prostředky relační databáze.

Abstract

There exists different approaches on database modeling. This work tries to identify strengths and weaknesses of nowadays technologies. Besides relational databases, object oriented databases, and methods to combine the world of objects with the world of tables of relational databases. Last part gives attention to design of methodology of process of development of information systems using technology that implements object access to the data throw build on relational database facilities .

Obsah

1 Úvod **7**

1.1 Cíle....7

1.2 Zvolená metodika....7

2 Životní cyklus dokumentu – workflow **9**

2.1 Zadání pro DMS....9

3 Relační přístup **11**

3.1 Základy relačního modelu databáze....11

3.2 Návrh relačního schématu databáze....11

3.3 Dotazovací jazyk SQL....14

3.4 Objektově relační přístup....14

3.5 Možnosti přístupu k datům uloženým v RSŘBD....14

3.6 SWOT analýza relačních databází....16

4 Principy OOP **20**

4.1 4.1 Historie OOP....21

5 Objektové databáze **22**

5.1 Rozvoj OOSŘBD....22

5.2 Datový model....23

5.3 Dotazování do databáze....25

5.4 Návrh objektového modelu....25

5.5 SWOT analýza OOSŘBD....26

6 Objektově relační mapování (ORM) **29**

6.1 Obecný popis ORM....29

6.2 Úkoly mapování....29

6.3 Metadata a jejich získávání....30

6.4 Způsoby chování a užívání ORM....30

6.5 SWOT analýza ORM....31

7 Garuda **34**

7.1 Původ....34

7.2 Filozofie GA....34

7.3 Architektura....34

7.4 Garuda definition language (GDL)....35

7.5 Garuda API....38

7.6 Objektové vlastnosti Garudy....40

7.7 Vzorový příklad....41

7.8 SWOT analýza Garudy....45

8 Návrh metodiky vývoje pro Garudu **48**

8.1 Důvody pro užívání metodik....48

8.2 Tvorba metodiky pro GA....49

8.3 Shrnutí popsané metodiky vývoje pro GA....57

9 Závěr **59**

9.1 Posouzení výsledků SWOT analýz....59

10 Prameny **61**

10.1 Seznam použité literatury a online zdrojů....61

10.2 Přílohy 63

11 Abecední rejstřík **64**

12 Seznam použitých zkratek **65**

1 Úvod

Spolu s rozšířením objektově orientované architektury informačních systémů v 80. a 90. letech dvacátého století vznikla celá řada metodik, které kombinují objektový model aplikace s ukládáním aplikačních dat v databázových systémech.

Rozdílné, někdy i protichůdné požadavky na objektový a relační datový model vedly v devadesátých letech k renesanci objektově orientovaných systémů řízení báze dat (OOSŘBD). Nejrozšířenějším řešením však jsou databáze relační, proto vznikla celá řada platforem, které zajišťují ukládání aplikačních objektů do relačních databází, tzv. objektově relační mapování. Svět relačních databází se postupně přizpůsoboval požadavkům souvisejícím s ukládáním objektů postupně začleňoval objektové prvky do standardů pro relační databáze, implementace podporující tato rozšíření bývají nazývány objektově relační databáze.

Základní myšlenkou objektového přístupu je zapouzdření dat a operací, které je s nimi možné provádět, do jednoho celku, který je nazýván objektem. Tento přístup lze do jisté míry praktikovat přímo v databázové vrstvě za pomocí jazyka vnořených procedur. Stejnou cestou se vydali vývojáři projektu Garuda (GA).

1.1 Cíle

Cílem této práce je popsat základní vlastnosti relačních databází, objektových databází, objektově relačního mapování a systému GA, který kombinuje objektový i relační přístup na databázové vrstvě.

U každého přístupu pak zhodnotit jeho klady, zápory, rizika a potenciál řešení metodou SWOT analýzy.

Pro systém Garuda pak chceme na základě výsledků analýzy navrhnout metodiku vývoje informačních systémů, která využije její klady a pokusí se minimalizovat v analýze zjištěné zápory.

1.2 Zvolená metodika

Pro posuzování výhod a nevýhod byla již v zadání práce zvolena metodika SWOT analýz jednotlivých problematik.

SWOT analýza [1][2] je analytická metoda, která je založena na principu posouzení kladů a záporů, podrobně: výhod, nevýhod, příležitostí a hrozeb zkoumané problematiky. Zkratkovitý název „SWOT“ vznikl jako akronym z počátečních písmen anglických slov **s**trengths (výhody, silné stránky), **w**eaknesses (slabiny, nedostatky), **o**pportunities (příležitosti) a **t**hreats (hrozby, rizika, nebezpečí). Výhody a nevýhody tvoří vnitřní popis jevu. Příležitosti a hrozby se pak zabývají vnějšími jevy, které nelze jednoduše z pohledu zkoumaného jevu ovlivnit. Viz tabulka 1.

Původně byla metodika SWOT navržena Albertem Humphrey, v 60. a 70. letech dvacátého století, pro užití v oblasti podnikového plánování, dnes je využívána v celé řadě odvětví. V marketingu, při řízení lidských zdrojů, projektovém řízení a v našem případě v oblasti informačních technologií.

Dělení dle významu zmíněných čtyř kritérií znázorňuje následující tabulka.

	Klady	**Zápory**
Vnitřní popis (stávající stav)	Výhody	Nevýhody/slabiny
Vnější vlivy (mohou nastat)	Příležitosti	Hrozby,rizika

Tabulka 1: struktura SWOT analýzy

Metodika SWOT není určena k exaktnímu srovnávání měřitelných jevů. Jejím hlavním účelem je vytvoření podkladů pro strategické rozhodování. Metoda umožňuje shromáždit široké spektrum argumentů roztříděných na výhody, nevýhody, příležitosti a hrozby. V okamžiku rozhodnutí je pak třeba argumenty aplikovat, tedy zvážit jejich relevanci, pro zkoumanou problematiku.

Při rozhodování hledáme řešení, které je nejméně nevýhodné resp. nejvíce výhodné. V případě vnějších vlivů pak takové, které do budoucna nabízí nejvíce příležitostí, resp. které je nejméně rizikové. Metodika SWOT nabízí následující možnosti pro tvorbu strategií na základě výsledků analýzy:

- max-max – maximalizace výhod – maximalizace příležitostí
- min-max – minimalizace slabin – maximalizace příležitostí
- max-min – maximalizace výhod – minimalizace hrozeb
- min-min – minimalizace slabin – minimalizace hrozeb

Příkladem toho, že se metodika SWOT analýzy používá v oblasti IT, může být analýza Koncepce informačního systému výzkumu a vývoje [3].

2 Životní cyklus dokumentu – workflow

Součástí zadání této práce je: „posoudit výhody a nevýhody objektových databází při návrhu a realizaci informačních systémů. Zvláště pak systémů pro řízení životního cyklu dokumentu v podniku.“ Tuto kapitolu věnujeme rozboru životního cyklu dokumentu a jeho zachycení v informačních systémech. Seznámíme se s termínem workflow a zformulujeme vzorový příklad pro workflow dokumentu, pro který pak postupně navrhneme relační model, objektový model, ukážeme si na něm objektově relační mapování a nakonec přístup vlastní frameworku Garuda.

Příkladem dokumentu v podniku mohou být objednávka, smlouva, faktura, analýza, korespondence, předávací protokol, příjemka na sklad, výdejka ze skladu, reklamační protokol, dodací list, technický výkres, ale i fotografie, mapa a tak podobně.

Každý ze jmenovaných dokumentů obsahuje údaje, které tvoří obsah dokumentu, v některých případech je pak lze i dále strukturovat. Ku příkladu objednávku můžeme dále strukturovat na zboží, objednavatele, jeho adresu, adresu pro doručení, způsob doručení zboží a tak dále. Kromě údajů, které dokument obsahuje, evidujeme také údaje, které se k dokumentu vztahují, ale nemusí být jeho součástí, tzv. metadata – data o dokumentu. Příkladem metadat může být informace o umístění dokumentu, ať už v elektronické, nebo fyzické podobě, datum založení záznamu v systému, uživatelé, kteří k dokumentu přistupovali, typ dokumentu a tak dále. Obsah dokumentu spolu s metadaty tvoří dohromady aktuální stav dokumentu.

Pro zachycení životního cyklu dokumentu, potřebujeme v každém možném stavu popsat podmínky pro přechod do dalších stavů. Tedy popis procesů s dokumentem ve firmě. Zajímá nás, které osoby mají práva k němu přistupovat, co s ním mohou provádět, co s ním mají provádět, jakým způsobem a za jakých okolností.

Popis stavů dokumentu a procesů souvisejících s přechody mezi stavy se nazývá workflow dokumentu. Do češtiny se termín většinou nepřekládá, šlo by jej přeložit jako tok práce, běh práce, nebo pracovní proces, případně volněji jako životní cyklus. Z matematického hlediska jde o deterministický konečný automat, jehož konečnou množinu stavů tvoří stavy dokumentu a přechodová funkce je tvořena podmínkami pro přechod mezi jednotlivými stavy.

Obecněji můžeme workflow nějaké entity definovat jako orientovaný graf, ve kterém vrcholy představují stavy entity, tj. soubor všech dat entity a meta dat o entitě, hrany jsou pak ohodnoceny podmínkami a úkoly, které je třeba splnit pro přechod z jednoho stavu do druhého (ve směru orientace příslušné hrany). Podmínky a úkoly, které je třeba splnit, nutně nemusí souviset s daty ani s metadaty entity, může jít například o interakci s workflow nějaké jiné entity.

Pro hlubší studium problematiky workflow můžeme doporučit internetové stránky Asociace „The Workflow Management Coalition“ [4], která si klade za úkol standardizovat jazyk XPDL ("XML Process Definition Language") pro definici workflow.

2.1 Zadání pro DMS

Pro ilustraci jednotlivých přístupů k modelování báze dat nám slouží návrh systému pro správu dokumentů v podniku. Pokusme se přiblížit, co by takový systém měl umět formulací zadání vzorového příkladu, který nás bude provázet v jednotlivých kapitolách práce.

2.1.1 Vzorové zadání

Vyslovme si nyní rámcové zadání informačního systému, na které se budeme odkazovat v dalších kapitolách, kde můžeme zadání dále upřesňovat.

2.1.1.1 Zadání

Navrhněte základ DMS (document management systému) pro řízení životního cyklu dokumentů v podniku tak, aby umožňoval správu různých druhů dokumentů: Objednávka, Faktura a Smlouva, s ohledem na to, že v budoucnu dojde k rozšíření systému o další druhy dokumentů. Pro každý druh dokumentu bude dodán seznam pojmenovaných stavů, které může dokument nabývat spolu s podmínkami pro přechod mezi těmito stavy. Přechody mezi stavy mohou být iniciovány pojmenovanou akcí. Číselníky stavů se mohou do budoucna rozšiřovat. Podmínky pro přechod mezi stavy předpokládají přístup k datům uloženým v dokumentu a oprávnění ze strany uživatele, který přechod vyvolává.

3 Relační přístup

3.1 Základy relačního modelu databáze

Za zakladatele relačního přístupu je dnes považován Ted Codd, který v publikaci z roku 1970 [1] stanovil základní teze relačního modelu.

Nejmenší, z pohledu relačního modelu dále nedělitelnou část dat tvoří atribut. Každý atribut má jméno a je vymezen doménou, tj. množinou hodnot, kterých může nabývat. Název relace spolu s uspořádanou enticí atributů a vymezením jejich domén tvoří relační schéma. Samotná relace nad určitým relačním schématem je pak podmnožina kartézského součinu domén atributů této relace.[5]

Pro jednoznačné určení záznamu slouží klíčové atributy. Klíč, je nejmenší podmnožina atributů, která jednoznačně určuje prvek relace.

V databázové terminologii je pojem „relace“ nahrazen intuitivnějším pojmem „tabulka“. Tabulka se skládá z řádků a sloupců. Řádky tabulky jsou tvořeny prvky relace tj. uspořádanými enticemi. Pojem „schématu relace“ nahrazuje záhlaví tabulky, jméno atributu tvoří jméno sloupce.

Zatímco matematicky vymezený pojem relace neurčuje pořadí prvků, při představě tabulky je pořadí prvků dáno. Relace, jakožto množina, nemůže obsahovat duplicitní entice, databázová tabulka jejich výskyt připouští. Navíc relační model pracuje s doménami atributů, zatímco relační databáze používá datové typy.

V praxi databázové systémy sice vypisují tabulku v nějakém pořadí, ale pokud není explicitně určeno, pak může stejný databázový dotaz vrátit záznamy, které jsou pokaždé jinak seřazené. Obranou proti duplicitním záznamům je pak určení primárního klíče.

3.2 Návrh relačního schématu databáze

Při návrhu schématu relační databáze stojíme před otázkou po jakých částech a do jakých tabulek rozdělit ukládaná data tak, aby práce s uloženými daty byla rychlá, efektivní a hlavně aby při aktualizaci uložených dat nemohlo dojít ke ztrátě jejich konzistence tzv. aktualizačním anomáliím. Tedy pokud situaci, kdy je při potřebě ukládat, resp. mazat, data příslušející nějaké entitě, je také potřeba vkládat, nebo mazat data patřící entitě jiné. Obdobně pokud se změní jedna kopie redundantních dat, je také potřeba změnit kopie jiné. Řešením je normalizace DB schématu využitím funkčních závislostí atributů.[6]

3.2.1 Normalizace

Proces normalizace spočívá v postupném aplikování tzv. pravidel normalizace na postupně vytvořená schémata. Převodem do vyšší normální formy zmenšujeme redundanci schématu a snižujeme výskyt možných anomálií. Pravidla normalizace jsou součástí většiny knih zabývajících se SQL [7] [8][9][6], kde lze dohledat jejich přesné definice.

V souvislosti se srovnáváním OOSŘBD s RSŘBD jen připomeneme definici první normální

formy, která má zajišťovat atomičnost atributů.

3.2.1.1 První normální forma

První normální forma (1NF) má zajistit atomičnost atributů.

Definice:Každý atribut schématu relace je elementárního typu a je nestrukturovaný.

1NF je základní podmínka „plochosti databáze“ - tabulka je opravdu dvourozměrné pole, ne např. skrytý graf[5]

Existuje celá řada definic 1NF [10]. Výše zmíněná formulace skrývá úskalí v pojmech „relace“ a „elementární typ“. Při striktně matematickém výkladu nedovoluje relace ukládání prázdných hodnot NULL a elementární typ nebyl v kontextu definice definován vůbec.

Některé různé definice 1NF vedou ke sporům v jejich interpretaci. Definice každé vyšší normální formy navíc obsahuje jako předpoklad splnění předchozí normální formy, tedy pokud dojde ke zpochybnění 1NF zpochybníme tím platnost všech vyšších normálních forem. Na což upozorňuje skalní zastánce čistě relačního modelu Chris Date [11].Ke kritice dochází v souvislosti s použitím hnízděných datových typů ANSI 1999. V praxi však má užití hnízděných typů svůj význam, zvláště v případě ukládání objektů do relační databáze. Dalším důvodem pro jejich přijmutí do jazyka SQL je jejich přínos při psaní vnořených procedur, tam se však nedostáváme do sporu s 1NF, jelikož se nejedná o perzistentní data. Kritika je však do jisté míry oprávněná, jistě je nerozumné zbytečně využívat hnízděné datové typy v návrhu databáze na místě klíčových atributů.

3.2.2 Denormalizace

S pojmem normalizace úzce souvisí pojem tzv. denormalizace. Jedná se o zpětné spojování již normalizovaných tabulek. Tímto vznikne schéma, které nesplňuje některou z normálních forem. Důvodem k denormalizaci je režie na spojování tabulek při dotazování se do databáze. [12]

3.2.3 Konceptuální modelování

V případě konceptuálního modelování se vytváří nejprve konceptuální datový model, který reprezentuje schématem nebo obrázkem data nezávisle na jejich fyzickém uložení.

Mezi nejznámější patří ER modelování. Slouží jako nástroj pro vytváření databázového schématu pomocí modelování entity a vztahy mezi nimi, pro jednotlivé entity se pak identifikují atributy a klíčové atributy entit. Výhodou ER modelování je přehledná grafická notace.

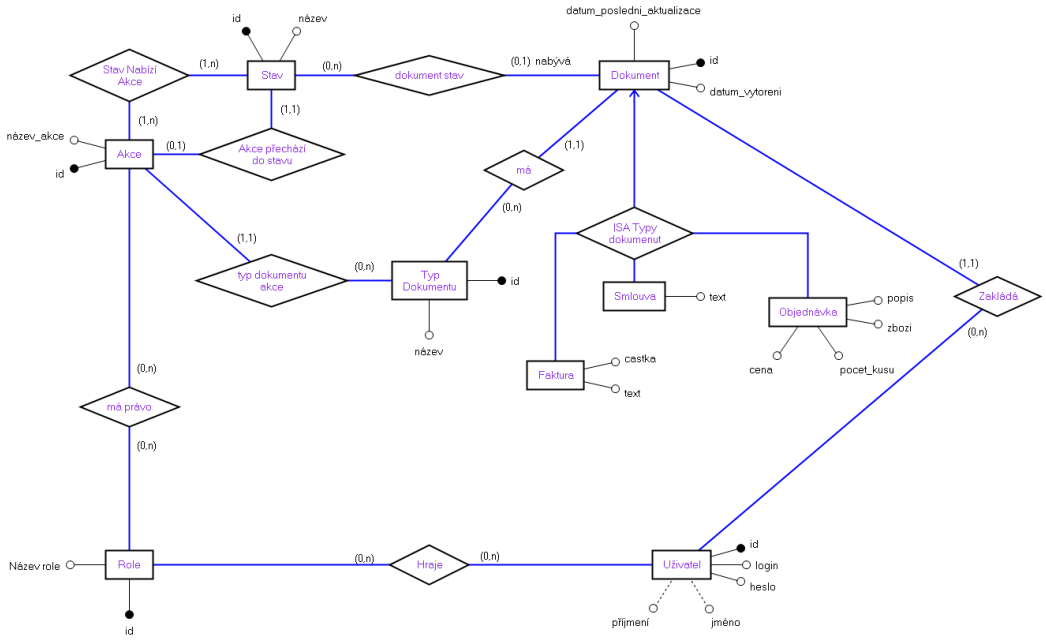
Z ER diagram pak lze algoritmy převést na logický návrh tabulek relační databáze a nakonec fyzický návrh, který doplňuje datové typy, integritní omezení a indexace tabulek.

Pro ER modelování jsou známy algoritmy na převod ER modelu na normalizované schéma relační databáze[6], stejně tak existují Case nástroje pro vytváření ER diagramů, umožňují vygenerovat na základě diagramu normalizované schéma databáze.

Relační model lze taktéž modelovat prostřednictvím UML Class diagramu. Ten sice v rámci definice UML nepostihuje všechny údaje nutné pro korektní vygenerování databázového schématu, například definici primárního klíče, nicméně Case nástroje využívají rozšíření, které definici primárních klíčů postihuje. Příkladem může být Enterprice Architect od společnosti Sparx Systems [13].

3.2.3.1 Modelování vzorového příkladu

Následující diagram byl modelován pomocí programu ERTOS [14] a zobrazuje ER diagram pro Vzorové zadáni DMS viz 2.1.1.1. Vygenerované schéma je součástí přílohy [Příloha7], spolu s dalšími možnými diagramy pro vzorové zadání. ER model níže vědomě popírá některé zásady normalizace, šlo nám o postihnutí možného modelování workflow v relační databázi a nezacházeli jsme do detailů , dokument a jeho potomci jistě mají více atributů a některé atributy by mohli být dále strukturované (např. zboží). Námi předvedené schéma umožňuje ukládat možné přechody mezi stavy na základě aktuálního stavu, typu dokumentu, a vybranému uživateli.



Ilustrace 1: ER model k příkladu DMS

Veškerá logika je prováděna na aplikační vrstvě. Pomocí následujícího návrhu, bychom mohli naprogramovat systém, u kterého by bylo možno dynamicky administrovat, typy dokumentu a číselníky stavu a prováděných akcí.

Typy dokumentu však vždy, ať už se budou jmenovat jakkoliv, budou z hlediska datového modelu, Dokumenty, Objednávky nebo Faktury.

Stav objektu z hlediska workflow jsme si definovali jako soubor všech informací o něm. V případě našeho návrhu nabízíme stav pouze jako pojmenovanou číselnou hodnotu, tedy určitou význačnou událost v životě objektu. Framework GA, o kterém budeme mluvit později, používá možná vhodnější termín **krok** workflow. Relační návrh výše umožňuje nabízet přechod do dalšího stavu pouze na základě hodnoty aktuálního stavu. Veškeré další omezení, nebo rozšíření logiky přechodu mezi stavy je tedy nutné doprogramovat na aplikační úrovni. Právě na frameworku GA si ukážeme, že za pomoci ORSŘBD lze implementovat aplikační logiku i na databázové vrstvě.

3.3 Dotazovací jazyk SQL

Vznik jazyka SQL (Structured Query Language) spadá do 70. let minulého století a zpočátku souvisel s vývojem systému R společnosti IBM. Po prvních verzích, které se ještě jmenovaly SEQUEL a SEQUEL2, byl v roce 1986 jako jazyk SQL prohlášen standardem ANSI (SQL 86) a o rok později standardem ISO (SQL 87).[9]

V dalších letech, až do roku 1999, bylo SQL postupně rozšiřováno o regulární výrazy, rekurzivní dotazy, triggery, neskalární typy a některé objektové vlastnosti. Poslední verze z té doby SQL1999, je de facto závěrem vývoje SQL pro čistě relační SŘBD. V literatuře se někdy zmiňuje SQL3, jednalo se o koncepci, z které vychází standard SQL 1999 a tvoří podmnožinu celkové koncepce.

3.4 Objektově relační přístup

Motivací pro objektově relační přístup bylo využít co možná nejvíce výhod a zkušeností ze stávajících relačních databází s tím, že se rozšíří tak, aby vyšly vstříc požadavkům ze světa objektově orientovaného programování. V literatuře bývá objektově relační přístup zmiňován ve spojení s SQL3, ANSI 1999, resp. 2000 viz podkapitola Dotazovací jazyk SQL.

SQL-1999. Novinky, které přináší standart SQL 1999 se dají rozdělit na dva typy, relační a objektové. Tedy normy, které dále rozšiřují relační model a ty, které usnadňují objektově orientovaný přístup.

Novinky normy SQL 1999 z pohledu relačního přístupu:

Nové datové typy: SQL: 1999 přináší nové datové typy: Boolean pro uložení pravdivostní hodnoty (ano, ne a neznámá), LOB (large data object) – typ pro uložení velkých datových entit, dále se dělí na BLOB (binary LOB) pro ukládání binárních dat např. Multimediální data a CLOB (character LOB) pro ukládání textu. SQL1999 nepovoluje užití těchto typů pro predikáty PRIMARY KEY a UNIQUE.

Dalšími novinkami jsou tzv. Hnízděné datové typy pole ARRAY a ROW, uživatelem definované datové typy, dědičnost datových typů, tabulek a jazyk pro psaní vnořených procedur.

U hnízděných datových typů podotýkají zastánci striktního relačního modelu, že je v rozporu už s 1NF, jelikož nesplňují atomičnosti jednoho záznamu.

Za objektově-relační databázové systémy jsou považovány na poli komerčních databází např. ORACLE (od verze , na poli open source databází je to především PostgreSQL.

3.5 Možnosti přístupu k datům uloženým v RSŘBD

Nejrozšířenější způsob přístupu k datům uloženým v relačních databázích je pomocí dotazů formulovaných v jazyce SQL(simple query language), popišme si základní postupy pro jejich umístění v kódu aplikace. Představme si situaci, ve které máme navrženo schéma databáze podle nejlepšího svědomí databázového specialisty a zároveň diagram aplikačních objektů, které pro svůj běh potřebují pracovat s daty z databáze. Předvedeme si několik přístupů jak k datům uloženým v databázi, tedy databázové vrstvě, přistupovat z aplikační vrstvy.

3.5.1 Přímý přístup do databáze

Postup, kdy jsou SQL dotazy do databáze kladeny prostřednictvím příslušného konektoru pokaždé, když chceme přistupovat k uloženým datům, bývá nazýván přímý přístup do databáze, někdy také přístup hrubou silou [15] .

Výhodou přímého přístupu do databáze je bezprostřední formulace dotazu v jazyce, který je pro dotazování se do databáze určený. Dotaz klademe v kontextu potřeby práce s perzistentními daty, bez nutnosti znalosti architektury aplikace a aplikačního rozhraní vrstev, které by přímý přístup nahrazovaly.

Nad jednoduchostí a přímočarostí zmíněného řešení však mohou převážit nevýhody. V aplikaci mohou být stejné dotazy formulovány vícekrát. V případě nutnosti dotazy měnit, ať už z důvodu optimalizace, změny databázového schématu, nebo dokonce přechodu na jiný databázový systém, je třeba dohledat, přepisovat a znovu testovat související části aplikace. Navíc některé SQL dotazy bývají při použití jejich čitelného zápisu rozsáhlé a jejich kombinace se zbylou logikou aplikace, napsanou v jiném programovacím jazyce, snižuje přehlednost a čitelnost kódu. Samozřejmostí přímého přístupu je i dobrá znalost SQL u programátorů, kteří potřebují v aplikaci pracovat s perzistentními daty.

3.5.2 Zapouzdření databázové vrstvy

Alternativou k přímému přístupu do databáze je konstrukce aplikačního rozhraní, nebo vrstvy aplikace, která tento přístup k perzistentním datům zprostředkuje a oddělí dotazování, nebo konstrukci dotazů do databáze, od zbylé aplikační logiky. Zmíněné postupy lze souhrnně nazvat zapouzdření databázové vrstvy.

Myšlenka zapouzdření nemusí nutně souviset s objektově orientovaným programováním (OOP). I v případě, že OOP nepraktikujeme, můžeme přístup k datům zapouzdřit například sadou funkcí, které budou poskytovat rozhraní pro přístup do databáze, prostředky programovacího jazyka použitého pro psaní zbytku aplikace. Nadále se však budeme věnovat pouze řešením popisovaným v kontextu objektového přístupu.

Pro popis a pojmenování obecných řešení, které se často opakují při návrzích a implementaci objektového modelu aplikací, se využívají tzv. návrhové vzory. Nyní se seznámíme s nejznámějšími návrhovými vzory, které se dotýkají problematiky zapouzdření databázové vrstvy. [16]

3.5.2.1 Table Data Gateway

Vzor Table Data Gateway navrhuje pro každou tabulku databáze vytvořit objekt, který zprostředkovává přístup k jejím záznamům. Instance tohoto objektu obsluhuje všechny záznamy v tabulce.

3.5.2.2 Row Data Gateway

Analogie Table Data Gateway s tím rozdílem, že navrhovaný objekt zpřístupňuje právě jeden záznam v databázi.

3.5.2.3 Active Record

Objekt nazývaný Active Record reprezentuje právě jeden záznam v databázi, zpřístupňuje perzis-

tentní data a přidává k nim aplikační logiku.

3.5.2.4 Data Mapper

Návrhový vzor popisující vrstvu aplikace, která zprostředkovává přenos dat mezi databází a objekty tak, aby objekty aplikace nepotřebovaly znalost datového zdroje. Z tohoto návrhového vzoru vycházejí ORM řešení, kterým se budeme věnovat ve zvláštní kapitole.

3.5.2.5 Data Access Object

Návrhový vzor Data Access Object spočívá v zapouzdření SQL dotazů, například pro všechny dotazy příslušící určitému business objektu, do jedné třídy. Instance této třídy pak slouží pro přenos dat mezi aplikačními objekty a databází, nebo jiným datovým úložištěm.

3.6 SWOT analýza relačních databází

3.6.1 Výhody

3.6.1.1 Nejrozšířenější řešení

O tom, že se jedná v dnešní době o nejrozšířenější technologii není pochyb. Na poli komerčních databází jsou to např. Oracle,Microsoft,IBM,Sybase, atd [17]., mezi open source databázemi jsou nejznámější PostgreSQL, MySQL nebo Firebird. Přínosem rozšířenosti je znalost SQL mezi programátory, dostatek literatury nebo například velká nabídka školení.

3.6.1.2 Existence metodik pro návrh relačního schématu

Jak jsme již zmínili, relační schéma je možné navrhovat různými způsoby, důležité však je, že existují pravidla normalizace umožňující zhodnotit jeho kvalitu. Zároveň algoritmy, pro převod konceptuálního modelu do normalizovaného relačního schématu.

3.6.1.3 Určené pro hromadné zpracování dat

Velkou konkurenční výhodou relačních databází je optimalizace na rychlost pro typické konfigurace hardware postaveného na soudobých technologiích. Pevná délka záznamu v tabulce umožňuje efektivní pohyb v rámci souboru. V relačních databázích jsou typicky data uchovávána v souborech s pevnou délkou věty. Pro práci s jedním záznamem nebo určitým atributem záznamu je potřeba méně operací seek, které jsou časově náročné. Databáze záznam nemusí vyhledávat, při přístupu k záznamu si spočítá, na které stránce a na které pozici se záznam nachází.

Podobně normalizace dat zajišťuje dekompozici tabulek se zbytečně velkým množstvím atributů a eliminuje redundanci dat.

3.6.1.4 Standardizovaný dotazovací jazyk

Při dodržení standardizovaných SQL dotazů je zajištěna přenositelnost mezi databázemi, které příslušnou normu splňují.

3.6.1.5 Možnosti optimalizace

Dotazy do databáze je možné optimalizovat za pomocí reformulace dotazu, vytváření indexů, nebo pohledů. Další možností je denormalizace již normalizovaného schématu, kdy dochází ke spojení více tabulek, ušetří se spojování pomocí funkce JOIN, čímž dochází k časové úspoře pří načítání dat z databáze na úkor zvýšení redundance, díky které naopak dochází k možnému zpomalení aktualizace pomocí příkazu UPDATE. Optimalizace znamená přizpůsobení struktury uložených dat potřebám aplikace. Při optimalizaci typicky dochází ke zlepšení výsledku u jedné veličiny na úkor jiné.[18]. Zrychlení vkládání dat na úkor zpomalení výpisu dat. Zrychlení na úkor vyššího nároku na paměťový prostor a zpomalení vkládání.

3.6.1.6 Dlouhá tradice

Dnešní RSŘBD jsou prověřeny časem. Dlouhá tradice je určitou zárukou spolehlivosti, která je ceněna konzervativními zákazníky.

3.6.1.7 Znalost mezi programátory

Díky tradici technologie relačních databází a díky jejím teoretickým základům patří výuka relačního modelu a SQL mezi základní předměty vyučované na vysokých školách se zaměřením na informatiku. Stejně tak existuje mnoho učebnic a školení, které se problematice věnují.

3.6.1.8 Client server technologie

Při přístupu k databázi je pro dnešní relační databáze typická architektura klient server. Kromě proprietárních konektorů do jednotlivých implementací existuje standardizovaný ODBC konektor, zajišťující jednotné API k databázi nezávislé na prostředí, které využívá klient tj. programovacím jazyku, databázovém systému ani operačním systému.

3.6.2 Nevýhody

3.6.2.1 Nepřenositelnost nad rámec implementovaného standardu

V souvislosti s postupným vydáváním novějších a novějších standardů pro jazyk SQL docházelo při vývoji jednotlivých databázových platforem k implementaci vlastních funkčností nad rámec standardů. Díky tomu je pak databázové schéma využívající nadstandardních funkčností hůře přenositelné na jinou platformu. Tato problematika se nejvíce dotýká jazyků pro psaní triggerů a vnořených procedur, proto platí, že čím více aplikační logiky je skryto v databázové vrstvě, tím obtížnější je jeho přenos na jinou platformu.

3.6.2.2 Problémy při změně schématu

Úprava databázového schématu musí být na některých platformách spojená s výpadkem, tedy dočasnou nedostupností dat. Změna schématu může způsobit také chybu v běhu aplikace. Dotaz může skončit chybou například při příkazu INSERT, nebo UNION, pokud bude do nějaké tabulky přidán sloupec a dosavadní dotazy v aplikaci nepočítají s takovou změnou a explicitně nevyjmenovávají sloupce, se kterými pracují. Těchto chyb se však můžeme vyvarovat důsledným používáním dotazů, ve kterých explicitně vyjmenovávají sloupce, se kterými pracují.

3.6.2.3 Nestarají se o správu verzí

Sledování změn v uložených datech nepatří mezi vlastnosti relačních databází. Správu verzí však lze doprogramovat přímo v databázové vrstvě pomocí triggerů spouštěných před aktualizací, nebo smazáním záznamu. Trigger pak zkopíruje přepisovaný, respektive mazaný záznam, nebo jiným způsobem uschová informaci o provedených změnách. Takto můžeme v rámci databáze naprogramovat verzování záznamů v tabulkách, nikoliv však už verzování samotných tabulek.

U platforem, které podporují transakce, pak do jisté míry lze mluvit o verzi záznamu v tom smyslu, že umožňují návrat do stavu, tedy k verzi záznamu, která byla aktuální na počátku transakce, tzv. rollback.

3.6.2.4 Pomalé a nedůsledné přijímání standardů

Přes velkou snahu o standardizaci se některé detaily jazyka SQL liší pro různé RSŘBD, což stěžuje přenos aplikací mezi nimi.

3.6.3 Příležitosti

3.6.3.1 Růst hardware a snižování jeho cen

Snižování cen pevných disků a růst jejich velikostí zvyšuje potenciál pro rozvoj v oblasti databázových aplikací. Díky své efektivitě při správě velkých objemů dat mohou relační databáze dále posílit svůj podíl na trhu.

Rozvoj NAND pamětí tzv. flash disků pak přináší potenciál pro další zrychlení a rozvoj technologií v oblasti SŘBD. Díky rozdílům oproti magnetickým diskům, zejména náhodnému čtení, může urychlit načítání dat. V tomto směru již byly zaznamenány pozitivní výsledky při použití indexů uložených v NAND paměti [19].

3.6.3.2 Zvyšování implementace standardu

Zvýšení míry implementace standardu u jednotlivých platforem usnadní migraci mezi jednotlivými platformami a umožní vytváření databázových aplikací nezávislých na použité platformě.

3.6.3.3 Zdokonalování ORM systémů

Problematice systémů zabývajících se objektově relačním mapováním (ORM) se budeme podrobněji věnovat ve zvláštní kapitole. Neměli bychom na ně však zapomenout pokud vyčíslujeme příležitosti pro SŘBD. Každý projekt, který využívá ORM nutně využívá i SŘBD.

3.6.4 Hrozby

3.6.4.1 Konkurenční technologie

Masivní nástup konkurenční technologie, kromě objektových databází to mohou být například proudové databáze nebo XML databáze.

3.6.4.2 Zkrácení životního cyklu aplikace

Vzhledem k tomu, že relační databáze typicky nepodporují verzování, tak v případě, že bude za života aplikace často docházet ke změnám struktury, může pak každá změna přinášet nemalou režii na změnu návrhu databáze a konverzi dat ze staré verze do nové.

3.6.4.3 Ústup od magnetických disků

V dnešní době dochází k růstu kapacity a snižování cen NAND pamětí tzv. flash disků, které umožňují náhodné čtení s náhodným přístupem.[19] Ztráta potřeby optimalizace na operaci seek by mohla vést ke zvýšení efektivity konkurenčních objektově orientovaných technologií.

4 Principy OOP

V této kapitole se seznámíme se základní terminologií OOP, která bude dále v práci používána.

> Objektově orientované paradigma je vývojová strategie založená na myšlence, že systém by měl tvořit soubor znovupoužitelných komponent, které nazýváme objekty.[20]

Podle knihy Thinking in Java [21] lze základní principy OOP formulovat takto:

- Cokoliv je objekt. Z čistě programovacího hlediska je objekt jednotka umožňující jednak skladovat nějaká data a jednak s nimi manipulovat definovanou sadou metod.
- Program je souhrn objektů, které si navzájem posílají "zprávy" co mají dělat. Zprávou se rozumí vyvolání nějaké metody.
- Každý objekt má svou paměť nezávislou na ostatních objektech (byť by šlo i objekty stejného typu, tedy třídy, viz dále)
- Každý objekt má svůj typ, tj. representuje nějaké konstantní vlastnosti definované popisem typu, všeobecně v OOP označovaném jako třída – class.
- Každý objekt konkrétního typu může přijímat stejné zprávy nebo jinak poskytuje stejný soubor metod.

Stručně, bez nároku na úplnost, lze tyto vlastnosti zjednodušit do věty:

> Každý objekt má stav, chování a identitu. Tedy objekt má interní data, jimiž je definován jeho stav, sadu metod, jimiž je definováno jeho chování a každý objekt je jednoznačně odlišen od ostatních objektů nějakou unikátní vlastností, např. v některých případech adresou v paměti.

Důležité vlastnosti objektů z hlediska objektově orientovaného programování:

Zapouzdření – k datové části objektu lze přistupovat pouze pomoci metod definovaných prostřednictvím **rozhraní**.

Opakovaná použitelnost – objekt se může vyskytovat v programu opakovaně (tj. ve více instancích). S tím souvisí tzv. skládání objektů, tj. objekt může obsahovat jiné objekty (a to včetně případných instancí téže třídy).

Dědičnost – určitý soubor objektů má některé vlastnosti shodné, a to ty, které objekty podědily od svých společných předků.

Polymorfismus – detailní chování objektu v závisí na jeho typu. Tedy když několik typů objektů poskytuje stejné rozhraní, pracujeme s nimi stejným způsobem, ale jejich konkrétní chování se liší podle typu konkrétního objektu.

4.1 4.1 Historie OOP

Vznik objektově orientovaných programovacích jazyků spadá už do šedesátých let 20. století, kdy postupně vznikal jazyk SIMULA, v nejznámější versi SIMULA67, který již byl plně objektově orientovaný jazyk v dnešním smyslu. Od sedmdesátých let dále vznikal jazyk Smalltalk, který existuje dodnes a byl svého času nejobecnějším objektovým jazykem.

Ovšem prvním v praxi masově nasazeným objektově orientovaným programovacím jazykem bylo C++, vycházejícího z programátory oblíbeného jazyka C. V čem spočívala revolučnost jazyka C++ oproti jazyku C, když kód stejné aplikace bylo možné napsat stejně tak v jazyce C? Rozdíl samozřejmě spočívá ve srozumitelnosti, délce a redundanci programového kódu.

Možná právě příklad stavby objektového programovacího jazyka nad procedurálním vedl k analogické myšlence postavit objektovou databázi nad relační databází.

Postupné rozšiřování programování v C++ mělo vliv na vývoj celé oblasti softwarového inženýrství a k obrovskému vývoji programovacích technologií jako např. návrhových vzorů, opakovaně využitelných frameworků a pod.

Z hlediska vlastního vývoje OOP je C++ zajímavé používáním vícenásobné dědičnosti. Jak se posléze ukázalo, tento aspekt OOP má své vedlejší nežádoucí efekty a proto byl postupně opouštěn, až při vývoji jazyka Java byl nahražen metodou jednoduché dědičnosti, ale s možností implementace dalších rozhraní, definovaných abstraktním prostředkem zvaným interface.

Vývoj od C++ k Javě lze považovat za určitou hlavní větev vývoje OOP, kromě těchto jazyků ovšem vznikla celá řada dalších objektově orientovaných jazyků jeko je Object Pascal, C#, Visual Basic,PHP, Python, Ruby a další.

5 Objektové databáze

V této kapitole se zaměříme na objektové databáze. V první části se seznámíme jejich historií, základními principy a odlišnostmi od relačních databází. Druhá část je věnována základním požadavkům na objektový model. V poslední části opět provedeme SWOT analýzu tohoto přístupu.

5.1 Rozvoj OOSŘBD

Objektové databáze vznikly z myšlenky vytvořit systém pro řízení báze dat, který by reflektoval vlastnosti známé z objektově orientovaného programování (OOP). Počátky objektových databází se datují do 80. a 90. let minulého století. Počátkem 80. let vedl Won Kim výzkumný projekt ORION v Texaské společnosti MMC. Z práce na projektu ORION pak čerpali vývojáři prvních komerčních OOSŘBD Versant a ITASCA. V 90. letech se objevují GemStone, O2, Vbase, ObjectivityDB a další. V roce 1991 vnikla ODMG (Object Database Management Group) spojující hlavních pět výrobců OOSŘDB za účelem standardizace v oblasti objektových databáze. ODMG vydalo postupně 3 standardy v letech 1991, 1995 a 2001. Poslední standard se pak stal základem pro specifikaci Java Data Objects (JDO). Potom bylo ODMG rozpuštěno.

Největší růst zaznamenal prodej OOSŘBD v 90. letech se špičkou v roce 2000. [22]Další postup nastal okolo roku 2004 s rozvojem open source OOSŘBD v čele s db4object.

Od roku 2006 OMG připravuje nový standard. Klade si za cíl postavit standard na teoretickém zázemí, které předchozí standardy postrádaly.[23] Využívá tzv. stack-based model vycházející z práce Kazimierz Subieta [24].

Přestože OOSŘBD měli ambici nahradit relační databáze pro jejich blízkost s objektově orientovaným programováním, dodnes se tak v masivní míře nestalo.

Srovnání základních rozdílů oproti relačním databázím je popsáno v následující tabulce:

Objektově orientovaný model	**Relační model**	Rozdíly
Objekt	Typ, entita	K objektu lze přistupovat pouze metodami jeho třídy.
Třída objektů	Schéma tabulky	Třída objektů obsahuje definice metod pro práci s objektem.
Hierarchie tříd	Databázové schéma	Hierarchie tříd je tvořena dědičností a ukazateli na cizí třídy, zatímco u (relačního) schématu je tvořena pouze cizími klíči.
Atribut	Atribut	Relační model, nepřipouští hnízděné atributy, např. Množina nebo pole, norma ANSI 1999 přináší i hnízděné datové typy.
Zasílání zpráv	Nemá	
Zapouzdření	Nemá	
Identifikátor objektu	Primární klíč	
Dědičnost	Některé implementace zavádějí dědičnost tabulek	Např. PostgreSQL

Tabulka 2: Srovnání OOSŘBD z hlediska datového modelování, převzato z [25] a doplněno

5.2 Datový model

V objektově orientovaných databázích jsou data modelována prostřednictvím objektů. Každý objekt je instancí právě jedné třídy, která popisuje datové typy jeho atributů a jeho chování prostřednictvím metod.

V této kapitole popíšeme základní vlastnosti objektového přístupu v databázových systémech. Jednotlivé platformy OOSŘBD se liší v závislosti na tom, jaké vlastnosti objektového přístupu podporují. Některé rozdíly pramení i z odlišností jednotlivých objektově orientovaných programovacích jazyků, jejichž vlastnosti se snaží jednotlivé přístupy reflektovat v datovém modelu.

V následujících podkapitolách popíšeme společné charakteristiky pro datový model, případně upozorníme na možné odlišnosti v závislosti na konkrétním programovacím jazyku.

5.2.1 Objekty a jejich identifikátory

Každý objekt má svůj objektový identifikátor (OID), který slouží k jeho jednoznačné identifikaci v rámci celé databáze. OID vzniká při vytvoření, tj. zpravidla při prvním uložení objektu v databázi.

Jeho hodnota se nemění po celý životní cyklus objektu. Vlastnost jednoznačné identifikace je podobná roli primárního klíče u relačních databází, nebo ukazatele či reference u objektově orientovaných programovacích jazyků. Na rozdíl od nich však volba OID nezávisí na hodnotě uložených atributů, ani na adrese v paměti.

5.2.2 Třídy, zapouzdření a extent

Pro popis obsahu, chování a hierarchie objektů slouží třídy. Každá třída se skládá z deklarační a implementační části. Deklarace zahrnuje názvy a datové typy atributů spolu s názvy a parametry metod. U atributů a metod mohou být v deklaraci dále odlišená práva přístupu na

- veřejné, které jsou přístupné i mimo instanci objektu
- privátní, ty jsou přístupné pouze prostředky objektu
- chráněné přístupné pouze z potomků.

Souhrn veřejně přístupných metod se nazývá rozhraní třídy.

Implementační část třídy pak obsahuje programový kód jednotlivých metod. Třídy tedy zapouzdřují datové úložiště tak, aby byl přístupný pouze pomocí rozhraní třídy. Zapouzdření v popsaném smyslu se řadí mezi jednu ze základních charakteristik objektově orientovaného přístupu.

Kromě toho, že třídy slouží k popisu struktury svých instancí, tj. objektů, mohou také obsahovat atributy společné pro všechny své instance. Příkladem mohou být kvantitativní a kvalitativní údaje o vytvořených instancích příslušné třídy, tj. počty objektů, statistické údaje o jejich atributech a podobně. Pro tyto účely slouží atributy a metody, které nazýváme statické a v programovacích jazycích bývají uvozovány klíčovým slovem *static*.

Některé objektové databáze [26] nám ke každé třídě nabízí kolekci obsahující všechny její instance, tato kolekce je nazývána extent třídy. Extent si lze představit jako implicitní statický atribut každé třídy tvořený kolekcí všech svých instancí. Extent třídy tvoří analogii tabulky v relační databázi a bývá využíván pro přístup pomocí dotazovacích jazyků.

5.2.3 Dědičnost a rozhraní

Další charakteristikou objektového přístupu je dědičnost. Třída B může podědit atributy a metody od jiné třídy A. Pak se třída B nazývá potomek třídy A a třída A se nazývá předek třídy B. Pro podědění používá většina programovacích jazyků klíčové slovo *extends*, v našem případě:

```
1: class B extends A
```

Z pohledu datového modelu jsou tedy veškeré atributy ukládané pro třídu B ukládané spolu s atributy třídy A. Některé databázové systémy umožňují vícenásobnou dědičnost, tj. třída B může mít dva různé přímé předky. Tuto vlastnost nazýváme vícenásobnou dědičností. Příkladem OOP jazyka, který tuto vlastnost má, je např. C++. Jiné databázové systémy inspirované např. Javou sice neumožňují vícenásobnou dědičnost, ale podporují práci s rozhraními.

Rozhraní se od třídy liší tím, že neobsahují implementační část, ale pouze předpisy deklarace metod, které musí implementovat jejich potomek. Při deklaraci rozhraní se většinou postupuje analogicky s definicí třídy, místo klíčového slova *class* je však užito klíčové slovo *interface*. Jedna třída obvykle může zdědit jinou třídu a zároveň implementovat některá rozhraní. Implementace rozhraní může být použita pro přijmutí nějaké role, do které je objekt postaven. Příkladem z praxe by

mohlo být rozhraní *prodejné*:

```
1: interface nacenitelne() {
2:   cena
3:  public vraťCenu();
4: }
```

Pokud chceme v naší třídě B implementovat rozhraní *prodejné,* rozšíříme hlavičku třídy:

```
1: class B extends A implements nacenitelne
```

V těle třídy B pak musíme doplnit tělo metody *vraťCenu*.

5.2.4 Vztahy mezi objekty, hierarchie

Téměř pro všechny databázové modely, má každý atribut doménu určující třídu všech objektů, které mohou být do tohoto atributu přiřazeny [26]. Vztahy mezi objekty jsou, kromě již zmíněné dědičnosti, vytvářeny pomocí atributů obsahujících jiné objekty nebo kolekce objektů. Vztah, kdy jeden objekt obsahuje jiné objekty, nazýváme agregace. Příkladem agregace může být objednávka agregující své položky. V případě, že jeden objekt obsahuje jako svou nedílnou součást další objekt nebo objekty, nazýváme kompozice. Ve zmíněných případech požadujeme po datovém modelu, aby při smazání objektu zajistil i smazání všech objektů, které obsahuje.

Poslední možnou vazbou mezi objekty je asociace. V tomto případě objekt odkazuje na další objekty prostřednictvím referencí. Samotný odkaz na objekt je pak realizován prostřednictvím již zmíněných OID. Vztahy, které jsou v relačním modelu řešeny vazební tabulkou, bývají v objektovém modelu řešeny pomocí kolekce referencí.

Hierarchie datového modelu objektových databází je tedy tvořena dědičností, agregací, kompozicí a referencemi.

5.3 Dotazování do databáze

5.3.1 Pomocí programovacího jazyka

Většina objektových databází podporuje hlavní objektově orientované jazyky jakými jsou JAVA, C++, nebo C#. Požadavek na načtení dat z databáze je pak zprostředkován čistě pomocí konstruktů příslušného jazyka.

5.3.2 Dotazovací jazyky

Další možností je OQL, který ODMG navrhla jako rozšíření SQL. V některých případech, jako jsou hromadné UPDATY a podobně je využití dotazovacího jazyka výhodnější. Navíc programátoři jsou na tento způsob zvyklí.

5.4 Návrh objektového modelu

Návrh modelu objektové databáze kopíruje objektový model na aplikační úrovni. Schéma databáze je pak tvořeno pouze těmi třídami pro které na aplikační úrovni vyžadujeme persistenci.

Při návrhu objektového schématu aplikace se využívají návrhové vzory [16], metoda podobná normalizaci relačního schématu, která se však v praxi příliš neujala je pak normalizace tríd, kterou navrhuje Larry Ambler v [20].

5.5 SWOT analýza OOSŘBD

5.5.1 Výhody

5.5.1.1 Hodí se pro procházení hierarchií objektů

Objektové databáze kopírují v databázové vrstvě propojení objektů pomocí referencí. Za běhu programu se jedná o adresu v paměti PC, v případě OOSŘBD se jedná o OID. OOSŘBD se proto hodí v případě, že na aplikační vrstvě potřebujeme k datům uloženým v databázi přistupovat prostřednictvím hierarchie referencí mezi nimi. Zatímco tabulka v databázi kopíruje plochou strukturu.

5.5.1.2 Ukládání komplexních datových struktur

Pro komplexní GIS aplikace (grafické informační systémy) je vhodná objektová databáze[27].

5.5.1.3 Praktické a efektivní pro vytváření prototypů

Ukládání objektů do databáze bez nutnosti mapování do relační databáze snižuje režii při vývoji systému. Dochází ke zkrácení a zpřehlednění programového kódu, který již není zatížen režií pro překlad objektového schématu do relačního, nebo konfigurací ORM systému.

5.5.1.4 Možnost větší vazby na programovací jazyk a menší vazby na SŘBD

Obecně je vývoji programovacích jazyků věnována větší pozornost a jejich použití je provozně levnější než použití SŘBD. Např. ceny překladačů jazyků jsou obvykle výrazně nižší než ceny SŘBD. I když u obou komodit existují i GNU verze, u překladačů je tento případ běžnější.

Ještě výraznější je tato úspora nákladů při používání aplikací. Zatímco použití překladače jazyka pro vytvoření aplikačního programu sebou nenese žádné náklady pro jeho uživatele, pak používá-li tento program SŘBD, znamená to obvykle pro uživatele náklady na DB server.

Předchozí hodnocení se týká architektury klient server. Neplatí pro tzv. embeded databáze, tj. databáze, které se přilinkují jako knihovna, zkompilují spolu s aplikací a za běhu využívají pouze zdroje klientského zařízení.

5.5.2 Nevýhody

5.5.2.1 Malá znalost mezi programátory

Objektové databáze nejsou příliš rozšířené.

5.5.2.2 Absence obecného datového modelu

Zatímco relační databáze stojí na solidním teoretickém základu relační algebry. Objektové databáze sledují hlavně potřeby z praxe, tedy snahu přizpůsobit perzistentní vrstvu aplikace OOP principům provozovaným na aplikační úrovni. Absence pravidel pro kontrolu korektnosti objektového modelu mohou komplikovat odhady časové náročnosti operací a případnou optimalizaci schématu.

5.5.2.3 Nehodí se pro množinové operace

Čistě objektové databáze, které nepodporují OQL rozšiřující SQL se nehodí pro hromadné operace nad velkými „plochými“ číselníky.

5.5.3 Příležitosti

5.5.3.1 Standardizace a přijímání standardů

Vyšší míra implementace standardů může usnadnit přechod mezi jednotlivými OOSRBD. Rozhraní pro perzistenci objektů i OQL mohou být použity pro OOSŘBD i pro ORM systémy (viz kapitola o ORM). V ideálním případě by pak výběr konkrétního databázového systém šlo přehodnotit v průběhu vývoje aplikace nebo dokonce i po jeho dokončení bez nutnosti refaktorizace kódu.

5.5.3.2 Ústup od magnetických disků

V dnešní době dochází k růstu kapacity a snižování cen NAND pamětí, tzv. flash disků, které umožňují náhodné čtení s náhodným přístupem.[19] Ztráta potřeby optimalizace na operaci seek by mohla vést ke zvýšení efektivity OOSŘBD.

5.5.4 Hrozby

5.5.4.1 Problémy s výkonem

Při použití OOSŘBD hrozí, že narazíme na problémy s výkonností v souvislosti s užitím paralelních transakcí nad rozsáhlou komplexní hierarchií objektů nebo pří potřebě hromadných uprav uložených dat.[28]

5.5.4.2 Vysoká režie s přechodem na jinou platformu

Z důvodu vzájemné nekompatibility dnešních OOSŘBD bude v případě nutnosti přechodu na jinou platformu nutná refaktorizace, tj. přepsání části stávajícího kódu aplikace. Potřeba migrace může být vynucena ukončením vývoje původně zvoleného databázového systému nebo již výše zmíněný-

mi problémy s výkonem.

5.5.4.3 Podstatně kratší historie

U produktů s menší tradicí hrozí větší nebezpečí použití technologie, jejíž vývoj bude v budoucnu zastaven.

6 Objektově relační mapování (ORM)

Jak jsme již zmínili v kapitole věnované relačním databázím, existuje více způsobů, jak při vývoji projektu kombinovat objektový model aplikačních objektů s užitím relační databáze pro uchovávání perzistentních dat. V této kapitole popíšeme a zhodnotíme nejrozšířenější řešení, kterým jsou ORM frameworky vycházející z návrhového vzoru Data Mapper [16].

6.1 Obecný popis ORM

Stejně jako u vzniku OOSŘBD stála u vzniku ORM systémů potřeba zajistit perzistenci aplikačních objektů. ORM systémy se snaží využívat schopností a potenciálu stávajících relačních databází. Rozdíly mezi oběma světy překlenuje až na aplikační úrovni. ORM tvoří vrstvu, která zapouzdřuje přístup do databáze a stará se o přenos dat mezi aplikací a relační databází v průběhu životního cyklu perzistentních objektů.

Při používání ORM systémů tedy dochází k zobrazení datového modulu do objektového modelu, a to pokaždé, když načítáme data z databáze při zavádění objektů do operační paměti, případně při požadavku o jejich aktualizaci, dle dat uložených v databázi. Naopak k zobrazení objektového modelu do relačního modelu databáze, dochází při vzniku nových objektů nebo při ukládání v paměti aktualizovaných objektů.

Přestože se v této kapitole snažíme o popsání obecných principů ORM, je třeba říci, že jsme vycházeli z některých vlastností konkrétních ORM systémů, jmenovitě Hibernate [29], PHP frameworku CakePHP [30] Doctrine PHP[31]. .

6.2 Úkoly mapování

6.2.1 Překlad entit

ORM zajišťuje překlad objektů na záznamy v databázi, tedy musí mít v době překladu informace o tom, které objekty se ukládají do kterých tabulek. Dále musí zajistit pro každou instanci objektu unikátní identifikátor. Ten se vytváří většinou až při ukládání objektu do databáze (pomocí sekvence, nebo sloupce typy autoincrement, v závislosti na databázové platformě). V databázi slouží pro zachování referenční integrity. V příkladu objednávky a jejích položek bude tabulka reprezentující položky objednávky obsahovat OID objednávky jako cizí klíč.

6.2.2 Překlad atributů

Datové typy relační databáze se liší od datových typů v programovacím jazyce. ORM nástroj má za úkol provést příslušnou konverzi. Některé datové typy, například pole, nelze uložit do databáze přímo. Pak je třeba použít jiný způsob, například uložení do jiné, tomuto atributu vyhrazené tabulky. Pokud je hodnotou atributu ukazatel na jiný objekt, rozhoduje o způsobu uložení mimo jiné druh a kardinalita příslušné asociace. Hodnoty některých atributů slouží pouze pro uchovávání dočasných (transientních) dat. Taková data nemají být ukládána do databáze.

6.2.3 Překlad asociací

Překlad asociací mezi entitami. V případě překladu asociací ORM frameworky využívají stejné postupy pro generování databázového schématu jako se využívají při převodu z ER- modelu [18].

6.2.4 Překlad dědičnosti

Zde je patrný největší rozdíl mezi relačním a objektovým přístupem k datům. Dědičnost v relační databázi může být zobrazena buď rozšiřováním jedné tabulky o sloupce, jejichž obsah má smysl jen pro některé řádky tabulky, nebo připojováním (joinováním) dalších a dalších tabulek zobrazujících vždy ty atributy potomka dané třídy, které rozšiřují množinu atributů základní třídy. První způsob je z praktického hlediska možný jen pokud počet odvozených tříd není velký a je předem znám. V každém případě pak vede k plýtvání objemem dat a porušuje normalitu databázového schématu. Druhý způsob komplikuje zacházení s daty, generované SQL dotazy do relačních databází jsou dlouhé a složité, vyhledávání a ostatní operace s daty se tím zpomalují.

6.3 Metadata a jejich získávání

Údaje o struktuře dat nazýváme metadata. Na jejich základě ORM framework rozhoduje o způsobu mapování. Některé frameworky metadata načítají z konfiguračních souborů, jiné z konfigurací obsažené přímo v programovém kódu. Dalším zdrojem mohou být příkazy popisující datový model, v SQL např. SQL příkaz DESCRIBE, případně příkazem SELECT ze systémových tabulek databázového systému. Metadata o struktuře objektového modelu mohou být, kromě konfiguračních souborů a anotací, získávána pomocí konstruktů příslušného programovacího jazyka, nebo pomocí skriptů, které je získávají na základě analýzy zdrojových kódů obsahujících deklarace jednotlivých tříd.

Dalším významným a u mnoha ORM frameworků často využívaným zdrojem metadat pro tvorbu vzájemného zobrazení datového a objektového modelu může být jmenná konvence. Tedy jména tabulek se převádí dle konvence na jména objektů, jména atributů na jména sloupců, třídy mají názvy odvozené od rodičů a tak podobně.

6.4 Způsoby chování a užívání ORM

V této kapitole se seznámíme s různými způsoby chování ORM frameworků. Názvosloví je převzato z frameworku Hibernate [29], v jeho případě se jedná o způsoby jeho užití. Zdaleka ne všechny frameworky nabízejí všechny čtyři scénáře se kterými se seznámíme. Chování některého mapperu může být v detailech odlišné od Hibenate. Pokud vycházejí z návrhového vzoru Data mapper měly by v nějaké podobě nabízet podmnožinu z následujících způsobů mapování.

6.4.1 Top Down (Shora dolů)

Při tomto přístupu ORM framework vygenerují tabulky databáze na základě existujících tříd objektového modelu a konfiguračních metadat.

6.4.2 Bottom Up (Odspodu nahoru)

Opačný přístup než u Top Down. Na základě hotového relačního schématu a případných konfiguračních metadat se vygenerují prázdné třídy, připravené pro doplnění business logiky.

6.4.3 Middle out (Od prostředka)

Tento přístup je specifický pro Hibernate [29]. Při jeho užití dochází k vygenerování objektového i relačního schématu pouze na základě konfigurace mapování, tj. konfiguračního souboru s metadaty popisujícími mapování. V praxi se nejprve spustí utilita pro vygenerování Java kódu s deklaracemi tříd a potom utilita pro vygenerování datového modelu na základě Java kódu a konfiguračního souboru s metadaty.

6.4.4 Meet In The Middle (Setkání v půli)

V tomto případě se vychází z hotového objektového modelu i hotového datového schématu, pro které je dodána konfigurace s metadaty. Tedy způsob mapování je ponejvíce v režii programátora. Vždy je dobré používat takový framework, který také umožňuje svému uživateli definovat mapování pro případy, kdy automatizované chování selže.

6.4.5 Generické mapování

Poslední ze způsobů mapování je tzv. generické mapování. ORM, které ho využívají, nepřekládají objekty na tabulky, ale ukládají si všechna data do společných tabulek: Názvy tříd, typy atributů, hodnoty atributů, vazební tabulku svazující třídu, typ atributu a hodnotu atributu a nakonec vazební tabulku nebo tabulky reprezentující různé asociace mezi objekty. Takový metadatový model popisuje Scott W. Ambler [32]. S podobným přístupem se setkáme i v kapitole věnované databázovému frameworku Garuda, který vlastně implementuje tuto formu ORM mapování a to přímo v ORSŘBD PostgreSQL.

6.5 SWOT analýza ORM

6.5.1 Výhody

6.5.1.1 Transparentnost databázové vrstvy

Při práci s daty programátor využívá abstraktní vrstvu. Nemusí psát SQL dotazy, o to se postará ORM. Aplikační programátor nemusí umět jazyk SQL.

6.5.1.2 Zůstává prostor pro přímý přístup do relační databáze

Přestože primárním úkolem ORM je překlenout rozdíly mezi aplikačními objekty a relační databází, vždy zůstává prostor pro přímé dotazy do databáze.

6.5.1.3 Přenositelnost kódu

Ten samý kód programu lze využívat nad různými platformami, konkrétně nad všemi, které podporuje ORM vrstva, která se v aplikaci použila. V případě přechodu na nepodporovanou platformu je třeba přizpůsobit pouze ORM vrstvu.

6.5.1.4 Hodí se pro tvorbu prototypů

Toto platí hlavně pro ORM systémy, které sami generují schéma databáze. Při změně požadavků, například změn kardinalit asociací mezi objekty, odpadá režie se změnou schématu databáze.

6.5.1.5 Standardizace

V případě, že využití ORM frameworku, který implementuje JPA, je možné migrovat mezi ORM frameworky v mezích implementovaného standardu. Tento bod se týká zejména ORM systémů pro JAVU.

6.5.1.6 Hojně rozšířené řešené

Zvláště díky široké podpoře a standardizaci v JAVE se jedná o rozšířené řešení.

6.5.1.7 Perspektiva snadného budoucího přechodu k OOSŘBD

6.5.2 Nevýhody

6.5.2.1 Skrývá chyby v návrhu databáze

V případě automatického generování databázového schématu může docházet k chybám, které způsobují redundanci uložených dat. ORM často nemá dostatečné údaje o relacích mezi ukládanými entitami, proto vychází z apriorních předpokladů a výsledný návrh může být zbytečně obecný. Tyto nedostatky se pak mohou projevit až v době, kdy je režie na změnu schématu a příslušných dotazů příliš vysoká.

6.5.2.2 Menší možnosti pro optimalizaci

Jednou z forem optimalizace dotazů do databáze je expedimentování s reformulací dotazů . ORM typicky dotazy do databáze generují a dále neumožňují jejich reformulaci.

6.5.2.3 Automatizace zvyšuje režii

ORM systémy často provádějí více dotazů do databáze, než by jich prováděl programátor při přímém přístupu. Některé dotazy navíc mohou být pro zjišťování metadat (DESCRIBE, COUNT a podobně). Jiné a mnohdy časově náročnější dotazy mohou být způsobeny načítáním dat do zásoby. Například načítání dat agregovaných objektů. Jindy naopak může docházet k mnohonásobně většímu počtu dotazů z důvodu, že agregované objekty nebyly načteny.

Tyto nedostatky se projevují, pokud ORM framework dostatečně nevyužívá znalosti metadat, neumožňuje programátorovi explicitně definovat, které objekty se mají načíst, nebo pokud programátor v dostatečné míře nevyužívá prostředků ORM a příliš se spoléhá na automatizaci načítání.

6.5.3 Příležitosti

6.5.3.1 Potenciál pro přechod mezi ORM a OOSŘBD

Standardizační proces pro ORM pro JAVU navázal na standard ODMG 3.0. Tedy principy pro přístup k objektové databáze. V případě sjednocení standardů pro ORM a OOSŘBD bude možné mezi těmito řešeními volně migrovat, v závislosti na tom, které poskytne optimální výkon pro vyvíjenou aplikaci.

6.5.4 Hrozby

6.5.4.1 Vítězství objektových databází

V případě masivního rozvoje objektových databází zanikne potřeba mapovat objekty do relačních databází.

6.5.4.2 Ztráta integrity dat při změnách hierarchie objektů

6.5.4.3 Skryté zvýšení režie programu

V případě využití automatizačních prvků hrozí, že ORM vrstva bude zvyšovat nároky na systémové zdroje. Kromě větších nároků na paměť můžou chod programu zpomalovat i některé dotazy do databáze, které nesouvisí s aplikační logikou, ale zjišťují pouze načítání metadat potřebná pro chod frameworku.

7 Garuda

V této kapitole prozkoumáme základní vlastnosti a architekturu frameworku Garuda (GA). Porovnáme jeho schopnosti s OOSŘBD a seznámíme se s nativním jazykem GDL pro deklaraci a implementaci tříd. I v této kapitole zhodnotíme klady, zápory, příležitosti a hrozby za pomoci metody SWOT analýzy, na základě analýzy se pokusíme navrhnout metodiku vývoje na GA.

7.1 Původ

Framework Garuda je příkladem systému, který vznikl na základě potřeb z praxe. Systém byl vyvinut ve společností ILIKETHIS! s.r.o., autorem programového kódu je Petr Krontorád, na analýze a návrhu frameworku se podílel Pavel Stěhule, který figuruje jako konzultant této práce.

Původní ideou autorů bylo napsat framework, který usnadní tvorbu systémů pro správu dokumentů (DMS). Výsledný produkt dnes v mnoha aspektech naplňuje principy objektově orientovaných databází. V době psaní této diplomové se jedná o komerční řešení, jeho kód [Příloha 2]zatím není volně dostupný.

7.2 Filozofie GA

Celý framework je založen na modelování objektů umístěných v rámci stromové hierarchie. Autoři GA vyšli z myšlenky, že při práci s dokumenty jsou uživatelé zvyklí na práci v rámci souborového systému, který stromovou hierarchii respektuje. V případě práce s tištěnými dokumenty je stejně tak běžné skladovat dokumenty v šanonech a jiných typech pořadačů. Při vyhledávání dokumentu například v knihovně, taktéž vycházíme ze znalosti hierarchie, kterou tvoří různá odvětví a jejich specializace.

7.3 Architektura

Systém využívá pro ukládání dat objektově relační databázi. V tomto případě PostgreSQL verze 8.1. Veškerá manipulace s daty probíhá prostřednictvím vnořených procedur PL/pgSQL[33]. Díky uložení atributů objektů v tabulkách relační databáze tu ale vždycky zůstává možnost pro přímý „relační “ přístup k tabulkám prostřednictvím SQL. Hodí se zejména pro potřeby komplexnějších dotazů, zároveň zůstává jako alternativa pro případy, kdy prostředky GA nejsou dostatečně výkonné. Přímý přístup do tabulek relační databáze je omezen pouze pro čtení, aby nemohlo dojít k narušení integrity objektového modelu.

GA využívá principů objektově orientovaného programování s tím rozdílem, že aplikační (business) objekty implementuje přímo v databázové vrstvě prostřednictvím API sestaveném z vnořených procedur PostgreSQL (dále jen Garuda API). Podobnou konstrukci pro ukládání objektového schéma navrhuje také Robert J. Muller [18]. Garuda API tvoří ORM systém přímo v databázové vrstvě. Kromě rozhraní pro práci s objekty zároveň přichází s vlastním programovacím jazykem pro definice tříd.

Kromě základních vlastností OOP, jakými jsou deklarace tříd, jejich metod a atributů, přidává GA svým objektům některé implicitní vlastnosti. Každý objekt nese informaci o příslušnosti k některé rodině objektů – family_id, umístění ve stromové struktuře – path_identifier a je mu přiděleno workflow. Tyto vlastnosti vycházejí z ambice GA býti framework, který usnadní vývoj systémů pro

správu dokumentů (zkráceně DMS z anglického document management systems).

7.4 Garuda definition language (GDL)

Garuda definition language dále jen GDL [Přílohy 4,5]byl původně vytvořen pro tvorbu workflow. (Původní název WFScript se dosud vyskytuje v části dokumentace věnované tvorbě workflow[Příloha 5]). Postupně byly přidány další jazykové konstrukty a v současné době slouží také pro deklaraci tříd GA. Nyní si popíšeme základní syntaxi jazyka GDL.

7.4.1 Definice tříd

Syntaxe pro definici tříd:

```
1: class [domain_name.]class_name [extends base_class_name[[ another_base_class_name]{..}]
2:                  [>> Popis třídy
3:                   >> Může být rozdělen na více řádek. Každá řádka komentáře začíná >>]
4:                  [flags: [droppable][,[abstract]][,[versioned]][,[materialized]]]
5:                  [children: [allowed_child_class_name,]{..}]
6:                  properties:
7:                          [::property_name property_type
8:                               [>> popis atributu]
9:                               [default: default value]
10:                              [fill: const|api|sql : fill_value]
11:                              [flags:[array][,[protected]][,[required]][,[immutable]]]
12:                         ]
13:                         {..}
14:                 [methods:
15:                         [::method_name [returns [setof] return_datatype]
16:                                 [>> popis metod
17:                                 [[param: param_name param_datatype
18:                                         >> parameter description
19:                                  ]
20:                                 {..}
21:                                 ]
22:                                 [flags:[protected][,][static][,][private]]
23:                                         method source code
24:                         ]
25:                         {..}
26:                 ]
```

Popis syntaxe (popis je uvozený číslem popisovaného řádku)

1: Každá třída je deklarovaná v rámci domain_name. Může být potomkem jedné nebo více dalších

tříd.

4: Třídám může být přiřazen flag, který mění jejich vlastnosti:

- **droppable** - instance třídy lze mazat
- **abstract** - nelze vytvářet instance třídy, klasický konstrukt OOP, deklarace třídy slouží jako vzor, instancovat lze až potomky této třídy
- **versioned** - instance třídy si uchovává historii změn
- **materialized** - objekty třídy jsou uloženy ve vlastní tabulce

5: klíčové slovo **children** - umožňuje vyjmenovat povolené třídy, které je možno přiřazovat do ve stromové struktuře GA pod deklarovaný objekt

7.4.2 Definice GA worklfow

7.4.3 Krok

Definice workflow se vždy vztahuje k uvedené třídě a jejím potomkům, pokud potomci nemají definováno jiné. V průběhu své existence nebo životního cyklu prochází instance třídy, tj. objekt, různými stavy, v terminologii GA kroky. Krok je uvozen klíčovým slovem **step**.

7.4.4 Fáze

Jeden krok se dále dělí na fáze uvozené klíčovým slovem **faze.** Fáze specifikuje určitou operaci s objektem. V rámci jedné fáze dále rozlišujeme události. Fáze dělíme na implicitní, které vyvolá framework, a explicitní, definované programátorem, ošetřující rozhodnutí zaslané typicky uživatelem. Události v rámci fáze jsou uvozené klíčovým slovem **on.**

Mezi implicitní fáze patří:

- IS_CREATED - Nastává po vytvoření instance.
- BECOME_CHILD - Nastává po té, co je mu přiřazen otcovský objekt v rámci stromu objektů.
- ACCEPT_CHILD - Jedná se o obsluhu pojmenované události, kterou zaslal uživatel.
- Nastane při přiřazení potomka v rámci stromové hierarchie objektů
- WANT_LEAVE - Nastane při pokusu o odpojení se od rodičovského objektu v rámci stromu.
- CAN_LEAVE - Blok příkazů je vykonán v případě úspěchu fáze WANT_LEAVE.
- WANT_GET_SELF - Je vyvolána při požadavku na získání (klíčové slovo **get**) vlastnosti objektu.
- WANT_GET_CHILDREN - Je vyvolána při požadavku na získání vlastnosti potomka.

Vyvolání zmíněných fází nelze explicitně ovlivnit. V rámci definice obsluhujeme jejich události, každá implicitní fáze obsahuje pouze jednu událost uvozenou klíčovým slovem **default**.

Explicitní fáze obsluhují pojmenované události. Blok explicitní fáze je uvozen klíčovým slovem **SUBMIT**.

V rámci jednotlivých fází, resp. eventů v rámci fází, lze nejen přecházet do dalšího kroku a provádět kontroly zda se smí přejít do dalšího kroku, ale také lze explicitně vyvolávat operace s objekty – programovat business logiku.

Příklad workflow:

```
27:class Reklamace
28:  step new_refund
29:  ["/cs:Nová reklamace/en:New refund"]
30:    SUBMIT
31:      on SEND["/cs:Poslat ke schvaleni/en:Send for acceptance"]
32:          goto waiting_for_acceptance
33:  step waiting_for_acceptance
34:    SUBMIT
35:      on REJECT["/cs:Zamítnout/en:Reject"]
36:          move ^inflictor:/Acceptance to ^object_uri@/Refunds:/Rejected
37:          goto rejected_refunds
38:      on ACCEPT["/cs:Prijmout/en:Accept"]
39:          move ^inflictor:/Acceptance to ^inflictor:/Running
40:          move ^inflictor:/Acceptance to ^object_uri@/Refunds:/Running
41:          link ^object_uri@/Refund:/Running
42:          goto finished_refund
43:  step rejected_refund
44:  step finished_refund
45:end class
```

Příklad ukazuje workflow pro schválení reklamace.

Na řádcích 29, 31, 35 a 38 je užitá syntaxe pro vkládání metadat, dat, které se mohou zobrazovat v uživatelském rozhraní v závislosti na nastaveném jazyce.

V příkladu zobrazuje 4 kroky. Počáteční krok new_refund, obsahuje explicitní fázi s událostí SEND, která způsobí přechod do stavu waiting_for_acceptance.

Krok waiting_for_acceptance obsahuje dvě události, reprezentující schválení a neschválení reklamace. V závislosti na události dochází k přesunu objektu do uzlu obsahující probíhající reklamace, resp. zamítnuté reklamace a přechodu do některého z koncových stavů.

V kódu se objevuje **inflictor.** Jedná se o odkaz na objekt, který vyvolal událost.

7.5 Garuda API

Veškerá manipulace s objekty GA je zpřístupněna pomocí Garuda API, která je tvořena sadou PL/pgSQL funkcí.[Příloha 3].

Následující ukázka zdrojových kódů, která je převzata z testovacích skriptů v [Příloze 3], demonstruje úspornost jazyka GDL ve srovnání se zápisem stejného kódu v pomocí PL/pgSQL API frameworku Garuda. Jeho kód testuje transformaci cyklů v GDL do PL/pgSQL

Zápis v jazyce GDL:

```
1: // Loops
2: // using method call result as row iterator
3: let $obj = object get 1:'/G'
4: let $i = 0
5: while call $obj.ListNodes( 1, 2 )
6:         $i = $i + 1
7: end while
8: // reading row properties
9: while call $obj.ListNodes( 1, 2 )
10:        let $path = row.path_identifier
11:end while
12:// using loop to return setof values (creating new row values)
13:let $my_property_name = 'date'
14:let $dyn_property = 'version'
15:while call $obj.ListNodes( 1, 2 )
16:        row.my_path = 'my_prefix' + row.pathIdentifier
17:        // dynamically changing record property name
18:        row.$my_property_name = row.dateCreated
19:        // acquiring property by name dynamically
20:        row.dynamic = row.$dyn_property
21:        // doing it all dynamically
22:        row.$my_property_name = row.$dyn_property
23:        return next
24:end while
```

Zkompilováním vnikne následující kód v jazyce PL/pgSQL:

```
1: --/ Loops
2: --/ using method PERFORM garuda.wfscript___LibraryVoidCall('result as row iterator
3: _obj := object get 1:'/G';
4: _i := 0;
5: FOR _call_response_in IN SELECT * FROM garuda' || '.' || 'object_Call', ARRAY
   [coalesce(_obj || '.'
6: || 'ListNodes'::varchar,''), coalesce( ARRAY [coalesce( 1::varchar::varchar,''),
7: coalesce(''):::varchar,''), coalesce( coalesce( 2 ::varchar::varchar,''),
```

```
8: coalesce('')::varchar,'')]::varchar[],              _request.inflictee_id]::varchar[],
   _request.inflictee_id);
9: LOOP
10:IF 2<>_call_response_in.return_type OR 0 = _call_response_in.row_total THEN
11:EXIT;
12:END IF;
13:_row_in       :=       _call_response_in.row;       _actual_row_iterator_in       :=
   _call_response_in.row_iterator;
14:_i := _i + 1;
15:END LOOP;
16:--/ reading row properties
17:FOR _call_response_in IN SELECT * FROM garuda.object_Call(_obj || '.' || 'ListNodes',
   ARRAY
18:[coalesce(      1::varchar,''),      coalesce(      2      ::varchar,'')]::varchar[],
   _request.inflictee_id) LOOP
19:IF 2<>_call_response_in.return_type OR 0 = _call_response_in.row_total THEN
20:EXIT;
21:END IF;
22:_row_in       :=       _call_response_in.row;       _actual_row_iterator_in       :=
   _call_response_in.row_iterator;
23:_path := garuda.wfscript___GetRowValue(_row_in,'path_identifier');
24:END LOOP;
25:--/ using loop to return setof values (creating new row values)
26:_my_property_name := 'date';
27:_dyn_property := 'version';
28:FOR _call_response_in IN SELECT * FROM garuda.object_Call(_obj || '.' || 'ListNodes',
   ARRAY
29:[coalesce(      1::varchar,''),      coalesce(      2      ::varchar,'')]::varchar[],
   _request.inflictee_id) LOOP
30:IF 2<>_call_response_in.return_type OR 0 = _call_response_in.row_total THEN
31:EXIT;
32:END IF;
33:_row_in       :=       _call_response_in.row;       _actual_row_iterator_in       :=
   _call_response_in.row_iterator;
34:_row_out.names[_actual_row_column_iterator_out] := 'my_path';
35:_row_out.values[_actual_row_column_iterator_out] := coalesce(('my_prefix' +
36:garuda.wfscript___GetRowValue(_row_in,'pathIdentifier'))::varchar,'');
37:_actual_row_column_iterator_out := _actual_row_column_iterator_out + 1;
38:--/ dynamically changing record property name
39:_row_out.names[_actual_row_column_iterator_out] := _my_property_name;
40:_row_out.values[_actual_row_column_iterator_out] :=
41:coalesce((garuda.wfscript___GetRowValue(_row_in,'dateCreated'))::varchar,'');
42:_actual_row_column_iterator_out := _actual_row_column_iterator_out + 1;
```

```
43:--/ acquiring property by name dynamically
44:_row_out.names[_actual_row_column_iterator_out] := 'dynamic';
45:_row_out.values[_actual_row_column_iterator_out] :=
46:coalesce((garuda.wfscript___GetRowValue(_row_in,_dyn_property))::varchar,'');
47:_actual_row_column_iterator_out := _actual_row_column_iterator_out + 1;
48:--/ doing it all dynamically
49:_row_out.names[_actual_row_column_iterator_out] := _my_property_name;
50:_row_out.values[_actual_row_column_iterator_out] :=
51:coalesce((garuda.wfscript___GetRowValue(_row_in,_dyn_property))::varchar,'');
52:_actual_row_column_iterator_out := _actual_row_column_iterator_out + 1;
53:return next
54:END LOOP;
```

První zápis v jazyce GDL je podstatně úspornější a díky tomu srozumitelnější. GDL je implementováno pomocí sady pravidel zapsaných pomocí regulárních výrazů. Jeho kompilace spočívá v aplikaci těchto pravidel a převodu do jazyka PL/pgSQL. Nevýhodou tohoto konceptu je znesnadnění ladění chyb, protože ty jsou odhaleny až při spuštění PL/pgSQL skriptu. Programátor tedy musí být dostatečně znalý jazyka PL/pgSQL, aby dokázal odladit vygenerovaný kód.

7.6 Objektové vlastnosti Garudy

V kapitole věnované OOSŘBD jsme popsali základní principy objektového přístupu. Podle stejných kritérií zhodnotíme objektové vlastnosti GA.

7.6.1 Zapouzdření

Omezení přístupu k Zapouzdření objektů je realizováno zakázáním přímého přístupu do tabulek.

7.6.2 Class extent

Výpis instancí třídy je možný pomocí garuda API voláním funkce:

```
1: garuda.object_LookupWithProperties    (_family_id    integer,    _class_name    varchar,
   _include_inherited bool, _properties varchar, _skip_root boolean)
```

7.6.2.1 Perzistence

GA nabízí implicitní perzistenci, zajišťuje ji v GDL pomocí operátoru new:

```
1: new Person as Jan with ['name', 'Jan'],['email', 'jan@janik.com']
2: let $o4 = new Class as $path with $params at 1:'/G/domains/public'
```

7.6.3 Explicitní mazání

ano

```
3:  ldelete $family:$path_2
```

7.6.4 Přímý přístup k atributům

Přímý přístup k atributům objektů je umožněn v GDL pomocí příkazů GET a SET

let $name= get $obj property name

Specifikace domény pro atributy

7.6.5 Referenční integrita

Zachování referenční integrity si GA implicitně hlídá. Smazání objektu z databáze se provádí pomocí funkce garuda.object___Delete . Tato funkce před smazáním objektu ověřuje, že na objekt není odkazováno odjinud.

7.6.6 Zapouzdření metod v databázi

Klasické OOSŘBD rozšiřují aplikační objekty o perzistenci jejich dat, metody objektů nejsou vyvolatelné přímo v databázi. Oproti tomu GA už v databázové vrstvě splňuje další princip známý z objektově orientovaného programování, kterým je zapouzdření aplikačních dat spolu s metodami pro práci s nimi. Díky tomu je možné přistupovat k datům, řídit život a umístění objektů pouze prostřednictvím SQL funkcí Garuda API.

7.7 Vzorový příklad

Vraťme se k zadání vzorového příkladu řešeného v předchozích kapitolách. Jeho zadání zde již nebudeme opakovat.

7.7.1 Řešení

Deklarujeme třídy Dokument a Objednávka, protože Smlouva a Faktura jsou analogické.

```
1: Class Dokument
2:    properties:
3:                ::datum_vystaveni_date
4:                ::datum_posledni_aktualizace
5:
6: Class Objednavka extends Dokument
7:    properties:
```

```
8:                ::popis name
9:                ::pocet_kusu number
10:               ::number-float-positive
11:   children: Zbozi
12:
13:Class Zbozi
14:   properties:
15:               ::kod_zbozi name
16:               ::cena number-float-positive
```

Objednávka agreguje zboží. Na řádce 11. jsme povolili připojování objektů typu zboží k objektům typu objednávka.

Nyní se zaměříme na změny stavů. Je třeba říci, že z hlediska GA jsme už v tuto chvíli splnili zadání na té úrovni obecnosti, na které bylo zadáno. Framework totiž po bližší specifikaci zadání již nyní umožňuje definovat změny stavů pomocí workflow a povolovat jednotlivé přechody stavů, volání Garuda API funkce :

```
1: garuda.object_GrantPermission (_obj_id integer, _role_id integer, _permission_type_code varchar, _wf_step_name varchar)
```

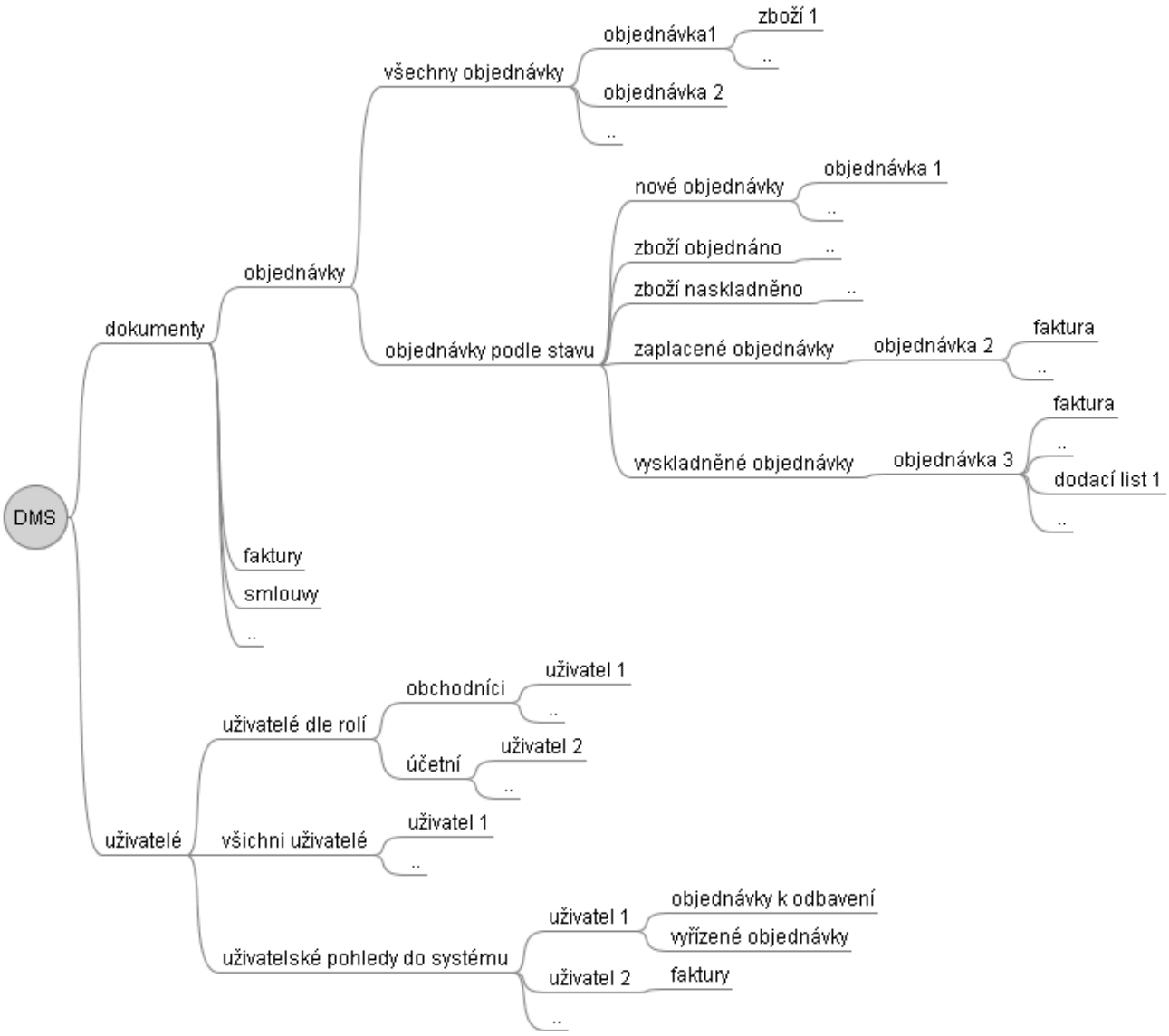
Na druhou stranu je třeba podotknout, že GA nepředpokládá dynamickou tvorbu workflow. Definici Workflow je třeba při každé změně znovu zkompilovat. Workflow tedy budeme moci třídě doplnit teprve na základě seznamu stavů (v terminologii GA kroků) a povolených přechodů mezi kroky.

Pro každý krok objednávky je pak třeba vytvořit definiční blok, ve kterém je definován přechod do dalšího kroku na základě příchozích událostí rozdělených do jednotlivých fází kroku.

Možné události vyvolané z uživatelského rozhraní se deklarují v rámci explicitní fáze SUBMIT. Zbylé, implicitní, události jsou vyvolané změnami v hierarchii objektů. Příkladem může být přidání zboží do objednávky. Pokud je zboží v objednávce agregováno pomocí stromové struktury GA, je po přidání nového potomka, v rámci stromu, vyvolána implicitní fáze ACCEPT_CHILD.

7.7.2 Využití potenciálu GA

Uvedené řešení příkladu zkusíme trochu rozšířit a předvést potenciál GA. Některé význačné stavy objednávky můžeme uživateli reprezentovat polohou objednávky ve stromové hierarchii. Obrázek 7.1 na stránce 44 zobrazuje prostřednictvím myšlenkové mapy možnou hierarchii objektů v souladu s filozofií GA. V obrázku se vyskytují některé objekty vícekrát, toto je možné v GA dosáhnout pomocí linků(odkazů), podobně jako u filesystému PC. Notace list a dvě tečky naznačuje možnost umístění více objektů.



Obrázek 7.1: Příklad možné hierarchie GA pro vzorový DMS

7.8 SWOT analýza Garudy

7.8.1 Výhody

7.8.1.1 Podpora stromové struktury

GA se hodí pro tvorbu aplikací, ve kterých se k objektům přistupujeme průchodem stromové struktury. Například aplikací, které umožňují uživatelům přístup k aplikačním datům pomocí emulace souborového systému (např. Web Dav).

7.8.1.2 Podpora workflow

Díky podpoře workflow umožňuje GA vývoj systémů založených na vzájemné interakci modelovaných objektů.

7.8.1.3 Podpora verzování

GA podporuje verzování objektů užitím klíčového slova versioned při deklaraci třídy.

7.8.1.4 Aplikační logika je prováděna v databázi

Aplikační logika pro manipulaci s uloženými objekty je uložena a prováděna přímo v databázi. Díky tomu k přístupu a manipulaci s objekty vystačí klientský software schopný připojit se k SQL databázi. Informační systém tedy může být dohromady tvořen mnoha různými rozhraními pro práci s uloženými objekty.

7.8.2 Nevýhody

7.8.2.1 Nepřenositelnost

Celý framework Garuda je v současné době implementován v jazyce PL/pgSQL platformy PostgreSQL. Pro přenesení projektu napsaného za pomoci Garudy by bylo potřeba přepsání Garudy do jazyka vložených procedur jiné platformy.

7.8.2.2 Obtížná optimalizace

V tuto chvíli má GA poměrně malou možnost pro optimalizaci dotazů do databáze. Pokud nepočítáme změnu hierarchie objektů, nebo kladení přímých dotazů do tabulek, kam GA mapuje objekty, zbývá užití příznaku materialized při deklaraci třídy, které způsobí namapování objektů do pro třídu vyhrazené tabulky databáze.

7.8.2.3 Vlastní definiční jazyk GDL

Každý framework nabízí své prostředky, které byly vymyšleny proto, aby sjednotili a zjednodušili práci programátorů. Cenou za ně je nutnost osvojit si práci s ním natolik, abychom nabízených prostředků využili. Pro práci s GA je nutné osvojit si další programovací jazyk, odlišný od jazyka v

kterém je naprogramován zbytek aplikace. Nutnost školení zvyšuje režii pro vývoj produktů nad GA. Samotná implementace, která spočívá v překladu GDL do pg/PLSQL, navíc vyžaduje i dobrou znalost jazyka pg/PLSQL pro ladění a odstraňování chyb ve výsledném kódu.

7.8.2.4 Nekompletní dokumentace

Dokumentace GA [Přílohy 3-5] není příliš obsáhlá ve srovnání s funkcemi, které framework nabízí. U některých funkčností nezbývá než vyzkoušet, jak fungují nebo nahlédnout do zdrojového kódu. Obě možnosti jsou časově náročné. Dokumentaci by prospělo, kdyby obsahovala více vzorových příkladů, případně dobře zdokumentovaný postup pro vytvoření vzorové aplikace.

7.8.3 Příležitosti

7.8.3.1 Vhodně vytvořená metodika vývoje

Vhodně vytvořená metodika vývoje by mohla zjednodušit vývoj informačních systémů za pomocí Garudy a tím přispět k nasazení na komerčních projektech. Jakékoliv další využívání Garudy naopak může mít pozitivní zpětnou vazbu pro další rozvoj systému.

7.8.3.2 Prostor pro další rozvoj

Tvorba objektů, které je možné využít ve více projektech. Součástí zdrojových kódů je konektor jazyka PHP do GA. Dopsání konektorů pro další objektově orientované programovací jazyky (například Java, C#, C++, Python) by zvyšovalo možnosti dalšího využití frameworku.

7.8.3.3 Otevření kódu

Architektura GA je jistě inspirativní architektura, která by mohla zaujmout open source komunitu a přilákat schopné programátory. „Velcí programátoři většinou trvají na použití open source softwaru. Ne jen protože je lepší, ale také proto, že jim dává možnost větší kontroly. Velcí programátoři trvají na kontrole.“[34]. Součástí dokumentace každé procedury GA API je také její zdrojový kód. Otevření kódu pro účely zpětné kontroly, ale i pro dohledání nezdokumentovaných funkčností, je již dnes pozitivním rysem frameworku[Příloha3].

7.8.3.4 Portace pod další databázové systémy

Blízkost jazyků pro psaní vnořených procedur pg/PLSQL databáze PostreSQL s jazykem PLSQL databáze Oracle by měla umožňovat přenos GA pod nejrozšířenější komerční databázový systém Oracle.

7.8.4 Hrozby

7.8.4.1 Snížení výkonu po naplnění reálnými daty

Jak jsme již zmínil, GA je svého druhu ORM systém, tedy vztahují se na něj podobná rizika a hrozby jaké jsme již diskutovali ve SWOT analýze ORM. Pro uživatele GA jsou relační tabulky dostupné, nicméně nemůže příliš ovlivnit podobu databázového schématu. Jedinými prostředky pro něj jsou již zmíněné „materializované“ třídy a případně přímé dotazy do relační databáze.

7.8.4.2 Ukončení vývoje.

V případě ukončení vývoje a podpory GA se může jeho využití na projektu stát přítěží. Vzhledem k specifickým vlastnostem frameworku pak případný přenos pod jinou platformu, znamená přepsání významné části kódu aplikace.

7.8.5 Diskuze SWOT analýzy

Podpora stromové struktury a chování workflow jsou do jisté míry výhody, proto jsme je také mezi výhody zařadili. Je však třeba podotknout, že jsou to základní vlastnosti GA, na kterých je chování celého frameworku postaveno. Tedy pokud budeme na GA pohlížet jako na Framework pro podporu workflow, pak je do jisté míry redundantní uvádět podporu workflow mezi jeho výhody. Pokud však budeme na GA pohlížet jako na datové úložiště pro DMS. Pak může být podpora workflow ve srovnání s jinými DMS systémy považováno za konkurenční výhodu.

8 Návrh metodiky vývoje pro Garudu

V této kapitole představíme návrh metodiky pro vývoj informačních systémů za pomoci frameworku Garuda. V návrhu se pokusíme zohlednit výsledky předchozí analýzy.

8.1 Důvody pro užívání metodik

Snad každý programátor si vyzkoušel psaní aplikace způsobem, kdy začal rovnou programovat bez předem stanovené koncepce. Cesta k možnému úspěchu, často také neúspěchu, zřejmě vedla přes naprogramování dílčích částí aplikace a jejich postupnému propojení. Někdo raději začne řešením jednodušších částí, tam kde je mu známý postup, tedy řeší nejprve problémy, u kterých dopředu zná řešení. Jiný se naopak nejprve snaží překonat úskalí, u kterých si je vědom největších těžkostí. To už závisí na osobní preferenci programátora. Jisté však je, že výsledky práce a čas pro realizaci projektu lze jen stěží při podobném postupu odhadovat. Výsledky takových snah budou o to horší, kolik programátorů s různými názory na způsob řešení a osobními zvyklostmi při programování se na projektu podílí. Čím větší systém je vyvíjen, tím větší se klade důraz na metodiku analytických činností a softwarový design.

Stejně jako lidé i programátoři a softwarové firmy se svými chybami učí, proto je dobrým zvykem vyvíjet aplikace koncepčně, podle předem stanovených a ověřených postupů, které souhrnně nazýváme metodikou. Metodikami budeme rozumět souhrny postupů, které se zabývají vývojem a řízením projektu (nebo softwarového produktu) po celý jeho životní cyklus. Tedy od začátku, tj. sběru požadavků, přes analýzu, design, realizaci až po testování, předání. Práce na projektu tím mnohdy nekončí, tedy metodika se může zaměřit i na postupy pří následných úpravách a vývoji nových verzí projektu nebo produktu.

Analytický návrh informačního systému má podobný význam jako architektonický plán při stavbě domu. Dům lze postavit bez projektu, nicméně případná přestavba domu je mnohokrát nákladnější, než změna ve fázi projektování. Podobně je tomu i při tvorbě informačních systémů. Analýza a projekce tvoří nezanedbatelnou část při vývoji projektu, ale případná režie spojená s neúspěchem je mnohonásobně vyšší než režie pro tvorbu analýzy.

Vývoj softwaru lze rozdělit do několika fází:

1. Analýza - sběr a formalizace požadavků, tvorba modelů systému
2. Design - výběr technologie, návrh realizace
3. Realizace - programování, testování, nasazování do provozu

Na jednotlivé fáze vývoje lze navíc pohlížet z pohledu různých aktérů vývoje, kterými jsou zejména:

Projektový management - Jeho úkolem je naplánovat a řídit projekt. Zodpovídá za časový plán projektu, alokaci lidských zdrojů a dohlíží na komunikaci mezi participanty projektu.

Analytici - Úlohou analytiků je formalizovat zadání a připravit podklady pro programátory a informační architekty. Pro formalizaci je zvykem využívat grafickou notaci pro vytváření různých pohledů na systém.

Informační architekti – Vytvářejí návrh architektury aplikace

Programátoři - Programují kód programu.

V praxi se mohou jednotlivé role překrývat v závislosti na znalostech a velikosti vývojového týmu.

Metodiky pro vývoj software se snaží postihnout jak práci analytiků a informačních architektů, tak práci programátorů a hlavně projektových managerů. Poslední jmenovaní musí celý proces vývoje řídit a organizovat. V našem případě se však soustředíme na práci analytiků a informačních architektů. Tedy naším cílem bude popsat způsoby jak uchopit zadání projktu tak, abychom byli schopni jeho vývoje pomocí frameworku GA, případně se dokázat včas rozhodnout, že tento framework, se pro vývoj konkrétního projektu vůbec nehodí.

8.2 Tvorba metodiky pro GA

Naším cílem je tvorba metodiky pro vývoj informačních systémů, zejména pak systémů pro řízení životních cyklů dokumentu v podniku nad Frameworkem GA. Nebudeme se však snažit navrhnout zcela novou metodiku, která by nevycházela z běžně užívaných postupů při tvorbě aplikací prostředky OOP. Naší snahou bude nalézt popsat prostředky pro modelování specifických vlastností objektového modelu GA. Pro výběr možných pohledů na systém budeme vycházet z dnes nejrozšířenějšího jazyka pro modelování UML[35],

8.2.1 Od SWOT analýzy k metodice

Projdeme SWOT analýzu GA a pro jednotlivé body budeme hledat uplatnění při návrhu metodiky vývoje nad GA.

8.2.1.1 Možnosti využití výhod GA

Pro každou výhodu popsanou ve SWOT analýze se pokusíme nalézt roli při tvorbě metodiky.

8.2.1.1.1 Podpora stromové struktury

Součástí metodiky by měly být postupy jak identifikovat stromové struktury, resp. hierarchie. Při rozhodování, zda a jak využít předností stromové struktury, nám může pomoci simulace chodu systému pomocí uložení a přesouvání objektů smyšlenou adresářovou strukturou. Pro simulaci můžeme využít nástroje pro tvorbu myšlenkových map, nebo si celou hierarchii opravdu vytvořit jako adresářovou strukturu ve filesystému svého PC.

Další možným pohledem na stromovou strukturu je jako na agregaci objektů. Proto se pokusíme využít diagram tříd UML [36] a nalézt vhodné agregační objekty.

Pokud se nám nepodaří identifikovat stromovou strukturu ani agregační objekty, máme poměrně silný argument nevyužívat pro vývoj analyzovaného projektu GA. Modelování objektů umístěných ve stromové hierarchii je jeden s pilířů, na kterých je GA postaven.

8.2.1.1.2 Podpora workflow

Pro každý business objekt hledejme možné stavy a pravidla pro přechod mezi stavy. Pro projekci

možných kroků objektu využijeme stavový diagram UML. Při modelování podmínek pro přechod mezi stavy, resp. kroky, vždy zvažme, zda nelze přechod provést v rámci některé z implicitních fází GA. Tedy například zda podmínku pro přechod mezi stavy nelze nasimulovat změnou stromové hierarchie, tedy přesunem objektu pod agregační objekt, nebo přesunu některého jeho potomka v rámci stromové hierarchie.

Podpora workflow je druhý ze dvou pilířů GA, proto i zde platí, že pokud se nám nepodaří identifikovat workflow objektů, pak zvláště pokud jsme neuspěli ani u hledání stromové struktury, zkoušejme modelovat znovu, nebo se porozhlédněme po jiné, pro daný problém vhodnější technologii.

8.2.1.1.3 Podpora verzování

V rámci analýzy si označíme objekty, pro které chceme zavést verzování.

8.2.1.1.4 Aplikační logika je prováděna v databázi

Při návrhu vyšších vrstev aplikace, klientů, kteří se budou připojovat přes Garuda API, se vyvarujme implementaci aplikační a stavové logiky, kterou lze modelovat uvnitř GA. Tím si zachováme možnost využívat implementovanou aplikační logiku nezávisle, z více různých klientských aplikací.

8.2.1.2 Reflexe nevýhod vhledem k tvorbě metodiky

V rámci metodiky nepodceňujme nevýhody GA. Pokud se nevýhody neslučují s vývojem daného projektu, zvolme raději jinou technologii.

8.2.1.2.1 Nepřenositelnost

Před tím, než učiníme rozhodnutí využít GA, je třeba ověřit, že zákazník souhlasí s využitím open source databáze. Rozhodnutí pracovat s PostgreSQL již nelze v průběhu vývoje vzít zpět!

8.2.1.2.2 Obtížná optimalizace

Ve fázi analýzy je třeba vytvořit prototyp, vytypovat nejnáročnější operace a provést zátěžové testy v souladu s plánovanou zátěží systému.

8.2.1.2.3 Vlastní definiční jazyk GDL

Na prvních projektech je třeba počítat s režií na zaškolení vývojového týmu.

8.2.1.2.4 Nekompletní dokumentace

Vývojový tým by si měl v průběhu učení se práce s frameworkem doplňovat dokumentaci o zjištěné znalosti. Práce investovaná do zlepšování dokumentace se vrátí při zaškolování dalších vývojářů.

8.2.1.3 Příležitosti

I v případě příležitostí zhodnotíme, zda lze do metodiky zakomponovat jejich potenciál.

8.2.1.3.1 Vhodně vytvořená metodika vývoje

Metodiku můžeme neustále zlepšovat. Po každém projektu provedeme revizi metodiky a obohatíme ji o nabyté zkušenosti.

8.2.1.3.2 Prostor pro další rozvoj

Pokud dochází k dalšímu rozvoji frameworku, motivujme své vývojáře k tomu, aby se podělili s vývojáři GA o zkušenosti nabyté při tvorbě projektů. Čas obětovaný komunikací s nimi se nám může vrátit při práci na dalším projektu. Tento bod samozřejmě závisí na další budoucnosti frameworku GA. Pokud dojde ke zveřejnění kódu například pod GNU, MIT, nebo jinou „otevřenou“ licencí, může to být motivací pro vývojáře projektů nad GA podílet se na jejím dalším rozvoji.

8.2.1.3.3 Otevření kódu

Otevření kódu jsme již diskutovali v předchozím bodě. Každopádně v rámci metodiky rozhodnutí otevřít zdrojové kódy ovlivnit rozumně nemůžeme. V případě, že stojíme na straně majitele autorských práv a rozhodneme se pro zveřejnění kódu, pak můžeme zvýšit potenciál zveřejnění kódu pomocí zveřejňování výsledků naší práce, snahy o zapojení open source komunity do dalšího vývoje GA. Toto však nebudeme zahrnovat do metodiky vývoje informačního systému.

8.2.1.4 Vliv příležitostí GA na metodiku

Příležitosti GA uvedené ve SWOT analýze souvisí s tvorbou metodiky vývoje pouze okrajově. Jedná se o externí vlivy, které při vývoji nad GA nelze příliš zohlednit. Za zmínku stojí snad jen bod týkající se metodiky vývoje nad GA. Jakákoliv zpětná vazba mezi vývojovým týmem GA a programátory projektů nad GA je přínosná pro další rozvoji frameworku. Stejně tak metodika vývoje by měla být po každém projektu doplněna v závislosti na potřebách z praxe.

8.2.1.5 Minimalizace Hrozeb SWOT

8.2.1.5.1 Snížení výkonu po naplnění reálnými daty

Jak jsme již zmínili v bodu věnovanému obtížné optimalizaci, tomuto riziku je možné předcházet pomocí vhodně zvolených zátěžových testů.

8.2.1.5.2 Ukončení vývoje GA

Této hrozbě stěží můžeme čelit v rámci metodiky vývoje. Nicméně pokud Ve fázi rozhodování jakou využít technologii

8.2.1.6 Dílčí výsledky

U každého bodu SWOT analýzy jsme provedli diskuzi možného využití pro tvorbu metodiky. Tyto výsledky budeme dále aplikovat na existující metodiky vývoje OOP aplikací.

8.2.2 Rozhodnutí využít GA

Dříve, než začneme sestavovat metodiku vývoje pomocí GA, zaměřme se na problém, kdy je vhodné framework pro vývoj projektu využít a kdy nikoliv. V průběhu SWOT analýzy jsme odhalili množství rizik. Zvážení kdy a k čemu GA využít je tedy důležitý krok, který může ušetřit mnoho práce strávené vývojem nad chybně zvolenou architekturou.

8.2.2.1 V jaké fázi projektu se rozhodnout pro GA

Pokud se budeme řídit pravidlem, že výběr technologie probíhá až na základě výstupu analýzy[37], metodika vývoje nad frameworkem GA by měla začít ve fázi, kdy už má analytik formalizované požadavky od klienta. Požadavky mohou být formalizovány pomocí metodiky UML, využitím diagramu případů užití doplněných o slovní popis scénářů užití.

Pokud již ve fázi analýzy zvažujeme využití GA a navrhujeme systém, který sleduje životní cyklus dokumentů v podniku, měl by analytik zmapovat stávající stav. Tedy zjistit, jakým způsobem jsou zaměstnanci zvyklí pracovat se souborovým systémem, zda například nevyužívají nějakou konvenci pro ukládání dokumentů na síťový disk, jak jsou se stávajícím stavem spokojeni. Pokud totiž navrhneme nový systém tak, aby kopíroval zaběhlý způsob práce, uživatelé nový systém lépe přijmou a bude se jim s ním lépe pracovat.

8.2.2.2 V jakém případě se rozhodnout pro použití GA

Pokud zvažujeme použití GA, měli bychom projít jednotlivé body SWOT a zhodnotit si, která kritéria na daném projektu převažují. Využití frameworku má smysl zejména pokud jsme schopni maximálně využít výhod frameworku, tedy pokud chceme modelovat workflow objektů, nejlépe pak pomocí jejich přemisťování v rámci stromové hierarchie. Důležité také je, aby podmínky, které vyhodnocují jednotlivé události řídící workflow, bylo možné jednoduše formulovat v jazyce GDL.

Příklad kdy logika nepůjde snadno vyhodnotit pomocí GDL. Pokud budeme chtít před odesláním faktury ověřit přes webové rozhraní, že firma není uvedena na online seznamu dlužníků. Podobnou logiku lze jednoduše a rychle naprogramovat na aplikační vrstvě například v PHP, v GDL to bude o něco obtížnější, pro podobné případy musíme napsat vnořenou proceduru např. v jazyce Pearl a tu zavolat v příslušné definici workflow. Pokud se v analýze systému odhalí příliš mnoho podobných podmínek pro běh workflow, je rozumné zvážit nějaký alternativní framework, který implementuje workflow na aplikační úrovni.

Pro posouzení podmínek použití GA nám poslouží logický návrh objektové hierarchie.

8.2.3 Konceptuální modelování objektové hierarchie GA

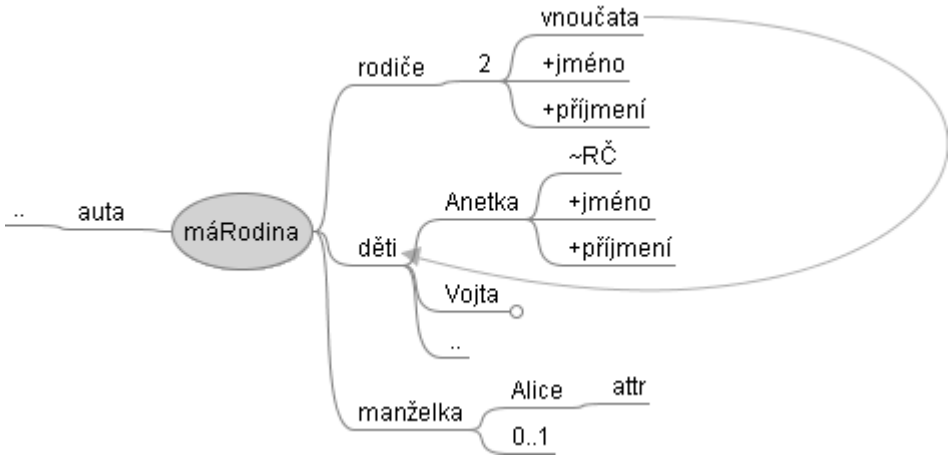
Jak jsme uvedli u modelování schéma relační databáze, konceptuální datový model popisuje data v abstraktní úrovni nezávisle na jejich fyzickém uložení. Nyní navrhneme způsob konceptuálno modelování objektové hierarchie GA pomocí tzv .myšlenkových map.

Pro tvorbu myšlenkových map lze použít různé druhy software, např. Microsoft Office 2003 Visio (kategorie Debata > diagram brainstormingu). Pro účely této práce jsme mapy modelovali za pomocí nástroje Freemind [38].

Připomeňme, že v GA může být v rámci jedné aplikace využito více izolovaných stromů,

identifikovaných různými family_id. V kořeni myšlenkové mapy tedy bude vždy uveden název rodiny objektů modelované hierarchie.

Jednotlivé větve reprezentují vztah agregace objektů a atributů. Uzel, který obsahuje číslici, dvě tečky (..), nebo číselný rozsah (1..3), případně (1..) vyznačuje kardinalitu agregace. Uzel uvozený znamínkem pak označuje atribut, v případě, že název obsahuje složené závorky, tak metodu. Pokud chceme v diagramu vyznačit linky, použijeme spojení uzlů za pomocí šipky. Na obrázku Obrázek 8.1: na straně 53 je zobrazena hierarchie rodina.



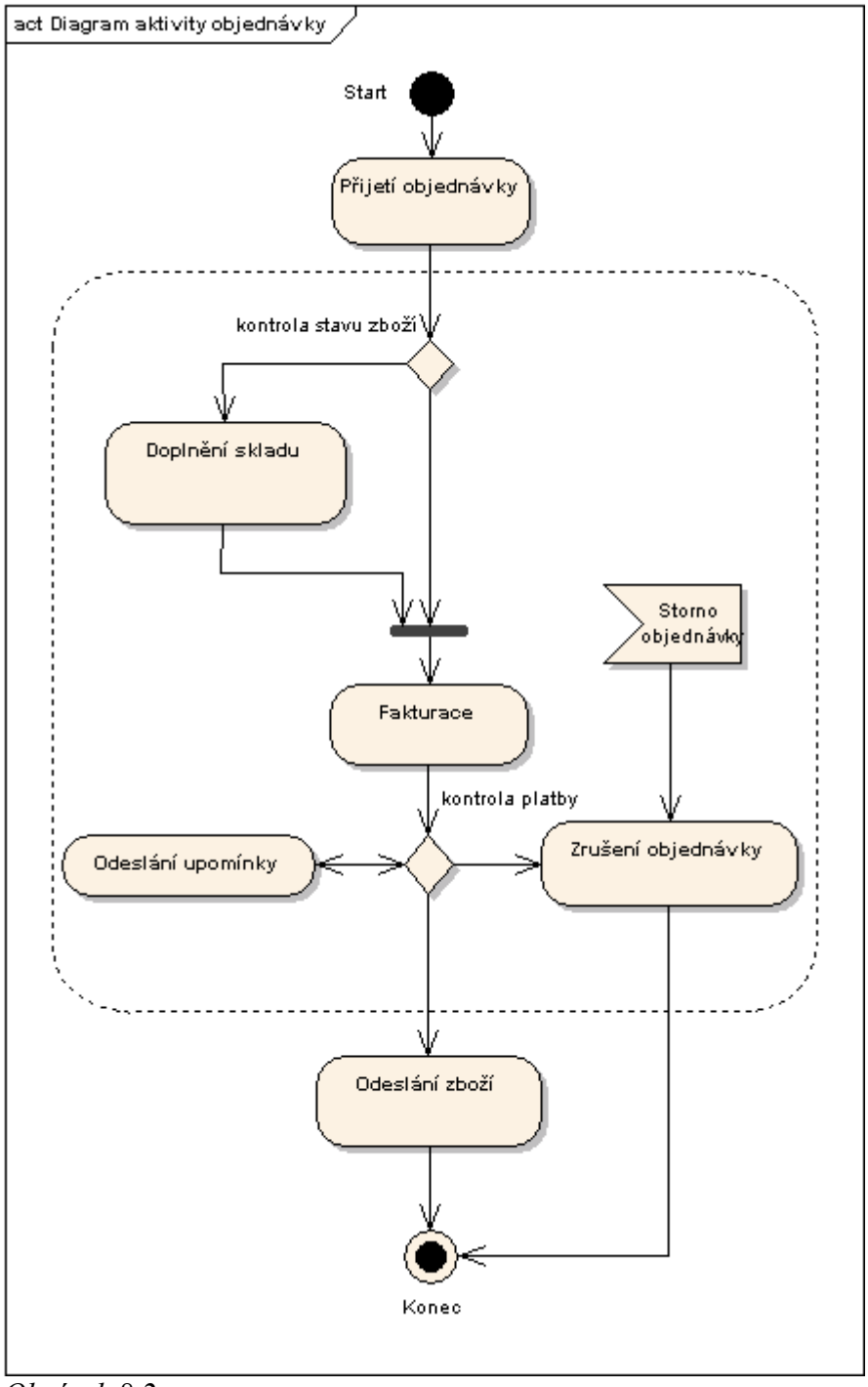
Obrázek 8.1:

Popis obrázku: Rodina „rodina“ agreguje právě dva „rodiče“, libovolný počet „dětí“ a libovolný počet „aut“. Z obrázku je vidět, že pro většinu objektů máme agregační objekty. Navíc vnoučata jsou tvořena odkazem na děti a vidíme, že dítě reprezentované Anetkou má tři atributy. Protected RČ a public jméno a příjmení.

Obrázek 8.1 ilustruje možnosti projekce pomocí myšlenkové mapy. Tento způsob se hodí pro navrhování stromu objektů. V další části se zaměříme na způsoby projekce, které nám umožní zachytit workflow jednotlivých objektů.

8.2.4 Modelování workflow

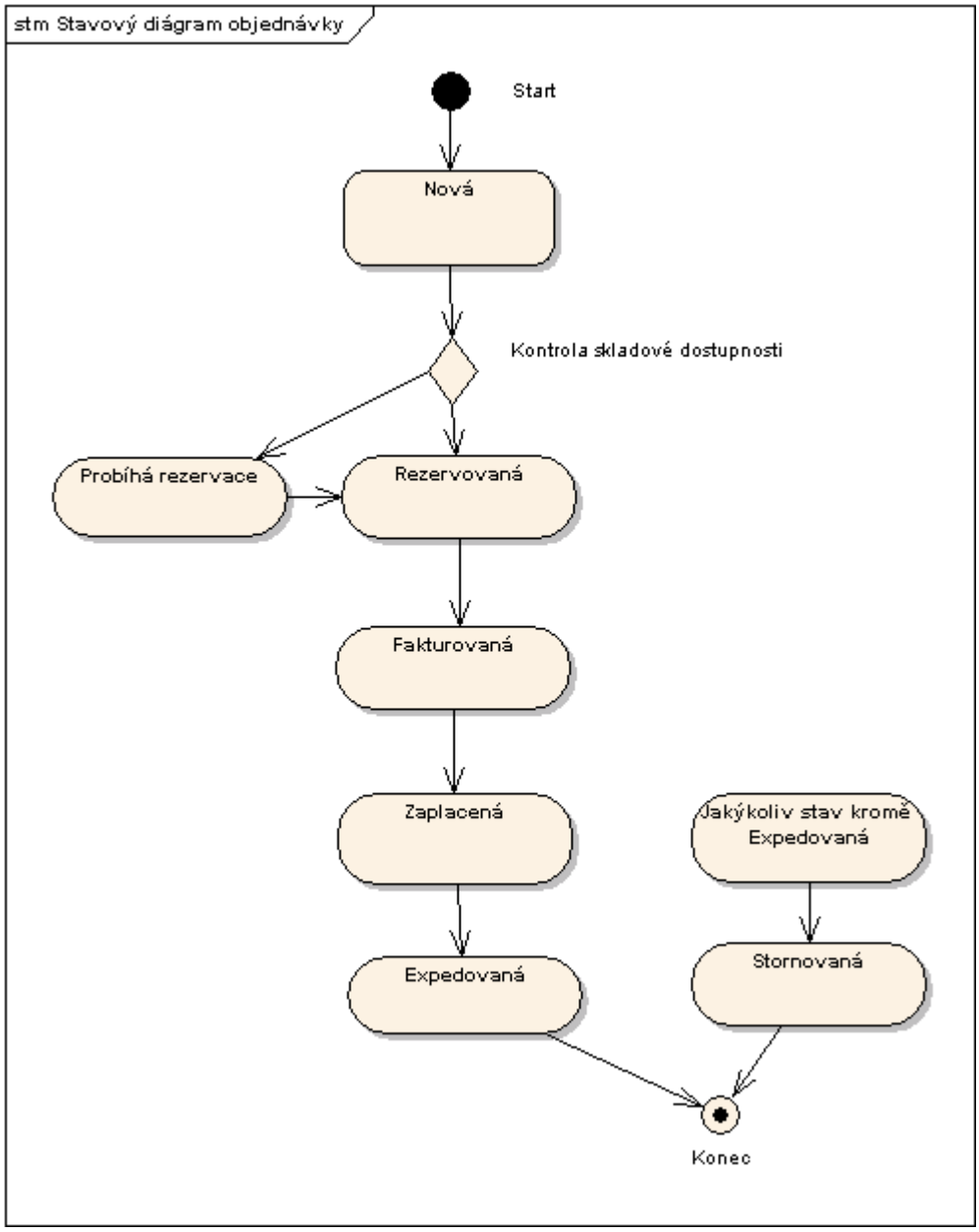
Pro modelování workflow využijeme dva způsoby projekce jazyka UML: Diagram aktivity a stavový diagram. Diagram aktivity, někdy také překládaný do češtiny jako diagram činností, využijeme pro shromáždění procesů, které objekt za svůj životní cyklus absolvuje. Na jeho základě pak modelujeme stavový diagram, který identifikujeme jednotlivé stavy – kroky workflow zkoumaného objektu. Příkladem mohou být následující dva diagramy, které modelují workflow objednávky v informačním systému (diagramy byly vytvořeny pomocí Case nástroje Entreprice Architect):



Obrázek 8.2:

Obrázek 8.2 popisuje základní činnosti s objednávkou. Systém přijímá objednávku, dochází ke kontrole stavu zboží, v případě nedostatku zboží dochází k doplnění skladu. V případě, že je zboží dostatek, následuje fakturace. Po fakturaci kontrola platby. Kontrola způsobuje odesílání upomínek, za určitých okolností pak zrušení objednávky. V případě úspěšné kontroly platby dochází k odeslání zboží a životní cyklus objednávky končí. Navíc je tečkovanou čárou zobrazena oblast interakce,

která obsahuje událost storno objednávky. Tedy kdykoliv po přijetí objednávky a před odesláním zboží je možné objednávku stornovat, což způsobí zrušení objednávky a taktéž konec jejího životního cyklu. Na základě tohoto diagramu je pak možné pojmenovat jednotlivé stavy objednávky a modelovat přechody mezi nimi pomocí stavového diagramu:



Obrázek 8.3:

Dodejme, že stavový diagram může zároveň obsahovat podmínky pro přechod do dalšího stavu.

8.2.4.1 Definice workflow GA

Oba diagramy spolu se scénáři případů užití využijeme při implementaci jednotlivých kroků v definici workflow jednotlivých business objektů GA.

Názvy stavů ze stavového diagramu nám budou sloužit jako názvy kroků objektu v definici jeho workflow. Z diagramu aktivit a scénářů užití pak potřebujeme identifikovat všechny aktéry a akce, které mají provést pro přechod objektu do dalších možných stavů. Z pohledu workflow GA hledáme události, které musí být vyvolány pro vyvolání workflow a přechod mezi jednotlivými stavy.

8.2.4.2 Scénář změny události

Popis jednotlivých událostí nazvěme scénář události a pro každou událost budeme vyžadovat popis obsahující následující strukturu údajů:

- název výchozího stavu
- aktér – aktérem může být uživatel resp. uživatelská role, výjimečně se může jednat o systém samotný, například plánovač událostí v systému, v případě periodické události
- způsob vyvolání - některá z následujících možností
  - zavolání explicitní události - například zmáčknutí tlačítka uživatelského rozhraní
  - změnu hierarchie objektů GA -například přiřazení platby objednávce – tedy některá z implicitních fází
- popis změn v hierarchii objektů GA
- název dalšího stavu

Seznam událostí je kompletní, pokud je na jejich základě možné procházet všemi hranami Stavového diagramu ve směru šipek.

Při sepisování událostí pravděpodobně dojde i k rozšíření stavového diagramu o patologické stavy, tedy o případy, kdy je vyvolána akce, není možný přechod do dalšího stavu a uživatel systému, který akci vyvolal, ji nemůže vyřešit v rámci původního stavu. Příkladem může být situace, kdy systémový robot provádí párování plateb s objednávkami a některé platby není možné spárovat.

V případě interakce více objektů a aktérů můžeme využít dalších možností jazyka UML a sestavit například diagram interakce. Vytvoření každého dalšího pohledu na systém může být přínosem při modelování workflow. Na druhou stranu tvorba diagramu má smysl jen pokud nejsme schopni z dosavadních výsledků analýzy sestavit kompletní scénáře změny událostí.

Kompletní scénáře událostí je vhodné podat k oponentuře ze strany programátora, který zná možnosti GDL, aby zhodnotil možnost jejich implementace v rámci GDL.

Na základě scénářů je již možné implementovat workflow objektů GA. Při implementaci navíc můžeme využít dědičnosti workflow, tedy pokud máme více objektů se stejným chováním, můžeme vytvořit společného předka, u kterého budeme workflow implementovat. Díky vícenásobné dědičnosti můžeme definovat společného předka čistě kvůli workflow bez nutnosti deklarovat atributy a metody příslušné třídy.

8.2.4.3 Naposledy zvažme využití GA

Jsme v situaci, kdy máme sepsané scénáře událostí a můžeme začít s implementací projektu. V tuto chvíli je rozumné naposledy zvážit využití GA, zvážit časovou náročnost vývoje pomocí GA, pří-

padně vytipovat nejnáročnější operace při přístupu k datům a naprogramovat pro vytipované části prototyp, který podstoupíme zátěžovým testům.

8.3 Shrnutí popsané metodiky vývoje pro GA

Předchozí tvorbu metodiky shrneme do několika bodů.

8.3.1 Bod 1. Analýza požadavků

Analýza požadavků nevyžaduje specifický postup, výhodné pro případné nasazení GA může být zmapování stávajících postupů ve firmě pro práci s elektronickými dokumenty. Přesto pro analýzu požadavků doporučujeme modelování případů užití spolu s textovým popisem případů užití.

8.3.2 Bod 2. Zvážení využití GA

Popsali jsme podle jakých kritérií na základě dosavadního popisu systému zvážit využití GA.

8.3.3 Bod 3 Konceptuální modelování objektového modelu GA

Navrhli jsme metodu pro přehledné modelování hierarchie objektů za využití grafické notace pro myšlenkové mapy.

8.3.4 Bod 4 Návrh workflow

Ukázali jsme příklad využití prostředků UML modelování pro přípravu podkladů scénářů událostí. Navrhli jsme popis scénáře události pro sestavování podkladů pro tvorbu workflow a validaci dosavadního návrhu.

8.3.5 Bod 5 Konečné rozhodnutí využít framework GA

Na základě návrhu workflow pak doporučujeme znovu zvážit využití frameworku pro daný projekt.

8.3.6 Další postup při vývoji

Další postup při vývoji se neliší od vývoje ostatních projektů. Při zavádění nové technologie je naopak výhodné využít na zbylé části systému dosavadních zvyklostí a technologií, aby režie na školení vývojového týmu a zavádění změn nepřevážila nad přínosy nově použité technologie. Navíc můžeme po ukončení projektu lépe zhodnotit přínos nově využité technologie.

Pro použití GA je třeba zaškolit programátory. Školení programátorů by mělo proběhnout před konečným rozhodnutím využít framework GA tak, aby mohli na základě scénářů změn událostí zhodnotit náročnost implementace pomoci GA.

Základní postup vývoje může začít následujícími body:

1. realizace objektové hierarchie
2. realizace workflow

3. naplnění testovacími daty
4. tvorba GUI (grafické uživatelské rozhraní) – jeho tvorbu lze dále rozdělit na
    1. plnohodnotné GUI
    2. simulace souborového systému

Další postup se už odvíjí od charakteru příslušného projektu a zvyklostí vývojového týmu.

9 Závěr

V práci jsme postupovali podle pokynů k jejímu vypracování, které byly součástí zadání. Postupně jsme popsali vlastnosti a provedli SWOT analýzy RSŘBD, OOSŘBD, ORM systémů a frameworku GA. GA jsme se věnovali podrobněji, abychom pro ni na základě SWOT analýzy navrhli metodiku vývoje.

9.1 Posouzení výsledků SWOT analýz

Nyní se pokusíme shrnout výsledky jednotlivých analýz, provedených v předchozích kapitolách a navrhnout jejich možné využití při výběru prostředků pro vývoj informačních systémů.

9.1.1 RSŘBD

U relačních databází převládají pozitiva, mají tradici, jsou dostupné, výkonné, široce rozšířené a zdokumentované. Důvodem, proč vůbec zvažovat použití jiné technologie, jsou rizika a režijní náklady spojené s jejich propojením s objektově orientovaným návrhem aplikace, případně potřeba ukládání specifických dat.

9.1.2 OOSŘBD

Hlavní výhoda objektově orientované databáze spočívá v její podobnosti aplikační vrstvě a tedy možným pozitivům z hlediska rychlosti implementace a čitelnosti kódu. Přestože jednotliví výrobci deklarují rychlost přístupu k datům srovnatelnou, v některých ohledech dokonce vyšší, převažují zde obavy z možných rizik spojených s rychlostí. Ve SWOT analýze této technologie převažuje potenciál. V první řadě spojený s větší mírou standardizace, která by při sjednocení rozhraní pro ORM systémy a OOSŘBD umožňovala migraci mezi jednotlivými přístupy. Dalším neméně podstatným bodem je potenciál rychlého rozvoje pamětí s náhodným přístupem. V takovém případě ztrácí relační databáze určitý náskok získaný optimalizací na sekvenční přístup k datům uloženým na magnetických pevných discích.

9.1.3 ORM systémy

Dalším předmětem zkoumání byly ORM systémy, které vznikly z přirozené potřeby pro koexistenci relačního návrhu databáze s praktikováním OOP na aplikační vrstvě. Důvody pro použití ORM systému jsou podobné jako u OOSŘBD. Rozhraní některých systémů se dokonce vyvinula na základě standardu, který vznikal pro čistě objektové databáze. Jejich využití je hojně rozšířené. V zásadě platí, že pozitiva jsou vyvážena negativy a naopak. Konkrétně, čím více využijeme automatických prostředků ORM, tím více hrozeb a nevýhod zmíněných ve SWOT analýze nás může postihnout. Naopak, čím více budeme mapování sami ovlivňovat, tím méně práce nám ušetří. Při použití ORM je vždy třeba počítat s určitou režií pro samotný překlad, práci s metadaty a neefektivitě způsobené generováním dotazů. Jeho využití považujeme za nejvýhodnější, pokud chceme rychle vytvořit prototyp, který pak budeme dále zdokonalovat. Rozdíly mezi jednotlivými ORM jsou velké, proto je důležité při aplikaci SWOT analýzy vždy posuzovat konkrétní framework a zhodno-

tit, které body jsou pro něj relevantní.

9.1.4 Garuda

Framework GA nepatří na úroveň zbylých analýz. Jedná se o účelový framework, který všechny tři předchozí přístupy spojuje a využívá pro účely modelování workflow.

Z hlediska aplikační vrstvy je ho možné srovnávat s OOSŘBD. Je naprogramovaný prostředky relační, respektive objektově relační databáze a zároveň zprostředkovává objektově relační mapování. Do určité míry tedy kombinuje výhody i nevýhody předchozích tří SWOT analýz.

Vzhledem k tomu, že se jedná o původní řešení a zdrojem pro posouzení nám byly pouze vlastní zkušenosti, nepříliš obsáhlá dokumentace a zdrojové kódy frameworku, nemůže si jeho SWOT analýza klást velké nároky na objektivitu. Přesto jsme se o objektivitu pokusili, abychom její výsledky mohli v práci dále zužitkovat.

9.1.5 Navržení metodiky vývoje

V návaznosti na popis GA a jeho SWOT analýzu jsme navrhli metodiku vývoje nad tímto frameworkem. Při návrhu jsme nejprve zhodnotili jednotlivé body analýzy a pro každý z nich se pokusili nalézt možné uplatnění při návrhu metodiky. Kvůli specifickému zaměření frameworku, jsme při tvorbě metodiky kladly velký důraz na posouzení, kdy je vhodné framework využít. Metodika vývoje informačního systému je široký pojem, proto jsme se soustředili zejména na fázi analýzy, která je pro tvorbu dobrého informačního systému klíčová a bohužel v mnoha případech opomíjená.

9.1.6 Kde by šlo na práci navázat

Pokud bychom chtěli navázat na porovnávání jednotlivých přístupů, mohli bychom například zvolit jiné více exaktní metodiky.

Návrh metodiky by taktéž šlo dále rozvíjet a doplňovat funkčními příklady. Jak bylo v práci zmíněno, pro lepší zdokumentování systému by pomohlo vytvoření vzorové aplikace, která by demonstrovala možnosti, které GA nabízí.

Stejně tak bychom mohli hodnotit další SŘBD, jakými jsou například XML databáze, případně srovnat framework GA s jinými frameworky zaměřené na práci s workflow.

Doufáme, že tato práce může čtenářům posloužit jako zdroj relevantních argumentů, které můžou uplatňovat pro vlastní strategické rozhodování při hledání perzistentního úložiště pro OOP aplikace. Stejně tak může sloužit jako inspirace a návod pro práci s frameworkem Garuda, u kterého věříme, že bude v dohledné době zveřejněn pod některou volnou licencí.

10 Prameny

10.1 Seznam použité literatury a online zdrojů

[1] Thaddeus Mallya- *Základy strategického řízení a rozhodování*, 2007

[2] Wikipedia, *SWOT analysis*, Srpen,2008, http://en.wikipedia.org/wiki/SWOT_analysis

[3] , *Příklad SWOT - Koncepce informačního systému výzkumu a vývoje*, 2002 http://www.vyzkum.cz/FrontClanek.aspx?idsekce=8549

[4] - *Homepage of workflow management coalition*, http://www.wfmc.org/

[5] Tomáš Skopal, *MFF UK Slidy k přednášce DBI025* http://siret.ms.mff.cuni.cz/skopal/DBI025.htm

[6] Pokorný J., Halaška I.: *Databázové systémy: Vybrané kapitoly a cvičení*. , 1998

[7] Codd, E.F.: *A Relational Model of Data for Large Shared Data Banks*. , 1970

[8] Terry Halpin- *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*, 2001

[9] Robert Sheldon- *SQL začínáme programovat*, 2005

[10] Wikipedia, *First normal form*, , http://en.wikipedia.org/wiki/First_normal_form

[11] Chris Date: *Date on Databases*. Apress L. P. , 2006

[12] Gavin Powell- *Beginning Database Design*, 2005

[13] Sparx Systems, *Enterprice Architect - UML tutorial*, http://www.sparxsystems.com.au/uml-tutorial.html

[14] Hana Kozelková, Bakalářská práce Editor ER diagramů s podporou převodu do relčního modelu, 2006

[15] , *Encapsulating Database Access: An Agile "Best" Practice*, http://www.agiledata.org/essays/implementationStrategies.html

[16] Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional , 2002

[17] - *Object-Oriented Database systems: A Survey*, 2003, http://www.cse.ucsc.edu/~lcx/courses/cmps277/cmps277-project.pdf

[18] Robert J. Muller: *Database Design for Smarties: Using UML for Data Modeling*. Morgan Kaufmann , 1999

[19] Daniel Myers, On the Use of NAND Flash Memory inHigh-Performance Relational Databases, 2008

[20] Scott W. Ambler, Larry L. Constantine: *The Unified Process Construction Phase: Best Practices in Implementing the UP*. Focal Press , 2000

[21] Bruce Eckel, Chuck Allison: *Thinking in Java 3rd Edition*. Prentice Hall , 2003

[22] - *Object Database Management Systems resources portal*, , tp://www.odbms.org/

[23] OMG www.omg.org- *Next-Generation Object Database Standardization* , 2007

[24] Kazimierz Subieta- *Theory and Construction of Object-Oriented Query Languages.*, 2004

[25] Gheorghe SABĂU- *Comparison of RDBMS, OODBMS and ORDBMS*,Revista Informatica Economică, nr. 4 (44)/2007, 2007

[26] Elisa Bertino, Giovanna Guerrini- *Object-Oriented Databases*,Wiley Encyclopedia of Computer Science and Engineering, 2008
http://mrw.interscience.wiley.com/emrw/9780470050118/ecse/article/ecse279/current/pdf

[27] Martin Cajthaml, Testování objektové a rela£ní geodatabáze, 2008

[28] ROBERTO V. ZICARI, Mike Card, *ODBMS INDUSTRY WATCH -Object Database Systems: Quo vadis?*, 2008, http://www.odbms.org/blog/2008_05_01_archive.html

[29] , *Homepage of Hibernate ORM framework*, http://www.hibernate.org/

[30] Cake Software Foundation, *CakePHP - The Cookbook*, http://book.cakephp.org/

[31] The Doctrine Project , *Doctrine - PHP Object Relational Mapper*, http://www.doctrine-project.org/

[32] Scott W. Ambler, *Mapping Objects to Relational Databases: O/R Mapping In Detail*, , http://www.agiledata.org/essays/mappingObjects.html

[33] , *Homepage databáze PosetgreSQL*, http://www.postgresql.org/

[34] Paul Graham, *Great Hackers*, 2004, http://www.paulgraham.com/gh.html

[35] Jim Arlow,Ila Neustadt: *UML and theUnifiedProcess - Practical object-oriented analysisand design*. Pearson Education Limited , 2002

[36] Jim Arlow, Ila Neustadt: *UML 2 a unifikovaný proces vývoje aplikací*. Computer Press , 2007

[37] Jaroslav Král, Jiří Demner: *Softwarové inženýrství*. Academia , 1991

[38] *Domovská stránka - FreeMind - free mind mapping software*, http://freemind.sourceforge.net/wiki/index.php/Main_Page

10.2 Přílohy

Část práce se věnuje frameworku Garuda zrojové kódy a dokumentace k frameworku byly uvolněny pro účely diplomové práce s laskavým svolením ředitele společnosti ILIKETHIS! s.r.o., terá je majitelem autorských práv ke GA.

[Příloha 1]. Veškeré zdroje týkající se frameworku jsou součástí příloh vypálených na přiloženém kompaktním disku.

[Příloha 1] Ujednání o poskytnutí zdrojových kódů a dokumentace k frameworku Garuda společností ILIKETHIS! s.r.o.

[Příloha 2] Zdrojové kódy frameworku Garuda - /cdrom/GARUDA/src/

[Příloha 3] Dokumentace Garuda API - /cdrom/GARUDA/doc/api/index.html

[Příloha 4] Manual k Frameworku GA - /cdrom/GARUDA/doc/manual/index.html

[Příloha 5] Manual k praci s Workflow - /cdrom/GARUDA/doc/manual-workflow/index.html

[Příloha 6] Obecný úvod do prostředí GF - /cdrom/uvod do prostredi GF.doc - článek, autor ing Pavel Stěhule

[Příloha 7] Přiklady ER modelů a SQL schémat pro vzorový příklad - /cdrom/DMS/ER/

11 Abecední rejstřík

Datový model....23
Extent....**24, 39**
GDL....48
Objektový identifikátor (OID)....**23**
OOSŘBD....7, 12, **22,** 23, 28, 31, 33, 40, 57
RSŘBD....**12,** 14, 18, 57
SWOT....7, **8,** 16, 26, 30, 33, 42, 44, 47, 57
Workflow....9, 13, 36, 41, 44, 47, 50p., 55

12 Seznam použitých zkratek

API	Application programming interface – soubor funkcí, nebo tříd a jejich metod, které tvoří rozhraní užívání nějaké knihovny, frameworku nebo jiné aplikace
JPA	Java persistence interface, standardizované rozhraní pro práci s perzistentními daty, bývá implementováno ORM frameworky pro JAVU
DMS	Systém pro správu dokumentů, z anglického Document Management System
GA	Garuda – objektový framework, samotný název není zkratkou, Garuda je jméno okřídleného stvoření z Indické mytologie
GIS	Grafický informační systém.
SŘBD	Systém (systémy) pro řízení báze dat, v angličtině DBMS Database Management System (Systems)
OOP	Objektově orientované programování (Object oriented programming)
OOP databáze	OOP není běžně používanou zkratkou pro databáze. Do názvu této práce se zkratka dostala nedopatřením. Z důvodu možných procedurálních problémů se změnou schváleného zadání práce jsme překlep v názvu ponechali. Správně mělo být použito celého názvu objektově orientované databáze, nebo zkratky OOSŘBD, případně mezinárodní OODBMS, nebo ODBMS.
OOSŘBD	Objektově orientovaný SŘBD, v některých pramenech se objevuje pouze OSŘBD jako zkratka za objektový SŘBD, používá se ve stejném významu jako OOSŘBD, v anglickém jazyce se používá OODBMS, nebo ODBMS
ORSŘBD	Objektově relační SŘBD, v angličtině ORDBMS
PL/pgSQL	Procedurální jazyk databáze PostgreSQL,PL z anglického procedural language,
RSŘBD	Relační SŘBD, v angličtině RDBMS
SŘBD	Systém (systémy) pro řízení báze dat, v angličtině DBMS Database Management System (Systems)
SQL	Standardizovaný dotazovací jazyk, z anglického Structured Query Language (strukturovaný dotazovací jazyk)
SWOT	Akronym z počátečních písmen anglických slov **s**trengths (výhody,silné stránky), **w**eaknesses (slabiny, nedostatky), **o**pportunities (příležitosti) a **t**hreats (hrozby,nebezpečí).- název analytické metody
UML	Unified Modeling Language - standardizovaný jazyk pro tvorbu analytických diagramů

Tato práce obsahuje 113703 znaků.