UNIVERZITA PARDUBICE

Fakulta Elektrotechniky a Informatiky

Systém komponent s využitím distribuovaného programování

Bc. Zdeněk Bejr

Diplomová práce

2011

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2010/2011**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Zdeněk BEJR**
Osobní číslo:	**I08343**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Systém komponent s využitím distribuovaného programování.**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Zásady pro zpracování (cíl, obsah teoretické a implementační části): Cílem práce je navrhnout framework usnadňující tvorbu webových prezentací. V teoretické části budou popsány komponentové systémy a jejich využívání. Dále budou popsány a porovnány databázové systémy včetně databází objektových. V praktické části se pokuste vytvořit distribuovaný framework s jehož pomocí vytvořte funkční webovou prezentaci.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

PAGE-JONES, M. Základy objektově orientovaného návrhu v UML, Grada Publishing, 2001. 368s ,ISBN 80-247-0210-X ARLOW, J., NEUSTADT, I. UML 2 a unifikovaný proces vývoje aplikací. Computer Press, 2007. 568 s. ISBN 978-80-251-1503-9 MCCONNELL, S. Dokonalý kód. Computer Press, 2006. 896 s. ISBN 80-251-0849-X

Vedoucí diplomové práce: **Ing. Karel Šimerda**
Katedra softwarových technologií

Datum zadání diplomové práce: **27. října 2010**

Termín odevzdání diplomové práce: **20. května 2011**

Univerzita Pardubice Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

doc. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 3. listopadu 2010

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 29. srpna 2011 Zdeněk Bejr

Poděkování

Tímto bych chtěl poděkovat Ing. Karlovi Šimerdovi za odborné vedení a cenné rady, které mi byly poskytnuty během zpracovávání této práce.

Zvláštní poděkování patří také slečně Monice Píšové a mé rodině za vytrvalou podporu a asistenci při dokončování této práce a v průběhu celého studia.

Všem velice děkuji

Anotace

Tématem této diplomové práce je vytvoření webového frameworku na platformě Java Enterprise Edition.

Diplomová práce nejprve hledá vhodnou databázi. Porovnává jednotlivé modely databází, jak se s nimi pracuje a na praktických ukázkách předvádí konkrétní databáze. Dále specifikuje komponentové systémy, které jsou v aplikaci použity.

Praktická část popisuje postup tvorby frameworku a jeho výsledné řešení. Na konkrétním příkladu demonstruje vytváření dalších komponent.

Klíčová slova

Java Enterprise Edition, framework, Java Server Faces, web, komponenty, distribuované programování

Title

The system of components with using of distributed programming

Annotation

The theme of this thesis is to create a web framework for Java Enterprise Edition.

The thesis firstly looks for a suitable database. It compares different models of databases, how to work with them and demonstrates the specific databases on practical examples. Then it describes the component systems that are used in the application.

The practical part deals with the procedure of framework creating and its resulting solution. The concrete example shows the creation of another components.

Keywords

Java Enterprise Edition, framework, Java Server Faces, web, components, distributed programming

Obsah

Typografické konvence

V této práci jsou použity následující typografické konvence. Slouží k lepší orientaci v textu a k jeho snadnějšímu pochopení.

- **Důležité** Důležité texty jsou vysazeny tučně.
- *Názvy* Texty, které označují důležité názvy nebo zkratky jsou napsány kurzívou.
- Program Pokud jde o větší ucelenou část kódu, je ohraničena a podbarvena světle šedou barvou.
- `Program` Krátké texty programů (jako ukázka přímo v textu) jsou vysazeny neproporcionálním písmem.

Předpokládané znalosti čtenáře

Pro plné pochopení praktické části se u čtenáře předpokládá znalost objektově orientovaného programování a jazyku Java. Dále je pak vhodná alespoň základní znalost dotazovacího jazyka SQL a návrhových vzorů. Neznalého čtenáře odkazuji na [1] a [34].

Seznam obrázků

Seznam tabulek

Seznam grafů

Seznam zkratek

- API – *Application Programming Interface*. Rozhraní aplikace.
- BCNF – *Boyce-Coddova Normální Forma*. Používaná při normalizaci.
- COM – *Component Object Model*. Komponentová technologie.
- CORBA – *Common Object Request Broker Architecture*. Komponent. technologie.
- DBMS – *DataBase Management system*. Neboli Systém řízení báze dat.
- DCOM – *Distributed Component Object Model*. Komponentová technologie.
- DDL – *Data Definition Language*. Jazyk pro definici dat.
- DML – *Data Manipulation Language*. Jazyk pro manipulaci s daty.
- EJB – *Enterprise JavaBeans*. Komponentová technologie.
- IDL – *Interface Definiton Language*. Jazyk pro definici rozhraní.
- IIOP – *Internet Inter-Orb Protocol*. Protokol pro ORB.
- JDBC – *Java Database Connectivity*. Jednotné rozhraní pro přístup do databáze.
- JEE – *Java Enterpise Edition*. Platforma jazyka Java.
- JPA – *Java Persistence Api*. Framework pro přístup k databázi.
- JSE – *Java Standard Edition*. Platforma jazyka Java.
- JSP – *Java Server Page*. Jazyk pro definici stránky.
- NF – Normální forma.
- OO – Objektově Orientované
- ODBC – *Open Database Connectivity*. Standardizované API pro přístup k DBMS.
- ODL – *Object Definition Language*. Jazyk pro definici objektů.
- ODM – Objektovy Databázový Model.
- OID – *Object ID*. ID objektu.
- OMG – *Object Management Group*. Standardizační komise.
- ONF – Objektova Normalni Forma.
- OODBMS – Object Oriented DataBase Management System.
- OOP – *Object Oriented Programming*. Objektově orientované programování.
- OOSŘBD – Objektově Orientované Systémy Řízení Báze Dat.
- OQL – *Object Query Language*. Jazyk pro dotazování na objekty.
- ORB – *Object Request Broker*. Stará se o navazování spojení mezi komponenty.
- ORDBMS – Object Relations DataBase Management System.
- RDBMS – Relational DataBase Management System.
- RDM – Relační Databázový Model.
- RPC – *Remote Procedure Calling*. Vzdálené volání metod.
- SŘBD – Systém Řízení Báze Dat.
- UML – Unified Modeling Language.

Úvod

V dnešní době se většina programů, které známe jako klasické desktopové, stěhují na Internet. Internet nabývá stále větších rozměrů a schopností. Příkladem budiž Google Docs, kde lze vytvořit textový nebo tabulkový soubor, prezentaci, nakreslit obrázek atd. Existují i servery, kam můžeme nahrávat neomezené množství fotografií, videí a hudby. Tato migrace programů má velké výhody, můžeme mít přístup ke svým dokumentům odkudkoliv, sdílet je s jinými uživateli a zároveň se nemusíme starat o zálohu dat nebo aktualizacemi svých programů. Díky tomuto trendu vznikají neustále nové technologie pro podporu internetového vývoje. Nejrozšířenější programování v PHP je nahrazováno novějšími, výkonnějšími a čistšími technologiemi.

Cílem této práce je vytvořit framework usnadňující tvorbu internetových aplikací v programovacím jazyku Java. Framework by měl být složen z komponent. Většina dnešních aplikací se nepíše úplně od začátku, ale využívají se existující kódy. Drtivá většina stránek obsahuje správu uživatelů, textů a fotogalerií. Proto je výhodné, když nemusíme tyto části neustále programovat. Zároveň je však velice snadná aktualizace těchto komponent.

V teoretické části se práce bude zabývat komponentovými standardy a samotnými komponenty. Popisuje, jak má komponenta vypadat a co má obsahovat. Druhá část teoretické práce se zabývá porovnáním databázových modelů a snaží se nalézt nejvhodnější databázi pro vytvářený framework.

Část I.

Teoretické podklady

Před tvorbou samotného frameworku je potřeba nalézt vhodné technologie, které budou použity a seznámit s nimi čtenáře.

Klíčová je především volba vhodné databáze. Nejedná se jenom o výběr mezi konkrétními databázemi, ale i o volbu modelu databáze. S každým modelem se pracuje odlišně a proto je vhodné zvážit klady a zápory jednotlivých technologií. Tato část obsahuje nejprve obecný popis a ukázku práce s konkrétními modely. Dále porovnává vhodné kandidáty z hlediska dvou nejčastějších operací (ukládání a vyhledávání dat). Výsledky jsou pak zobrazeny v grafech.

Konec této části se věnuje komponentovým systémům. Ty jsou nosnou technologií pro tento projekt. Přestože se nejedná o úplně nejnovější technologii, je na nejvýše vhodné ji zde uvést.

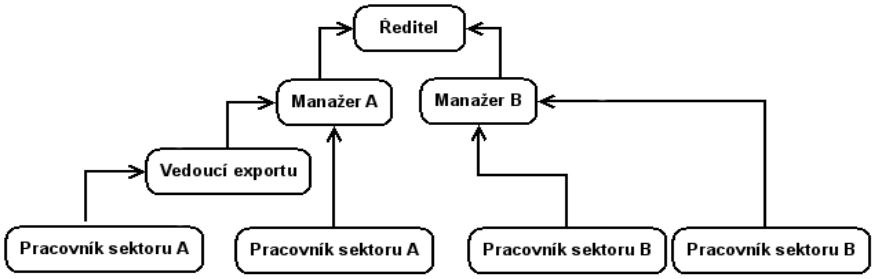
1 Databázové systémy

Databáze (datová základna) je místo, kde se ukládají data. Ta se většinou ukládají strukturovaně a organizovaně. K datům se přistupuje pomocí SŘBD (Systém Řízení Báze Dat), v angličtině označováno jako DBMS (DataBase Management System). SŘBD je tedy softwarové vybavení tvořící rozhraní mezi uloženými daty a aplikací. Pod běžně používaným slovem databáze se myslí jak uložená data, tak i software pracující nad uloženými daty. [9] [10]

Databáze se dají dělit dle různých kritérií. Nás ale bude zajímat především dělení dané modelem databáze. To je architektura, podle které ukládá databázový systém objekty a podle které je vzájemně spojuje. Dnes se používají výhradně databáze relační nebo objektové. Kromě těchto dvou zástupců se podíváme i na další zástupce této kategorie. Databázový svět také prošel vývojem a každá verze ovlivnila svými vlastnostmi a hlavně nedostatky svého následníka.

1.1 Hierarchický model databáze

V tomto modelu jsou data strukturována hierarchicky jako datový typ strom. Jedna z tabulek se nazývá kořen a ostatní jsou na ni navázány. Vztahy mezi jednotlivými tabulkami jsou definovány jako rodič – potomek. Tabulka může mít jednoho nebo několik potomků, ale potomek má jenom jednoho předka. K jednotlivým záznamům se přistupuje od kořene (kořenová tabulka) ve směru hierarchie. Pokud tabulka má dalšího následníka, nazývá se uzel, pokud následníka nemá, jedná se o list. V datech se pohybujeme třemi způsoby, a to: od rodiče k potomkovi (směrem dolů), od potomka k rodiči (směrem nahoru) a k sourozenci (vodorovně). [7]



Obrázek 1: Struktura hierarchického modelu databáze

Na obrázku vidíme strukturu fiktivní firmy, kdy potomek podléhá předkovi. Kořenem je zde ředitel firmy. Ředitel má přímé podřízené a ti mají zase své podřízené. Tímto způsobem se dopracujeme k jednotlivým pracovníkům.

Výhody této databáze jsou, že uživatel získá data velmi rychle, neboť jsou propojena přes přímé spojení. Dále model zaštiťuje fyzické uložení na disk, což znamená, že se programátor nemusí starat o to, jak jsou data uložena. Naopak nevýhodou u tohoto

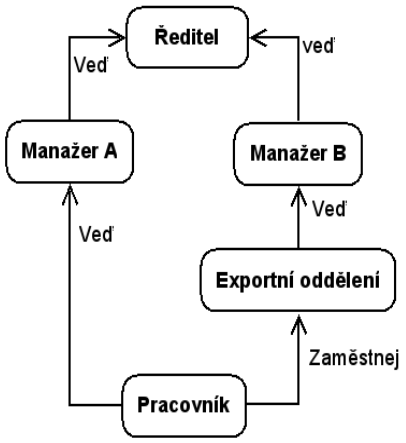
typu databází je možná redundance dat, která vzniká při realizaci vztahu typu M:N. Tento vztah se dá řešit jenom za cenu zdvojení dat. V uvedeném příkladu by redundance vznikla, kdyby pracovník sektoru B byl zároveň vedoucím exportu v sektoru A. Zde by muselo dojít k redundanci dat. Další nevýhodou je přidávání a ubírání prvků, kdy je nutné přepracovat celou strukturu databáze.

Tento model se využívá tam, kde se ukládají data ve tvaru stromové struktury. Příkladem mohou být rodokmeny, skladové systémy atd. Ale i u těchto typů jsou dnes rychlejší novější technologie. Nejznámějším zástupcem této kategorie je systém IMS, který byl vyvinut v 60. letech společnostmi IBM a NAA. IMS byl vyvinut u projektu Apollo, kde sloužil ke skladovému hospodářství. [7]

1.2 Síťový model databáze

Byl vyvinut jako zobecnění modelu hierarchického. Model byl konkrétně specifikován v roce 1971 na konferenci CODASYL skupinou DBTG (Data Base Task Group). Model je složen z uzlů (soubor záznamů) a množinové struktury (vztahy v síťové databázi). Jeden z uzlů je vždy definován jako vlastník a druhý jako prvek. V databázi je celá řada výskytů záznamů každého z deklarovaných typů. Dokonce mohou být i stejné, protože se rozlišují hodnotou databázového klíče, který je automaticky generován, a je tedy jedinečný pro každou položku.

Síťový model zná vztahy typu 1:1 nebo 1:N mezi dvěma typů záznamů. Ostatní typy vztahů se musí předem na konceptuální úrovni rozložit – to se nazývá množina (set). Set je definován pomocí svého vlastníka a členů. Ve schématu je tedy definován typ setu který zahrnuje: přidělené jméno, typ záznamu vlastníka a typ záznamu členu.



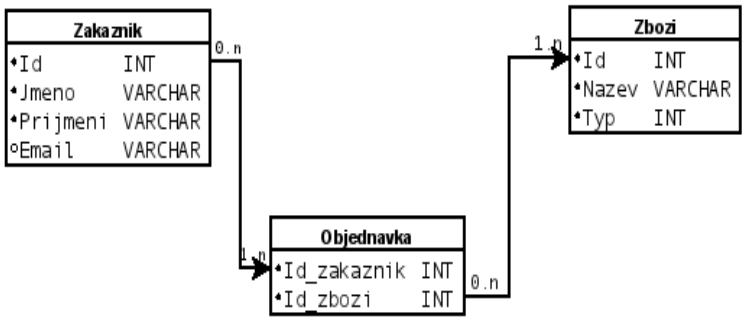
Obrázek 2: Struktura síťového modelu databáze

Vyhledávání nemusí začínat u kořene jako v hierarchickém modelu, ale může začít kdekoli v libovolném uzlu a procházet přidruženými záznamy.

Mezi výhody patří rychlý přístup k požadovaným datům a možnost realizace dat vztahu typu M:N. Mezi hlavní nevýhody patří nutnost znalosti struktury databáze. Při změně většinou dochází k přepracování databázového modelu. Mezi konkrétní zástupce patří například: IDMS, DBMS nebo IMAGE. [7]

1.3 Relační model databáze

Relační model je dnes určitě nejrozšířenějším typem ukládání dat v databázi. V roce 1969 přišel doktor E. F. Codd se svou představou databáze založené na matematickém aparátu relačních množin a predikátové logiky. Databázová relace obsahuje navíc (oproti matematické) schéma relace, kde je uveden název relace, počet sloupců, názvy sloupců a domény (definice tabulky). Doména je vlastně soubor přípustných hodnot pro každý sloupec. Relací ale nemusí být jenom tabulka, ale jakákoliv struktura, která je rozdělena na sloupce a řádky (výsledek dotazu, pohled, ...). Jednotlivý záznam je vlastně n-ticí (řádkem) tabulky. [9]



Obrázek 3: Ukázka relačního modelu

Příkladem mohou být například množiny objednávek, zákazníků a zboží. Z nich se vytvoří kartézský součin, který reprezentuje všechny možné vazby mezi množinami. Za relaci je možné považovat buď celý kartézský součin nebo jeho libovolnou podmnožinu.

Kromě kartézského součinu lze použít i následující základní operace: sjednocení, rozdíl, selekce, projekce a spojení. Pomocí těchto operací lze uskutečnit všechny operace nad daty a ostatní už jsou jenom jejich kombinací.

U výsledků dotazů nelze určit první, druhý nebo n-tý řádek. Jednotlivé řádky v relaci nemají dané pořadí, nemůžeme se na ně tedy odkazovat číslem řádku. K adresování jednotlivých řádků nám pomáhá konstrukce nazvaná primární klíč. Primární klíč je složen z jednoho nebo více atributů, které jednoznačně identifikují daný řádek. Relační model dále zavedl některé důležité pojmy v databázovém světě:

- **Klíče** – Jsou definovány nad jednotlivými sloupci v tabulkách. Pomáhají vylučovat duplicitu a zrychlují vyhledávání.

- **Pohled** – Poskytuje stejnou strukturu jako tabulka. Na rozdíl od ní neobsahuje data ale jenom předpis pro získávání těchto dat z tabulek, nebo jiných pohledů.
- **Trigger** – Mechanizmus určující, co se stane při změně (vložení, smazání) tabulky nebo řádku v tabulce.
- **Procedury a Funkce** – Jsou to určité pojmenované sekvence příkazů, které provedou nad daty nějakou operaci.

Výhodou relačního modelu je především přirozená reprezentace zpracovávaných dat, lehko definované vazby nebo integrita modelu.

Tento model je dnes velmi rozšířený. Až na výjimky ho dnes používají všechny komerční projekty. Mezi nejznámější patří: MySQL, Oracle Database, SQL Server (Microsoft), PostgreSQL, Firebird, DB2 (IBM) a další. [9]

1.4 Normální formy

Před začátkem vývoje aplikace, která využívá relační databázi, se musí nejprve navrhnout samotná struktura této databáze. Návrh je velmi důležitý, protože pozdější úpravy bývají složité, ne-li nemožné. Základním předpokladem je, že by tabulky neměly obsahovat žádná redundantní data. Pokud se vyplňují stále ty samé údaje, něco není v pořádku. V tabulkách by se neměly vyskytovat ani sloupce typu *zbozi_1*, *zbozi_2*, *zbozi_3*. V lepším případě si zákazník objedná jenom jedno zboží a dva sloupce zůstanou nevyužity (znepřehledňují a brzdí databázi). V horším případě si objedná 10 druhů zboží a nastává téměř neřešitelný problém. Tyto všechny nedostatky řeší proces jménem **normalizace**. Obecně platí, že v čím vyšší normální formě je tabulka, tím lépe je navržena. Tyto normy pomáhají k vyššímu výkonu databáze, ke snadnější údržbě databáze a v neposlední řadě k její přehlednosti. [10] [12]

1.4.1 Nultá Normální Forma (0. NF)

Odpovídá nenormalizovanému modelu. Tabulka obsahuje alespoň jeden atribut, který obsahuje více než jednu hodnotu. Jako příklad můžeme uvést sloupec *Jméno* z tabulky 1, který obsahuje jak příjmení, tak i jméno osoby. Problém u takto nenormalizované tabulky nastane v případě, kdy budeme chtít vyhledat záznamy podle konkrétního příjmení.

RC	Jméno	Funkce	Plat	Daň
8212143112	Zdeněk Bejr	Programátor	30000	6000
8011062345	Jiří Novák	Analytik	35000	7000
8612013208	Jakub Malý	Tester	25000	5000

Tabulka 1: Ukázka nenormalizované tabulky (0. NF)

1.4.2 První Normální Forma (1. NF)

Tabulka je v první normální formě, pokud všechny její atributy jsou dále nedělitelné (atomické). Jeden atribut tedy obsahuje jeden typ dat. V uvedeném příkladu by to znamenalo rozložení atributu *Jméno* na *Jméno* a *Příjmení*.

RC	Jméno	Příjmení	Funkce	Plat	Daň
8212143112	Zdeněk	Bejr	Programátor	30000	6000
8011062345	Jiří	Novák	Analytik	35000	7000
8612013208	Jakub	Malý	Tester	25000	5000

Tabulka 2: Všechny atributy jsou atomické (1. NF)

Někdy může nastat výjimka, pokud například používáme název ulice a popisné číslo domu. Pokud víme, že aplikace nebude vyžadovat jenom názvy ulic, spojí se tyto atributy v jeden textový řetězec. Odpadne tím zbytečné načítání dvou sloupců a jejich spojování. Je ale jenom málo případů, kde se vyplatí „obejít" 1. NF.

1.4.3 Druhá Normální Forma (2. NF)

Tabulka je ve druhé normální formě, jestliže je v 1. NF a zároveň platí, že každý atribut, který není součástí primárního klíče, je na primárním klíči zcela závislý. Jinými slovy to znamená, že se v tabulce nesmí objevit řádek, který by byl závislý jen na části primárního klíče. U tabulek, které mají za primární klíč jenom jeden atribut, je tato norma splněna automaticky. V předchozí tabulce byl primární klíč tvořen sloupci *RC* a *Funkce*. Atribut *Příjmení* ale na funkci nezáleží, proto musí dojít k dekompozici tabulky na dvě následující.

RC	Jméno	Příjmení	ID Funkce
8212143112	Zdeněk	Bejr	4
8011062345	Jiří	Novák	9
8612013208	Jakub	Malý	1

ID	Funkce	Plat	Daň
4	Programátor	30000	6000
9	Analytik	35000	7000
1	Tester	25000	5000

Tabulka 3: Dekompozitované tabulky splňující 2. NF

1.4.4 Třetí Normální Forma (3. NF)

Tabulka je ve třetí normální formě, splňuje-li obě předešlé a zároveň platí, že žádný z atributů není tranzitivně závislý na primárním klíči. Čili jsou-li neklíčové atributy na sobě nezávislé. V našem příkladu je *daň* závislá na *platu*. Proto se musí i pro tyto údaje vytvořit nová tabulka.

RC	Jméno	Příjmení	ID Funkce
8212143112	Zdeněk	Bejr	4
8011062345	Jiří	Novák	9
8612013208	Jakub	Malý	1

ID	Funkce	ID Mzda
4	Programátor	3
9	Analytik	8
1	Tester	6

ID	Mzda	Daň
3	30000	6000
8	35000	7000
6	25000	5000

Tabulka 4: Tabulky splňující 3. NF

1.4.5 Boyce-Coddova Normální Forma (BCNF)

Tato normální forma je pokládána za variaci třetí normální formy. Většinou je, pokud splníme všechny 3 normální formy, automaticky i v BCNF. O co zde jde? Relace je v BCNF, pokud pro každou závislost X->Y platí, že X je nadmnožinou nějakého schématu R. Jinými slovy nesmí existovat žádná funkční závislost mezi kandidátními klíči. Kandidátní klíč je takový, který by mohl zastávat funkci primárního klíče (určovat jedinečnost záznamu). [10]

Vše vysvětlím na příkladu. Pokud bychom měli tabulku se sloupci *Město*, *Ulice*, *PSČ*, tak zde jsou dva kandidátní klíče {*město*, *ulice*} a {*ulice*, *PSČ*}. Protože oba kandidáti mají více atributů a některé z nich jsou společné, porušuje se BCNF.

1.4.6 Čtvrtá Normální Forma (4. NF)

Tabulka je ve čtvrté normální formě, pokud je v BCNF a všechny atributy popisují pouze jeden fakt nebo souvislost.

1.4.7 Pátá Normální Forma (5. NF)

Tabulka je v páté normální formě, pokud je ve 4. NF a přidáním libovolného nového sloupce by se rozpadla na více tabulek.

V praxi se využívá normalizace do 3. NF nebo do Boyce-Coddovy normální formy. Jak už bylo zmíněno, není dobré vše rozkládat a bezhlavě dodržovat normalizaci do co nejvyššího řádu, ale spíše by nám normalizace měla pomoci, abychom neudělali v návrhu závažné chyby. Jedním z postupů, jak navrhnout databázi, je pokusit se rozložit vše do co nejvyšší normy a posléze pospojovat na takovou úroveň, která zbytečně nevytěžuje databázový server.

1.5 Objektově relační model databáze

Patří k třetí generaci databázových systémů (do I. generace patří síťové a hierarchické databáze, do II. řady pak relační koncept), kde se uplatňují objektově orientované systémy. První požadavky na vznik objektově relačních systémů spadají do roku 1990, ale první je konkretizoval v roce 1996 M. Stonebraker. Nejedná se o napsání nové koncepce, ale o rozšíření stávajícího relačního modelu. Základní doporučení stanovují následující:

- zachování jazyka SQL
- mechanizmus vytváření relací
- kompatibilita s existujícími relačními databázemi
- existence pohledů

Mezi další doporučení patří převzetí hlavních rysů z objektově orientovaného přístupu, tedy: komplexních datových typů, zapouzdření, dědičnosti, polymorfismu. Tento model podporuje nenormalizované relace (nejsou v 1. NF), ale rozšiřuje o bohatší typový systém, OO rysy a složité datové typy:

- kolekce (množiny, multimnožiny, pole)
- rozsáhlé objekty (LOB – Large Objects, BLOB – rozsáhlá binární data, CLOB – rozsáhlá znaková sada)
- uživatelem definované typy (UDT – například objekt s vlastním konstruktorem)
- dědičnost – na úrovni typů nebo tabulek
- typ reference (REF) – lze vytvářet explicitní reference řádků tabulek, stejně jako u OO databází

Tento model funguje tak, že do tabulek začal ukládat složitější datové struktury (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. ORDBMS jsou nadmnožinou RDBMS a pokud se nevyužije žádné objektové rozšíření, jsou ekvivalentní s SQL2. Proto má určitá omezení týkající se dědičnosti, polymorfismu a integrace s programovacím jazykem.

Většina velkých výrobců relačních databází, kteří chtěli reagovat na nástup objektově orientovaných databází, zvolila právě tuto variantu. Například firma Oracle implementovala objektově orientované prvky ve své verzi 8 v roce 1997. [8]

1.6 Objektový model databáze

S rozvojem objektově orientovaného programování začaly vznikat i objektově orientované databázové systémy. Základním prvkem není tabulka jako u relačních databází, ale data se sdružují do objektů. U objektových databází existuje stejně jako v OOP dědičnost, zapouzdření a polymorfismus. Každý takový objekt má svoje vlastnosti, metody a svoji identitu, tedy je jednoznačně identifikovatelný (OID). OID na logické úrovni odpovídá ukazateli do virtuální paměti počítače. Primární klíče tedy nejsou v objektových databází potřeba. Požadavky na OID:

- je nezávislé na místě, protože se v době programu může spustit defragmentace
- je nezávislé na hodnotách programu, tj. nemění se ani po změně atributů objektu
- nesouvisí ani se změnou struktury objektu

Nástup OODBMS ani v nejmenším neznamená, že by relační databáze byly na ústupu. Každá technologie má své využití v různých oblastech. U relačního modelu je výhodou lehká pochopitelnost, má základ v matematice a je masivně rozšířená. Tato jednoduchost ale znamená malou modelovací sílu pro složité objekty.

Hlavní výhodou OODBMS je možnost přímého vyjádření složitosti modelované reality v databázi. Základem je ukládání nejen dat ale i chování objektů. Navíc odpadá složitá konverze do normalizovaných tabulek, jak je tomu u relačních databází. Je rovněž jednodušší i samotné načítání objektů, kdy se ihned pracuje s objekty, kterým rozumí aplikace, a nemusí se provádět složité konverze na relační tabulky. Tyto konverze nejenom snižují výkonnost serveru, ale narůstá složitost aplikace a vzniká větší riziko výskytu chyby. Objektové třídy programovacího jazyka jsou totožné (konzistentní) s třídami používanými v OODBMS. Proto je není nutno nikterak konvertovat.

Hlavním požadavkem na OO databázi je perzistence, tj. aby objekt existoval v databázi i po ukončení programu a při dalším spuštění se mohl znova načíst. Další požadavky na OO databázi by měly být totožné s relačními, tj. omezení přístupu k objektům pro dané uživatele, možnost vrátit změny na objektu (ukládání historie), transakce atd.

Pokud si shrneme výše uvedené informace, relační databáze se hodí pro velké množství jednoduchých dat, obsahující propracované prostředky pro správu a selekci těchto dat. Naproti tomu OO databáze dobře vystihují modelovanou realitu, ale nemají tak propracovaný systém dotazovacích jazyků. Existuje standard pro OO databáze, který vytvořila skupina ODMG (Object Database Management Group). Tato skupina se snaží definovat následující:

- objektový model dat
- ODL (Object Definition Language) – specifikace objektově orientovaných schémat
- OQL (Object Query Language) – definice dotazů nad databází
- Vazbu OOSŘBD na jazyky: C++, Smaltalk, Java

ODMG je sestavena z předních společností dodávajících či vyvíjejících OO databázové produkty. Jasným cílem bylo zavedení standardů v oblasti objektově orientovaných databázových jazyků. V současné době je verze ODMG 3.0 vydaná v roce 2001. Mezi OO databázové systémy patří: Caché, db4o, perst a další. [11] [18]

1.7 Objektová normalizace

Většina odborných textů tvrdí, že objektové databáze není třeba normalizovat. Ale bohužel pro objektový datový model dnes neexistuje žádná uznávaná technika nebo metoda návrhu. Mohou se převzít relační techniky, ale vznikne nám jenom relační databáze v objektovém prostředí. To však nevyužijeme všechny výhody, které má objektový datový model. Jako lepší se jeví převzetí schematu objektů a tříd přímo z aplikace, protože právě proto byl tento objektový model navržen. Na druhou stranu mohou některé konstrukce působit problémy při ukládání. Proto by se měla provádět objektová normalizace už při návrhu aplikace. [25]

Problém normalizace je daleko více potřebnější u relačního modelu, protože tam se přímo pracuje s jednotlivými atributy. Objektový model pracuje s objekty, které navenek vystupují jako nedělitelné jednotky. Přesto bychom tuto problematiku neměli přehlížet.

Rozdíl mezi relačním a objektovým modelem:

- Normalizace u RDM je závislá především na závislosti mezi atributy. Tyto atributy jsou dále nedělitelné. U ODM ale pracujeme s objekty, které shrnují atributy dohromady a navíc jako atribut může být použit jiný objekt.
- Dalším rozdílem je identita základního prvku. U RDM je to primární klíč, který se může skládat z jednoho nebo více atributů. U ODM je ale identita objektu zachována i tehdy, změní-li se všechny jeho atributy.
- Objekty se také liší v tom, že kromě atributů obsahují i metody.

Na téma objektové normalizace vzniklo několik prací, jako třeba Chodorkovského ONF, Nootenboomova OONF nebo Ambler-Beckovy tři normální formy. Žádná z nich a ani žádná jiná není oficiálně uznávána za standard. Na toto téma vznikl článek od doc. Ing. Vojtěcha Meruňky, Ph.D. [25]. Jedná se o článek s názvem *Normalizace v objektových databázích*. Shrnul v něm dosavadní existující teorie. Zde uvádím krátký přehled objektových normálních forem, jak je ve svém článku uvedl:

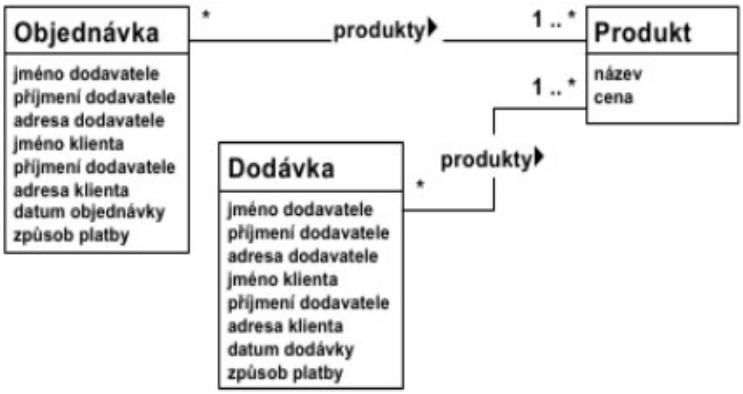
1.7.1 Nultá Objektová Normální Forma (0. ONF)

Třída je v nulté objektové normální formě, jestliže její objekty obsahují skupinu opakujících se atributů.

Obrázek 4: Ukázka nenormalizovaných objektů (0. ONF) [25]

1.7.2 První Objektová Normální Forma (1. ONF)

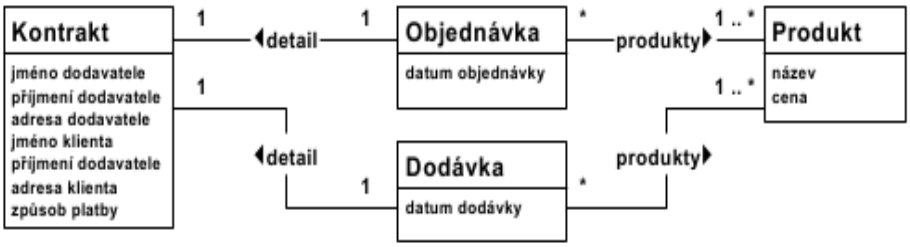
Třída je v první objektové normální formě, jestliže její objekty neobsahují skupinu opakujících se atributů. Takové atributy je třeba vyčlenit do objektů nové třídy. Schéma je v 1. ONF, jestliže všechny třídy objektů v něm jsou také v 1. ONF.



Obrázek 5: Ukázka 1. ONF [25]

1.7.3 Druhá Objektová Normální Forma (2. ONF)

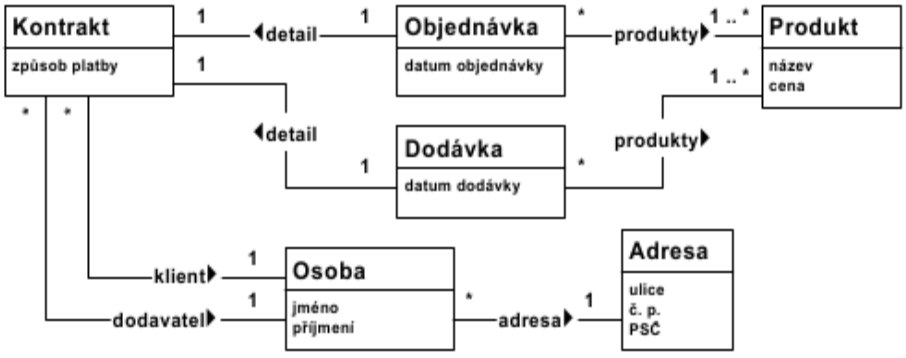
Třída je ve druhé objektové normální formě, jestliže její objekty neobsahují atribut nebo skupinu atributů, které by byly sdílené s nějakým jiným objektem. Schéma je v 2. ONF, jestliže všechny třídy objektů v něm jsou v 2. ONF.



Obrázek 6: Ukázka 2. ONF [25]

1.7.4 Třetí Objektová Normální Forma (3. ONF)

Třída je ve třetí objektové normální formě, jestliže její objekty neobsahují atribut nebo skupinu atributů, které mají samostatný význam nezávislý na objektu, ve kterém jsou obsaženy. Schéma je v 3. ONF, jestliže všechny třídy objektů v něm jsou v 3. ONF.



Obrázek 7: Ukázka 3. ONF [25]

1.8 Budoucnost v modelech

Dále mezi post-relační modely, kromě objektově orientovaného a objektově relačního, můžeme zařadit některé další vznikající formy modelů. Tyto modely jsou většinou specializované a zatím málo rozšířené.

1.8.1 Deduktivní databáze

Deduktivní databázový systém vytváří dedukce založené na faktech a pravidlech uložených v deduktivní databázi. Kombinuje se logické programování s relačními databázemi tak, aby vznikl systém rychlý a schopný pracovat s velkými objemy dat. Typický jazyk, který specifikuje fakta, pravidla a dotazy se nazývá *Datalog*. Tento druh databází zatím nemá uplatnění mimo výzkumné ústavy a použití v praxi je spíše teoretické.

1.8.2 Multimediální databáze

Je koncipována pro správu multimediálních dat. Tyto data se vyznačují velkým objemem od 1MiB (malé obrázky) do několika desítek gigabitů (filmy). Je zde kladen velký důraz na vyhledávání, protože mnohdy se vyhledává podle obsahu – tzv. podobnostní vyhledávání. Známým zástupcem této kategorie je Oracle interMedia. [8]

1.8.3 Podpora modelování času

V temporálních databázích se pracuje zejména s časovými údaji. Příkladem může být kurz na tiketech jednotlivých sportovních zápasů. Kurz se neustále mění a je potřeba zpracovávat stará i nová data. U těchto databází se zavádí pojem *granuralita*, což je nejmenší časový úsek, který se dá rozlišit. Čím menší je granuralita, tím je větší přesnost, ale větší objem dat. Dále je zde pak pojem *časové razítko*, které se skládá z „času transakce“, tedy čas vložení a čas platnosti. To je údaj, jak dlouho bude daný záznam aktuální. Někdy se tento systém označuje jako systém časových razítek. Jako dotazovací jazyk se zde používá TSQL, který je téměř totožný s SQL, ale je doplněn o speciální klauzule: `first`, `last`, `then`. [8]

1.8.4 Systémy na podporu rozhodování

Jsou to vlastně speciální databáze podporující analytické dotazování nad daty. Jedná se o datové sklady (zvláštní typ relační databáze), kde jsou uložena komplexní data ve struktuře, podporující efektivní analýzu a dotazování. Datové sklady se plní z primárních informačních zdrojů. Často se používá třívrstvá architektura: [8]

- **spodní**: Zahrnuje server skladu, kde běží relační databáze. Je to tedy datový sklad.
- **prostřední**: Provádí operace nad multidimenzionálními daty. Je to OLAP server.
- **vrchní**: Nástroje pro provádění dotazů a analýz. Je to tedy klient.

1.9 Shrnutí

V současnosti jsou určitě nejrozšířenější a nejvyspělejší relační modely databáze zejména pro svoji dlouholetou osvědčenost, výkonnost a bezpečnost dat. I jejich směr ale ovlivnil nástup OOP, a tak začaly vznikat objektově relační databáze, které dávají dohromady výhody relačních databází doplněné o některé základní rysy OOP. Zcela novou koncepci pak přinesly čistě objektové databáze, kde se pracuje s čitými objekty. Tyto databáze nemají tak vysokou ochranu dat a dotazovací schopnosti jako relační databáze, ale jejich přístup je pro některé aplikace jasnou výhodou. Rychle se srovnává i výkon obou modelů, kdy jsou zatím o trochu rychlejší relační databáze. Výběr jednotlivého modelu závisí čistě na potřebách aplikace využívající databázi. Co se týče ostatních post-relačních modelů, jejich rozšířenost není zatím tak velká, ale hodí se do specializovaných aplikací, kde vynikají svými výkony, pro které jsou navrženy.

Vše nasvědčuje tomu, že v budoucnu se začne ustupovat od jednotného modelu, používajícího se v řadě různých aplikací (dnes relační). Budou ale vznikat specializovanější modely (např. datové sklady, multimediální databáze), které budou pro konkrétní aplikaci nejlepší. Na výběr bude více konkrétních a specializovaných modelů.

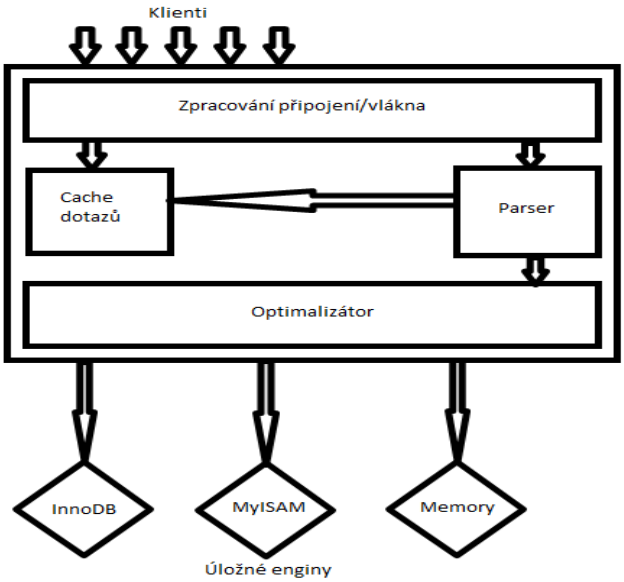
2 Popis nejznámějších databází

V předešlé kapitole byly nastíněny jednotlivé databázové modely. Tato kapitola se zaměří na nejznámější hráče na trhu, které připadají pro vytvářený systém v úvahu. To znamená, že musí být distribuovány pod volně šiřitelnou licencí a musí být kompatibilní s programovacím jazykem projektu. Rozhodně není cílem popsat konkrétně každou databázi nebo vyjmenovat všechny databáze. Je zde ale ukázána odlišnost databází relačních a objektových. Z každé této kategorie je pak vybrán jeden zástupce. Ten je pak detailněji popsán a je na něm ukázáno, jak se s daným modelem pracuje. Z relačních databází se jedná o MySQL, protože je to nejpoužívanější databáze pro webové technologie. Objektové databáze pak zastupuje db4o.

2.1 MySQL

Jedná se o multiplatformní databázi původně vytvořenou švédskou firmou MySQL AB, kterou později koupila společnost Sun Microsystems, kterou nedávno odkoupila společnost Oracle Corporation. Její hlavní výhodou je snadná instalace a konfigurace a volně šiřitelná licence. Je rozšířená zejména v oblasti vývoje webových stránek, protože je na tuto oblast především specializována. Je tedy optimalizována na jednoduchost a rychlost. Drtivá většina webových vývojářů používá jenom základní příkazy (`Insert`, `Delete`, `Select`, `Update`) a tím degradují databázi na prosté skladiště dat. Hlavním dotazovacím jazykem je zde SQL s některými přidanými funkcemi.

Architektura MySQL serveru se dělí do tří vrstev. Vrchní vrstva spravuje připojení klientů k databázi. Druhá vrstva je nejdůležitější vrstvou, ve které dochází k parsování dotazů, analýze a optimalizaci dotazů. Po předání dotazu se nejprve hledá v cache dotazů. Zde se ukládají příkazy `select` a jejich výsledná data. Pokud najde dotaz v cache, server nemusí znova optimalizovat dotaz a někdy ho nemusí ani vykonat nad daty, ale vrátí výsledek z cache. Pokud dotaz v mezipaměti není, MySQL provádí složitý rozbor dotazu. V něm si zjišťuje nákladnost jednotlivých operací, pořadí čtení tabulek nebo jaké indexy bude používat. Poslední vrstva obsahuje úložné enginy. Server s nimi komunikuje pomocí jejich API a mají na starosti ukládání dat. MySQL podporuje několik úložných enginů z nichž nejznámější jsou *MyISAM* nebo *InnoDB*. [14] [35]



Obrázek 8: Architektura serveru MySQL [14]

První verze MySQL byly optimalizovány pro rychlost a záměrně neměly tolik funkcí pro práci s daty jako konkurenti. Postupem času ale přibývají potřebné funkce, které se stávají víceméně standardem. Od verze 5 jsou to: uložené procedury, triggery, pohledy, kurzory, replikace na úrovni řádků atd.

Pro ukázky prací se všemi relačními databázemi je použit jazyk Java v prostředí NetBeans. To se hodí v následující kapitole **Výběr nejvhodnější databáze**, kde je nutné, aby porovnání proběhlo ve stejném programovacím jazyce a prostředí.

2.1.1 O rozhraní JDBC (Java Database Connectivity)

Je to jednotné rozhraní (API) pro přístup k jakékoliv databázi z jazyku Java. Je inspirováno technologií ODBC (Open Database Connectivity), kterou vytvořila firma Microsoft. Rozhraní JDBC je jednotné a nezávislé na konkrétní databázi. Výrobce databáze musí pak dodat ovladač JDBC. Rozhraní umožňuje přístup k výsledkům dotazů tak, že jako výsledek SQL dotazů vrací instance Java tříd, se kterými pracuje aplikace. Rozhraní dále umožňuje vykonávání SQL dotazů způsobem, že do dotazu vkládá přímo instance objektů. Protože se s rozhraním pracuje u všech databází stejně, je ukázána práce jenom na MySQL, u ostatních databází je práce totožná. [15]

2.1.2 Připojení k databázi

Prvně se musí v prostředí NetBeans připojit knihovna *mysql-connector-java*. Lze ji stáhnout přímo z oficiálních stránek MySQL [35]. Připojení se provede pravým tlačítkem myši na Libraries a stiskne se položka „Add JAR/Folder...“. Zde se vybere stažený soubor. Následuje přesun do implementační části a v kódu se musí nejprve registrovat ovladač pomocí příkazu `forName(String nazevTridy)`. [15]

```
Class.forName("com.mysql.jdbc.Driver");
```

Pokud je zaregistrován ovladač JDBC, přistoupí se k samotnému připojení k databázi. To zajišťuje metoda `getConnection(String url, String name, String heslo)`. Metoda vrací spojení jako instanci třídy Connection. Ukázka ukazuje přístup na lokální databázi MySQL s uživatelským jménem root a s nezadaným heslem.

```
String connectionUrl =
"jdbc:mysql://localhost/netbeans?"+"user=root&password=";
Connection con = DriverManager.getConnection(connectionUrl);
```

2.1.3 Zadávání dotazů

Nejprve se musí získat objekt typu statement, pomocí něhož se zadávají jednotlivé příkazy na databázi. Tento objekt je získán metodou `createStatement()` zavolaného na objektu spojení. [15]

```
Statement stmt = (Statement) con.createStatement();
```

Konkrétní příkazy jsou vkládány do následujících metod instance Statement. Jako parametr všech metod je očekáván řetězec v podobě SQL dotazu a v případě neúspěšného volání vracejí SQLException. Metody jsou následující:

a) `executeUpdate()` - Metodou se zadávají příkazy DML (`insert`, `delete`, `update`) a DDL (`create`, `alter`, `drop`). V případě DML vrací metoda počet ovlivněných řádků a u DDL vrací 0. Příklad:

```
stmt.executeUpdate("INSERT INTO Tabulka (id, jmeno) VALUES ('33', 'insert');");
```

b) `executeQuery()` – Metoda je určená k získávaní dat z databáze. Výsledek vrací v podobě objektu ResultSet. Příklad:

```
ResultSet vysledky = stmt.executeQuery("SELECT id, jmeno FROM Tabulka");
```

c) `execute()` – Používá se, pokud výsledkem dotazu není jedna hodnota (počet nebo ResultSet). Návratová hodnota vrací true, pokud první v řadě výsledků je ResultSet, a false, pokud je to int. Návratové hodnoty se procházejí pomocí metody `getMoreResult()` a hodnoty se získávají pomocí `getResultSet()`.

2.1.4 Čtení vrácených dat

Jak je uvedeno výše, metoda `executeQuery()` vrací objekt ResultSet, který se dá představit jako tabulka s vrácenými daty. V objektu existuje vnitřní ukazatel, který ukazuje na některý řádek. Při vrácení (vytvoření) ukazuje na nultou pozici. Na následující pozici se přesune metodou `next()`. Pokud existuje další záznam v objektu, metoda vrací *true*, v opačném případě vrací *false*. Příklad přečtení dat z výsledku dotazu:

```
while (vysledky.next()) {
   id = vysledky.getString(1);
   jmeno = vysledky.getString(2);
   System.out.println("id:" + id + " jmeno:" + jmeno);
}
```

JDBC API nabízí rozšiřující funkce procházení a načítání dat výsledků. Při pohybu v řádcích tabulky zavádí další příkazy:

- `previous()` – Přesun ukazatele na předchozí řádek.
- `first()` – Přesun vnitřního ukazatele na první řádek.
- `last()` – Přesun na poslední řádek.
- `beforeFirst()` – Nastavení kurzoru před první řádek.
- `absolute(int cisloRadku)` – Přesun kurzoru na řádek se zadaným číslem.
- `relative(int pocetPozic)` – Posun kurzoru o zadanou hodnotu.

Pro čtení hodnot jsou tu metody `getString()`, `getInt()`, `getObject()` a každá z nich je přetížena na dvě varianty parametru. Buď zadáváme string jako název sloupce, nebo int jako číslo sloupce z výsledné množiny. [15]

2.1.5 Zavření spojení s databází

Po ukončení práce s databází zavoláme metodu `close()` na objektu Connection. Pokud zavoláme close na tomto objektu, automaticky se zavolají metody `close()` pro objekty typu Statement a ResultSet. U těchto objektů je možné volat metodu `close()` i samostatně, ale zavře se jenom daný objekt.

2.2 PostgreSQL

Jedná se o objektově relační databázový systém určený primárně pod systém Linux (Unix), existují ale i balíčky pod Windows. Databáze je vydána pod licencí MIT (svobodná licence, ale text licence musí být dodán s aplikací). [17]

Databáze vznikla z původního projektu Ingers na university of California, Berkeley. Michael Stonebraker jako vedoucí projektu odešel a začal pracovat na své vlastní proprietární verzi Ingres. V roce 1985 se do Berkeley vrátil a začal pracovat na projektu post-Ingers. Projekt měl 4 oficiální verze, po kterých v roce 1994 skončil. Jeho zdrojový kód se ale dostal do rukou open-source vývojářům pod licencí MIT. První verze už pod novým názvem PostgreSQL byla vydána 1. srpna 1996. Nyní je již databáze ve verzi 9 a podporuje velké množství databázových operací a vlastností [17]:

- možnost spouštění funkcí v jiných jazycích (C++, Java, plPHP, pl/Perl)
- multi-version Concurrency Control (MVCC), kde každý uživatel vidí „snapshots“ a

může v něm pracovat, aniž by ostatní uživatelé viděli změny před transakcí

- veliký výběr datových typů jako IPv4, IPv6, MAC adresy, geometrické tvary a další
- rozsáhlá podpora indexů (včetně částečných indexů nad částí tabulky)
- online backup (zálohování za plného provozu databáze) atd.

Omezení uvedená na domovských stránkách [16]:

- maximální velikost tabulky: 32 TB
- maximální velikost řádku: 1,6 TB
- maximální velikost záznamu: 1 GB
- maximální databázová velikost: neomezeno

Mezi nejznámější zákazníky patří Skype VoIP aplikace (centrální databáze) nebo MySpace sociální síť. Databázi rovněž využívá i Yahoo!, který ukládá chování uživatelů na internetu. Jedná se asi o největší datový sklad (2 petabyty) [16].

2.3 Shrnutí relačních databází

Jednoznačně nejrozšířenější databází v oblasti webu je MySQL. Hlavním důvodem je, že ho podporují snad všechny webhostingové servery. Dalším důvodem je oblíbenost balíčků LAMP (Linux, Apache, MySQL, PHP), což je odladěná konfigurace pro tvorbu webu a je velice snadno instalovatelná. Díky tomu se dnes začínající vývojáři v oblasti web tvorby uchylují k MySQL a prozatím je jeho místo zcela neohrožené. V ostatních aplikacích (tedy ne na webu) je nejrozšířenější databáze Firebird následována PostgreSQL. Co se týče komerčních projektů, tam zcela jasně vede Oracle, který je lídrem na trhu s databází a udává směr relačních a objektově-relačních databází.

2.4 db4o (database for objects)

Je databází americké společnosti *Versant Corporation* a je to objektová databáze. Podporuje jazyky Java a C#. Tato databáze má i některé velice významné zákazníky, kteří ji využívají v elektronice letadel Boeing, u palubních počítačů vozů BMW, u výrobků firmy Bosch nebo giganti jako Intel a Seagate. Db4o má dvojí licencování, což znamená, že je dostupná ve dvou verzích. První je volná (free) a je podmíněná tím, že výsledný projekt bude pod licencí GPL (General Public License). Placenou verzi (US$ 1,200) využijete, pokud databázi používáte v komerčním projektu nebo potřebujete profesionální podporu. [18]

Db4o je jak typu embedded, tak typu klient/server. Embedded databáze znamená, že se databáze distribuuje v podobě knihovny, která je přilinkována k programu. Nemusí se tedy instalovat a zprovozňovat externí server. Databáze podporuje i víceuživatelský přístup nebo transakce.

Následuje ukázka základních operací s objektovou databází. Db4o ukládá data do souboru, který se jmenuje stejně jako databáze.

2.4.1 Otevření databáze

Databáze je reprezentována instancí třídy ObjectContainer. Tuto instanci vrací statická metoda `openFile()` třídy db4o. Metoda má dva parametry. Nepovinný parametr Configuration, kde se nastavují parametry týkající se indexů nebo unikátnosti datových složek. Druhý parametr je povinný a je zde očekáván název databáze. Pokud databáze existuje, otevře se, jinak se vytvoří databáze nová. [18]

```
Configuration conf = Db4o.configure();
conf.objectClass(NazevTridy.class).objectField("atribut").indexed(true);
conf.add(new UniqueFieldValueConstraint(NazevTridy.class, "atribut"));
db = Db4o.openFile(conf, „nazevDatabaze“);
```

2.4.2 Uzavření databáze

I objektová databáze by se měla uzavřít, až nebude její instance potřeba. K tomu slouží metoda `close()`. Tato metoda volá automaticky metodu `commit()`, což zabraňuje tomu, aby se databáze neuzavřela v nekonzistentním stavu. Nakonec je vhodné instanci přiřadit hodnotu null. [18]

```
db.close();
db = null;
```

2.4.3 Ukládání do databáze

K uložení slouží metoda `store()`, která je rovněž instancí třídy ObjectContainer. Ta se zavolá nad objektem, který chceme uložit. Objekt je následně zapsán do bufferu databáze. Buffer se fyzicky zapíše do souboru zavoláním metody `commit()`, nebo použitím příkazu, který commit obsahuje automaticky (`close()`). Do databáze se dají ukládat libovolné objekty.

```
Trida nazevInstance = new Trida(„Nazev“, pocet);
db.store(nazevInstance);
db.commit();
```

2.4.4 Dolování dat

Data lze z databáze získat dvěma základními způsoby. Buď pomocí QBE (Query By Example) nebo pomocí predikátu.

a) Query By Example – Jedná se o šablonu objektu, podle které se provede vyhledání. Šabloně se nastaví jen ty vlastnosti, které mají mít i objekty ve výsledku. Ostatní hodnoty jsou nastaveny jako nulové (0 u primitivních datových typů, null

u objektových typů), a tím se při vyhledávání neberou v úvahu. Na objektu databáze se zavolá metoda `queryByExample()` s parametrem instance šablony. Mezi hlavní nevýhody patří u tohoto druhu dotazů nemožnost vyhledání dle nulových atributů a dále pak vyhledávání dle jiného kritéria než ekvivalence. Následující příklad vytáhne z databáze všechny instance se jménem Zdeněk. [18]

```
Trida sablona = new Trida("Zdeněk", 0);
ObjectSet vysledek = db.queryByExample(sablona);
```

b) prohledávání pomocí predikátu – Je to způsob, který umožňuje prohledávání podle přesně zvolených atributů. K tomuto účelu se využívá abstraktní třída Predicate a její metoda Match. Tato metoda se chová obdobně jako metoda equals v programovacím jazyku Java. Jako vstupní parametr je jí předán objekt a výstupem je hodnota true, v případě shody srovnání, false v opačném případě. V těle metody `Match()` lze provést libovolné srovnání. [18]

```
List <Trida> vysledek = db.query(new Predicate<Trida>() {
   public boolean match(Trida vstup) {
      return vstup.getPoints() == 100;
   }
});
```

2.4.5 Aktualizace dat

Úprava objektu se provádí totožně jako ukládání objektu. Nejprve se musí objekt získat z databáze, poté se změní příslušné atributy a opět se na objekt zavolá metoda store. Je zde ovšem nutné dát pozor na to, aby se pracovalo s načteným objektem, jinak by mohlo dojít k nechtěnému opětovnému vložení. Následující příklad změní jméno.

```
ObjectSet vysledek = db.queryByExample(new Trida("Zdeněk", 0));
Trida nalezeno = (Trida) vysledek.next();
nalezeno.setName(„Franta“);
db.store(nalezeno);
```

2.4.6 Mazání dat

Principiálně je mazání podobné aktualizaci dat. Nejprve se musí získat požadovaný objekt. Tento objekt je předáván jako parametr metodě `delete()` nad instancí ObjectContainer. Db4o neumožňuje mazání objektů podle prototypů. Snaží se tím zamezit nechtěnému smazání velkého počtu položek. Jak smazat z databáze Zdeňka ukazuje následující příklad.

```
ObjectSet vysledek = db.queryByExample(new Trida("Zdeněk", 0));
Trida nalezeno = (Trida) vysledek.next();
db.delete(nalezeno);
```

2.4.7 Závěr

S databází db4o se pracuje velmi jednoduše a intuitivně. Jedná se o oblíbenou objektovou databázi pro jazyk Java. Mezi výhody také patří možnost zabudování přímo do aplikace. Je to jedna z nejvyspělejších OO databází. Nevýhodou může být menší počet česky psaných textů a informací k této databázi, v angličtině jich je ale velké množství.

2.5 Perst

Je to open source objektově orientovaný embedded databázový systém. Je vytvořen ve dvou verzích v Jave (pro Javu) a v jazyce C# (pro technologie .NET). Je pod dvojím licencováním a to jako komerční verze a GPL verze k volnému použití. Existuje i verze Lite, která je určená pro mobilní telefony, PDA a další zařízení na základě Sun Microsystems. Databáze podporuje transakce, několik druhů indexů, vícevláknový přístup, XML export/import atd. Existují nástroje podporující spolupráci s UML nebo pomáhající vizuálnímu zobrazování. [24]

2.6 NeoDatis ODB

Je dostupná ve dvou verzích, a to jak v módu client/server, tak ve verzi embedded. Je distribuována jako jediný soubor jar, který se přidá do aplikace. Podporuje transkace, index na jednotlivých atributech, multiuživatelský přístup, xml import/export atd. K databázi existuje program Object Explorer, kde můžeme procházet, vytvářet, měnit a mazat objekty. Dále je zde možnost dotazování. Databáze má jako jedna z mála velice kvalitní domovské stránky [22], kde se uživatel dozví vše potřebné. Databáze po celou dobu testů fungovala spolehlivě a práce s ní byla intuitivní. [22]

2.7 Přehled dalších objektových databází

Protože existuje přibližně deset objektových databází, které jsou pod volnou licencí a jsou si velice podobné, v následující tabulce jsou shrnuty jejich důležité informace.

Název	Jazyk(y)	Licence	www	Popis
Databeans	Java	GPL	http://databeans.sourceforge.net/	Plně objektový rámec pro Javu, klient/server, transakční, XML
Db4o	C#, Java	GPL, Komerční	http://www.db4o.com/	Asi nejznámější a nejlepší nekomerční objektová databáze.
GigaBASE Database Management System	C++, C#, Java, PHP, PERL	MIT	http://sourceforge.net/projects/gigabase/	Objektově-relační databáze poskytující SQL i práci s objekty.
JDOInstruments	Java	LGPL	http://www.jdoinstruments.org/	Zcela vlastní koncepce určená jenom pro integraci do Netbeans.
JODB (Java Objects Database)	Java	GPL	http://www.java-objects-database.com/	Čistě komunitní databáze podobná db4o.
MyOODB	Java	GPL/LGPL	http://www.myoodb.org/	Distribuovaný objektový framework s OO databází.
NeoDatis ODB	C#, Java, Mono	LGPL	http://www.neodatis.org/	Jednoduchá databáze podporující Embedded a Client/Server přístup.
Ozone Database Project	Java	GPL, LGPL	http://www.ozone-db.org	Čistě komunitní databáze psaná v jazyku Java.
Perst	C#, Java	GPL, Komerční	http://www.mcobject.com/perst	Propracovaná embedded databáze.
JOAFIP object persistence in file	Java	LGPL, Open source	http://joafip.sourceforge.net/	Jednoduchá embedded databáze.

Tabulka 5: Seznam objektových databází

3 Výběr nejvhodnější databáze

Následuje porovnání některých relačních a objektových databází. Je ukázána především standardní práce, tedy ukládání a vyhledávání, což jsou nejčastější operace nad databází. Každá operace byla testována vícekrát a výsledky zde budou uvedeny jako průměr naměřených hodnot. Jedná se pouze o databáze, které jsou volně dostupné.

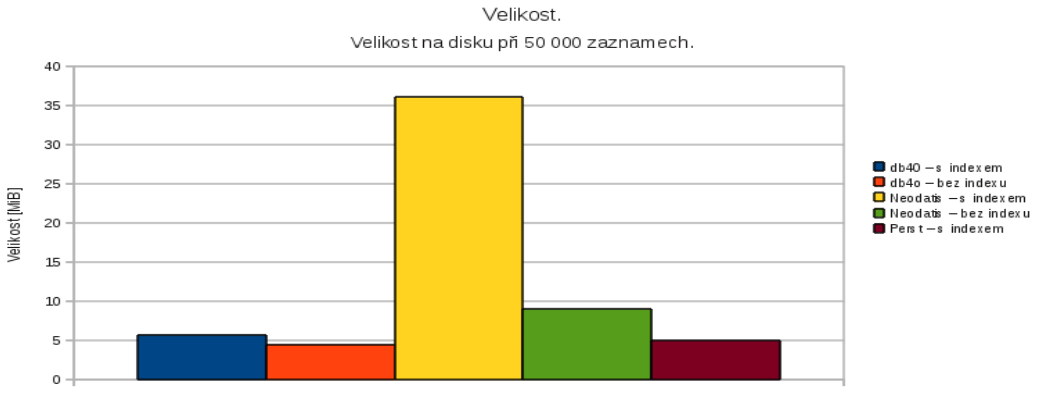
Databáze jsou testovány pod operačním systémem Linux openSUSE 11.3 (x86_64). Hardwarová konfigurace je Intel(R) Core(TM)2 Duo CPU T5670 @ 1.80GHz. Paměť RAM má velikost 2GiB. Testovací prostředí je NetBeans 6.9.1.

3.1 Testování objektových databází

U objektových databází byly testovány databáze db4o, NeoDatis a Perst. Pro testování byl vytvořen jednoduchý objekt, který obsahuje tři položky: *klic (long)*, *jmeno (String)*, *prijmeni(String)* a jednu metodu pro výpis objektu. Kód objektu:

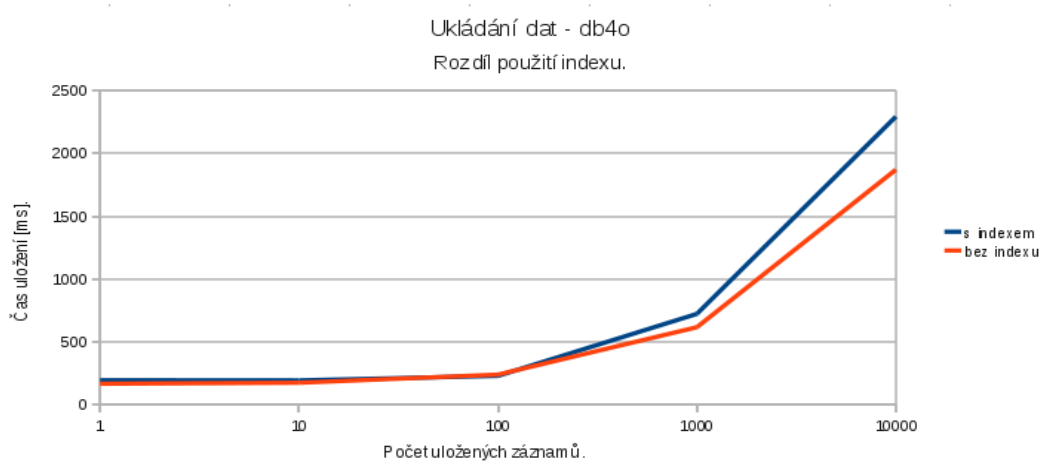
```
public static class Zaznam {
    long klic;
    String prijmeni;
    String jmeno;
    @Override
    public String toString() {
        return "Jsem objekt: " + klic + " - " + jmeno + " " + prijmeni;
    }
};
```

Nejprve byly otestovány velikosti datových úložišť (souborů) při uložení 50 000 objektů. Nejméně místa zabírala databáze db4o bez použití indexu na atributu *klic* (4,5MiB), nejvíce pak databáze Neodatis s použitím indexu (36MiB) – viz. graf 1.



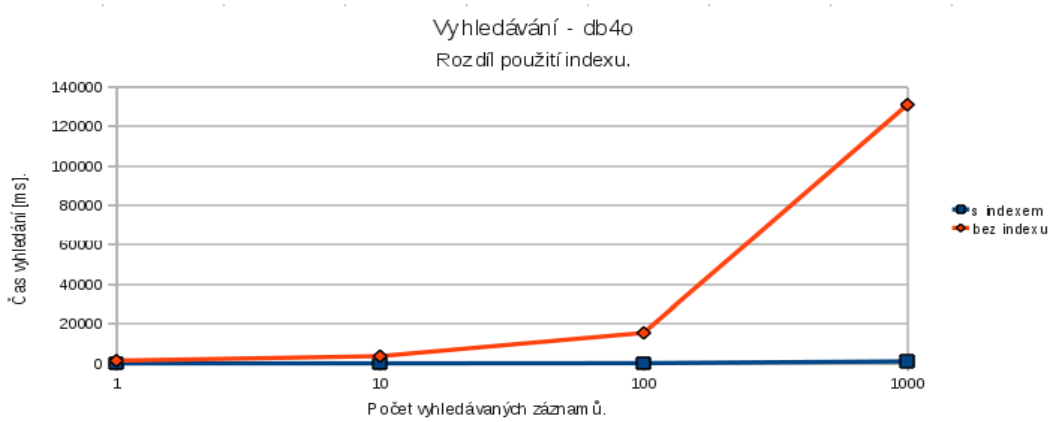
Graf 1: Velikosti databází

Dále byl zkoumán rozdíl v rychlosti ukládání s a bez použití indexu na atributu *klic*. Testy jsou prováděny na databázi db4o. Jak je vidět na grafu 2, rozdíl je minimální a projevuje se až od cca 500 ukládaných objektů, kdy je mírně rychlejší ukládání bez indexu.



Graf 2: Rozdíl při ukládání dat s a bez použití indexu

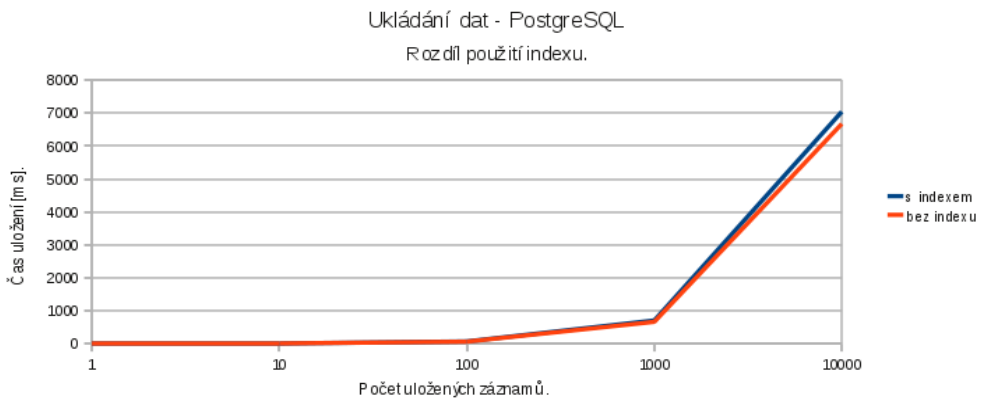
Druhým nejčastějším a zde zkoumaným procesem je vyhledávání. Rozdíl v použití indexu je už naprosto odlišný. Při použití indexu na 1000 záznamech vyhledá za 1,6 s a bez použití indexu je to pak 131 s. Testovaná základna měla 50 000 objektů. Všechna měření potvrzují teoretické předpoklady pro dané operace. Index zpomaluje ukládání, zrychluje vyhledávání a pro uložení indexu je potřeba určitá režie v databázi.



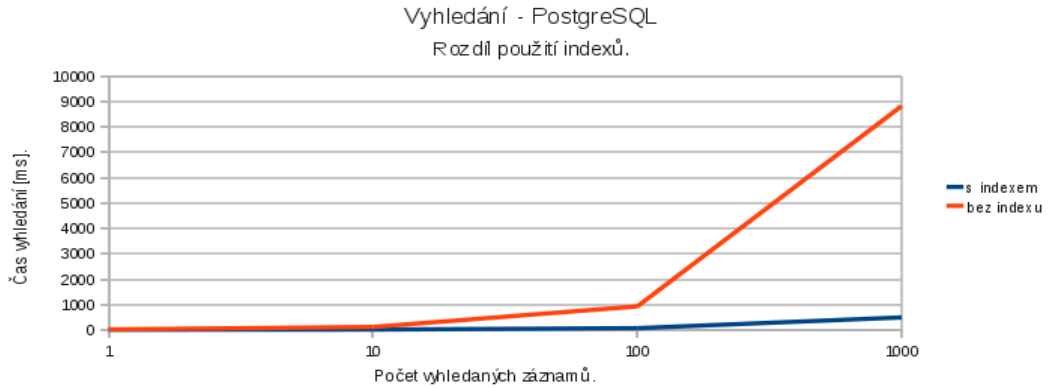
Graf 3: Rozdíl při vyhledávání objektů s a bez použití indexu

3.2 Testování relačních databází

Zajímavé také bylo srovnání použití indexů u relačních databází. K porovnání byla vybrána databáze PostgreSQL. Výsledky jsou obdobné jako u objektových databází. Ukládání s indexem je o něco pomalejší, ale vyhledávání je pak znatelně rychlejší. Naměřené hodnoty si můžeme prohlédnout na následujících dvou grafech.



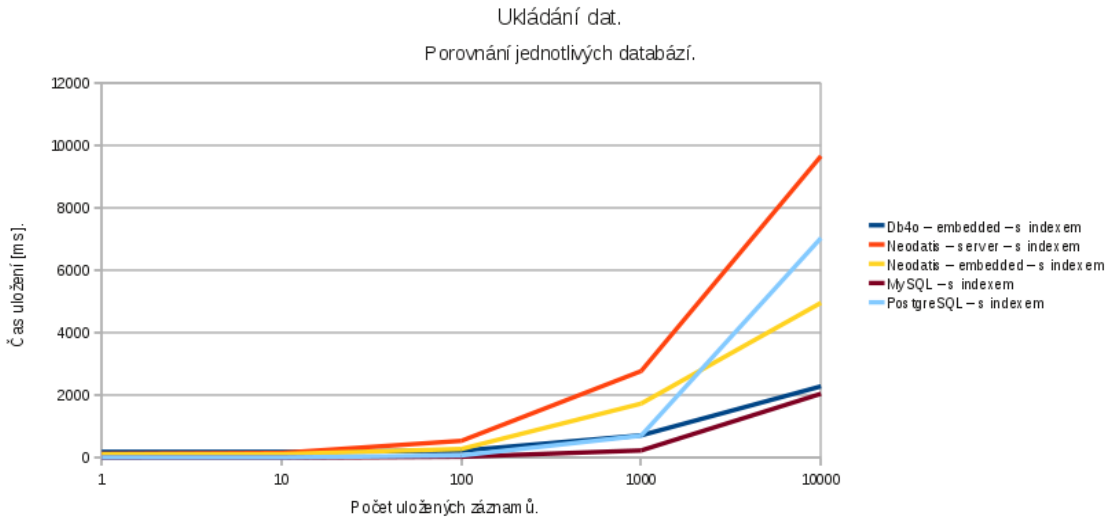
Graf 4: Rozdíl při ukládání dat s a bez indexu u relační databáze PostgreSQL



Graf 5: Rozdíl při vyhledávání objektů s a bez použití indexu u relační databáze

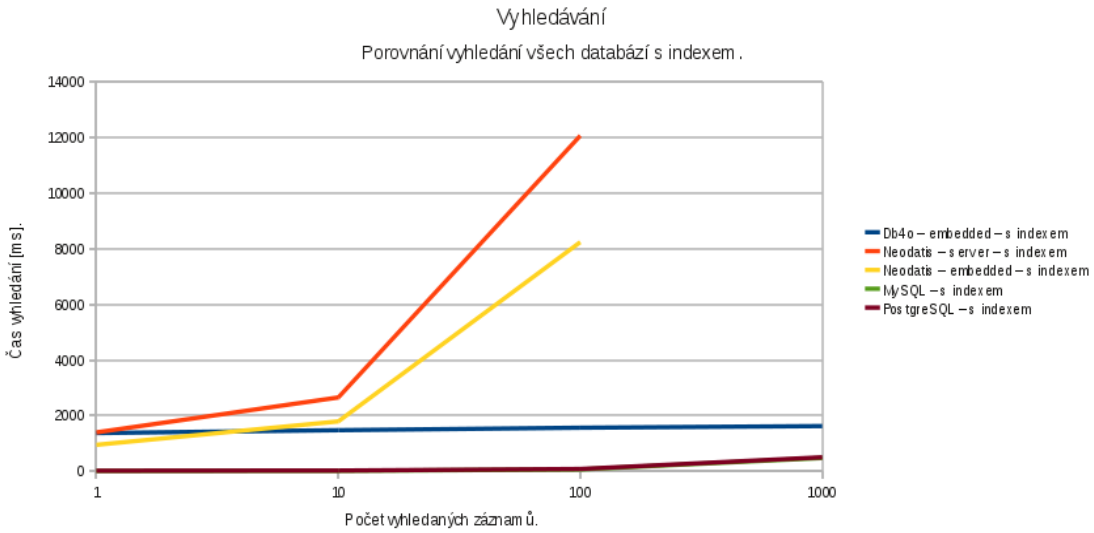
3.3 Porovnání všech testovaných databází

Dále byly porovnány všechny databáze společně - relační i objektové. U všech je použit index na sloupci, podle kterého se vyhledává.



Graf 6: Ukládání dat – všechny databáze

Nejrychlejší databází je relační databáze MySQL. Výsledky ale mohou být zkresleny zvoleným testovacím objektem. Tento objekt je velmi triviální a není zde potřebná tak složitá konverze objektu na relační tabulku. V případě složitějšího objektu nebo při použití dědění by konverze trvala déle.



Graf 7: Vyhledávání – všechny databáze

V případě vyhledávání je vidět, že nejrychlejšími databázemi jsou MySQL zároveň s databází PostgreSQL (křivky splývají). Zde mají náskok ještě výraznější než v případě ukládání. Operace vyhledávání byla spouštěna nad 50 000 uloženými záznamy.

3.4 Shrnutí

Objektové databáze už se velice přiblížily databázím relačním co se týče rychlosti a různých dalších služeb, které poskytují. Jedná se o funkce jako multiuživatelský přístup, podpora transakcí atd. Přesto pokud jde o zcela nekomerční databáze, mají velice slabou podporu ze strany komunity i vývojářů, a tudíž jejich nasazení autor nedoporučuje. Pokud jde o nekomerční projekt, lze úspěšně použít již vyspělé databáze pod dvojím licencováním, jako je například db4o nebo perst.

Relační databáze mají za dlouhou dobu vývoje velice silnou podporu. Rovněž jsou vyspělejší než databáze objektové. Daleko více se hodí i do komerčních projektů.

Pro praktickou část se tedy ukazuje jako nejvhodnější relační databáze MySQL. Kromě toho, že v testech vychází jako nejrychlejší, její velkou výhodou je značná podpora komunity a vývojářů. Dalším kladem jsou externí podpůrné programy, jako např. *MySQL WorkBench*.

4 Metodiky programování – přehled

Kapitola uvádí komponentové programování. Programovací techniky se postupem času vyvíjely a doplňovaly. Postupně se snažily odstraňovat nedostatky verzí předchozích.

V počátcích programování byl důležitý funkční kód a styl, kterým se tento kód psal, byl opomíjen. Postupem času se ale programy začaly stávat složitějšími a nepřehlednějšími. Zejména při spolupráci více programátorů se problém přehlednosti stával velice aktuálním. Proto vzniklo postupně několik programovacích stylů.

4.1 Strukturované programování

Definuje základní pravidla, podle kterých by se měl řídit vývoj SW. Hlavním cílem vzniku tohoto programování je přehlednost zdrojového kódu. Takto psané programy byly sice o trochu delší a pomalejší, ale zato byly napsány v kratším čase a byla i výrazně menší pravděpodobnost vzniku chyby. Strukturované programování dělí výsledný kód do tří základních struktur: posloupnost příkazů, cyklů a podmínek. Za počátek strukturovaného programování se dá označit článek E. W. Dijkstry pod názvem *Go To příkaz považován za škodlivý*. Zde se snaží dokázat přímou úměru mezi počtem skoků v programu a nepřehledností zdrojového kódu.

4.2 Modulární programování

Základem je zde modul, který se dá chápat jako samostatná část programu. Tato část je obvykle uzavřená a komunikaci s ní zajišťuje rozhraní daného modulu. Výsledkem je systém vzájemně provázaných modulů. Ty mohou být členěny do více úrovní, kde jeden modul může být složen z několika modulů nižší úrovně. Tento proces se nazývá **technika modulární dekompozice**. Ta rozděluje problém na několik menších problémů, které je možno řešit odděleně, a dále je rekurzivní (rozkládá se už rozložená část). Nejčastější postup dekompozice je shora dolů. Tato technika ale vytváří moduly, které řeší primárně nadřazený problém. Obecně bychom se měli snažit, aby moduly byly univerzální a šly znovu použít.

4.3 Objektově orientované programování

OOP (Object Oriented Programming) se snaží řešit doposud problémovou práci se složitými strukturami. Snaží se aplikovat obrácený styl (oproti předchozím stylům), kdy nositelem informace se stala data. Základním stavebním kamenem je zde objekt, který definuje jak atributy (data), tak metody nad těmito daty. Z těchto objektů je pak celý program skládán. Navenek se objekt tváří jako černá skříňka, ke kterému přistupujeme pomocí metod. Programové objekty jsou obrazem objektů reálného světa. Mimo jiné jsou zde zavedeny základní prvky OOP:

- **dědičnost** – Jeden objekt může dědit od jiného objektu a přebírá tak jeho schopnosti. Získané vlastnosti může přepisovat nebo k nim přidávat další. V některých programovacích jazycích lze dědit od více objektů najednou.
- **polymorfismus** – Pokud má více objektů stejné rozhraní, pracuje se s nimi stejným způsobem, ale každý se chová podle vlastní implementace, nebo lze na objektu volat jednu metodu s odlišnými parametry.
- **zapouzdření** – Objekt si schovává svoje vnitřní uspořádání. Pracujeme s ním pouze pomocí rozhraní, které zaručuje vždy konzistentní stav.

4.4 Komponentové programování

Hlavní devízou je zde práce se znovu použitelnými komponentami. Základem je OOP, ale komponenta je daleko komplexnější programový celek oproti objektům. Povětšinou také komponenta zapouzdřuje i více objektů, které dohromady plní určitou funkcionalitu. Tato komponenta je pak využívána uvnitř programu (program je sestaven z komponent). Mnohdy je využívána několika programy zároveň. Velice často jsou naprogramovány a odladěny třetí stranou. Komponentami se budeme zabývat v následující velké kapitole *Komponentová architektura*.

4.5 Generické programování

Jeho podstatou je oddělení algoritmu od datových typů. Řeší potřebu napsat jeden algoritmus pro více datových typů. Klade si za cíl implementovat algoritmy co nejobecněji tak, aby zůstaly efektivní i při dosazení konkrétních datových typů. Při tomto dosazení se z nich stává konkrétní kód. Parametrem je zde tedy konkrétní datový typ.

4.6 Aspektově orientované programování

Snaží se zdokonalit (doplnit) objektově orientované programování tak, že přidává aspekty. Do nich se snaží soustředit stále se opakující části. V případě OOP se i přes veškerou snahu objekty musí starat o věci, které nesouvisejí s konkrétním objektem. Příkladem může být zjišťování, zda je uživatel přihlášen. V případě aspektově orientovaného programování by se informace o přihlášení přesunula do aspektu. Ten by byl volán před metodami zajišťujícími práci v přihlášeném stavu pro různé objekty a jejich metody. Aspekt je tedy kód, který se volá před, po a nebo kolem určité metody nebo skupiny metod.

4.7 Agentově orientované programování

Jedná se o programové entity (agenty), které autonomně a kontinuálně plní zadané úkoly v definovaném prostředí. Agent je vybaven určitou dávkou inteligence, která je

použita k řešení problému. Spojením více agentů vzniká tzv. multiagentní systém, kde dochází ke spolupráci agentů k dosažení společného cíle.

4.8 Současnost - metodiky programování

Dnes je nejrozšířenější objektově orientované programování a je vyžadováno v nejrůznějších oblastech tvorby SW. Stále častěji se ale používají komponenty od třetích stran. Tyto odladěné komponenty značně urychlují vývoj aplikace. Je ale jasné, že se programování neustále vyvíjí a tyto dva druhy nepředstavují konečný směr tvorby SW.

5 Komponentová architektura

Základní snahou komponentové architektury je sestavení programu z co nejméně závislých komponent. Jednotlivé komponenty se mohou skládat z dalších komponent a toto dělení se může rekurzivně opakovat, dokud jednotlivé problémy nejsou snadno zvládnutelné. Tím se počáteční problém rozdělí na dílčí, lépe zvládnutelné části. Velice často se komponenty nedodávají ve zdrojovém tvaru, ale už jako vykonatelný (přeložený) kód. Sestavení z komponent nám ulehčuje spolupráci více vývojářů, postupnou tvorbu systému, snadnou aktualizaci atd. Komponenta se liší od třídy tím, že je zbavena vazby na okolí a tím se velice snadno opakovaně používá. Každá takováto komponenta má definováno rozhraní, pomocí kterého komunikuje s okolím. [27]

5.1 Komponenta

Je to samostatná softwarová jednotka, která je využívána v komponentovém modelu. Tato jednotka je definována jako opakovatelně použitelný blok programu. Dle programovacího jazyka může komponentou být balíček, modul nebo jmenný prostor. Komponenta je programována samostatně od zbytku aplikace a klade velký důraz na zapouzdření. Samotná implementace, tj.: data, funkce, objekty by měly zůstat skryty. Přístupovým bodem by mělo být rozhraní, přes které se s komponentou komunikuje. Tomuto principu se říká black-boxový model a je prakticky nedosažitelný (zkompilovaný kód lze také rozložit). [29] [28]

Autor komponenty:

- Dodržuje stanovené rozhraní.
- Přesně neví, jak a kdo bude jeho komponenty využívat.

Autor aplikace:

- Komunikuje přes stanovené rozhraní.
- Neví, kdo bude dodávat komponenty, ani jak přesně budou fungovat uvnitř.

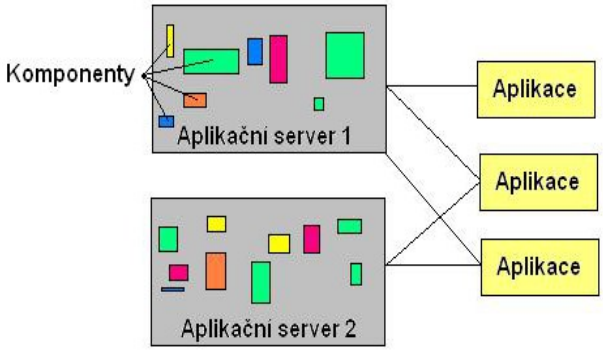
Komponenty v návrhu architektury: Velikou výhodou je možnost rozdělení do menších a lépe zvládnutelných částí. Při použití hierarchického skládání se pak problém postupně rozloží. Na nejvyšší úrovni je problém rozložen na několik málo spolupracujících komponent, které se dále dělí. Na nejnižší úrovni pak vzniknou komponenty, které jsou podobné třídám a jsou tak jednoduché, že je není potřeba dále dělit. Tato myšlenka rozhodně není inovativní a používá se již dlouhou dobu (např. modulární programování). Na rozdíl od modulů a tříd však nejsou komponenty zatíženy žádnými vazbami na prostředí pro implementaci programů. Hodí se tedy lépe pro dekompozici. [27]

Komponenty v implementaci programu: I zde je nejzásadnější možnost znovupoužití již hotových a odzkoušených komponent. Výhody znovupoužitelnosti:

- zvýšení výkonu programu (komponenty se neustále optimalizují – pro jejich časté použití se vyplatí věnovat více času zrychlení)
- zvýšení produktivity (není potřeba dokola psát ten samý kód)
- zvýšení kvality programu (komponenty jsou testovány na více místech a odhalení chyb je více pravděpodobné)

Výše zmíněné výhody nejsou uvedeny v absolutním slova smyslu a samozřejmě s sebou přinášejí určité problémy. I když lze předpokládat, že celková cena programu poskládaného z komponent bude nižší, některé částky porostou. Budou to zejména náklady dodavatelů komponent, kteří musí vyvíjet obecnější kód, nebo počáteční náklady výrobce programu, který musí nejprve pořídit komponenty. Tyto počáteční náklady může být obtížné obhájit, protože většina firem se snaží všechny náklady co nejvíce minimalizovat. [27]

Nejasné také určitě bude, zda komponentu opravdu znovu použijeme. Většina programátorů používá raději vlastní naprogramované kódy, než aby vyhledávala v cizích komponentách. Jedná se zde i o volnost programování, kdy se neradi vážeme na cizí komponenty. Navíc je problém s vyhledáním vhodné komponenty. Neexistuje ucelená metoda, jak třídit a podle čeho vyhledávat komponenty v archivech. [27]



Obrázek 9: Architektura komponentně orientovaných systémů

Požadavky na komponentu:

- Spolu s komponentou by měla být dodána dokumentace, která detailně popisuje danou komponentu.
- Kontrola vstupních parametrů.
- Důkladné testování (nevíme, k čemu přesně bude komponenta použita).
- Případné vracení dostatečných chybových zpráv.

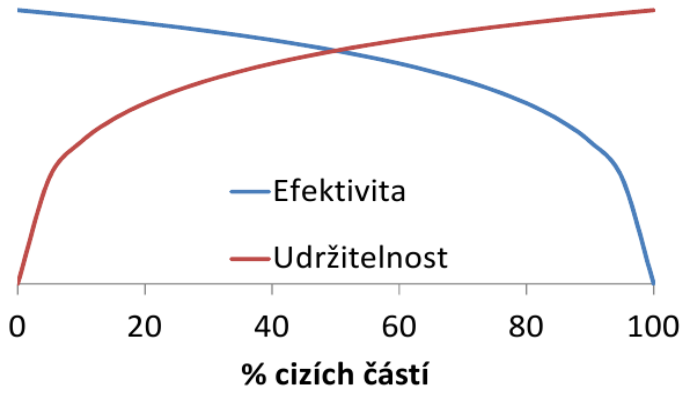
Výhody komponent:

- Urychlení vývoje aplikace.
- Větší stabilita v chování aplikace.
- Snížení ceny vývoje SW (znovupoužitelnost již odladěných komponent).

Nevýhody komponent:

- Režie při přechodu na komponentový systém.
- Komponenty třetích stran se vyvíjejí a musíme sledovat jejich vývoj.
- Pomalejší odezva při požadavku na změnu komponenty třetí strany, je-li možná!
- Složitější instalace, kdy nemáme jenom jeden exe soubor, ale i další dll soubory.

Následující graf 8 ukazuje, že není dobré se snažit za každou cenu aplikaci poskládat z existujících komponent. Sice se maximálně urychlí vývoj aplikace, kdy stačí doprogramovat jenom vazby mezi jednotlivými komponentami, ale je složitá údržba programu. Je velice obtížné požadovat jakoukoliv změnu v již vytvořené komponentě. Pokud se podaří přemluvit výrobce komponenty k aktualizaci, je to pak velice zdlouhavý proces počínající specifikováním nových požadavků a končící testováním. SW poskládaný z komponent nebude nikdy tak efektivní jako kód psaný přímo pro danou aplikaci. [39]



Graf 8: Množství cizích komponent v aplikaci [39]

5.1.1 Komponenty v UML

Komponenty se značí jako obdélník. V něm je pak značka komponenty v pravém horním rohu a stereotyp <<component>>. Příklad komponenty vidíme na obrázku 10. [2]

Obrázek 10: Příklad komponenty v UML

Pokud je potřeba ukázat vnitřní uspořádání komponenty, může se použít styl tzv. *bílé skříňky*. Zde uvádíme zpřístupněná a požadovaná rozhraní nebo nejrůznější artefakty. Příklad bílé skříňky je na obrázku 11. [2]

Obrázek 11: Příklad komponenty jako bílé skříňky v UML

5.2 Komponentové rozhraní (Interface)

Základní myšlenkou komponentového programování je oddělení jednotlivých komponent (Isolation of Components). Jediná možná cesta, jak komunikovat s komponentou, je přes dané rozhraní, a jiná komunikace není možná. Implementace je tedy skrytá za rozhraní. Samotné rozhraní popisuje, co komponenta potřebuje ke svému korektnímu fungování a co naopak poskytuje volajícímu subjektu. Rozhraní komponenty může být popsáno buď programovacím jazykem nebo jazykem *Interface Definiton Language* (IDL). Při každé změně komponenty (při vydání nové verze) je nutné zkontrolovat rozhraní. Většinou se pro přehlednost zavádí tzv. *verzování rozhraní*.

Při návrhu rozhraní by se mělo postupovat tak, aby jednotlivé vazby na okolní entity byly v co nejmenší míře. Vzájemné vazby brání snadnému pochopení a systém se stává obtížněji udržovatelný. Pokud bude mít rozhraní každá metoda v programu, je to nanejvýše nevhodné a zcela zbytečné. Rozhraní by mělo být pouze u těch částí, které se v budoucnu mohou měnit. Při návrhu je tedy nutné dávat přednost správnému chodu aplikace před maximální flexibilitou. [28]

5.2.1 IDL (Interface Definiton Language)

Tento jazyk specifikuje, které metody jsou přístupné klientům komponenty. Aby klient věděl, které metody to jsou, stačí mu IDL definice objektu, nepotřebuje tedy znát nic z implementace komponenty. Jinými slovy je IDL jazyk pro definici jednotlivých rozhraní. IDL je nezávislý na jazyku a umožňuje implementaci klienta a serveru v libovolném jazyce, který podporuje mapování jazyka (language maping). Jedná se o převod konstrukcí jazyka IDL na konstrukce pro daný programovací jazyk. Například typ long v jazyce IDL se mapuje na long v C++ a int v jazyku Java. Existuje velké množství těchto mapování do různých jazyků (C, C++, Java, Smalltalk atd.). Při návrhu se nejprve napíše definice rozhraní v jazyku IDL. V druhém kroku se vybere konkrétní jazyk a IDL kompilátor vygeneruje část zdrojového kódu, který musí programátor dopsat do finální podoby.

5.2.2 Publikované rozhraní

U komponent se jedná zásadně o publikované rozhraní. Takto označené rozhraní se po zveřejnění už nemůže měnit. A to ani tak, že bychom ostatním vývojářům změnu oznámili. Mezi nejznámější případy špatně navrženého publikovaného rozhraní patří metody `resume()`, `stop()`, `destroy()` ve třídě *Thread* jazyku Java. Ihned po vydání se zjistilo, že tyto metody jsou špatně navržené a prakticky nepoužitelné – vzápětí byly prohlášeny za zavřené. Protože ale bylo rozhraní již publikováno, zůstávají jeho součástí dodnes. [1]

5.2.3 Interní rozhraní

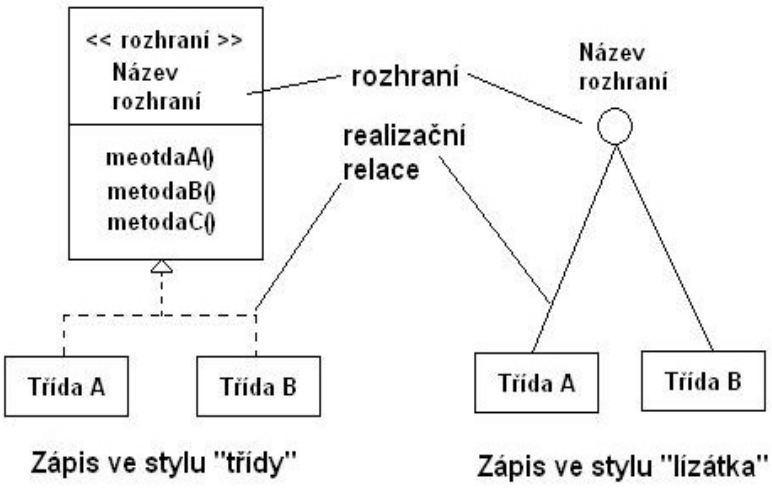
Je protiklad publikovaného rozhraní. Jedná se o rozhraní dostupné pouze třídám, které má autor přímo či nepřímo dostupné a může je měnit. I u interního rozhraní bychom ale měli provádět změny minimálně a při návrhu rozhraní dávat velký pozor! [1]

5.2.4 Rozhraní v UML

Vše, co používá dané rozhraní, musí dodržovat kontrakt definovaný tímto rozhraním. Kontrakt se stará o to, co je vyžadováno a co nabízí jednotlivé strany. Rozhraní by tedy mělo obsahovat:

- signaturu operace (název, typy argumentů, návratový typ)
- sémantiku operace (lze použít prostý text)
- název a typ atributů

V UML existují dva způsoby, jak popsat rozhraní. První je notace ve stylu *třídy*, kde se mohou zobrazit jednotlivé operace. Druhý způsob je pak stručnější a jedná se o notaci ve stylu *lízátka*. Obě varianty jsou uvedeny na obrázku 12. [2]



Obrázek 12: Popis rozhraní v UML

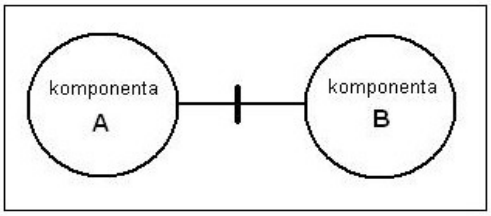
Na dalším obrázku je znázorněna situace, kdy třída vyžaduje nějaké rozhraní. V tomto případě třída auto potřebuje služby definované v rozhraní *poháněj*. Naopak třída motor tyto služby nabízí. [2]

Obrázek 13: Sestavení rozhraní v UML

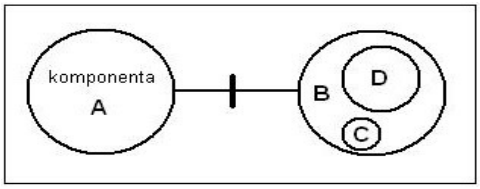
5.3 Formy kompozice

Komponenty mohou spolupracovat klasicky, kdy každá plní svůj úkol. Druhý typ spolupráce je, že jedna komponenta je zanořena do druhé a vnější komponenta bez vnitřní nemůže fungovat. [28]

Komponenty mohou být spojeny:



Obrázek 14: Klasické spojení komponent



Obrázek 15: Komponenty zanořeny do sebe (hierarchie)

5.4 Komponentový model

Komponentový model určuje některé specifikace. Sem patří zejména, jak vytvořit komponentu a co všechno musí obsahovat. Rovněž určuje, jakým způsobem spolu dvě komponenty komunikují. Dále vymezuje povolené datové typy, požadované služby, styl dokumentace atd. Model je vlastně soubor různých pravidel, která musí komponenty dodržovat, aby byly vzájemně kompatibilní. [28] [29]

5.5 Komponentový framework

Daný framework nabízí a poskytuje velké množství služeb, které se starají o běh, komunikaci komponent a další služby. Komponentový framework by se dal vzdáleně přirovnat k OS s tím, že pracuje na mnohem vyšší úrovni abstrakce. Je to vlastně

implementace komponentového modelu, kdy jeden framework může implementovat více takovýchto modelů, pokud mají vzájemně slučitelné vlastnosti. Na druhou stranu může být k jednomu modelu vytvořeno více frameworků. [28] [29]

5.6 Proces vývoje komponent

Vývoj komponenty se dá zachytit do několika základních fází. U různých modelů se pak tyto fáze mohou vytrácet nebo mohou naopak klást velký důraz na patřičný krok. Základní fáze vývoje jsou: [28]

5.6.1 Návrhová fáze

Tato fáze (i když zde je uvedena jako první) navazuje na sběr požadavků a následnou tvorbu analytického modelu. Zde se přesně určí implementace funkcí uvedených v analýze. Dále se aplikace rozdělí na několik komponent (logických celků), které mají přesně vymezenou funkcionalitu. Pro každou komponentu se definují služby, které potřebuje ke své práci a které komponenta poskytuje. Postup dekompozice se opakuje tak dlouho, dokud nemáme lehce naprogramovatelné a přehledné komponenty. Z těchto komponent a z komponent třetích stran pak v pozdějších fázích poskládáme výslednou aplikaci.

5.6.2 Vývojová fáze

V této fázi se vytváří kód pro navržené komponenty. Tedy převádí se návrhový model do spustitelného kódu. Některé nástroje umožňují generování kódu na základě popisu komponenty a zbytek je doprogramován. Zkompilovaný kód pak tvoří komponentu, která je dále používána.

5.6.3 Fáze sestavení

Sestavení se dá chápat jako strom, kde kořenem je sestavená výsledná aplikace. Listy jsou pak jednotlivé naprogramované nebo znovu použité komponenty. Jednotlivé větve (komponenty) se pak dle návrhu spojují až ke kořenu (výsledná aplikace). V této fázi se přidávají defaultní konfigurační hodnoty, které bývají nejčastěji zadány v XML.

5.6.4 Fáze testování

Softwarové komponenty mohou být velice složité a při propojování komponent mohou vznikat chyby. Proto je nutné provést důkladné testování. To by mělo probíhat na více úrovních:

- Testování komponenty bez ostatních vazeb. Nejčastěji se testuje vytvořením testovacího kódu, který zkoumá, zda-li je správný výstup při určitém vstupu.

- Testování rozhraní komponenty. Musí se prověřit, zda jsou všechny požadované funkce dostupné přes rozhraní.
- Testování integrace komponenty. Testuje se výsledný systém jako celek. Je to nejsložitější fáze a velice špatně se odhalují chyby. Jedním z postupů je postupné přidávání komponent, kdy se po úspěšném testu přidává další komponenta.

5.6.5 Fáze nasazení

Zde se komponenty rozdělí do deployment units podle toho, na kterém serveru mají běžet. Deployment units jsou po jedné na každém serveru. V případě jednoho serveru se všechny komponenty nahrají do jedné deployment unit. Nejmodernější technologie dokáží pro jednotlivé komponenty sdílet různé deployment units, a tím přerozdělovat zátěž rovnoměrně na všechny servery.

5.6.6 Certifikace

Třetí důvěryhodná strana vydá certifikát. Tento certifikát se může udělit frameworku, komponentě nebo systému (složen z komponent a frameworku).

6 Komponentové systémy

Tvorba komponenty probíhá dle určitého standardu, který je popsán v některé z komponentových architektur. Mezi nejznámější zástupce patří CORBA, Enterprise Java Beans a COM. V krátkosti zde budou uvedeny.

6.1 CORBA (Common Object Request Broker Architecture)

Jedná se o architekturu distribuovaných objektů vyvinutou konsorciem OMG (Object Management Group). Distribuované systémy spoléhají na definované rozhraní komponent. A právě CORBA definuje mechanizmus rozhraní komponent. Jako další definuje nástroje implementace samotných komponent ve zvoleném jazyce. Tato architektura je tedy nezávislá na platformě a na programovacím jazyku. První verze vznikla v roce 1991 a definovala hlavně jazyk IDL a aplikační programové rozhraní. To umožňuje, aby server nebo klient volal ORB. [40]

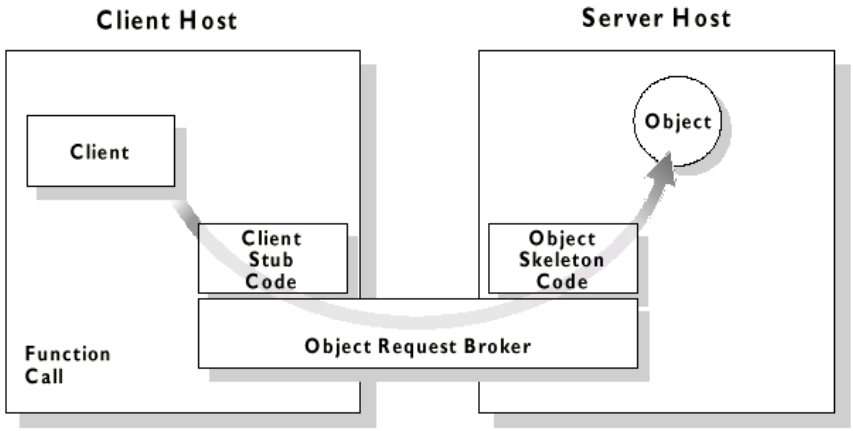
ORB (Object Request Broker) je základním kámenem celé technologie CORBA. Stará se o navazování spojení mezi jednotlivými komponenty aplikace a předávání požadavků na jednotlivá rozhraní. Jednotlivé požadavky obsahují:

- adresu volaného objektu
- požadované operace
- parametry operací

Ve verzi druhé byl navíc definován protokol IIOP (Internet Inter-Orb Protocol), který přinesl úplnou nezávislost a umožnil komunikaci dvou ORB od různých výrobců. CORBA je na platformě nezávislá. To znamená, že objekty mohou být vytvářeny v libovolném programovacím jazyku. Podporuje například jazyky Java, C++, Smalltalk.

Je zde umožněno volání metod vzdálených objektů bez ohledu na jejich umístění (mohou být kdekoliv na síti). Požadovaný objekt je lokalizován pomocí jednoznačného identifikátoru (reference). Tento identifikátor získáme buď jako výsledek nějaké operace a nebo ho explicitně definujeme. [40]

Při přeložení například do C++ jsou vygenerovány dvě třídy. Na straně serveru je to skeleton, do kterého se doplňuje kód poskytovaných operací. Kód se nejčastěji doplňuje v podobě odvozené třídy. Druhá třída je na klientské straně a jedná se o proxy objekt. Ten má na starosti transparentní volání metod prostřednictvím ORB. Klient musí tedy přiřadit identifikátor tomuto proxy objektu a pak může začít používat jeho metody. Komunikace probíhá tak, že metoda proxy objektu předá všechny potřebné informace místnímu ORB, který vše přepošle na serverový ORB. Takovýto server většinou pasivně čeká na příchod požadavku. Mezi oběma servery ORB probíhá komunikace v nezávislém formátu a kód je přeložen z (na) cílovou platformu. [40]



Obrázek 16: Ukázka komunikace přes ORB [40]

CORBA má tu výhodou, že umožňuje zajistit komunikaci mezi objekty napsanými v různých programovacích jazycích a běžících na různých počítačích a platformách (platformě nezávislý). Umožňuje vytvářet velmi propracované systémy, které využívají pro jednotlivé činnosti nejvhodnější platformu. Nevýhodou je pak složitost.

6.2 COM (Component Object Model)

Vytvořeno firmou Microsoft pro vytváření softwarových komponent. Vznikla z technologie OLE 2 v roce 1994. Je to vlastně binární standard a tím se dosahuje nezávislosti na implementačním programovacím jazyku. Takto napsané knihovny mají i po zkompilování pevně danou strukturu, a proto jsou přenosné i do jiných jazyků. Technologie COM je určená pro znovupoužití již existujících komponent. Nejčastěji je nalezneme ve formě *dll* a *exe*.

Technologie COM dodržuje následující pravidla:

- **Modularita:** Komponenta má jasně dané rozhraní, pomocí kterého se s ní pracuje. Toto rozhraní musí zůstat vždy stejné a v případě potřeby se mění jenom samotná komponenta, případně lze do rozhraní metody jenom přidávat. Tím je zajištěno, že program bude fungovat stejně i poté, co uživatel aktualizuje knihovny.
- **Univerzálnost:** COM je vlastně standard, který se používá pro zkompilované binární soubory. Tyto komponenty můžeme použít ve všech jazycích, které umí vytvořit kód v tomto standardu.
- **Správa paměti:** Životní cyklus komponent je určován referencemi, které směřují na daný objekt. Při používání komponent může počet referencí stoupat nebo klesat. Pokud však klesne na 0, komponenta se sama uvolní z paměti.

Technologie COM se stejně jako CORBA stará o komunikaci mezi klientem (libovolná aplikace) a volanou komponentou (nebo-li serverem COM). Server může být implementován dvěma způsoby:

- **In-process:** Nejčastěji jsou ve formě knihovny *dll*, a aplikace si je spustí ve vlastním adresovém prostoru. Následně volá metody rozhraní objektu, které se nacházejí na stejné adrese jako samotná aplikace.
- **Out-of-process:** Většinou jsou spouštěny jako soubory *exe*. Ty po spuštění vytvoří COM objekt, který běží ve vlastním adresovém prostoru. Tato kategorie se dá dělit na dva druhy podle umístění serveru a klienta – na lokální, kdy server a klient jsou na stejném PC, a vzdálené, kdy běží na různých PC.

Výhodou je, že COM a jeho následníci DCOM a COM+ jsou úzce spjatí s výrobky firmy Microsoft. Tím můžeme například ve svých aplikacích využívat nástroje Microsoft Office nebo další produkty této firmy. Nevýhodou je slabá podpora jiných platforem.

6.3 DCOM (Distributed Component Object Model)

Je to rozšíření technologie COM o komunikaci komponent na vzdálené systémy (distribuované zpracování). DCOM využívá technologii RPC (Remote Procedure Calling). Každé rozhraní má jednoznačný identifikátor. Komponenta může obsahovat i více rozhraní pro zajištění zpětné kompatibility. Aplikace kontaktuje pomocí identifikátoru příslušné rozhraní a vyžádá si jeho popis. Poté už následuje volání konkrétních metod.

6.4 COM+

Služba je založená na modelu COM a byla vydána v roce 2000. Jedná se o spojení standardní technologie (D)COM a aplikace hostitele MTS (Microsoft Transaction Server). Je to označení několika komponent, které jsou zahrnuty v operačním systému windows 2000/XP. Oproti COM obsahuje:

- vylepšenou práci s vlákny
- lepší podporu databází
- vyšší bezpečnost

6.5 .NET

Jedná se o nejnovější platformu od firmy Microsoft, která se snaží mimo jiné podporovat komponentový vývoj. Framework .NET poskytuje robustní a bezpečné prostředí pro běh aplikací. Je složen z velkého množství knihoven a kompletního běhového prostředí pro spouštění a běh aplikací. Spolupracuje se všemi programovacími jazyky pro tuto platformu. Komunikace mezi klientem a komponentou probíhá přímo, neboť třídy a rozhraní jsou dostupné jako zdrojový kód. Některé další novinky:

- Sjednocení všech chyb, které jsou řešeny formou výjimek.
- .NET framework se sám stará o systémové zdroje. Pokud nejsou využity, dojde k jejich uvolnění (navrácení).

- Zavádí společný typový systém pro všechny jazyky.

Princip je v překládání do společného jazyka. Řízený kód (kód napsaný ve vyšším programovacím jazyce, jako například C#, C++) se zkompiluje do binárního formátu (sestavení). Jazyk, do kterého se překládá, se jmenuje CIL (Common Intermediate Language). Je to jazyk nezávislý na procesoru a platformě. Díky tomu lze jednu aplikaci napsat několika programovacími jazyky.

U platformy .NET je zajištěna plná kompatibilita spolupráce se starší technologií COM. Tato kompatibilita je obousměrná. Lze využívat COM komponenty v .NET programech a obráceně lze využívat .NET komponenty v COM aplikacích.

6.6 Architektura Enterprise Java Beans (EJB)

Praktická část diplomové práce ukazuje konkrétní práci s EJB. Na tomto místě tedy popsány uvedeny teoretické základy této technologie. Konkrétní příklady budou uvedeny spolu s popisem frameworku Fyx.

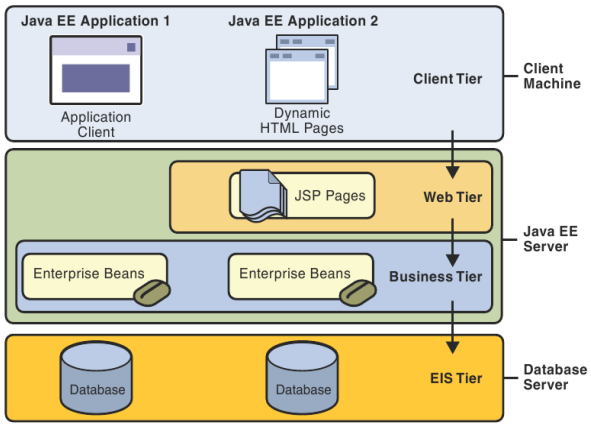
EJB vytvořila firma Sun Microsystems. Jde o serverové komponenty umožňující modulární tvorbu podnikových aplikací. Hlavním cílem EJB je oddělit jednotlivé vrstvy: business logiku, prezentační logiku a perzistentní vrstvu. EJB je tedy implementací aplikační logiky. Dále zajišťuje podporu ostatních technologií (JNDI, CORBA atd.).

6.6.1 JEE

V této kapitole bude nastíněna architekturu JEE (dříve J2EE), protože její součástí jsou právě Enterprise Java Beans. Tato architektura rozšiřuje standardní JSE (Java Standard Edition). Jedná se především o nástroje pro práci s více vrstvami aplikace, jako aplikační server, webový server, databázový server atd. JEE poskytuje komponentový přístup jak k tvorbě, tak i návrhu a nasazení. Dále umožňuje tvorbu webových aplikací s využitím technologií, jako např.: JSF, JSP nebo Java Servlets. [41]

Architektura specifikuje tři vrstvy:

- **Prezentační vrstva** – Zajišťuje komunikaci aplikace s uživatelem. Příkladem prezentační vrstvy může být klasická GUI aplikace, webová aplikace nebo mobilní aplikace. Pro webové rozhraní se používá například JSP nebo JSF. Pokud je aplikace součástí nějakého portálu, používá se Java Portlets. Pokud máme klasickou Java aplikaci, nejčastěji se využívá Java Web Start.
- **Aplikační vrstva** – Jedná se o vlastní funkcionalitu programu. Bývá použit framework, který se stará o další služby: vzdálený přístup, persistence, transakce, atd. Příkladem může být právě EJB.
- **Perzistentní vrstva** – Má za úkol ukládání dat do perzistentního úložiště (objektová nebo relační databáze). Nejčastěji se pro komunikaci využívá JDBC.



Obrázek 17: Vícevrstvá architektura [41]

EJB aplikace se po odladění zabalí do ear archivu. Jedná se o kompresi jar, ale pro odlišení (že se jedná o EJB aplikaci) se používá koncovka ear. Takto zabalený soubor má danou strukturu:

- lib – knihovny
- war soubor – obsahuje webového klienta
- jar soubor – obsahuje beany

6.6.2 Kontejnery JEE

Jedná se o dedikovaný virtuální prostor, který je umístěn na aplikačním serveru. Do tohoto prostoru se nahrávají jednotlivé EJB komponenty, které mají být spuštěny. Pro každou komponentu se nastavují její specifické vlastnosti, jako např. zabezpečení, transakce atd.

Kontejner se stará o prostředí pro běh komponent a zajišťuje obsluhu rozhraní těchto komponent. Existuje i tzv. embedded kontejner, který umožňuje spouštění EJB aplikace v prostředí Java Standard Edition. Výhodou je zejména snadné testování a ladění. Nevýhodou je, že podporuje jenom určitou podmnožinu klasického EJB. [37]

6.6.3 Typy EJB

Do kontejneru lze nahrávat několik druhů Enterprise Java Beanů. Zde je jejich výčet s krátkým popisem:

a) Stateless Session Beans (bezstavové beany)

Neudržují stav pro konkrétního klienta mezi jeho jednotlivými požadavky. Při každém požadavku je serverem přidělena vždy nová a samostatná instance. Tím pádem jsou bezstavové beany vláknově bezpečné. Bezstavové beany se obvykle shromažďují v poolu, odkud jsou odebrány pro vyřízení klientského požadavku a následně jsou vráceny zpět. Anotace pro tento typ beany je @Stateless.

Životní cyklus: Nejprve je vytvořena nová instance, která je zařazena do poolu na serveru. Pokud přijde klientský požadavek, instance ho obslouží a pokud není vyvrhnuta výjimka, vrátí se opět do poolu, kde čeká na další zavolání. Pool jde samozřejmě dle potřeby zvětšovat a zmenšovat. [37]

b) Stateful Session Beans (stavové beany)

Jedná se o objekty, které si udržují svůj stav v rámci jedné session. V paměti je vytvořena unikátní instance pro každého nového klienta, která uchovává data mezi jednotlivými voláními od klienta. Tato třída je anotována @Stateful.

Životní cyklus: Ten je podobný jako u nestavové beany, je zde však možnost odložení do pasivního módu, pokud server vyčerpá všechnu svoji volnou paměť. V tom případě se volají metody `@PrePasivate()` resp. `@PostActivate()` pro obnovení beanu. Bean může také zaniknout, pokud vyprší jeho timeout (čas životnosti) nebo nastane-li výjimka. [37]

c) Singelton Session Beans

Jedná se o objekty, které globálně sdílejí svůj stav. Synchronizovaný přístup se řídí pomocí kontejneru nebo samotným beanem. Tato třída se uvozuje anotací @Singelton.

d) Message Driven Bean

Nebo-li zprávami řízené beany. Jedná se o speciální asynchronní chování JMS (Java Message Service), které nevyžaduje okamžitou odpověď.

6.7 Ostatní

Kromě zde výše uvedených hlavních komponentových modelů existuje i celá řada dalších. Zde je krátký výčet:

- **Kparts** – Jedná se o grafické komponenty pro prostředí KDE. Jsou založeny na modelu CORBA, ale upraveny pro vyšší rychlost odezvy. Podobným modelem je Bonobo pro prostředí GNOME.
- **ActiveX** – Je navržen jako nadmnožina technologie COM. ActiveX je určen pro realizaci prvků s vizuálním rozhraním. Tyto prvky mohou umožňovat klasické zobrazení nebo zobrazení se zpětnou vazbu od uživatele.
- **JavaBeans** - S Enterprise Java Beans má velice málo společného. Jedná se také o komponentovou architekturu, ale komponenty zde nejsou instalovatelné a využívají se jenom při návrhu a tvorbě samotné aplikace. Na rozdíl od EJB (business vrstva) patří JavaBeans do prezentační vrstvy.

Část II.

Fyx framework

Vznik frameworku Fyx byl podnícen zakázkou na webový portál týkající se tématu makrobiotiky. Má se jednat o zaštítění všech menších makrobiotických klubů. Tento rozsáhlý portál v sobě bude obsahovat různé funkce jako např. recepty, správu konferencí, místa makrobiotického ubytování, zdravotní poradnu, e-shop atd.

Požadavky kladené zadavatelem na tento systém jsou samozřejmě rychlost a nulová chybovost. Z hlediska návrhu je ale důležité, aby tento systém byl poskládán z menších částí. Zadavatel si nejprve přeje spustit menší stránky, které bude postupně doplňovat o různé další funkcionality. Hlavním požadavkem z hlediska autorů je tedy také přehlednost kódu a snadná rozšiřitelnost.

Protože budou ve Fyxu vyvíjeny i jiné stránky, musí být navržen univerzálně. Základem bude tudíž malé a rychlé jádro, které se bude vhodně doplňovat dalšími komponenty. Většina kódu i u zcela odlišných stránek se opakuje. Snad všechny stránky zahrnují práci s fotogalerií, správu textů nebo např. diskuzi. K tomu všemu se nejlépe hodí komponentová technologie.

7 Analýza

Před popsáním samotného frameworku bude provedena analýza. Ta by měla zmapovat volbu programovacího jazyka. Dále se zaměří na existující podobné aplikace.

7.1 PHP versus Java

Nejpoužívanějším jazykem na webu je bezesporu PHP. Jeho jednoduchost a dlouholeté používání z něj vytvořili odladěný a oblíbený jazyk. Většina placených i neplacených webhostingových serverů poskytuje právě PHP. Přesto existují i jiné, modernější jazyky. Příkladem může být jazyk Java. Jeho výhodou je čistší a přehlednější kód. Zároveň se při deployi komponenty na server kód parsuje a kompiluje a je tedy rychlejší než PHP. To tyto úkony provádí při každém načtení (vyjma použití cache).

Výhody PHP:

- lepší pro malé weby
- jednodušší
- maximální podpora od webhostingů

Výhody Java:

- kvalitnější kód
- rychlejší

Pro projekt byla zvolena platforma Java EE. K výše uvedeným důvodům se též přidává to, že framework je také koncipován pro rozsáhlé a vytížené aplikace.

7.2 Existující aplikace na trhu

Na internetu existují podobné projekty na architektuře Java EE. Většinou se ale jedná o ECM (systém pro správu podnikového obsahu) aplikace. Tyto aplikace tedy v sobě implementují zobrazování internetové prezentace, ale i další vlastnosti jako Subversion, předávání dokumentů dalším pracovníkům atd. Mezi tyto produkty patří Alfresco, OpenEdit, Xinco atd. Nejedná se tudíž o čistě webové frameworky.

8 Postupný vývoj

Od začátku vývoje se projekt velmi výrazně měnil. Přestože je tato podkapitola krátká, vývoj a psaní kódu, který nakonec nebyl použit, zabral 80% času. Trend byl spíše zjednodušujícího charakteru. Původně velká a vše obsahující aplikace se postupem času změnila na co nejpřímější řešení s důrazem na jednoduchost a rychlost. Následuje výčet „výrazných“ postupných iterací. Každá takováto iterace byla přitom vyvíjena i několik týdnů, než se změnila koncepce.

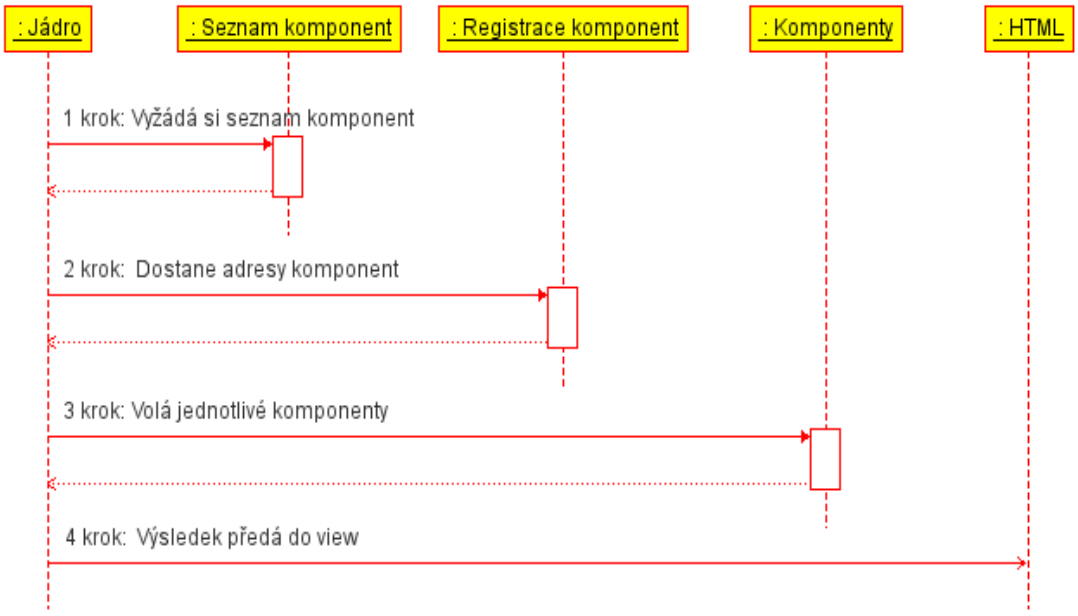
8.1 Verze první

Nebo-li počáteční návrh. Ten vypadal tak, že v něm byly shrnuty všechny požadavky (nápady) a měl být vyvíjen v Java SE. Princip spočíval v komunikaci jádra, které mělo zajišťovat veškerou komunikaci.

Po obdržení konkrétní adresy se načetly moduly, které byly potřebné pro danou stránku. Dále se měla zjišťovat adresa příslušných modulů. V případě shodných modulů (třeba na různých JVM) by se vrátil méně vytížený modul. To by se dalo určovat z některého parametru jako ping, vytíženost CPU, počet užití komponenty atd. Následně by se volaly jednotlivé moduly, kterým by navíc přes jádro byla zpřístupněna databáze. V konečné fázi by jednotlivé výsledky z modulů byly poslány do výsledné stránky.

Nevýhody:

- Velice složitá komunikace pro jádro.
- Složité zjišťování vytíženosti jednotlivých modulů.



Obrázek 18: Schéma první verze Fyx frameworku

8.2 Verze druhá

Nebo-li přechod na JEE. Druhá verze již přešla z Javy Standard Edition na Javu Enterprise Edition. Důvodem byla podpora komponent při návrhu i realizaci a také přechod na novější technologii. V této verzi se již nemusela řešit komunikace komponent, která je zde vyřešena implicitně. Jednotlivé komponenty se připojují přes Library a Enterprise Project. Automaticky je tím získán přístup k nové komponentě a jejím metodám. Navíc byl jako aplikační server použit Glassfish 3.1. Ten zahrnuje práci s různými Java Virtual Machine (JVM) a další technologie. Protože tento server také podporuje automatické vytížení různých PC (přidají se do jednoho clusteru), z návrhu se odstranila nutnost použití více stejných komponent. Tím elegantně odpadá složité zjišťování vytíženosti jednotlivých komponent a jejich volání. Tato starost byla předána odpovídající vrstvě, která je na tyto úkony specializována.

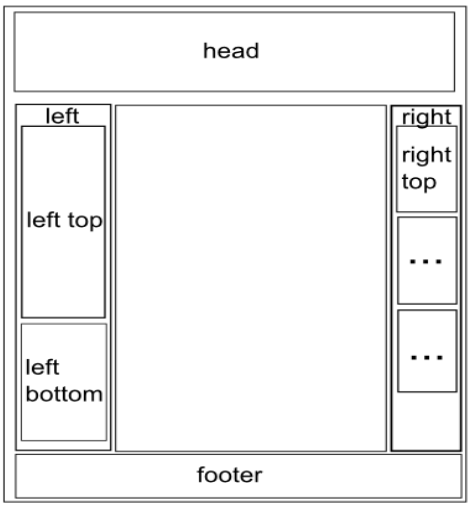
Pro přístup k databázi se začalo využívat JPA, což je nová technologie zaštiťující veškeré operace s databází. Patří sem tvorba entit, objektově relační mapování, tvorba sequencí atd.

Výhody:

- Odpadá zjišťování vytíženosti jednotlivých komponent.
- Odpadá navazování spojení s komponentami.
- Zjednodušení práce s databází využitím služeb JPA.

8.3 Verze třetí

Nebo-li základem je Webová aplikace a jádro. Do této doby byla zobrazovaná stránka poskládána z různých oblastí (například `<DIV>`) a ke každé oblasti byla volána příslušná komponenta. Tyto rámy by se při naplnění zobrazily na příslušném místě. Ukázka zobrazení stránky přes rámy je vidět na následujícím obrázku.



Obrázek 19: Zobrazení stránky pomocí rámů

Toto se ukázalo jako ne zcela vhodné řešení, protože většina stránek je složena právě z jedné komponenty, resp. z její jedné metody (přidání uživatele, přehled uživatelů, ...). Proto je vhodnější, když místo univerzální stránky je použita konkrétní stránka pro každou jednotlivou akci. V případě složené stránky (například diskuse pod článkem) je hlavní komponenta článek a dolů se vloží závislá komponenta diskuse.

Při zakládání nového projektu jsou dvě části jako jádro (komponenta FyxCore a webová aplikace FyxWeb) přidány do Enterprise aplikace. Každá další přidávaná komponenta se musí přidat jednak do Enterprise aplikace a grafické rozhraní v podobě xhtml souborů se vkládá do webové aplikace. Každá komponenta si tak může ovládat zcela svoji stránku nebo jenom její část. Postup přidání je detailněji popsán v kapitole *Komponenty Fyx*.

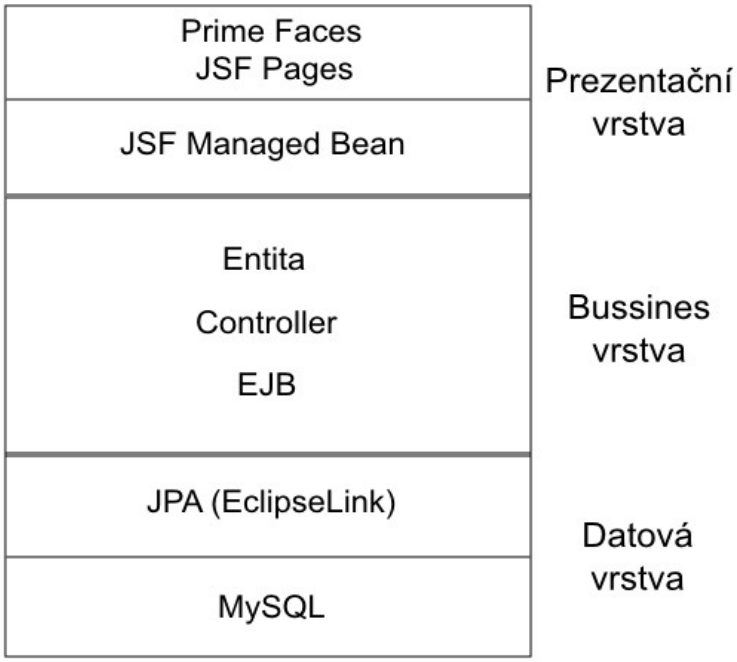
8.4 Verze čtvrtá

Nebo-li co nejméně návazností. Poslední změnou v konceptu bylo odstranění co nejvíce návazností komponent na framework. Původně bylo zamýšleno, že komponenty budou muset obsahovat různé metody, aby byly kompatibilní s jádrem. Například v první verzi by musely obsahovat metody pro zjištění aktuální vytíženosti. To ale není zcela univerzální a mizí zde možnost použití různých komponent.

Jedinou návazností tedy zůstává, kdo smí danou komponentu používat, pokud je takováto komponenta a její metoda zahrnuta v menu. Pokud v menu zahrnuta není, je zcela libovolná implementace. To se hodí při použití již existujících komponent a odpadá nutnost programovat speciální komponenty speciálně pro tento framework. Obrácenou výhodou je, že se dají komponenty využívat i v jiných projektech.

9 Jádro projektu

Po všech změnách je jádro ustáleno v první verzi Release Candidate. Jádro by mělo být maximálně jednoduché. Jednak je tím eliminován výskyt chyb, ale hlavním důvodem je rychlost. Nemusí se načítat nepotřebné moduly a připojí se jenom ty potřebné. Jádro tedy obsahuje tři základní moduly: mula (MUlti LAnguage), linker (správce odkazů), secure (spravuje uživatelské role a jejich práva). Kromě těchto základních modulů je v jádře obsažena přepravka (idiom návrhového vzoru). Ta má za úkol přenášet informace typu session, aktuální jazyk, aktuální uživatel atd. Jádro obsahuje spoustu dalších tříd starajících se o naplnění a přenášení systémových informací, metody pro refresh a redirect stránky, správu konstant atd. Do budoucna se plánuje ještě rozšíření jádra o logování všech chyb. Souborově by se dalo rozdělit na dvě části. Jedna je uložena v EJB modulu, kde je uložena business logika. Druhá část je pak uložena ve Web Application a obsahuje managed beany a JSF soubory. Aplikace je napsána ve třívrstvém modelu. Následuje popis, jak jsou jednotlivé vrstvy řešeny:

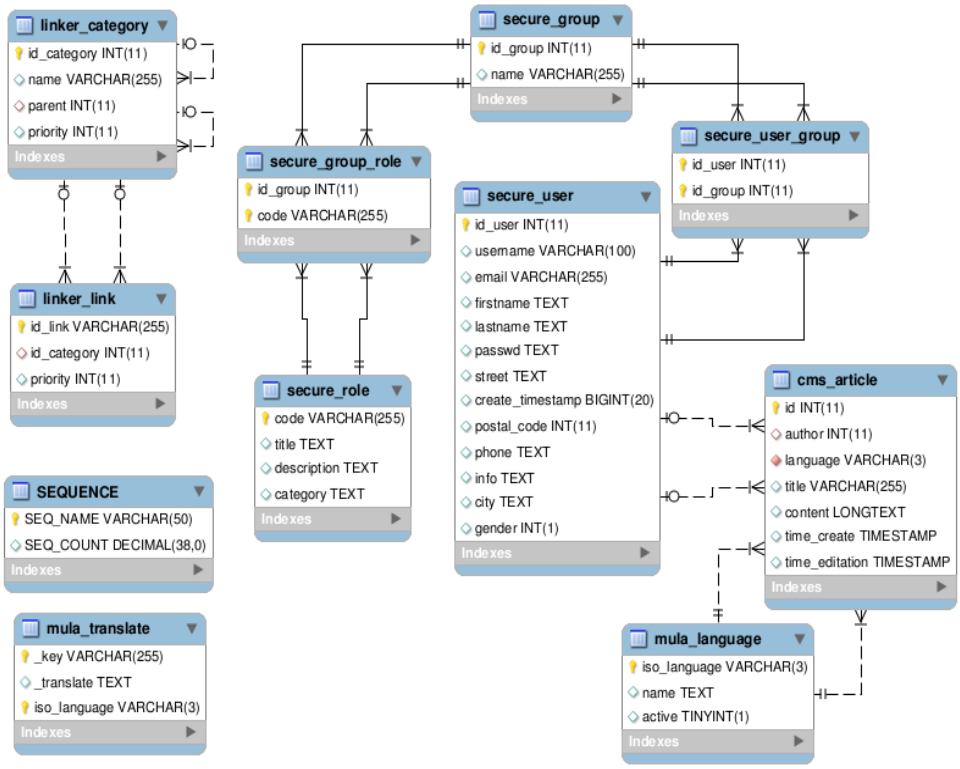


Obrázek 20: Třívrstvá architektura projektu

9.1 Komunikace s databází

Jak je vidět na obrázku 20, *Datová vrstva* se dělí na dvě části. Úplně spodní vrstvu zajišťuje již popisovaná databáze MySQL. Tato databáze v testech vyšla jako nejvhodnější pro vytvářený projekt. Další vrstvou je framework JPA od Eclipsu – EclipseLink. Přináší několik zásadních výhod, jako je objektově relační mapování nebo automatické hlídání transakcí. Řídí se v souboru persistence.xml. Zde můžeme upravovat chování JPA:

- Nastavení datového zdroje, což je připojení k databázi.
- Strategie generování tabulek – v případě, že není tabulka v databázi a existuje příslušná datová entita, je na výběr ze dvou možností. První je, že se tabulka vytvoří (create), nebo druhou, kdy se nevytvoří a aplikace hodí výjimku.
- Dá se nastavit, zda se mají používat transakce. Pokud je nějaká funkce v transakci a ke konci vznikne výjimka, celá funkce se vrátí zpět do konzistentního stavu.

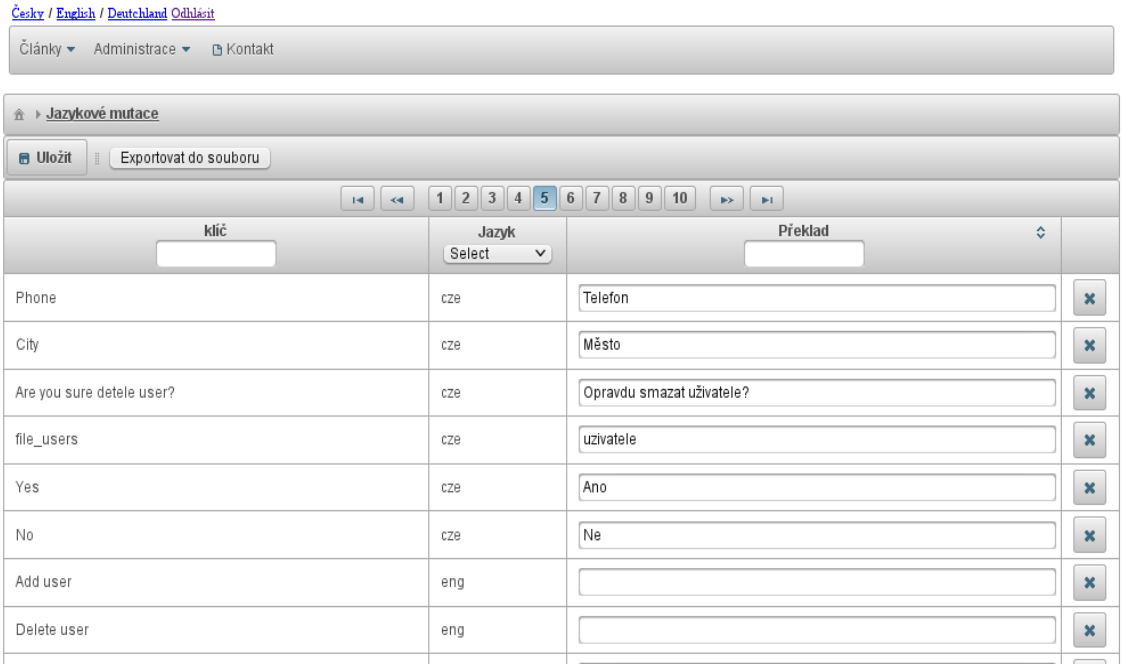


Obrázek 21: Databázové schéma frameworku

9.2 Bussines vrstva

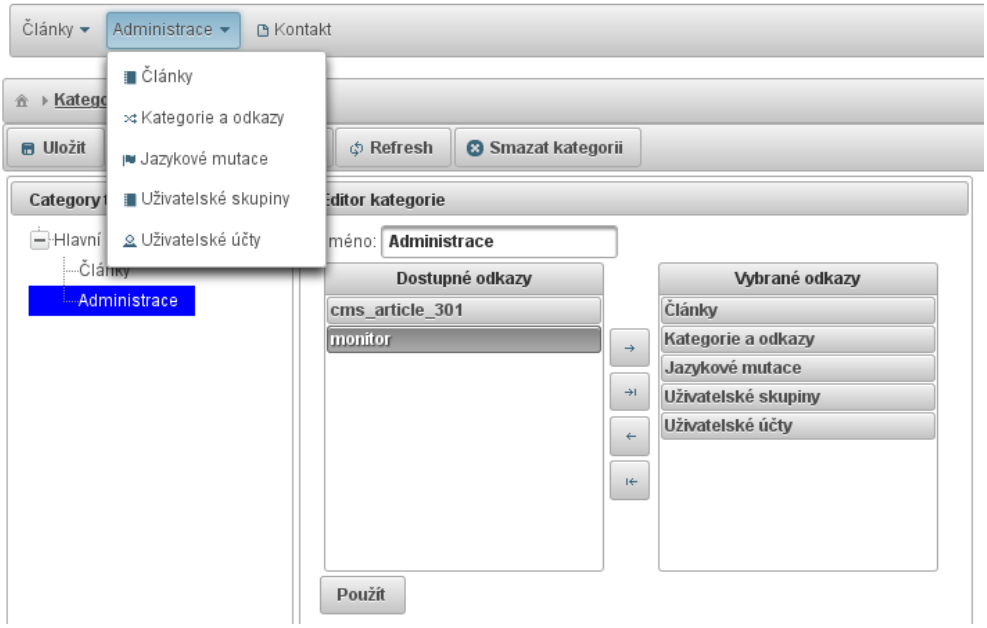
Jak bylo napsáno výše, jádro obsahuje některé základní komponenty. Jedná se o moduly, které by bylo problematické postupně zahrnovat do dalších komponent. Téměř na všech internetových stránkách jsou navíc potřebné.

Mula – Jedná se o komponentu zajišťující překlad stránek do různých jazyků. Jde však jenom o texty natvrdo zařazené do stránek (tlačítka, nápovědy, popisky atd). Texty článků si musí každý autor přeložit sám a založit pro ně speciální URL adresu. Jednak je to lepší pro SEO optimalizaci a navíc překládaná stránka nemusí mít svůj cizojazyčný protějšek. Při spuštění aplikace se do její tabulky nahrají všechny překládané texty v angličtině. Ta je zvolena jako primární jazyk. Překlady se pak ovládají přes webové rozhraní, kde doplníme překlady. Pokud překlad není nalezen, použije se implicitně angličtina. V kódu se Mula volá jako EJB `mula.get("Congratulations.");` a vrací překlad v aktuálně zvoleném jazyce.



Obrázek 22: Ukázka administrace komponenty Mula

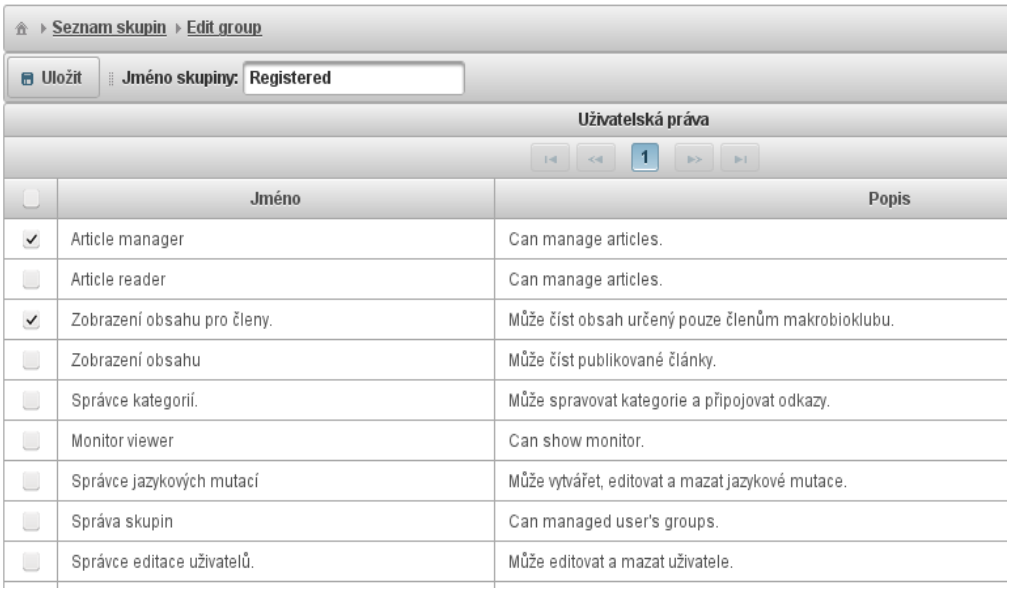
Linker – Tato komponenta má na starost tvorbu odkazů. Jedná se o usnadnění správy odkazů zejména pro vývojáře. Pokud potřebujeme například přesměrovat na list.xhtml, použijeme příkaz: `WebPage.redirect(linker.getLink("cms/list.xhtml"));` Linker se nám postará o doplnění celé korektní adresy. Pokud se změní jakkoliv cesta k adresáři nebo souboru, stačí změnu modifikovat na jednom místě. Komponenta se nám dále stará o tvorbu víceúrovňového menu. Nastaví se popisek, ikonka a URL adresa. Odkaz se nám poté přidá do do menu jako nezařazený. K samotné tvorbě a zařazování odkazů používáme webové rozhraní, kde lze menu kompletně budovat.



Obrázek 23: Tvorba menu a zařazování odkazů

Secure – Neboli komponenta zabezpečení. Má na starosti správu uživatelů, skupin a práv. Samozřejmostí je přidávání, editování a mazání uživatelů. Zajímavější je však řešení práv v systému. Práva jsou řešena přes skupiny a každé skupině se mohou přidělovat různá práva. Tento způsob nám dává opravdovou variabilitu při udělování práv. Uživatel může být zároveň ve více skupinách a každá skupina může mít nastaveno více práv.

Příklad: Příchozí uživatel (bez registrace) je ve skupině *Guest* a může prohlížet články a psát do diskuse. Po registraci zůstává ve skupině *Guest* a navíc se mu přiřadí skupina *Registered*. Ta mu k stávajícím právům přidá právo editovat svůj profil a například psát články.



Obrázek 24: Nastavení práv skupině Registered

9.3 Prezentační vrstva

Jedná se o uživatelské rozhraní. Je použit framework od společnosti Sun Microsystems - JSF (Java Server Faces). Na obrázku 20. - *Třívrstvá architektura projektu* je vidět, že je JSF rozdělen do dvou částí. První je čistě uživatelská a druhá obsahuje aplikační logiku v takzvaných *Managed bean*.

10 Komponenty Fyx

O teorii komponent pojednává kapitola *Komponentové systémy*. Rovněž komponenty pro framewrok Fyx jsou normálními komponentami bez nutnosti popisovat speciální metody. Jediné omezení je, pokud má být výstupní stránka v menu, vzniká nutnost přihlásit se k observeru (návrhový vzor). Zde se definují dvě registrované metody `menuCreate()` a `roleCreate()`. První metoda přidává odkaz do menu (název, ikonku, URL adresu, roli) a druhá definuje role, které smějí odkaz používat (název role, popis role).

V několika krocích zde bude uvedeno vytvoření nové komponenty. K vytvoření byla vybrána komponenta *Cms* a její metoda *přidání článku*. Cms má na starosti správu textů, tedy jejich vytvoření, editaci, mazání a publikaci. Celý postup bude prezentován na praktických ukázkách. Oproti skutečným zdrojovým kódům jsou vypuštěny některé nepodstatné části kódu z důvodu lepšího pochopení a šetření místa. Celý zdrojový kód komponenty Cms je pro ukázku uveden v příloze A. Následný postup je ukázán v prostředí NetBeans za použití frameworku JPA.

10.1 Vytvoření databázové tabulky

Nejprve se musí vytvořit databázová tabulka. K vytvoření se dá použít přímo vývojové prostředí Netbeans. V záložce Services → Databases se vytvoří nové připojení a zde jdou vytvářet a editovat tabulky. Alternativou je použití externího programu, jako je například MySQL Workbench. Zde lze vytvářet tabulky a vztahy mezi nimi také vizuálně. Následuje tedy vytvoření tabulky nazvané *cms_article* obsahující uvedené atributy:

- id = Jedná se o jednoznačný identifikátor článku, který bude použit i v URL.
- title = Titulek článku.
- time_create a time_editation = Čas vytvoření a čas poslední editace článku.
- author = Jedná se o cizí klíč odkazující na id tabulky autora.
- language = Cizí klíč odkazující na primární klíč iso_language u tabulky languages.

10.2 Generování Entity

Po vytvoření tabulky se musí vygenerovat entita. Tato entita je třída, která má za úkol reprezentovat tabulku. Instance třídy pak reprezentují jednotlivé řádky v tabulce. Třídu pro přehlednost vytvoříme v samostatném balíku nazvaném org.fyx.cms.entity. Samotné vygenerování provedeme vytvořením nového souboru *Entity Classes from Database*. Automaticky se nabídne seznam všech tabulek, kde vybereme potřebnou entitu (tabulku). S tabulkou se zároveň přetáhnou její závislé tabulky, které jsou spojeny přes cizí klíč (secure_user a mula_language). Vygeneruje se klasická třída doplněná o:

- anotaci *@Entity* z balíku javax.persistence.Entity
- pokud bude používána jako remote, musí implementovat interface *Serializable* z balíku java.io.Serializable
- ke všem atributům lze přistupovat pouze přes metody get a set
- musí mít bezparametrický privátní nebo chráněný konstruktor

Kromě metod get a set pro každý atribut se nám dále vygenerují tři metody:

- `hashCode()` - Vrací id záznamu, pokud je null vrátí 0.
- `equals(Object object)` - Tato metoda je určena k doprogramování a vrací true v případě shody objektů, jinak false.
- `ToString()` - Klasická metoda poděděná z java.lang.Object. Jedná se o konverzi objektu na textový řetězec.

10.3 Generování kontrolerů

Dalším krokem je vygenerování kontrolerů. Pro ně se vytvoří zvláštní balík org.fyx.cms.controll. Soubory jsou vygenerovány jako typ souboru *Session Beans For Entity Classes*. Jedná se o návrhový vzor Facade. Tento vzor definuje rozhraní vyšší úrovně tak, aby zlepšil přehlednost libovolného systému. Nahrazuje sadu rozhraní jednotlivých subsystémů jediným rozhraním, které zastupuje celý systém. Vygenerují se nám tři soubory:

- AbstractFacade – Abstraktní třída definující základní operace jako vytvoření, editaci, odebrání, nalezení dle id záznamu nebo nalezení všech entit. Pracuje s obecným typem entity – je tedy vždy stejná (`Class<T>`).
- CmsArticleFacade – Třída rozšiřuje AbstractFacade a implementuje lokální rozhraní.
- CmsArticleFacadeLocal – Jedná se o lokální rozhraní.

10.4 Tvorba formuláře

Ve Web Application (FyxWeb) v adresáři templates se vytvoří nová složka *cms*, kde budou uloženy všechny šablony pro komponentu Cms. Dále následuje ukázka jedné šablony, a to přidávání článků – add.xhtml. Protože se JSF stránka liší od klasické html, následuje popis a struktura některých tagů.

`ui:composition` – Stránka add.xhtml je celá obalována šablonou, proto se obalí do tagu <ui:composition>. Dále se definuje tag `<ui:define>` s id center. Při obalení šablonou definovanou v atributu template se pak add.xhtml vloží na řádek v šabloně: `<ui:insert name="center">Vkládaná stránka</ui:insert>`. Je zde rozdíl od klasického html, kde se volá vrchní obal a do něj se pak vkládají různé

další části kódu. Zde se jedná o návrhový vzor Inversion of Control(IoC), kdy se volaná stránka obaluje dalšími částmi. Umožňuje to volnější vazbu mezi komponenty.

`xmlns:ui` – Protože se v souboru xhtml mohou mísit tagy z různých knihoven, musí se rozlišit pomocí jmenných prostorů. Před použitím tagu se tedy musí deklarovat jeho jmenný prostor a prefix. Základní jmenné prostory:

- **ui:** = Obsahuje tagy pro tvorbu šablony.
- **h:** = Tagy pro komponenty definující uživatelské rozhraní.
- **f:** = Tagy, které se nezobrazují, tedy nejsou generovány.
- **c:** = Sjednocuje podmínky, cykly atd.
- **p:** = Nepatří mezi základní jmenné prostory, ale ve frameworku je využíván. Značí grafické komponenty PrimeFaces.

`#{}` - Přes tento zápis se volají metody v Managed Bean. Pokud se má například zobrazit nebo nastavit titulek, zavolá se příslušná Managed Bean. V tomto případě *cmsArticleEdit*. Při zápisu se automaticky volá odpovídající metoda set a při čtení je to pak metoda get. Atribut article tedy musí tyto dvě metody implementovat:

```
public CmsArticle getArticle() {
   return article;
}
public void setArticle(CmsArticle article) {
   this.article = article;
}
```

Přes daný zápis se nemusí volat jenom metody set a get, ale i různé jiné vlastnosti. Příkladem může být zápis: `#{cmsArticleEdit.initArticle()}`, kde se nastavují vlastnosti odpovídající zcela novému článku.

Na stránce se pak vyskytují některé další grafické komponenty, kterých je ale velká spousta a jejich podrobný popis by byl na několik desítek, možná i stovek stran. Proto autor odkazuje na zdroje primefaces.org [36] a oracle.com [37]. Ukázka zdrojového kódu:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:p="http://primefaces.prime.com.tr/ui"
                template="#{webContext.skin0Culomns}"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core">
   <ui:define name="center">
   <h:form id="addArticleForm">
      <p:messages id="messages" />
      #{cmsArticleEdit.initArticle()}
      <h1 class="title ui-widget-header ">#{ml.tr('Adding articles')}</h1>
```

```
      <p:breadCrumb>
         <p:menuitem value="#{ml.tr('Home')}" url="#{webContext.baseUrl}" />
         <p:menuitem value="#{ml.tr('Article list')}"
                     url="#{lm.ml('cms/list.xhtml')}" />
         <p:menuitem value="#{ml.tr('Article detail')}" url="#" />
      </p:breadCrumb>
      <h:panelGrid columns="2" columnClasses="label, value"
                   styleClass="grid">
         <h:outputText value="#{ml.tr('Title')}: * " />
         <p:inputText required="true" label="#{ml.tr('Title')}"
                      value="#{cmsArticleEdit.article.title}"
                      requiredMessage="#{ml.tr('The title is required')}" />
         <h:outputText value="#{ml.tr('Language')}: * " />
         <h:selectOneMenu id="selectLanguage"
                          value="#{cmsArticleEdit.article.language}"
                          converter="#{languageConverter}">
            <f:selectItems value="#{cmsArticleEdit.languages}" var="l"
                           itemValue="#{l}" itemLabel="#{l.isoLanguage}" />
         </h:selectOneMenu>
         <h:outputText value="#{ml.tr('Article')}: * " />
         <p:editor value="#{cmsArticleEdit.article.content}"
                   widgetVar="editor" width="600" />
      </h:panelGrid>
      <p:commandButton  value="#{ml.tr('Save')}"
                        image="ui-icon ui-icon-disk"
                        actionListener="#{cmsArticleEdit.save}"
                        update="messages"/>
   </h:form>
   </ui:define>
</ui:composition>
```

10.5 Tvorba Managed Bean

Slouží k propojení prezenční vrstvy s business logikou. Veškerá business logika by měla probíhat až v komponentě. Proto by se zde měly vyskytovat hlavně funkce umožňující komunikaci s komponentou, popřípadě některé konverzní, inicializační nebo ověřovací funkce.

Nejprve je nutné získat instanci Enterprise Java Bean, která obsahuje business logiku. Protože se o takovou instanci stará EJB kontejner, který řídí její cyklus, neprovádí se klasickým zápisem: `Trida instance = new Trida();` Místo toho se použije:

```
@EJB
private CmsLocal cms;
```

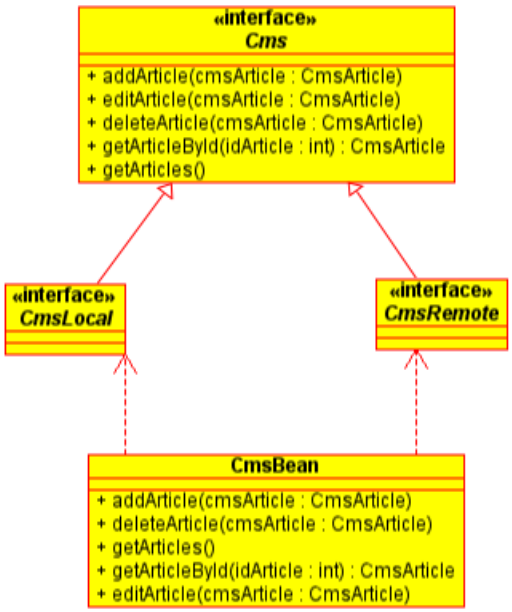
Dále se klasicky vytvoří privátní instance *CmsArticle* a přidají se metody get a set. Nakonec je napsána funkce `save()`, která lokální instanci CmsArticle uloží do instance JavaBean. Zároveň naplní přepravku (idom návrhového vzoru) WebContext hláškou o vložení.

```
@ManagedBean
@SessionScoped
public class CmsArticleEdit {
   @EJB
   private MulaLocal mula;   //Mula zabezpečuje překlad textů do různých jazyků
   @EJB
   private LinkerLocal linker;        //Linker má nastarosti tvorbu všech odkazů
   @EJB
   private CmsLocal cms;                          //instance aplikační logiky
   private CmsArticle article = new CmsArticle();

   public CmsArticleEdit() {                      //povinný prázdný konstruktor
   }
   public CmsArticle getArticle() {
      return article;                             //metoda get pro instanci article
   }
   public void setArticle(CmsArticle article) {   //metoda set pro instanci article
      this.article = article;
   }
   public void save() {                //tato metoda provede vlastní uložení dat
      cms.addArticle(this.article);
      FacesMessage msg = new FacesMessage(mula.get("Article was created."),
      mula.get("Congratulations."));
      WebContext.getMessages().add(msg);
      WebPage.redirect(linker.getLink("cms/list.xhtml"));
   }
   public void initArticle() {                    //inicializační metoda pro článek
      this.article = new CmsArticle();
   }
}
```

10.6 Vytvoření Session Bean

V posledním kroku následuje vytvoření Session Bean a její rozhraní v balíku *org.fyx.cms.ejb*. Nejprve se vytvoří rozhraní Cms, které definuje metodu `void addArticle(CmsArticle ca);` Z něj dědí další dvě rozhraní, a to lokální (CmsLocal) a vzdálené (CmsRemote). V případě, že existují rozdílné metody pro vzdálené volání, uvedeme je dle potřeby jenom do lokálního nebo vzdáleného rozhraní. Protože ale ukazovaná metoda nemá odlišné metody, jsou obě rozhraní na obrázku prázdné! Jako poslední se vygeneruje soubor CmsBean, kde je umístěna business logika.



Obrázek 25: UML diagram Session Bean Cms

Následuje kód souboru CmsBean:

```
@Stateless
public class CmsBean implements CmsLocal {
   @EJB
   private CmsArticleFacadeLocal cmsArticleFacade;   //vytvoří se instance...
                                                     // controlleru – pro vložení do databáze
   @Override
   public void addArticle(CmsArticle article) {
      if (!FyxContextFactory.getContext().isUserInRole(
      CmsRoles.ARTICLES_MANAGER)) {                  //má uživatel právo přidat
         throw new PermissionDeniedException();
      }
      article.setAuthor(FyxContextFactory.getContext().getUser());  //přidání autora
      article.setTimeCreate(new Date());             //aktuální čas vložení
      article.setTimeEditation(null);                //editace ještě neproběhla
      cmsArticleFacade.create(article);              //samotné přidání článku
   }
}
```

Závěr

Cílem práce bylo vytvořit framework pro usnadnění vývoje webových aplikací. Tento framework je napsán na platformě Java Enterprise Edition a podporuje komponentový vývoj. Zároveň je připraven pro distribuované programování, a tím splňuje všechny body zadání.

V teoretické části jsou popsány databázové systémy. V grafech je porovnáno ukládání a vyhledávání objektů, což jsou dvě nejčastější operace s databází. Porovnání bylo provedeno na vlastní aplikaci. Jedná se o porovnání databází relačních a objektových jakožto představitelů dvou moderních trendů v oblasti vývoje databází. Jako nejvhodnější pro projekt byla zvolena databáze MySQL. Dále se práce věnuje komponentovým technologiím a komponentám. Udává, jak a co by měla komponenta obsahovat a jak fungují tyto technologie.

Praktická část popisuje jádro a vznik komponenty. Na praktickém příkladu byla předvedena tvorba komponenty *Cms*, která má na starosti správu textů.

Výsledný projekt Fyx vznikal ve spolupráci s další diplomovou prací na téma *Distribuovaný webový framework* [38]. Jejím autorem je Bc. Jan Kysela. Spolupráce na frameworku přinesla jeho vyšší kvalitu jak v kódu, tak i v návrhu. Zároveň se oba autoři učili práci v týmu. Přestože na návrhu i implementaci spolupracovali, autor měl hlavně na starosti komponentu *Cms* a v jádru modul *Mula*. Druhý autor pak moduly *Secure*, *Linker* a další věci v jádru.

Jádro frameworku bude v budoucnu rozšířeno o logování chyb do souboru. Z tohoto souboru se poté bude provádět analýza a rekonstrukce případného problému. Dále by měla vzniknout on-line dokumentace pro vývojáře a oficiální stránky frameworku *Fyx*.

Framework bude nadále rozvíjen a využíván. V plánu je postavit na tomto projektu rozsáhlý portál o makrobiotice zahrnující recepty, makrobiotické ubytování, diskuse, plánování srazů pro makrobiotiky atd. Bude se jednat o celosvětový web. V nejbližší době bude využit i k naprogramování menších webů.

Použitá literatura

[1] PECINOVSKÝ, Rudolf. *Návrhové vzory.* Brno: Computer Press a.s., 2007. 527 s. ISBN 978-80-251-1582-4.

[2] ARLOW, Jim; NEUSTADT, Ila. *UML 2 a unifikovaný proces vývoje aplikací.* Brno: Computer Press a.s., 2008. 567 s. ISBN 978-80-251-1503-9.

[3] GEARY, David; HORSTMANN, Cay. *Core Java Server Faces.* 3rd edition. Michigan: Prentice Hall, 2010. 636 s. ISBN:13-978-0-13-701289-3.

[4] RUBINGER Andrew; BURKE, Bill. *Enterprise JavaBeans 3.1.* 6rd edition. Sebastopol: O'Reilly Media, 2010. 738 s. ISBN:978-0-596-15802-6.

[5] ROMAN, Ed; SRIGANESH, Rima; BROSE, Gerald. *Mastering Enterprise JavaBeans.* 3rd edition. Indianapolis: Wiley Publishing, Inc., 2005. 815 s. ISBN:0-7645-7682-8.

[6] SRIGANESH, Rima; BROSE, Gerald; SILVERMAN, Micah. *Mastering Enterprise JavaBeans 3.0.* 1rd edition. Indianapolis: Wiley Publishing, 2006. 685 s. ISBN-13:978-0-471-78541-5.

[7] *Databázový svět* [online]. 2001 [cit. 2010-12-05]. Databáze nejsou jen relační. Dostupné z WWW: <http://www.dbsvet.cz/view.php?cisloclanku=2001111201>.

[8] *Databázové systémy* [online]. 2009 [cit. 2010-12-05]. Databázové systémy – Relační, Deduktivní, Temporální. Dostupné z WWW: <http://www.managed-dedicated-servery.net/databazove-systemy.html>.

[9] *Databázový svět* [online]. 2002 [cit. 2010-12-05]. Databáze nejsou jen relační, díl čtvrtý – RDM. Dostupné z WWW: <http://www.dbsvet.cz/view.php?cisloclanku=2002010902>.

[10] *www.manualy.net* [online]. 2007 [cit. 2011-04-12]. Teorie relačních databází: Normalizace. Dostupné z WWW: <http://www.manualy.net/article.php?articleID=13>.

[11] *ODBMS.ORG* [online]. 2011 [cit. 2011-02-02]. Object Database (ODBMS) | Object-Oriented Database (OODBMS) | Free Resource Portal. Dostupné z WWW: <http://www.odbms.org/>.

[12] *interval.cz* [online]. 2000 [cit. 2011-04-12]. Databáze a jazyk SQL. Dostupné z WWW: <http://interval.cz/clanky/databaze-a-jazyk-sql/>.

[13] *fi.muni.cz* [online]. [cit. 2011-01-12]. Relační vs. objektově-relační vs. objektové databáze. Dostupné z WWW: <http://www.fi.muni.cz/~xbatko/oracle/compare.html>.

[14] *Wikipedie* [online]. 2011 [cit. 2011-01-12]. MySQL. Dostupné z WWW: <http://cs.wikipedia.org/wiki/MySQL>.

[15] *interval.cz* [online]. 2003 [cit. 2011-02-28]. Úvod do JDBC. Dostupné z WWW: <http://interval.cz/clanky/uvod-do-jdbc/>.

[16] *PostgreSQL* [online]. 2011 [cit. 2011-02-21]. PostgreSQL: The world's most advanced open source database. Dostupné z WWW: <http://www.postgresql.org/>.

[17] *PostgreSQL* [online]. 2011 [cit. 2011-02-21]. PostgreSQL. Dostupné z WWW: <http://www.pgsql.cz/index.php/PostgreSQL>.

[18] *db4o* [online]. 2011 [cit. 2011-02-28]. db4o :: Java & .NET Object Database – Open Source Object Database, Open Source Persistence, Oodb. Dostupné z WWW: <http://www.db4o.com/>.

[19] *databeans* [online]. 2009 [cit. 2011-03-02]. databeans - a new, fully object oriented persistence framework for java. Dostupné z WWW: <http://databeans.sourceforge.net/>.

[20] *JDOInstruments* [online]. [cit. 2011-03-04]. JDOInstruments -java object database-oodb -oodbms. Dostupné z WWW: <http://www.jdoinstruments.org/>.

[21] *MyOODB* [online]. 2009 [cit. 2011-03-04]. MyOODB (Object Database & Application Framework) - Free SDK for Database, Web and System Development. Dostupné z WWW: <http://www.myoodb.org/>.

[22] *NeoDatis* [online]. 2008 [cit. 2011-03-05]. NeoDatis - ODB : The Native Object Oriented Database. Dostupné z WWW: <http://www.neodatis.org/>.

[23] *ozone* [online]. [cit. 2011-03-12]. Ozone. Dostupné z WWW: <http://www.ozone-db.org/frames/home/what.html>.

[24] *Perst Embedded Database* [online]. 2011 [cit. 2011-03-13]. Perst Embedded Database – an open source, object-oriented Java database / .NET database. Dostupné z WWW: <http://www.mcobject.com/perst>.

[25] MERUŇKA, Vojtěch. [online]. [cit. 2011-01-16]. Normalizace v objektových databázích. Dostupné z WWW: <*czu.vasekk.cz/Ing/szz/PIS/pis_12-13_Merunka.pdf*>.

[26] *Java.net*[online]. 2011 [cit. 2011-04-18]. JavaServer Faces Community. Dostupné z WWW: <http://javaserverfaces.java.net/users.html>.

[27] *Automa* [online]. 2010 [cit. 2011-04-18]. Co nového přinášejí moderní softwarové architektury?. Dostupné z WWW: <http://www.odbornecasopisy.cz/index.php?id_document=32371>.

[28] *Wiki Kivu* [online]. 2011 [cit. 2011-05-11]. Úvod do komponent. Dostupné z WWW: <http://wiki.kiv.zcu.cz/UvodDoKomponent/HomePage>.

[29] *Živě.cz* [online]. 2005 [cit. 2011-04-22]. Umíme to s Delphi: 163. díl – komponentní technologie, úplný úvod. Dostupné z WWW: <http://www.zive.cz/clanky/umime-to-s-delphi-163-dil--komponentni-technologie-uplny-uvod/sc-3-a-126591/default.aspx>.

[30] *Lesson: JDBC Basics* [online]. 2011 [cit. 2011-04-17]. Lesson: JDBC Basics. Dostupné z WWW: <http://download.oracle.com/javase/tutorial/jdbc/basics/index.html>.

[31] *NetBeans.org* [online]. 2011 [cit. 2011-03-19]. *NetBeans*. Dostupné z WWW: <http://netbeans.org/>.

[32] *oracle.com* [online]. 2011 [cit. 2011-02-20]. The Java Persistence API - A Simpler Programming Model for Entity Persistence. Dostupné z WWW: <http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html>.

[33] *NetBeans.org* [online]. 2011 [cit. 2011-03-25]. Java EE & Java Web Learning Trail. Dostupné z WWW: <http://netbeans.org/kb/trails/java-ee.html>.

[34] ZAKHOUR, Sharon. *Java 6.* Brno: Computer Press a.s., 2007. 533 s. ISBN 978-80-251-1575-6.

[35] *MySQL* [online]. 2011 [cit. 2011-08-26]. MySQL :: The world's most popular open source database. Dostupné z WWW: <http://www.mysql.com/>.

[36] *PrimeFaces* [online]. 2011 [cit. 2011-08-26]. PrimeFaces. Dostupné z WWW: <http://www.primefaces.org/>.

[37] *Oracle* [online]. 2008 [cit. 2011-08-26]. The Java EE 5 Tutorial. Dostupné z WWW: <http://java.sun.com/javaee/5/docs/tutorial/doc/>.

[38] Kysela, Jan. *Distribuovaný webový framework*. Pardubice, 2011. 81s. Diplomová práce na fakultě Elektrotechniky a Informatiky Univerzity Pardubice. Vedoucí diplomové práce Ing. Karel Šimerda.

[39] KOHOUT, Josef. [online]. 2010 [cit. 2011-04-28]. Programování a užití komponent. Učební text pro studenty předmětu KIV/PUK. Západočská univerzita v Plzni. Dostupné z WWW: <*portal.zcu.cz/wps/PA_Courseware/DownloadDokumentu?id=35641*>.

[40] *java.cecrdle.net* [online]. [cit. 2011-05-13]. Úvod do architektury CORBA. Dostupné z WWW: <http://java.cecrdle.net/vyzkum/kapitola_3.htm>.

[41] *Sun Microsystems* [online]. 2008 [cit. 2011-05-18]. The Java EE 5 Tutorial. Dostupné z WWW: <http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf>.

Část III.

Přílohy

Příloha A – komponenta Cms

Následuje kompletní zdrojový kód komponenty *Cms*. Postupně jsou uvedeny všechny soubory, jak byly uvedeny v kapitole *Komponenty Fyx*. Následující kód byl upraven pro tisk a zároveň je doplněn o některé komentáře.

CmsArticle.java

Entita reprezentující tabulku cms_article. Instance této třídy pak zastupují jednotlivé řádky v tabulce. Jedná se o klasickou třídu doplněnou o anotaci @Entity. Dále obsahuje metody get a set pro všechny metody a bezparametrický konstruktor.

```
package org.fyx.cms.entity;
/**
 * @author ydenek
 */
@Entity
@Table(name = "cms_article")
@XmlRootElement
@NamedQueries({
   @NamedQuery(name = "CmsArticle.findAll", query = "SELECT c FROM
CmsArticle c"),
   @NamedQuery(name = "CmsArticle.findById", query = "SELECT c FROM
CmsArticle c WHERE c.id = :id"),
   @NamedQuery(name = "CmsArticle.findByTitle", query = "SELECT c FROM
CmsArticle c WHERE c.title = :title"),
   @NamedQuery(name = "CmsArticle.findByTimeCreate", query = "SELECT c
FROM CmsArticle c WHERE c.timeCreate = :timeCreate"),
   @NamedQuery(name = "CmsArticle.findByTimeEditation", query = "SELECT c
FROM CmsArticle c WHERE c.timeEditation = :timeEditation")})
public class CmsArticle implements Serializable {
   private static final long serialVersionUID = 1L;
   @TableGenerator(name = "CMS_ARTICLE_GEN",
      table = "SEQUENCE",
      pkColumnName = "SEQ_NAME",
      valueColumnName = "SEQ_COUNT",
      pkColumnValue = "CMS_ARTICLE")
   @Id
   @GeneratedValue(strategy = GenerationType.TABLE,
generator="CMS_ARTICLE_GEN")
   @Basic(optional = false)
   @Column(name = "id")
   private Integer id;
   @Size(max = 255)
```

```
@Column(name = "title")
private String title;
@Lob
@Size(max = 2147483647)
@Column(name = "content")
private String content;
@Column(name = "time_create")
@Temporal(TemporalType.TIMESTAMP)
private Date timeCreate;
@Column(name = "time_editation")
@Temporal(TemporalType.TIMESTAMP)
private Date timeEditation;
@JoinColumn(name = "author", referencedColumnName = "id_user")
@ManyToOne
private SecureUser author;
@JoinColumn(name = "language", referencedColumnName = "iso_language")
@ManyToOne(optional = false)
private MulaLanguage language;
public CmsArticle() {                              //povinný prázdný konstruktor
}

public CmsArticle(Integer id) {
    this.id = id;
}

public Integer getId() {
    return id;
}
public void setId(Integer id) {
    this.id = id;
}

public String getTitle() {
    return title;
}                                                  //metody get a set pro všechny atributy
public void setTitle(String title) {
    this.title = title;
}

public String getContent() {
    return content;
}
public void setContent(String content) {
    this.content = content;
}

public Date getTimeCreate() {
    return timeCreate;
}
```

```
    public void setTimeCreate(Date timeCreate) {
        this.timeCreate = timeCreate;
    }

    public Date getTimeEditation() {
        return timeEditation;
    }
    public void setTimeEditation(Date timeEditation) {
        this.timeEditation = timeEditation;
    }

    public SecureUser getAuthor() {
        return author;
    }
    public void setAuthor(SecureUser author) {
        this.author = author;
    }

    public MulaLanguage getLanguage() {
        return language;
    }
    public void setLanguage(MulaLanguage language) {
        this.language = language;
    }
    @Override                                           //metoda vraci id objeku
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
    @Override                                       //metoda pro porovnání objektů
    public boolean equals(Object object) {
        // TODO
        if (!(object instanceof CmsArticle)) {
            return false;
        }
        CmsArticle other = (CmsArticle) object;
        if ((this.id == null && other.id != null) || (this.id != null &&
this.id.equals(other.id))) {
            return false;
        }
        return true;
    }
    @Override                                       //přetížená metoda pro výpis objektu
    public String toString() {
        return "org.fyx.cms.entity.CmsArticle[ id=" + id + " ]";
    }
}
```

AbstractFacade.java

Abstraktní třída definující základní operace jako vytvoření, editaci, odebrání, nalezení záznamu dle id nebo nalezení všech entit. Pracuje s obecným typem entit – je tedy vždy stejná (Class<T>).

```
package org.fyx.cms.controll;

import java.util.List;
import javax.persistence.EntityManager;

/**
 *
 * @author ydenek
 */
public abstract class AbstractFacade<T> {
    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {
        getEntityManager().persist(entity);
    }

    public void edit(T entity) {
        getEntityManager().merge(entity);
    }

    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }

    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }

    public List<T> findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        return getEntityManager().createQuery(cq).getResultList();
    }

    public List<T> findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
```

```
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0]);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root<T> rt = cq.from(entityClass);
        cq.select(getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q = getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }

}
```

CmsArticleFacadeLocal.java

Konkrétní lokální rozhraní pro CmsArticle.

```
package org.fyx.cms.controll;

import java.util.List;
import javax.ejb.Local;
import org.fyx.cms.entity.CmsArticle;

/**
 * @author ydenek
 */
@Local
public interface CmsArticleFacadeLocal {
    void create(CmsArticle cmsArticle);
    void edit(CmsArticle cmsArticle);
    void remove(CmsArticle cmsArticle);
    CmsArticle find(Object id);
    List<CmsArticle> findAll();
    List<CmsArticle> findRange(int[] range);
    int count();
}
```

CmsArticleFacade

Třída rozšiřuje AbstractFacade a implementuje lokální rozhraní.

```
package org.fyx.cms.controll;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.fyx.cms.entity.CmsArticle;
```

```
/**
 * @author ydenek
 */
@Stateless
public class CmsArticleFacade extends AbstractFacade<CmsArticle> implements
CmsArticleFacadeLocal {
    @PersistenceContext(unitName = "FyxCorePU")
    private EntityManager em;

    protected EntityManager getEntityManager() {
        return em;
    }

    public CmsArticleFacade() {
        super(CmsArticle.class);
    }

}
```

add.xhtml

Stránka slouží k přidávání článků. Obaluje formArticle.xhtml, který je společný i pro soubor edit.xhtml. Výsledný soubor je pak dán šablonou definovanou v tagu template.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:p="http://primefaces.prime.com.tr/ui"
                template="#{webContext.skin0Culomns}"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:f="http://java.sun.com/jsf/core">

    <ui:define name="center">
        <h:form id="addArticleForm">
            <p:messages id="messages" />
            #{cmsArticleEdit.initArticle()}
            <h1 class="title ui-widget-header ">
                #{ml.tr('Adding articles')}
            </h1>

            <ui:include src="formArticle.xhtml">Formulář článku</ui:include>
            <p:commandButton  value="#{ml.tr('Save')}"
                              image="ui-icon ui-icon-disk"
                              actionListener="#{cmsArticleEdit.save}"
                              update="messages"/>
        </h:form>
    </ui:define>

</ui:composition>
```

edit.xhtml

Stránka slouží k editaci článků. Obaluje formArticle.xhtml, který je společný i pro soubor add.xhtml. Výsledný soubor je pak dán šablonou definovanou v tagu template.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
              xmlns:p="http://primefaces.prime.com.tr/ui"
              template="#{webContext.skin0Culomns}"
              xmlns:h="http://java.sun.com/jsf/html"
              xmlns:f="http://java.sun.com/jsf/core">

   <ui:define name="center">
      <h:form id="addArticleForm">
         #{cmsArticleEdit.initArticle(param["p1"])}
         <h1 class="title ui-widget-header ">
            #{ml.tr('Edditing articles')}
         </h1>

         <ui:include src="formArticle.xhtml">Formulář článku</ui:include>

         <h:panelGrid columns="2" style="margin-top:10px">
            <p:commandButton  value="#{ml.tr('Edit')}"
                        image="ui-icon ui-icon-disk"
                        actionListener="#{cmsArticleEdit.edit}" />
         </h:panelGrid>
      </h:form>
   </ui:define>

</ui:composition>
```

formArticle.xhtml

Vkládaný formulář pro stránky edit.xhtml a add.xhtml. Jedná se o část stejného kódu, který je vyčleněn pro jednodušší editaci zdrojového kódu.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
     xmlns:p="http://primefaces.prime.com.tr/ui"
     xmlns:h="http://java.sun.com/jsf/html"
     xmlns:f="http://java.sun.com/jsf/core">
   <h:body>
      <p:breadCrumb>
            <p:menuitem value="#{ml.tr('Home')}" url="#{webContext.baseUrl}" />
            <p:menuitem value="#{ml.tr('Article list')}"
url="#{lm.ml('cms/list.xhtml')}" />
            <p:menuitem value="#{ml.tr('Article detail')}" url="#" />
```

```
      </p:breadCrumb>

      <h:panelGrid columns="2" columnClasses="label, value" styleClass="grid">

         <h:outputText value="#{ml.tr('Title')}: * " />
         <p:inputText required="true" label="#{ml.tr('Title')}"
                  value="#{cmsArticleEdit.article.title}"
                  requiredMessage="#{ml.tr('The title is required')}" />

         <h:outputText value="#{ml.tr('Language')}: * " />
         <h:selectOneMenu id="selectLanguage"
value="#{cmsArticleEdit.article.language}" converter="#{languageConverter}">
            <f:selectItems value="#{cmsArticleEdit.languages}"
                     var="l"
                     itemValue="#{l}"
                     itemLabel="#{l.isoLanguage}" />
         </h:selectOneMenu>

         <h:outputText value="#{ml.tr('Article')}: * " />
         <p:editor value="#{cmsArticleEdit.article.content}" widgetVar="editor"
                  width="600" />

      </h:panelGrid>
   </h:body>
</html>
```

list.xhtml

Zobrazuje přehled všech uložených článků v systému. Ty se zobrazují v přehledné tabulce s možností vyhledávání a řazení. Tato stránka zároveň umožňuje mazání zobrazených článků.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
            xmlns:p="http://primefaces.prime.com.tr/ui"
            template="#{webContext.skin0Culomns}"
            xmlns:h="http://java.sun.com/jsf/html"
            xmlns:f="http://java.sun.com/jsf/core">
   <ui:define name="center">
      <h:form id="articlesForm">
         <!--  XXXXXXXXXXXXXXXX MENU XXXXXXXXXXXXXXXXXX -->
         <p:toolbar>
            <p:toolbarGroup align="left">
               <p:commandButton type="submit"
                        value="#{ml.tr('Add')}"
                        image="ui-icon-plus"
                        actionListener="#{man.redirect(lm.ml('cms/add.xhtml'))}" />
               <p:commandButton type="submit"
```

```
                        value="#{ml.tr('Show')}"
                        image="ui-icon-search"
                        actionListener="#{man.redirect(lm.ml('cms/show.xhtml?p1=',
                                        cmsArticleList.selectedArticle.id))}" />
                <p:commandButton type="submit"
                        value="#{ml.tr('Edit')}"
                        image="ui-icon-pencil"
                        actionListener="#{man.redirect(lm.ml('cms/edit.xhtml?p1=',
                                cmsArticleList.selectedArticle.id))}" />
                <p:commandButton type="button"
                        value="#{ml.tr('Delete')}"
                        image="ui-icon ui-icon-circle-close"
                        onclick="confirmationDelete.show()"
                        update="display" />
                <p:confirmDialog message="#{ml.tr('Are you sure?')}"
                        header="#{ml.tr('Deleting articles')}"
                        severity="alert" widgetVar="confirmationDelete">
                   <p:commandButton value="#{ml.tr('Yes')}"
                        update="articleList"
                        oncomplete="confirmationDelete.hide();
statusDialog.hide();"
                        onclick="statusDialog.show()"
                        actionListener="#{cmsArticleList.deleteSelectedArticle}"/>
                   <p:commandButton value="#{ml.tr('No')}"
                        onclick="confirmationDelete.hide()"
                        type="button" />
                </p:confirmDialog>
                <ui:include src="#{webContext.skinPatch}statusDialog.xhtml" />
                <p:divider />
             </p:toolbarGroup>
             <p:divider />
             <p:toolbarGroup>
                <p:menuButton value="#{ml.tr('Export')}">
                   <p:menuitem>
                      <h:commandLink>
                         <h:outputText value="Csv" />
                         <p:dataExporter type="csv" target="articleList"
                                fileName="#{ml.tr('file_articles')}"
                                excludeColumns="3,4" />
                      </h:commandLink>
                   </p:menuitem>
                </p:menuButton>
             </p:toolbarGroup>
          </p:toolbar>
          <!--  XXXXXXXXXXXXXXX TABULKA XXXXXXXXXXXXXXXXXX -->
          <p:dataTable id="articleList" var="article"
value="#{cmsArticleList.cms.articles}"
                 paginator="true" rows="12" selectionMode="single"
selection="#{cmsArticleList.selectedArticle}">
```

```
            <f:facet name="header">
               "#{ml.tr('List of articles')}"
            </f:facet>

            <p:column filterBy="#{article.title}"
                  headerText="#{ml.tr('Title')}"
                  filterMatchMode="contains">
               <h:outputText value="#{article.title}" />
            </p:column>

            <p:column filterBy="#{article.author.firstname}"
                  headerText="#{ml.tr('Name')}"
                  filterMatchMode="contains">
               <h:outputText value="#{article.author.firstname}" />
            </p:column>

            <p:column filterBy="#{article.author.lastname}"
                  headerText="#{ml.tr('Surname')}"
                  filterMatchMode="contains">
               <h:outputText value="#{article.author.lastname}" />
            </p:column>

            <p:column filterBy="#{article.language.isoLanguage}"
                  headerText="#{ml.tr('Language')}"
                  filterMatchMode="contains">
               <h:outputText value="#{article.language.isoLanguage}" />
            </p:column>

            <p:column filterBy="#{article.timeCreate.toLocaleString()}"
                  headerText="#{ml.tr('Created')}"
                  filterMatchMode="contains">
               <h:outputText value="#{article.timeCreate.toLocaleString()}" />
            </p:column>
         </p:dataTable>
      </h:form>
   </ui:define>
</ui:composition>
```

show.xhtml

Zobrazuje konkrétní článek uživateli.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
            xmlns:p="http://primefaces.prime.com.tr/ui"
            template="#{webContext.skin0Culomns}"
            xmlns:h="http://java.sun.com/jsf/html"
            xmlns:f="http://java.sun.com/jsf/core">
```

```
   <ui:define name="center">
      #{cmsArticleShow.initArticle(param["p1"])}
      <h:form id="testForm">

         <p:panel header="#{cmsArticleShow.article.title}">
           <h:outputText escape="false"
                         value="#{cmsArticleShow.article.content}" />
         </p:panel>
         <p:panel>
            <h:outputText value="#{cmsArticleShow.article.author.firstname}
#{cmsArticleShow.article.author.lastname}" />
            <h:outputText value="#{ml.tr('Poslední změna')}:
#{cmsArticleShow.lastModification.toLocaleString()}" style="float: right;"/>
         </p:panel>

      </h:form>
   </ui:define>
</ui:composition>
```

CmsArticleEdit.java

Společná ManagedBean pro edit.xhtml a add.xhtml. Obsahuje privátní proměnou CmsArticle z (do) které jsou zaznamenány všechny informace o článku. Dále jsou definovány metody pro inicializaci, editaci a uložení článku.

```
package org.fyx.cms.web;

import java.util.Date;
import java.util.List;
import javax.ejb.EJB;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import org.fyx.cms.ejb.CmsLocal;
import org.fyx.cms.entity.CmsArticle;
import org.fyx.linker.web.LinkerLocal;
import org.fyx.mula.controll.MulaLanguageFacadeLocal;
import org.fyx.mula.ejb.MulaLocal;
import org.fyx.mula.entity.MulaLanguage;
import org.fyx.secure.ejb.SecureLocal;
import org.fyx.web.context.WebContext;
import org.fyx.web.context.WebPage;

/**
 *
 * @author ydenek
 */
@ManagedBean
@SessionScoped
public class CmsArticleEdit {
```

```
@EJB
private MulaLocal mula;
@EJB
private LinkerLocal linker;                                          //proměnné EJB
@EJB
private CmsLocal cms;
@EJB
private MulaLanguageFacadeLocal mulaLanguageFacase;
@EJB
private SecureLocal secure;
private CmsArticle article = new CmsArticle();

/** Creates a new instance of CmsArticle */
public CmsArticleEdit() {                                  //bezparametrický konstruktor
}
public CmsArticle getArticle() {
   return article;
}
public void setArticle(CmsArticle article) {
   this.article = article;
}

public void save() {                                        //metoda vytvoří nový článek
   cms.addArticle(this.article);
   linker.recreateMenuModel();
   FacesMessage msg = new FacesMessage(mula.get("Article was created."),
         mula.get("Congratulations."));
   WebContext.getMessages().add(msg);
   WebPage.redirect(linker.getLink("cms/list.xhtml"));
}

public void edit() {                                       //metoda edituje aktuální article
   this.article.setTimeEditation(new Date());
   cms.editArticle(this.article);
   FacesMessage msg = new FacesMessage(mula.get("Article was changed."),
         mula.get("Congratulations."));
   WebContext.getMessages().add(msg);
   WebPage.redirect(linker.getLink("cms/list.xhtml"));
}

public void initArticle(int idArticle) {              //inicializace pro editovaný článek
   if (idArticle == 0) {
      FacesMessage msg = new FacesMessage(mula.get("Select the article."),
            mula.get("You must select article."));
      WebContext.getMessages().add(msg);
      WebPage.redirect(linker.getLink("cms/list.xhtml"));
      return;
   }
   this.article = cms.getArticleById(idArticle);
}
```

```
    public void initArticle() {                                    //inicializace pro nový článek
        this.article = new CmsArticle();
    }

    public List<MulaLanguage> getLanguages() {          //vrací List jazyků pro select
        return mulaLanguageFacase.findAll();
    }
}
```

CmsArticleList.java

ManagedBean pro stránku list.xhtml. Základem je privátní proměnná CmsArticle doplněná o set a get metodu. Dále je zde metoda deleteSelectedArticle(), která se stará o mazání vybraných článků.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.fyx.cms.web;

import javax.ejb.EJB;
import javax.enterprise.context.SessionScoped;
import javax.faces.bean.ManagedBean;
import org.fyx.cms.ejb.CmsLocal;
import org.fyx.cms.entity.CmsArticle;
import org.fyx.linker.web.LinkerLocal;
/**
 * @author ydenek
 */
@ManagedBean
@SessionScoped
public class CmsArticleList {
    @EJB
    CmsLocal cms;
    @EJB
    LinkerLocal linker;
    private CmsArticle selectedArticle;
    /** Creates a new instance of CmsArticleList */
    public CmsArticleList() {
    }

    public CmsLocal getCms() {
        return cms;
    }
    public void setCms(CmsLocal cms) {
        this.cms = cms;
    }

    public CmsArticle getSelectedArticle() {
```

```
        return selectedArticle;
    }

    public int getIdSelectedArticle() {
        return selectedArticle.getId();
    }

    public void setSelectedArticle(CmsArticle selectedArticle) {
        this.selectedArticle = selectedArticle;
    }

    public void deleteSelectedArticle() {
        cms.deleteArticle(selectedArticle);
        linker.removeMenuItem("cms_article_" +
Integer.toString(selectedArticle.getId()));
    }
}
```

CmsArticleShow.java

ManagedBean sloužící k zobrazení článků. Obsahuje tři metody pro inicializaci článku jeho Id, metodu get článku a metodu pro navrácení posledního času editace.

```
package org.fyx.cms.web;

import java.util.Date;
import javax.ejb.EJB;
import javax.enterprise.context.SessionScoped;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.context.FacesContext;
import javax.servlet.http.HttpServletRequest;
import org.fyx.cms.ejb.CmsLocal;
import org.fyx.cms.entity.CmsArticle;
import org.fyx.linker.web.LinkerLocal;
import org.fyx.mula.ejb.MulaLocal;
import org.fyx.web.context.WebContext;
import org.fyx.web.context.WebPage;

/**
 * @author ydenek
 */
@ManagedBean
@SessionScoped
public class CmsArticleShow {
    @EJB
    private CmsLocal cms;
    @EJB
    private MulaLocal mula;
    @EJB
```

```
    private LinkerLocal linker;
    private CmsArticle article;

    /** Creates a new instance of CmsArticleShow */
    public CmsArticleShow() {
    }
    public void initArticle(int idArticle) {
        if (idArticle == 0) {
            FacesMessage msg = new FacesMessage(mula.get("Select the article."),
                    mula.get("You must select article."));
            WebContext.getMessages().add(msg);
            WebPage.redirect(linker.getLink("cms/list.xhtml"));
            article = new CmsArticle();
            return;
        }
        article = cms.getArticleById(idArticle);
    }

    public CmsArticle getArticle() {
        return article;
    }

    public Date getLastModification() {
        return article.getTimeEditation() == null ? article.getTimeCreate() :
article.getTimeEditation();
    }
}
```

CmsObserver.java

Zde dochází k vytvoření dvou rolí. První role je *ARTICLES_READER*, která může články zobrazovat a druhá role je *ARTICLES_MANAGER*, která může články editovat. Dále se zde vytváří menu, kam se přidává odkaz na editaci článků (přístupný jenom pro managera). V cyklu následuje přidání do menu všech článků, které se mají zobrazovat a to jenom pro právo reader.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.fyx.cms.web;

import java.util.List;
import java.util.Observable;
import java.util.Observer;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ejb.LocalBean;
import org.fyx.cms.ejb.CmsLocal;
```

```
import org.fyx.cms.ejb.CmsRoles;
import org.fyx.cms.entity.CmsArticle;
import org.fyx.core.web.event.MenuCreatingEventHandler;
import org.fyx.core.web.event.RoleCreatingEventHandler;
import org.fyx.linker.web.LinkerLocal;
import org.fyx.mula.ejb.MulaLocal;
import org.fyx.secure.ejb.SecureLocal;
import org.fyx.secure.entity.SecureRole;
import org.primefaces.component.menuitem.MenuItem;

/**
 * @author ydenek
 */
@Stateless
@LocalBean
public class CmsObserver implements Observer{
    @EJB
    private SecureLocal secure;
    @EJB
    private LinkerLocal linker;
    @EJB
    private MulaLocal mula;
    @EJB
    private CmsLocal cms;
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof MenuCreatingEventHandler) {
            menuCreate();
        } else if (o instanceof RoleCreatingEventHandler) {
            roleCreate();
        }
    }

    private void menuCreate() {
        MenuItem manageLanguage = new MenuItem();
        manageLanguage.setValue(mula.get("cms_articles"));          //stejne jako id
        manageLanguage.setIcon("ui-icon ui-icon-note");
        manageLanguage.setUrl(linker.getLink("cms/list.xhtml"));
        manageLanguage.setId("cms_articles");               //musi byt stejne jako value
        linker.addMenuItem(manageLanguage, CmsRoles.ARTICLES_MANAGER);
        List<CmsArticle> articles = cms.getArticles();
        for (CmsArticle cmsArticle : articles) {                    //vkládá články do menu
            MenuItem item = new MenuItem();
            item.setValue(mula.get("cms_article_" +
Integer.toString(cmsArticle.getId())));
            item.setIcon("ui-icon ui-icon-document");
            item.setUrl(linker.getLink("cms/show.xhtml?p1=" +
Integer.toString(cmsArticle.getId())));
            item.setId("cms_article_" + Integer.toString(cmsArticle.getId()));
            linker.addMenuItem(item, CmsRoles.ARTICLES_READER);
```

```
        }
    }

    private void roleCreate() {                                    //vytváří role pro Article
        SecureRole role = new SecureRole(CmsRoles.ARTICLES_MANAGER);
        role.setCategory("Content management");
        role.setTitle("Article manager");
        role.setDescription("Can manage articles.");
        secure.addRole(role);
        role = new SecureRole(CmsRoles.ARTICLES_READER);
        role.setCategory("Content management");
        role.setTitle("Article reader");
        role.setDescription("Can read articles.");
        secure.addRole(role);
    }
}
```

CmsLanguageConverter.java

Jsou zde dvě metody, pro konverzi select boxu u formuláře přidání a editace článků. První metoda vrací objekt podle názvu, který dostaneme ze select boxu. Druhá provádí opačnou konverzi, kdy dostaneme textový řetězec pro vyplnění select boxu.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package org.fyx.cms.web;

import javax.ejb.EJB;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import org.fyx.mula.controll.MulaLanguageFacadeLocal;
import org.fyx.mula.entity.MulaLanguage;

/**
 * @author ydenek
 */
@ManagedBean
@RequestScoped
public class LanguageConverter implements Converter {
    @EJB
    private MulaLanguageFacadeLocal mulaLanguageFacade;

    @Override
    public Object getAsObject(FacesContext context, UIComponent component,
String value) {
```

```
        return mulaLanguageFacade.find(value);
    }

    @Override
    public String getAsString(FacesContext context, UIComponent component,
Object value) {
        return ((MulaLanguage) value).getIsoLanguage();
    }
}
```

CmsRoles.java

Definice dvou rolí pro zobrazení a editaci článků.

```
package org.fyx.cms.ejb;

/**
 *
 * @author ydenek
 */
public interface CmsRoles {
    static final String ARTICLES_MANAGER = "cms_articles_manager";
    static final String ARTICLES_READER = "cms_articles_reader";
}
```

Cms.java

Rozhraní pro definici metod s článkem.

```
package org.fyx.cms.ejb;

import java.util.List;
import org.fyx.cms.entity.CmsArticle;

/**
 *
 * @author ydenek
 */
interface Cms {
    void addArticle(CmsArticle ca);
    void editArticle(CmsArticle ca);
    void deleteArticle(CmsArticle ca);
    /**
     * Vraci list vsech clanku.
     * @return
     */
    List<CmsArticle> getArticles();
    public CmsArticle getArticleById(int idArticle);
}
```

CmsLocal.java

Lokání rozhraní rozšiřující rozhraní Cms. Je zde uvedeno jenom pro ukázku, jak oddělit metody lokální a vzdálené.

```
package org.fyx.cms.ejb;
import javax.ejb.Local;

/**
 * @author ydenek
 */
@Local
public interface CmsLocal extends Cms {

}
```

CmsRemote.java

Vzdálené rozhraní rozšiřující rozhraní Cms. Opět je zde jenom jako ukázka, jak oddělit vzdálené a lokální rozhraní.

```
package org.fyx.cms.ejb;
import javax.ejb.Remote;

/**
 * @author ydenek
 */
@Remote
public interface CmsRemote extends Cms {

}
```

CmsBean.java

Obsahuje implementaci rozhraní Cms. Je zde hlavní logika komponenty.

```
package org.fyx.cms.ejb;

import java.util.Date;
import java.util.List;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import org.fyx.cms.controll.CmsArticleFacadeLocal;
import org.fyx.cms.entity.CmsArticle;
import org.fyx.core.PermissionDeniedException;
import org.fyx.core.context.FyxContextFactory;
import org.fyx.mula.ejb.MulaLocal;

/**
 *
 * @author ydenek
```

```
 */
@Stateless
public class CmsBean implements CmsLocal {

    public static final String ARTICLE_PREFIX = "cms_article_";
    @EJB
    private CmsArticleFacadeLocal cmsArticleFacade;
    @EJB
    private MulaLocal mula;
    @Override
    public void addArticle(CmsArticle article) {
        if (!FyxContextFactory.getContext().
                            isUserInRole(CmsRoles.ARTICLES_MANAGER)) {
            throw new PermissionDeniedException();
        }
        article.setAuthor(FyxContextFactory.getContext().getUser());
        article.setTimeCreate(new Date());
        article.setTimeEditation(null);
        cmsArticleFacade.create(article);
        mula.setTranslate(ARTICLE_PREFIX + Integer.toString(article.getId()),
article.getTitle(),
                article.getLanguage().getIsoLanguage());
    }
    @Override
    public void deleteArticle(CmsArticle ca) {
        cmsArticleFacade.remove(ca);
        mula.removeTranslate(ARTICLE_PREFIX + Integer.toString(ca.getId()),
                FyxContextFactory.getContext().getIsoLanguage());

    }
    @Override
    public List<CmsArticle> getArticles() {
        return cmsArticleFacade.findAll();
    }
    @Override
    public CmsArticle getArticleById(int idArticle) {
        return cmsArticleFacade.find(idArticle);
    }

    @Override
    public void editArticle(CmsArticle ca) {
        cmsArticleFacade.edit(ca);
    }
}
```