VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

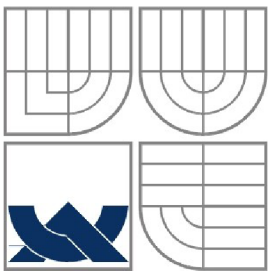
POROVNÁNÍ TECHNOLOGIÍ PRO OBJEKTOVĚ RELAČNÍ MAPOVÁNÍ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

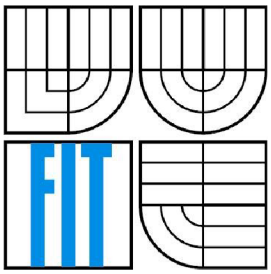
AUTOR PRÁCE
AUTHOR

Bc. Pavel Fatrdla

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POROVNÁNÍ TECHNOLOGIÍ PRO OBJEKTOVĚ RELAČNÍ MAPOVÁNÍ

COMPARISON OF TECHNOLOGIES FOR OBJECT-RELATIONAL MAPPING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Pavel Fatrdla

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. Jaroslav Zendulka, CSc.

BRNO 2010

Abstrakt

Diplomová práce se zabývá současnými technologiemi pro objektově relační mapování (ORM) v jazyce Java. Stručně se věnuje i konkurenčním technikám perzistence objektů v souborech, objektových a objektově-relačních databázích. Hlavním pilířem práce je však perzistence objektů v relačních databázích pomocí rámců pro ORM. Práce začíná studiem obecných postupů a problémů, které tyto rámce musí řešit. Dále jsou zde vybrány konkrétní ORM rámce a ty jsou podrobněji rozebrány a demonstrovány na ukázkové aplikaci. Součástí práce je podrobný popis problémů, se kterými jsem se při jejich implementaci setkal. V závěru práce dochází k ohodnocení a srovnání jednotlivých rámců.

Abstract

Diploma thesis deals with the contemporary object-relational mapping (ORM) technologies for Java. It briefly describes also competing technologies for persisting objects in files, object and object-relational databases. However main part of the thesis is the persistence of objects in relational databases using ORM frameworks. The work begins with studying general methods and issues, that these frameworks have to solve. Next, it chooses and deeply describes some ORM frameworks. They are later demonstrated on the demo application. In the following part there is a description of the problems I have been facing during the implementation of the persistence using these frameworks. Finally, there is an evaluation and a comparison of these frameworks.

Klíčová slova

ORM, objektově relační mapování, JPA, JDO, Hibernate, iBatis, POJO, Java, databáze, DAO, Spring, diagram tříd, databázové schéma, diagram případů použití

Keywords

ORM, object-relational mapping, JPA, JDO, Hibernate, iBatis, POJO, Java, database, DAO, Spring, class diagram, database schema, use-case diagram

Citace

Pavel Fatrdla: Porovnání technologií pro objektově relační mapování, diplomová práce, Brno, FIT VUT v Brně, 2010

Porovnání technologií pro objektově relační mapování

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Jaroslava Zendulky, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Fatrdla
26.5.2010

Poděkování

Na tomto místě bych rád poděkoval doc. Ing. Jaroslavu Zendulkovi, CSc. za poskytnutí užitečných rad a odborné pomoci.

Obsah

1 Úvod

Už od vzniku prvních počítačů se potýkáme s problémem, jak dlouhodobě uchovat a spravovat data. Od té doby se objevilo mnoho možností, jak toho docílit. V současné době jsou asi nejrozšířenějším úložištěm dat relační databáze, ty se používají téměř půl století a jejich pevné postavení neohrozil ani příchod jejich předpokládaného nástupce v podobě objektově orientovaných databází.

Nástup objektově orientovaného přístupu u programovacích jazyků byl však mnohem razantnější než tomu bylo u databází, a tak dnes patří objektové jazyky k nejrozšířenějším. Při vytváření software s kombinací objektového jazyka a relační databáze tak musíme řešit rozdílnost těchto přístupů. Tato práce se zabývá objektově relačním mapováním, které se snaží vývojáře od těchto rozdílností odstínit.

Po úvodu následuje kapitola zabývající se základními možnostmi, které dnes používáme pro perzistenci objektů v jazyce Java. Jako jedno z řešení je zde uvedeno objektově relační mapování, které je pak obecně popsáno v kapitole třetí. Jsou zde vysvětleny problémy, ke kterým při této technice dochází, jako jsou identita, granularita, dědičnost a nachází se zde způsoby, jak se s nimi vypořádat. Čtvrtá kapitola již popisuje vlastnosti vybraných implementací objektově relačního mapování. Konkrétně se jedná o JPA, Hibernate, JDO a iBatis. Následující kapitola pojednává o ukázkové aplikaci. Nejprve se zaměřuji na jednotlivé technologie použité pro analýzu, návrh i samotnou implementaci. Zdůvodnění, proč jsem je vybral a přehled, jak do sebe tyto technologie zapadají. Na to navazuji sběrem a analýzou požadavků, vytvořením a popsáním diagramu případu použití. Po dokončení specifikace se zaměřuji na samotný návrh a implementaci systému. Nalezneme zde konceptuální diagram tříd, podrobnější diagram návrhových tříd, dále pak návrh databázového schématu a popis tvorby uživatelského rozhraní. Konec této kapitoly je věnován samotným problémům, se kterými jsem se setkal v průběhu implementace jednotlivých řešení a testování výsledné aplikace. Kapitola šestá názorně ukazuje rozdíly zápisu metadat u jednotlivých řešení. Jsou zde zobrazeny a okomentovány vybrané problémy při mapování. V předposlední sedmé kapitole, je uvedeno srovnání jednotlivých řešení pomocí stanovených kritérií a ohodnocení jejich plnění jednotlivými použitými objektově relačními rámci.

Součástí této práce je i přiložené CD, obsahující jak zdrojové kódy ukázkové aplikace, tak i elektronickou podobu této práce.

2 Perzistence objektů

V současné době patří k nejrozšířenějším přístupům vývoje software přístup objektově orientovaný. Ten přináší řadu výhod, jako jsou dědičnost, zapouzdření či polymorfizmus. Ve většině případů však potřebujeme objekty z paměti také převést do perzistentního stavu[1]. Toho můžeme dosáhnout například uložením objektů do souboru či databáze.

2.1 Perzistence objektů v souborech

Ukládání do binárních či XML souborů pomocí serializace objektů je sice jednoduché, avšak neřeší důležité funkční požadavky, jako jsou vyhledávání objektů, oprávnění přístupu či konzistenci. Z těchto důvodů se problematice perzistence objektů do souboru nebudu věnovat a zaměřím se na perzistenci objektů v databázi.

2.2 Perzistence objektů v databázi

Ukládání objektů do databází naráží na mizivou rozšířenost objektových databází, které sice existují již od 90.let, ale stále se jim nepodařilo nahradit dosavadní relační databázové systémy. Ty jsou zde již od 70.let a stojí na silných principech relačního modelu dat. Mezi objektově orientovaným a relačním přístupem jsou však značné rozdíly, které je, jak si nyní ukážeme, potřeba řešit.

V následujících podkapitolách se podíváme na vybrané možnosti, jak v jazyce Java řešit perzistenci objektů v databázi. Začneme od jednoduchého přístupu přes JDBC, kde celý proces přemostění musíme řešit ručně. Dále se podíváme na SQLJ, ORM, objektově-relační databáze a dojdeme až k čistě objektovým databázím, kde už žádné přemostění řešit nemusíme.

2.2.1 JDBC

Java Database Conectivity (JDBC) API je standardem jazyka Java pro přímý SQL přístup k databázi. Veškeré přemostění mezi objektovým a relačním přístupem musí řešit programátor. Toto přemostění je natvrdo naprogramováno ve zdrojovém kódu. V tomto případě má programátor celý proces pod kontrolou, což vede k vyšší efektivnosti kladení dotazů na databázi. Avšak podle provedených studií[2] zabírá toto ruční přemostění, které musí programátor provádět, kolem 30% času.

---

1 Stav, ve kterém jsou data trvale uložena, a to nejen po dobu běhu programu, ale i mezi jeho ukončením a opětovným spuštěním.

2 Popsané ve zdroji [1]. Jednalo se o oslovování firem a jejich programátorů na osobní odhad.

2.2.2 SQLJ

SQLJ je technologie založená stejně tak jako JDBC na přímém přístupu do databáze. Jedná se však o hostitelskou verzi SQL pro jazyk Java. SQL příkazy jsou součástí kódu programu a jsou uzavírány do bloku uvozeném značkou `#sql`. V průběhu kompilace jsou pak tyto bloky kontrolovány a programátor je tak na případnou chybu upozorněn mnohem dříve, než tomu bylo u JDBC. Mezi výhody SQLJ patří silná typová kontrola, kratší kód. Naopak k nevýhodám můžeme zařadit nízkou podporu u databázových řešení či vývojových prostředí a nutnost dalšího kroku pro zpracovávání SQLJ bloků už při kompilaci.

Více informací lze nalézt ve zdroji [6].

2.2.3 Objektově relační mapování (ORM)

Jedná se o techniku, která umožňuje zjednodušit tvorbu přemostění mezi objektovým a relačním přístupem a tento proces částečně automatizovat. Tím i zvýšit efektivnost programátora a snížit náklady na vývoj software. Za tímto účelem pak vznikly standardy jako JPA[3], JDO[4]. Mezi nevýhody ORM můžeme zařadit nižší efektivnost při pokládání automaticky generovaných dotazů. Dále pak režii spojenou s použitím samotného nástroje pro ORM.

2.2.4 Objektově-relační databáze

Jedná se o databáze, které se snaží sloučit objektový a relační přístup. V roce 1999 byl vytvořen standard SQL:1999[5]. Ten přinesl podporu pro objektově-relační databáze. Zavedl uživatelem definované typy (UDT). UDT je struktura podporující jednoduchou dědičnost, složená ze základních datových typů, definovaná uživatelem. Standard také zavedl typované tabulky, kde tabulka je typu UDT a jednotlivé řádky reprezentují objekty. Dalo by se tedy říct, že typovaná tabulka je třída a její řádky jsou instance této třídy. Dále je zde podpora agregace třídy jinou třídou, čímž se ovšem dostáváme do sporu s první normální formou, která nám říká, že všechny atributy mají být atomické.

Data jsou tedy stále reprezentována v tabulkách. Přemostění probíhá ručně definováním UDT pro jednotlivé třídy. Veškeré mapování a psaní dotazů je tedy stále v rukou programátora.

---

3 Viz. kapitola 4.1

4 Viz. kapitola 4.3

5 Standard se často objevuje pod označením SQL3. Jedná se konkrétně o standard ISO/IEC 9075, který se skládá z pěti částí SQL/Framework, SQL/Foundation, SQL/CLI, SQL/PSM, SQL/Bindings. Více informací lze nalézt na http://www.iso.org

2.2.5 Objektové databáze

Objektové databáze rozšiřují objektově orientované jazyky o databázové schopnosti. Objektové databáze mohou spolu s daty obsahovat také aplikační logiku objektů. [10] Objektový model aplikace a objektový model databáze jsou si velmi podobné, což zvyšuje udržovatelnost a srozumitelnost. Jsou ale méně efektivní, nepříliš rozšířené, neexistují tak sofistikované nástroje pro práci se schématem, zálohování, reportování, dolování z dat, migraci dat, databázovou administraci, jako je tomu u relačních databází.

3 ORM

ORM je technika snažící se automatizovat a zjednodušit přemostění mezi dvěma paradigmaty, a to objektově orientovaným a relačním. Ta jsou však natolik rozdílná, že při těchto snahách se nejdříve musíme vypořádat s problémy, které při tomto přemostění vznikají. Podíváme se tedy na tyto základní problémy a na způsoby, kterými jsou možné řešit. Nejdříve se však blíže seznámíme se samotným principem, jak ORM funguje.

Při psaní této kapitoly jsem čerpal informace ze zdrojů [1], [2] a [8].

Princip ORM

Při použití ORM máme za cíl přistupovat k datům pomocí objektového přístupu, a to ať už jsou ve skutečnosti uložena v relační databázi, či jakémkoliv jiném úložišti. Proto ORM rámce poskytují vlastní objektové dotazovací jazyky, pomocí kterých můžeme k datům přistupovat. Rámec zajistí automatické generování potřebného nativního dotazu, který je pak kladen datovému úložišti. V případě použití relačních databází tedy dochází ke generování do nativního SQL.

K tomu, aby rámec mohl takto pracovat, potřebuje znát dvě věci. Tou první je popis datového úložiště. O jaký typ úložiště se jedná, jaké jsou přístupové údaje, zda chceme využít výhody opožděného načítání[6] a další potřebné informace lišící se dle použitého rámce. Druhou věcí, kterou musíme rámci dodat, je popis tříd, které budeme chtít ukládat. Ten se provádí pomocí metadat, která jsou uložena v samostatném souboru, či pomocí anotací. V těchto metadatech sdělíme rámci, která proměnná třídy slouží k identifikaci, či které vlastnosti třídy chceme ukládat. V případě relačních databází, například i název sloupců, na které mají být jednotlivé vlastnosti namapovány. Dále pak můžeme specifikovat, jaké vztahy mezi sebou třídy mají, a jak by se měly promítnout do datového úložiště. Možnosti nastavení se liší u jednotlivých rámců a použitých datových úložišť.

V konečném důsledku pak klademe objektové dotazy ORM, které je překládá a volá na úložiště. Získanými daty pak naplní požadované objekty, které jsou vráceny jako výsledek objektového dotazu. Programátor je tedy osvobozen od rutinního psaní přemostění pro ukládání či načítaní dat do objektů. Dochází zde tedy k abstrakci, kdy programátor nepřichází do styku s konkrétním úložištěm a veškerá komunikace s úložištěm prochází přes ORM.

Výhody užití ORM jsou tedy zřejmé. Objektový přístup ke kladení dotazů u jednoduchých úkonů, jako uložení, smazání či načtení konkrétního objektu dané třídy pomocí identifikátoru, nemusíme psát dotazy vůbec. Ty jsou vygenerovány na základě zadaných metadat. Další nespornou výhodou je nezávislost na konkrétním úložišti. Pokud potřebujeme změnit typ úložiště, pouze změníme nastavení rámce. Veškeré dotazy, jak už bylo zmíněno, jsou totiž zapsány v objektovém

6 Opožděné načítání je popsáno v kapitole 4.

dotazovacím jazyku rámce[7]. Po změně úložiště bude tedy jediná změna ta, že ORM rámec bude generovat jiné nativní dotazy a zasílat je na jiné úložiště.

Mezi nevýhody můžeme uvést nižší efektivnost vygenerovaného dotazu, než by tomu mohlo být u ručního přístupu. Kvůli pokládání dotazů v objektovém jazyce rámce a ne jazyce úložiště, nám pak může například chybět vlastnost úložiště, kterou rámec nepodporuje.

Podívejme se tedy nyní, jaké problémy musí ORM rámce řešit při tvorbě přemostění, aby nám tyto výhody mohlo poskytnout.

3.1 Identita

V objektovém přístupu jsme zvyklí, že každý objekt má svoji jedinečnou identitu. I když jsou objekty ve stejném stavu, tak nejsou považovány za identické. Pokud se podíváme například na jazyk Java, máme zde dvě možnosti porovnání identity objektu. Jednou z nich je porovnání `foo == bar`, kde dochází ke porovnání pomocí adresy umístění objektu v paměti. Druhou možností je pomocí implementace metody `equals()`[8].

Relační databáze však problém identity řeší hodnotou primárního klíče. Jak by se tedy měl ORM zachovat, pokud chceme z databáze nahrát dvakrát po sobě objekt se stejným primárním klíčem. Měl by vytvořit nový objekt tak, že by platilo `obj1.getId().equals(obj2.getId())`, ale neplatilo `obj1 == obj2`? Nebo zjistit si, že objekt se stejným primárním klíčem již nahrával a pouze vrátit odkaz na již existující instanci `obj1`, takže by platily obě předchozí podmínky. Druhý způsob vypadá rozhodně zajímavěji, ovšem vzniknou nám zde problémy s konzistencí. Představme si případ, kdy načteme `obj1` z databáze, na něm provedeme změny a v tomto okamžiku chceme nahrát z databáze `obj2`. Jak by se měl zachovat ORM a na jaké úrovni by měl toto řešení provádět? Na úrovni sezení, připojení nebo mezi všemi připojeními? Některé implementace ORM řeší tento problém zamykáním načtených řádků či verzováním. Nejedná se však o problém vlastní ORM, problém aktuálnosti dat uložených v perzistentním stavu vůči datům načteným v paměti se při násobném přístupu vyskytuje i při přímé perzistenci do relačních databází nebo souborů.

---

7 Některé ORM umožňují i dotazování přímo v nativním jazyku úložiště. Tím ovšem vznikne závislost aplikace na konkrétním datovém úložišti.

8 Pokud neimplementujeme na našem objektu metodu `equals()`, je děděna od svého předka `Object`. Zde je tato metoda implicitně implementována jako `foo == bar`. Tedy porovnání pomocí adresy umístění v paměti.

Rámce tento problém řeší porovnáváním primárních klíčů, a to většinou na úrovni sezení. Pokud tedy v rámci jednoho sezení klademe dotaz, jehož výsledkem je stejný objekt, je nám vrácen ukazatel na již načtený objekt. U některých rámců můžeme úroveň změnit i na úroveň připojení.

3.2 Granularita

Granularita je pojem definující velikosti či složitostí typů, se kterými pracujeme. Máme tedy nejjednodušší typy s nejnižší granularitou jako `String`, `Integer`. Z nich pak můžeme vytvořit nový složitější typ s vyšší granularitou, který se z těchto typů skládá.

Problém granularity můžeme snadno popsat následujícím příkladem[9]. Mějme třídu `Employee` reprezentující zaměstnance. U toho bychom rádi měli uložen záznam o jeho adrese. Adresa se ale skládá z více informací, jako je ulice, číslo popisné, město. Určitě by bylo tedy vhodné tyto informace zapouzdřit do samostatné třídy `Address` reprezentující adresu. Pak může třída `Employee` agregovat `Address`.

Podívejme se nyní na náš jednoduchý příklad od největší granularity u třídy `Employee`, na nižší u třídy `Address`, až po jednoduchou vlastnost číslo popisné, reprezentovanou například třídou `Integer`. Vidíme, že úroveň granularity u tohoto příkladu je tři. Pokud se zamyslíme, zjistíme, že úroveň granularity u objektového přístupu není omezená. U našeho příkladu bychom mohli dosáhnout vyšší granularity vytvořením nové třídy zapouzdřující některé z vlastností třídy `Address` a následnou agregací této třídy třídou `Address`.

Pokud se podíváme jakou úroveň nám nabízí relační databáze, zjistíme, že jsme omezeni pouze na dvě úrovně, a to pouze tabulky a sloupce. Tento problém se dá jednoduše řešit tak, že každou agregovanou třídu rozdělíme na více sloupců. Tabulka pro náš příklad by tedy vypadala[10].

```
create table EMPLOYEE (
    ID_EMPLOYEE int unsigned auto_increment,
    NAME varchar(255) not null,
    ADDRESS_STREET varchar(255),
    ADDRESS_STREET_NO int unsigned,
    ADDRESS_CITY varchar(255),
    primary key (ID_EMPLOYEE)
)
```

9 Část příkladu převzata z [1].

10 Příklad je zapsán v dialektu jazyka MySQL.

U objektově-relačních databází už je díky UDT situace jiná. Můžeme pro třídu `Address` vytvořit nový typ `tAddress`. Příklad by pak mohl vypadat takto[11].

```
create type tAddress as object (
    STREET varchar(255),
    STREET_NO int unsigned,
    CITY varchar(255)
)
create table EMPLOYEE (
    ID_EMPLOYEE int unsigned auto_increment,
    NAME varchar(255) not null,
    ADDRESS tAddress,
    primary key (ID_EMPLOYEE)
)
```

ORM rámce řeší většinou problém granularity rozdělením agregovaných tříd na nejjednodušší typy. Ty jsou pak uloženy jako další sloupce. Výsledná struktura tedy dopadne stejně, jak jsme mohli vidět na předminulém příkladě.

3.3 Asociace

V doménovém modelu máme asociace, které reprezentují vztahy mezi jednotlivými třídami. Asociace mají různé vlastnosti jako viditelnost či násobnost.

U relačních databázích jsou tyto asociace reprezentovány jako sloupce s cizími klíči. V nich jsme zvyklí procházet oběma směry asociace pomocí spojování tabulek. Vyskytují se zde vztahy s násobností 1:1, 1:N. Složitější vztah M:N se zde řeší rozbitím vztahu na dva vztahy M:1 a 1:N.

Naproti tomu objektový model řeší asociace pomocí referencí. Ty jsou jednosměrné[12] a umožňují i vztah M:N. ORM rámce však u násobné asociace nepoznají, o jakou násobnost se jedná. Můžeme se o tom přesvědčit na následujícím příkladě se třídami `Employee` a `Team`.

Rámce tyto rozdíly řeší následovně. Násobnost těchto vztahů musí být programátorem popsána v metadatech. Reprezentaci M:N řeší standardně rozbitím vztahu na dva vztahy M:1 a 1:N.

---

11 Příklad je zapsán v pseoudo-dialektu jazyka MySQL, který jsem rozšířil o UDT.

12 Pokud objekt `A` má referenci na objekt `B`. Je pak objekt `B` viditelný pro `A`, ale ne naopak. Pokud bychom chtěli zaručit viditelnost i v druhém směru, museli bychom ji definovat i v třídě `B`. Takovýto nově vzniklý vztah by jsme pak nazývali obousměrný.

```
/* ORM rámec nemá jak zjistit zda se jedná ze strany od
třídy Team k třídě Employee jedná o vztah 1:N nebo M:N. */
public class Employee {
    ...
}

public class Team {
    private Set<Employee> employees;
    ...
}
```

3.4 Navigace dat

Způsob, jakým se odlišuje navigace dat v objektovém a relačním přístupu, je zásadní. Navigace v objektovém přístupu je realizována za pomocí referencí, kterými jsou objekty provázány. V předchozím případě by například mohla vypadat navigace, jak se od třídy `Employee` dostat ke jménu vedoucího oddělení, ve kterém zaměstnanec pracuje, vypadat například takto: `load(Employee.class, PK_EMPLOYEE).getDepartment().getSupervisor().getName()`. V relačním přístupu by však tato varianta nebyla příliš efektivní kvůli několikanásobnému pokládání dotazu.

```
/* Pod proměnnými PK_XXX jsou primární klíče
jednotlivých instancí. */
select PK_DEPARTMENT
from EMPLOYEE where ID_EMPLOYEE = PK_EMPLOYEE

select PK_EMPLOYEE
from DEPARTMENT where ID_DEPARTMENT = PK_DEPARTEMENT

select *
from EMPLOYEE where ID_EMPLOYEE = PK_EMPLOYEE
```

Efektivní cestou, jak v relačním přístupu dojít ke stejnému výsledku, je použít spojování tabulek, které obsahují požadovaná data nebo data potřebná k navigaci, a z nich si nechat navrátit pouze požadované hodnoty. Zde ale narážíme na problém, jak by mělo ORM vědět, že procházíme všechny tyto objekty pouze za účelem zjištění jména nadřízeného a ne například jeho platu.

ORM rámce řeší tento problém způsobem, kdy při popisu naší třídy metadaty nastavíme, které vlastnosti třídy se mají včasně načítat[13]. Tímto způsobem rámci sdělíme, které vlastnosti budeme potřebovat. Rámec pak použije při dotazování spojování tabulek a načte i požadované vlastnosti v jednom dotazu.

3.5 Mapování

Entita

Entita je objekt aplikační domény světa, který je rozlišitelný od ostatních objektů, o nichž potřebujeme mít informace v databázi. [7] Má svůj životní cyklus a může existovat nezávisle na ostatních entitách.

Komponenta

Komponenta je objekt, který je závislý na entitě pro její identitu. Komponenta žádnou svoji identitu totiž nemá. Je pouze částí entity, která byla od entity odříznuta a uložena do samostatného objektu. [8]

V této části se zaměříme na mapování jednotlivých struktur na relační databázi. Začneme od těch nejjednodušších typů, přes výčtové typy, komponenty, až k samotnému mapování entit.

Nejdříve se však podíváme, jaký je rozdíl mezi entitami a komponentami. Asi nejjednodušší vysvětlení nalezneme pomocí jazyka UML. Mezi entitou a komponentou je totiž vztah kompozice.

Kompozice je typ asociace, která určuje, že jeden objekt je část druhého objektu, v našem případě tedy komponenta je částí entity. V tomto vztahu platí, že část patří pouze jednomu celku, a že za životní cyklus části je odpovědný celek. Tuto vazbu už jsme mohli vidět v příkladu kapitoly 3.2 mezi třídou `Employee` a `Address`.

3.5.1 Mapování jednoduchých typů

Mapování jednoduchých typů jako `String`, `Integer`, `Long`, `Double`, `Boolean` či `Byte[]` nebo `Character[]` není problém. Podpora těchto typů totiž v ekvivalentní nebo blízké podobě existuje i na straně relačních databází, jako například typy `VARCHAR`, `INTEGER`, `DOUBLE`, `BOOLEAN`, `BLOB`, `CLOB`. U jednotlivých dialektů SQL se tyto názvy můžou lišit, ale jejich podpora je v podstatě u většiny relačních řešení.

13 Včasné načítání je popsáno v kapitole 4.

3.5.2 Mapování výčtových typů

Výčtový typ je uživatelem definovaný typ, jak již název vypovídá, jedná se o hodnoty, které programátor vyjmenuje. Výčtový typ může být reprezentován v jednotlivých jazycích různě. Podívejme se například na jazyk Java.

Každá hodnota výčtového typu je implicitně ohodnocena ordinálním číslem, jež reprezentuje pořadí, v jakém byly hodnoty vyjmenovány. Z tohoto pohledu samozřejmě vyplývá namapovat tento typ jako číselný. Tím nám ale může později vzniknout nekonzistentní stav. Představme si, že máme typ `EmployeeLevel` a v něm hodnoty `SENIOR` s ordinálním číslem `0` a `VETERAN` s ordinálním číslem `1`. Tento typ je používán v nějaké třídě, jejíž objekty jsou již perzistovány v databázi. Nyní budeme chtít přidat hodnotu do tohoto typu např. `JUNIOR`. Dáme ji jako první v pořadí, čímž ovšem

```
public enum EmployeeLevel (
    SENIOR,
    VETERAN
)
```

změníme ordinální čísla všech hodnot. Museli bychom tedy ručně na databázi změnit všechny záznamy, kde byly tyto hodnoty uloženy.

Druhou možností je neuložit do databáze výčtový typ jako číselný typ, ale jako typ řetězec. Do databáze se tedy nebude ukládat ordinální číslo hodnoty, ale její název. Tím se zbavíme závislosti na pořadí hodnot, ale při změně názvu dostaneme stejný problém. Tato situace by již neměla být tak častá, jako je tomu u změny pořadí.

ORM rámce nám dávají na výběr způsob uložení, buď jako ordinální číslo, nebo jako řetězec. Toto nastavení se provádí v metadatech popisujících tuto vlastnost třídy.

3.5.3 Mapování komponent

Jak už jsme si říkali, komponenta je částí celku entity a sama o sobě nemá identitu. Tuto skutečnost můžeme namapovat tak, že atributy komponenty namapujeme na nové sloupce entity, jak je tomu na obrázku 3.1 z našeho příkladu s třídou `Employee` a `Address`.

EMPLOYEE		
+ID_EMPLOYEE	**integer(10)**	**Nullable = false**
NAME	varchar(255)	Nullable = false
ADDRESS_STREET	varchar(255)	Nullable = false
ADDRESS_STREET_NO	varchar(255)	Nullable = false
ADDRESS_CITY	varchar(255)	Nullable = false

Ilustrace 3.1: Schéma příkladu mapování komponent (ER diagram)

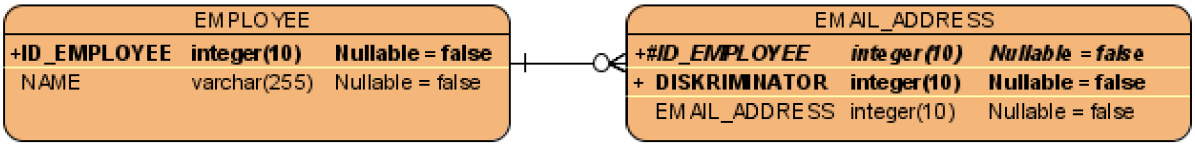
3.5.4 Mapování entit

Když už jsme si ukázali, jak namapovat jednoduché typy, výčtové typy či komponenty a řekli jsme si o identitě a asociacích mezi entitami, namapování jednoduchých entit by pro nás neměl být problém.

3.6 Mapování kolekcí

3.6.1 Komponent

Kolekci komponent už nemůžeme namapovat do nových sloupců entity, jak tomu bylo u jedné komponenty. V tomto případě musíme vytvořit druhou tabulku, ve které bude komponenta uložena. Jako primární klíč bude mít cizí klíč entity a svůj diskriminátor[14], jak můžeme vidět na obrázku 3.2.



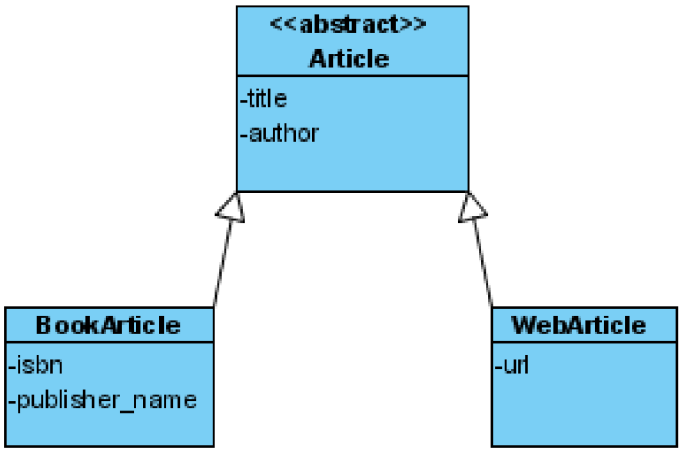
Ilustrace 3.2: Mapování kolekce komponent (ER diagram)

3.7 Dědičnost

Když se zamyslíme, jak byl mělo vypadat mapování dědičnosti na relační databázi, asi jako první řešení nás napadne možnost pro každou podtřídu vytvořit jednu tabulku. Toto řešení samozřejmě funguje bez problému než nastane doba, kdy potřebujeme využít, v objektovém přístupu, docela samozřejmou vlastnost a tou je polymorfizmus.

Nyní si zde ukážeme některé strategie, jak dosáhnout namapování dědičnosti do relační databáze. Tyto strategie se spíše hodí k přístupu shora dolů, tedy přístupu, kdy máme vytvořený model domény a z něho chceme vytvořit databázové schéma. Jak si později ukážeme, některé implementace ORM jsou zaměřeny právě na přístup shora dolů, tedy hlavně přístup, kdy například začínáme na novém projektu. Zatímco jiné implementace se výborně hodí i pro již nasazené projekty, kde mapujeme zdola nahoru, tudíž od pevného schématu databáze, na který mapujeme objekty naší domény.

14 Jedná se o částečný identifikátor používaný pro zdůraznění slabé entitní množiny. Diskriminátor není jedinečný v rámci všech komponent uložených v tabulce `EMAIL_ADDRESS`, ale je jedinečný v rámci všech komponent patřící konkrétní entitě z tabulky `EMPLOYEE`.



Ilustrace 3.3: Dědičnost v ORM - ukázkový příklad (Diagram tříd)

Různá řešení si ukážeme na následujícím případě, mějme třídu `BookArticle` reprezentující knižní články a třídu `WebArticle` reprezentující webové články. Jak vidíme na obrázku 3.3, obě tyto třídy jsou potomky abstraktní třídy `Article` a dědí od ní atributy `title` a `author`.

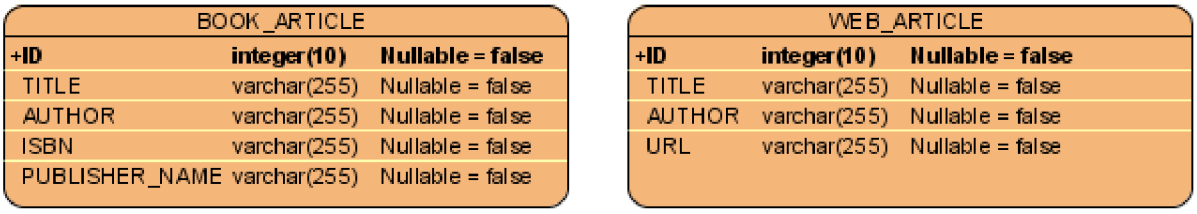
Polymorfní dotazy

Polymorfní dotazy si asi nejlépe vysvětlíme pomocí příkladu z obrázku 3.3. Máme danou hierarchii dědičnosti, kde máme rodiče, abstraktní třídu `Article` a dva potomky `BookArticle` a `WebArticle`. Polymorfním dotazem pak nazýváme dotaz, který je kladen nad částí této hierarchie a předem nevíme jaké jednotlivé třídy hierarchie bude výsledek obsahovat. Například dotaz, kolik článků obsahuje v názvu nějaký výraz. Víme, že výsledkem tohoto dotazu bude počet instancí tříd `Article` splňujících tuto podmínku. Nevíme však už, o které konkrétní třídy se jedná. Jestli tuto podmínku splňovaly pouze třídy `BookArticle` nebo `WebArticle` nebo jestli výsledek měl zástupce v obou třídách.

3.7.1 Tabulka pro konkrétní třídu

V této strategii je nutné, aby ORM na konkrétní databázi zajistilo, aby atribut `ID` byl jedinečný pro všechny potomky podtříd. Lze toho například dosáhnout pomocí vytvoření sekvencí. Na rozdíl od ostatních strategii nepotřebuje žádný přídavný uložený atribut jako diskriminátor tříd.

Negativum tohoto řešení je, že polymorfní dotazy jsou mnohem náročnější než u jiných řešení. Tato náročnost je způsobena faktem, že musí proběhnout dotazy samostatně pro každou podtřídu, nebo dotaz zahrnující všechny tabulky pomocí operace `UNION`, která je náročná při větším množství dat.



Ilustrace 3.4: Tabulka pro konkrétní třídu (ER diagram)

Databázové schéma pro tuto strategii by mohlo vypadat jako na obrázku 3.4. Mezi pozitiva této strategie patří případ, kdy jsou tabulky nerovnoměrně objemově zatížené a většina dotazů je kladena pouze nad tabulkou s méně daty. To by mělo za důsledek výrazné zvýšení efektivnosti oproti ostatním řešením.

Vícenásobný dotaz

```
/* První dotaz na získání všech knižních článků. */
select ID, TITLE, AUTHOR, ISBN, PUBLISHER_NAME
from BOOK_ARTICLE

/* Druhý dotaz na získání všech webových článků. */
select ID, TITLE, AUTHOR, URL
from WEB_ARTICLE

/* Na úrovni aplikace pak dojde k vytvoření všech instancí
knižních a webových článků a vložení do kolekce
abstraktního článku. */
```

Union dotaz

```
/* Sloupec CLAZZ slouží jako diskriminátor. */
select CLAZZ, ID, TITLE, AUTHOR, ISBN, PULISHER_NAME, URL
from
    (
        select 1 as CLAZZ, ID, TITLE, AUTHOR,
            ISBN, PUBLISHER_NAME, NULL as URL
        from BOOK_ARTICLE
    union
        select 2 as CLAZZ, ID, TITLE, AUTHOR,
            NULL as ISBN, NULL as PUBLISHER_NAME, URL
        from WEB_ARTICLE
    )
```

3.7.2 Společná tabulka

Asi nejběžnější strategie. Všechny podtřídy jsou uloženy v jedné tabulce, ta pak má atributy všech podtříd, jak můžeme vidět na obrázku 3.5. Každý řádek tabulky reprezentuje objekt jedné z konkrétních tříd.

Ne všechny sloupce musí mít v daném řádku hodnotu, protože k různým podtřídám patří různé atributy. Jak vidíme v našem případě tabulka obsahuje společné atributy jako `ID`, `ARTICLE_TYPE`, `TITLE`, `AUTHOR`. Dále pak atributy patřící k podtřídě `BookArticle` uvozeny prefixem `BA_` a atributy patřící k podtřídě `WebArticle` uvozeny prefixem `WA_`.

ARTICLE		
+ID	**integer(10)**	**Nullable = false**
ARTICLE_TYPE	integer(10)	Nullable = false
TITLE	varchar(255)	Nullable = false
AUTHOR	varchar(255)	Nullable = false
BA_ISBN	varchar(255)	Nullable = true
BA_PUBLISHER_NAME	varchar(255)	Nullable = true
WA_URL	varchar(255)	Nullable = true

Ilustrace 3.5: Společná tabulka (ER diagram)

Mezi negativa, která z této strategie vyplývají, je fakt, že jednotlivé řádky tabulky obsahují více sloupců než jednotlivé podtřídy potřebují a velké množství sloupců zůstává prázdné. Z této skutečnosti samozřejmě vyplývá, že všechny atributy pro podtřídy musí byt `nullable`. Další negativa plynoucí z tohoto řešení jsou příliš velké a pomalé indexy či vyšší náchylnost k přetěžování tabulky oproti ostatním řešením.

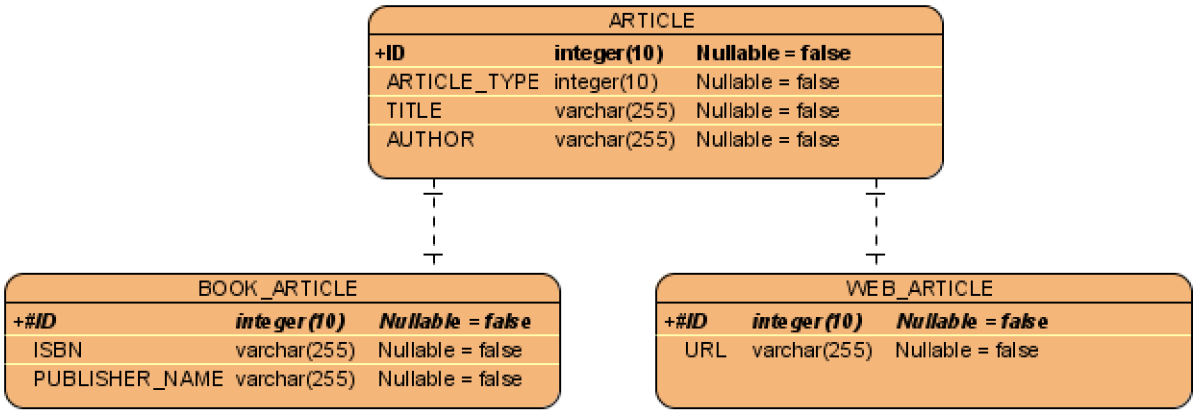
Diskriminátor

Pokud se na obrázek 3.5 podíváme důkladně, zjistíme, že se v tabulce vyskytuje další sloupec `ARTICLE_TYPE`, který nepatří k žádnému atributu mapovaných tříd. Jedná se o tzv. diskriminátor, tedy sloupec, podle kterého poznáme, jaká konkrétní podtřída se v jednotlivých řádcích nachází.

3.7.3 Join strategie

Asi nejpřirozenější je další ze strategií. Pro každou konkrétní i abstraktní třídu vytváří novou tabulku, což se asi nejvíce podobá logice z objektového přístupu. Dostaneme tak schéma, které můžeme vidět na obrázku 3.6.

Jak vidíme na našem příkladu, máme namapovanou abstraktní třídu `Article` na tabulku `ARTICLE`. Ta obsahuje společný identifikátor, diskriminátor a společné atributy, které od ní ostatní třídy dědí. Tabulky `BOOK_ARTICLE` a `WEB_ARTICLE` obsahují identifikátor `ID`, který je primárním a zároveň i cizím klíčem referujícím do tabulky `ARTICLE`.



Ilustrace 3.6: Join strategie (ER diagram)

Pokud tedy potřebujeme dostat všechny články můžeme použít techniku `join`.

```
select *
from ARTICLE
left outer join BOOK_ARTICLE using (ID)
left outer join WEB_ARTICLE using (ID)
```

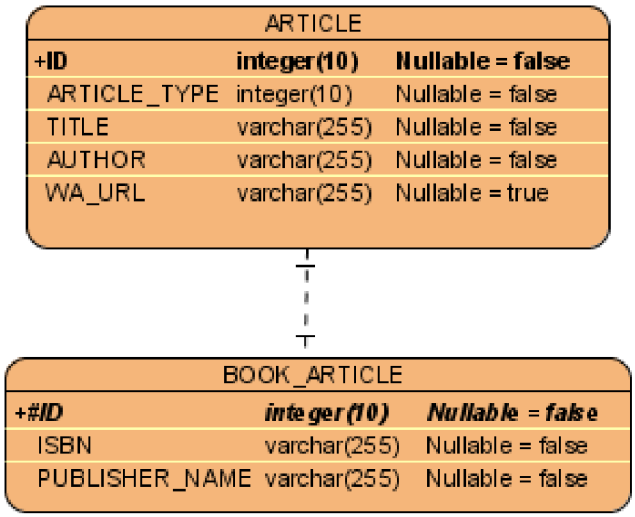
Výhodou této strategie je, že je prostorově efektivní při ukládání dat, která jsou ve více podtřídách naší hierarchie.

Nevýhoda vyplývá už ze samotného principu této strategie. Čím větší počet podtříd a složitější hierarchii tříd máme, tím vícekrát dochází k spojování pomocí techniky `join`. Ta je ovšem při velkém počtu dat a násobném opakování výpočetně náročnější.

3.7.4 Míchaná strategie

Další ze strategií můžeme nalézt namícháním předchozích řešení. Představme si například situaci, kde víme, že v našem systému se bude objevovat mnohonásobně častěji objekt `WebArticle` než `BookArticle`. Ve většině případů by tedy docházelo ke spojování tabulky `ARTICLE` a tabulky `WEB_ARTICLE`. V takovém případě můžeme použít míchané řešení, které kombinuje jak techniky ze strategie Společná tabulka tak Join strategie, jak můžeme vidět na obrázku 3.7.

Pokud tedy budeme nejčastěji chtít načíst objekty třídy `WebArticle`, poslouží nám výkonově nenáročný dotaz pouze na tabulku `Article`. Pokud však budeme chtít načítat všechny nebo jen objekty třídy `BookArticle`, budeme muset použít náročnější operaci `join`.



Ilustrace 3.7: Míchaná strategie (ER diagram)

```
/* Předpokládejme, že pro třídu WebArticle je hodnota
ARTICLE_TYPE 1 a pro třídu BookArticle je 2. */

/* Nejčastější načítaní pouze objektů WebArticle */
select *
from ARTICLE
where ARTICLE_TYPE = 1

/* Načítání všech objektů WebArticle a BookArticle */
select *
from ARTICLE
left outer join BOOK_ARTICLE using (ID)
```

3.7.5 Výběr vhodné strategie

Výběr vhodné strategie by se dal dle [1] rozhodnout podle následujících předpokladů:

1. Pokud nevyžadujeme polymorfní dotazy a nemáme třídu s asociací na rodičovskou abstraktní třídu `Article`, nabízí se nám použití strategie 3.7.1 Tabulka pro konkrétní třídu.

2. Pokud vyžadujeme polymorfní dotazy nebo máme třídu s asociací na rodičovskou abstraktní třídu `Article`. Dále pak platí, že potomci této třídy deklarují relativně málo nových atributů[15]. V tom případě je dobré použít strategii 3.7.2 Společná tabulka.

3. Pokud vyžadujeme polymorfní dotazy a potomci se vyznačují velkým počtem různých atributů, měli bychom použít strategii 3.7.3 Join strategie. V některých případech je lepší použít 3.7.1 Tabulka pro konkrétní třídu, a to v těch, kdy se jedná o velmi složitou hierarchii dědičnosti[16].

15 Tzn. že hlavní rozdíly mezi potomky jsou v jejich metodách(chování) a ne v atributech(vlastnostech).

16 Je tomu tak kvůli různé náročnosti join u Join strategie a union u strategie Tabulka pro konkrétní třídu u velkého počtu podtříd.

4 ORM implementace

V této kapitole představím vybrané ORM implementace a jejich základní charakteristiky. Konkrétní implementace jsem si vybral na základě jejich rozšíření a licencí, pod kterými jsou šířeny. Preferoval jsem svobodná řešení před placenými.

Než však takto učiním, popíši a vysvětlím zde některé pojmy týkající se ORM implementací, které je potřeba znát.

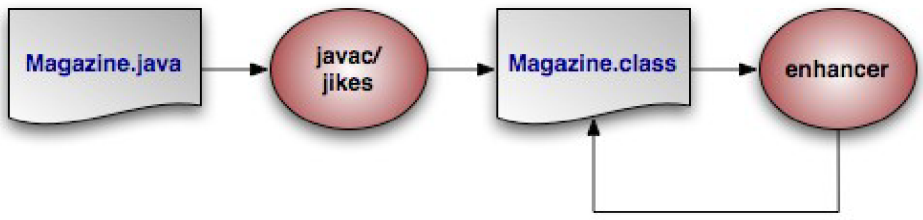
Enhancer

Některé implementace ORM vyžadují obohacení třídy o sadu metod, jež jsou pak využity při perzistenci instancí těchto tříd. Toto obohacování se dá provádět následujícími způsoby [13]:

1. Manuální rozšíření kódu
   U tohoto řešení dochází k nutnosti implementovat ručně rozhraní daného ORM řešení. Například u JDO se jedná o rozhraní `PersistenceCapable`. K tomu je však zapotřebí znát specifikaci ORM a podle ní při implementaci postupovat. Z tohoto důvodu není tento postup doporučován. Zbytečně totiž může dojít k zanesení těžko odhalitelných chyb.

2. Použitím nástroje pro obohacení zdrojového kódu
   Nástroj načte zdrojový kód a obohatí ho o vygenerovanou sadu metod potřebnou pro perzistenci. Toto řešení není příliš rozšířené.

3. Použitím nástroje pro obohacení byte kódu
   Asi nejrozšířenějším způsobem obohacení třídy je obohacování byte kódu (na obr. 4.1). Třída je nejdříve zkompilována do byte kódu. Následně je byte kód předán enhanceru,



Ilustrace 4.1: Obrázek ilustruje průběh kompilace při obohacování byte kódu enhancrem. Převzato z [12].

který jej obohatí o potřebnou sadu metod. Jedná se tedy o přidání dalšího kroku do procesu sestavení aplikace.

4. Obohacení za běhu [11]
   Při tomto způsobu dochází k obohacení třídy až v době běhu při pokusu o její načtení. Dochází tak ke zpomalení inicializace aplikace při každém spuštění z důvodu nutnosti vyhledání metadat pro příslušnou třídu a následného obohacení.

Opožděné/Včasné načítání (Lazy/Eager fetching)

V některých případech víme, že různé části entity budou jen velmi zřídka používány. Proto můžeme za účelem zvýšení výkonu načítat pouze části entit, u kterých víme, že se k nim bude často přistupovat. Zbylé části entity načteme pouze v případě, pokud budou potřeba. Tento postup pak nazýváme opožděným načítáním. [2]

Opačným postupem k opožděnému načítání je včasné načítání. Tento postup je vhodný v případech, kdy víme, že budeme k této části entity ve většině případů přistupovat.

Vyrovnávací paměť (Cache)

Mnoho aplikací při svém běhu načítá data častěji než ukládá. Poměr čtení vůči zápisu tak bývá ve výrazném nepoměru ve prospěch čtení. Přitom se často jedná o načítání stejných nezměněných dat z datového úložiště, která už jsme před nějakou dobou načítali.

Operace s databází často bývají hlavním zdrojem problému s rychlostí naší aplikace. I jednoduché dotazy totiž při velkém počtu uživatelů způsobují výkonnostní problémy. Přirozeně nás tedy napadá, zda by nebylo možné u těchto zřídka měnících se dat tyto informace uchovat ve vyrovnávací paměti pro případné budoucí načítání. Přístup k této paměti bývá řádově rychlejší než přístup k datovému úložišti. Dojde tak ke snížení zatížení datového úložiště a k vyšší pružnosti naší aplikace. [3]

Ukládání do vyrovnávací paměti může probíhat na různých úrovních naší aplikace. Zmíním zde následující:

1. Vyrovnávací paměť na úrovni aplikace
   Při tomto přístupu je vhodné využít některý z existujících frameworků jako EHCache[17], JBoss Cache[18] či některé další. Ty fungují tak, že obalí naše DAO[19] objekty do tzv.

17 EHCache - http://ehcache.org/

18 JBoss Cache - http://www.jboss.org/jbosscache

19 Viz. 5.1.3 DAO

proxy vyrovnávací paměťi. Pokud je pak volána metoda DAO, dojde k přerušení, kdy se proxy vyrovnávací paměť snaží získat tato data z vyrovnávací paměti. Pokud se v ní nevyskytují, pokračuje v kladení dotazu na datové úložiště. Hlavní výhodou tohoto řešení je absolutní nezávislost na konkrétním ORM řešení.

2. Vyrovnávací paměť na úrovni ORM
   Většina ORM má dvě vrstvy. První vrstva vyrovnávací paměti, která bývá povinná a pro každou transakci oddělená. Druhá vrstva bývá volitelná a může být sdílena mezi transakcemi (tzn. jednotlivými vlákny) nebo i mezi procesy (různými servery v clusteru). [1]

Podpora transakcí

Při provádění operací nad datovým úložištěm potřebujeme zajistit, aby stále byla zajištěna konzistence a integrita těchto dat. Z těchto důvodů používáme tzv. transakce. Jedná se o posloupnost operací, která se provede jako celek nebo se neprovede vůbec. U transakcí vyžadujeme následující vlastnosti, které označujeme zkratkou ACID[20] [7]:

1. Atomičnost (Atomicity)
   Transakce soubor operací, který je z hlediska provádění brán jako celek. Buď proběhne celá (tj. proběhnou všechny její operace) nebo neproběhne vůbec.

2. Konzistence (Consistency)
   Pokud se databáze vyskytovala v konzistentním stavu před provedením transakce, musí se vyskytovat v konzistentním stavu i po jejím dokončení.

3. Izolace (Isolation)
   Transakce jsou od sebe izolovány. Je zajištěno, že efekt souběžně běžících transakcí bude stejný, jako by neprobíhaly souběžně, ale šly po sobě.

4. Trvalost (Durability)
   Po úspěšném dokončení transakce budou všechny změny, které transakce provedla, mít trvalý charakter.

20 Počáteční písmena požadované vlastnosti transakcí v anglickém jazyce: Atomicity, Consistency, Isolation, Durability

> *ORM nástroje navíc samy vyžadují, abychom transakce použili. Změny, které jsme na objektech prováděli se obvykle analyzují a převedou do konkrétních SQL příkazů až při ukončení transakce. Jedna akce v ORM frameworku (např. uložení jednoho objektu) se může skládat z několika SQL příkazů (INSERT do různých tabulek apod.) Není samozřejmě možné, aby se provedla jen část těchto operací. [16]*

Odvozené proměnné (Derived property)

Tyto proměnné jsou pouze pro čtení. Hodnota proměnné sama o sobě nijak nefiguruje v datovém úložišti. Místo toho je odvozena či spočítána od ostatních proměnných v době načítání objektu. Bývá vyjádřena jako SQL výraz, který je v době načítání převeden jako pod-dotaz vygenerovaného dotazu. [14]

Proxy objekty

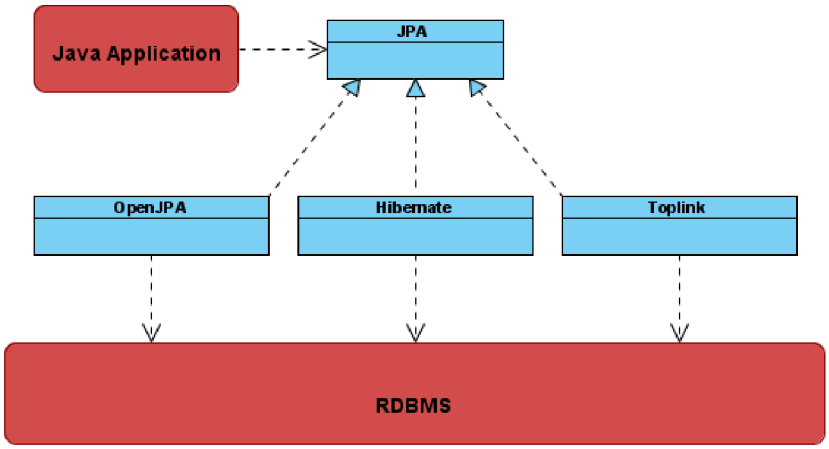
Proxy objekt je zástupce objektu, který nám v některých situacích může omezit zbytečný přístup k datovému úložišti. Jedná se o situaci, kdy potřebujeme vytvořit vztah mezi již existující entitou, u které známe její primární klíč, a jinou entitou. Protože pouze vytváříme vztah mezi těmito entitami, není potřeba, aby se celá entita načítala. Potřebujeme znát pouze její identifikátor pro vytvoření cizího klíče. Ukážeme si to na následujícím příkladu:

```
/* Příklad proveden v JPA řešení. Pod proměnnou PK_USER je nějaký
konkrétní primární klíč. */

/* Vytvoření proxy objektu bez přístupu do databáze. */
User user = entitiManager.getReference(User.class, PK_USER);
Project project = new Project();
project.setName("New Project");
project.setSupervisor(user);

/* Uložení objektu project a vztahu k objektu user. */
entityManager.persist(project);
```

Z příkladu je zřejmé, že pokud se při vytváření proxy objektu nepřistupuje k datovému úložišti, ORM nemůže ověřit existenci objektu s tímto primárním klíčem. Můžeme se pak tedy setkat s chybou při ukládání objektu, kdy se nepodařilo najít objekt s námi uvedeným primárním klíčem. [2]

4.1 JPA

Java Persistence API (JPA) je standard[21], který se snaží sjednotit persistenci objektů jazyka Java v relačních databázích. Nejedná se o konkrétní implementaci těchto postupů, ale o definování rozhraní a odstranění závislosti na konkrétní ORM implementací jako Hibernate[22], Toplink či OpenJPA. Aplikace pak může přecházet mezi různými implementaci JPA bez nutnosti měnit samotnou aplikaci.



Ilustrace 4.2: Architektura JPA

Konfigurace je možná jak pomocí externího XML souboru, tak pomocí anotací[23]. JPA používá pro kladení dotazů jazyk Java Persistence Query Language (JPQL). Jako alternativu lze od verze 2.0 i Criteria API[24].

První verze JPA 1.0 byla vydána v červnu 2006 jako součást specifikace EJB3[25]. Už o rok později v červnu byla zahájena práce na nové verzi JPA 2.0. Ta trvala do prosince 2009, kdy byla označena za dokončenou. V této verzi byla přidána již zmíněna podpora kritérií pro dynamické dotazy či vyrovnávací paměť druhé úrovně.

4.1.1 Criteria API

Jak už bylo zmíněno JPA 2.0 zavádí novinku v podobě Criteria API. Jedná se o další způsob dotazování nad databází, který na rozdíl od JPQL či SQL neklade dotazy pomocí skládání dotazů do

21 Tento standard je definován ve specifikaci EJB3 (JSR-220). Více informací naleznete na http://www.jcp.org

22 Viz. kapitola 4.2

23 Anotace jsou nový způsob zápisu metadat ve zdrojovém kódu umístěných přímo u programového elementu, který popisují. Tím může být například proměnná, metoda či celá třída. Anotace nemají přímý dopad na operace kódu, jež anotují. Možnost vytvářet vlastní anotace se v jazyce Java objevila od verze Java 5.0.

24 Viz. kapitola 4.1.1

25 Entity JavaBean 3

řetězců. Místo toho využívá kritéria objektů, u kterých se definují požadovaná omezení. JPA pak provede automatické převedení do JPQL dotazů.

Criteria API obsahují silnou typovou kontrolu, díky použití nového objektu `Root` už nejsou potřeba používat aliasy.

Podívejme se tedy na následující příklad[26], kde na prvním místě je vidět zápis pomocí JPQL a na druhém místě zápis pomocí nových Criteria API.

```
/* JPQL */
String jpql = "from User usr where usr.firstName like 'Jan'";
Query query = entityManager.createQuery(jpql);
List<User> result = (List<User>) query.getResultList();

/* Criteria API */
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery c = cb.createQuery(User.class);
Root<User> usr = c.from(User.class);
c.select(usr)
 .where(cb.equal(usr.get("firstName"), "Jan"));
```

Jak vidíme u jednoduchých dotazů je přehlednější použití JPQL. Pokud by dotaz ale obsahoval složitější podmínky, které by byly vkládány např. jen pokud je uživatel vyplnil ve formuláři, bylo by řešení pomocí JPQL mnohem méně přehledné.

4.1.2 Životní cyklus entity

V JPA se o životní cyklus entit stará EntityManager. Jednotlivé stavy a přechody mezi nimi můžeme vidět na obrázku 4.3. Entity se nachází vždy v jednom ze čtyř stavů [14]:

1. New (Transient)
   Jedná se o nově vytvořenou entitu, která nemá perzistentní reprezentaci v databázi a nebyla ji přidělena žádná databázová identita. JPA v tomto stavu o entitě neví. Pokud zanikne naše reference na tuto entitu, bude odstraněna garbage collectorem.

2. Managed (Persistent)
   Pokud se entita nachází v tomto stavu, je již svázaná se svojí reprezentací v databázi a vlastní svoji databázovou identitu. Jedná se o jediný stav, kdy se entita ukládá do databáze. Změny provedené v jiných stavech nejsou uloženy do doby, než se entita

26 Část příkladu převzata z [2].

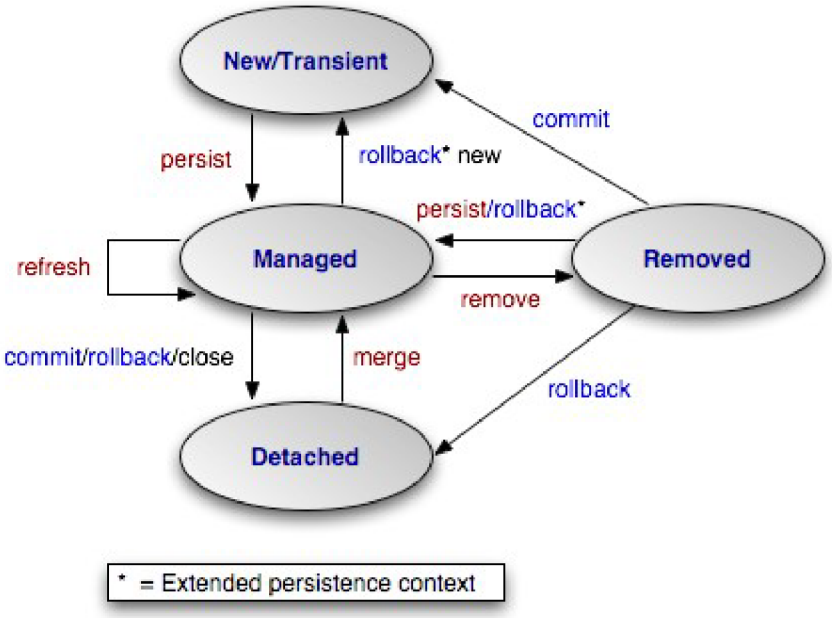
dostane do tohoto stavu. Pokud se entita v tomto stavu vyskytuje, veškeré změny na ní prováděné jsou automaticky ukládané do databáze.

3. Detached

   Po uzavření aktivního sezení, entity tohoto sezení, nacházející se ve stavu Managed, přejdou do stavu Detached. V tomto stavu má entita stále databázovou identitu, ale již není momentálně svázána se svojí reprezentací v databázi. Úpravy na entitě v tomto stavu nemají vliv na její databázovou reprezentaci. Entita, ale může přejít zpět do stavu Managed, kde budou tyto změny uloženy.

4. Removed

   Entita v tomto stavu má databázovou identitu a je spojena se svojí databázovou reprezentací, avšak je naplánována k odstranění z databáze. Změny na ní provedené se již do databáze nijak nepromítnou.



Ilustrace 4.3: Životní cyklus entity u JPA (Převzato z [17])

4.2 Hibernate

Hibernate je v dnešní době asi nejrozšířenější rámec pro objektově relační mapování. Je volně šířen pod licencí LGPL[27]. Hibernate je součástí projektu JBoss vyvíjený firmou Red Hat. Od verze 3.5 plně implementuje standard JPA 2.0.

Pro tvorbu dotazů používá vlastní rozšíření SQL jazyka nazývaného Hibernate Query Language (HQL). Hlavním rozdílem mezi SQL a HQL je, že HQL používá jméno třídy místo jména tabulky a název atributu místo názvů sloupce. HQL je nadmnožina jazyka JPQL. Jako alternativa ke kladení dotazů pomocí HQL slouží Criteria API. [1]

Metadata pro mapování tříd a jejich atributů na databázové schéma mohou být uvedena jako anotace nebo mohou být od zdrojového kódu oddělena do XML souborů.

4.2.1 Životní cyklus entity

U Hibernate se o stav entity stará PersistentManager. Na obrázku 4.4 můžeme vidět všechny stavy a přechody mezi nimi. Popis jednotlivých stavů si nyní vysvětlíme [14]:

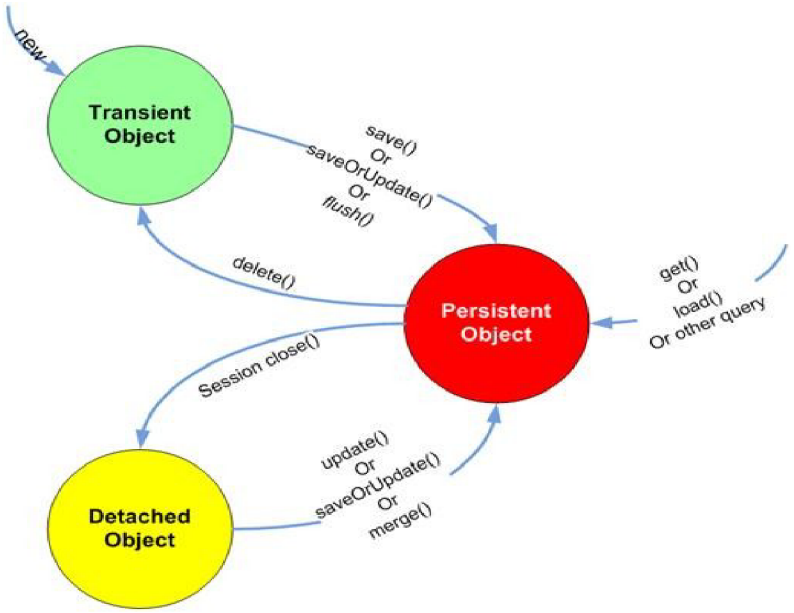
1. Transient
   Nově vytvořená entita, která není svázána s databázovou reprezentací ani nevlastní databázovou identitu. Tento stav odpovídá stavu New u životního cyklu JPA.

2. Persistent
   Entita je svázána s databázovou reprezentací a vlastní databázovou identitu. Tento stav odpovídá stavu Managed u životního cyklu JPA.

3. Detached
   Po ukončení sezení a odpojení od persistence manageru, dochází u jeho entit ve stavu Persistent přechod do stavu Detached. Tento stav odpovídá stavu Detached životního cyklu u JPA.

Na rozdíl od JPA tedy při mazání entita nepřechází do speciálního stavu Removed. Její reprezentace v databázi je smazána a entita se vrací se do stavu Transient.

27 GNU Lesser General Public License – viz. http://www.opensource.org/licenses/lgpl-license.php



Ilustrace 4.4: Životní cyklus entity u Hibernate (Převzato z [18])

4.2.2 Criteria API

Hibernate podporuje kromě dotazování pomocí HQL či SQL ještě další způsob, a tím je dotazování pomocí Criteria API. Ta se však značně liší od kritérií u JPA. A to i přestože kritéria u JPA byla vytvořena později než u Hibernate. Někteří vývojáři Hibernate se dokonce podíleli na specifikaci Criteria API u JPA.

U Criteria API klademe dotazy pomocí kritéria objektů, u kterých definujeme požadovaná omezení. V následujícím příkladě[28] vidíme ekvivalentní dotazy. První, zapsaný pomocí HQL a druhý pomocí Criteria API.

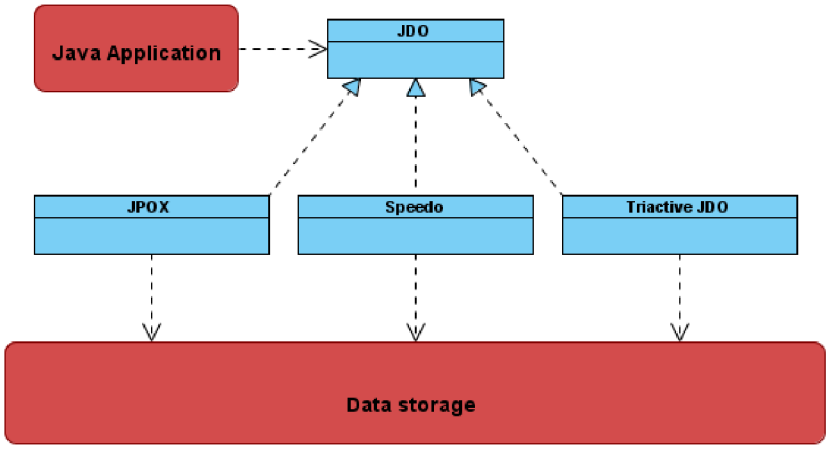
```
/* HQL */
String hql = "from User usr where usr.firstName like 'Jan'";
Query query = session.createQuery(hql);
List<User> result = (List<User>) query.list();

/* Criteria API */
Criteria criteria = session.createCriteria(User.class);
criteria.add( Restriction.like("firstName", "Jan") );
List<User> result = (List<User>) criteria.list();
```

28 Část příkladu převzata z [1].

Criteria API nám tedy umožňují definovat omezení pro dotaz bez nutnosti přímo zasahovat do HQL. HQL je pak generováno automaticky. Na druhou stranu je zápis v podobě HQL u jednoduchých dotazů přehlednější, než je tomu u Criteria API.

4.3 JDO

Java Data Objects (JDO) je dalším standardem[29] pro persistenci POJO vytvořeným firmou Sun Microsystems. Na rozdíl od JPA pevně nedefinuje typ úložiště. U různých JDO implementací se tedy můžeme setkat s úložišti typu relační databáze, objektová databáze, LDAP, XML či XLS. Stejně jako JPA se nejedná o konkrétní implementaci postupů, ale o definování rozhraní a odstranění závislosti na konkrétních řešeních jako DataNucleus[30], Speedo, Triactive JDO či Kodo.



Ilustrace 4.5: Architektura JDO

Konfigurece se provádí v externím xml souboru či pomocí anotací. Pro kladení dotazů používá jazyk JDO Query Language (JDOQL).

Verze JDO verze 1.0 byla vydána roku 2002. Od té doby již vznikla nová verze JDO 2.0 roku 2005 a v současnosti se pracuje na verzi 2.3.

4.3.1 Životní cyklus entity

Životní cyklus JDO je složitější než je tomu u JPA nebo Hibernate. Obsahuje velké množství stavů s mnoha přechody. Má tedy vysoký stupeň flexibility a může být nakonfigurován na velké množství různých módů[31]. Můžeme ale pohled na tyto stavy zjednodušit, že některé z nich sloučíme do skupin Transient, Persistent, Detached a Hollow. [12]

29 Jedná se o specifikaci JSR-243. Více informací naleznete na http://www.jcp.org.

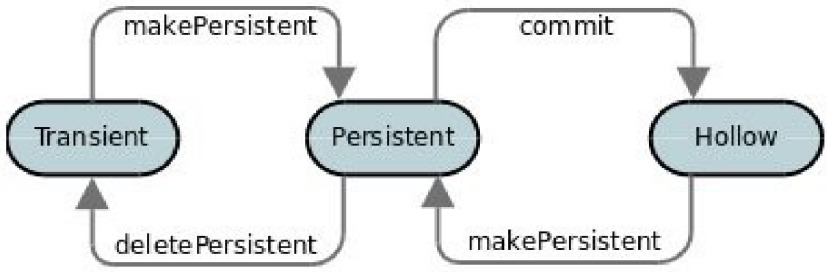
30 Projekt DataNucleus nyní pohltil a převzal vývoj nad rozšířenou otevřenou JDO implementací JPOX.

31 Podrobný popis životního cyklu u JDO naleznete v [19]

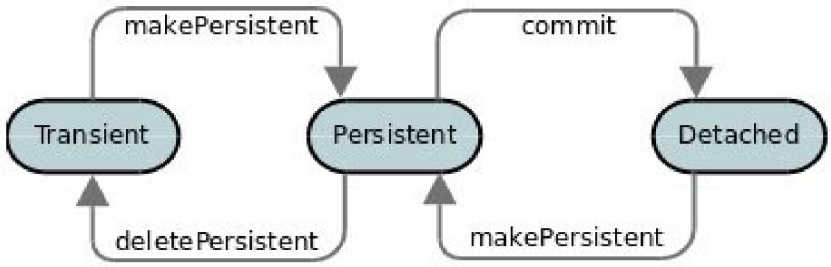
První tři stavy odpovídají stavům JPA. Stav Hollow je stav podobný stavu Detached s jediným výrazným rozdílem a tím je fakt, že u tohoto stavu po odpojení nemusí být všechny jednoduché vlastnosti entity načteny.

Pokud dochází k uzavření `PersistentManageru` nedochází automaticky, jak je tomu u JPA či Hibernate, k přechodu entit do stavu Detached ani Hollow. Entity tak přechází do stavu Transient.

U JDO máme možnost v nastavení zapnout DetachAllOnCommit. Pokud tato vlastnost není zapnuta, vypadal by životní cyklus entity tak, jak je zobrazeno na obrázku 4.6. Pokud však tuto vlastnost zapneme, získáme životní cyklus velice blízký JPA, jak můžeme vidět na obrázku 4.7.



Ilustrace 4.6: Zjednodušený životní cyklus JDO (Převzato z [11])



Ilustrace 4.7: Zjednodušený životní cyklus JDO při zapnutém nastavení DetachAllOnCommit (Převzato z [11])

4.4 iBatis

Než začnu se samotným popisem projektu iBatis, musím zde nejdříve vysvětlit, proč je zde vůbec tento projekt uveden. iBatis totiž nespadá do skupiny ORM, jedná se o rámec pro mapování dat v relační databázi. Svou jednoduchostí a nízkoúrovňovým přístupem a eliminací JDBC kódu si získal

řadu příznivců. Ve vybraných oblastech konkuruje tak komplexním řešením, jako rámcům postavených na JPA či JDO. Mezi jeho hlavní rysy patří:

- Odděluje SQL dotazy od kódu aplikace do samostatných XML souborů. Je zde i možnost využití zápisu do anotací[32].

- Eliminuje množství JDBC kódu.

- Nesnaží se abstrahovat a odstínit programátora od databáze. Nedochází zde k automatickému generování SQL. Psaní SQL dotazů je tedy ponecháno plně v rukou programátora.

Díky těmto vlastnostem se hodí k použití u již nasazených projektů. Velmi jednoduše totiž umožňuje namapovat složité existující databázové schéma na doménový model.

Princip iBatisu

Při použití iBatisu nám jde hlavně o oddělení SQL dotazů od zdrojového kódu, eliminaci JDBC kódu a o získání některých dalších výhod, které poskytuje plnohodnotné ORM. Mezi ty můžeme uvést podporu pro opožděné načítání či vyrovnávací pamět. Podívejme se tedy jak iBatis funguje.

Abychom s iBatisem mohli pracovat, potřebujeme nastavit konfigurační soubor, ve kterém jsou uvedeny například přihlašovací údaje k databázi. Dále pak musíme vytvořit tzv. mapové soubory, které obsahují popis třídy pomocí metadat. Mapový soubor by se dal rozdělit na dvě části. První část obsahuje mapy. Mapa nám mapuje sloupce výsledku SQL dotazu na vlastnosti třídy. Druhou částí mapového souboru je část s SQL dotazy. U každého z nich je uvedena mapa, na kterou se má výsledek dotazu namapovat. Ve zdrojovém textu pak tedy voláme pomocí iBatis objektů již vytvořené dotazy. Výsledky dotazu jsou iBatisem automaticky namapovány na příslušnou mapu, která odpovídá třídě. V konečném důsledku nám tedy iBatis vrátí požadovaný objekt. Ten má načteny ty vlastnosti, které byly uvedeny v použité mapě.

Na rozdíl od plnohodnotného ORM rámce iBatis nemá žádný vlastní dotazovací jazyk. Všechny dotazy jsou psány ručně programátorem v naitivním SQL.

Výhoda iBatisu oproti plnohotnotným ORM rámcům je v programátorově plné kontrole nad psaním dotazů. Jeho nevýhoda je naopak nutnost psaní všech i jednoduchých dotazů, a tedy nižší úspora programátorova času.

32 Zápis pomocí anotací je možný až od verze iBatis 3.

5 Ukázková aplikace

Pro účely porovnání ORM technologií jsem implementoval ukázkovou aplikaci, na které ilustruji způsoby použití z výše uvedených ORM technologií, jejich odlišnosti a problémy, se kterými jsem se v průběhu implementace setkal.

5.1 Použité technologie

Pro ukázkovou aplikaci jsem zvolil implementaci s webovým rozhraním. Zvolil jsem technologie, jež se v současnosti těší čím dál větší oblibě. Jde o spojení aplikačního serveru Apache Tomcat[33] + Spring MVC[34] + ORM.

Při porovnání budu v ukázkové aplikaci postupně měnit jednotlivé ORM implementace. Z tohoto důvodu jsem použil techniku DAO[35], která omezuje závislost aplikační logiky na konkrétní implementaci.

Priority při výběru technologií

Při výběru jednotlivých technologií jsem se zaměřil na technologie, které jsem se chtěl naučit a osvojit. Dával přednost těm, které jsou široce používané. Vysoká rozšířenost totiž zvyšuje možnost snáze nalézt potřebné informace (např. na fórech) při řešení různých problémů. Důležitým faktorem při výběru byla také jejich licence. Dával jsem přednost technologiím s otevřenou licencí, a to i pro použití v komerčních projektech.

5.1.1 UML

Unified Modeling Language (UML) je modelovacím jazykem hojně využívaném v odvětví softwarového inženýrství. Nabízí vizuální prostředky pro vytváření modelů zaměřených na různé aspekty navrhovaného systému. V dnešní době se považuje za obecně uznávaný standard, který byl vytvořen skupinou Object Managment Group (OMG).

5.1.2 Java/Java EE

Java je objektově orientovaný, multiplatformní jazyk od firmy Sun Microsystems, vydaný roku 1995. V současnosti jeden z nejrozšířenějších programovacích jazyků na světě. Své místo si našel jak

33 Viz. kapitola 5.1.10

34 Viz. kapitola 5.1.4

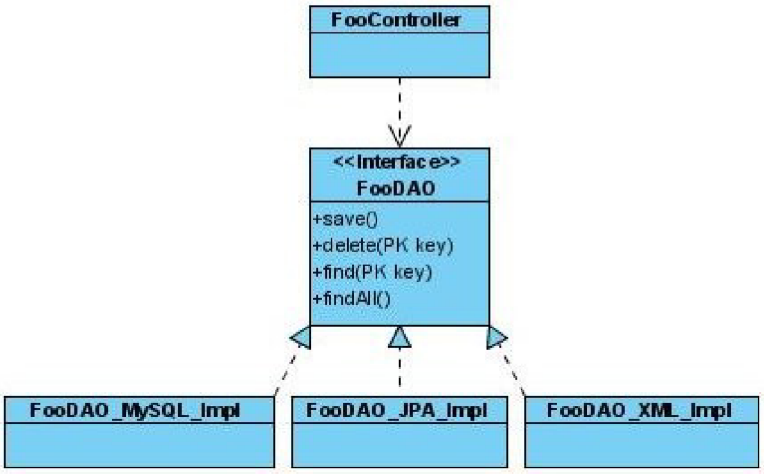
35 Viz. kapitola 5.1.3

v mobilních zařízeních (Java ME), stolních počítačích (Java SE), tak i na podnikových informačních systémech (Java EE).

„Java EE je průmyslovým standardem pro vývoj přenosných, robustních, škálovatelných a bezpečných serverových Java aplikací.“ [9] Jedná se o soubor standardů vytvářený konsorciem sezvaných firem. Své základy má založeny na platformě Java SE. Přidává k ní podporu pro tvorbu webových aplikací, distribuovaných vícevrstvých aplikací a webových služeb.

5.1.3 DAO

Data access object (DAO) je způsob, jak odstranit závislost z aplikační logiky od konkrétního způsobu perzistence dat. Je realizován pomocí speciálních rozhraní. Jak můžeme vidět na obrázku 5.1 třída `FooController`, která by byla závislá na konkrétní implementaci, je nyní závislá pouze na



Ilustrace 5.1: Příklad DAO (Diagram tříd)

rozhraní `FooDAO`. Pokud se tedy bude měnit perzistence z MySQL databáze na perzistenci v XML souboru, bude potřeba pouze realizovat metody rozhraní na nové třídě `FooDAO_XML_Impl`. Zbytek aplikační logiky zůstane nezměněn.

5.1.4 Spring/Spring MVC

Spring je svobodný aplikační framework pro Java Enterprise Edition (Java EE) platformu, jenž byl vyvinut roku 2002 australským programátorem Rodem Johnsonem. Později byl rozšířen o webový framework Spring MVC framework, vycházející z původního Spring framework. Snaží se nepřinášet

závislost aplikačního kódu na frameworku. Využívá technik jako jsou návrhový vzor dependency injection[36] (DI), či aspektově orientované programování[37] (AOP).

5.1.5 Spring Security

Framework Spring Security byl založený roku 2003. Snaží se o snadné zabezpečení aplikace, oddělení kódu pro zabezpečení od aplikační logiky pomocí techniky AOP. Podporuje zabezpečení na různých úrovních – na úrovni pohledu, objektů, či jednotlivých metod.

5.1.6 JFreeChart

JFreeChart je otevřená knihovna sloužící k vytváření a práci s nepřeberným množstvím různých druhů diagramů v jazyce Java. Může být použita jak v desktopových aplikacích, appletech, tak i v JSP.

Knihovna byla použita v ukázkové aplikaci pro vykreslení koláčového grafu ve výstupu pro management.

5.1.7 JSP/JSTL

JavaServer Pages (JSP) je technologie vyvinuta společností Sun Microsystems za účelem tvorby dynamických webových stránek. Zdrojový kód JSP obsahuje kód statických stránek, obohacený o dynamický kód jazyku Java.

JSP stránky jsou zpracovány na straně serveru. Pokud přijde požadavek na zobrazení stránky, je stránka, pokud ještě nebyla kompilována, nejprve převedena na Servlet[38], následně kompilována. Vytvoří se příslušná instance Servletu a jeho výstup je vrácen klientovi.

JavaServer Pages Standard Tag Library (JSTL) je řada předpřipravených knihoven tagů, které zapouzdřují základní funkcionalitu používanou u většiny JSP stránek jako jsou internacionalizace, podmínky, cykly či tagy pro práci s databází.

---

36 Vkládání závislostí, jedná se o novější název návrhového vzoru Inversion of Controll (IoC). Ten se snaží o zvolnění vazeb mezi jinak těsně svázanými komponentami. V podstatě se jedná o způsob, kdy třídy nevytváří samy další instance tříd, které potřebují. Tyto instance jsou jim předány nějakým způsobem z vnějšku, a to za pomocí konstruktoru či setteru. Díky nižší závislosti dosáhneme jednodušší údržby jednotlivých komponent, tak i výrazně lepší možnosti jak komponenty samostatně testovat.

37 AOP je nadstavba nad objektově orientovaným programováním (OOP). Není to náhrada OOP, ale jeho komponenta. Pomocí AOP můžeme odstranit opakující se kód z tříd, který přímo nesouvisí s funkcionalitou třídy. Jedná se například o kontrolu přihlášení.

38 Servlet je program v jazyce Java sloužící k generování dynamického obsahu webových stránek, běžící na straně serveru.

5.1.8 JSR-303: Bean Validation

Jedná se o standard validace java bean pomocí anotací a tím i oddělení kódu dříve potřebného pro validaci od aplikační logiky. Pomocí anotací omezení jako `@NotNull` či `@Email` můžeme vyjádřit omezení na našem doménovém modelu. Později pak můžeme z různých vrstev aplikace volat automatickou validaci, jako například při předání parametru metodě pomocí `@Valid` anotace `fooMethod(@Valid FooObject foo)`.

5.1.9 jQuery/jQuery UI

jQuery je malý svobodný javascriptový framework, který byl vydán roku 2006 americkým programátorem Johnem Resigem. Snahou tohoto frameworku je oddělit akce nebo chování od struktury dokumentu tak, jako je oddělen vzhled díky kaskádovým stylům.

Knihovna jQuery UI je nadstavba nad frameworkem jQuery sloužící k tvorbě interaktivního grafického rozhraní v RIA[39] aplikacích.

5.1.10 Apache Tomcat

Tomcat je jednoduchý aplikační kontejner a webový server s implementací Servlet/JSP technologií. Tento svobodný software byl vydán roku 1999 americkým softwarovým architektem Jamesem Duncanem Davidsonem z nadace Apache Software Foundation.

5.1.11 MySQL

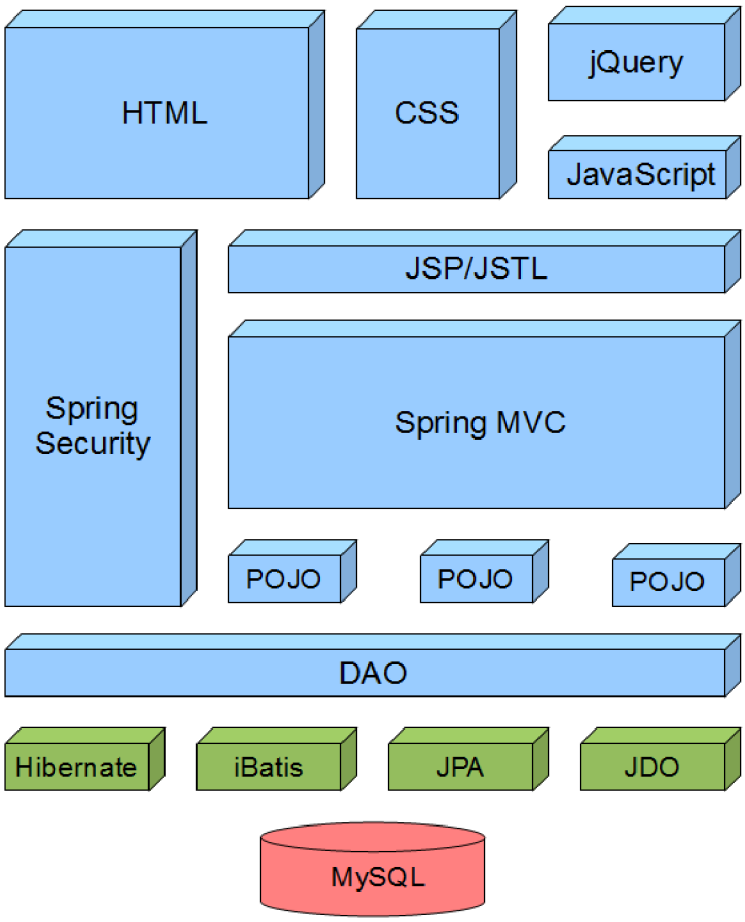
Relační multiplatformní databázový systém vyvíjený firmou Oracle. Je vydáván pod dvojí licencí, bezplatnou licenci GPL[40] bez podpory, tak i pod komerční placenou licencí s podporou.

V době svého vzniku bylo MySQL silně zaměřeno na rychlost a jednoduchost, chyběly zde některé důležité vlastnosti jako vnořené pod-dotazy, transakce, pohledy, uložené procedury. Všechny tyto nedostatky však již jsou napraveny a nyní podporovány.

---

39 Rich Internet Applications – webové aplikace, snažící se svým ovládáním a chováním napodobit desktopové aplikace.

40 General Public License – licence pro svobodný software



Ilustrace 5.2: Přehled, jak do sebe jednotlivé technologie zapadají

5.2 Specifikace a analýza požadavků

Jedná se o informační systém pro firmu zabývající se vývojem software agilními technikami. Tento informační systém by jí umožnil snáze řídit vývoj projektů, rozdělovat lidské zdroje a vytvářet výstupy pro management. Programátoři uvidí, které projekty řídí, a v nich budou moci zakládat sprinty a komponenty projektu. Každý programátor pak může vytvářet úkoly pro jakýkoliv projekt a může ho i komukoliv přiřadit. Úkol musí být zařazen do sprintu a komponenty a může být ohodnocen i odhadem času, který je potřeba k jeho splnění. Programátoři uvidí přednostně seznam všech otevřených úkolů, které mají přiřazené. Pod nimi uvidí seznam všech otevřených úkolů přidělených i ostatním programátorům. K úkolu pak programátoři zadávají, kolik hodin na něm odpracovali.

Hlavním cílem systému je tedy podrobná evidence práce programátorů, ze které lze vyčíst, jakou dobu pracoval programátor, na kterém projektu, úkolu nebo seznamu úkolů, které přesáhly odhadovaný čas na zpracování. Kolik odpracovaného času kam teče nebo kde se neplní plán.

Systém musí být navrhnut s ohledem na možné budoucí rozšíření o modul pro centralizovanou hotline podporu pro všechny zákazníky. Ti v budoucím modulu budou mít přístup do systému, ke svému projektu. Tam budou moci hlásit nalezené chyby, jež se budou zobrazovat programátorům, kteří mají daný projekt na starosti.

Programátoři pracují na různých platformách, proto je jedním z hlavních požadavků nezávislost na platformě. Nejlépe pak řešení webové aplikace a přístupu pomocí webového prohlížeče a podpora minimálně prohlížečů od následujících verzí – Internet Explorer 7, Mozilla Firefox 3.5, Opera 10.

5.2.1 Uživatelé systému

Uživatelé toho systému se v této podobě dají rozdělit do tří rolí. Role programátora, jenž bude vidět projekty, které řídí. Tyto projekty pak bude moci spravovat a vytvářet v nich komponenty a sprinty. Pokud projekt neřídí, může komponenty a sprinty pouze prohlížet. Dále pak může spravovat úkoly a zadávat odpracované hodiny.

Druhou rolí je role administrátora. Ten spravuje uživatelské účty a zakládá projekty a přiřazuje k nim řídící programátory

Třetí rolí je role manažera, který má přístup k výstupům o toku odpracovaného času, či o nedodržování plánu u jednotlivých projektů. Dále pak může prohlížet detailní informace o projektu, sprintu, komponentě či úkolu.

Každý uživatel může mít přiřazenu jednu či více rolí.

5.2.2 Diagram případů použití

Diagram případů použití můžeme vidět na obrázku 5.3, z důvodu větší přehlednosti jsou některé případy použití jako CRUD[41] operace spojeny do jednoho případu „Spravuj operace“.

5.2.3 Specifikace případů použití

Z důvodu zbytečného opakování v jednotlivých případech použití jsou některé části vyjmuty a shrnuty v následujících odstavcích.

Některé případy použití obsahují formulář se seznam údajů, které musí být pro dokončení případu povinně vyplněny. Pokud některé z povinných polí nebude vyplněno, bude uživateli zobrazena informace o nutnosti toto pole vyplnit. Dokud nebudou všechna pole vyplněna, nebude uživateli povoleno formulář potvrdit. Uživatel má však možnost formulář kdykoliv opustit.

41 Create Read Update Delete (CRUD) – jedná se o zkratku pro základní operace jako je Vytvoř, Čti, Uprav či Smaž

Všechny seznamy vyskytující se v jednotlivých případech užití musí být jednoduše filtrovatelné. Dále pak zde musí existovat možnost, jak seznam seřadit podle jednoho či více sloupců seznamu.

Každý seznam obsahující informace o projektu, sprintu, komponentě či úkolu musí umožňovat rychlý přechod na jejich detailní popis. Příklad: Uživatel si prohlíží seznam úkolů, ve kterém se mimo jiné vyskytuje informace o projektu, do kterého spadá. Může se jednoduchým krokem dostat k detailním informacím o daném projektu, aniž by musel tento projekt vyhledávat v seznamu projektů.

Autorizace a autentizace

Všichni uživatelé se přihlašují přes společnou obrazovku. Uživatel musí zadat svoje uživatelské jméno a heslo. Systém na základě těchto údajů rozpozná, zda-li má uživatel do systému přístup a jaké mu náleží role.

Uživatel má možnost se na svém účtu na počítači přihlásit k systému trvale. V tomto případě, přihlásí-li se uživatel ze stejného účtu a prohlížeče, systém se pokusí sám přihlásit s údaji, které naposledy uživatel zadal.

Vytvoř/Uprav uživatele

Přístup k agendě pro správu uživatelů má pouze uživatel s rolí administrátora. Administrátor zde vidí seznam všech aktivních i neaktivních uživatelů. Přímo ze seznamů může aktivovat, popř. deaktivovat uživatele. Při vytváření nového, či úpravě stávajícího uživatele, je nutné zadat uživatelovo jméno, příjmení, uživatelské jméno, heslo a role potřebné pro přístup k jednotlivým agendám.

Změna přiřazených rolí se musí projevit na přístupu k jednotlivým agendám už od příštího pokusu o přihlášení daného uživatele.

Deaktivuj/Reaktivuj Uživatele

Uživatel s rolí administrátora může přímo ze seznamu aktivních/neaktivních uživatelů deaktivovat/aktivovat uživatele.

Změna se musí projevit od příštího pokusu o přihlášení daného uživatele. Neaktivním uživatelům musí být odepřen přístup do systému.

Vytvoř/Uprav projekt

Uživatel s rolí administrátora vidí seznam všech existujících projektů. Při vytváření a úpravě projektu musí zadat jméno projektu a vedoucího projektu. Volitelně pak může zadat i popis projektu.

Prohlížej projekt

K této funkci má přístup uživatel s rolí programátor nebo manažer. Uživatel si prohlíží detailní informace o projektu. Zobrazí se mu seznam komponent a seznam sprintů. U každého sprintu bude zobrazena doba, kdy začíná a kdy končí. Dále pak následující informace:

- *Odhadovaný čas* – tento čas je odvozen ze sumy odhadovaných časů pro jednotlivé úkoly spadající do tohoto sprintu.
- *Strávený čas* – tento čas je odvozen ze sumy času stráveného na jednotlivých úkolech spadajících do tohoto sprintu.
- *Postup v procentech* – tato hodnota získána jako procentní podíl *Stráveného času* z hodnoty *Odhadovaného času.*
- *Grafické zobrazení postupu* – grafické znázornění *Postupu v procentech* se zvýrazněním, pokud *Strávený čas* přesáhne *Odhadovaný čas.*

Prohlížej sprint

Uživatel s rolí programátora nebo manažera si prohlíží informace o sprintu. Vidí zde přehled úkolů spadajících do tohoto sprintu. U každého úkolu je uveden, do jakého projektu a komponenty patří, odhadovaný čas, čas strávený na úkolu, o jaký typ úkolu se jedná, komu byl úkol přiřazen a kdy byl vytvořen. Dále pak grafické znázornění postupu v procentech.

Spravuj sprinty

Uživatel s rolí programátora může upravovat a přidávat sprinty u projektů, kde je uveden jako vedoucí projektu. Při vytváření či úpravě sprintu musí být uvedeno jméno sprintu, dále pak může být uvedeno datum, kdy sprint začíná a končí[42].

Prohlížej komponentu

Programátor nebo manažer prohlíží detailní informace o komponentě. Zároveň vidí i seznam úkolů spadajících do této komponenty. U každého úkolu budou zobrazeny následující informace: projekt a sprint, do kterých úkol patří, komu byl úkol přiřazen, typ úkolu, kdy byl úkol vytvořen, odhadovaný a strávený čas. Jako další informaci bude obsahovat grafické znázornění postupu v procentech.

Spravuj komponenty

Případ spravuj komponenty bude umožněn vykonávat pouze uživatelům s rolí programátor. To však platí pouze u projektů, u kterých se vyskytuje tento programátor jako vedoucí projektu. Při vytváření či úpravě komponenty musí uživatel zadat jméno komponenty, volitelně pak její popis.

42 Datum začátku a konce sprintu slouží pouze pro informaci uživatele, v systému nemá vliv na žádnou další funkcionalitu.

Zobraz přehled všech úkolů

Programátor si prohlíží přehled všech úkolů. Budou zde zobrazeny tři seznamy:

1. Seznam otevřených úkolů přiřazených přihlášenému programátorovi
2. Seznam všech otevřených úkolů
3. Seznam všech uzavřených úkolů

V každém seznamu budou u jednotlivých úkolů uvedeny informace jako: jméno úkolu, projekt, sprint a komponenta, do které úkol paří. Dále pak programátor kterému byl úkol přiřazen, odhadovaný čas, strávený čas a grafické znázornění postupu v procentech.

Prohlížej úkol

Uživatel s rolí manažera nebo programátora si prohlíží detailní informace o úkolu spolu se seznamem odpracovaných jednotek na tomto úkolu. U každé odpracované jednotky je uvedeno datum, ze kterého dne pochází, poznámka, kdo na jednotce pracoval a kolik hodin.

Spravuj úkol

Spravovat úkol může kterýkoliv z programátorů. Pro vytvoření úkolu musí uživatel nejdříve vybrat, do kterého projektu bude úkol zařazen. Dále pak musí vyplnit jméno úkolu, odhad, jak dlouho má splnění tohoto úkolu trvat, o jaký typ se jedná, do jaké komponenty a sprintu patří. Může vyplnit popis úkolu a rovnou ho i přiřadit některému programátorovi.

Uzavři/Otevři úkol

Kterýkoliv uživatel s rolí programátora může otevřít či uzavřít úkol. Po uzavření projektu se tento úkol přesune z prvního a druhého seznamu úkolů zobrazeného v případu „Zobraz přehled všech úkolu“ do seznamu všech uzavřených úkolů.

Zadej odpracované hodiny

Uživatel s rolí programátora může zadat hodiny, odpracované na vybraném projektu. Mezi povinné údaje patří datum, kdy pracoval, strávený čas, a o kterého uživatele se jedná. Další možností je zadat k tomuto záznamu popis.

Zobraz osobní stránku

Každý aktivní uživatel má možnost se podívat na svoji osobní stránku, na které vidí informace o svojí osobě a svém účtu. Je zde uvedeno jeho jméno, příjmení, uživatelské jméno a přiřazené role. V této agendě také může změnit svoje přístupové heslo. Pro změnu ostatních údajů musí kontaktovat administrátora, který je může změnit v agendě pro správu uživatelů.

Změň heslo

Tento případ je přístupný pro všechny role. Uživatel je vyzván k zadání nového hesla. To musí být neprázdné a dlouhé minimálně 8 znaků.

Vygeneruj graf „Na jakých projektech a jak dlouho programátor pracoval“

Uživatel s rolí manažer má možnost si nechat vygenerovat sestavu s koláčovým grafem. V něm vidí kolik hodin pracoval vybraný programátor na jednotlivých projektech. Pro vygenerování sestavy musí tedy uživatel vybrat programátora. Dále pak má možnost omezit časový interval, ze kterého chce sestavu provádět.

Vygeneruj sestavu „Kolik hodin bylo stráveno na jednotlivých projektech“

Uživatel s rolí manažera má možnost nechat vygenerovat sestavu, ve které vidí seznam projektů a počet hodin, kolik na nich bylo stráveno[43]. Z tohoto seznamu se může jednoduše dostat k detailnímu pohledu[44]. Uživatel má možnost omezit časový interval, ze kterého chce sestavu generovat.

Vygeneruj sestavu „Které projekty překročily plán“

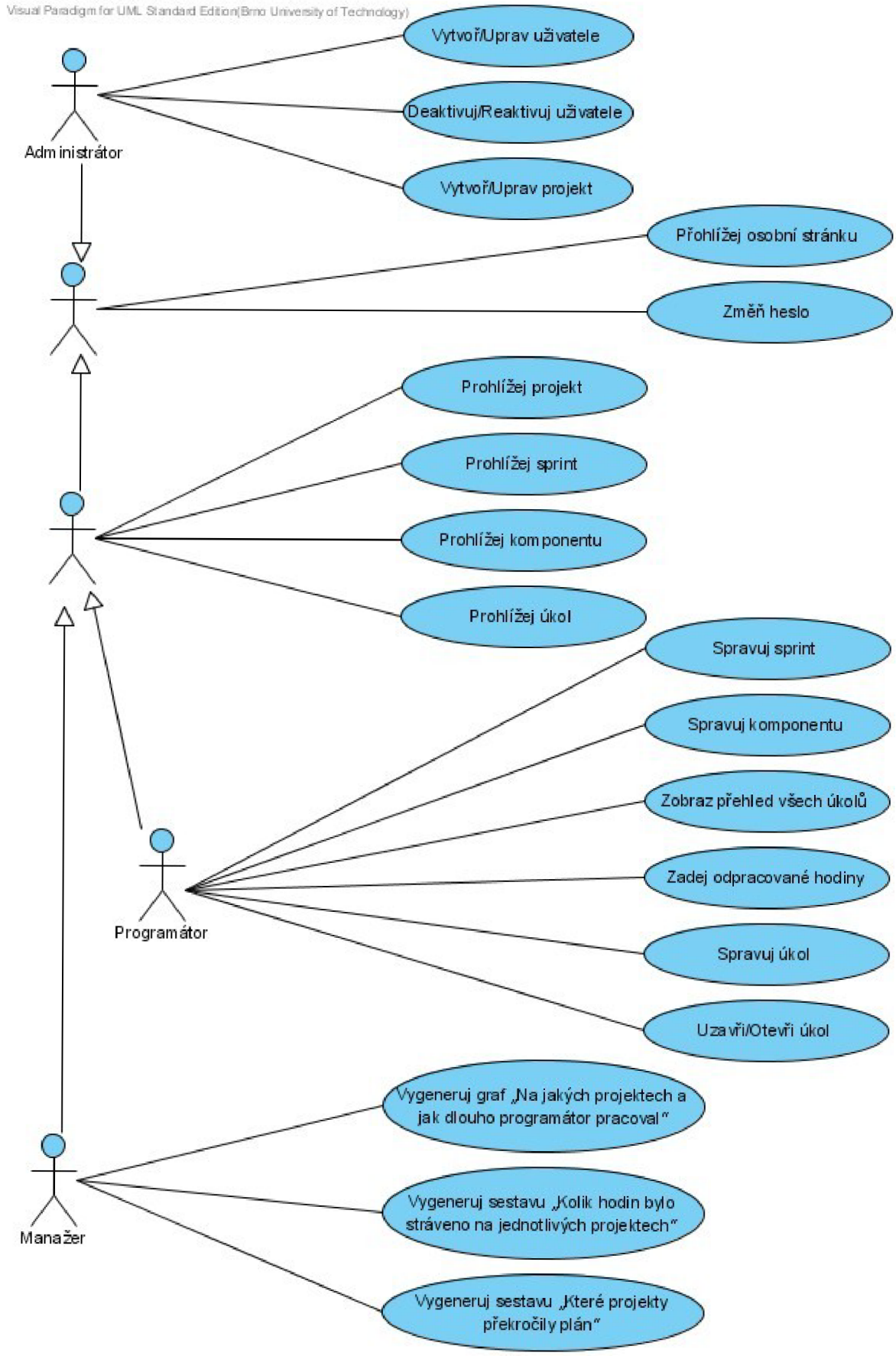
V této sestavě vidí uživatel s rolí manažera seznam projektů, které překročily plán. To jsou ty projekty, u kterých je odhadovaná doba[45] na jejich dokončení[46] kratší než doba na nich strávená.

43 Počet strávených hodin na projektu se spočítá jako suma času stráveného na sprintech spadající do tohoto projektu.

44 Viz. případ použití Prohlíží projekt

45 Viz. poznámka pod čarou 43 (str. 43)

46 Odhadovaná doba projektu se spočítá jako suma času odhadovaných pro jednotlivé sprinty spadající do tohoto projektu.



Ilustrace 5.3: Diagram případů použití ukázkové aplikace

5.3 Návrh systému

Po dokončení fáze analýzy požadavků a vytvoření specifikace jsem začal pracovat na návrhu systému. Sestavil jsem konceptuální diagram tříd, dále jsem vybral architekturu. Vytvořil jsem diagram návrhových tříd a přešel k diagramu toku obrazovek a jejich jednotlivým ručním náčrtkům. Jako poslední krok před zahájením implementace jsem vytvořil databázové schéma.

5.3.1 Konceptuální diagram tříd

Konceptuální diagram tříd můžeme vidět na obrázku 5.5.

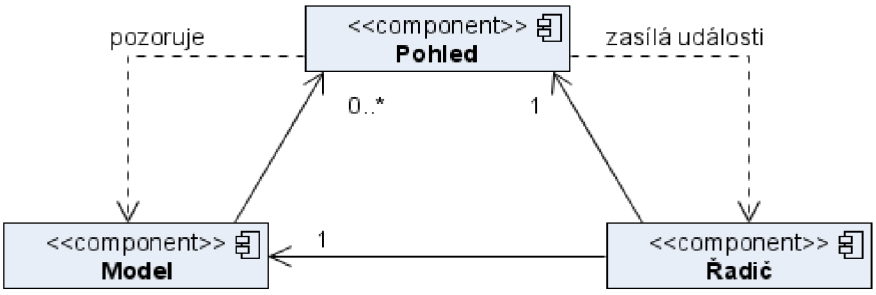
5.3.2 Diagram architektury

Jako základ architektury jsem vybral Model-View-Controller (MVC). Ta rozděluje aplikaci do tří oddělených komponent [20]:

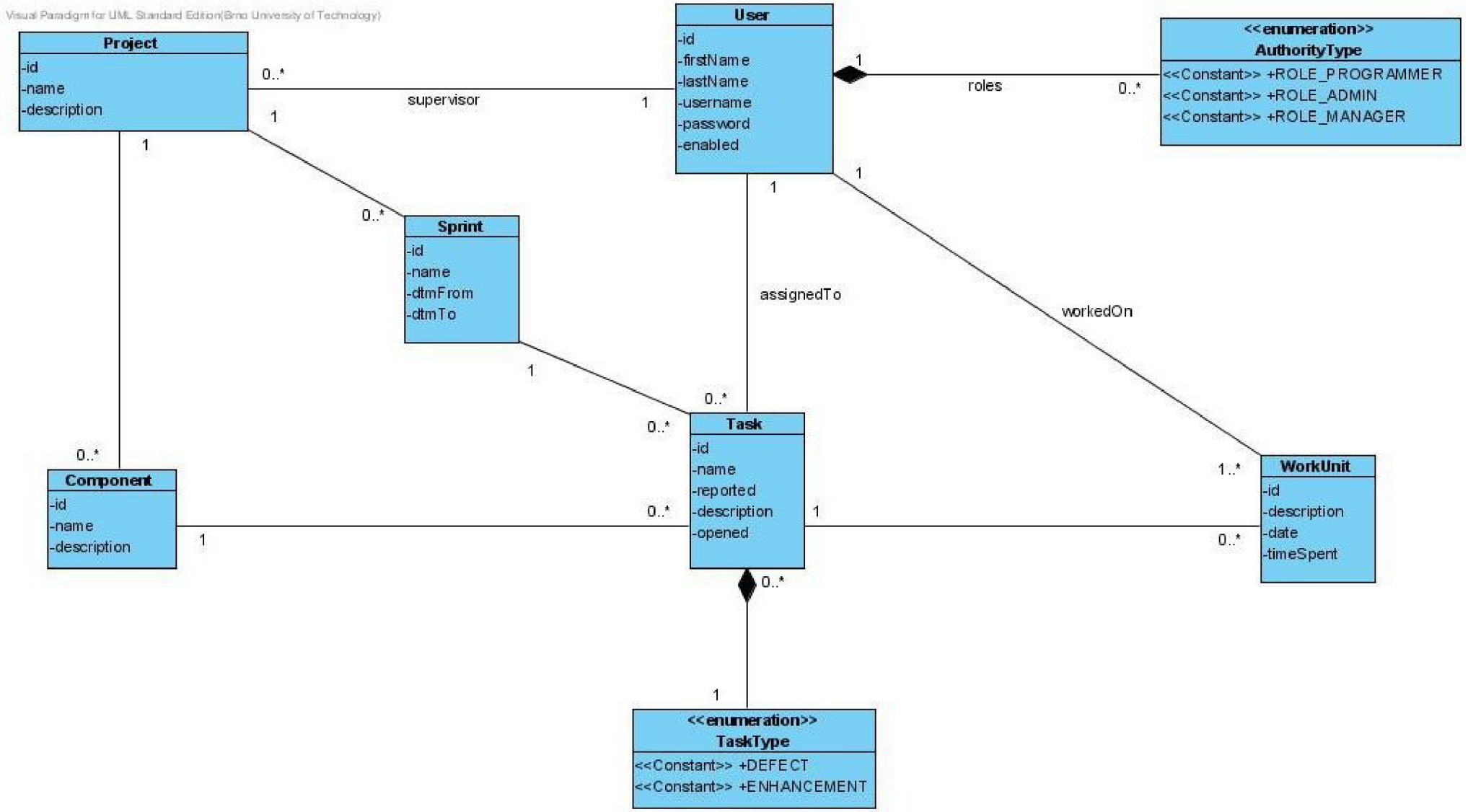
1. Model (Model) – Odpovídá vrstvě domény.

2. View (Pohled) – Má na starosti reprezentaci modelu.

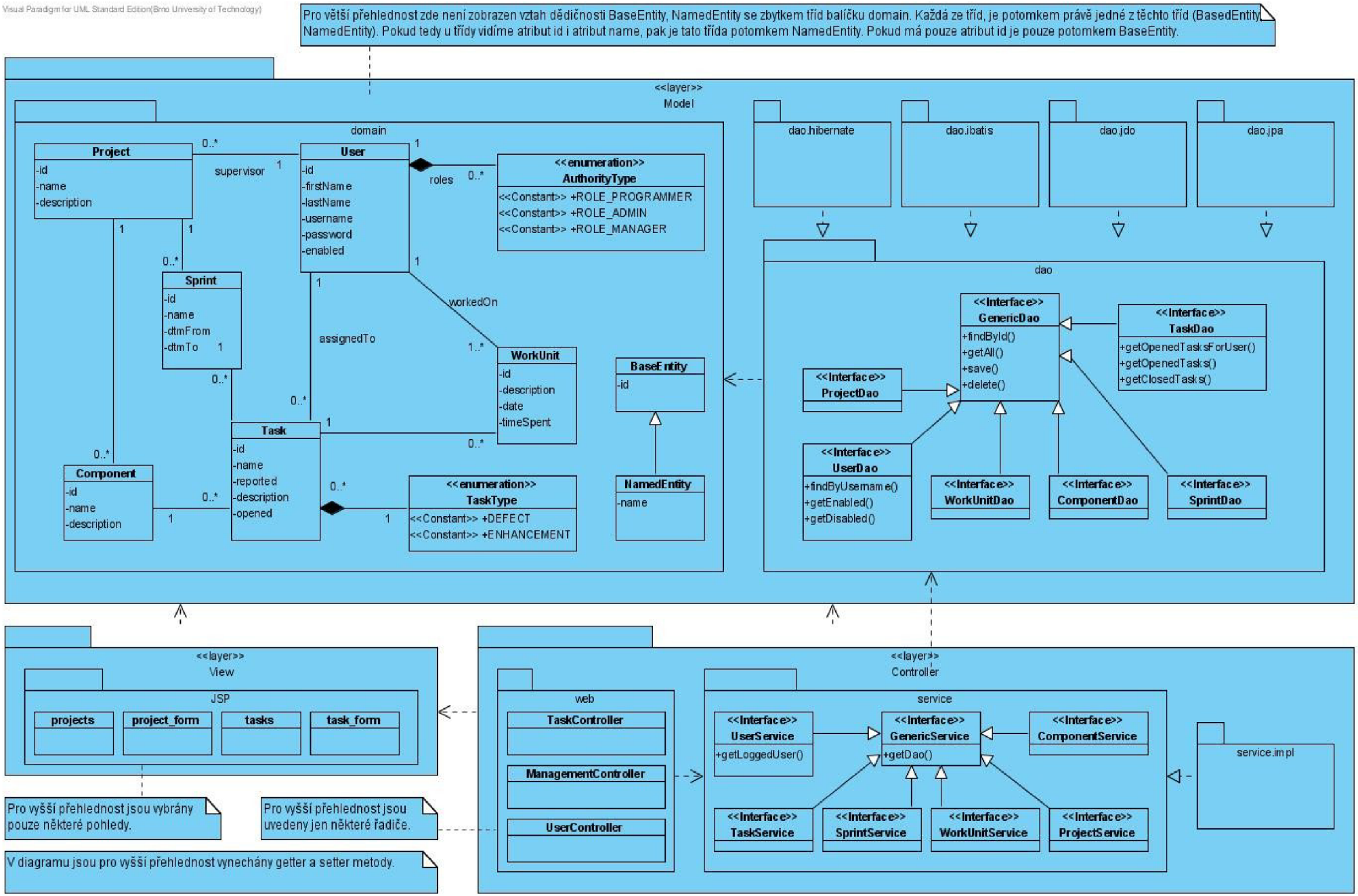
3. Controller (Řadič) – Jeho úlohou je reagovat na události přijaté z pohledu a zajišťovat změny v pohledu a modelu.



Podrobnější diagram architektury ukázkové aplikace ukazující rozdělení mnou implementovaných návrhových tříd do balíčků můžeme vidět na obrázku 5.6.



Ilustrace 5.5: Konceptuální diagram tříd ukázkové aplikace



Ilustrace 5.6: Podrobnější diagram architektury ukázkové aplikace

5.3.3 Uživatelské rozhraní

Systém je navržen jako webová aplikace s přístupem přes tenkého klienta. Jedná se tedy o návrh webového uživatelského rozhraní. Návrh probíhal náčrtem diagramu toku obrazovek a dále pak rozkreslením jednotlivých obrazovek.

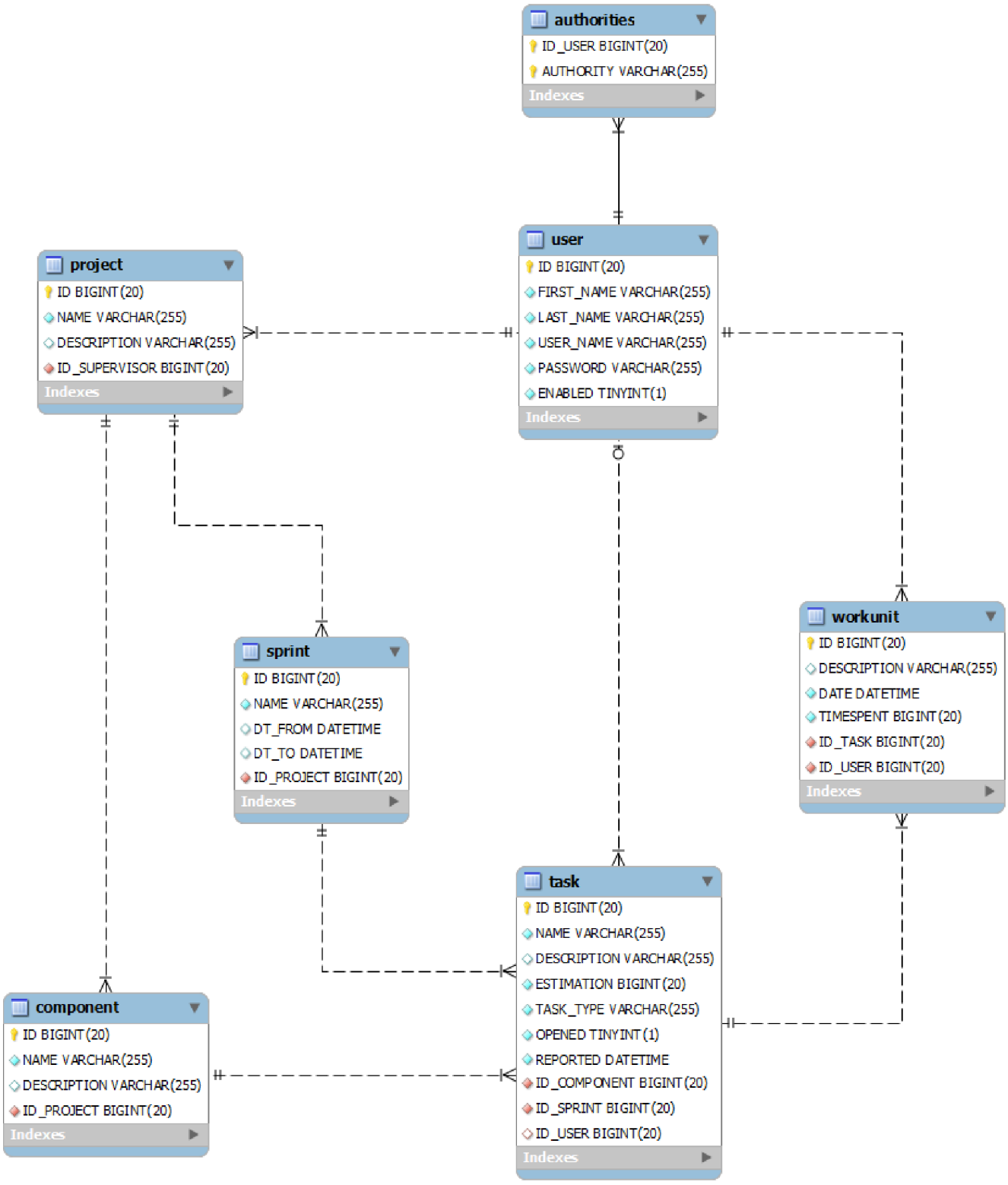
Návrh a realizace kvalitního uživatelského rozhraní je při vývoji software klíčová. Jedná se totiž o jedinou část systému, kterou uživatel a zákazník vidí. I když vytvoříme sebelepší systém s pokulhávajícím uživatelským rozhraním, nesetkáme se u uživatelů s kladnými odezvami. Kvalitním uživatelským rozhraním, nemyslíme jen vzhled či atraktivnost, které jsou ovšem také důležité. Mnohem důležitější je však například jednoduchost a srozumitelnost.

Při návrhu a vytváření uživatelského rozhraní jsem se tedy tyto zásady snažil dodržovat. Vytvořil jsem přehlednou navigaci, ze které se uživatelé mohou dostat na nejčastěji prováděné operace. Informace zobrazené na stránkách jsou provázané. Na detailní stránku komponenty či sprintu se lze dostat ze všech tabulek, kde je o nich zobrazena informace. Tedy například i ze seznamu úkolů.

Některé z obrazovek systému můžeme vidět v příloze této práce.

5.3.4 Databázové schéma

Schéma databáze můžeme vidět na obrázku 5.7. Bylo vygenerováno pomocí Hibernate. Pak jsem ho zkoumal, zda na něm není něco třeba změnit. Další ORM řešení jsem pak již mapoval do toho existujícího schématu.



Ilustrace 5.7: Databázové schéma ukázkové aplikace

5.4 Implementace systému

Implementaci jsem započal po zvolení konkrétních technologií a dokončení návrhu. Nejdříve však bylo nutné propojit vybrané technologie a vytvořit stabilní platformu, na které systém bude postaven.

U jednotlivých technologií jsem se snažil vždy použít co nejnovější verze. To se mi ovšem nepodařilo u JPA verze 2.0 (použita verze 1.0) a u mapovacího rámce iBatis 3.0 (použita verze 2.3.4.726). Tento problém byl způsoben nekompatibilitou s použitým rámcem Spring. Tyto verze budou podporovány až v příští verzi rámce Spring.

Při implementaci jsem používal nástroje Netbeans IDE od firmy Sun. Jako SQL konzoli jsem využil volného projektu HeidiSQL. Pro ladění jsem využil zejména prohlížeče Mozilla Firefox s přídavnými rozšířeními pro validaci a ladění webových stránek, jako je FireBug a HTML Validator.

5.4.1 Míchání ORM metadat

Při implementaci jednotlivých standardů JPA nebo JDO je možné míchat metadata s jejich konkrétními implementacemi. Například tedy můžeme použít část popisu metadat z Hibernate a zbytek z JPA. Přicházíme tak však o jednu z hlavních výhod používání těchto standardů, a tou je nezávislost na konkrétní implementaci. Proto jsem se tomuto řešení snažil vyhýbat. To se mi však ve dvou případech nepodařilo. O těchto případech se zmiňuji v následující kapitole 5.4.2.

5.4.2 Problémy s implementací u jednotlivých ORM řešení

Při implementaci perzistence u jednotlivých ORM řešení jsem se setkal s některými problémy, které bych zde rád popsal.

Hibernate

Asi největším problémem u tohoto řešení bylo uložení kolekce výčtových typů reprezentovanou řetězci[47]. Po delším hledání se mi to však podařilo. V metadatech musel být definován nový typ reprezentující výčtový typ, který má jako parametr magickou konstantu 12. O té se mi podařilo zjistit pouze to, že má reprezentovat nějakou konstantu z balíku `java.sql`. Dokumentace ani návody se o ní nikde nezmiňují.

JPA

U JPA se mi nepodařilo metadaty popsat kolekci výčtových typů. Musel jsem tedy použít míchání metadat s metadaty konkrétní implementace Hibernate. Dalším problémem byla chybějící podpora pro odvozené proměnné. To jsem musel obejít dalšími dotazy na databázi v DAO objektech. To

47 Popsáno v kapitole 3.5.2 Mapování výčtových typů

problém vyřešilo v případech, kdy jsem k těmto proměnným přistupoval po získání objektů přes DAO. Aplikace však používá opožděné volání, které k načítání nepoužívá přístup přes DAO. Proto bylo potřeba tento přístup ošetřit na úrovni kontoleru.

JDO

U JDO byl také velký problém metadaty popsat kolekci výčtových typů. Nakonec se to však podařilo. Výčtový typ bylo potřeba definovat v metadatech jako třídu a zacházet s ní jako s komponentou, tedy závislou třídou bez vlastní identity. Mezi další problémy patřila chybějící podpora pro odvozené proměnné. Řešení tohoto problému bylo stejné jako v případě JPA.

Kvůli možnosti použít opožděné načítání musely být entity ve stavu Persistent. V pohledu pak mohly být opožděně načteny další potřebné informace. Ovšem při dokončení vytváření pohledu a ukončení `PersistentManageru` pak entity přešly do stavu Transient[48]. To působilo problémy při editaci již existující entity a následném pokusu o uložení. Pro JDO se totiž podle stavu jednalo o nový objekt a místo úpravy stávajícího objektu vytvářel nový. Řešením by mohlo být v DAO ruční převedení do stavu Detached nebo automatickým nastavením `DetachedOnCommit`. Tím by ale došlo k problémům s opožděným načítáním. Jediným řešením tedy bylo vypnout opožděné načítání nebo přinutit `PersistentManager`, aby při ukončení převáděl entity do stavu Detached. Toho jsem docílil nastavením proměnné `DetachedOnClose`. Nutno je však dodat, že tato vlastnost se nevyskytuje ve specifikaci JDO. Jedná se tedy o rozšiřující vlastnost konkrétní JDO implementace DataNucleus.

iBatis

Jelikož je zde mapování zcela v rukou programátora, nesetkal jsem se zde s žádnými potížemi. Jediné, co stojí za zmínku, byl problém u oboustranné asociaci. Zde docházelo při vypnutí vyrovnávací paměti první úrovně k nekonečné smyčce načítání objektů z databáze.

5.4.3 Přepínání jednotlivých ORM řešení

Jak bylo zmíněno, snažil jsem se, aby přepínání použití jednotlivých ORM řešení v aplikaci bylo co nejjednodušší a mělo co nejmenší či žádný vliv na ostatní části aplikace. To se mi podařilo rozdělením inicializačních parametrů aplikace do oddělených souborů. Výběr těchto souborů se provádí v konfiguračním souboru `web.xml`. Můžeme se podívat, jak by vypadala tato inicializace pro případ, kdy chceme zapnout řešení pomocí iBatis.

48 O této vlastnosti jsem se již zmiňoval v kapitole 4.3.1 Životní cyklus entity.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
        <param-value>
        /WEB-INF/spapp-security.xml
        /WEB-INF/spapp-service.xml
        /WEB-INF/spapp-data.xml

        <!--/WEB-INF/spapp-service-hibernate.xml-->
        /WEB-INF/spapp-service-ibatis.xml
        <!--/WEB-INF/spapp-service-jpa.xml-->
        <!--/WEB-INF/spapp-service-jdo.xml-->

        <!--/WEB-INF/spapp-data-hibernate.xml-->
        /WEB-INF/spapp-data-ibatis.xml
        <!--/WEB-INF/spapp-data-jpa.xml-->
        <!--/WEB-INF/spapp-data-jdo.xml-->
    </param-value>
</context-param>
```

Jak vidíme, jsou odkomentovány dva řádky, které obsahují informace o nastavení konkrétního ORM a vytváření konkrétních DAO objektů. Změna ORM tedy vyžaduje pouze odkomentování konfiguračních souborů požadovaného ORM řešení a zakomentování ostatních konfiguračních souborů.

V případě použití JDO je potřeba ještě zapnout enhancer. Ten se zapíná v souboru `build.xml`, který řídí kompilaci a sestavování aplikace.

5.5 Testování

Jednou z posledních fází vývoje či vývojového cyklu software je testování. V této fázi vývoje se snažíme odhalit chybné chování našeho systému. Testování se dá rozdělit do více skupin podle toho, co nebo jak testujeme.

5.5.1 Typy použitých testování

Testování bezpečnosti

Testování bezpečnosti slouží k nalezení bezpečnostních mezer. Těmi mohou být například neoprávněný přístup, zranitelnost typu SQL-injection. Následně pak na základě získaných informacích o slabinách systému tyto bezpečnostní mezery minimalizovat.

V aplikaci jsem zkusil zmíněný útok SQL-injection či pokus o získání neoprávněného přístupu k operacím pomocí přímého přístupu přes URL. Vůči těmto útokům byla aplikace imunní.

Testování funkčnosti

Testováním funkčnosti se snažíme ověřit, zda systém splňuje funkční požadavky uvedené ve specifikaci. Testování tedy probíhá prováděním jednotlivých scénářů a porovnávání chování vůči specifikovaným požadavkům.

V průběhu implementace jsem prováděl testování funkčnosti jednotlivých scénářů. Po dokončení implementace jsem toto testování zopakoval nad všemi scénáři.

Testování použitelnosti

Při testování použitelnosti nás zajímá, s jakými problémy se setkává uživatel, když s naším systémem pracuje. Získáme tak informace, jak je náš systém pro uživatele přehledný, intuitivní a srozumitelný. Tyto testy se nejčastěji provádí při testech na větším počtu uživatelů. Každý z nich dostane seznam úkolů, které musí v systému provést. Na základě pozorování můžeme zjistit, které věci způsobují uživatelům v systému problémy.

Jako metriku lze použít například čas potřebný k úspěšnému dokončení úkolu či počet chybných úkonů na cestě k úspěšnému dokončení úkolu.

Toto testování jsem prováděl jen na své osobě. Snažil jsem se splnit některé scénáře a přijít tak na kroky, kterými by šly tyto scénáře zpřehlednit a urychlit.

5.5.2 Výsledky testování

Pokud by se jednalo o reálné nasazení do provozu, byl by kladen mnohem větší důraz na samotné testování včetně samotného testování použitelnosti s budoucími uživateli. To by pravděpodobně mělo za následek další požadavky na úpravu uživatelského rozhraní či samotné funkčnosti systému. Při vyvíjení by také bylo vhodné použít jednotkové testování. To by při budoucím zasahování či rozšiřování systému snižovalo pravděpodobnost zanesení nové chyby. Dále by pak umožnilo bezpečnější refaktoring kódu pro zvýšení jeho kvality.

Vývoj a testování probíhalo v následující konfiguraci verzí:

- Apache Tomcat 6.0.26
- Spring 3.0.2
- Spring Security 3.0.2
- Hibernate Validator 4.0.2
- Hibernate 3.5.1
- iBatis 2.3.4.726
- JDO 2.3 + DataNucleus 2.1.0

6 Porovnání implementace jednotlivých ORM řešení

V této části práce bych se rád zaměřil na porovnání kódu potřebného k namapování vybrané třídy. Vybral jsem třídu `Task`. Podívejme se tedy, jak se tato třída v jednotlivých ORM mapovala na databázi. Ukázky metadat jednotlivých ORM řešení jsou doplněny o komentáře, které popisují jednotlivé části těchto metadat.

Z důvodu délky metadat jsou vynechány některé jejich nezajímavé části, které obsahují například opakující se mapování některých jednoduchých typů.

Když se na jednotlivé případy podíváme, zjistíme, že JPA, JDO a Hibernate mají dost podobný zápis, který se liší často jen v názvech tagů. To je ovšem pouze u jednoduchých typů, či cizích klíčů apod. Pokud se jedná například o složitější věc jakou je například mapování výčtového typu, řešení se už dosti podstatně liší.

6.1 JPA metadata

Příklad 6.1 obsahuje popis třídy `Task` pomocí metadat JPA. Můžeme zde vidět, že namapování výčtového typu na typ řetězec zde nečiní žádný problém. Problémem v ukázkové aplikaci však byla chybějící podpora pro definování závislé proměnné `timeSpent`. Ta je v metadatech tedy označena jako `transient`. Takto JPA sdělujeme, že proměnná není namapovaná na databázi a o získání její hodnoty se postaráme sami.

6.2 JDO metadata

V následujícím příkladu 6.2 si ukážeme popis pomocí JDO metadat. Jak můžeme vidět, namapování výčtového typu zde není problém. Opakuje se tu stejný problém jako u JPA, a to chybějící podpora pro definování závislých proměnných. Řešení je tedy obdobné. U této proměnné je nastaven příznak `persistence-modifier="none"`. Ten JDO upozorní na fakt, že tato proměnná není mapována na databázi a o výpočet její hodnoty se nemá starat.

```
<entity class="spapp.domain.Task" metadata-complete="true" name="Task">
    <table name="TASK"/>
    <attributes>
        <basic name="description"><column name="DESCRIPTION"/></basic>
        <!-- vynechání mapování dalších jednoduchých typů -->

        <!-- uložení výčtového typu jako typ řetězec -->
        <basic name="type">
            <column name="TASK_TYPE"/>
            <enumerated>STRING</enumerated>
        </basic>

        <!-- popis vztahu N:1 -->
        <many-to-one name="sprint"
          target-entity="spapp.domain.Sprint">
            <join-column name="ID_SPRINT"/>
        </many-to-one>
        <!-- vynechání mapování dalších vazeb many-to-one -->

        <!-- popis vztahu 1:N -->
        <one-to-many name="workUnits"
          target-entity="spapp.domain.WorkUnit" mapped-by="task"/>

       <!-- JPA nepodporuje odvozené proměnné, tímto způsobem JPA
      sdělujeme, že tato proměnná nemá být mapovaná na databázi. O její
      výpočet se poté musíme postarat sami. -->
        <transient name="timeSpent"/>
    </attributes>
</entity>
```

Příklad 6.1: Popis třídy Task pomocí metadat JPA

```
<class name="Task" detachable="true" table="TASK"
      persistence-capable-superclass="spapp.domain.NamedEntity"
      identity-type="application">
    <inheritance strategy="new-table"/>

    <field name="description" column="DESCRIPTION"/>
    <!-- vynechání mapování dalších jednoduchých typů -->

    <!-- uložení výčtového typu jako typ řetězec -->
    <field name="type">
        <column name="TASK_TYPE" jdbc-type="VARCHAR" length="255"/>
    </field>

    <!-- popis vztahu N:1 -->
    <field name="sprint" column="ID_SPRINT"/>
    <!-- vynechání mapování dalších vazeb many-to-one -->

    <!-- popis vztahu 1:N -->
    <field name="workUnits" mapped-by="task">
        <collection element-type="spapp.domain.WorkUnit"/>
    </field>

    <!-- JDO nepodporuje odvozené proměnné, tímto způsobem JDO sdělujeme,
      že tato proměnná nemá být mapovaná na databázi. O její výpočet se
      poté musíme postarat sami. -->
    <field name="timeSpent" persistence-modifier="none"/>
</class>
```

Příklad 6.2: Popis třídy Task pomocí metadat JDO

6.3 Hibernate metadata

Na příkladu 6.3 můžeme vidět popis třídy `Task` pomocí metadat Hibernate. Jak už bylo zmíněno, je zde vidět nutnost definování nového typu `taskT` a použití již zmíněné magické konstanty. Definování nového typu bylo nutné pro možnost uložení výčtového typu jako typ řetězec. Dále zde můžeme vidět, jak se definuje závislá proměnná `timeSpent`.

```
<!-- Definování nového typu taskT kvůli možnosti uložit výčtový
    typ jako řetězec.  -->
<typedef class="org.hibernate.type.EnumType" name="taskT">
  <param name="enumClass">spapp.domain.TaskType</param>
  <!-- 12 = konstatní hodnota z java.sql.Types -->
  <param name="type">12</param>
</typedef>

<class name="spapp.domain.Task" table="TASK">

  <property column="DESRIPTION" name="description" not-null="true"/>
  <!-- vynechání mapování dalších jednoduchých typů -->

  <!-- uložení výčtového typu jako typ řetězec -->
  <property column="TASK_TYPE" name="type" not-null="true"
          type="taskT"/>

  <!-- popis vztahu N:1 -->
  <many-to-one name="sprint"
          class="spapp.domain.Sprint"
          column="ID_SPRINT"
          not-null="true"/>
  <!-- vynechání mapování dalších vazeb many-to-one -->

  <!-- popis vztahu 1:N -->
  <set access="property"
          inverse="true"
          name="workUnitsInternal">
    <key column="ID_TASK"/>
    <one-to-many class="spapp.domain.WorkUnit"/>
  </set>

  <!-- definování odvozené proměnné timeSpent -->
  <property name="timeSpent"
          formula="(select sum(wu.TIMESPENT)
                from WORKUNIT wu
                where wu.ID_TASK = ID)"
          type="long"/>
</class>
```

Příklad 6.3: Popis třídy Task pomocí metadat Hibernate

6.5 iBatis metadata

U iBatis je mapování řešeno jiným způsobem. Pro jednotlivé objekty se zde vytváří tzv. mapy. Mapa obsahuje seznam proměnných namapovaných na jednotlivé sloupce. Výsledky dotazů, které ručně vytváříme, jsou pak namapovány na tyto mapy. Mapování iBatisu můžeme vidět na příkladu 6.4.

Při mapování složitějších typů v iBatisu se nám nabízí implementace rozhraní `TypeHandlerCallback`. Implementací metod tohoto rozhraní tak můžeme iBatisu ukázat, jak tyto objekty může namapovat na jednoduché typy. To se mi v ukázkové aplikaci hodilo při ukládání výčtového typu na typ řetězec.

```
<sqlMap namespace="Task">
  <!-- vytvoření mapy result -->
  <resultMap id="result" class="spapp.domain.Task">
    <result property="description" column="DESCRIPTION"/>
    <!-- vynechání mapování dalších jednoduchých typů -->

    <!-- uložení výčtového typu jako typ řetězec -->
    <result property="type" column="TASK_TYPE"
            typeHandler="ibatis.TaskTypeHandler"/>

    <!-- popis vztahu 1:N -->
    <result property="sprint" column="ID_SPRINT" select="Sprint.findById"/>

    <result property="timeSpent" column="TIMESPENT"/>
  </resultMap>

  <!-- vytvoření sql dotazu s namapováním výsledku na mapu result -->
  <select id="findById" resultMap="result">
    select *,
        (
        select sum(WORKUNIT.TIMESPENT)
        from WORKUNIT
        where WORKUNIT.ID_TASK = TASK.ID
        ) as TIMESPENT
    from TASK
    where TASK.ID = #value#
  </select>
</sqlMap>
```

Příklad 6.4: Popis třídy Task pomocí metadat iBatis

7 Srovnání ORM řešení

V této kapitole se budu věnovat srovnání vlastností jednotlivých ORM řešení. Nejdříve provedu výběr kritérií. U každého kritéria stanovím jeho váhu, tedy jakou důležitost má. Následuje slovní hodnocení jednotlivých kritérií, jak dané ORM řešení toto kritérium splňuje. Na základě této informace bude ORM ohodnoceno body.

7.1 Kritéria

V této části bych se rád zaměřil na vybraná kritéria, ve kterých jsem jednotlivé ORM porovnával. Jedná se o výběr kritérií, které jsem buď použil v ukázkové aplikaci nebo se mi zdála zajímavá.

Některá z kritérií jsou značně subjektivní. Jedná se například o obtížnost osvojení, kde porovnávám, jak dlouho mi zabralo toto řešení zprovoznit či naučit se s ním efektivně pracovat.

Dědičnost

Toto kritérium popisuje škálovatelnost různých strategií mapování dědičnosti. Zda je strategie definována na úrovni celého stromu dědičností nebo jestli umožňuje nastavení různých strategií pro jednotlivé podstromy.

Vynucená závislost

Toto kritérium posuzuje vynucenou závislost ORM řešení na objektech určené k perzistenci. Jestli naše objekty musí implementovat nějaké rozhraní či rozšiřovat nějakou třídu konkrétního ORM řešení nebo jestli je toto zajištěno pomocí enhanceru.

Podpora různých datových úložišť

Jak si stojí různá ORM řešení v podpoře různých datových úložišť. Mohou být použity pouze relační databáze nebo i jiná datová úložiště.

Podpora odvozených proměnných

Podporuje ORM odvozené proměnné, jak bylo zmíněno v kapitole 4.

Obtížnost osvojení

Jak již bylo zmíněno toto kritérium je značně subjektivní. Jedná se v podstatě o porovnání času, který jsem potřeboval k pochopení a zprovoznění jednotlivých řešení. Tyto rozdíly totiž byly značné a informace o nich mohou být užitečné například pro některé firmy, které se rozhodují, zda se přechod na některou z těchto technologií vyplatí.

Podpora výčtových typů/kolekce výčtových typů

Jak jednotlivá ORM řešení podporují ukládání výčtového typu, popřípadě i kolekci výčtových typů.

Rozšířenost

V tomto kritériu se posuzuje, jak jsou jednotlivá řešení rozšířená a podporována. Jestli mají širokou komunitu. To může mít například pozitivní vliv při hledání problémů, které již určitě někdo řešil před námi.

Podpora opožděného/včasného načítání

Zda je možnost u ORM používat opožděné načítání.

Podpora různých způsobů dotazování

Jak nám umožňuje ORM pokládat dotazy na datové úložiště? Zda vytváří vlastní jazyk pro kladení dotazů, či používá nativní dotazy konkrétního úložiště. Rozšiřuje tyto možnosti kladením dotazů i pomocí API?

Podpora transakcí

Zda podporují jednotlivá ORM transakční zpracování.

7.2 Váhy kritérií

V tabulce 7.1 můžeme vidět seznam jednotlivých kritérií ohodnocený patřičnými váhami. Ty jsem stanovil podle posouzení jejich důležitosti. Váhy se pohybují ve stupnici od 0 do 5. Zde platí, čím vyšší důležitost, tím vyšší číslo.

7.3 Výsledné hodnocení

Výsledné hodnocení můžeme vidět v tabulce 7.1. U každého ORM a jednotlivého kritéria hodnocení je stručně popsáno, jak se ORM toto kritérium daří plnit. V dalším sloupci je pak úroveň těchto požadavků ohodnocena kladnými body. Body se pohybují ve stupnici od 0 do 5. Zisk 0 bodů je tedy nejhorší možné ohodnocení, naproti tomu zisk 5 bodů znamená nejlepší možné ohodnocení.

Výsledné hodnocení získaných bodů jednotlivých ORM je pak spočítáno jako suma bodů získaných za jednotlivá kritéria vynásobených jejich váhami.

Kritérium	Váha	JPA		JDO		Hibernate		iBatis	
Dědičnost	4	Jedna strategie pro celou hierarchii dědičnosti.	3	Každá třída má vlastní strategii.	5	Jedna strategie pro celou hierarchii dědičnosti.	3	S iBatisem máme možnost namapovat hierarchii dědičnosti jak chceme, protože mapování provádí ručně programátor.	3
Vynucená závislost	5	Ne. Třídy jsou obohaceny enhancerem, nemusí implementovat žádné rozhraní či rozšiřovat nějakou třídu.	5	Ne. Třídy jsou obohaceny enhancerem, nemusí implementovat žádné rozhraní či rozšiřovat nějakou třídu.	5	Ne. Třídy jsou obohaceny enhancerem, nemusí implementovat žádné rozhraní či rozšiřovat nějakou třídu.	5	Ne. Třídy jsou obohaceny enhancerem, nemusí implementovat žádné rozhraní či rozšiřovat nějakou třídu.	5
Podpora různých datových úložišť	3	Pouze relační databáze.	3	Datové úložiště může být jakékoliv. Implementace JDO podporují nejrůznější druhy datových uložišť od relačních databází přes LDAP, objektové databáze, XML, ODT a mnoho dalších.	5	Pouze relační databáze.	3	Pouze relační databáze.	3
Podpora odvozených proměnných	4	Ne.	0	Ne.	0	Ano.	5	Odvozenou proměnou si můžeme namapovat ručně.	3
Obtížnost osvojení	4	Osvojení a implementace pomocí JPA mi trvala 7 dní.	3	Osvojení a implementace pomocí JDO mi trvala 14 dní.	1	Osvojení a implementace pomocí Hibernate mi trvala 5 dní.	4	Osvojení a implementace pomocí iBatis mi trvala 1 den.	5
Podpora výčtových typů/kolekce výčtových typů	2	Podporuje ukládání výčtového typu jako řetězec. Chybí zde však již podpora pro kolekci výčtových. Ta by měla být dostupná od verze JPA 2.	2	Podporuje ukládání výčtového typu jako řetězec a to i pro kolekce výčtových typů.	5	Podporuje ukládání výčtového typu jako řetězec a to i pro kolekce výčtových typů.	5	Máme možnost vytvořit převáděcí objekt, který nám tuto podporu zajistí. Automatická podpora chybí.	3
Rozšířenost	3	Střední/Vysoká.	4	Nízká.	2	Vysoká.	5	Střední.	4
Podpora opožděného/včasného načítání	4	Ano.	5	Ano.	5	Ano.	5	Ano.	5
Podpora různých způsobů dotazování	3	JPQL, nativní SQL. Od verze JPA 2 i Ciriteria API.	2	JDOQL, SQL.	2	HQL, JPQL, Criteria API, nativní SQL.	5	Nativní SQL.	1
Podpora transakcí	4	Ano.	5	Ano.	5	Ano.	5	Ano.	5

Tabulka 7.1: Tabulka s hodnocením

Pokud tedy v ohodnocení zohledníme váhy jednotlivých kritérií dostaneme výsledky, které můžeme vidět v tabulce 7.2.

Kritérium	Váha	JPA	JDO	Hibernate	iBatis
Dědičnost	4	3 x 4 = 12	5 x 4 = 20	3 x 4 = 12	3 x 4 = 12
Vynucená závislost	5	5 x 5 = 25	5 x 5 = 25	5 x 5 = 25	5 x 5 = 25
Podpora různých datových úložišť	3	3 x 3 = 9	5 x 3 = 15	3 x 3 = 9	3 x 3 = 9
Podpora odvozených proměnných	4	0 x 4 = 0	0 x 4 = 0	5 x 4 = 20	3 x 4 = 12
Obtížnost osvojení	4	3 x 4 = 12	1 x 4 = 4	4 x 4 = 16	5 x 4 = 20
Podpora výčtových typů/kolekce výčtových typů	2	2 x 2 = 4	5 x 2 = 10	5 x 2 = 10	3 x 2 = 6
Rozšířenost	3	4 x 3 = 12	2 x 3 = 6	5 x 3 = 15	4 x 3 = 12
Podpora opožděného/včasného načítání	4	5 x 4 = 20	5 x 4 = 20	5 x 4 = 20	5 x 4 = 20
Podpora různých způsobů dotazování	3	2 x 3 = 6	2 x 3 = 6	5 x 3 = 15	1 x 3 = 3
Podpora transakcí	4	5 x 4 = 20	5 x 4 = 20	5 x 4 = 20	5 x 4 = 20
Celkem		120	126	162	139

Jak vidíme, nejvíce bodů získal Hibernate, velmi kvalitní ORM se širokým rozšířením mezi uživateli a podporou u různých rámců. O něco méně bodů získal iBatis. U toho však nesmíme zapomenout, že na rozdíl od ostatních řešení neautomatizuje psaní dotazů. Tudíž v tomto ohledu nešetří tolik času při vývoji jako jeho protivníci. JPA a JDO pak obdrželo skoro stejné množství bodů. U JPA je nutno ještě podotknout, že některé výtky, kvůli kterým ztratil body, už jsou zapracovány v nové verzi JPA 2.

8 Závěr

V této diplomové práci bylo mým úkolem porovnat technologie pro objektově relační mapování v jazyce Java a demonstrovat je na ukázkové aplikaci. A to způsobem, aby aplikace měla minimální závislost na konkrétní použité technologii.

Práci jsem započal studiem postupů a problémů této techniky s hledáním vhodných kandidátů k porovnání. Dále pak sběrem požadavků a návrhu ukázkové aplikace. Hlavním pilířem práce pak byla implementace jednotlivých technologií pro objektově relační mapování, a to JPA, JDO, Hibernate a iBatis. Během implementace jsem si dělal poznámky, s jakými jsem se setkal problémy či nedostatky, které jsem pak v práci popsal. V závěru práce jsem pak na základě zjištěných informací provedl celkové ohodnocení. To jsem provedl stanovením kritérií a vah podle jejich důležitosti. Každou ORM technologii jsem pro dané kritérium slovně a bodově ohodnotil podle toho, jak toto kritérium splňovala. Vybraná kritéria a hodnocení můžete vidět v předposlední kapitole.

Nejlépe si vedl rámec Hibernate, který získal nejvíce bodů. Na druhé příčce se umístilo řešení iBatis, které získalo o 15% bodů méně. Standard JPA s implementací Hibernate a standard JDO s implementací DataNucleus získaly téměř shodně bodů, zaostávající za rámcem Hibernate téměř o 25% získaných bodů.

Při tvorbě této práce jsem se detailně seznámil s tím, jak objektově relační mapování funguje. Jaké výhody ale i nástrahy nám přináší. To mi při tvorbě budoucích projektů umožní lépe vybírat vhodné techniky pro perzistenci a zlepšit odhad přínosu či nákladů na použití těchto technik. Dále jsem se seznámil i s technikami, které nebyly přímou součástí zadání práce. Mezi ně patří standard Java EE, technika Bean Validation a další, které se mi budou hodit v mém dalším působení.

Literatura

[1] BAUER, Christian, KING, Gavin. *Java Persistence with Hibernate : Revised Edition of Hibernate in Action*. 2nd rev. edition. [s.l.] : Manning, 2006. 880 s. ISBN 1-932394-88-5.

[2] KEITH, Mike, SCHINCARIOL, Merrick. *Pro JPA 2 : Mastering the Java™ Persistence API*. [s.l.] : Apress, 2009. 550 s. ISBN 978-1-4302-1956-9.

[3] WALLS, Craig, BREIDENBACH, Ryan. *Spring in Action : Second Edition*. 2nd edition. [s.l.] : Manning, 2007. 768 s. ISBN 1-933988-13-4.

[4] ŠENK, Zdeněk. *Technologie pro perzistenci objektů v Javě*. [s.l.], 2007. 124 s. FIT VUT v Brně. Vedoucí diplomové práce Doc. Ing. Jaroslav Zendulka, CSc.

[5] MASSOL, Vincent, HUSTED, Ted. *JUnit in Action*. [s.l.] : [s.n.], 2003. 384 s. ISBN 1-930110-99-5.

[6] PRICE, Jason. *Java Programming with Oracle SQLJ*. [s.l.] : O'Reilly, 2001. 400 s. ISBN 0-596-00087-1.

[7] ZENDULKA, Jaroslav. *Databázové systémy : IDS* [online]. Brno, 2006. 217 s. Studijní opora. FIT VUT v Brně. Dostupné z WWW: <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IDS-IT/texts/IDS_predn.pdf>.

[8] SEDDIGHI, Ahmad Reza. *Spring Persistence with Hibernate*. [s.l.] : Packt Publishing, 2009. 460 s. ISBN 978-1-849510-56-1.

[9] *Developer Resources for Java Technology* [online]. 2006 [cit. 2010-04-01]. Java EE at a Glance. Dostupné z WWW: <http://java.sun.com/javaee/>.

[10] ŠVEC, Martin. *Objektové databáze* [online]. [s.l.], 2003. 20 s. Referát. FIT VUT v Brně. Dostupné z WWW: <http://www.fit.vutbr.cz/study/courses/VPD/public/0203VPD-Svec.pdf>.

[11] *DataNucleus AccessPlatform* [online]. 2010, 15.04.2010 [cit. 2010-04-15]. DataNucleus. Dostupné z WWW: <http://www.datanucleus.org/products/accessplatform/>.

[12] *Kodo™ 4.2.0.load03 Developers Guide for JPA/JDO* [online]. 2008 [cit. 2010-04-17]. Kodo™ 4.2.0.load03 Developers Guide for JPA/JDO. Dostupné z WWW: <http://otndnld.oracle.co.jp/document/products/wls/docs103/kodo/full/html/manual.html>.

[13] KINSKÝ, Filip. *Databázový svět : Informační portál ze světa databázových technologií* [online]. 30.4. 2004 [cit. 2010-04-23]. Technologie Java Data Objects. Dostupné z WWW: <http://www.dbsvet.cz/view.php?cisloclanku=2004043001>.

[14] KING, Gavin. *HIBERNATE : Relational Persistence for Idiomatic Java* [online]. 3.5.1-Final. April 14, 2010 [cit. 2010-04-23]. Hibernate Reference Documentation. Dostupné z WWW: <http://docs.jboss.org/hibernate/core/3.5/reference/en-US/html/>.

[15] BEGIN, Clinton; GOODIN, Brandon; MEADORS, Larry. *IBATIS in Action*. [s.l.] : Manning, 2007. 384 s. ISBN 1-932394-82-6.

[16] VALÍČEK, Arnošt. *Objektově-relační mapování v Javě*. Brno, 2007. 91 s. Diplomová práce. Masarykova Univerzita.

[17] *Apache OpenJPA 2.0 User* [online]. 2.0. 22-Apr-2010 [cit. 2010-04-26]. Apache OpenJPA 2.0 User. Dostupné z WWW: <http://openjpa.apache.org/builds/2.0.0/apache-openjpa/docs/>.

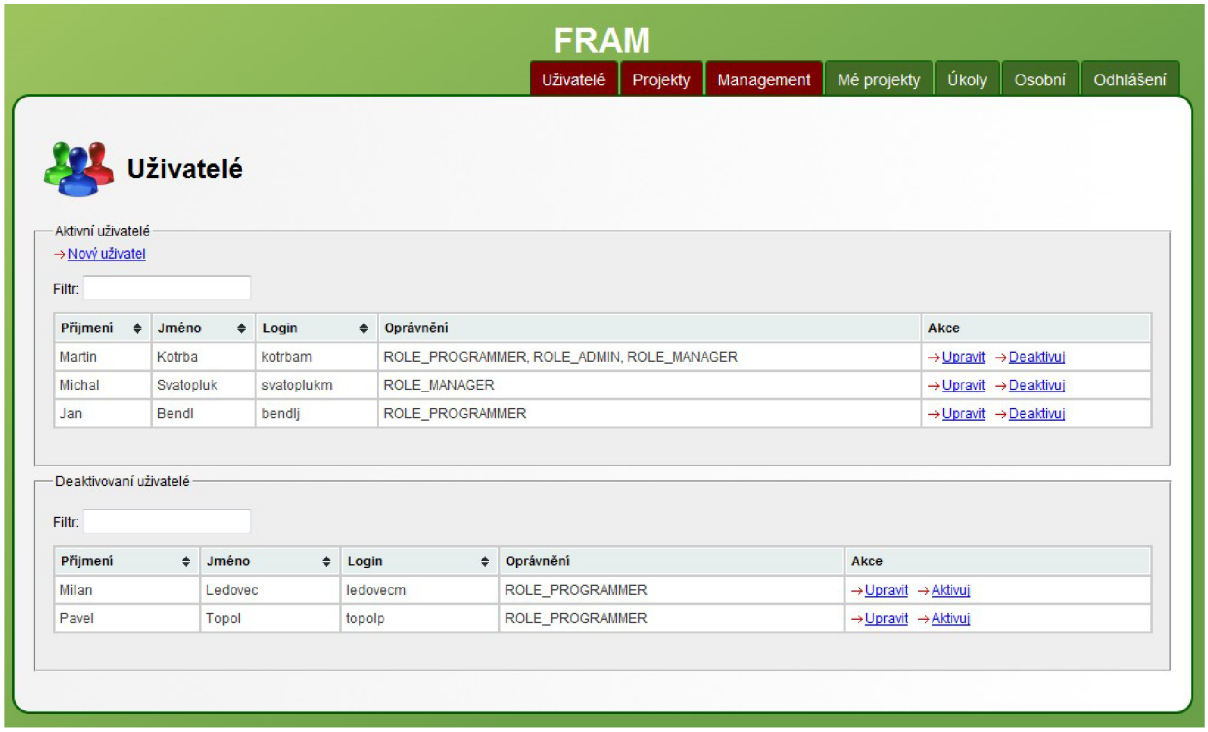
[18] *Hibernate Tutorial* [online]. 2008 [cit. 2010-04-26]. Hibernate Tutorial. Dostupné z WWW: <http://www.hibernate-training-guide.com/>.

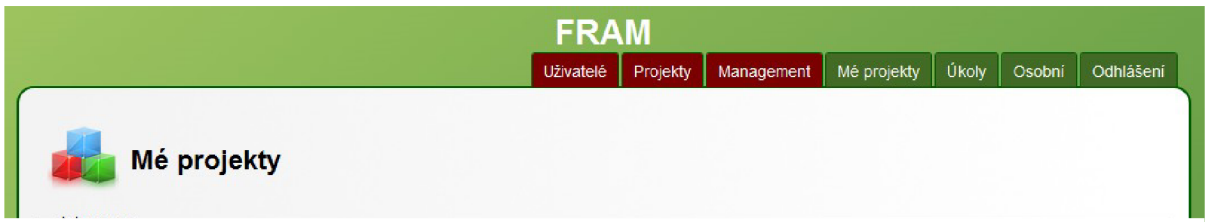
[19] ROOS, Robin M. *Java Data Objects*. Great Britain : Pearson Education, 2003. 263 s. ISBN 0-321-12380-8.

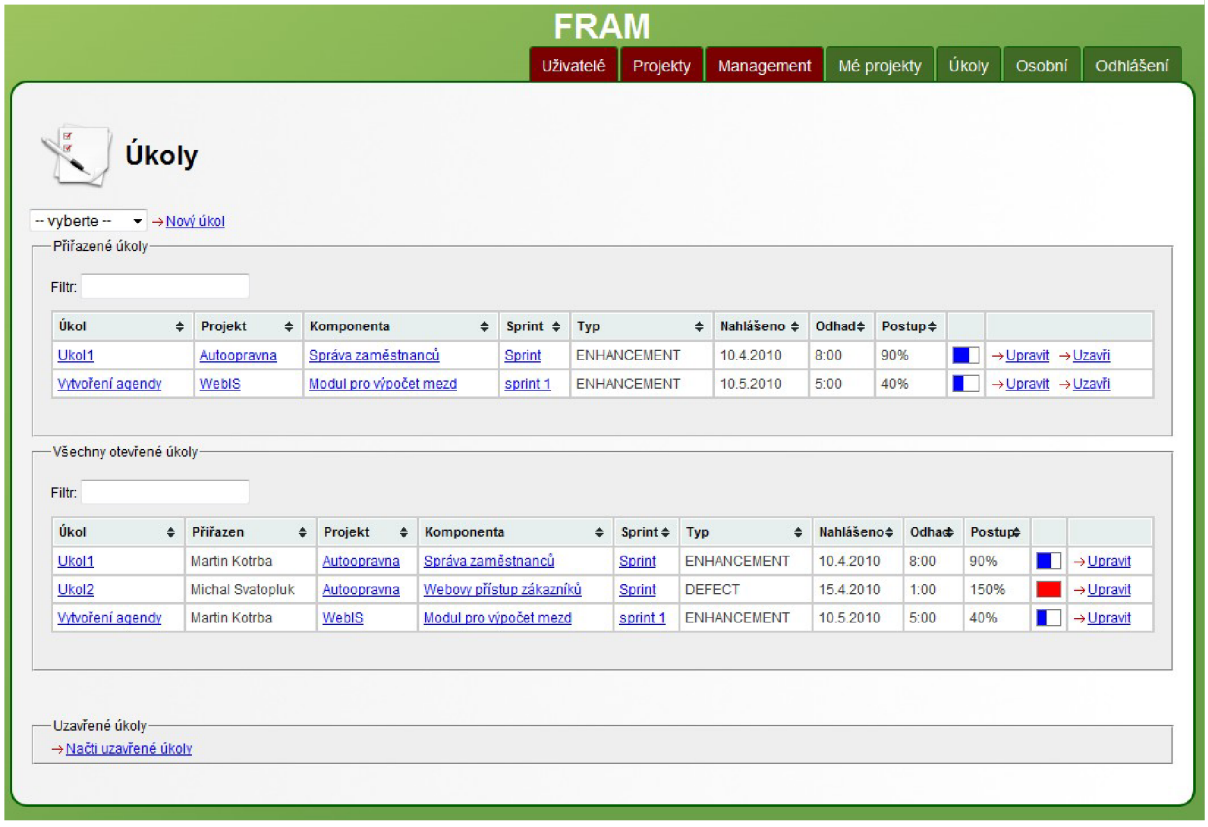
[20] DUDEK, Jan. *Návrhové vzory pro webové aplikace* [online]. [s.l.], 2008. 86 s. Diplomová práce. FIT VUT v Brně. Dostupné z WWW: <http://www.fit.vutbr.cz/study/DP/rpfile.php?id=6190>.

Seznam příloh

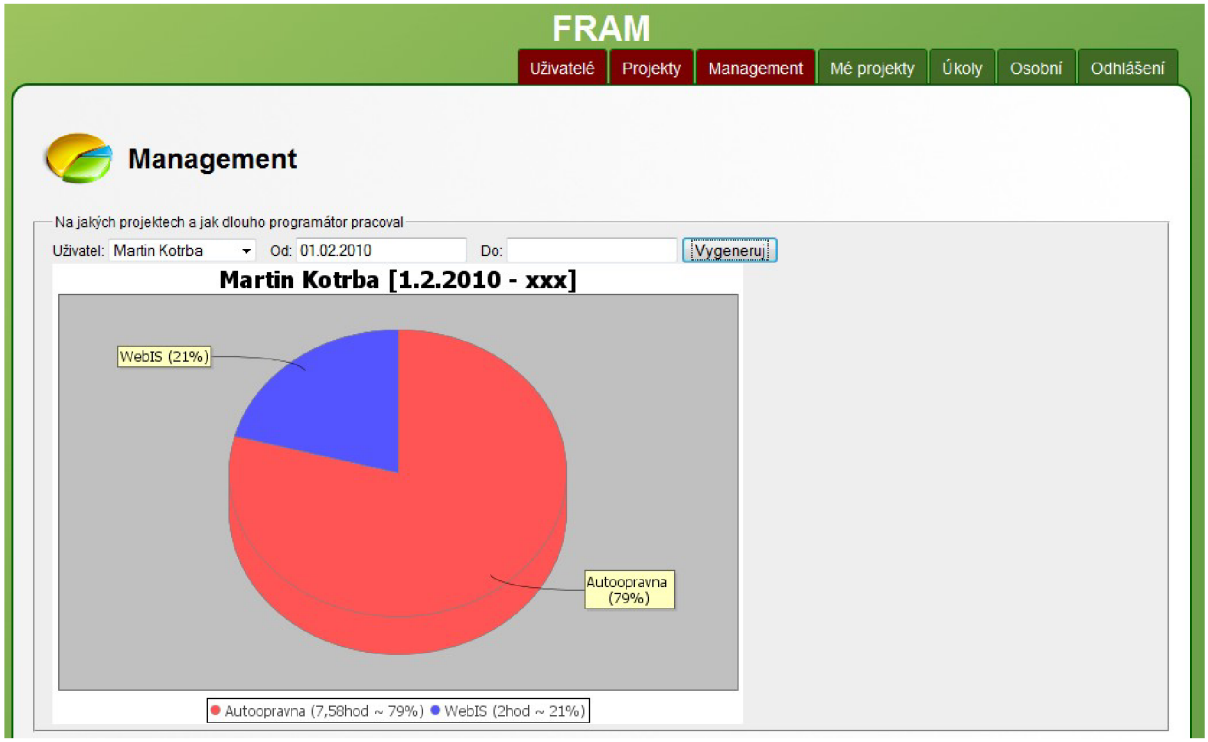
Příloha 1 - Některé z obrazovek aplikace







Ilustrace 8.3: Ukázková aplikace: Seznam úkolů



Ilustrace 8.4: Ukázková aplikace: Management výstup