VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

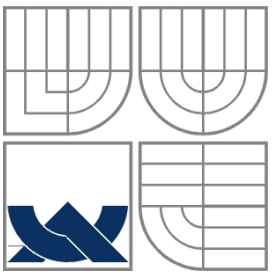
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TECHNOLOGIE PRO PERZISTENCI OBJEKTŮ V JAVĚ

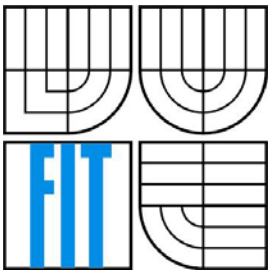
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ZDENĚK ŠENK
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TECHNOLOGIE PRO PERZISTENCI OBJEKTŮ V JAVĚ

TECHNOLOGIES FOR PERSISTENT OBJECTS IN JAVA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ZDENĚK ŠENK

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. JAROSLAV ZENDULKA, CSc.

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání diplomové práce

Řešitel: **Šenk Zdeněk, Bc.**
Obor: Informační systémy
Téma: **Technologie pro perzistenci objektů v Javě**
Kategorie: Databáze

Pokyny:

1. Seznamte se s problematikou použití perzistentních objektů v jazyce Java a přístupy a technologiemi na podporu práce s nimi.
2. Proveďte klasifikaci používaných přístupů, standardů a technologií a vzájemně je porovnejte.
3. Po dohodě s vedoucím diplomové práce navrhněte ukázkovou aplikaci, na které budete ilustrovat způsob použití dvou různých technologií.
4. Ukázkovou aplikaci implementujte a ověřte funkčnost na vhodně zvoleném vzorku dat.
5. Zhodnoťte použité technologie z hlediska vašich praktických zkušeností.

Literatura:

- Arnold, K., Gosling, J., Holmes, D.: Java programming language, Addison-Wesley, 2006.
- Java SE - Java Database Connectivity (JDBC). Available at http://java.sun.com/javase/technologies/database/index.jsp.
- Java Data Objects (JDO). Web site http://java.sun.com/products/jdo/.
- Enterprise JavaBeans technology. Available at http://java.sun.com/products/ejb/.
- Hibernate. Available at http://www.hibernate.org/

Při obhajobě semestrální části diplomového projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zendulka Jaroslav, doc. Ing., CSc.**, UIFS FIT VUT
Datum zadání: 28. února 2006
Datum odevzdání: 22. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Bc. Zdeněk Šenk**
Id studenta: 22559
Bytem: Staré Město pod Landštejnem 16, 378 81 Slavonice
Narozen: 31. 07. 1982, Jihlava
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
diplomová práce

Název VŠKP: Technologie pro perzistenci objektů v Javě
Vedoucí/školitel VŠKP: Zendulka Jaroslav, doc. Ing., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☒ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Diplomová práce se zabývá technologiemi pro perzistenci objektů v Javě. Stručně popisuje možnosti perzistence objektů do souborů a dále především moderní technologie pro perzistenci objektů do databázových systémů, zejména relačních. Poměrně podrobně se zabývá technologií JDBC, rámcem Hibernate, aplikačním rozhraním JDO, technologií SQLJ, rámci OJB a TopLink a stručně i dalšími nástroji. Důraz je kladen zejména na jejich použití, které je doprovázeno množstvím příkladů, vzájemné porovnání jejich vlastností a doporučení pro tvorbu aplikací. Druhá část práce se věnuje jednotlivým životním fázím vývoje informačního systému, na kterém jsou prezentovány odlišnosti použití JDBC a Hibernate, jejich vzájemné porovnání z mnoha hledisek a význam softwarových architektur a návrhových vzorů pro tvorbu aplikací v prostředí J2EE.

Klíčová slova

Databáze, ORM, J2EE, JDBC, Hibernate, EJB, JDO, SQL, SQLJ, TopLink, OJB, MVC, DAO, JSP

Abstract

This master thesis deals with technologies for persistence of objects in Java. It briefly describes options of persistence of objects into files and further especially modern technologies for persistence of objects into database systems, particularly relational. It describes JDBC technology, framework Hibernate, application interface JDO, SQLJ technology, frameworks OJB and TopLink in detail and briefly the other tools. The emphasis is especially put on their usage, which is illustrated by amount of examples, their properties comparison and recommendation for the application building. The other part of thesis deals with life phases of information system development, where the differences of usage of JDBC and Hibernate, their mutual comparison from different points of view, importance of software architecture and design patterns for the application development in J2EE are presented.

Keywords

Database, ORM, J2EE, JDBC, Hibernate, EJB, JDO, SQL, SQLJ, TopLink, OJB, MVC, DAO, JSP

Citace

Zdeněk Šenk: Technologie pro perzistenci objektů v Javě, diplomová práce, Brno, FIT VUT v Brně, 2007

Technologie pro perzistenci objektů v Javě

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.

.....
Zdeněk Šenk
22. 5. 2007

Poděkování

Chtěl bych na tomto místě poděkovat doc. Ing. Jaroslavu Zendulkovi, CSc. za odborné vedení a konzultace, které mi poskytl v teoretické i implementační části diplomové práce.

Obsah

1 Úvod

Počítače zpracovávají data od počátku své historie a schopnost aplikace ukládat data, se kterými pracuje, je dnes považována za naprostou samozřejmost. Při vývoji aplikací stráví programátor značné množství času právě psaním kódu, který s těmito daty pracuje. K tomuto účelu byly postupem času vyvinuty standardní postupy a nástroje, které tuto oblast řeší a mohou nám výrazně pomoci při implementaci perzistence.

Tato diplomová práce se zabývá moderními technologiemi pro perzistenci objektů jazyka Java. V druhé kapitole se seznámíme se základními vlastnostmi jazyka Java a stručně i s nejjednoduššími způsoby perzistence objektů, kterými jsou serializace a ukládání do XML souborů. Dále se v této práci budeme věnovat ukládání objektů do databázových systémů, které nám oproti ukládání objektů do souborů přinášejí řadu výhod. Ve třetí kapitole se seznámíme s aplikačním rozhraním JDBC, pomocí kterého lze velmi jednoduchým způsobem pracovat s daty, která jsou strukturována do tabulek. Typickým zdrojem dat, ke kterému pomocí JDBC přistupujeme, jsou relační databáze. Uvidíme, jakým způsobem nám JDBC umožňuje pracovat s daty v databázi, jak je možné provést optimalizaci jeho výkonu a seznámíme se s novinkami, které přináší jeho verze 3.0 a 4.0.

V současné době se největší pozornost soustřeďuje na oblast tzv. objektově-relačního mapování (ORM). Cílem nástrojů, které toto mapování zajišťují, je řešení problému, který se nazývá *paradigm mismatch*. Tento termín, do češtiny přeložitelný jako nesoulad paradigmat, označuje problém převodu dat mezi objektovou a relační formou. Tento problém se postupem času stává poměrně naléhavým, protože převážná většina aplikací je implementována v objektově orientovaných jazycích, zatímco data jsou z důvodu nízkého rozšíření objektově orientovaných databázových systému stále ukládána převážně do relačních databází, které díky několika desetiletím vývoje poskytují velmi propracované a výkonné techniky pro ukládání a získávání dat, podpořené matematickým aparátem v podobě relační algebry.

Čtvrtá kapitola se poměrně podrobně věnuje rámci Hibernate, který je volně dostupným a v současné době nejpoužívanějším nástrojem pro ORM. Podíváme se zde na jeho architekturu, způsoby konfigurace v neřízením i řízeném prostředí (v rámci nějakého aplikačního serveru), možnosti, jak lze nastavit mapování tříd do databáze, a samozřejmě se seznámíme i se způsoby práce s perzistentními objekty, dotazovacím jazykem HQL a dalšími technikami pro dotazování se. V závěru této kapitoly se ještě stručně seznámíme s některými nástroji, které nepatří do samotného jádra Hibernate, ale které nám mohou výrazně pomocí při vývoji aplikace. V páté kapitole se seznámíme s technologií EJB. Tato technologie je velmi rozsáhlá a proto je tato kapitola pouze stručným úvodem do ní. Je zde stručně popsána architektura EJB, typy Enterprise beanů a možnosti perzistence dat v EJB 3.0. Šestá kapitola je zaměřena na standardizované aplikační rozhraní JDO od společnosti Sun. Je zde opět zmíněna jeho architektura, způsoby konfigurace perzistence a možnosti

práce s perzistentními objekty. V sedmé kapitole se již pouze velmi stručně seznámíme s některými dalšími technologiemi pro perzistenci objektů v Javě. První z nich je SQLJ, což je technologie, která se na první pohled velmi podobá JDBC, avšak oproti němu přináší několik velmi podstatných výhod, na které se zaměříme především. Další technologií je ObJectRelationalBridge. OJB patří mezi volně dostupné rámce pro perzistenci a disponuje některými zajímavými vlastnostmi. Opět se seznámíme s tím, jakým způsobem je třeba provést jeho konfiguraci, co musí splňovat perzistentní třídy a jak lze pomocí něj pracovat s perzistentními objekty. Třetí technologií, která je v této kapitole popsána, je TopLink. Ten je jedním z nejvyspělejších nástrojů pro ORM, který má za sebou více než desetiletí vývoje. Vzhledem k jeho rozsáhlosti se omezíme pouze na stručný popis jeho hlavních rysů a výhod oproti jiným nástrojům. V závěru této kapitoly se již pouze v několika větách zmíníme o dalších rámcích pro perzistenci, kterými jsou CocoBase, Torque a Cayenne, kde se zaměříme zejména na rysy, kterými se tyto nástroje nějakým způsobem vyvyšují nad ostatní.

Následující, osmá kapitola, tvoří předěl mezi částí teoretickou a praktickou. Nachází se v ní stručná specifikace vyvíjeného systému a definice jeho hlavního cíle, kterým je porovnání implementace perzistence pomocí JDBC a Hibernate. Navazující kapitoly se po řadě zabývají jednotlivými fázemi životního cyklu projektu. Devátá kapitola je věnována sběru a analýze požadavků kladených na systém. Jsou zde detailně probrány jednotlivé případy použití, které jsou graficky znázorněny pomocí příslušných diagramů. Na analýzu požadavků navazuje desátá kapitola, která se zabývá návrhem budoucího systému. Nejprve je v této kapitole probrán význam softwarových architektur a návrhových vzorů. Dále jsou zde uvedeny základní informace o architektuře MVC a návrhovém vzoru DAO, které jsou spojeny a zasazeny do kontextu prostředí J2EE aplikace, čímž tvoří návrh architektury budoucího systému. V dalších částech této kapitoly jsou uvedeny diagramy tříd a rozhraní jednotlivých vrstev aplikace, které jsou vždy podrobně okomentovány. V závěru této kapitoly je popsán návrh uživatelského rozhraní a základní pravidla, kterými jsem se při jeho implementaci řídil. Jedenáctá kapitola se zabývá samotnou implementací navrženého systému. Za pomocí ukázek zdrojových kódů jsou zde podrobně probrány implementační aspekty jednotlivých vrstev aplikace, které jsou nějakým způsobem zajímavé nebo výjimečné. Dále jsou zde zdůrazněny odlišnosti v implementaci vrstvy DAO v jednotlivých verzích aplikace a je zde upozorněno na naopak minimální změny ostatních vrstev aplikace, za což vděčíme rámci MVC a vzoru DAO. Dále je v této kapitole prostor pro popis způsobu, jakým byl systém testován a na úplném závěru je provedeno porovnání implementace tříd vrstvy DAO pomocí JDBC a Hibernate z hlediska programátora. Předposlední, kapitola, se opět zaměřuje na třídy vrstvy DAO, ale tentokrát z pohledu porovnání výkonnosti jednotlivých implementací. Poslední kapitolou je závěr, který celou práci shrnuje.

Diplomová práce navazuje na semestrální projekt, v jehož rámci byly splněny první dva body z jejího zadání.

2 Jazyk JAVA a perzistence objektů

2.1 Jazyk JAVA

Historie jazyka JAVA sahá do roku 1995. Tehdy firma Sun Microsystems, která tento jazyk vyvíjí, uvedla jeho první verzi. V současné době (říjen 2006) je používána verze 1.5 s označením Tiger a k dispozici je již beta verze Javy 1.6 s označením Mustang.

Díky výborným vlastnostem, které budou stručně popsány dále, se postupem času stal jazyk Java jedním z nejpoužívanějších programovacích jazyků a do budoucna je možné ještě očekávat nárůst jeho popularity.

V Javě lze elegantně psát nejrůznější druhy aplikací. Existuje technologie *JavaCard*, která je určena pro bezpečný běh aplikací, nazývaných JavaCard aplety, na paměťových kartách a podobných zařízeních s omezenou pamětí. Další platformou je *Java Micro Edition* (Java ME), která je určena k vývoji aplikací pro mobilní telefony a vestavěné systémy. K vývoji aplikací pro desktopové počítače (ať již konzolových nebo s grafickým uživatelským rozhraním) slouží technologie *Java Standard Edition* (Java SE). V neposlední řadě také Java umožňuje psát rozsáhlé distribuované systémy, které mohou pracovat na různých počítačích propojených pomocí internetu – platforma *Java Enterprise Edition* (Java EE).

2.1.1 Základní vlastnosti jazyka

Přenositelnost

Program napsaný v Javě je přenositelný mezi nejrůznějšími architekturami (operačními systémy), jak na úrovni zdrojového kódu, tak i na úrovni přeloženého kódu. Jedinou podmínku přenositelnosti je existence JVM (Java Virtual Machine) pro danou architekturu.

Objektová orientace

Kromě několika primitivních datových typů je vše objektem.

Robustnost a bezpečnost

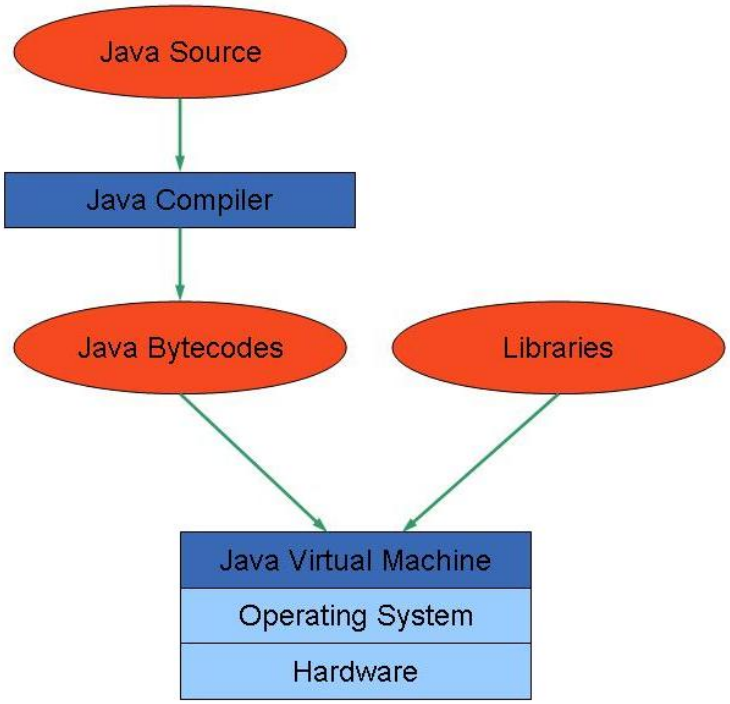
Neumožňuje některé konstrukce, které bývají častým zdrojem chyb. Poskytuje mechanismus výjimek, které slouží k ošetření chybových stavů programu. Překladač si vynucuje jejich ošetření na místech, která mohou být potenciálně nebezpečná (Např.: čtení ze souboru). Správu paměti realizuje pomocí Garbage collectoru, který automaticky odstraňuje nepotřebné objekty z paměti.

Jednoduchost

Syntaxe vychází s jazyků C a C++, přičemž z nich odpadly některé konstrukce způsobující problémy.

Provádění programu

Namísto překladu do strojového kódu, jako je tomu například u jazyků C, C++ a dalších, je program v Javě překládán do tzv. mezikódu (bytecode). Tento kód je poté interpretován virtuálním strojem JVM. Nevýhodou interpretace je ovšem pomalý běh aplikace. Proto novější JVM obsahují technologie JIT (Just in Time) kompilátor, který za běhu překládá bloky programu do nativního kódu. V současnosti většina kompilátorů používá technologii zvanou HotSpot kompilace, která převádí často používané bloky programu do nativního kódu a provádí optimalizace.



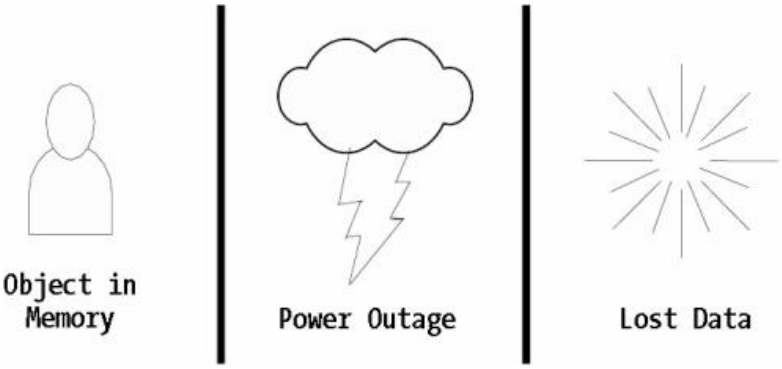
obrázek 2.1

Kompilace a provedení programu

2.2 Způsoby perzistence objektů

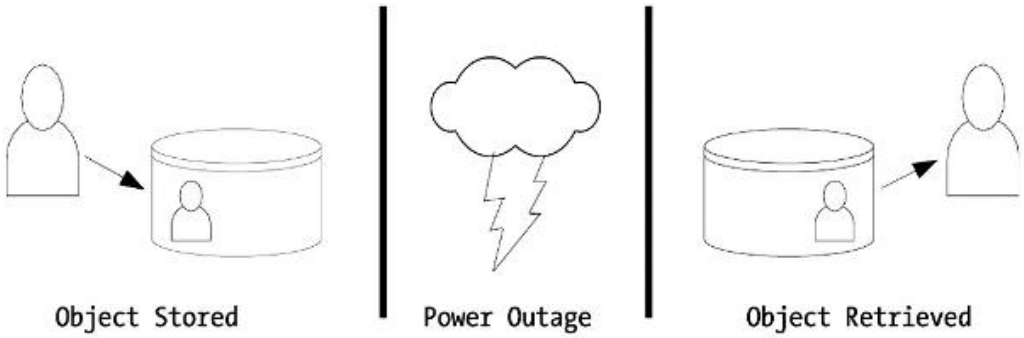
Pokud je spuštěna aplikace, tak objekty se kterými pracuje jsou uloženy v paměti. Nespornou výhodou paměti je rychlost přístupu k datům, která je o několik řádů vyšší než rychlost přístupu k datům, které jsou uloženy na pevném disku. Oproti tomu je však velkou nevýhodou nestálost takto umístěných dat. Jak je obecně známé, tak klasické paměti jsou elektricky závislé a při přerušení jejich napájení (ať již výpadkem napájení, nebo vypnutím počítače), jsou všechna data z nich ztracena.

Z tohoto důvodu vyvstává potřeba perzistence objektů. Perzistencí je míněno trvalé uložení objektů. Způsobů jakým lze objekt Javy uložit je poměrně mnoho. Lze jej uložit na disk pomocí XML, pomocí Java serializace nebo ve vlastním formátu. Tyto metody, zde budou zmíněny pouze okrajově a pro úplnost. Další možností uchování dat je jejich uložení do databáze. Tento způsob uchování dat je dnes velmi používaný a jazyk Java společně s nejrůznějšími rámci pro perzistenci tuto činnost velice usnadňuje. Tomuto způsobu perzistence objektů se bude v dalších kapitolách věnovat převážná většina této práce.



obrázek 2.2

Ztráta objektu při výpadku napájení (Převzato z [2])



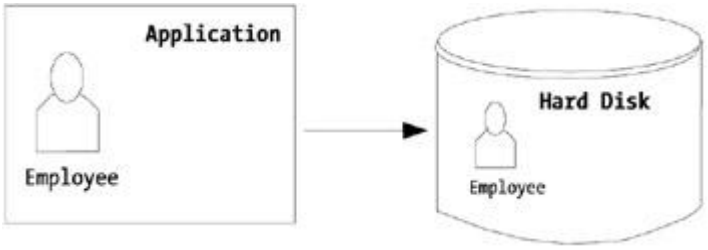
obrázek 2.3

Bezpečné uložení objektu na disk a jeho obnova po výpadku (Převzato z [2])

2.3 Ukládání objektů do souborů

Ukládání objektů do souborů lze využít zejména pro jednoduché aplikace, ve kterých nejsou data příliš rozsáhlá, nebo v aplikacích, ve kterých je ukládání dat souboru logické už z jejich účelu (např.: textový nebo grafický editor). Pokud bychom se rozhodli tímto způsobem ukládat rozsáhlejší data, setkali bychom se s mnoha problémy. Tím největším by byla ztráta výkonu aplikace, ke které by

došlo z důvodů nedostatečné podpory pro organizaci takto uložených dat. Příčina těchto problémů je velmi jednoduchá a vyplývá ze způsobu práce se soubory. Pokud zapisujeme data na konec souboru, vše je v pořádku a probíhá rychle. Problém nastává, pokud budeme chtít zapsat nebo odstranit objekt uprostřed souboru. Takováto změna uprostřed souboru vyžaduje posun všech dat, které se nachází v souboru za místem změny. Dalším omezením při takovéto práci je nemožnost zápisu z více procesů současně. Zpravidla není možné otevřít soubor pro zápis více procesy najednou.



obrázek 2.4

Uložení objektu do souboru (Převzato z [2])

V následující části budou stručně popsány dvě nejčastěji používané metody perzistence objektů do souborů. První je perzistence pomocí Java serializace, která je součástí jazyka Java od verze 1.0. Druhým způsobem je pak ukládání pomocí XML, který se v současnosti stává rozšířenější.

2.3.1 Perzistence pomocí Java serializace

Serializace je jedním z nejjednodušších způsobů, jak učinit objekt perzistentním. Jak již bylo uvedeno výše, jedná se o mechanizmus, který je přímo vestavěný do jazyka Java. Serializace umožňuje uložit jakýkoliv objekt do souboru. Objekt, který je takto uložen může být později opětovně ze souboru obnoven (deserializace). Nutnou podmínkou k tomu, aby serializace fungovala, je implementace rozhraní `Serializable` třídou, jejíž objekty chceme ukládat.

Ukládání objektů tímto způsobem je velmi jednoduché. Objekt, který má být uložen, je vložen do proudu dat a tento proud dat je následně zapsán jako soubor na disk. Objekty, které tento proces zajišťují jsou `ObjectInputStream` a `ObjectOutputStream`. Tyto objekty mají množství užitečných metod, ale nejdůležitějšími pro serializaci jsou `readObject()` a `writeObject()`.

Pokud programátor potřebuje ukládat data v jiném formátu, než s jakým tyto metody standardně pracují, má možnost je překrýt a řídit si tak proces serializace sám.

Serializace je někdy označována také jako „*lehká*“ perzistence. Důvodem, k přívlastku lehká je nemožnost definovat jednoduše u objektu, že má být perzistentní, například nějakým klíčovým slovem a nechat systém vyřešit detaily. Namísto toho musí programátor explicitně serializovat a deserializovat objekt ve svém programu.

Nevýhody serializace

Serializace přestane fungovat, pokud dojde ke změně třídy. Vezměme jednoduchý příklad třídy *student*, který má pouze atributy *jméno* a *příjmení*. Pokud máme nějaké objekty této třídy serializované a dojde k její změně (například přidání atributu pro *rodné číslo*), není již možné obnovit tyto objekty, přestože všechny uložené atributy jsou součástí nové verze třídy. Pokud se pokusíme provést jejich deserializaci, dojde k vyhození výjimky `InvalidClassException`. Důvodem k tomuto chování je skutečnost, že všechny třídy mají svoje sériové číslo. Toto číslo je ukládáno i do serializovaného objektu. Tím, že jsme do třídy přidali nový atribut, došlo ke změně tohoto čísla a při deserializaci původních objektů čísla nesouhlasí. JVM tuto skutečnost zjistí a vyhodí výjimku. Tento problém lze řešit explicitním nastavováním sériového čísla.

Dalším problémem serializace je neexistence vlastního způsobu vyhledávání v uložených objektech. Pokud budeme hledat jeden konkrétní objekt nějaké třídy, musíme deserializovat všechny objekty této třídy a poté v nich onen konkrétní objekt najít. Pokud budeme mít takovýchto objektů tisíce, může to způsobovat značné zpomalení aplikace.

Dalším problém serializace je neexistence nějakého mechanismu, jakým jsou transakce u databázových systémů. V praxi to znamená, že pokud dojde k nějakému problému během procesu serializace není zajištěno, že se buď provede vše nebo nic (atomičnost).

Posledním zde zmíněným problémem se serializací, je nemožnost získat pomocí deserializace pouze část atributů objektu. Lze jej tedy obnovit pouze celý nebo vůbec.

Výhody serializace

Největší výhoda serializace spočívá v tom, že s její pomocí lze překonávat rozdíly mezi různými architekturami. V praxi to znamená, že je možné serializovat nějaký objekt na určité systému (Např.: Windows), přenést jej po sítí a deserializaci provést na jakékoliv jiné architektuře (Unix, Solaris, atd.). Naprosto tím odpadají problémy s rozdílnou reprezentací dat na, pořadím bytů a dalšími detaily na jednotlivých architekturách.

2.3.2 Perzistence objektů do XML

XML (eXtensible Markup Language) je velmi jednoduchý značkovací jazyk, který lze využít pro popis dat. Jeho velkou výhodou je dobrá čitelnost pro člověka a existence mnoha různých nástrojů, které jeho čtení ještě usnadňují.

Pro perzistence objektů do XML existuje poměrně velké množství volně dostupných nástrojů. Jedním z nejpopulárnějších je Castor od firmy Exolab, který umožňuje převádět objekty Javy do XML velice jednoduchým způsobem. Nástroj Castor je dostupný na http://www.castor.exolab.org

Flexibilita XML pomáhá překonávat některé problémy spojené se serializací. Při používání XML není problémem růst a změna dat. Pokud aplikace, která data čte, má jejich správný popis, není

problém obnovit objekt, pokud došlo k nějaké změně (Např.: Po uložení do XML došlo u dané třídy k přidání nějakého atributu).

Pokud dojde k opačnému případu (objekt uložený v XML obsahuje atributy, které byly v nové verzi třídy odstraněny), opět nedojde k žádným problémům při jeho obnově. Atributy, které nejsou třídou podporovány, jsou při obnově automaticky ignorovány. Tyto dvě skutečnosti poskytují značnou robustnost proti serializaci.

Přestože perzistence do XML překonává některé problémy serializace, není zdaleka ideálním řešením. Důvodem je, že se stále jedná o práci se soubory, při které se setkáváme s problémy při vyhledávání, neexistencí transakcí, nemožností zápisů z více procesů ve stejném okamžiku a mnoha dalšími.

V této kapitole byly popsány principy a základní myšlenky perzistence objektů jazyka Java. Jelikož je tato práce primárně zaměřena na perzistenci objektů do databází (zejména relačních), nebudu se již podrobněji zmiňovat o metodách popsaných v této kapitole.

3 JDBC

Java Database Connectivity (JDBC) je aplikační rozhraní (API), které umožňuje aplikacím jazyka Java přistupovat k datům, která jsou strukturována do tabulek. Primárním zdrojem dat, ke kterému pomocí JDBC přistupujeme, jsou data v relační databázi. Mezi další možné zdroje dat, se kterými můžeme pracovat, patří například textové soubory, data vytvořená v tabulkovém procesoru a další. Těmito zdroji dat se však v této práci zabývat nebudu. Primární zdroje, ze kterých jsem čerpal informace pro psaní této kapitoly, jsou uvedeny v literatuře pod body [2] a [4].

JDBC definuje tzv. low-level API navržené pro podporu základní funkcionality. Vztah JDBC a databáze je možné přirovnat ke vztahu mezi rozhraním a třídou. Díky tomu nabízí jednoduchý a konzistentní způsob práce s relační databází nezávisle na jejím typu.

Současná verze Javy (Java SDK 1.5) obsahuje všechny potřebné knihovny pro JDBC verze 3. Dále je v ní obsažen jeden JDBC driver, který slouží jako most k ostatním ODBC driverům nainstalovaným v systému. Pro připojení ke konkrétnímu databázovému systému může použít, buď tento JDBC-ODBC bridge, anebo konkrétní JDBC driver pro daný databázový systém, což je častější a používanější způsob. Konkrétní JDBC driver lze získat v podstatě pro jakýkoliv databázový systém. Například pro MySQL lze stáhnout Connector/J ze stránek výrobce.

3.1 Typy driverů

Velkou výhodou JDBC je snadná možnost vyměnit driver podle toho, který nejlépe vyhovuje pro daný účel. Některé podporují transakce, jiné mají vestavěný connection-pooling, jiné jsou volně dostupné atd.

Celkově je těchto driverů velmi velký počet, ale obecně je možné jejich rozdělení do 4 skupin:

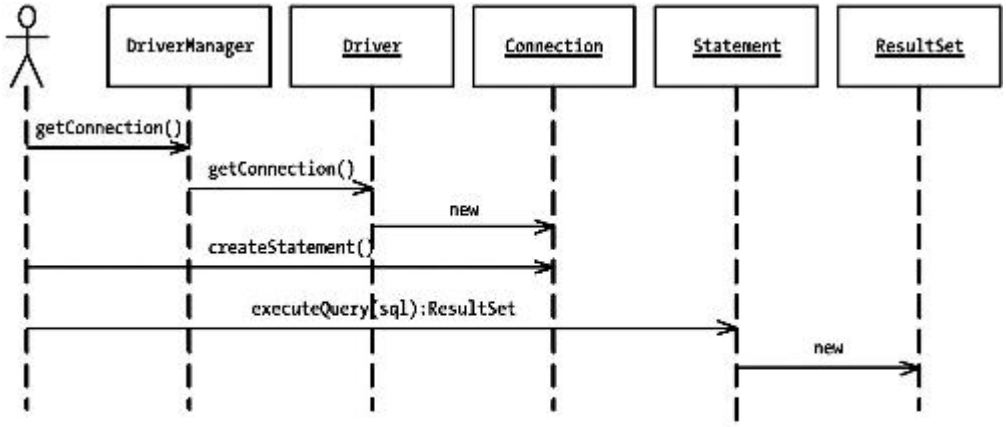
1) **JDBC-ODBC bridge** – vyžaduje existenci nativního ODBC driveru nainstalovaného v systému
2) **Native API partly Java driver** – vyžaduje nainstalování nativního databázového klienta
3) **Net protocol Java drive**r – konvertuje JDBC volání do nezávislého protokolu, který je poté přeložen serverem do databázového protokolu.
4) **Native protocol Java driver** – umožňuje přímé volání mezi klientem a DB serverem. Díky tomu, že je napsaný přímo v Javě, nevyžaduje žádnou speciální konfiguraci klientského stroje. Stačí pouze sdělit aplikaci, kde daný driver najde.

3.2 Použití JDBC

Základní použití JDBC je velmi jednoduché. Pomocí několika tříd a rozhraní lze dosáhnout veškeré základní funkcionality. Základem pro práci je třída `java.util.DataManager` a rozhraní `java.sql.Connection`, `java.sql.Statement` a `java.sql.ResultSet`.

Použití sestává z několika kroků:

1) Import potřebných tříd.
2) Načtení JDBC driveru.
3) Identifikace zdroje dat.
4) Alokace objektu `Connection`.
5) Alokace objektu `Statement`.
6) Provedení dotazu pomocí `Statement` objektu.
7) Získání data z vráceného objektu `ResultSet`.
8) Uzavření `ResultSetu`.
9) Uzavření objektů `Statement` a `Connection`.



obrázek 3.1

Práce s třídou a rozhraními JDBC (Převzato z [2])

3.2.1 Načtení databázového driveru

Před tím, než může vůbec načíst driver, je třeba korektně nastavit proměnnou `classpath`. Pokud driver umístíme například do adresáře `lib`, lze tuto činnost provést elegantně pomocí nástroje Ant. Následuje fragment skriptu Antu, který zajistí vše potřebné:

```
<path id="compile.classpath">
      <fileset dir="${lib.dir}">
```

```
            <include name="**/*.jar" />
        </fileset>
    </path>
```

Samotnou kompilaci a spuštění poté provedeme následovně:

```
    <javac srcdir="${source.dir}" destdir="${classes.dir}" >
        <classpath refid="compile.classpath" />
    </javac>
    <java classname="MyClass">
        <classpath refid="compile.classpath" />
    </java>
```

Pokud vše korektně proběhne, můžeme přistoupit k samotnému načtení driveru. To provedeme pomocí statické metody `forName` třídy `Class`. Jako parametr této metodě předáme jméno třídy, kterou si přejeme načíst. Pro MySQL driver bude tedy použití vypadat takto:

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

3.2.2 Získání spojení do databáze

Spojení do databáze získáme pomocí metody `getConnection`, třídy `DriverManager`. Jako parametry jí předáme URL databázového serveru (spolu s jménem databáze), jméno a heslo. Konkrétní příklad pro databázi *testDB*, uživatele *user* a heslo *123* vypadá takto:

```
connection = DriverManager.getConnection(
            "jdbc:mysql://localhost/testDB", "user", "123");
```

3.2.3 Použití objektu Statement

Pokud jsme se úspěšně připojili k databázi, můžeme s ní začít komunikovat pomocí objektu `Statement`. Objekt samotný je součást JDBC driveru, implementující rozhraní `Statement`. Získáme jej pomocí voláním metody `createStatement` objektu `Connection`. Tento objekt obsahuje řadu metod. Pro povedení SQL dotazu lze použít metodu `executeUpdate`, které bude mít jako parametr SQL dotaz. Tato metoda je vhodná zejména pro SQL dotazy `INSERT, UPDATE, DELETE` a dotazy z podmnožiny DDL jazyka SQL:

```
statement = connection.createStatement();
String sqlDotaz = "CREATE DATABASE pokus";
statement.executeUpdate(sqlDotaz);
```

3.2.4 Práce s objektem ResultSet

Objekt `ResultSet` po provedení SQL dotazu obsahuje jeho výsledek. Tato výsledná množina se získá provedením metody `executeQuery` objektu `Statement`. Stejně jako v případě metody `executeUpdate` je jejím parametrem dotaz v jazyce SQL. Samotný `ResultSet` lze poté procházet pomocí kurzoru. Rozhraní `ResultSet` obsahuje mnoho nejrůznějších metod. Pro procházení výsledku lze s výhodou použít metodu `next` a cyklu `while`:

```
while (resultSet.next())  {
      // práce s daty, která jsou uložena v jednotlivých
      // sloupcích výsledné tabulky
}
```

Tímto způsobem lze řádek po řádku projít celou výslednou tabulkou, kterou jsme získali SQL dotazem.Pro přístup k jednotlivým sloupců tabulky potřebuje znát jejich jméno (nebo případně jejich pozici ve výsledné tabulce) a datový typ. Pokud máme ve výsledné tabulce sloupec pojmenovaný *mesto*, typu `varchar()`, můžeme data, které jsou v něm uložena získat pomocí:

```
resultSet.getString("mesto");
```

V praxi je však výhodnější použití objektu `ResultSetMetaData`, který nám umožňuje zjisti vše potřebné o výsledné tabulce. Zejména počet, jména a datové typy sloupců.

Použití všech výše uvedených metod, může vést k vyhození některé z výjimek. Proto je potřeba, aby byly uzavřeny do `try-catch` bloků. Na závěr je třeba všechny použité objekty uvolnit. Tato činnost by měla být prováděna v bloku `finally`. Samotné uvolnění může vyhodit výjimku, takže musí být opět prováděno v `try-catch` bloku. Následující příklad ilustruje uvolnění objektu `ResultSet` :

```
finally {
      if (resultSet != null) {
            try {
                  reseultSet.close();
            } catch (SQLException e)  {
                  e.printStackTrace();
            }
      }
      // dále by následovalo uvolnění  objektů Statement
      // a Connection
}
```

3.3 Optimalizace výkonu

Přístup k datům v databázi bývá často tou nejpomalejší částí aplikace. Je proto vhodné zamyslet se na tím, zda by jej nebylo možné nějakým způsobem optimalizovat. Pokud provádíme často stejný SQL dotaz, lze s výhodou využít předpřipravený dotaz `PreparedStatement`. Toto rozhraní je specializací rozhraní `Statement`. Pokud databázový systém zpracovává SQL dotaz, musí jej vždy zkompilovat a poté teprve provést. Výhoda `PreparedStatement` spočívá v tom, že dotaz je zkompilován pouze jednou.

Pokud máme v naší aplikaci například nějaký často používaný formulář pro vyhledávání podle různých kritérií, je pro tento dotaz velmi vhodné použití `PreparedStatement`. Hodnoty, které budou do formuláře zadávány, jsou v SQL dotazu nahrazeny znaky "?". Takový dotaz by mohl vypadat nějak následovně:

```
SELECT * FROM filmy WHERE rok_vydani = ? AND kategorie = ?
```

Konkrétní hodnoty argumentů doplníme pomocí volání metody `setType(index, value)`. Jméno této metody závisí na datovém typu argumentu, který si přejeme nastavit. Pro naplnění argumentů předcházející SQL dotazu lze použít:

```
preparedStatement.setInt(1,1996);
preparedStatement.setString(2, "Akcni");
```

Je důležité poznamenat, že argumenty jsou číslovány od 1 a ne od 0, jak bývá často zvykem.

3.4 Pokročilejší možnosti JDBC

Všechna výše uvedená rozhraní a práce s nimi byla již součástí JDBC verze 1. Pro základní práci s databází jsou samozřejmě plně dostačující. Postupem času ale v nových verzích JDBC přibylo mnoho zajímavých u užitečných věcí, z nichž zde některé krátce zmíním.

3.4.1 Posunovatelný a aktualizovatelný ResultSet

V původní verzi JDBC bylo možno `ResultSet` procházet pouze od začátku do konce, což s sebou často přinášelo potřebu jej celý opětovně procházet. Ve verzi JDBC 2.1 je možné se v rámci objektu `ResultSet` posouvat dopředu, dozadu a skokem se přesunout na určitou jeho položku. Další novinkou je možnost měnit data (aktuálně zpracovávané položky), přidávat a odebírat záznamy. Všechny provedené změny jsou samozřejmě promítnuty do databáze. Následující příklad ilustruje vytvoření objektu `ResultSet`, který je posunovatelný oběma směry a aktualizovatelný:

```
statement = connection.createStatement(
```

```
            ResultSet.TYPE_SCROLL_INSENSITIVE
            ResultSet.CONCUR_UPDATABLE);
      resultSet = statement.executeQuery("SQL dotaz");
```

3.4.2 Dávkové zpracování dotazů

Další významnou novinkou v JDBC 2 je možnost seskupit několik SQL příkazů a provést je společně. Tímto lze pozitivně ovlivnit zatížení sítě a komunikaci s databázovým serverem, což může ve výsledku vést k celkovému zlepšení výkonu. Následující příklad provede najednou příkazy `UPDATE` a `SELECT`:

```
      String upadate1 = "SQL dotaz update1";
      statement.addBatch(update1);

      String select1 = "SQL dotaz select1";
      statement.addBatch(select1);

      int queries[] = statement.executeBatch()
```

V rámci jedné dávky lze míchat různé dotazy SQL, jako `SELECT`, `UPDATE` apod. K otestování, jak jednotlivé dotazy dopadly, stačí jednoduše projít výsledné celočíselné pole.

3.4.3 Dávkové zpracování předpřipravených dotazů

Dávkové zpracování předpřipraveného dotazu je velmi podobné jako u normálního dotazu. Jediný významný rozdíl je v tom, že SQL dotazy musejí být homogenní. Nelze tedy míchat v rámci jedné dávky různé druhy dotazů. Aktualizace emailů u více záznamů v tabulce `zamestnanec` najednou by mohlo vypadat nějak takto:

```
String sql = "UPDATE zamestnanec SET email = ? WHERE idzam = ?";
      statement = connection.prepareStatement(sql);

      statement.setString(1,"nejaky@email.cz");
      statement.setLong(2, 1);
      statement.addBatch();

      statement.setString(1,"dalsi@email.cz");
      statement.setLong(2, 2);
      statement.addBatch();
      statement.executeBatch();
```

3.5 JDBC 3.0 a novinky připravované verze 4.0

JDBC verze 3.0 přišlo s velkou řadou novinek, jejichž kompletní popis zde není možný. Zaměříme se proto pouze na ty nejdůležitější, z nichž některé budou použity v ukázkové aplikaci

3.5.1 Rozhraní MetaData

Rozhraní `DatabaseMetaData` umožňuje získat informace o hierarchickém uspořádání tabulek v databázi a informace o uživatelsky definovaných typech. K tomuto účelu lze použít metody `getSuperTables()` a `getSuperTypes()`.

Dále se zde objevuje nové rozhraní `ParameterMetaData`, které se používá pro získávání informací o parametrech v `PreparedStatement` objektech.

3.5.2 Zjišťování automaticky generovaných klíčů

JDBC 3.0 umožňuje zjišťování hodnot automaticky generovaných klíčů. Stačí pouze nastavit příznak u metody `execute()` objektu `Statement`. Samotný klíč se získá pomocí metody `getGeneratedKeys()`, která vrátí `ResultSet` obsahující řádek pro každý generovaný klíč. Příznaky pro metodu `execute()` jsou následující:

1) NO_GENERATED_KEYS
2) RETURN_GENERATED_KEYS

3.5.3 Savepointy v transakcích

`Savepoint` umožňuje jemnější způsob kontroly nad transakcí. Někdy může být jedna transakce poměrně složitá a nemusí být žádoucí použití `ROLLBACK`, který ji celou vrátí zpět. Definice `Savepointu` nabízí možnost rozdělení transakce na logické celky a případný `Rollback` k některému z definovaných `Savepointů`. Je nutné poznamenat, že budou pracovat, pouze pokud máme vypnutý (false) auto-commit pomocí metody `setAutoCommit()` objektu `Connection`.

3.5.4 JDBC 4.0

Verze JDBC 4.0 bude součástí nově připravované Javy 6.0. Jeho nejzajímavějšími novinkami by mělo být:

1) **Automatické nahrávání JDBC driveru** – pomocí nadeklarování třídy v jar manifestu.
2) **Connection management** – podpora správy `Connection` a `Statement` objektů.
3) **Vylepšení odchytávání výjimek** – nové třídy a od nich odvozené hierarchie.
4) **Změna API** – podpora pro nové datové typy.
5) **Rozhraní RowId** – jednoznačná identifikace záznamu v databázi.

4 Hibernate

Tato kapitola diplomové práce se bude poměrně podrobně zabývat v současné době nejrozšířenějším volně dostupným rámcem pro objektově-relační mapování (ORM), rámcem Hibernate.

Přestože existuje i port NHibernate pro platformu .NET, je rámec Hibernate primárně vyvíjen pro použití v jazyce Java. V Javě lze Hibernate použít jak v klasických (standalone) J2SE aplikacích, tak samozřejmě v aplikacích J2EE .

Aplikace v prostředí J2EE zpravidla obsahují nejrůznější nástroje a knihovny třetích stran (third-party), se kterými musí Hibernate korektně spolupracovat. Příkladem může být nástroj pro sestavování aplikací Ant, rámec pro testování JUnit, rámce Struts, Spring a mnoho dalších. Popis těchto nejrůznějších spojitostí a souvislostí přesahuje rámec této práce.

Kromě samotného jádra systému, které zajišťuje ORM, obsahuje Hibernate užitečné nástroje pro podporu softwarového inženýrství a vývoje aplikací podle rámce MDA (Model Driven Architecture). Tyto nástroje a jejich základní možnosti použití budou krátce zmíněny na konci této kapitoly. Převážnou většinu informací pro tuto kapitolu jsem čerpal z oficiálních dokumentací Hibernate, které jsou uvedeny v literatuře pod body [6] a [7] a dále z [8].

4.1 Architektura

Rámec Hibernate je velmi flexibilní a umožňuje několik různých přístupů pro jeho použití při vývoji aplikací. Existují však dva extrémní způsoby jeho použití.

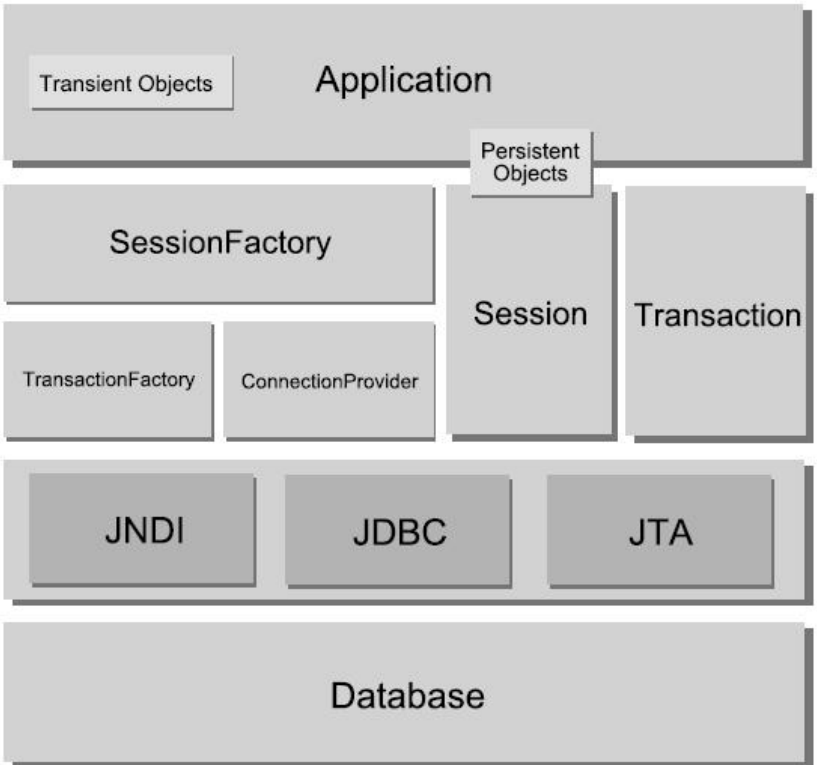
Prvním z nich je tzv. “odlehčená“ architektura, kdy aplikace poskytuje vlastní JDBC spojení a spravuje vlastní transakce.

Druhým extrémem je maximální využití možností rámce, kdy je aplikace abstrahována od vrstvy s JDBC a JTA a nechává na něm, aby se o vše potřebné postaral sám. Tento způsob použití je zobrazen na obrázku 4.1 na další stránce.

Význam objektů z obrázku 4.1 je následující:

- **`SessionFactory`** – poskytuje cache, ve které je uložena konfigurace objektově-relačního mapování.
- **`Session`** – velmi důležitý objekt, který reprezentuje komunikaci mezi aplikací a perzistentním úložištěm, obaluje JDBC spojení a slouží pro vytváření objektů `Transaction`. Dále slouží jako cache pro perzistentní objekty.
- **`Perzistentní objekty`** – jsou objekty, které jsou spojeny s aktuální `Session`. Po ukončení této `Session` se z nich stanou „odpojené“ objekty a mohou být dále použity v některé aplikační vrstvě.

- **`Tranzientní a odpojené objekty`** – instance perzistentních tříd, které nejsou aktuálně spojeny s žádnou `Session`.
- **`Transaction`** – objekt využívaný aplikací pro ohraničení atomické jednotky práce (transakce). Abstrahuje také aplikaci od JDBC, JTA transakcí. Ve většině případů jedna `Session` obsahuje více transakcí `Transaction`.
- **`ConnectionProvider`** – Objekt sloužící k vytvoření JDBC spojení. Abstrahuje aplikaci od použitého zdroje pro získání JDBC spojení.
- **`TransactionFactory`** – Objekt „továrna“ určený k vytváření objektů `Transaction`



obrázek 4.1

Architektura Hibernate (Převzato z [6])

4.2 Konfigurace Hibernate

Jak již bylo výše zmíněno, Hibernate musí koexistovat v různých prostředích od aplikačních serverů po standalone aplikace. Aplikační server je příkladem řízeného prostředí, které poskytuje služby hostujícím aplikacím. Příkladem služeb poskytovaným aplikačním serverem může být connection-pooling a řízení transakcí. V současné době patří mezi nejpopulárnější servery JBoss a WebSphere. Druhou alternativou je neřízené prostředí, které zpravidla tyto služby postrádá. Příkladem takovéto

aplikace mohou být klasické Swing nebo SWT aplikace. Z těchto důvodů podporuje Hibernate velké množství konfiguračních metod a nastavení pro podporu různých scénářů.

První možností konfigurace Hibernate je konfigurace přímo v programu pomocí funkcí, které poskytuje API. Toto není zrovna nejlepší způsobem konfigurace a proto se mu dále nebudu věnovat.

Mnohem vhodnějším způsobem jsou konfigurační soubory. U tohoto způsobu existují dvě alternativy:

1) `hibernate.properties` – standardní Java properties soubor.
2) `hibernate.cfg.xml` – konfigurační soubor v jazyce XML.

Osobně preferuji konfiguraci pomocí XML souboru, která je i obecně používanější (důvod bude zřejmě spočívat ve velkém množství volně dostupných programů usnadňujících jejich tvorbu). Je však důležité zdůraznit, že oba soubory mají stejný význam, kterým je konfigurace Hibernate. Pokud jsou použity oba dva, potom konfigurace v `hibernate.cfg.xml` přepisují stejné vlastnosti nastavené v `hibernate.properties`. Tyto konfigurační soubory je třeba umístit do kořene `CLASSPATH`, kde je Hibernate automaticky hledá při spuštění.

4.2.1 Neřízené prostředí

Následující fragment souboru `hibernate.cfg.xml` je typickou ukázkou konfigurace pro neřízené prostředí.

```
<hibernate-configuration>
      <session-factory>
            <property name="connection.username">user</property>
            <property name="connection.password">pwd</property>
            <property name="connection.url">
                  jdbc:mysql://localhost/dbName
            </property>
            <property name="connection.driver_class">
                  com.mysql.jdbc.Driver
            </property>
            <property name="dialect">
                  org.hibernate.dialect. MySQL5InnoDBDialect
            </property>
            <mapping resource="package/Class.hbm.xml" />
            <mapping class="package.Class" />
      </session-factory>
</hibernate-configuration>
```

Pro použití JDBC spojení je důležitých prvních pět elementů, kde je nastaveno uživatelské jméno, heslo, adresa DB serveru (včetně databáze, se kterou se bude pracovat), třída driveru pro konkrétní DB server a dialekt. Mohlo by se zdát, že je zbytečné nastavovat dialekt, jakým bude Hibernate s DB serverem komunikovat, když je už typ DB serveru nastaven pomocí driveru. Je to však důležité z toho důvodu, že DB servery poskytují různé typy úložišť (např.: MySQL nabízí InnoDB, MyISAM a další typy tabulek), z nichž každé se hodí pro něco jiného a něco jiného poskytují (kontrola integritních omezení, kaskádní mazání atd.).

Pokud tedy používáme `InnoDB` tabulky u MySQL, je třeba dialekt nastavit na `org.hibernate.dialect.MySQL5InnoDBDialect` . Celkem Hibernate pracuje s více než 20-ti dialekty všech významných DB serverů. Jejich kompletní seznam lze samozřejmě nalézt v API dokumentaci.

Poslední dva elementy se týkají umístění popisu perzistentní třídy. První z nich používá popisu umístěného do zvláštního XML dokumentu a druhý novinku JDK 5.0 – umístění popisu přímo do třídy pomocí anotací. Oba tyto způsoby budou zmíněny později. Vlastností, které je možné v tomto souboru nastavit, jsou desítky. Lze je samozřejmě opět dohledat v API dokumentaci.

Hibernate dále umožňuje použít vestavěný connection-pooling . Obecně se doporučuje toto řešení používat pouze při vývoji a při nasazení aplikace do provozu se doporučuje pro zvýšení výkonu použít řešení nějaké třetí strany.

4.2.2 Řízené prostředí

Při použití Hibernate v rámci nějakého aplikačního serveru je vhodné nakonfigurovat Hibernate tak, aby požíval spojení z datového zdroje daného aplikačního serveru dostupné přes JNDI. Následuje fragment souboru `hibernate.cfg.xml`, který se o dané nastavení postará:

```
<hibernate-configuration>
      <session-factory
            name="java:comp/env/hibernate/SessionFactory">
            <property name="connection.datasource">
                  jdbc/myDataSource
            </property>
            <property name="dialect">
                  org.hibernate.dialect.MySQLDialect
            </property>
            ......
      </session-factory>
</hibernate-configuration
```

4.3 Perzistentní třídy

Perzistentní třídy jsou třídy aplikace, které reprezentují entity modelované pro danou aplikační oblast. Obecně je vhodné (ne však nezbytné), aby tyto třídy byly vytvořeny podle modelu POJO (Plain Old Java Object). POJO je obyčejnou třídou jazyka Java, která obsahuje pouze atributy, `get/set` metody pro práci s nimi a případně další základní metody (`equals`, `hashCode`, apod.).

Dalším požadavkem, který musí tyto třídy splňovat, je implementace bezparametrického konstruktoru. Stejně jako u `get/set` metod není u tohoto konstruktoru striktně vyžadován modifikátor přístupu `public`, ale je vhodný pro zvýšení výkonu. Další vlastností, kterou by měly splňovat perzistentní třídy pro zvýšení výkonnosti, je definice atributu, který bude použit jako primární klíč u odpovídající tabulky v databázi.

Instance těchto tříd se mohou nacházet vzhledem k perzistentnímu kontextu ve třech různých stavech:

1) **Dočasný** – objekt aplikace, který ještě nebyl uložen a nemá žádný vztah k perzistentnímu kontextu.
2) **Perzistentní** – objekt spojení s perzistentním kontextem. Je jednoznačně identifikovatelný primárním klíčem a odpovídá řádku v příslušné databázi. Pokud s tímto objektem pracujeme, Hibernate zaznamenává všechny jeho změny a při potvrzení transakce provede synchronizaci provedených změn s databází.
3) **Odpojený** – objekt, který byl spojený s nějakým v minulosti uzavřeným perzistentním kontextem. Odkaz na tento objekt však stále existuje a s objektem lze normálně pracovat. Tento objekt lze učinit perzistentním tak, že jej připojíme k perzistentnímu kontextu.

Následuje příklad perzistentní třídy `Transaction`:

```
@Entity
@Table(name="tbl_transaction")
public class Transaction implements Serializable{
      private long id;
      private Account account;
      private int amount;

      public Transaction() {}

      @Id @GeneratedValue
      public long getId() {
            return id;
      }
```

```
	public void setId(long id) {
		this.id = id;
	}
	@ManyToOne
	@OnDelete(action = OnDeleteAction.CASCADE)
	public Account getAccount() {
		return account;
	}

	public void setAccount(Account account) {
		this.account = account;
	}

	@Column(name="amount", nullable=false, unique=false)
	public int getAmount() {
		return amount;
	}

	public void setAmount(int amount) {
		this.amount = amount;
	}
}
```

Tento příklad ilustruje další dvě zajímavé věci. První z nich je použití anotací (v tomto případě umístěných u `get` metod) a druhou vztah s kardinalitou `ManyToOne` mezi třídami `Transaction` a `Account`. Obojí bude podrobněji vysvětleno v následující kapitole.

4.4 Mapování objektů na tabulky databáze

Aby mohlo Hibernate provádět převody mezi objektovým a relačním modelem dat, potřebuje vědět, jak se instance jednotlivých tříd budou mapovat na tabulky v databázi. Způsoby, jakými mu to lze sdělit, jsou dva. Prvním z nich je použití mapovacího XML souboru a druhým je použití anotací v kódu dané třídy.

4.4.1 Mapování pomocí XML souboru

Při používání mapovacího souboru máme dvě možnosti. První z nich je vytvořit jeden mapovací soubor pro všechny perzistentní třídy. Tento způsob se nevyužívá kvůli značné nepřehlednosti a proto se jím nebudu dále zabývat. Druhým způsobem je vytvoření mapovacího souboru pro každou perzistentní třídu aplikace zvlášť. Konvence pro pojmenování tohoto souboru je `JmenoTridy.hmb.xml` (tedy pro příklad z předchozí kapitoly: `Transaction.hbm.xml`). Tento soubor je třeba umístit do adresáře, ve kterém se nachází soubor `Transaction.java`.

Následuje fragment mapovacího souboru pro třídu `Transaction` z minulé kapitoly:

```
<hibernate-mapping package="packageName">
      <class name="Transaction" table="tbl_transaction">

            <id name="id">
                  <generator class="native"/>
            </id>

            <many-to-one name="account" column="account_id"
             class="Account"/>

            <property name="amount" not-null="false"/>
      </class>
</hibernate-mapping>
```

Existuje mnoho nejrůznějších parametrů, které umožňují nastaveních různých aspektů mapování pro dosažení optimálního výkonu. Jejich přesný význam a popis lze opět dohledat v API dokumentaci.

Osobně si myslím, že je lepší používání anotací a proto se dále tomuto způsobu mapování nebudu v této práci věnovat.

4.4.2 Mapování pomocí anotací

Jak jsem již uvedl dříve, používaní anotací v kódu je umožněno od Javy verze 5. K tomu, aby anotace fungovaly, je třeba stáhnutí balíku *Hibernate Annotations*, protože nejsou součástí jádra Hibernate. Anotace jsou v kódu umisťovány před třídu (týkají se nastavení odpovídající tabulky) a dále buď ke `get` metodám jednotlivých atributů nebo přímo k těmto atributům.

Možnosti nastavení mapování pomocí anotací jsou opět velmi široké a nelze je proto zde podrobně rozepisovat. Kompletní seznam lze opět najít v API dokumentaci.

Nebudu zde již uvádět žádný nový příklad, neboť použití anotací je ilustrováno na třídě `Transaction` v kapitole 4.3, ale zaměřím se zde na dvě zajímavá nastavení v něm.

Prvním je anotace:

```
@Id @GeneratedValue
public long getId() { ... }
```

Tato anotace říká, že se jedná o primární klíč tabulky, o jehož generování se bude starat databázový server.

Druhou zajímavou anotací je:

```
@ManyToOne
@OnDelete(action = OnDeleteAction.CASCADE)
public Account getAccount() { ... }
```

Třída `Transaction` obsahuje jako jeden atribut objekt třídy `Account`. Mezi těmito třídami existuje vztah 1:M znázorněný na obrázku 4.2. Tato anotace říká, že sloupec odpovídající tabulky (na kterou se budou objekty třídy `Transaction` mapovat) bude cizím klíčem ukazujícím na primární klíč tabulky, na kterou se budou mapovat objekty třídy `Account`. To, že dojde v databázi k tomuto provázání tabulek, není třeba nijak speciálně nastavovat. Pokud požadujeme tuto standardní funkčnost, stačí prosté použití anotace `@ManyToOne`.

Druhá anotace `@OnDelete` nastavuje, jak se má databáze vypořádat s odstraněním objektu. Akce `CASCADE` je standardní ve většině DB serverů a znamená, že pokud dojde k odstranění určitého objektu `Account`, jsou odstraněny i všechny odpovídající objekty `Transaction`, které s ním byly spojeny.

obrázek 4.2

Vztah 1:M mezi objekty Transaction a Account

Vztah 1:M není samozřejmě jediným vztahem, který umí Hibernate mapovat. Podobným způsobem jsou mapovány i další vztahy, kterými jsou 1:1, M:N, polymorfizmus a dědičnost. U každého z těchto vztahů lze nastavit poměrně velké množství parametrů, které lze opět dohledat v API dokumentaci.

Poslední možnost, jak namapovat perzistentní třídy na databázi, tkví v použití anotací nástroje XDoclet. Tento nástroj využívá upravených komentářů k tvorbě dokumentů nutných pro běh aplikací.

Vzhledem k tomu, že od Javy verze 5 lze pro mapování přímo v kódu třídy použít anotací, pozbývá tato možnost své dřívější výhody a nebudu se jí proto dále věnovat.

4.5 Práce s objekty v aplikaci

Abychom mohli začít s Hibernate pracovat v aplikaci, potřebujeme získat objekt třídy `Session`, který reprezentuje připojení do databáze. K získání tohoto objektu slouží `SessionFactory`, což je rozhraní poskytující instance třídy `Session`. Na rozdíl od instancí třídy `Session` jsou instance třídy `SessionFactory` tzv. thread-safe a typicky sdílené v rámci celé aplikace. Oproti tomu instance `Session` by měly být používány pouze pro jednu transakci. Touto transakcí není myšlena klasická databázová transakce, ale tzv. *business transakce*, což je jakási logická jednotka práce, která provede konkrétní business úlohu. Zpravidla tato transakce sestává z více databázových transakcí.

4.5.1 Konfigurace SessionFactory

Pro vytvoření `SessionFactory` je třeba použít třídu `Configuration`, která načte mapovací soubory. Objekt třídy `Configuration` lze vytvořit následujícími třemi způsoby.

Prvním z nich je automatické načtení vlastností a mapovacích souborů nadefinovaných v souboru `hibernate.cfg.xml` a vytvoření `SessionFactory`:

```
Configuration cfg = new Configuration();
SessionFactory factory = cfg.configure().buildSessionFactory();
```

Druhou alternativou je načtení mapovacích souborů podle perzistentních tříd. Následující fragment kódu zajistí, že Hibernate bude hledat soubor Trida.hbm.xml v adresáři package:

```
Configuration cfg = new Configuration();
SessionFactory factory = cfg.addClass(package.Trida.class);
```

Třetí způsob je vhodný, pokud aplikace obsahuje velké množství perzistentních tříd a s tím spojené množství mapovacích souborů. V tomto případě můžeme vytvořit JAR soubor, který bude tyto třídy a mapovací soubory obsahovat. Následně stačí takto aktualizovat soubor `hibernate.cfg.xml`:

```
<mapping jar="soubor.jar"/>
```

Pokud nepoužíváme mapovací soubory k perzistentním třídám, ale anotace uvnitř těchto tříd, můžeme využít speciální verzi třídy `Configuration` a to `AnnotationConfiguration`.

V případě, že máme správně nakonfigurovanou `SessionFactory`, můžeme velmi jednoduchým způsobem získat objekt `Session`:

```
Session session = factory.openSession();
```

4.5.2 Ukládání objektů

Následující jednoduchý příklad ilustruje uložení objektu `Transaction` z kapitoly 4.3. Protože tento objekt v sobě obsahuje objekt `Account`, je třeba nejdříve tento objekt získat. K tomu slouží dotaz v jazyce HQL, který bude popsán v následující kapitole. Pokud jsme úspěšně získali tento objekt, zbývá již jen nastavit atributy objektu `Transaction` a danou transakci uložit.

```
Session session = null;
Transaction tx = null;

try{
      session = factory.openSession();
      tx = session.beginTransaction();

      Account account = (Account) session.createQuery(HQL);
      Transaction transaction = new Transaction();

      transaction.setAccount(account);
      transaction.setAmount(100);

      session.save(transaction);
      tx.commit();
} catch (HibernateException ex){
      if (tx != null) tx.rollback;
} finally {
      if (session != null) session.close();
}
```

Abychom mohli výše uvedené provést, je třeba získat perzistentní kontext, který reprezentuje objekt `session`. V rámci `session` zahájíme transakci, ve které budou provedeny samotné operace s objekty. Tato transakce je tzv. *business transakcí* zmíněnou v úvodu této kapitoly.

4.6 Dotazování nad perzistentními objekty

Abychom mohli v naší aplikaci pracovat s dříve uloženými objekty, je zapotřebí nějakého mechanismu, pomocí kterého tyto perzistentní objekty znovu získáme. Hibernate nám v tomto směru nabízí tři různé přístupy. Prvním z nich je použití dotazovacího jazyka HQL, druhým je použití vyhledávacích kritérií a třetím je dotazování pomocí jazyka SQL. Pokud použijeme tento třetí způsob, připravíme se tím zejména o možnost přenositelnosti aplikace mezi různými databázovými systémy. Měl by tedy být používán pouze ve výjimečných případech, kdy požadovanou funkčnost Hibernate nepodporuje. Z tohoto důvodu se v dalším textu tomuto způsobu dotazování nebudu již dále věnovat.

4.6.1 Jazyk HQL

Jazyk HQL (Hibernate Query Language) je plně objektově orientovaným jazyk. Syntaxe jazyka se velmi podobá jazyku SQL. Oproti jazyku SQL však nabízí mnoho výhod, mezi něž patří již výše zmíněná plně objektová orientace a možnost práce s pojmy jako je dědičnost, polymorfismus a asociace. Kromě těchto odlišností však HQL poskytuje řadu běžných konstrukcí jakými jsou agregační funkce, řazení, poddotazy, spojování (join) atd. Výsledkem dotazu v jazyce HQL je přímo objekt nebo případně kolekce objektů.

Nejjednodušší dotaz, který lze v HQL vytvořit je ve formátu `from Trida`. Pro náš příklad třídy `Transaction` by mohl dotaz, který vybere všechny objekty vypadat následovně:

```
List transactions = null;
transactions  = session.createQuery("from Transaction").list();
```

Pokud bychom chtěli získat jeden konkrétní objekt `Transaction` (například transakci s pořadovým číslem 100), dotaz by mohl vypadat takto:

```
Transaction transaction = (Transaction) session.createQuery("
       select t from Transaction as t where t.id = ?1")
       .setLong(1,100).uniqueResult();
```

4.6.2 Vyhledávací kritéria

Vyhledávací kritéria (`Criteria` API), poskytují alternativní způsob pro získávání perzistentních objektů. Jejich nejvýznamnějším rysem je to, že výsledný dotaz je sestavován dynamicky postupným přidáváním podmínek. Použití kritérií je vhodné zejména tam, kde je počet vyhledávacích parametrů značně proměnný. Typickým příkladem, pro který je použití kritérií vhodné, je funkce pokročilého vyhledávání, kde uživatel vybere atributy, podle kterých bude vyhledávání prováděno a nastaví pro ně případně nějaké omezující podmínky.

Příklad pro ukázkovou třídu `Transaction`, který vybere transakce s obnosem mezi 100 a 200 a seřadí je podle něj vzestupně, vypadá následovně:

```
criteria.add(Restriction.between("amount", new Integer(100),
                                           new Integer(200));
criteria.addOrder(Order.asc("amount"));
List transactions = criteria.list();
```

Pokud bychom stejný dotaz chtěli zapsat pomocí HQL, vypadal by takto:

```
from Transaction t where (t.amount between 100 and 200)
order by t.amount
```

4.7 Užitečné nástroje

Hibernate poskytuje řadu různých nástrojů, které nám umožňují usnadnění vývoje aplikace. V podstatě lze přistoupit k vývoji aplikace třemi následujícími způsoby.

Shora dolů

Tento přístup je založen na tom, že nejdříve vytvoříme objektový model aplikace. Dalším krokem vývoje je tvorba mapovacích informací. K tomu lze využít anotací, mapovacích XML souborů nebo značek XDocletu. Pokud máme objektový model a k němu odpovídající mapovací informace, je dalším logickým krokem vytvoření databázového schématu. K této činnosti lze s výhodou použít nástroj `hbm2dll`, který podle mapovacích definicí vygeneruje schéma databáze pro konkrétní databázový systém. Dále lze tento nástroj užít k aktualizaci databázového schématu při každém restartu aplikace, což je velmi výhodné během vývoje aplikace.

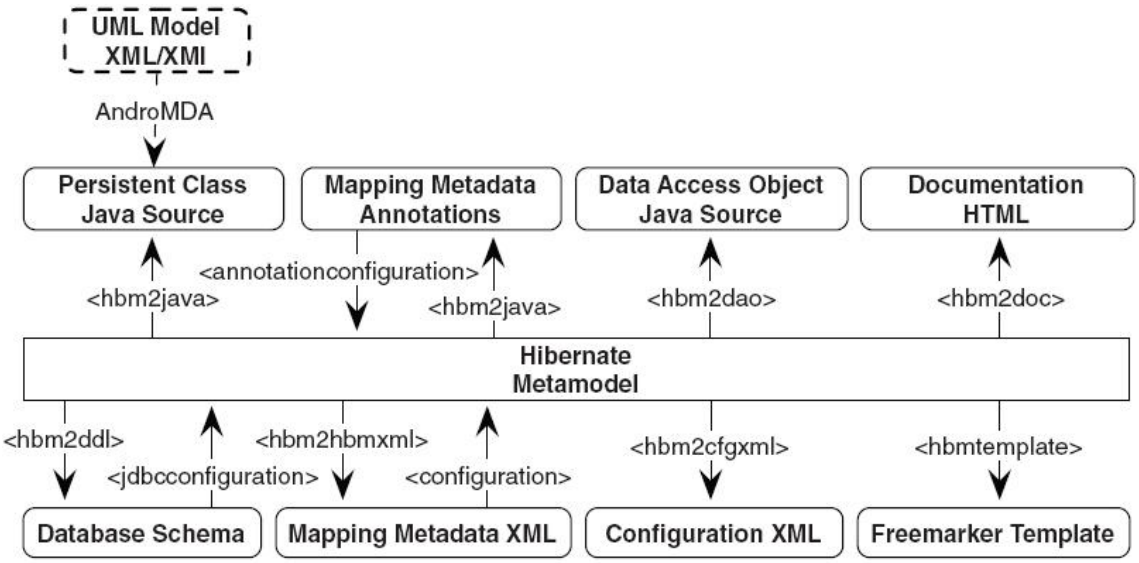
Zdola nahoru

Tento přístup je opakem předchozího. Pokud již máme nějaké databázové schéma a naším úkolem je napsat aplikaci, která s touto databází bude pracovat, můžeme využít dvou nástrojů, které nám opět tuto činnost poměrně výrazně usnadní. Prvním z nich je vygenerování mapovacích informací pomocí nástroje `hbm2hbmxml` a konfiguračního souboru pomocí nástroje `hbm2cfgxml`. Dalším krokem je vygenerování skeletonu tříd Javy pomocí nástroje `hbm2java`.

Zprostředka

Tento postup označovaný jako middle out je založen na tom, že ručně napíšeme mapovací XML soubory. Poté pomocí nástroje `hbm2ddl` vygenerujeme databázové schéma a pomocí `hbm2java` třídy Javy.

Vztahy a vazby mezi zmíněnými nástroji a jednotlivými částmi aplikace (perzistentní třídy, konfigurační soubor, mapovací soubory a databázovým schématem) jsou zobrazeny na obrázku 4.3.



obrázek 4.3

Nástroje pro snadnější vývoj aplikací (Převzato z [8])

4.7.1 Závěrečné shrnutí

V této kapitole jsem se poměrně podrobně zabýval v současné době nejpopulárnějším volně dostupným rámcem pro objektově-relační mapovaní – rámcem Hibernate. Práce s tímto nástrojem je vcelku jednoduchá, přestože počáteční konfigurace mohou zabrat poměrně velké množství času. Hibernate lze použít v mnoha aplikačních architekturách. Může být použito jak v klasické desktopové aplikaci (napsané například ve Swingu), tak i v aplikaci založené na J2EE. V podstatě dnes nelze najít databázový systém, se kterým by Hibernate neuměl spolupracovat. Pro dotazování se nad daty uloženými v databázi nabízí jednoduchý a přitom plně objektový dotazovací jazyk syntaxí podobný SQL, který však oproti SQL přináší mnoho výhod, vyhledávají kritéria a v případech, kdy je nezbytně nutné i možnost poslat databázi klasický SQL dotaz, což však není doporučováno, protože bychom se tím mohli připravit o výhodu přenositelnosti aplikace mezi databázovými servery bez jakékoliv změny v kódu.

Kromě samotného jádra zajišťujícího perzistenci objektů obsahuje Hibernate (nebo lze získat z www.hibernate.org) řadu nástrojů, které mohou usnadnit vývoj aplikací.

5 EJB

Tato kapitola se věnuje technologii Enterprise JavaBeans (dále jen EJB). Na rozdíl například od Hibernate není EJB žádný nástroj, ale specifikace. Tato specifikace je průmyslovou iniciativou vedenou a řízenou společností Sun a některých dalších klíčových dodavatelů a je dnes v podstatě standardem. Je v ní definován komponentový model pro vývoj, rozmístění a vzájemné propojení znovupoužitelných komponent na straně serveru. Technologie EJB podporuje vícevrstvou a zejména distribuovanou architekturu. Aplikační logika je umístěna do tzv. *business objektů*, které jsou součástí aplikačního serveru.

Cílem EJB je pomocí sady různých návrhových vzorů usnadnit vývoj aplikace. Tyto návrhové vzory oddělují business logiku od systémové infrastruktury. Při dodržení specifikace jsou EJB komponenty snadno přenositelné mezi různými platformami a typy middleware, které podporují specifikaci EJB. Informace obsažené v této kapitole pocházejí zejména z [10].

Současná specifikace EJB 3.0 byla vyvinuta pod Java Specification Request (JSR) 220 a je rozdělena na následující tři specifikační dokumenty:

1) Jádro EJB a požadavky. Tento dokument definuje služby poskytované rozhraními mezi instancí beanu a kontejnerem, různé služby poskytované kontejnerem a další detaily týkající se vývoje a nasazování beanů všech typů.
2) Druhým dokumentem je EJB 3.0 specifikace API, která je oproti minulým verzím podstatně zjednodušena.
3) Poslední dokumentem je specifikace API pro perzistenci dat. Java Persistence API (JPA) specifikuje vývoj perzistentní entit psaných POJO stylem.

Nová specifikace EJB přinesla oproti verzím EJB 2.x velké novinky. Problémem dřívějších verzí byla značná složitost. Komponenty psané podle těchto specifikací vyžadovaly kromě samotné aplikační logiky tvorbu velkého počtu doprovodných tříd a XML popisovačů.

Velkým přínosem specifikace EJB 3.0 je použití anotací pro definici metadat. Jejich princip je velmi podobný práci s nástrojem XDoclet. Rozdíl je však v tom, že anotace jsou kompilovány jako součásti tříd Javy. Výhoda anotací je v tom, že se na základě těchto informací generují různé artefakty, kterými lze nahradit popisovače nasazení.

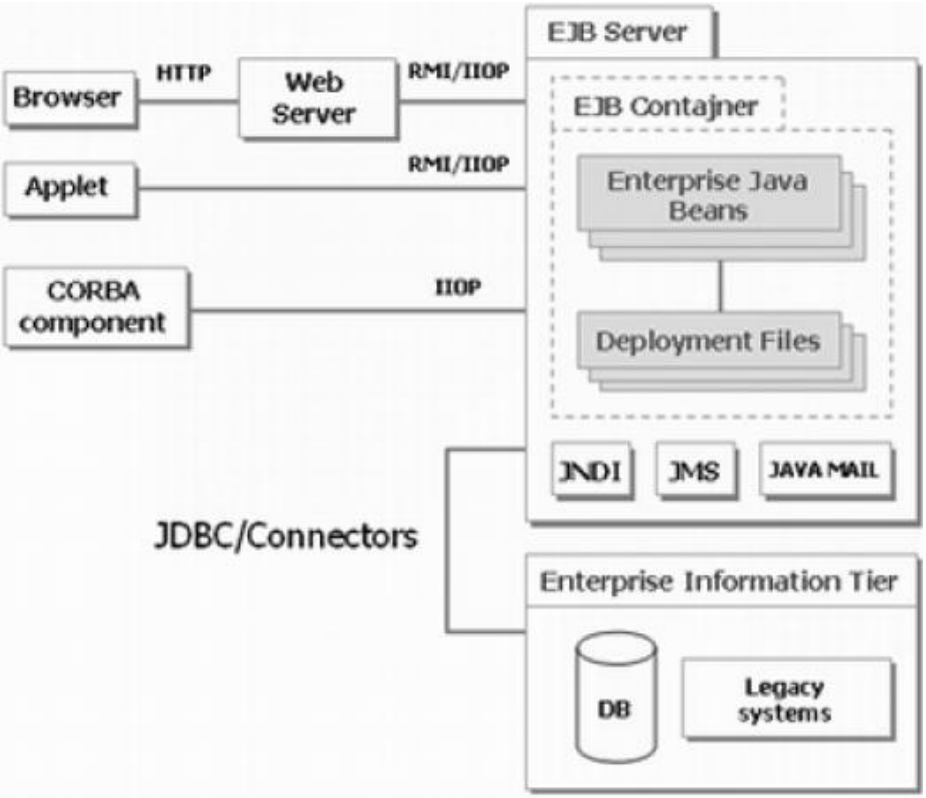
5.1 Architektura EJB

Enterprise beany jsou softwarové komponenty napsané pomocí EJB API (balík `javax.ejb.*`), které mohou být nasazeny v distribuovaném vícevrstvém prostředí. Tyto komponenty vyžadují pro

svoji činnost speciální běhové prostředí, kterým je EJB kontejner. Tento kontejner poskytuje beanům různé služby, které budou dále stručně popsány. Klientem beanu může být v podstatě cokoliv - servlety, JSP, standalone Java aplikace, applet a samozřejmě i další bean. Díky tomu mohou být komplexní úlohy rozdělovány mezi více beanů a při návrhu aplikace lze s výhodou využít strategii rozděl a panuj.

5.1.1 EJB kontejner

EJB kontejner je zodpovědný za řízení enterprise beanů. Nejdůležitější je jeho zodpovědnost za zajištění bezpečnosti v distribuovaném prostředí, ve kterém může být bean spuštěn. Kontejner tyto beany izoluje od přímého přístupu z klientské aplikace.



obrázek 5.1

Architektura EJB (Převzato z [12])

Následující seznam popisuje některé nejdůležitější funkce, které kontejner beanům poskytuje.

Řízení transakcí

Kontejner poskytuje transakční služby, které jsou zpřístupněny pomocí Java Transaction API (JTA). JTA je tzv. high-level rozhraní pro kontrolu transakcí.

Bezpečnost

Bezpečnost je hlavní kritérium pro vícevrstvé aplikace. EJB přidává ke standardním bezpečnostním mechanismům z platformy Java SE tzv. transparentní bezpečnost. Ta znamená, že přístup k metodám beanu je zabezpečen nastavením bezpečnostních atributů.

Perzistence dat

Kontejner může řídit správu perzistentních dat. Dále poskytuje potřebné nástroje pro objektově-relační mapování.

Řízení zdrojů a životního cyklu

EJB kontejner řídí zdroje, jako vlákna, sokety a databázová spojení. Dále řídí životní cyklus enterprise beanů. Je zodpovědný za vytváření instancí beanů, ruší je, provádí jejich pasivaci a aktivaci (ukládání resp. získávání beanů ze sekundárního úložiště).

Vzdálený přístup

Klient umístěný na vzdáleném JVM může vyvolávat metody enterprise beanu.

Podpora pro souběžné požadavky

Kontejner se stará o souběžné požadavky od klientů. Poskytuje vestavěnou podporu pro řízení vláken. Pokud více klientů volá metodu na instanci nějakého beanu, kontejner může serializovat tyto požadavky, čímž povolí pouze jednomu klientu volat instanci beanu v daný okamžik. Pokud k tomuto dojde, jsou ostatní klienti směřováni k jiným instancím beanu, nebo musejí vyčkat, dokud se originální instance beanu neuvolní. Díky vestavěnému mechanismu řízení vláken kontejnerem odpadají při vývoji problémy se synchronizací vícevláknového kódu, což je velkým přínosem.

Clustering a load-balancing

Přestože to není vyžadováno specifikací, většina kontejnerů je vybavena podporou pro clustering a load-balancing. Tato podpora je velmi významná zejména u aplikací, které obsluhují velké množství požadavků a je zapotřebí jejich vysoká škálovatelnost.

5.2 Typy Enterprise beanů

Enterprise JavaBeans jsou rozděleny do třech kategorií. V závislosti na požadavcích vyplývajících z návrhu, lze vybrat vhodný typ beanu z následujících.

5.2.1 Session beany

Session beany (beany sezení) modelují business proces. Tyto beany obsluhují jen jednoho klienta a vykonávají v jeho zájmu akce. Akce prováděné beanem mohou být čímkoliv: výpočtem, přístupem k databázi nebo volání jiného enterprise beanu. Session beany jsou obvykle transientní, existují jen v rámci jedné relace mezi klientem a serverem, jsou pro každého klienta jiné a nemohou být sdíleny. Session beany jsou následujících dvou typů.

5.2.1.1 Stateful

Obhospodařuje data pro určitého klienta mezi jednotlivými voláními metod.

5.2.1.2 Stateless

Lze jej využít mezi jednotlivými požadavky různými klienty, protože si neuchovává informaci o stavu. Vykoná celou požadovanou činnost v rámci jednoho volání metody. Tím pádem není potřeba uchovávat informace o stavu. Výsledkem je, že zabírají méně prostředků, protože jejich instance jsou identické.

5.2.2 Entity

Entity (v dřívějších verzích EJB zprostředkovávány entitním beanem) jsou komponenty, které reprezentují perzistentní data. Jsou to objekty Javy, které v podstatě provádějí cachování informací uložených v databázi. Entity jsou využívány session beany k vykonávání business transakcí. Na rozdíl od session beanu mají entity výrazně delší dobu života. Díky tomu, že reprezentují data v databázi, mohou přežít i havárii celého systému.

Přestože specifikace EJB 3.0 umožňuje používání entitních beanů, měla by být perzistence řešena pomocí entit, což jsou obyčejné POJO třídy jazyka Java. Samotnou vrstvu perzistence pak kontejner řeší pomocí některého mapovacího (ORM) nástroje. Příkladem může být populární aplikační server JBoss, který pro perzistenci používá nástroj Hibernate popsaný v předchozí kapitole.

V souvislosti s entitními beany existují dva typy perzistence, které synchronizují obsah beanu s databází.

5.2.2.1 Bean-Managed Persistence

Beanem řízená perzistence vyžaduje, aby entitní bean obsahoval kód volání, která přistupují k datům v databázi. Výhodou tohoto přístupu je větší kontrola nad tím, jak entitní bean přistupuje k databázi (tento kód musí naprogramovat vývojář zpravidla pomocí JDBC).

5.2.2.2 Container-Managed Persistence

Při použití kontejnerem řízené perzistence zajišťuje EJB kontejner přístup a manipulaci s entitními beany. Kontejner tedy sám načítá data z databáze podle nastavení v deployment deskriptoru

(popisovači nasazení). Tento způsob perzistence je v mnohém výhodnější než předchozí. Jednak šetří čas tím, že umožňuje vývojářům soustředit se na logiku komponenty a ne na to, jakým způsobem jsou získávána a uchovávána perzistentní data. Další velkou výhodou tohoto způsobu perzistence je možnost přenášení beanů mezi různými databázovými servery.

5.2.3 Message-Driven bean

Beany řízené zprávami byly přidány do specifikace EJB až ve verzi 2.0 Jsou to enterprise komponenty, které zpracovávají asynchronní zprávy. Tyto beany byly vytvořeny pro zpracování příchozích JMS zpráv. Tyto zprávy mohou být poslány jakoukoliv enterprise komponentou, JMS aplikací nebo případně systémem, který nepoužívá technologii Java EE. Nejzásadnější rozdíl mezi message-driven beany a session beany je v tom, že klienti k nim nepřistupují pomocí rozhraní. Klientské komponenty tedy přímo nevolají metody těchto beanů, ale přistupují k nim pomocí JMS zasílání zpráv. Základní vlastnosti těchto beanů jsou následující:

- Jsou spuštěny okamžitě po přijetí zprávy od klienta.
- Pracují asynchronně a doba jejich života je relativně krátká.
- Nereprezentují přímo data, ale mohou k nim přistupovat a aktualizovat je.
- Jsou bezstavové.

5.3 Perzistence v EJB 3.0

Jak jsem již uvedl v úvodu této kapitoly, byla to právě perzistence dat, které byla ve verzi EJB 3.0 nejvíce předělána a značně zjednodušena. V nové verzi je použit tzv. lightweight model perzistence, se kterým jsme se již seznámili v kapitole o Hibernate. Namísto entitních beanů jsou nyní používány perzistentní třídy.

5.3.1 Perzistentní třída

Perzistentní třídy musejí splňovat podobné požadavky jako například u rámce Hibernate. Prvním požadavkem je, že třída nesmí být finální (`final`). Dalším požadavkem je, že musí obsahovat bezparametrický konstruktor. Volitelným požadavkem je implementace rozhraní `Serializable`. To je zapotřebí, pokud jsou její instance předávány hodnotou při použití vzdáleného volání. Mezi další požadavky patří nepřístupnost atributů, resp. jejich přístupnost pomocí `get/set` metod. Samotné anotace se umísťují k příslušným atributům nebo jejich `get` metodám. Nový příklad zde již nemá význam uvádět, protože by vypadal v podstatě stejně jako příklad v kapitole 4.3.

5.3.2 Práce s entitami

S entitami pracujeme pomocí tzv. Entity Manageru. Instance `EntityManager` je spojena s perzistentním kontextem. `EntityManager` API poskytuje metody pro tři různé druhy operací:

1) **Řízení životního cyklu entity** – „Život" instance entity má dva hlavní aspekty. Je to vztah ke specifickému perzistentnímu kontextu a synchronizace stavu instance s databází. `EntityManager` rozlišuje mezi čtyřmi stavy (*nový*, *řízený*, *odpojený* a *odstraněný*) v životním cyklu entity.
2) **Operace pro synchronizaci databáze** – Synchronizace s databází probíhá zpravidla po provedení `commit`. Mohou se ale vyskytnout případy, kde je důležité provádět synchronizaci před potvrzením transakce. Nastavení tohoto chování lze dosáhnout pomocí tzv. `flush mode`.
3) **Dotazování** – `EntityManager` poskytuje metodu `find()`, pomocí které lze vyhledávat data podle primárního klíče. Pokud potřebujeme získat více než jednu instanci nebo nechceme vyhledávat podle primární klíče (protože jej například neznáme), můžeme použít dotazování. To se skládá ze tří následujících kroků:
    - Získání instance `javax.persistence.Query`.
    - Přizpůsobení `Query` objektu (nastavení parametrů).
    - Spuštění dotazu.

   `EntityManager` poskytuje možnost psaní dotazů v EJB QL nebo v nativním SQL.

Životní cyklus perzistentního kontextu může být řízen dvěma známými způsoby:

1) **Kontejnerem řízená perzistence** – perzistentní kontext instance `EntityManager` je automaticky kontejnerem rozšířen do všech komponent aplikace, které používají instanci `EntityManager` v rámci stejné JTA transakce. Instanci `EntityManager` lze potom získat takto:

```
@PersistenceContext
EntityManager em;
```

2) **Aplikací řízená perzistence** – Při tomto přístupu není perzistentní kontext šířen automaticky, ale každý `EntityManager` si vytváří izolovaný perzistentní kontext, který je vytvářen a rušen explicitně kódem aplikace. Na následující příkladu je ilustrováno získání `EntityManageru:`

```
@PersistenceUnit
EntityManagerFactory emf;
EntityManager em = emf.createEntityManager();
```

Stejně jako anotace lze použít i `EntityManager` při práci s Hibernate. Není sice součástí jeho jádra, ale lze jej získat v balíku *Hibernate EntityManager* z webu www.hibernate.org

5.4 Závěrečné shrnutí, vhodnost použití

Technologie EJB je určena zejména pro výstavbu rozsáhlých aplikací. U těchto aplikací může toto řešení přinést velkou flexibilitu systému. Naopak pro jednoduché aplikace je zbytečně složitým řešením.

Použití EJB je vhodné zejména tam, kde daná aplikace vyžaduje splnění alespoň jednoho z těchto požadavků (definovaných společností Sun v oficiální dokumentaci) :

- **Škálovatelnost** – pokud se zvyšuje počet uživatelů, je třeba rozložit komponenty na více strojů. Umožňuje distribuované psaní aplikací se zachováním transparentního přístupu.
- **Integrita** – transakce musejí zachovávat integritu
- **Různorodost klientů** – Komponenty jsou snadno dostupné pro vzdálené klienty, čímž je usnadněno psaní tenkých klientů.

6 JDO

Java Data Objects (dále pouze JDO) je standardizované aplikační rozhraní pro perzistenci objektů Javy, vyvinuté samotnou společností Sun Microsystems. Pomocí JDO lze implementovat libovolný způsob perzistence objektů. Může posloužit jako aplikační rozhraní objektového databázového systému, což se zatím mnoho nevyužívá kvůli doposud nízkému rozšíření objektových databází. Mnohem častějším je jeho použití pro mapování objektů a ukládání dat do relačních databází. Důležité je, že aplikační rozhraní je nezávislé na samotné implementaci perzistence.

Sun Microsystems poskytuje tzv. referenční implementaci JDO, která je sice volně dostupná, ale její funkčnost je značně omezena. Z toho důvodu je mnohem vhodnější využití implementace některé třetí strany. Naštěstí v tomto směru existuje velký poměrně kvalitních implementací. Existují komerční implementace (Kodo, JCredo a další), které poskytují kvalitní dokumentaci, plug-iny do vývojových prostředí a další nástroje pro usnadnění vývoje, které zpravidla nabízejí nějakým způsobem omezenou funkčnost zdarma. Samozřejmě kromě komerčních existuje i velké množství volně šířených implementací, z nichž nejkvalitnější je zřejmě JPOX.

V této kapitole se stručně seznámíme s architekturou JDO a s důležitými třídami potřebnými pro perzistenci objektů. Dále se podíváme na způsob konfigurace perzistence a na samotnou práci s perzistentními objekty v aplikaci. Hlavní zdroj informací pro tuto kapitolu je uveden v literatuře pod číslem [13].

6.1 Architektura

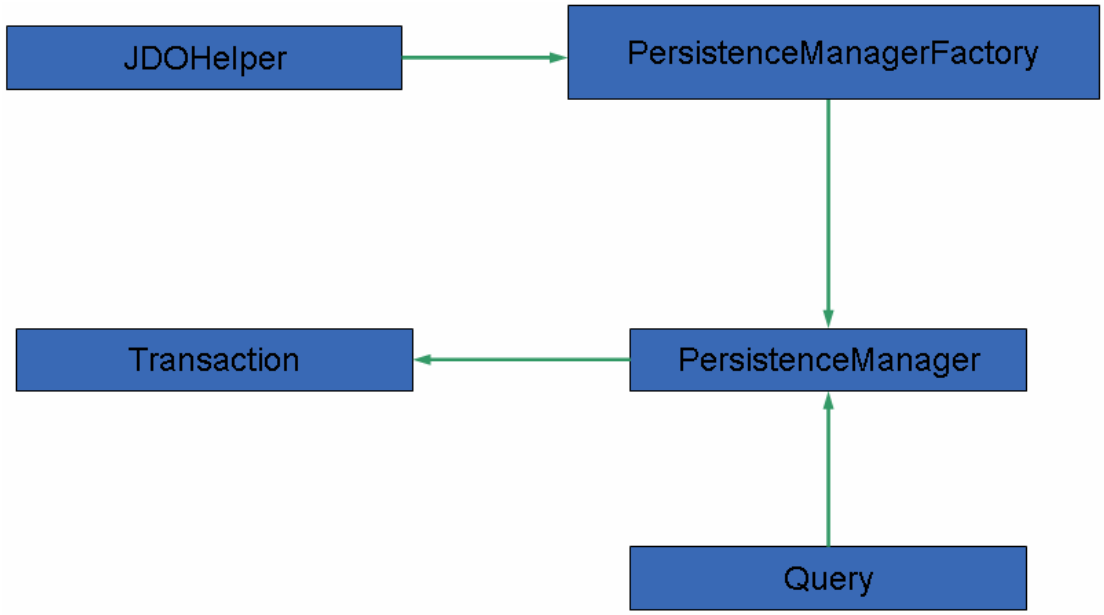
Architektura JDO je velmi jednoduchá a sestává z několika základních tříd. Všechny tyto třídy jsou součástí balíku `javax.jdo`. Těmito třídami jsou:

- `JDOHelper` – třída určená k získání reference na `PersistenceManagerFactory`. Dále nabízí několik statických metod, které poskytují informace o stavu JDO instancí. Ačkoliv obsahuje konstruktor, všechny její metody jsou statické a zpravidla jsou volány přímo, aniž by se získávala instance třídy `JDOHelper`.
- `PersistenceManagerFactory` – je rozhraní, které musí být naimplementováno každou konkrétní JDO implementací. Jeho úkolem je spravování konfigurace perzistence a pomocí něj jsou získávány instance `PersistenceManager`. Konfigurace perzistence je složena z množiny vlastností (`Properties`), které obsahují obvyklé parametry, jako způsob připojení k databázi, uživatelské jméno, heslo apod.
- `PersistenceManager` – je dalším rozhraním nezbytným pro práci. Používá se pro vytváření objektů `Query` a obsahuje metody pro perzistenci objektů. Je navrženo tak, aby

poskytovalo podporu jak pro jednoduché (embedded) systémy, tak pro enterprise aplikační servery.

- `Transaction` – instanci rozhraní `Transaction` lze získat voláním metody `currentTransaction()` rozhraní `PersistenceManager`. Kromě obvyklých metod, kterými jsou `begin()`, `commit()` a `rollback()` obsahuje i zajímavější metody, jako například `setOptimistic()` (pokud ji daná implementace podporuje). Standardně pracují transakce v pesimistickém módu. Pokud v tomto módu pracuje nějaká transakce s daty, ostatní transakce mají k těmto datům vyloučen přístup, dokud daná transakce neskončí. Tím se sice dosáhne větší míry bezpečnosti, ale na druhou stranu dochází při velkém množství přístupu ke ztrátě výkonu. Obecně by tento mód měl být používán výhradně pro tzv. transakce s krátkou dobou života (short-lived transakce).
- `Query` – rozhraní `Query` slouží pro získávání perzistentních objektů. Typickým způsobem je použití dotazovacího jazyka JDOQL (JDO Query Language).

Schéma architektury a vztahy mezi třídami jsou zobrazeny na následujícím obrázku.



obrázek 6.1

Architektura JDO

6.2 Konfigurace perzistence

Abychom mohli ukládat objekty do databáze je třeba specifikovat informace o způsobu jejich uložení (podobně jako u Hibernate a EJB) pomocí tzv. metadat. Metadata jsou obdobou XML mapovacích

souborů používaných u Hibernate. Jedná se o mapovací XML soubor `package.jdo`, který musí být umístěn ve stejném adresáři jako třídy daného balíku. Všechny perzistentní třídy tohoto balíku musí mýt v tomto souboru uvedeny mapovací informace. Alternativou k tomuto, je vytvoření mapovacího souboru `JmenoTridy.jdo` pro každou třídu zvlášť. Následující příklad obsahuje fragment třídy `Account`:

```
package bank;
public class Account {
      private String name;
      private int amount;

      public Account() {}
}
```

Jediným požadavkem, který má JDO na třídu, je implementace bezparametrického konstruktoru. Odpovídající fragment souboru `package.jdo` vypadá následovně:

```
<package name="bank">
      <class name="Account" identity-type="datastore">
            <field name="name">
                  <extension vendor-name="jpox" key="length"
                                    value="max 20"/>
            </field>
            <field name="amount"/>
      </class>
</package>
```

Význam jednotlivých elementů je poměrně jasný. Zajímavé je použití `extension`, které omezuje velikost datového typu výsledného sloupce databázové tabulky. V tomto případě při použití MySQL bude daný sloupec typu `VARCHAR(20)`.

6.3 Práce s perzistentními objekty

Samotná práce s perzistentními objekty je velmi jednoduchá. Nejdříve se podíváme, jak perzistentní objekt vytvořit a dále na způsoby práce s perzistentními objekty.

6.3.1 Vytvoření perzistentního objektu

V následujícím fragmentu kódu je ilustrováno uložení objektu třídy `Account` z předchozí kapitoly:

```
Transaction tx = pm.currentTransaction();
try {
      tx.begin();
            Account account = new Account();
            account.setName("jmeno");
            account.setAmount(1000);

            pm.makePersistent(account);
      tx.commit();
} finally {
      if (tx.isActive()) { tx.rollback(); }
}
```

Na příkladu je jasně vidět, že uložení objektu v aplikaci je velmi jednoduché a značně se podobá práci s objekty v Hibernate.

6.3.2 Editace a zrušení objektu

Editace a stejně tak smazání objektu je velmi jednoduché. Pokud máme objekt třídy `Account` v paměti a přejeme si změnit jeho jméno, které jsem získali pomocí volání metody `getName()`, stačí pouze zavolat jeho metodu `setName("noveJmeno")`. Tento kód musí být uzavřen v `try` bloku v rámci transakce. Tuto transakci musíme samozřejmě korektně ukončit pomocí `tx.commit()`, čímž daný objekt převedeme do stavu hollow.

Analogickým způsobem můžeme objekt odstranit. Pro smazání objektu `account` lze použít metodu `deletePersistent(account)` objektu `PersistenceManager`. Opět tyto metody musejí být uvnitř `try` bloku v rámci transakce. Po ukončení transakce pomocí `tx.commit()` se objekt `account` dostane do stavu transient.

6.3.3 Získávání perzistentních objektů

Objekty z databáze lze získávat pomocí následujících tří možností:

1) **Pomocí identifikátoru objektu**
2) **Pomocí extentu**
3) **Dotazem v jazyce JDOQL**

6.3.3.1 Identifikátor objektu

Získání objektu pomocí identifikátoru je vhodné zejména v případě, že máme v databázi uloženy objekty, jejichž identifikátory si generujeme sami. Pokud bychom chtěli získat účet se jménem *xx26767*, mohl by příslušný kód vypadat následovně:

```
String accountName = "xx26767";
Account account = null;
Transaction tx = pm.currentTransaction();
tx.begin();
      account = (Account) pm.getObjectById(accountName, true);
tx.commit();
```

6.3.3.2 Extent

Extentem třídy se rozumí množina všech objektů dané třídy. Přístup přes extent je vhodný zejména v případě, že chceme získat větší množství dat. Následující příklad získá všechny objekty třídy `Account`:

```
Extent accounts = pm.getExtent(Account.class,true);
```

Objekt třídy `Extent` kromě jiných metod obsahuje i metodu `iterator()`, která vrací objekt třídy `Iterator`, pomocí kterého můžeme získat jednotlivé objekty `Account`.

6.3.3.3 JDOQL

JDO stejně jako ostatní technologie (popsané v předchozích kapitolách) obsahuje svůj vlastní dotazovací jazyk. Tento jazyk nevychází syntaxí z SQL (jako je tomu například u jazyka HQL), ale ze syntaxe Javy. Samotný dotaz je vždy prováděn nad extentem pomocí objektu `Query`. Jazyk JDOQL umožňuje porovnávat hodnoty primitivních datových typů, dále může obsahovat číselné a řetězcové literály, nabízí standardní možnosti pro seřazování výstupu apod. Příklad dotazu, který vybere všechny objekty třídy `Account`, které mají stav vyšší než 1000 a seřadí je podle stavu sestupně, vypadá následovně:

```
Extent accounts = pm.getExtent(Account.class,true);
String filter = "amount > minimalAmount";
Query q = pm.newQuery(accounts, filter);
q.declareParametrs("int minimalAmount");
Integer minimal = new Integer(1000);
q.setOrdering("amount descending");
Collection c = (Collection) q.execute(minimal);
```

6.4 Závěrečně shrnutí

Stejně jako u předchozích technologií, je možné JDO použít v neřízeném i řízeném prostředí v rámci nějakého aplikačního serveru. Za problém JDO by bylo možné označit velké množství různě kvalitních implementací. Jak již bylo zmíněno v úvodu, kvalitní volně dostupnou implementací je zejména JPOX. Podobně jako jiné technologie JDO umožňuje zachytit různé vztahy mezi objekty. Jsou jimi 1:1, 1:N, M:N, dědičnost apod. Různé implementace JDO poskytují obvykle i něco navíc kromě samotného mapování. Zejména užitečné jsou nástroje pro generování databázového schématu.

7 Ostatní technologie

Technologie pro perzistenci objektů v Javě, které byly popsány v předchozích kapitolách nejsou zdaleka jediné, přestože momentálně patří mezi nejpoužívanější. Kromě nich existuje ještě velké množství více či méně kvalitních produktů, jak volně dostupných, tak komerčních. Popsat všechny (byť jen stručně) přesahuje rámec této práce. Proto se zde seznámíme pouze se základními informacemi o SQLJ, ObJectRelationBridge a TopLink a další technologie zmíníme již jen přehledově.

7.1 SQLJ

Jak již sám název napovídá, jedná se o technologii přístupu k databázi založenou na jazyce SQL. Nejedná se tedy o žádný rámec typu Hibernate apod., ale o technologii podobnou rozhraní JDBC. Přestože jsou si JDBC a SQLJ podobné ve způsobu, jakým pomocí nich z naší aplikace pracujeme s daty v databázi, existuje mezi nimi několik velmi podstatných rozdílů. Zdroje, ze kterých jsem převážně čerpal při psaní této kapitoly, jsou [17] a [18].

7.1.1 Zpracování SQL příkazů

Rozhraní JDBC je založeno na tzv. **dynamickém SQL**. To v praxi znamená, že příkazy jazyka SQL se v naší aplikaci vyskytují ve formě běžných řetězů (případně řetězcových proměnných). Tento přístup je na jednu stranu velmi pohodlný a flexibilní, ale na stranu druhou v sobě nese poměrně nepříjemný fakt, kterým je to, že kontrola těchto SQL příkazů je prováděna až za běhu. Z pohledu kompilátoru jazyka Java je takový SQL příkaz obyčejným řetězcem, který není v době překladu žádným způsobem analyzován a případné chyby v něm se projeví až v době běhu aplikace.

Oproti tomu SQLJ používá tzv. **hostitelskou verzi** jazyka SQL. V této verzi jazyka SQL (oproti dynamickému SQL) nejsou SQL příkazy ve formě řetězců, ale přímo součástí zdrojového kódu daného programu. To s sebou přináší nutnost odlišit nějakým způsobem příkazy SQL od ostatních příkazů jazyka Java. K tomu slouží direktiva `#sql`, pomocí které provede preprocesor jejich extrakci a předzpracování. Díky tomu, že je tento přístup postavený na statickém SQL, je možná kontrola SQL příkazů, jak z hlediska syntaktického, tak sémantického, již v době kompilace. Poté, co jsou preprocesorem SQL příkazy zpracovány a zkontrolovány, provede se jejich nahrazení voláními odpovídajících běhových knihoven, pomocí kterých je pak realizován samotný přístup do databáze.

7.1.2 Zpracování výsledků dotazů

Další velmi podstatná výhoda SQLJ oproti JDBC vyplývá ze skutečnosti, že SQLJ pracuje na vyšší úrovni abstrakce. Toho lze s výhodou využít zejména u SQL dotazů typu `SELECT`, kdy nám SQLJ umožňuje vložit výsledek takového dotazu do proměnných příkazem `SELECT INTO` a tím nám ušetřit práci se zpracováváním výsledného objektu s rozhraním `ResultSet`, které sice není nijak složité, ale právě možná díky tomu, že se jedná o monotónní a opakující se činnost, u ní dochází často k chybám.

Uvažujme jednoduchý příklad, kdy z tabulky `zamestnanec` potřebujeme získat `jmeno` a `prijmeni` zaměstnance identifikovaného podle nějakého jednoznačného identifikátoru `id`.

V případě JDBC by příslušný fragment kódu mohl vypadat nějak takto:

```
String sql, jmeno, prijmeni;
sql = "SELECT jmeno, prijmeni FROM zamestnanec WHERE id = 1";
ResultSet rs = st.executeQuery(sql);
while (rs.next()) {
     jmeno = rs.getString(1);
     prijmeni = rs.getString(2);
}
```

Oproti tomu u SQLJ můžeme s výhodou využít příkazu `SELECT INTO`:

```
String jmeno, prijmeni;
#sql { SELECT jmeno, prijmeni INTO :jmeno, :prijmeni FROM
       Zamestnanec WHERE id = 1 };
```

Na tomto velmi jednoduchém příkladu nemusí být výhoda SQLJ tak patrná, ale pokud by výsledná tabulka vznikala spojováním více tabulek a obsahovala značné množství sloupců, byla by již dobře viditelná (minimálně počtem řádků zdrojového kódu).

Tímto způsobem provedeme zpracování příkazu SELECT, jehož výsledkem je tabulka obsahující jeden řádek. Pokud má výsledná tabulka řádků více, potřebujeme pro její zpracování iterátory (někdy též nazývané kurzory). SQLJ nám v tomto směru nabízí dva druhy iterátorů:

1) **Pojmenované** – při jeho vytváření je třeba specifikovat jednak typ proměnné jazyka Java a současně i název sloupce iterátoru použitého k uložení hodnoty každého sloupce získaného z databáze

   Př.: `#sql iterator ZamestnanecIter (String jmeno, …);`

2) **Poziční** – u tohoto druhu iterátoru se specifikuje pouze typ proměnné, do které se budou ukládat hodnoty získané z databáze.
   Př.: `#sql iterator ZamestnanecIter (String, …);`

Od verze 8.1.7 poskytuje SQLJ od Oracle tzv. **scrollable** iterátory, které nám na rozdíl od klasických iterátorů poskytují metody `previous()`, `last()`, `first()` a mnoho dalších, pomocí nichž se můžeme ve výsledné tabulce pohybovat libovolným směrem a ne pouze sekvenčně od prvního k poslednímu. Tyto iterátory mohou být opět buď pojmenované nebo poziční.

Dále SQLJ od Oracle verze 9i a vyšších poskytuje tzv. **scrollable result set** iterátor, který je podobný posunovatelnému pozičnímu iterátoru.

7.1.3 Závěrečné shrnutí

V závěru této podkapitoly provedeme ještě stručné porovnání SQLJ a JDBC.

Výhody SQLJ

- Kontrola SQL příkazů v době kompilace.
- Kompaktnější a úspornější kód, díky čemuž naprogramování stejné funkčnosti jako v JDBC zabírá zpravidla méně řádků zdrojového kódu a tím potenciálně snižuje počet chyb.
- Silně typové iterátory.

Nevýhody SQLJ

- Potřeba mezikroku pro preprocessing.
- Nízká podpora databázových platforem. Standard SQLJ podporují zejména velcí producenti SŘBD, jako například Oracle.

7.2 ObJectRelationalBridge

ObJectRelationalBridge (dále jen OJB) je dalším volně dostupným rámcem pro perzistenci objektů. Cílem tohoto projektu je poskytnou různé API pro perzistenci postavená na jednom jádře. Je to umožněno tím, že rámec má nízko-úrovňový mechanismus pro perzistenci `PersistenceBroker` sloužící ke specifickým účelům, nad kterým je postaveno ODMG 3.0 a JDO API.

Podobně jako Hibernate i OJB pracuje s různými databázovými systémy a poskytuje obvyklé mechanismy pro vztahy mezi objekty, líné načítání kolekcí, řízení transakcí, generování klíčů apod.

Zdrojem pro tuto podkapitolu byla především kniha uvedená v literatuře pod číslem [2] a [19].

7.2.1 Konfigurace OJB

Než můžeme přistoupit k samotné práci s OJB, je třeba provést jeho konfiguraci, která se skládá z několika poměrně jednoduchých kroků.

1) **Instalace knihoven** – prvním krokem je stažení knihoven potřebných pro kompilaci, sestavení a běh aplikace a jejich umístění do `CLASSPATH`.
2) **`Soubor OJB.properties`** – lze nalézt v dokumentaci a zpravidla jej stačí bez úprav nakopírovat do adresáře, který je v `CLASSPATH`. Jeho obsahem je odkaz na XML soubor `repository.xml`, ve kterém se nachází většina nastavení.
3) **Soubor `repository.xml`** – je zodpovědný za načtení dalších tří souborů.
    - `repository_database.xml` – obsahující informace nutné pro připojení k databázi. Jeho struktura je podobná souboru `hibernate.cfg.xml` používaného Hibernate.
    - `repository_internal.xml` – používá OJB pro generování klíčů, zamykání, apod. Tento soubor by neměl být modifikován, ale pouze zkopírován z dokumentace.
    - `repository_user.xml` – obsahuje popis perzistentních tříd (obdoba mapovacích XML souborů Hibernate)

    Tento soubor lze opět nalézt v dokumentaci.

7.2.2 Mapování objektů na databázi

Mapování objektů na tabulky relační databáze je nastaveno ve výše zmíněném souboru `repository_user.xml`. Vzhledem k tomu, že filozofie tohoto mapování je velice podobná mapovacím souborům používaným v Hibernate, nebudeme se zde zaměřovat na žádný příklad.

Je však třeba upozornit na jeden podstatný rozdíl, který spočívá v tom, že OJB očekává, že v tomto jednom souboru bude nastaveno mapování pro všechny perzistentní třídy najednou. Pokud si přejeme mít zvláštní mapovací soubor pro každou třídu, lze toho docílit tím, že jednotlivé mapovací soubory můžeme do `repository_user.xml` vložit pomocí `include`.

7.2.3 Perzistentní třídy

Třídy, jejichž objekty budou ukládány pomocí OJB do databáze musí splňovat dva požadavky:

1) **Konstruktor** – podobně jako Hibernate vyžaduje existenci bezparametrického konstruktoru, který může mít viditelnost `private`.
2) **Pomocný atribut pro asociace** – je specialitou OJB. Pokud má naše třída atribut, který je typem nějakého jiného objektu (má tedy asociaci na jinou třídu), vyžaduje existenci přidaného atributu `ID` typu `Long`. Tento atribut je z pohledu OOP redundantní, ale je nutný

pro možnost tvorby asociací mezi objekty. Následuje fragment třídy `Zamestnanec`, která má asociaci 1:1 na objekt třídy `Adresa`:

```
public class Zamestnanec {
      private Adresa adresa;
      private Long adresaId;
      ...
}
```

7.2.4 Práce s objekty

Samotná práce s objekty probíhá pomocí objektu `PersistenceBroker` a výchozích dotazovacích mechanismů OJB.

7.2.4.1 Ukládání objektů

Uložení objektu do databáze je velmi jednoduchým procesem sestávajícím ze tří kroků. Prvním je zahájení transakce, dále je třeba označit objekt jako perzistentní a nakonec transakci potvrdit. Následující fragment kódu tyto kroky ilustruje:

```
broker.beginTransaction();
broker.store(objekt);
broker.commitTransaction();
```

7.2.4.2 Dotazování se nad perzistentními objekty

K dotazování lze použít OQL (Object Query Language), který je součástí ODMG API, objektu `Criteria` nebo objektu `Query`, který umožňuje zpracovávat dotazy v SQL. Následující dotaz v OQL vrátí kolekci `zamestnanecCol` všech objektů třídy `Zamestnanec`:

```
OQLQuery query = odmg.newOQLQuery();
query.create("SELECT zamestnaneccol FROM " +
             Zamestnanec.class.getName());
DList zamestnanecCol = (DList) query.execute();
```

7.2.4.3 Aktualizace objektu

Aktualizace objektů se typicky provádí ve třech krocích. Prvním je získaní příslušného objektu nějakým z výše uvedených způsobů dotazování se. Druhým krokem je samotná aktualizace příslušného atributu a posledním zajištění toho, že se změna atributu promítne i do databáze. K tomu slouží stejně jako u vkládání metoda `store(objekt)`, které v tomto případě neprovede vložení nového záznamu, ale aktualizaci stávajícího. Pro zvýšení výkonnosti je však vhodné použití metody `store(objekt, ObjectModification.UPDATE)`, které přímo sdělíme, že se má provádět operace `UPDATE`.

7.2.4.4 Mazání objektů

Pokud si přejeme odstranit objekt z databáze, je postup analogický jako při jeho ukládání. Rozdílem je pouze to, že v rámci příslušné transakce voláme místo metody `store(objekt)` metodu `delete(objekt)`.

7.2.5 Závěr

Přestože vývoj OJB nepostupuje takovým tempem, jako například vývoj rámce Hibernate, jedná se o produkt, který ve své aktuální verzi (1.0.4) patří rozhodně k těm kvalitnějším nástrojům pro ORM, které jsou v současné době v kategorii open source dostupné a v některých ohledech se vyrovná poměrně drahým komerčním nástrojům. Mezi jeho hlavní výhody patří:

- Plná implementace ODMG 3.0
- Podpora JDO (plná implementace plánována do OJB verze 2.0).
- Vysoká míra škálovatelnosti využitelná pro distribuování zátěže mezi více databázovými servery.
- Možnost současného použití více různých databázových systémů.
- Integrace více než 800 testovacích případů JUnit pro regresní testování.
- Možnost manipulace s mapovacími daty ze chodu.
- Součást rozsáhlého projektu Apache DB Project, který se stejně jako ostatní projekty Apache Software Fundation vyznačuje velmi kvalitní dokumentací.

7.3 TopLink

TopLink je jedním z nejvyspělejších ORM nástrojů. Jeho vývoj sahá do počátku devadesátých let minulého století, kdy byl vyvíjen jako perzistentní rámec pro jazyk Smalltalk. V roce 1997 byla vytvořena větev projektu nazvaná TopLink for Java. Od roku 2002 vlastní TopLink společnost Oracle a jeho vývoj pokračuje dále ve skupině produktů Oracle Fusion Middleware.

TopLink patří mezi placené produkty, ale po zaregistrování se na stránkách www.oracle.com do komunity OTN je možné jeho stažení a vyzkoušení pro nekomerční účely zdarma.

Vzhledem k tomu, že většina konfigurace a práce s nástrojem TopLink se odehrává pomocí grafických nástrojů, jejichž použití je velmi jednoduché a přímočaré, nebudeme zde uvádět popisy jejich jednotlivých obrazovek, ale soustředíme se především na hlavní rysy nástroje TopLink.

7.3.1 Hlavní rysy a výhody

7.3.1.1 Grafické nástroje

Pro usnadnění mapování, připojení k datovým zdrojům a další obvyklé činnosti.

7.3.1.2 Datové zdroje

Toplink podporuje velké množství různých zdrojů dat.

1) **Relační SŘBD** – kromě databázového systému Oracle, je podporována práce s více než 10-ti v současné době nejběžnějšími databázovými systémy.
2) **EIS** – pomocí J2C lze přistupovat k SŘBD, které nemají relační charakter. Hlavním zástupcem této kategorie, který je podporován díky jeho stálému rozšíření na sálových počítačích, je systém IMS od IBM, který je založený na hierarchickém modelu.
3) **XML** – dále je možné pracovat pomocí architektury JAXB se zdroji dat, které jsou uloženy v dokumentech XML.

7.3.1.3 Query framework

Je velmi propracovaný rámec pro dotazování, který poskytuje mnoho typů dotazů. Základní typy jsou Session queries, Database queries, Named queries a Call queries. Dotazy všech těchto druhů (které ještě doplňuje pět pokročilých) poskytují možnosti přesného nastavení pro danou úlohu. Umožňují mimo jiné použití kurzorů, různých druhů zamykání, spojování, vnořené dotazy, uložených procedur a mnoho dalších.

7.3.1.4 Jazyky pro dotazování

Dotazy lze mimo jiné pokládat pomocí SQL, EJB QL, XML Query a QBE (dotazování se příkladem).

7.3.1.5 Další vlastnosti

Dále TopLink nabízí velmi propracovanou podporu transakcí, různé druhy cache a mnoho nástrojů pro zvýšení výkonnosti a sledování aplikace.

7.3.2 Závěr

TopLink je velmi propracovaným a vyspělým nástrojem, který na rozdíl od většiny volně dostupných rámců poskytuje grafické uživatelské rozhraní, pomocí něhož lze urychlit běžné činnosti. Díky širokým možnostem nastavování umožňuje optimální vyladění výkonu konkrétní aplikace. O míře jeho propracování svědčí i uživatelské dokumentace (čítající bezmála 1300 stránek), ze které jsem čerpal a je uvedena v literatuře pod bodem [21].

7.4 Přehled dalších technologií

7.4.1 CocoBase

Patří mezi další placené rámce pro perzistenci objektů. Jedná se opět o velmi vyspělou technologii od společnosti THOUGHT Inc., jejíž vývoj trvá více než deset let. Podobně jako TopLink poskytuje grafické nástroje, pomocí nichž lze urychlit některé běžné činnosti. Mezi jeho hlavní výhody patří zejména vrstva pro dynamické mapování a velké množství mechanismů pro zvýšení výkonu, které podle společnost THOUGHT Inc přinášejí výkonnostní nárůst při standardním nastavení zhruba 300% oproti ručně psanému kódu v JDBC. Tento výkon lze ještě výrazně navýšit pomocí dalších plug-in nástrojů, zapojením cache a celkovým vyladěním aplikace na více než 1000% v porovnání s JDBC. Podrobnější informace o jeho možnostech lze dohledat v [22].

7.4.2 Torque

Je velmi jednoduchým nástrojem vyvíjeným jako součást rámce pro tvorbu webových aplikací Turbine, který patří do projektu Apache Jakarta. Oproti jiným rámcům se Torque vyznačuje zejména jednoduchostí jeho použití. Jeho hlavní výhodou je, že z příslušného mapovacího souboru XML umožňuje vygenerovat jak schéma databáze, tak i skeleton tříd Javy. Vzhledem k tomu, že se jedná o velmi jednoduchý nástroj, který neposkytuje příliš možností k vylaďování výkonu, je vhodný zejména pro jednoduché aplikace, jejichž vývoj je pomocí něj velmi rychlý. Více informací o něm lze nalézt v [23].

7.4.3 Cayenne

Patří mezi propracovanější volně dostupné rámce. Kromě běžně dostupných vlastností, kterými jsou cache záznamů, líné načítaní, konfigurovatelné zamykání a další, se nad ostatní volně dostupné rámce vyvyšuje především grafickým nástrojem CayenneModeler, který patří do kategorie GUI nástrojů běžných spíše u komerčních produktů.

Kromě toho, že usnadňuje tvorbu mapovacích XML souborů a správu databáze, obsahuje podporu pro reverzní inženýrství, tvorbu dotazů a mnoho dalších užitečných funkcí. Více informací o tomto rámci lze dohledat v [24].

7.5 Závěr

Přestože jsme se v této a předcházejících kapitolách seznámili s poměrně velkým množstvím technologií pro perzistenci objektů jazyka Java, není tento výčet ani zdaleka úplný. Obecně lze rozdělit tyto technologie do dvou hlavních kategorií.

V první kategorii jsou technologie založené na přímém přístupu k tabulkám relační databáze pomocí jazyka SQL. Mezi zástupce této kategorie patří JDBC a SQLJ.

Druhá kategorie obsahuje rámce a technologie, které zajišťují objektově-relační mapování a více či méně nám usnadňují práci s perzistentními daty. Kromě technologií, které byly v této práci uvedeny, existuje ještě nejméně deset rámců různé kvality, jež zde nebyly uvedeny ani názvem. Neexistuje způsob jakým je korektně porovnat a prohlásit, že některý z nich je nejlepší. Pokud se rozhodneme pro perzistenci objektů některý z nich použít, je třeba se vždy zamyslet zejména nad tím, co od něj požadujeme, jak rozsáhlá příslušná aplikace má být, jaké je její předpokládané zatížení a mnoho dalších aspektů, které nemusí být na první pohled zjevné.

Vezmeme-li v úvahu pouze volně dostupné rámce, tak v současné době je zdaleka nejpopulárnějším Hibernate. Důvod tkví zejména v širokých možnostech jeho využití od jednoduchých aplikací až po rozsáhlé distribuované systémy, velkých možnostech nastavení, výkonnosti, rychlosti jakou probíhá jeho vývoj, množstvím kvalitní dokumentace a v neposlední řadě i v relativní jednoduchosti jeho použití.

Neznamená to však, že ostatní volně dostupné rámce, které byly v této práci uvedeny nebo i další, které jsme zde vůbec nezmínili, jako např.: SimpleORM, Java Ultra-Lite Persistence, Ammentos a jiné, jsou špatné, nebo že je jejich vývoj zbytečný. Všechny disponují základní funkčností, která zajišťuje objektově-relační mapování, a u každého z nich lze nalézt něco, čím převyšuje ty ostatní.

8 IS videopůjčovny

V této a zbývajících kapitolách diplomové práce se budeme věnovat popisu tvorby aplikace realizující informační systém pro potřeby videopůjčovny v prostředí Java Enterprise Edition s hlubším zaměřením zejména na vrstvu aplikace zajišťující komunikaci s databázovým systémem, použití architektonického rámce MVC (Model-View-Controller) a návrhového vzoru DAO (Data Access Objects).

8.1 Hlavní cíle aplikace

Na začátku je třeba zdůraznit, že tato demonstrační aplikace si v žádném případě neklade za cíl postihnout všechny požadavky na informační systém pro videopůjčovnu a nebyla vyvíjena ani konzultována s žádným konkrétním zákazníkem. Jak vyplývá ze samotného názvu diplomové práce, tak jejím hlavním zaměřením je porovnání různých technologií pro perzistenci objektů Javy a ne vybudování konkrétní aplikace podle přesně specifikovaných požadavků.

V rámci této aplikace jsem provedl porovnání dvou zcela rozdílných technologií. První z nich je technologie Hibernate a druhou JDBC (obě popsané v předchozích kapitolách). Toto porovnání bylo provedeno tak, že objekty vrstvy DAO, které mají na starost komunikaci s databází, byly kompletně naimplementovány v obou technologiích.

Cílem, který jsem si v tomto směru vytkl bylo zajistit, že obě verze aplikace budou pracovat naprosto stejným způsobem a budou nerozlišitelné z pohledu řídící vrstvy aplikace, která je postavena nad vrstvou DAO.

Přestože to není uvedeno v zadání práce, tak dalším aspektem na který se poměrně podrobně zaměříme je návrh J2EE aplikace podle architektonického rámce MVC a návrhového vzoru DAO. Jak rámec MVC, tak vzor DAO budou podrobněji popsány v kapitole týkající se návrhu systému.

8.2 Stručná specifikace IS

Hlavním úkolem IS videopůjčovny je vést agendu spojenou s zákazníky, jejich výpůjčkami, rezervacemi, spravovat tituly, jejich jednotlivé exempláře a umožnit vyhledání titulů a to jak jednoduché (pouze podle názvu titulu), tak podrobné a přesné vyhledávání a třídění podle velkého množství kritérií.

Jak je dnes běžný trendem, tak uživatelské rozhraní bude realizováno tzv. tenkým klientem, kterým je webový prohlížeč, což přináší podstatnou výhodu v tom, že si uživatelé pro práci s IS nemusejí na svoje PC instalovat žádný speciální SW.

9 Analýza a sběr požadavků

Přestože demonstrační aplikace není vyvíjena pro žádného konkrétního zákazníka, jak již bylo zmíněno v minulé kapitole, provedeme obvyklou analýzu požadavků, která by měla být prvním krokem životního cyklu každého SW produktu.

Pro modelování těchto požadavků (a diagramů souvisejících s dalšími fázemi životního cyklu vývoje aplikace) použijeme produkt Visual Paradigm for UML, který je pro nekomerční využití volně dostupný v edici Community a podporuje UML verze 2.1.

Z této analýzy vyplynuly následující požadavky na aplikaci.

9.1 Druhy přístupu k aplikaci

IS rozlišuje tři skupiny uživatelů:

1) **Zaměstnanci videopůjčovny** – Speciální skupina uživatelů, pro kterou je vytvořena administrátorská část aplikace přístupná pouze po přihlášení. Umožňuje správu, titulů, exemplářů, rezervací, výpůjček a dalších součástí aplikace. Podrobné možnosti této části aplikace jsou zobrazeny v diagramu případů použití na obrázku 9.1 a vysvětleny v jeho následujícím popisu.

2) **Registrovaní uživatelé** – Jsou uživatelé, kteří mají plnohodnotný přístup k uživatelské části aplikace. Kompletní popis jejich možností práce s aplikací je opět ilustrován na diagramu případů použití (obrázek 9.2) a v následném popisu.

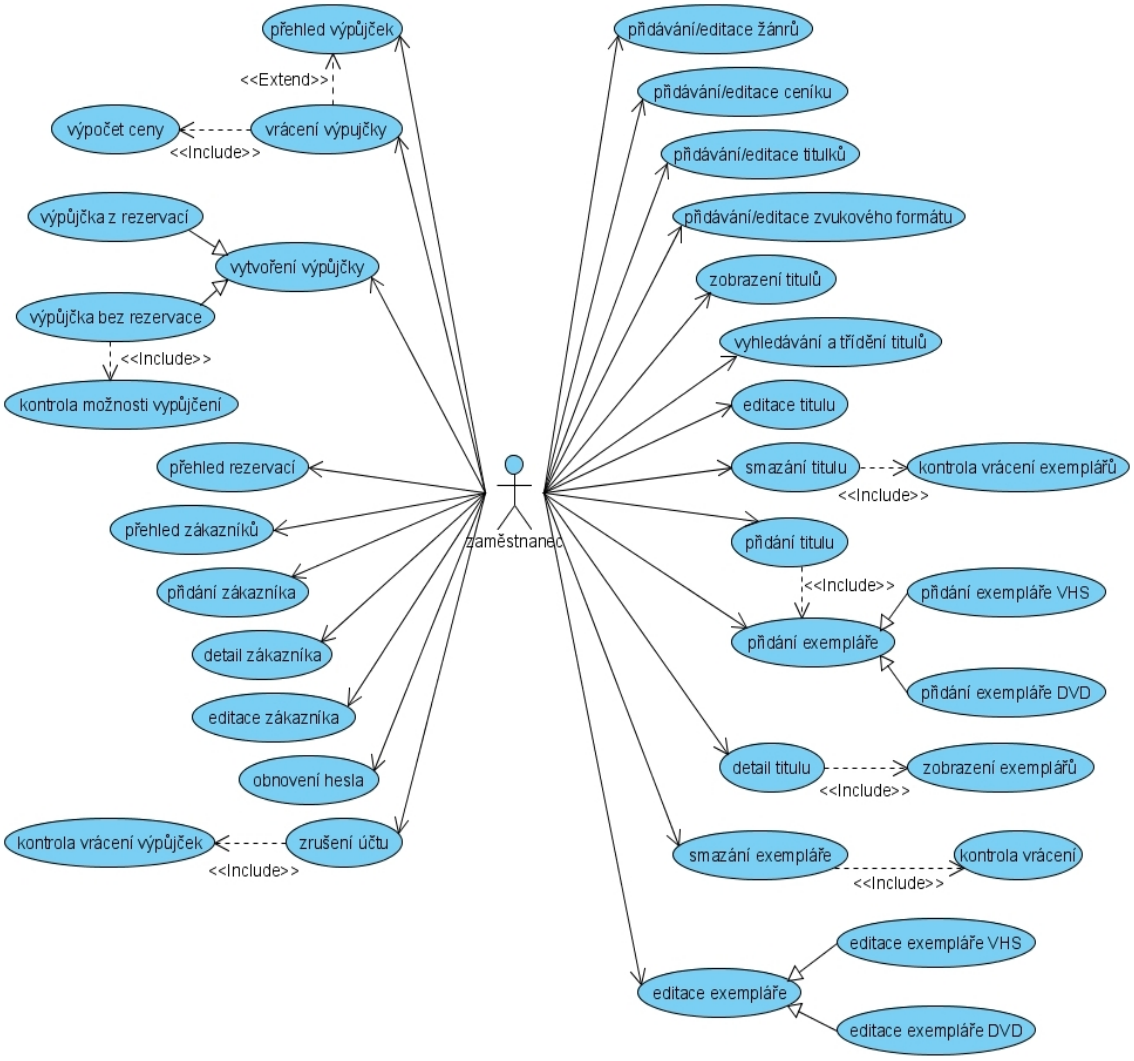
3) **Neregistrovaní uživatelé –** Skupina uživatelů, která sdílí některé případy použití s předchozí skupinou, ale je jim odepřena možnost rezervací, výpůjček apod. Detailní informace o možnostech jejich práce s aplikací je opět vysvětlena na obrázku 9.2 a v jeho popisu.

9.2 Diagramy případů použití

Na následujících dvou stranách jsou zobrazeny digramy případů použití pro jednotlivé skupiny uživatelů. Pevnou součástí těchto digramů je textová specifikace jednotlivých případů použití. Přestože UML pro tuto specifikaci nezavádí žádný pevně daný formát, bývá pravidlem použití nějaké

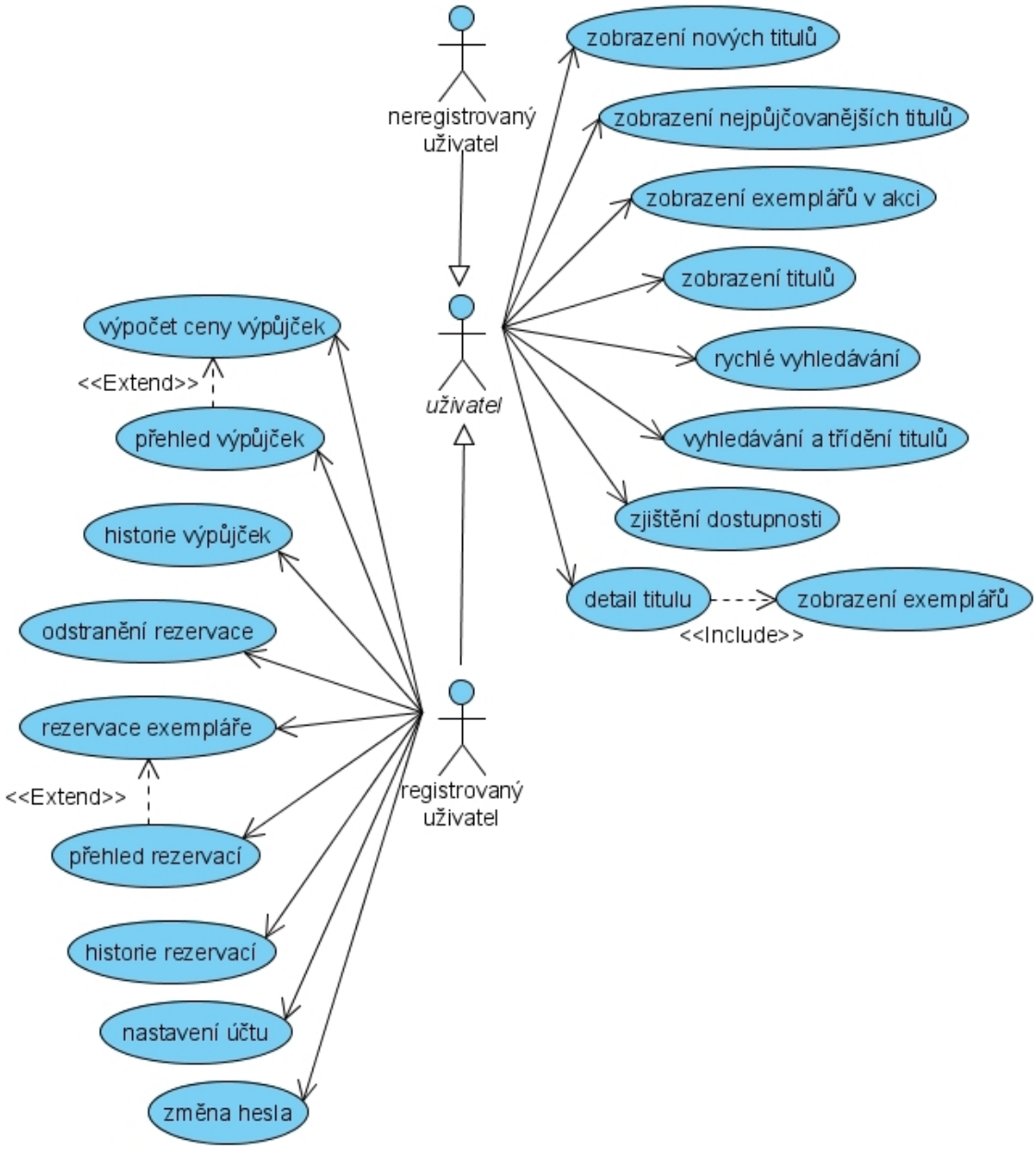
tabulkové struktury, která obsahuje *jméno případu použití*, *identifikátor případu použití*, *popis*, *seznam aktérů*, *hlavní tok*, *alternativní toky* a další elementy.

Vzhledem k tomu, že specifikace případů použití tímto způsobem by díky jejich množství zabrala několik desítek stránek, omezíme se pouze na nestrukturovaný a stručný textový popis.



obrázek 9.1

Funkčnost aplikace z pohledu zaměstnanců videopůjčovny



obrázek 9.2

Funkčnost aplikace z pohledu uživatelů

9.2.1 Specifikace případů použití zaměstnanecké části aplikace

9.2.1.1 Přidávání/editace žánrů

Jedná se o případ použití, který stejně jako další tři následující umožňuje správu tabulek (někdy též nazývaných číselníky), u nichž se předpokládá, že budou naplněny pomocí instalačního skriptu při

nasazení aplikace a následné úpravy v nich budou prováděny pouze výjimečně. V tomto případě jde o možnost přidávat a editovat filmové žánry.

Každý objekt žánr má (kromě atributu *ID*, který mají všechny objekty a nebudu jej proto dále uvádět) pouze atribut *název*. Pokud provedeme přidání nového žánru, vytvoří se nový objekt se zadaným názvem, v případě editace se nastaví atribut název na zadaný text. Aplikace provádí kontrolu jedinečnosti tohoto atributu a neumožní vytvořit dva žánry se stejným názvem.

9.2.1.2 Přidávání/editace ceníku

Umožňuje přidávání nebo změnu položky v ceníku. Objekt ceník má následující atributy:

1) *zkratka* – řetězec dlouhý jeden znak pro označení cenové kategorie
2) *cena* – celočíselné číslo uvádějící cenu za jeden den vypůjčení titulu v dané cenové kategorii.

Aplikace kontroluje, zda jsou oba tyto atributy korektně zadány a vyžaduje v rámci ceníku jejich jedinečnost.

9.2.1.3 Přidávání/editace titulků

Tento případ použití umožňuje přidávání a editaci titulků. Každý objekt titulků obsahuje pouze jeden atribut *jazyk*, který je povinný. Aplikace opět kontroluje, zda byl atribut korektně zadán a zda obsahuje jedinečnou hodnotu.

9.2.1.4 Přidávání/editace zvukového formátu

Poslední případ použití týkající se tabulek číselníků. Umožňuje vytvářet a editovat zvukové formáty. Jediným atributem, který je opět povinným, je v tomto případě atribut *formát*. Aplikace opět kontroluje jeho korektní zadání a jedinečnost hodnoty. Pokud nastane porušení této jedinečnosti, tak (stejně jako u předchozích případů použití) dojde k výpisu chybového hlášení a příslušný objekt se do databáze neuloží.

9.2.1.5 Zobrazení titulů

Provedení výpisu titulů a jejich základních atributů, kterými jsou: *název*, *vydáno*, *cena* (zobrazí se zkratka cenové kategorie) a *žánr*. Tento výpis je směrován do jednoduché tabulky, kde každý řádek reprezentuje právě jeden titul.

Kromě výše zmíněných atributů obsahuje tabulka odkazy „*smazat*“, „*upravit*“, „*detail*“ a „*nový*“ pro přechod na další případy použití. Dále jsou v záhlaví tabulky umístěny ikony pro setřídění titulů podle jednotlivých atributů a v zápatí formulář, který umožňuje vyhledávání v titulech podle všech atributů viz. následující případ použití.

9.2.1.6 Vyhledávání a třídění titulů

Tento případ použití se skládá ze dvou částí.

Prvním je třídění titulů pomocí ikon umístěných v záhlaví tabulky u každého zobrazeného atributu titulu. Tyto ikony mají tvar šipek (a). Pokud uživatel na některou ze šipek klikne, provede se setřídění výpisu titulů podle daného atributu a to buď vzestupně nebo sestupně. Pokud je šipka u některého atributu vyplněna (resp.), je tím naznačeno, že výpis titulů je aktuálně setříděn podle daného atributů v příslušném pořadí. Implicitně je toto třídění nastaveno podle atributu *název* vzestupným směrem.

Druhou součástí je formulář umístěný v zápatí tabulky. Jak již bylo uvedeno výše, tak umožňuje vyhledávání podle všech atributů zobrazených v tabulce. Tyto atributy jsou třech různých druhů, podle nichž se vyhledávání chová:

1) **Textové atributy** – v tomto případě pouze atribut *název*. Pokud je provedeno vyhledávání podle textového atributu neprovádí se porovnání rovnost, ale na podobnost (`LIKE '%hledaný_text%'`) zadaného textu se záznamy v daném sloupci tabulky.

2) **Číselné hodnoty** – v tomto případu použití atribut *vydáno*. V případě vyhledávání podle tohoto atributu, je nejprve provedena kontrola toho, zda byla zadána číselná hodnota. Pokud proběhne kontrola v pořádku, je provedeno vyhledávání s porovnáním na rovnost. Pokud uživatel nezadal číselnou hodnotu, je o této skutečnosti informován chybovým hlášením.

3) **Výběrové hodnoty** – jsou atributy objektů, které nejsou jednoduchými datovými typy, ale typem jiného perzistentního objektu. V rámci tohoto případu použití jde o atributy *ceník* a *žánr*. Výběr takového typu atributu se provádí pomocí roletového menu, ve kterém jsou zobrazené možné položky (existující objekty dané třídy).

Tento výpis titulů, které vyhovují filtru je opět směrován do jednoduché tabulky, kde každý řádek reprezentuje právě jeden titul. Podobně jako tomu je u případu použití "*zobrazení titulů*", je každý řádek tabulky doplněn o odkazy „*smazat*", „*upravit*", „*detail*" a „*nový*" pro přechod na další případy použití.

Důvod toho, že jsem zde provedl takovéto podrobné a obecné vysvětlení způsobu vyhledávání je v tom, že je použito i v jiných případech použití, kde se chová stejným způsobem a nebudu se proto k němu dále vracet. Na závěr ještě uvádím jednoduchý příklad, jakým se s vyhledáváním a tříděním pracuje.

Př.: *Nalezení všech titulů s žánrem komedie v cenové skupině E (30kč/den výpůjčky) a jejich setřídění podle roku vydání.*
Jednotlivé kroky postupu mohou vypadat takto:

1) Omezení výpisu na cenovou skupinu *E*.

2) Setřídění podle *roku vydání* požadovaným způsobem
3) Přidání omezení výpisu pouze na žánr *komedie*.

Pořadí nastavování těchto omezení a třídění není důležité. Zrušení tohoto filtru lze provést odkazem „*zrušit*“, který je umístění v zápatí tabulky a uvede výpis titulů do původního stavu.

9.2.1.7 Editace titulu

U každého titulu lze kdykoliv změnit všechny jeho atributy. Povinnými jsou *název*, *rok vydání*, *cenová kategorie* a *žánr*. Volitelnými pak *originální název* a *délka*. Všechny položky formuláře jsou kontrolovány a pokud jej uživatel nevyplní korektně, je o tom informován chybovým hlášením a k aktualizaci nedojde.

9.2.1.8 Smazání titulu

Tento případ použití je závislý na tom, zda úspěšně proběhne *kontrola vrácení exemplářů* popsaná v dalším odstavci. O tom, jak případ dopadl, je uživatel informován pomocí stručného výpisu.

9.2.1.9 Kontrola vrácení exemplářů

Provede nalezení a porovnání všech exemplářů daného titulu s vypůjčenými exempláři. Končí úspěšně, pokud žádný z exemplářů není vypůjčen. Jinak končí neúspěchem.

9.2.1.10 Přidání titulu

Zaměstnanec provádějící přidávaní nového titulu musí vyplnit všechny povinné atributy (*název*, *rok vydání*, *žánr* a *cenovou kategorii*) a dále může volitelně vyplnit *originální název* a *délku*. Kombinace atributů *název* a *rok vydání* musí být jedinečná. Lze tedy mít dva tituly se stejným názvem, ale nesmí být vydané ve stejném roce. Vzhledem k tomu, že aplikace je navržena tak, že nemůže existovat titul, pokud od něj neexistuje žádný exemplář, je úspěch tohoto případu použití závislý na navazujícím případu použití, kterým je *přidání exempláře*. Všechny hodnoty formuláře jsou opět kontrolovány a případné problémy jsou oznámeny pomocí chybového hlášení.

9.2.1.11 Přidání exempláře

Obecný případ použití pro exempláře všech druhů. Pomocí něj jsou nastaveny atributy *kód exempláře* (musí být jedinečný) a *akční cena*, které jsou společné pro všechny druhy exemplářů. Dále je chování ovlivněno podle toho o jaký druh exempláře se jedná. Momentálně umožňuje aplikace mít exempláře VHS a DVD.

Přidání exempláře VHS

Exemplář VHS má navíc pouze jeden volitelný atribut, kterým jsou *titulky*

Přidání exempláře DVD

Exemplář DVD má kromě dvou atributů obecného exempláře ještě volitelné atributy *bonusy* (typu boolean tedy pouze Ano/Ne), *titulky* a *zvuky*. Poslední dva atributy jsou kolekce, čímž je povoleno k exempláři typu DVD nastavit předem neomezený počet zvukový formátů a titulků.

Jak je obvyklé, tak všechny vstupní hodnoty jsou kontrolovány a pokud dojde k zadání nekorektní hodnoty, je o tom uživatel informován a případ použití skončí neúspěšně. V opačném případě končí úspěchem.

9.2.1.12 Detail titulu

K tomuto případu použití zaměstnanec přistoupit pomocí odkazu „*detail*“ z tabulky titulů. Najde zde všechny dostupné informace o daném titulu. Na tento případ použití navazuje *zobrazení exemplářů.*

9.2.1.13 Zobrazení exemplářů

Provede výpis exemplářů do dvou tabulek. První z nich obsahuje detailní informace o všech VHS exemplářích náležících k tomuto titulu a druhá detailní informace o všech DVD exemplářích. Obě tabulky obsahují odkazy „*nový*“ pro vytvoření nového exempláře daného druhu, „*smazat*“ a „*upravit*“.

9.2.1.14 Smazání exempláře

Tento případ použití je závislý na tom, zda úspěšně proběhne *kontrola vrácení* popsaná v dalším odstavci. Pokud dopadne kontrola úspěšně je příslušný exemplář smazán. O tom, jak případ použití dopadl, je uživatel informován pomocí stručného výpisu.

9.2.1.15 Kontrola vrácení

Provede porovnání příslušného exempláře s vypůjčenými exempláři. Pokud v nich není nalezen, končí úspěšně, v opačném případě neúspěšně.

9.2.1.16 Editace exempláře

Podobně jako *přidání exempláře* jde o obecný případ použití, pomocí kterého lze provést editaci atributů *kód exempláře* (musí být jedinečný) a *akční cena.* Další chování je ovlivněno podle toho, o jaký druh exempláře se jedná.

Editace exempláře VHS

Umožňuje změnu atributu *titulky*.

Editace exempláře DVD

U DVD lze editovat jeho atributy: *bonusy*, *titulky* a *zvuky*.

Všechny vstupní hodnoty jsou opět kontrolovány a pokud dojde k zadání nekorektnímu vstupu, je o tom uživatel informován a případ použití skončí neúspěšně. Neúspěchem v tomto případě znamená, že se žádná změna neprojeví a atributy budou obsahovat původní hodnoty.

9.2.1.17 Přehled výpůjček

Tabulka obsahující seznam výpůjček. Obsahuje informace o *titulu*, *exemplář*i, *datu vypůjčení* , *datu vrácení* (pokud ještě nebyla vrácena obsahuje zobrazí se řetězec "nevrácen") , *zákazníkovi*, který jí provedl a pokud se jedná o nevrácenou výpůjčku odkaz „*vrátit*" pro přechod na další případ použití. Stejně tak informace o titulu, exemplář i a zákazníkovi slouží jako odkazy na případy použití zobrazující o daném druhu objektu detailní informace.

9.2.1.18 Vrácení výpůjčky

Při spuštění tohoto případu použití je zaměstnanec nejprve dotázán (pokud jej nespustil z případu použití *přehled výpůjček*) na číslo karty uživatele. Pokud zadané číslo karty odpovídá číslu karty nějakého zákazníka, zobrazí se tabulka jeho vypůjčených exemplářů. V každém řádku této tabulky je zobrazen jeden vypůjčený exemplář a odkaz „*vrátit*". Po kliknutí na tento odkaz dojde k vrácení výpůjčky a spuštění navazujícího případu použití *výpočet ceny*. Pokud zaměstnanec zadal neexistující číslo karty, je o tom informován pomocí chybového hlášení.

9.2.1.19 Výpočet ceny

Výpočet ceny výpůjčky se provádí podle následujících pravidel. Nejdříve se vypočítá počet dnů, kdy byl exemplář vypůjčen. Pokud je to 0 (tedy zákazník vrací titul ještě ten den, kdy si ho vypůjčil), bere se tato délka výpůjčky jako jeden den. Následně se počet dnů násobí cenou za den vypůjčení. Pokud je exemplář v akci (má nastavený atribut *akční cena*) násobí se touto cenou a pokud ne provede se násobení cenou objektu ceník příslušného titulu, ke kterému exemplář patří. Pokud se najednou provádí vrácení více vypůjčených exemplářů, tak se tyto ceny sečítají a průběžně zobrazují.

9.2.1.20 Vytvoření výpůjčky

Tento případ použití vyžaduje ke svému spuštění opět zadání čísla karty uživatele. Po jeho úspěšném zadání se zobrazí tabulka rezervovaných exemplářů (pokud daný uživatel nějaké rezervoval). Dále se ve vytváření výpůjčky může postupovat dvěma způsoby:

Výpůjčka z rezervací

Pokud existuje nějaká aktivní nevypůjčená rezervace, je u ní v tabulce rezervací zobrazen odkaz „*vypůjčit*", pomocí kterého se provede odstranění exempláře z rezervací a jeho následné vypůjčení.

Výpůjčka bez rezervace

Tento druh výpůjčky vyžaduje zadání kódu exempláře, který si chce uživatel vypůjčit. Zda tímto způsobem dojde k úspěšnému vypůjčení závisí na tom, jak dopadne navazující případ použití *kontrola možnosti vypůjčení*.

Ať již se provede vypůjčení jakýmkoliv způsobem, aplikace neomezuje výpůjční lhůtu. Pokud by se aplikace nasazovala do praxe, záviselo by na zákazníkovi, zda by si přál tuto lhůtu omezit například na jeden týden.

9.2.1.21 Kontrola možnosti vypůjčení

Podle zadaného kódu exempláře se zkontroluje, zda není momentálně vypůjčen. Pokud ano, končí se neúspěšně.

Pokud ne, provede se ještě kontrola, zda daný den nemá exemplář někdo rezervován. Pokud ano, tak se opět končí neúspěšně. Jedinou výjimkou je pokud ho má rezervovaný uživatel, který si jej chce vypůjčit a z nějakého důvodu zaměstnanec neprovedl výpůjčku z jeho rezervací. V takovém případě se mu daný exemplář odstraní z rezervací a případ použití končí úspěšně. Poslední možností je, že exemplář nemá na příslušný den nikdo rezervován. V takovém případě končí případ použití opět úspěšně.

9.2.1.22 Přehled rezervací

Výpis ve formě tabulky obsahující seznam rezervací. Obsahuje informace o *titulu*, *exemplář*i, *datu od* kdy je exemplář rezervován, *datu do* kdy je exemplář rezervován a *zákazníkovi*, který rezervaci provedl. Informace o titulu, exemplář*i* a zákazníkovi jsou zároveň odkazy zobrazující detailní informace o daném objektu.

9.2.1.23 Přehled zákazníků

Tabulka obsahuje základní informace o všech zákaznících. Těmito informacemi jsou *číslo karty*, *jméno* a *příjmení*, *uživatelské jméno* a *datum založení účtu*. Dále jsou u každého zákazníka přítomny odkazy „*zrušit účet*“, „*detail*“ a „*upravit*“ sloužící pro přechod na další případy použití. V samotném záhlaví tabulky je pak odkaz “*nový*“ pro přechod na případ použití *přidání zákazníka*.

9.2.1.24 Detail zákazníka

Zobrazí ve formě jednoduché tabulky veškeré dostupné informace o daném zákazníkovi. Těmito informacemi jsou: *jméno*, *příjmení*, *uživatelské jméno*, *číslo karty*, *email*, *adresa* (ulice, číslo, město, PSČ), *telefon*, *nedoplatek* a *datum založení účtu.*

9.2.1.25 Editace zákazníka

Umožňuje změnu veškerých výše popsaných atributů zákazníka. Všechny vstupy jsou opět kontrolovány a při jakékoliv chybě je o její příčině zaměstnanec informován chybovým hlášením. Jediný atribut, který nelze měnit je *heslo* zákazníka, pro jehož nastavení slouží navazující případ použití *obnovení hesla*.

9.2.1.26 Obnovení hesla

Vzhledem k tomu, že všechna hesla jsou do databáze ukládána v zahašované podobě, nelze zákazníkovi toto heslo sdělit, pokud jej zapomene. Jediná možnost je jeho opětovné nastavení v tomto případě na uživatelské jméno příslušného zákazníka.

9.2.1.27 Zrušení účtu

Tento případ použití je závislý na tom, zda úspěšně proběhne *kontrola vrácení výpůjček* popsaná v dalším odstavci. Pokud dopadne úspěšně, je příslušný uživatelský účet zrušen. O výsledku je zaměstnanec opět informován pomocí stručného výpisu.

9.2.1.28 Kontrola vrácení výpůjček

Zkontroluje, zda uživatel vrátil všechny vypůjčené exempláře. Pokud ano, končí úspěšně, v opačném případě končí neúspěchem.

9.2.2 Specifikace případů použití společných pro všechny uživatele

Případy použití, které jsou dostupné všem uživatelům jsou asociovány s abstraktním aktérem *uživatel*, ze kterého tyto případy použití dědí aktér *neregistrovaný uživatel* a aktér *registrovaný uživatel,* který má asociace s dalšími případy použití dostupné pouze zaregistrovaným uživatelům.

9.2.2.1 Zobrazení nových titulů

Ve formě tabulky je zobrazeno 15 nejnovějších titulů ve videopůjčovně. Tabulka obsahuje informace *název titulu*, *originální název*, *rok vydání*, *cenu* za den vypůjčení, *žánr* a odkaz „*detail"* sloužící k přechodu na další případ použití. Třídění této tabulky je nastaveno od nejnovějších titulů ke starším.

9.2.2.2 Zobrazení nejpůjčovanějších titulů

Opět do jednoduché tabulky je zobrazeno 15 titulů, které jsou nejvíce půjčované. Tabulka obsahuje stejné informace jako v předchozím případu použití a přidává navíc informace o *počtu výpůjček* daného titulu a *počtu exemplářů*, které jsou od příslušného titulu ve videopůjčovně dostupné. Data v této tabulce jsou setříděna sestupně podle *počtu výpůjček.*

9.2.2.3 Zobrazení exemplářů v akci

Tato tabulka obsahuje opět 15 exemplářů, které mají momentálně nastavenu akční cenu. Zobrazené informace o těchto exemplářích jsou *název* a *originální název* titulu ke kterému patří, *akční cenu* za den vypůjčení, *žánr*, *kód exempláře*, *druh exempláře* (VHS nebo DVD) a odkaz „*detail"* pro přechod na další případ použití. Třídění této tabulky je nastaveno podle hodnoty *akční ceny* vzestupně.

9.2.2.4 Zobrazení titulů

Provedení výpisu titulů a jejich základní atributů, kterými jsou: *název*, *originální název*, *vydáno*, *cena* (zobrazí se zkratka cenové kategorie) a *žánr*. Tento výpis je směrován do jednoduché tabulky, kde každý řádek reprezentuje právě jeden titul.

Kromě výše zmíněných atributů obsahuje tabulka odkaz „*detail*" pro přechod na další odpovídající případ použití. Dále jsou v záhlaví tabulky umístěny ikony pro setřídění titulů podle jednotlivých atributů a v zápatí formulář, který umožňuje vyhledávání v titulech podle všech atributů viz následující případ použití.

9.2.2.5 Rychlé vyhledávání

Jednoduchý formulář umožňující rychlé vyhledání titulu pouze podle jeho názvu. Podobně jako na jiných místech se toto porovnání provádí ne na rovnost, ale na podobnost pomocí (`LIKE '%hledaný_text%'`). Minimální délka hledaného textu jsou tři znaky. Pokud uživatel zadá kratší text, obdrží příslušné chybové hlášení. Výsledek vyhledávání je rozdělen na dvě tabulky. V první jsou uvedeny tituly, jejichž český název odpovídá hledanému řetězci a v druhé tituly, u nichž odpovídá hledanému řetězci jejich originální název.

9.2.2.6 Vyhledávání a třídění titulů

Tento případ použití odpovídá stejně pojmenovanému případu použití ze zaměstnanecké části aplikace. Rozdíl je v neexistenci odkazů „*smazat*", „*upravit*" a „*nový*" a naopak přidání textového atributu „*originální název*".

9.2.2.7 Detail titulu

Tento případ použití opět odpovídá stejně pojmenovanému ze zaměstnanecké části aplikace. Jediným rozdíl je v navazujícím případu použití *zobrazení exemplářů.*

9.2.2.8 Zobrazení exemplářů

Rozdíl v tomto případu použití je v neexistenci odkazů „*smazat*" a „*upravit"* u jednotlivých exemplářů a naopak přítomností odkazů „*dostupnost*" a pokud je přihlášen registrovaný uživatel ještě odkazu „*rezervovat*", které slouží k přechodům na odpovídající případy použití.

9.2.2.9 Zjištění dostupnosti

Provede zobrazení dostupnosti příslušného exempláře. Pokud je exemplář někým rezervován, zobrazí informaci o tom, kolik je na něj rezervací a dále od kdy, do kdy jsou tyto rezervace aktivní. V případě, že je exemplář momentálně vypůjčen, zobrazí navíc informaci o tom, od kterého dne je vypůjčen.

9.2.3 Specifikace případů dostupných pouze pro registrované uživatele

9.2.3.1 Výpočet ceny výpůjček

Tento případ použití provede výpočet ceny všech výpůjček příslušného uživatele a jejich vypůjčených exemplářů k aktuálnímu dni. Po zobrazení této ceny pak nabídne možnost pomocí odkazu „*Zobrazit výpůjčky*“ přejít k případu použití *přehled výpůjček.*

9.2.3.2 Přehled výpůjček

Provede zobrazení všech nevrácených výpůjček včetně všech nevrácených exemplářů náležících do těchto výpůjček. Tento výpis je opět směrován do tabulky obsahující *název titulu*, *cenu* za den vypůjčení, *žánr*, *datum vypůjčení* a *cenu* za vypůjčení příslušného exempláře k aktuálnímu dni.

9.2.3.3 Historie výpůjček

Případ použití, který je velmi podobný předchozímu. Jediným rozdílem je to, že jsou zde zobrazeny všechny výpůjčky uživatele (tedy i ty, které již byly vráceny). Z tohoto důvodu je zde přítomný ještě atribut *datum vrácení*. Dále se u vrácených exemplářů samozřejmě nevypisuje cena za vypůjčení exempláře k aktuálnímu dni, ale cena kterou uživatel zaplatil při vrácení tohoto exempláře. Vrácené exempláře jsou dále od nevrácených barevně odlišeny.

9.2.3.4 Odstranění rezervace

Případ použití provede odstranění příslušné rezervace. Po jeho provedení informuje uživatele o svém úspěšném provedení.

9.2.3.5 Rezervace exempláře

Pomocí tohoto případu použití lze rezervovat příslušný exemplář, od kterého byl spuštěn. Pravidla pro rezervace jsou následující. Každá rezervace se provádí na pět dní. Pokud již existují na exemplář nějaké rezervace, tak se tato nová rezervace zařadí za poslední z nich. Tedy pokud je poslední rezervace například na období od 15. do 20. dubna, nová rezervace se provede na období 21. až 26. dubna. Pokud na exemplář žádné rezervace neexistují, nastaví se počátek rezervace na aktuální den. V případě, že je exemplář vypůjčen, tak vzhledem k tomu, že aplikace žádným způsobem neomezuje

počet dní výpůjčky, nemá tato skutečnost na rezervace vliv. Pokud tedy uživatel rezervuje momentálně vypůjčený exemplář, na který zatím neexistuje žádná rezervace, je počátek této jeho rezervace nastaven na aktuální den. Poslední pravidlem pro rezervace je nemožnost rezervovat příslušný exemplář daným uživatelem vícekrát. Pokud tedy má uživatel exemplář již v budoucnosti rezervovaný a pokusí se jej rezervovat znovu obdrží chybové hlášení, které jej o tom informuje.

Pokud proběhne rezervace úspěšně, nabídne aplikace uživateli možnost přechodu na případ použití *Přehled rezervací* pomocí odkazu „*Zobrazit rezervace*“.

9.2.3.6 Přehled rezervací

Tento případ použití provede výpis všech aktivních rezervací ve formě jednoduché tabulky. Tato tabulka obsahuje *název titulu*, *cenu* za den výpůjčky, *žánr*, *typ exempláře* (DVD nebo VHS), *datum vytvoření* rezervace, *datum platnosti* rezervace a odkaz “*odstranit*“ pro přechod na odpovídající případ použití.

9.2.3.7 Historie rezervací

Případ použití, který je opět velmi podobný předchozímu. Rozdílem je to, že jsou zde zobrazeny i rezervace, které byly provedeny v minulosti a nedošlo k jejich vyzvednutí a tím i převedení do výpůjček, které jsou opět barevně odlišeny.

9.2.3.8 Nastavení účtu

Registrovaný uživatel má možnost změnit některé svoje údaje. Konkrétně jsou těmito údaji jeho *adresa*, *email* a *telefonní číslo.* Aplikace opět provádí ověřování správnosti údajů a v případě potřeby vypíše příslušné chybové hlášení.

9.2.3.9 Změna hesla

Uživatel má samozřejmě možnost změnit svoje heslo. Jak bývá zvykem, tak pro jeho změnu musí nejdříve zadat *staré heslo* a následně 2x *heslo nové*. Aplikace se dále zachová podle jednoho z následujících scénářů:

1) *Nesprávné staré heslo* – informace o chybě, heslo se nezmění.
2) *Nové heslo a jeho opakování nesouhlasí* – informace o chybě, heslo se nezmění.
3) *Vše proběhne v pořádku* – informace o úspěchu, změna hesla.

10 Návrh systému

Po provedení analýzy požadavků kladených na systém a jejich podrobném popisu dochází k posunu projektu do další z jeho životních fází, kterou je návrh. Dříve než přistoupíme k popisu aplikace samotné, je třeba zmínit filozofii návrhu podle architektonického rámce Model-View-Controller (MVC) a návrhového vzoru Data Access Object (DAO) a zasadit je do kontextu J2EE aplikace.

10.1 Význam návrhových vzorů a softwarových architektur

Návrh softwarové architektury by měl být jedním z prvních kroků při vývoji každé aplikace. Důvodem je to, že další fáze návrhu se od námi zvolené architektury dále odvíjejí. Veškeré architektury rozdělují návrh aplikace do více či méně vrstev, které tvoří hierarchickou strukturu. Architektura může být otevřená (komunikace probíhá nejen mezi sousedními vrstvami) nebo uzavřená, kde spolu komunikují pouze vrstvy bezprostředně sousedící, což je výhodnější zejména z hlediska udržovatelnosti.

V rámci architektury pak v jednotlivých jejích vrstvách vystupují do popředí návrhové vzory, což jsou jakási osvědčená vzorová řešení problémů, která se v příslušných vrstvách vyskytují.

Je třeba poznamenat, že jak softwarové architektury, tak návrhové vzory nejsou určeny k tomu, aby programátora vedly k tvorbě „pouze“ funkčního softwaru. To, že bude výsledný produkt splňovat požadavky zákazníka a bude plně funkční, se pokládá za samozřejmost. Jejich pravým účelem je poskytnout vodítka k tvorbě softwaru, který bude kromě svojí funkčnosti poskytovat jakousi přidanou hodnotu zejména v podobě **podporovatelnosti**.

Abychom mohli prohlásit, že námi vytvořený systém je podporovatelný, musí splňovat tyto vlastnosti:

1) **Srozumitelnost a pochopitelnost** – kvalitní zdrojový kód.
2) **Udržovatelnost** – možnost odstraňovaní chyb, měnit a přidávat funkcionalitu.
3) **Škálovatelnost** – schopnost přizpůsobení se změně parametrů (např.: velký nárůst počtu uživatelů, růst velikosti databáze, apod.).

V souvislosti se softwarovými architekturami vystupují ještě další metriky související zejména s mírou udržovatelnosti softwaru, kterými jsou:

1) **Koheze** – sleduje závislosti v příslušném balíčku
2) **Propojení** – vyjadřuje složitost vazeb, které budou mezi příslušnými balíčky.

V tomto smyslu bychom se měli vždy snažit maximalizovat kohezi a propojení naopak minimalizovat a tím vytvářet uzavřenou architekturu. Hlavní výhodou takového návrhu je to, že při změně funkčnosti (nebo chybě) v některé z vrstev, nedojde k lavinovitému šíření do dalších. V tomto směru lze s výhodou využít zejména rozhraní, která představují jakýsi jediný vstupní bod do příslušné vrstvy. Všechny tyto principy jsem se snažil při návrhu aplikace dodržovat a na příslušných místech na ně upozorním.

10.2 Architektonický rámec MVC

MVC je velice populárním rámcem pro tvorbu aplikací v prostředí J2EE. Je definován tak, že rozděluje architekturu do tří vrstev podle jejich účelu:

1) **View** – vrstva zajišťující prezentaci.
2) **Controller** – vrstva oddělující prezentaci od dat, která obsahuje většinu aplikační logiky příslušné aplikace.
3) **Model** – vrstva obsahující objekty z aplikační domény.

10.3 Návrhový vzor DAO

Jedná se o návrhový vzor, patřící mezi tzv. *Core J2EE* vzory, definovaný přímo společností Sun. Motivací pro jeho vznik byl stále se opakující problém potřeby přístupu k datům, která jsou uložena ve zdrojích různého charakteru. Těmito zdroji mohou být soubory, relační SŘBD, objektově orientované SŘBD a další.

Nemusí však jít pouze o různé druhy zdrojů dat, aby nastal problém. Pokud máme například aplikaci postavenou na JDBC a provedeme přechod například z MySQL na Oracle, může dojít k podobným problémům. Příčinou problémů v tomto případě není JDBC API, které je standardizované a pro všechny databázové platformy stejné, ale SQL příkazy, které ve svém kódu používáme, protože jejich syntaxe a formát se mohou u konkrétních databázových systémů lišit.

Řešení

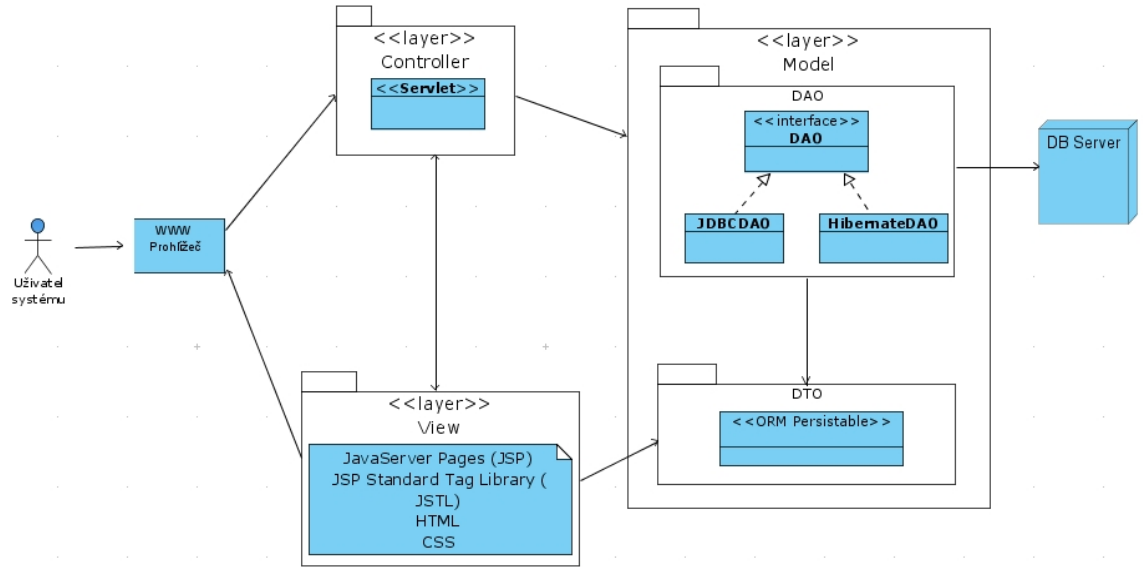
Řešení, které nám tento návrhový vzor předkládá, je založeno na vytvoření další vrstvy aplikace, jejímž úkolem je ukrýt detaily procesu získávání dat před vyššími vrstvami aplikace a poskytnout těmto vrstvám jednotné rozhraní pro přístup k nim.

Tento vzor je tedy naprosto ideálním pro řešení problémů se změnou technologie na perzistenci objektů, což je případ nejen mojí diplomové práce, ale v současné době poměrně často řešený problém u společností, které mají svoje stávající aplikace založené na JDBC a snaží se přejít na nějaký rámec pro perzistenci objektů. Pokud mají perzistentní vrstvu těchto stávajících aplikací

postavenu na tomto návrhovém vzoru, tak může zmíněný přechod být poměrně jednoduchý a nemusí při něm dojít k ovlivnění ostatních vrstev, jak jsem se přesvědčil při implementaci své aplikace. Detailní popis tohoto návrhového vzoru lze najít v [20].

10.4 Spojení MVC a DAO do J2EE aplikace

Pokud spojíme výše zmíněné informace o rámci MVC, návrhovém vzoru DAO a zasadíme je do kontextu J2EE aplikace, výsledkem bude následující obrázek, představující návrh architektury aplikace.



obrázek 10.1

Návrh architektury

Tento obrázek nepředstavuje pouze návrh samotné architektury, ale je ještě obohacen o celkový kontext okolí, ve kterém bude naše aplikace provozována, kde na jedné straně aplikace vystupují klienti, kteří pracují s internetovým prohlížečem a na straně druhé je databázový server, do něhož jsou ukládány perzistentní objekty.

V následujících kapitolách budou podrobně popsány jednotlivé vrstvy a příslušné balíčky, které se v nich nacházejí.

10.5 Vrstva Model

Je z pohledu zaměření mojí diplomové práce vrstvou stěžejní a proto se jí budu věnovat podrobněji. Jak již bylo zmíněno v kapitole o rámci MVC, tak jde o vrstvu obsahující objekty z aplikační domény. Nacházejí se zde tedy třídy, jejichž objekty je potřeba ukládat do databáze.

Výše zmíněný návrhový vzor DAO při použití MVC tuto vrstvu rozděluje na dvě části. První je sekce (v prostředí Javy balík) DTO (Data Transfer Object), která obsahuje zmíněné perzistentní objekty a druhou sekce DAO, která má na starosti práci s databází a nabízí nadřazené vrstvě (Controller) jednotné rozhraní.

10.5.1 Sekce DTO – diagram perzistentních tříd

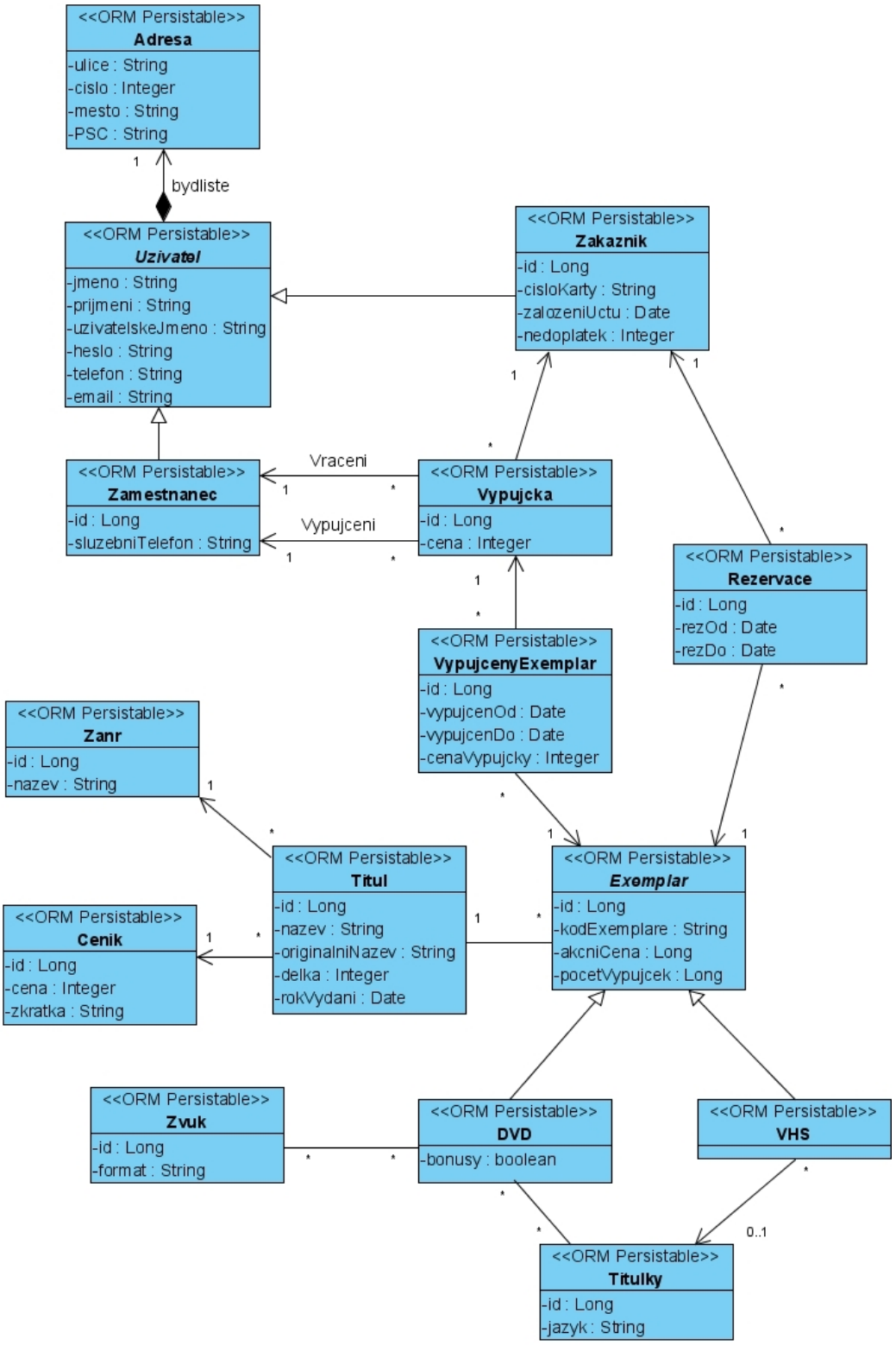
Pro splnění požadavků, které vyplynuly z analýzy, jsem provedl návrh diagramu perzistentních tříd, který je zobrazen na následující stránce.

Diagram tříd je zde zobrazen bez metod. Důvodem je jeho čitelnost po vytištění. Pokud by v něm byly zobrazeny i všechny metody, stal by se po vytištění zcela nečitelným. Úplný diagram je spolu se všemi dalšími umístěn na přiloženém médiu v adresáři *Diagramy*.

Model těchto tříd je z pohledu Javy zcela shodný pro obě verze aplikace. Jediným rozdílem v Hibernate verzi oproti JDBC je přidání anotací do kódu pro mapování. Pokud bych pro Hibernate použil mapování pomocí XML souboru, mohl by být shodný opravdu 1:1, ale jak jsem uvedl v kapitole o Hibernate, tak se mi použití anotací zdá být výhodnější. Odstranit tyto anotace do JDBC verze není žádným problémem, protože v každé třídě zabírají pouze několik málo řádků a díky tomu, že jsou uvozeny direktivou `@`, je lze nalézt velmi rychle.

Při návrhu tříd jsem se kromě splnění požadavků snažil postihnout všechny možné typy vztahů, které se v objektově orientovaných aplikacích vyskytují, abych na nich ověřil možnosti Hibernate a mohl provést porovnání složitosti implementace obou verzí aplikace. Díky tomu se v diagramu nachází jednosměrné i obousměrné asociace (dle UML 2 jsou obousměrné asociace značeny jako asociace bez navigovatelnosti) s různou kardinalitou. Zejména je v tomto směru zajímavá kardinalita m:n. Dále se zde nachází silnější forma asociace – kompozice a samozřejmě vztahy dědičnosti.

K jednotlivým zajímavostem u konkrétních vztahů se vrátíme v příští kapitole, kde bude popsána implementace.



obrázek 10.2

Diagram perzistentních tříd

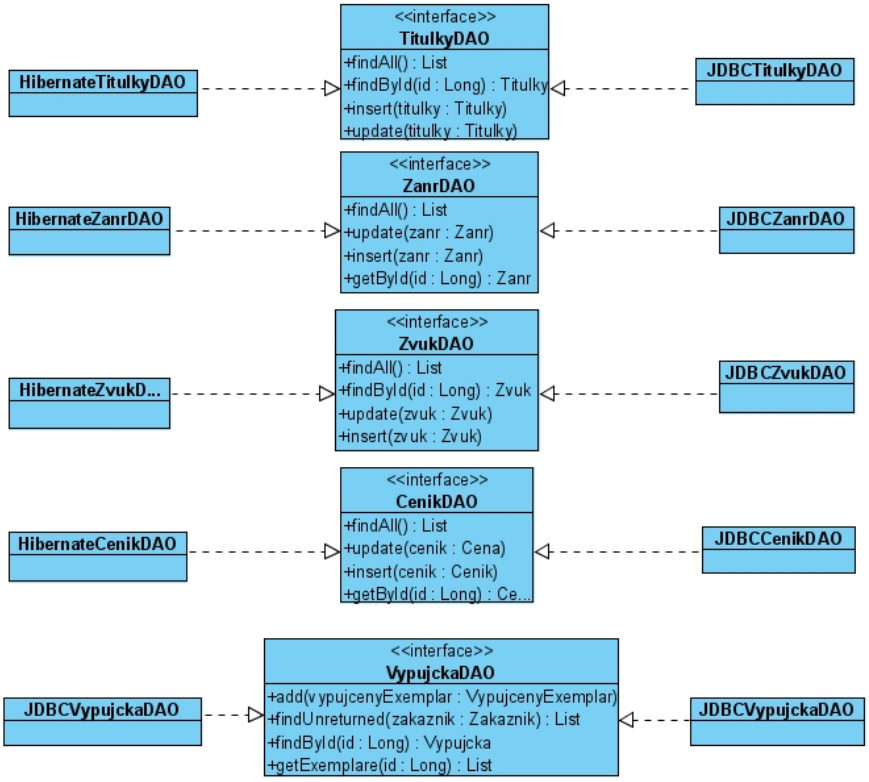
10.5.2 Sekce DAO – digram rozhraní a příslušných tříd

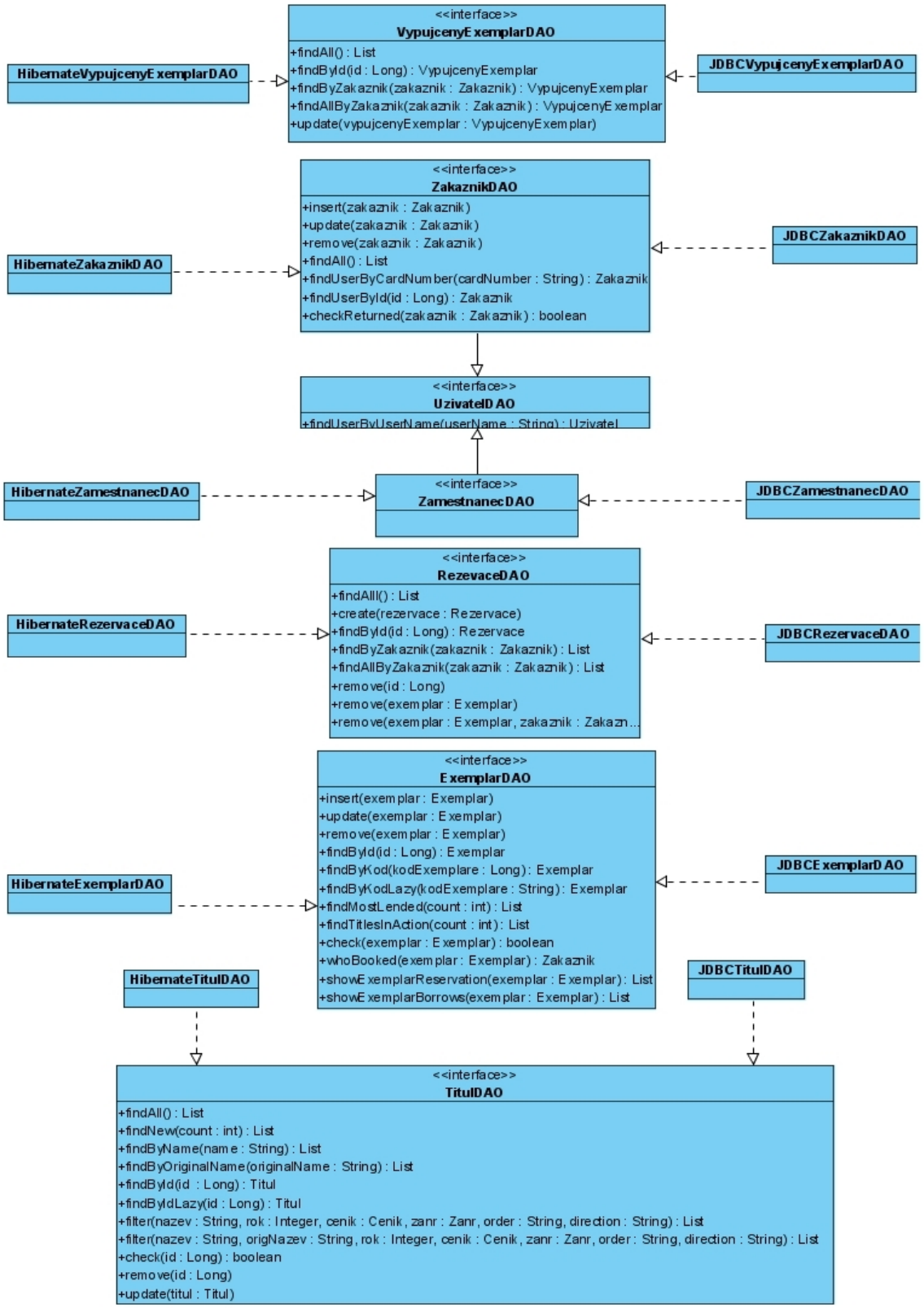
Na této a následující stránce je zobrazen diagram rozhraní sekce DAO a příslušných tříd obou verzí aplikace, které daná rozhraní implementují. Tato rozhraní jsou naprosto shodná pro obě verze aplikace. Nacházejí se v nich pouze definice metod, které musejí být naimplementovány příslušnými třídami jak v JDBC, tak i v Hibernate verzi aplikace. Každá z těchto tříd implementuje pouze metody předepsané příslušným rozhraním a žádná z nich neobsahuje žádnou metodu navíc.

Tato rozhraní obsahují metody (často označované jako CRUD metody – Create, Read, Update a Delete) sloužící pro komunikaci mezi aplikací a databází, které provádějí potřebné operace s perzistentními objekty. Těmito operacemi jsou typicky uložení do databáze, nalezení podle `id`, aktualizace, smazání, nalezení všech objektů příslušné třídy apod.

Díky návrhu pomocí MVC a vzoru DAO jsou třídy implementující tato rozhraní jediným místem, ve kterém se obě verze aplikace zcela odlišují. V diagramu by mělo být ještě zobrazeno rozhraní `MySQLDB`, které implementují pouze třídy JDBC verze aplikace. Toto rozhraní nedefinuje žádné další metody, ale obsahuje pouze deklarace konstant nutných pro připojení k MySQL databázi, aby nemusely být uváděny v každé třídě zvlášť. Z důvodu čitelnosti po vytištění je toto rozhraní z diagramu vypuštěno a je zobrazeno pouze na fragmentu tohoto diagramu (obrázek 10.4). Úplný obrázek 10.3 s oběma rozhraními je opět dostupný na přiloženém médiu.

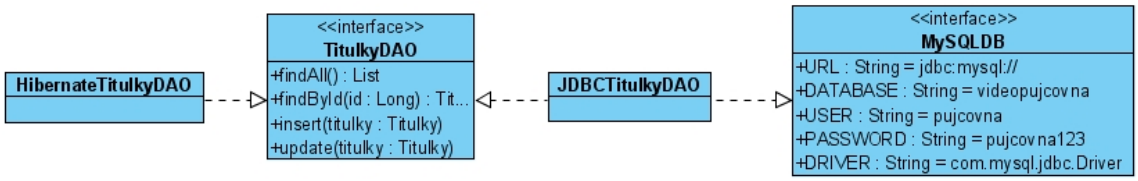
Vzhledem k zaměření této práce jsou implementace těchto metod tím nejzásadnějším. Z tohoto důvodu se k nim podrobně vrátím v následující kapitole, která se týká samotné implementace a provedu podrobné porovnání jednotlivých verzí aplikace z různých hledisek.





obrázek 10.3

Digram rozhraní a tříd sekce DAO



obrázek 10.4

Fragment diagramu rozhraní a tříd sekce DAO se zobrazením MySQLDB rozhraní

10.6 Vrstva Controller

V prostředí J2EE aplikace je řídicí vrstva (Controller) tvořena typicky servlety. Servlet je třída jazyka Java, která má zpravidla za svého předka abstraktní třídu `HttpServlet` z balíku `javax.servlet.http`. V této třídě se nachází několik metod, které můžeme v naší třídě překrýt a naimplementovat je požadovaným způsobem.

Všechny servlety mojí aplikace překrývají metody `doPost(…)` a `doGet(…)`. Z důvodu velikosti diagramů jsem u nich neuvedl typ předávaných parametrů `request` a `response`. Jedná se vždy o `HttpServletRequest` a `HttpServletResponse`, což jsou rozhraní z balíčku `javax.servlet.http`.

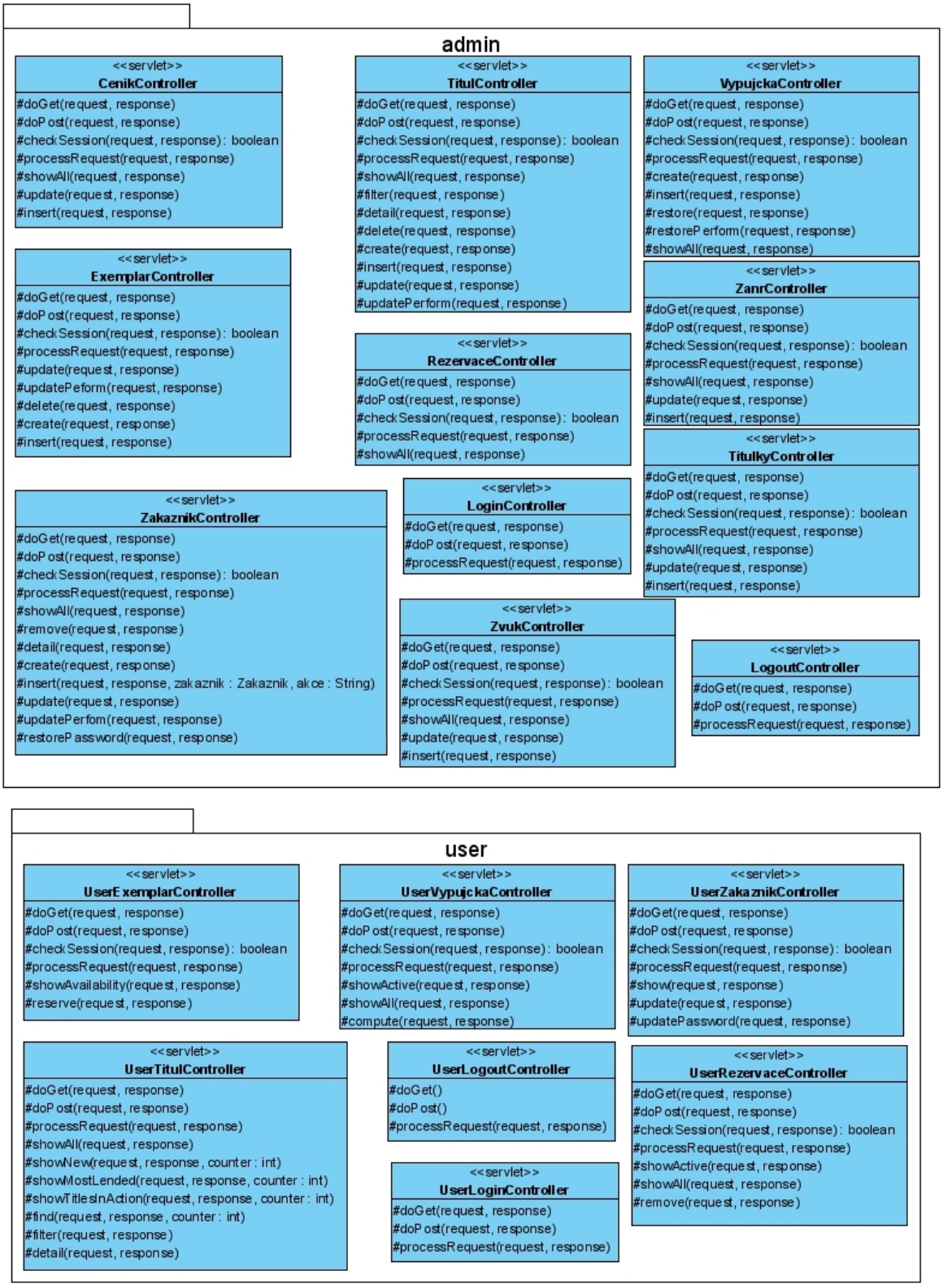
Jak již bylo uvedeno dříve, tak vrstva Controller obsahuje většinu aplikační logiky a tvoří most mezi perzistentními třídami a prezentační vrstvou. Stejně jako v ostatních vrstvách jsem se i v této snažil dosáhnout maximální koheze a minimálního propojení. Toho jsem docílil díky tomu, že tato vrstva nemá žádný jiný přístupový bod k perzistentním objektů než příslušné DAO rozhraní a zároveň ani „nic neví“ o vrstvě prezentační. Pokud budeme například provádět výpis všech titulů, tak tato vrstva pomocí příslušné metody třídy DAO vrstvy získá kolekci titulů a provede pouze její předání prezentační vrstvě, která se postará o její zobrazení například ve formě tabulky.

Tyto třídy jsou opět téměř naprosto shodné pro obě verze aplikace. Jediným místem, kde se odlišují, je instancování příslušných DAO objektů. Tedy například v servletu `TitulController`, ve kterém potřebujeme pracovat s objekty, které implementují rozhraní `TitulDAO`, je jedinou odlišností právě instancování tohoto objektu:

```
TitulDAO titulDAO = new HibernateTitulDAO();     // Hibernate verze
TitulDAO titulDAO = new JDBCTitulDAO();          // JDBC verze
```

Pokud bychom prováděli přechod některé stávající aplikace na jinou technologii perzistence v praxi, mohly by třídy této vrstvy zůstat zcela nezměněny. Důvodem je to, že bychom po tomto přechodu zřejmě používali již pouze novou technologii a nebylo by tedy třeba více tříd, implementujících stejná rozhraní.

K detailům implementace této vrstvy se ještě jednou podrobněji vrátíme v příští kapitole. Přestože to není v diagramu architektury uvedeno, rozdělil jsem servlety do dvou sekcí (balíčků) podle toho, ve které části aplikace se nacházejí.



obrázek 10.5

Diagram servletů vrstvy Controller

10.7 Vrstva View

Je nejvyšší vrstvou rámce MVC, která má na starosti prezentaci. Typickými technologiemi, které se v případě J2EE aplikace v této vrstvě vyskytují, jsou zejména JavaServer Pages (JSP), knihovna JavaServer Pages Standard Tag Library (JSTL), HTML a související technologie, kterými jsou CSS a jazyk JavaScript. Prezentační vrstva je v obou verzích aplikace naprosto shodná bez jediné výjimky.

Vzhledem k tématu své práce se těmto technologiím nebudu podrobněji věnovat a zmíním se zejména o obecných zásadách, kterými jsem se v rámci implementace této vrstvy řídil. Vzhledu uživatelského rozhraní je věnována příloha 3, ve které se nachází několik ukázkových obrazovek.

10.7.1 Jednotný vzhled a přehlednost celé aplikace

10.7.1.1 Rozložení prvků

Na všech stránkách je stejné rozložení prvků. Každá stránka zaměstnanecké části aplikace je rozdělena na dvě části. První je vertikální menu umístěné na levé straně a druhou samotná pracovní plocha, kam jsou směrovány veškeré výstupy.

Všechny stránky uživatelské části obsahují logo, pod ním je umístěno horizontální menu obsahující operace dostupné i nepřihlášeným uživatelům, dále samotnou pracovní plochu a pod ní ještě v případě, že je přihlášen registrovaný uživatel, další menu.

10.7.1.2 Grafická jednotnost

Zaměstnanecká i uživatelská část aplikace má nadefinovaná barevná schémata. Tato schémata obsahují velmi omezený počet barev, jejichž úkolem je zpřehlednit aplikaci a nikoliv ji zkrášlovat. Stejně jako v případě rozložení prvků je toto schéma jiné v obou částech aplikace.
Veškeré definice týkající se rozložení prvků a barev v obou částech aplikace jsou umístěny v příslušných CSS souborech.

10.7.2 Reakce na omyly

Každý uživatel může ať již omylem, nebo díky zlému úmyslu někdy zadat aplikaci nekorektní vstupní hodnoty. Typickým příkladem může být vstupní políčko, do kterého má být zadána číselná hodnota (např.: rok vydání u titulu), do něhož uživatel zadá nějaký text. V takovém případě aplikace vypíše příslušné chybové hlášení. Dále aplikace samozřejmě provádí kontrolu zadání všech povinných vstupních hodnot. Veškerá chybová hlášení jsou vypisována červeným písmem, které není nikde jinde v aplikaci použito.

Speciálním případem je jedna chybová stránka, na kterou je provedeno přesměrování v případě, že dojde k nějakému vážnějšímu problému. Typický případ, kdy dojde k přesměrování na tuto stránku, je nedostupnost databázového serveru, pokus o neautorizovaný přístup apod.

11 Implementace a testování aplikace

V této kapitole se zaměříme, za pomocí fragmentů zdrojových kódů, na některé implementační aspekty aplikace v jednotlivých jejích vrstvách.

11.1 Vrstva Model – sekce DTO

Jak již bylo několikrát zmíněno, vrstva model je z pohledu zaměření této práce tou nejzásadnější. Z tohoto důvodu se i v této kapitole zaměříme především na ni a za pomoci fragmentů diagramu perzistentních tříd a ukázek zdrojového kódu se seznámíme s některými zajímavými aspekty aplikace a především upozorníme na výhody, které nám Hibernate poskytne při implementaci metod rozhraní DAO.

11.1.1 Implementace kolekcí, dědičnosti a vztahů m:n

Na obrázku 11.1, který je uveden na následující stránce, je fragment diagramu perzistentních tříd (obrázek 10.2), jehož „centrálním bodem“ je třída titul. V této části diagramu se vyskytuje několik zajímavých aspektů, na které se podrobněji zaměříme.

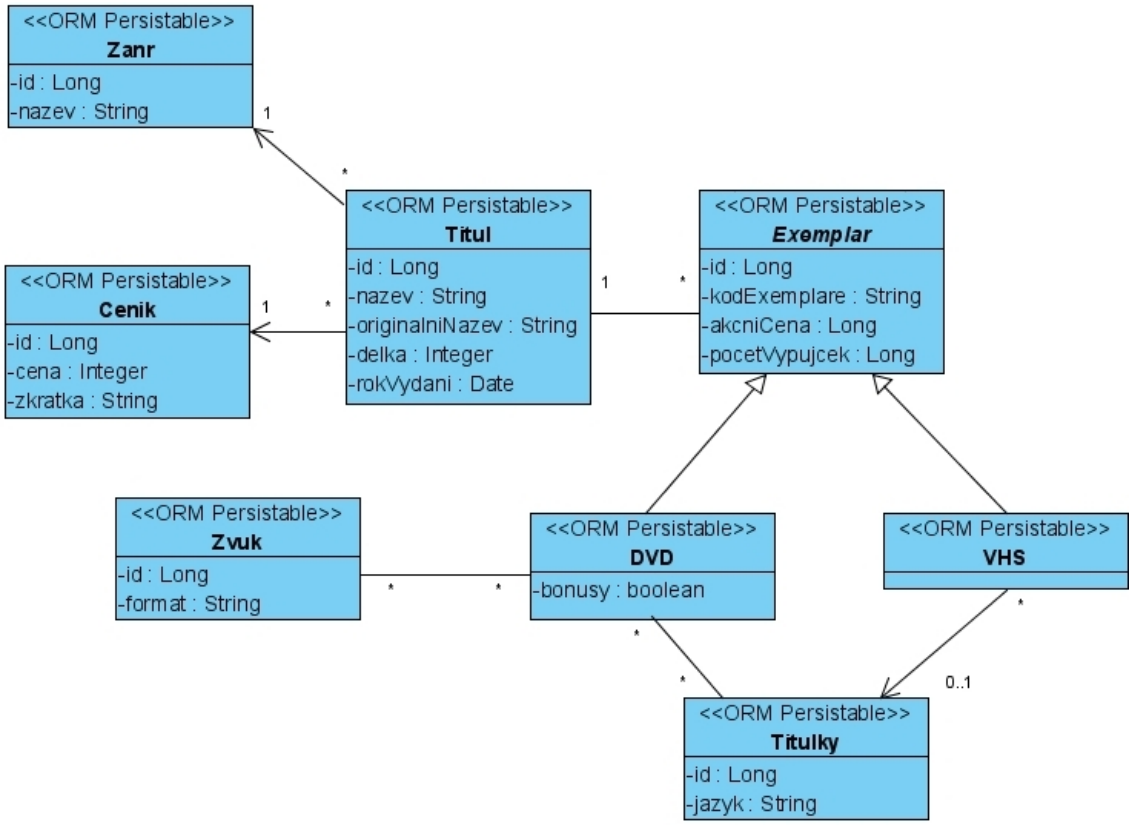
Prvním z nich je asociace mezi třídou *Titul* a abstraktní třídou *Exemplar*. Zajímavostí této asociace není její kardinalita, která není ničím výjimečná, ale její navigovatelnost, která je v tomto případě obousměrná. Z této skutečnosti při implementaci vyplývá potřeba atributu typu kolekce ve třídě *Titul*.

Dalším zajímavým aspektem této části diagramu perzistentních tříd je vztah dědičnosti mezi abstraktní třídou *Exemplar* a jejími potomky *DVD* a *VHS*. Pokud se nyní oprostíme od objektového pohledu na aplikaci a zamyslíme se nad tím, jak lze tuto dědičnost promítnout do schématu databáze, máme v tomto směru několik možných řešení.

- Prvním z nich je vytvoření jedné tabulky, která bude obsahovat všechny atributy třídy *Exemplar* a všechny atributy všech tříd (*DVD* a *VHS*), které z této třídy dědí. Tento přístup není příliš vhodný zejména v případě, že má velký počet potomků. Důvodem je to, že konkrétní objekt je typu vždy jen jednoho z těchto potomků a sloupce tabulky, které budou obsahovat atributy ostatních potomků, budou mít hodnoty `null` a výsledná tabulka bude velmi řídká. Dále je při tomto řešení vhodné zavést další sloupec sloužící jako diskriminátor. Velká výhoda tohoto přístupu však spočívá ve vysoké rychlosti dotazů, protože při dotazování není třeba spojovat data z více tabulek.
- Druhým přístupem je strategie, která je založena na tom, že vytvoříme tolik tabulek, kolik máme potomků dané třídy (v tomto případě *Exemplar*). Nevýhodou tohoto přístupu je, že všechny tabulky konkrétních tříd musejí obsahovat všechny atributy nadtřídy.

- Posledním přístupem je vytvoření jedné tabulky pro rodičovskou třídu a dalších tabulek pro každou ze tříd potomků. Nevýhodou této strategie je získání většího počtu tabulek, které musíme při dotazování spojovat. Naopak velkou výhodou je odstranění redundance atributů z předchozí varianty a odstranění množství atributů (z nichž některé jsou vždy `null`) z první varianty (použití jedné tabulky). Obecně lze říci, že tato strategie je zpravidla nejlepší a ve většině případů nejvýhodnější. Velká výhoda spočívá v jakési „rezervě“ do budoucna. Pokud se v době provozu aplikace vyskytne potřeba přidání dalších tříd potomků, není to v případě tohoto přístupu žádným problémem.

Posledním zajímavým druhem vztahu, který se v této části digramu vyskytuje, jsou dvě obousměrné asociace s kardinalitou m:n mezi třídou *DVD* a třídami *Titulky* a *Zvuk*. Tento druh vztahu díky jeho obousměrnosti vyžaduje v obou třídách, mezi kterými se nachází, existenci atributu typu kolekce.



obrázek 11.1

Obousměrné asociace, vztahy s kardinalitou m:n a dědičnost v diagramu perzistentních tříd

11.1.1.1 Perzistentní třídy

Nyní se podíváme, jak se výše zmíněné druhy vztahů promítnou do implementace perzistentních tříd a jak vypadají příslušné anotace pro Hibernate. Dále využijeme těchto fragmentů zdrojových kódů

k ukázkám dalších zajímavých možností anotací. Jak bylo již uvedeno v předchozí kapitole, tak perzistentní třídy jsou z pohledu jazyka Java shodné pro obě verze aplikace a jediným, čím se od sebe odlišují, je právě přítomnost anotací v Hibernate verzi.

Obousměrná asociace 1:n mezi *Titul* a *Exemplar*

Část zdrojového kódu třídy *Titul*, která je uvedena na následující stránce obsahuje několik (v tomto výpisu tučně uvedených) zajímavých anotací.

- První anotace, která se týká celé třídy, je zajímavá v tom smyslu, že definuje integritní omezení **unique** na kombinaci atributů *nazev* a *rok_vydani*, které bude kontrolováno databází.
- Další anotací je definice atributu *id*, který bude sloužit jako primární klíč dané tabulky generovaný databázovým serverem
- Dále je zde uvedena ukázka způsobu zamapování jednoduchých atributů (v tomto případě atributu typu `Date` *rokVydani)*. Anotace **`@Column`** je volitelnou a pokud ji neuvedeme, Hibernate samo vytvoří v tabulce databáze sloupec, který se bude jmenovat stejně jako atribut. Osobně vždy raději provedu „ruční“ nastavení tohoto názvu. Důvodem je to, že mám zavedené rozdílné konvence pro spojování těchto víceslovných názvů v případě Javy a v případě pojmenování příslušných sloupců, abych je na první pohled odlišil. Dále je zde provedeno nastavení **`nullable = false`**, které zajistí, že příslušný sloupec tabulky databáze bude mít nastaveno integritní omezení `NOT NULL`.
- Další uvedená anotace (**`@ManyToOne`**) řeší jednosměrnou asociaci 1:n mezi třídami *Titul* a *Zanr*. Z pohledu databáze je tato asociace reprezentována cizím klíčem v tabulce *Titul*, který referuje primární klíč tabulky *Zanr*. Je třeba poznamenat, že pro zachycení této asociace plně postačuje tato jedna anotace. Následující anotace **`@JoinColumn`** má stejný význam jako anotace **`@Column`** z minulého odstavce. Dále je zde uvedena anotace **`@org.hibernate.annotations.OnDelete`**, pomocí které sdělujeme Hibernate, že si přejeme při generování schématu databáze tomuto cizímu klíčí nastavit akci `on delete` na `cascade`.
- Posledním a z našeho pohledu nejzajímavějším druhem vztahu je obousměrná asociace kardinality 1:n. K jejímu namapování slouží anotace **`@OneToMany`**, ve které máme dále nastaveno kaskádní šíření všech druhů operací (`Merge`, `Persist`, `Refresh` a `Remove`) do všech asociovaných entit, inverzní atribut (atribut *titul* ve třídě *Exemplar*, který je zde zamapovaný **`@ManyToOne`** anotací) a na závěr máme nastavenu pro tuto kolekci „líné“ (**`fetch=FetchType.LAZY`**) načítání jejího obsahu. Tento druh načítání obsahu kolekce je velice užitečný pro zvýšení výkonnosti aplikace. V praxi se líné načítání projevuje tím, že pokud provedeme dotaz na objekty třídy *Titul*, nedojde k načítání objektů z této kolekce

(z pohledu SQL spojování tabulky *Titul* s tabulkou *Exemplar* a její spojování s tabulkami *DVD* a *VHS*), dokud k této kolekci nepřistoupíme a nezačneme načítat její obsah. O podrobnostech „líného“ načítání se zmíním v následující podkapitole, kde budou uvedeny ukázky implementace některých metod tříd DAO.

```
@Entity
@Table(name = "TITUL",
        uniqueConstraints =
        {@UniqueConstraint(columnNames={"NAZEV","ROK_VYDANI"})}
)
public class Titul {
      @Id @GeneratedValue
      @Column(name = "ID")
      private Long id;

      @Column(name = "ROK_VYDANI", nullable = false)
      private Date rokVydani = new Date();
      // namapování dalších atributů třídy Titul

      @ManyToOne
      @JoinColumn(name = "ZANR_ID", nullable = false)
      @org.hibernate.annotations.OnDelete(action =
            org.hibernate.annotations.OnDeleteAction.CASCADE)
      private Zanr zanr;
      // analogicky je provedeno mapování na asociace na třídu Cenik


      @OneToMany(cascade=CascadeType.ALL, mappedBy = "titul",
                  fetch=FetchType.LAZY)
      private List<Exemplar> exemplar = new ArrayList<Exemplar>();
```

Dědičnost mezi třídou *Exemplar* a jejími potomky *DVD* a *VHS*

Pro vyřešení tohoto vztahu dědičnosti byla použita strategie jedné tabulky pro rodičovskou třídu a dále zvláštních tabulek pro třídu každého z potomků, tak jak jsme si tuto strategii definovali v kapitole 11.1.1. V našem případě si tedy přejeme, aby schéma databáze vypadalo tak, že budeme mít tabulku *Exemplar*, která bude obsahovat informace společné všem druhům exemplářů a dále

tabulky *VHS* a *DVD*, které budou kromě informací specifických pro tyto druhy exemplářů obsahovat cizí klíč do tabulky *Exemplar*, který bude pro tyto tabulky současně i klíčem primárním. Následuje opět ukázka tříd a anotací, které nám požadované chování zajistí. Důležité anotace jsou opět tučně zvýrazněny.

Nastavení této strategie řešení dědičnosti je velmi jednoduché. Jedinou anotací, kterou je třeba použít, je anotace **`@Inheritance`** v rodičovské třídě *Exemplar*, pomocí které určíme požadovanou strategii řešení dědičnosti:

```
@Entity
@Table(name = "EXEMPLAR")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Exemplar {

      @Id @GeneratedValue
      @Column(name = "ID")
      private Long id;
      // další atributy a jejich mapování
```

Ve třídách, které z této třídy dědím, není již potřeba v tomto směru provádět žádné další speciální nastavování. Vše je z pohledu Hibernate v tomto směru určeno ve třídě *Exemplar* a není třeba označovat tuto třídu jako potomka nějakým speciálním způsobem zvlášť pro Hibernate. K tomuto účelu postačí „klasická“ dědičnost jazyk Java, označená klíčovým slovem `extends`.

```
@Entity
@Table(name = "VHS")
public class VHS extends Exemplar {
      @ManyToOne
      @JoinColumn(name = "TITULKY_ID", nullable = true)
      @org.hibernate.annotations.OnDelete(action =
            org.hibernate.annotations.OnDeleteAction.CASCADE)
      private Titulky titulky;
```

Jak je vidět na tomto fragmentu třídy *VHS*, tak jediným atributem, který obsahuje, jsou titulky. Přítomnost tohoto atributu je způsobena asociací mezi třídami *VHS* a *Titulky*. K zamapování je zde použito již dobře známých anotací, které mají zcela stejný význam jako u dříve vysvětlené asociace mezi třídami *Titul* a *Zanr*.

Obousměrné asociace s kardinalitou m:n mezi třídou *DVD* a třídami *Zvuk* a *Titulky*

Vztah m:n mezi entitami (ať už jedno nebo obousměrný) je zapotřebí v databázi vyřešit vazební tabulkou. Podívejme se proto na následující ukázku namapování třídy *DVD* a dalších tříd, které nám požadovanou funkčnost zajistí.

```
@Entity
@Table(name = "DVD")
public class DVD extends Exemplar {
      @Column(name = "BONUSY", nullable = true)
      private Boolean bonusy;

      @ManyToMany(cascade = {CascadeType.PERSIST,
                  CascadeType.MERGE}, fetch=FetchType.LAZY)
      @JoinTable(
                  name = "TITULKY_DVD",
                  joinColumns = {@JoinColumn(name = "DVD_ID")},
                  inverseJoinColumns = {@JoinColumn(name =
                                        "TITULKY_ID")}
      )
      private Set<Titulky> titulky = new HashSet<Titulky>();

      // anotace jsou zde analogické jako u předchozí kolekce
      private Set<Zvuk> zvuky = new HashSet<Zvuk>();
```

Třída *DVD* obsahuje kromě jednoho atributu typu Boolean, jehož mapovaní je jasné, dva atributy kolekcí, které vyplývají z m:n asociací. Pro základní namapování by zcela postačilo uvést pouze **`@ManyToMany`** anotaci a Hibernate by samo vytvořilo vazební tabulku s oběma potřebnými atributy. U této anotace je opět nastaveno „líné“ načítání a kaskádní šíření některých druhů operací, jak ho již známe z příkladu asociace mezi třídou *Titul* a *Exemplar*. Anotací, která je zde „nová“, je **`@JoinTable`**, která zde slouží pouze k nastavení jména vazební tabulky a jmen jejích jednotlivých sloupců.

Anotace u kolekce *zvuky* je v tomto výpisu vypuštěna, neboť její zamapování je analogické kolekci *titulky*.

Vzhledem k tomu, že tyto asociace jsou obousměrné, je třeba tuto skutečnost sdělit pomocí mapování i Hibernate. Na následujícím výpisu můžeme vidět, jak vypadá mapování inverzní strany této asociace u třídy *Titulky*.

```
@Entity
@Table(name = "TITULKY")
```

```
public class Titulky {
      @Id @GeneratedValue
      @Column(name = "ID")
      private Long id;

      @Column(name = "JAZYK", nullable = false,
              unique = true, length = 50)
      private String jazyk;

      @ManyToMany(mappedBy = "titulky", fetch=FetchType.EAGER,
                  cascade = {CascadeType.PERSIST, CascadeType.MERGE})
      private Set<DVD> dvds = new HashSet<DVD>();
```

Jak je z výpisu zřejmé, tak na opačné straně asociace není již potřeba uvádět žádné informace týkající se vazební tabulky. Je zde však třeba uvést název atributu druhé strany asociace pomocí **`mappedBy`**. S ostatními nastaveními jsme se seznámili již v předchozích příkladech. Na této straně asociace je načítání prováděno pomocí strategie ***`EAGER`***, což je pravý opak lazy načítání. Tato strategie (do češtiny překládána jako „hltavé načítaní“) funguje tak, že pokud bychom provedli dotaz na nějaký objekt třídy *Titulky*, dojde k okamžitému načítání objektů této kolekce, na rozdíl od lazy, kdy by Hibernate tento dotaz provedlo až v momentě přístupu do kolekce.

Analogickým způsobem je opět provedeno zamapování třídy *Zvuk*, tudíž nemá význam zde provádět výpis jejího kódu.

Na závěr této podkapitoly je třeba upozornit na dva druhy anotací, které se v ní vyskytly. Kromě jedné jsou všechny součástí standardního API jazyka Java, Java Persistence API (JPA), které rámec Hibernate implementuje. Jedinou zde použitou anotací, která není v JPA, ale patří mezi specifické anotace Hibernate, je **`@org.hibernate.annotations.OnDelete`**. Důvodem, proč je u této anotace použito celé jméno včetně balíku (místo toho, aby byl použit jeho `import`), je následující obecně udržovaná konvence. Anotace, které patří do JPA nejsou uváděny celým jménem (je prováděn `import javax.persistence.*`) a naopak speciální anotace Hibernate ano. Výsledem toho je snadná orientace v kódu, kdy na první pohled může rozlišit o jaký druh anotace se jedná. Tato konvence se může na první pohled zdát zbytečnou, ale v případě přechodu na jiný rámec pro perzistenci by jistě mohla ušetřit dost času a problémů.

11.2 Vrstva Model – sekce DAO

Poté, co jsme se podrobně seznámili s některými třídami aplikace, podrobnostmi jejich mapování a odpovídajícími tabulkami databáze, na které se tyto třídy se tyto třídy mapují, můžeme přistoupit k porovnání tříd, které implementují příslušná rozhraní DAO.

Jak již bylo dříve uvedeno, tak tyto třídy, které implementují jednotlivá DAO rozhraní, jsou jediným místem, kde jsou obě verze zcela odlišné.

Vzhledem k tomu, že jsme se podrobně seznámili s třídou *Titul* a odpovídajícími třídami, na které má tato třída asociace, budeme se i nadále věnovat právě jim. Dalším důvodem pro věnování se těmto třídám je to, že na nich lze velmi dobře ilustrovat rozdíly v práci s Hibernate a JDBC.

11.2.1 Dotazování se nad objekty

Nejprve se zaměříme na dotazování se nad perzistentními objekty. V následujících dvou podkapitolách se podíváme na způsoby implementace metody `findById(Long id)` z rozhraní `TitulDAO`. Úkolem této metody je podle zadaného číselného identifikátoru (`id`) získat kompletní informace o daném titulu a dále kompletní informace o všech jeho exemplářích.

Vzhledem k tomu, že třída *Titul* má asociace se třídami *Zanr*, *Cenik* a *Exemplar*, je potřeba při dotazu provést získání příslušných objektů těchto tříd. Samotná třída *Exemplar* je však pouze abstraktní třídou, ze které dále dědí třídy *VHS* a *DVD*. Třída *Titul* může mít více exemplářů (tedy objektů tříd *VHS* nebo *DVD*). Je tedy třeba získat i objekty všech těchto exemplářů. Exempláře třídy *DVD* mají ještě dále asociace se třídami *Titulky* a *Zvuk* a exempláře *VHS* asociace se třídou *Titulky*.

Z tohoto popisu je zřejmé, že metoda `findById(Long id)` musí provést spojení poměrně velkého množství tabulek, aby získala všechna potřebná data z databáze.

11.2.1.1 Implementace v Hibernate

V této podkapitole se podíváme, jak lze v Hibernate naimplementovat výše zmíněnou metodu `findById(Long id)`. Díky tomu, že tato implementace se skládá z jednořádkového dotazu v jazyce HQL, můžeme si zde dovolit výpis celé metody.

```
public Titul findById(Long id) throws Exception {
      Titul titul = new Titul();
      Session session = HibernateUtil.getSessionFactory().
                                        openSession();

      Transaction tx = session.beginTransaction();
      try {
            titul = (Titul) session.createQuery("from Titul where
                  id = :id").setParameter("id", id).uniqueResult();
```

```
            tx.commit();
        } catch (Exception e) {
            throw e;
        } finally {
            // zde by obvykle bylo session.clear();
        }
        return titul;
}
```

Přestože je metoda velice jednoduchá, zmíníme zde dvě skutečnosti. První je samotný dotaz (v textu tučně zvýrazněný). Zde je ilustrováno použití metody **`setParameter("id", id)`**, která do HQL dotazu vloží identifikátor titulu, který byl metodě `findById(Long id)` předán jako parametr. Další zajímavost se nachází (tedy spíše nenachází) v sekci `finally`. Na tomto místě by mělo být volání metody `clear()` objektu `session`, tedy `session.clear();`. Příčinou neexistence tohoto volání je kolekce exemplářů. Jak jsme viděli v předchozí kapitole, tak tato kolekce má nastaveno „líné" načítání. Musíme tedy nechat otevřenou `session`, aby mohlo dojít později v případě potřeby k jejímu načtení. Pokud dojde k načtení této kolekce, je dále třeba načíst (pokud je to nutné) kolekce zvuků a titulků pro jednotlivé exempláře typu DVD.

K načítání těchto kolekcí dojde v případě této aplikace až v momentě, kdy v příslušné stránce JSP provádíme zobrazení atributů výsledného objektu třídy Titul a přistoupíme k nim. Hibernate má příslušné dotazy připravené a jakmile je jejich provedení potřeba, tak je „na pozadí" provede. Díky tomu na první pohled ani nepoznáme, že tyto kolekce původně obsahovaly pouze `null` hodnoty.

11.2.1.2 Implementace v JDBC

Naimplementovat příslušnou metodu v JDBC je podstatně složitější. Dotaz, který byl v HQL dlouhý jeden řádek, zde díky množství tabulek, které je třeba spojit, velice naroste. Díky velikosti tohoto dotazu a zejména pak náročnosti zpracování výsledné tabulky, kterou dotaz vrátí, zde není prostor pro výpis celé metody jako to bylo v případě Hibernate. Omezíme se proto pouze na ukázku příslušného SQL dotazu, který by mohl vypadat nějak takto:

```
SELECT * FROM titul
      join zanr on (titul.zanr_id = zanr.id)
      join cenik on (titul.cenik_id = cenik.id)
      join exemplar on (exemplar.titul_id = titul.id)

      left join dvd on (dvd.id = exemplar.id)
         left join titulky_dvd on (titulky_dvd.dvd_id = dvd.id)
         left join titulky  on (titulky.id = titulky_dvd.titulky_id)
         left join zvuk_dvd on (zvuk_dvd.dvd_id = dvd.id)
```

```
        left join zvuk on (zvuk.id = zvuk_dvd.zvuk_id)

     left  join vhs on (vhs.id = exemplar.id)
        left join titulky  as tVHS on (titulky.id = vhs.titulky_id)
where titul.id = ?;
```

Jak je vidět, tak příslušný dotaz je opravdu poměrně rozsáhlý, což však není tím největším problémem. Samozřejmě můžeme tento dotaz rozdělit na několik částí a provést jej postupně. Skutečným problémem je procházení výsledného objektu `ResultSet`, který tímto dotazem vznikne, vytváření příslušných objektů, jejich spojování do kolekcí apod. Problémem to není z důvodu nějaké algoritmické složitosti, ale spíše kvůli počtu sloupců výsledné tabulky, kdy je třeba dávat dobrý pozor, ve kterém sloupci se nachází jaký atribut, neboť je pak poměrně pracné dohledávat případnou chybu.

11.2.1.3 Závěr

V předchozích dvou podkapitolách byla ilustrována síla Hibernate v porovnání s JDBC při dotazování se nad perzistentními objekty. Pokud používáme Hibernate a dotazovací jazyk HQL, je většina dotazů (v případě, že si nepřejeme nastavovat nějaké omezující podmínky, třídění apod.) naprosto triviální a velikostí nepřesahuje jeden řádek zdrojového kódu. To je umožněno pomocí mapovacích informací, podle kterých Hibernate „pozná“ jakým způsobem má vygenerovat odpovídající SQL dotaz, který je předán databázi.

Další velmi podstatnou výhodou je to, že výsledkem HQL dotazu je „hotový“ objekt, čímž odpadá velmi zdlouhavá a pracná činnost s procházením objektu `ResultSet`.

11.2.2 Ukládání objektů

V této podkapitole se zaměříme na to, jakým způsobem můžeme ukládat objekty. Opět se zde budeme věnovat dříve vysvětleným perzistentním třídám a podíváme se, jakým způsobem jsou ve vrstvě DAO naimplementovány metody potřebné pro případ použití *přidání titulu*. V diagramu případů použití (obrázek 9.1) je vidět, že tento případ použití závisí na případu použití *přidání exempláře*, ze kterého dále dědí případy použití *přidání exempláře VHS* a *přidání exempláře DVD*. Důvod této závislosti vyplývá z požadavků na aplikaci, kdy si nepřejeme, aby bylo možné uložit do systému titul, od kterého neexistuje žádný exemplář.

Odpovídající metoda, která obstarává ukládání exemplářů a titulů a všech jejich podrobností, je metoda `insert(Exemplar: exemplar)` rozhraní `ExemplarDAO`. Podívejme se tedy opět ve dvou podkapitolách, jak je tato metoda naimplementována třídami jednotlivých verzí aplikace.

11.2.2.1 Implementace v Hibernate

Implementace v Hibernate je opět velice jednoduchá. Díky tomu si můžeme opět dovolit vypsat celou tuto metodu a upozornit na dvě zajímavá místa v ní.

```
public void insert(Exemplar exemplar) throws Exception {
      Session session = HibernateUtil.getSessionFactory().
                                        openSession();
      Transaction tx = session.beginTransaction();

      try {
            session.save(exemplar);
            tx.commit();
      } catch (ConstraintViolationException e) {
            tx.rollback();
            throw e;
      } catch (Exception e) {
            throw e;
      } finally {
            session.close();
      }
}
```

Prvním zajímavým místem je metoda **`session.save(exemplar)`**, která provede samotné uložení objektu *Exemplar*. Zajímavé je toto místo z několika důvodů. Tím prvním je to, že se zde neprovádí explicitní uložení objektu *Titul*. Důvodem je to, že objekt třídy *Exemplar* má na *Titul* asociaci a Hibernate provede toto uložení automaticky. Pokud však již objekt *Titul* existuje (pokud provádíme pouze případ použití *přidání exempláře*), Hibernate opět „pozná“, že příslušný *Titul* existuje (atribut *titul* třídy *Exemplar* není `null`) a neprovádí jeho ukládání. Další zajímavostí je to, že třída *Exemplar*, jak jsme si již dříve vysvětlili, je pouze rodičovskou třídou a navíc abstraktní. Hibernate tedy musí zjistit, jaké ze tříd jejich potomků je předaný objekt a podle toho provést uložení do příslušných tabulek databáze.

Další zajímavostí je „odchytávání“ výjimky **`ConstraintViolationException`**. Je zde umístěno z toho důvodu, že tabulky *titul* a *exemplar* mají nastaveny na některé sloupce integritní omezení `unique`. Pokud k tomuto porušení nesprávným zadáním vstupních hodnot dojde, provedeme v této metodě částečné ošetření této výjimky pomocí vrácení transakce **`tx.rollback()`** a dále tuto výjimku propagujeme do metody, která tuto metoda zavolala. V tomto případě je to některý ze servletů vrstvy controller, který výjimku „odchytí“ a vytvoří objekt s chybovým hlášením, který se nakonec zobrazí v JSP stránce uživateli.

11.2.2.2 Implementace v JDBC

Implementace této metody v JDBC je opět poměrně dlouhá a musí obsahovat všechny kroky, které za nás dělá automaticky Hibernate. Proto se omezíme pouze na jejich popis.

1) Nejdřív je třeba otestovat, zda se přidává pouze exemplář nebo i titul. Otestujeme to tedy pomocí `exemplar.getTitul().getId()`
    - Pokud je výsledek `null`, provedeme příkaz `INSERT` do tabulky *titul* a zajistíme si pomocí `Statement.RETURN_GENERATED_KEYS` vrácení identifikátoru *id* tohoto nového záznamu, který získáme pomocí metody `getGeneratedKeys()`.
2) V dalším kroku je třeba provést příkaz `INSERT` do tabulky *exemplar* a opět zjistit *id* tohoto nového záznamu.
3) Nyní je třeba pomocí metody `exemplar.getClass().getSimpleName()` zjistit, o kterou ze tříd potomků třídy *Exemplar* se jedná.
    - Pokud jde o třídu *VHS*, provedeme příkaz do `INSERT` tabulky *vhs*.
    - Pokud jde o třídu *DVD*, připravíme si příkaz `INSERT` do tabulky *dvd*. Dále je třeba zjistit, kolik titulků má dané dvd nastaveno a připravit příslušný počet příkazů `INSERT` do tabulky *titulky_dvd*, která „řeší“ kardinalitu m:n mezi *titulky* a *dvd*. Stejným způsobem je třeba připravit příkazy `INSERT` do tabulky *zvuk_dvd*. Všechny tyto příkazy postupně pomocí `addBatch` řadíme „za sebe“ a na závěr provedeme jejich hromadné provedení pomocí `executeBatch()`.

Stejně jako v případě Hibernate řešení i zde kontrolujeme, zda nedošlo k porušení integrity. Pokud k němu dojde, musíme zkontrolovat, zda k tomu došlo při vložení prvního exempláře a pokud ano, provést odstranění titulu.

11.2.2.3 Závěr

V předchozích dvou podkapitolách jsem se seznámili s rozdíly při ukládání objektů v obou verzích aplikace. Opět je zde patrná velká síla a úspornost kódu při použití Hibernate, kde pomocí jedné metody dojde k uložení často velmi složité hierarchie objektů do databáze.

11.2.3 Aktualizace objektů

V této podkapitole se již jen velmi stručně zaměříme na aktualizaci objektů. Nebudeme zde již uvádět žádné ukázky zdrojového, neboť principy které byly zmíněny v předcházející podkapitole týkající se ukládání objektů plně platí i pro jejich aktualizaci. Pokud v aplikaci pracujeme s objektem třídy *Titul* a provedeme změnu některého z jeho atributu, nebo některého atributu kteréhokoliv objektu ke kterému můžeme pomocí asociací přistoupit, tak jediné, co je třeba pro aktualizaci tohoto objektu (a případně jakéhokoliv dalšího, který je „zanořen“ v hierarchii asociací) provést, je v případě Hibernate

zavolání metody `session.update()` a předání příslušného objektu jako parametr této metodě. Hibernate samo již zajistí, že se případná hierarchie tvořená asociacemi mezi objekty „projde“ a všechny změny všech objektů se korektně promítnou do databáze.

V případě JDBC je postup stejný jako v případě ukládání objektu. V tomto případě je třeba projít příslušnou hierarchii objektů, u kterých mohlo dojít ke změně a provést příkazy **UPDATE** do odpovídajících tabulek.

11.2.4 Mazání objektů

Poslední typickou operací, kterou je třeba s objekty provádět, je mazání. V následující podkapitole se podíváme na některé možné způsoby, kterými lze provádět mazání objektů.

Nejjednodušším případem je mazání objektu, na který není žádná asociace. Jinými slovy z pohledu databáze je uložen v tabulce, na kterou není žádná reference v podobě cizího klíče. Smazat takovýto objekt je velmi jednoduché jak v Hibernate, tak i v JDBC. V Hibernate nám na to poslouží metoda `session.delete()`, která zajistí vše potřebné, a v případě JDBC jednoduchý příkaz `DELETE`.

Tento případ, kde nemáme na objekt žádnou asociaci (potažmo cizí klíč na příslušnou tabulku), je však poměrně vzácným. Podívejme se tedy stručně na již dobře známou třídu *Titul* a metodu *remove(Long id)* rozhraní *TitulDAO*, která provede odstranění titulu podle jedinečného identifikátoru. Je třeba poznamenat, že způsoby implementace této metody v obou verzích aplikace, které zde budou nastíněny, nejsou v aplikaci použity. Důvody proč tomu tak je, budou vysvětleny na konci této podkapitoly.

Hibernate nám samozřejmě nabízí podporu i pro tento druh operací. Díky nastavené anotaci **`cascade=CascadeType.ALL`** u kolekce exemplářů ve třídě *Titul,* provede Hibernate automatické odstranění i těchto objektů a objektů, které jsou v hierarchii „pod nimi“.

V případě, že bychom chtěli takto naimplementovat tuto metodu v JDBC, museli bychom opět provádět „ruční“ mazání řádků ve všech odpovídajících tabulkách.

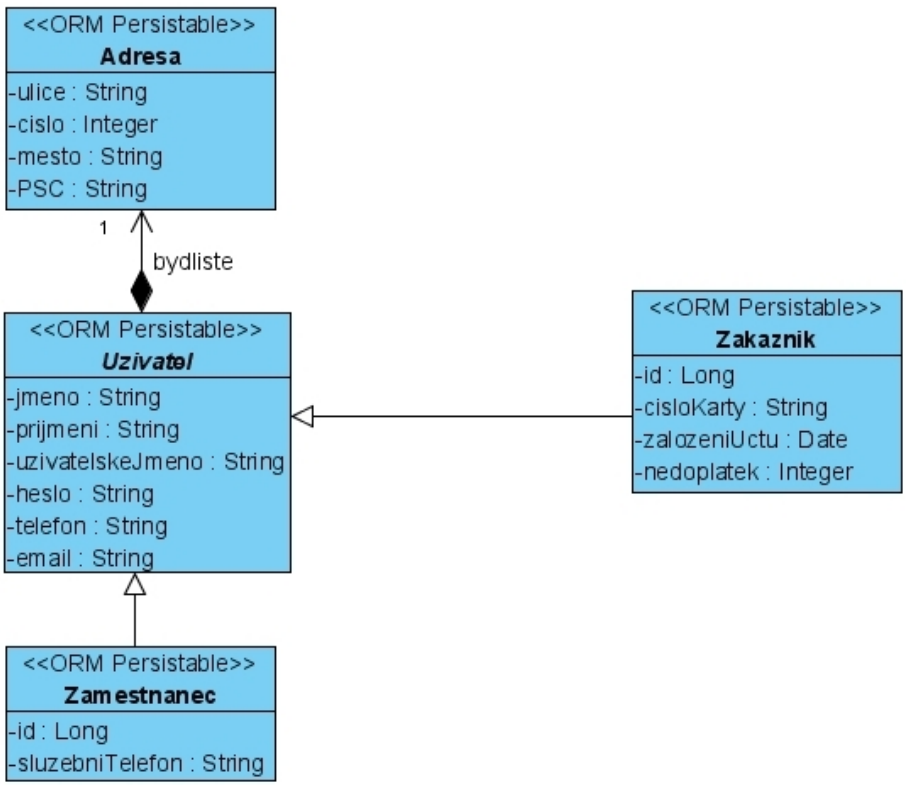
Jak jsem uvedl dříve, tak tento způsob mazání hierarchie objektů není v aplikaci použitý. Důvodem je to, že tuto funkcionalitu nám je schopen zajistit samotný databázový server pomocí nastavení akcí `ON DELETE CASCADE` u příslušných cizích klíčů. Tento způsob zajišťuje několik výhod:

1) **Zajištění integrity** – vždy je lepší ponechat zajištění integrity na databázovém serveru a ne na aplikaci. Důvodem je to, že si nikdy nemůžeme být jistí, zda některý uživatel nepřistoupí k databázi jiným způsobem (SQL konzole, jiná aplikace, která nebude integritu zajišťovat, apod.) než prostřednictvím naší aplikace.

2) **Zjednodušení dotazů** – díky kaskádnímu šíření mazání se zjednoduší implementace metod provádějících odstraňování objektů. U našeho příkladu třídy *Titul* postačí provést příkaz `DELETE` na tabulku *titul* a databázový server provede již mazání z tabulek *exemplar*, *dvd*, *vhs* a dalších sám. Díky tomu stačí provést příkaz `DELETE` pouze na tabulku, která stojí na vrcholu pomyslné hierarchie a o její promazání směrem dolů se již dále nemusíme starat.
3) **Rozdělení zátěže** – další výhodou tohoto přístupu je rozdělení zátěže mezi aplikačním a databázovým serverem.

11.3 Vrstva Model – sekce DTO (další vztahy)

V této podkapitole se ještě jednou vrátíme do sekce perzistentních objektů a podíváme se na některé další zajímavé druhy vztahů a způsoby jejich mapování. Opět zde vidíme fragment diagramu perzistentních tříd (obrázek 10.2), kterému se budeme v této části věnovat.



obrázek 11.2

Dědičnost a vztah kompozice

Prvním zajímavým druhem vztahu, který zde může vidět, je opět vztah dědičnosti. Věnuji se mu zde znovu z toho důvodu, že na tomto místě aplikace je pro jeho mapování použitá jiná strategie než v případě tříd *Exemplar*, *VHD* a *DVD*. Zde je požita strategie nazývaná *table per concrete class*,

která znamená, že vytvoříme tolik tabulek, kolik máme tříd potomků a každá z nich bude obsahovat i atributy z rodičovské třídy. V našem případě tedy dvě tabulky *Zamestnanec* a *Zákazník*.

Dalším zajímavým vztahem je vztah kompozice mezi abstraktní třídou *Uzivatel* a třídou *Adresa.* Pří vztahu kompozice je celek vždy plně zodpovědný za jednotlivé části a řídí jejich životní cyklus. Tato skutečnost se samozřejmě musí odrazit na návrhu a mapování třídy *Adresa*, jak dále uvidíme.

11.3.1 Implementace kompozice

V této podkapitole se podíváme na fragment kódu třídy adresa. Opět si všimneme tučně zvýrazněných řádků kódu, které budou dále vysvětleny.

```
@Embeddable
public class Adresa {
      @Column(name = "ULICE", length=30, nullable = false)
      private String ulice;
      // stejným způsobem jsou mapovány ostatní atributy
```

Jak je vidět, tak zamapování kompozice je velice jednoduché. Díky tomu, že životní cyklus objektu třídy Adresa je řízen objektem celku, nenachází se zde před třídou obvyklá anotace `@Entity`, která označuje to, že pro tuto třídy má být vytvořena tabulka databáze. S neexistencí anotace `@Entity` souvisí i neexistence jedinečného identifikátoru `id`.
Místo nich zde můžeme vidět anotaci **`@Embeddable`**, která objekty této třídu označuje za tzv. „vestavitelné“ do objektů jiných tříd.

11.3.2 Implementace dědičnosti a vložení třídy *Adresa*

V tomto fragmentu kódu vidíme, jakým způsobem lze vložit třídu *Adresa* do třídy *Uzivatel* a jak namapovat výše vysvětlený způsob řešení dědičnosti. Důležité anotace jsou opět tučně zvýrazněny.

```
@MappedSuperclass
public abstract class Uzivatel {

      @Column(name = "JMENO", length=40, nullable = false)
      private String jmeno;
      // stejným způsobem jsou zamapovány i další atributy.

      @Embedded
      private Adresa adresa;
```

Jak jsme si dříve již vysvětlili, tak při tomto způsobu řešení dědičnosti nebude pro abstraktní třídu *Uzivatel* vytvořena tabulka. Díky tomu se zde opět nenachází anotace `@Entity` ani jedinečný identifikátor `id`. Místo toho zde můžeme vidět anotaci **`@MappedSuperclass`**, která označuje tento druh řešení dědičnosti.

Dále je zde vidět způsob, jakým lze vložit třídu *Adresa* do této třídy a vyřešit tím vztah kompozice. K tomuto účelu slouží jednoduchá anotace **`@Embedded`**, která zařídí vše potřebné.

11.3.2.1 Implementace konkrétních tříd

V závěru této části se ještě krátce podívejme, jakým způsobem vypadá mapování ve třídě *Zákazník*.

```
@Entity
@Table(name = "ZAKAZNIK")
public class Zakaznik extends Uzivatel {

    @Id @GeneratedValue
    @Column(name = "ID")
    private Long id;

    @Column(name = "CISLO_KARTY", nullable = false, unique = true)
    private String cisloKarty;
    // stejným způsobem jsou zamapovány i další atributy.
```

Jak je vidět, tak v této třídě se již nevyskytují žádné speciální mapovací informace. Dostatečnou informací potřebnou pro to, aby Hibernate zjistilo, že tato třída dědí požadovanou strategií dědičnosti atributy třídy *Uzivatel* včetně „vnořené“ třídy *Adresa*, je obyčejné označení dědičnosti klíčovým slovem `extends` jazyka Java.

Vzhledem k tomu, že tato třída již bude mít svoji reprezentaci v databázi ve formě tabulky, jsou v ní uvedeny nám již dobře známe anotace **`@Entity`** a **`@Id`**.

Fragment kódu další třídy (*Zamestnanec*), která je také potomkem třídy *Uzivatel*, zde již uveden nebude, neboť je obdobný jako u zde uvedené třídy *Uzivatel*.

Touto podkapitolou končíme vysvětlení mapování perzistentních objektů. Vysvětlili jsem si zde, jakým způsobem lze provést namapování všech běžných vztahů, které se mohou v naší aplikaci vyskytnout.

Ostatním třídám aplikace, jejichž mapování zde nebylo podrobně vysvětleno, se již věnovat nebudeme. Důvodem je to, že se mezi nimi vyskytují vztahy s kardinalitou 1:n, jejichž mapování je velice jednoduché a bylo již dostatečně vysvětleno.

11.4 Vrstva Controller

V této podkapitole se seznámíme s tím, jakým způsobem jsou naimplementovány servlety vrstvy controller. Vzhledem k tomu, že metody těchto servletů obsahují většinu logiky aplikace, jsou poměrně dlouhé a není možné zde uvést jejich kompletní výpis. Omezíme se proto pouze na vysvětlení filozofie jednotlivých metod a ukázky důležitých částí zdrojového kódu.

Způsob implementace těchto servletů si vysvětlíme na konkrétním případu použití, kterým je opět *detail titulu*, ve kterém je také použita metoda `findById(Long id)` rozhraní `TitulDAO`, se kterou jsme se podrobně seznámili v předchozí podkapitole. Konkrétně se budeme věnovat metodě `detail(request,response)` servletu `TitulController` z balíku `admin`.

11.4.1 Kontrola existence session

Vzhledem k tomu, že všechny případy použití zaměstnanecké části aplikace vyžadují autorizovaný přístup, je třeba nejdříve provést kontrolu existence příslušné session proměnné.

Díky tomu metody `doGet(request,response)` a `doPost(request,response)` obsahuji pouze volání metody `checkSession(request,response)`.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
                     response) throws ServletException, IOException{
     checkSession(request, response);
}
```

Samotná metoda `checkSession` provede otestování existence session. Pokud session existuje, pokusí se z ní získat atribut *zamestnanec* třídy *Zamestnanec.* V případě, že takovýto atribut neexistuje, provede se vytvoření chybové objektu a přesměrování na speciální chybovou stránku `error.jsp`, kde se tento chybový objekt zobrazí. Pokud atribut existuje, pokračuje se voláním metody `processRequest(request,response)`. Následující výpis obsahuje příslušný fragment zdrojového kódu.

```
if (zamestnanec == null) {
      String destinationPage = "/admin/error.jsp";
      ApplicationError ae = new ApplicationError();
      ae.setZahlaviStranky("Neautorizovaný přístup");
      ae.setHlavniNadpis("Pokus o neautorizovaný přístup");
      ae.setPopisChyby("Pokoušíte se přistoupit do části aplikace,
                        ke které nemáte potřebné oprávnění");
      ae.setTextOdkazu("Zpět");
      ae.setAdresaOdkazu("../admin/index.jsp");
      request.setAttribute("ae", ae);
```

```
        RequestDispatcher dispatcher =  getServletContext().
                                getRequestDispatcher(destinationPage);
        dispatcher.forward(request, response);
} else {
        processRequest(request, response);
}
```

11.4.2 Metoda processRequest

Metoda `processRequest(request, response)` je jakýmsi „srdcem“ každého servletu. Jejím úkolem je zjistit z objektu `request` typ požadované operace a podle něj dále provést volání příslušné metody.

```
protected void processRequest(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException{

        String actionName = request.getParameter(ACTION_KEY);

        if (SHOW_ALL_ACTION.equals(actionName)) {
                showAll(request, response);
        }

        if (FILTER_ACTION.equals(actionName)) {
                filter(request, response);
        }


        if (DETAIL_ACTION.equals(actionName)) {
                detail(request, response);
        }

        // stejným způsobem jsou volány další metody
 }
```

Je třeba poznamenat, že konstanty (uvedené velikými písmeny) jsou nadeklarovány jako atributy příslušného servletu. Například pro akci detail vypadá tato deklarace následovně:

```
private static final String DETAIL_ACTION = "detail";
```

11.4.3 Metoda detail

Díky tomu, že metoda `detail` nevyžaduje žádné vstupní hodnoty, díky čemuž není potřeba provádět kontroly vstupů a také díky tomu, že jejím úkolem je pouze získat příslušný objekt z databáze, nenachází se zde žádné složité operace a můžeme si tedy dovolit uvést ji zde téměř celou.

```
protected void detail(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException{
     TitulDAO titulDAO = new HibernateTitulDAO();
     Titul titul = new Titul();

     Long id = Long.valueOf(request.
               getParameter(DETAIL_ACTION_ID)).longValue();
     String destinationPage = "/admin/showTitulDetail.jsp";

     if (id != null) {
          try {
               titul = titulDAO.findById(id);
               request.setAttribute("titul", titul);
          } catch (Exception e) {
               destinationPage = "/admin/error.jsp";
               ApplicationError ae = new ApplicationError();
               ae.setZahlaviStranky("Nedostupná databáze");
               ae.setHlavniNadpis("Databáze není dostupná");
               ae.setPopisChyby("Došlo k problémům při komunikaci s
                                 databází");
               ae.setTextOdkazu("Zpět");
               ae.setAdresaOdkazu("../admin/index.jsp");
               request.setAttribute("ae", ae);
          }
          // dále zde provedeme rozdělení kolekce exemplářů
          // získaného titulu na kolekci exemplářů VHS exemplářů
          // a kolekci exemplářů DVD

     RequestDispatcher dispatcher =  getServletContext().
               getRequestDispatcher(destinationPage);
     dispatcher.forward(request, response);
    }
```

Jak je vidět, tak tato metoda je velmi jednoduchá. Pokud se podaří získat objekt třídy *Titul* z databáze, provede se jeho nastavení na kontext. V opačném případě, pokud došlo k chybě databáze, se provede sestavení chybového objektu.

Na závěr této metody (ve výpisu již není uvedeno) se nachází ještě rozdělení kolekce exemplářů daného titulu na dvě kolekce podle druhů exemplářů.

V závěru této kapitoly si ukážeme rozdíl implementací těchto servletů v jednotlivých verzích aplikace. V každém servletu se liší jednotlivé metody obou verzí aplikace pouze v jediném řádku kódu, který provádí instancování objektu implementujícího příslušné DAO rozhraní. Jak jsme viděli, v metodě detail (request,response), tak toto instancování může vypadat buď:

- `TitulDAO titulDAO = new HibernateTitulDAO()` – Hibernate verze

nebo

- `TitulDAO titulDAO = new JDBCTitulDAO()` – JDBC verze

Díky tomu, že je instancování těchto objektů opravdu jediným místem vrstvy Controller, kde se verze aplikace odlišují, je zde dosaženo velké koheze a nízkého propojení, díky čemuž přechod na jiný způsob perzistence nepředstavuje pro tuto vrstvu žádný problém.

11.5 Vrstva View

Nyní se ještě stručně podívejme na způsob, jakým je naimplementována prezentační vrstva. Budeme zde pokračovat v případu použití *detail titulu* a navážeme na metodu `detail` z předchozí (11.4.3) podkapitoly. Jak jsme si uvedli, tak tato metoda získá objekt *Titul*, provede rozdělení jeho kolekce exemplářů na dvě (kolekce exemplářů *VHS* a *DVD*), vše potřebné nastaví na kontext a provede přesměrování na stránku `showTitulDetail.jsp`.

Nyní se tedy stručně podívejme na tuto stránku, která se skládá z několika částí běžných u JSP stránek.

11.5.1 Začátek stránky JSP

V úvodu stránky (ještě před jejím záhlavím) se nachází část importů, kde pomocí direktivy JPS `<%@ page` a jejího atributu `import` provádíme import potřebných tříd, provádíme načítání knihovny JSTL, nastavení kódování apod., jak je vidět na následujícím příkladu.

```
<%@ page import="java.util.List"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page language="java" contentType="text/html; charset=ISO-8859-2"
```

```
    pageEncoding="ISO-8859-2"%>
```

11.5.2 Záhlaví stránky

Záhlaví stránky obsahuje tagy jazyka HTML, které jsou v těchto místech obvyklé, jako například import souborů jazyka CSS, jazyka JavaScript a další běžné informace.

```
<head>
     <meta http-equiv="Content-Type" content="text/html;
           charset=ISO-8859-2">
     <title>Seznam titulů</title>
     <link rel="stylesheet" type="text/css" href="default.css">
     <script src="js/script.js"></script>
</head>
```

11.5.3 Tělo stránky

Samotné tělo stránky je v případě administrátorské části aplikace rozděleno tagy <DIV> na dvě části. První z nich je vertikální menu umístěné v levé části obrazovky a druhou samotná plocha, která vyplňuje zbylý prostor.

11.5.3.1 Menu

Menu je strukturováno do jednoduché tabulky a až na jednu výjimku obsahuje pouze tagy jazyka HTML. Touto jedinou výjimkou je jednoduchý scriptlet, který zjistí jméno přihlášeného zaměstnance a pomocí výrazu <%= jej vypíše.

```
<div class="menu">
     <table class="menuTable">
          <tr>
               <td><img src="img/table.png"></td>
               <td>
                    <a href="ZanrController?action=showAll">
                         Seznam žánrů
                    </a>
               </td>
          </tr>
          // další položky menu
          <tr>
               <td><img src="img/logout.gif"></td>
               <td>
                    <a href="LogoutController"">Logout</a>
```

```
<%    HttpSession mySession= request.getSession(false);
      Zamestnanec zamestnanec = (Zamestnanec)mySession.
                                getAttribute("zamestnanec");
%>
                  <%= zamestnanec.getUzivatelskeJmeno()%>
                 </td>
            </tr>
      </table>
</div>
```

11.5.3.2 Plocha

Na pracovní plochu se pak směrují samotné výpisy. V případě stránky `showTitulDetail.jsp` je zde třeba vypsat tři tabulky. První obsahuje obecné informace o titulu a další dvě pak provedou za pomoci tagů JSTL zobrazení kolekcí *listVHS* exemplářů a *listDVD* exemplářů (a kolekcí, které jsou v jednotlivých exemplářích DVD obsažené, jako jejich atributy – *titulky* a *zvuky*).

Celý zdrojový kód zde nelze pro svoji velikost uvést, proto se podíváme pouze na několik fragmentů. Důležité části jsou zde opět tučně zvýrazněny.

```
<div class="plocha">
<p class="actionTableHead">Detail titulu</p>
 <table class="detailTable" cellpadding="3">
      <tr>
            <th style="width: 130px;">Název:</th>
            <td style="width: 200px;">${titul.nazev}</td>
      </tr>
      // výpis dalších atributů
      ...
      // výpis kolekce DVD exemplářů
      <%
            List listDVD = (List)request.getAttribute("listDVD");
            if ((listDVD != null) && (listDVD.size() > 0)) {
      %>
      <p class="actionTableHead">Exempláře DVD</p>
      <table class="actionTable" cellpadding="3">
            <tr>
                  // hlavička tabulky
            </tr>
```

```
<c:forEach items='${listDVD}' var='dvd'>
<tr onMouseOver="this.className='in';"
    onMouseOut="this.className='out';">

     <td>${dvd.kodExemplare}</td>
     <td>
     <c:choose>
          <c:when test="${dvd.bonusy == false}">
               NE
          </c:when>
          <c:otherwise>
               ANO
          </c:otherwise>
     </c:choose>
     </td>
     // zobrazení kolekce titulků
     <c:forEach var="titulky" items="${dvd.titulky}"
        varStatus="status">
          <c:choose>
               <c:when test="${status.count == 1}">
                 ${titulky.jazyk}
               </c:when>
               <c:otherwise>
                | ${titulky.jazyk}
               </c:otherwise>
          </c:choose>
     </c:forEach>
     // zobrazení dalších atributů a kolekcí
</tr>
<table >
// za touto tabulkou se vyskytuje stejným způsobem sestavena
// tabulka vzniklá z kolekce exemplářů VHS
```

Na úplném závěru této podkapitoly je třeba zdůraznit, že díky dobrému návrhu pomocí MVC a DAO je vrstva View v obou verzích aplikace **naprosto shodná** a kód žádné z JSP stránek se mezi nimi neodlišuje ani v jediném znaku.

11.6 Testování

Testování aplikace jsem prováděl v několika fázích. Ještě před samotným zahájením implementace jsem provedl otestování databáze, jejíž schéma jsem vygeneroval pomocí nástroje `hbm2dll`. Zde jsem testoval zejména to, zda jsou dobře vytvořené cizí klíče a zda mají správně nastaveny referenční akce `ON DELETE`. Toto testování jsem prováděl tak, že jsem si připravil sadu testovacích dat, které jsem nahrál do databáze a prováděl na nich dotazy, které jsou pak prováděny aplikací.

Samotné testování obou verzí aplikace jsem prováděl v několika krocích, od testování jednotlivých metod během jejich implementace, až po testování aplikace jako celku. Toto testování jsem prováděl pouze inspekcí kódu, ladicími nástroji prostředí Eclipse a vizuálním porovnáním toho, co aplikace vrací a jak se chová, vzhledem k tomu, co je od ní očekáváno.

Pokud by měla být aplikace reálně nasazena do provozu, fáze testování by musela být výrazně delší a prováděna ve spolupráci s budoucími uživateli. Během takového způsobu testování by zcela jistě vyvstaly nové požadavky na její funkčnost a na úpravu uživatelského rozhraní. Dále by bylo zapotřebí před nasazením aplikace do reálného provozu otestovat aplikaci nějakou formou spustitelných testů, které by provedly korektnější ověření její funkčnosti. K tomuto účelu by bylo možné použít rámce JUnit, pomocí nějž lze poměrně jednoduchým způsobem potřebné testy naimplementovat.

Samostatnou částí fáze testování bylo „odladění“ designu stránek (tedy zejména souborů CSS) takovým způsobem, aby byly plně funkční ve všech běžně používaných prohlížečích.

Na závěr jsem provedl celkové otestování obou verzí aplikace. Všechny chyby, na které jsem narazil, jsem odstranil a danou část znovu podrobně otestoval.

Vývoj a finální testování jsem prováděl s následující konfigurací:

- Microsoft Windows XP
- JDK 1.6.0
- Apache Ant 1.7.0
- Hibernate 3.2
- MySQL 5.0.27
- MySQL Connector Java 5.0.4
- JBoss 4.0.5

Aplikaci jsem vyvíjel ve volně dostupném vývojovém prostředí Eclipse za pomoci volně dostupného modulu pro tvorbu J2EE aplikací WTP (Web Tools Project), který výrazně urychluje práci zejména tím, že automaticky vytváří soubory potřebné pro nasazení na aplikační server a pokud je aplikace spuštěna v „debug mode“, provádí její automatický „redeploy“ na server po každé změně kódu.

Prohlížeče, pro které bylo odladěno uživatelské prostředí:

- Mozilla Firefox 2.0.0.3
- Microsoft Internet Explorer 6.0
- Netscape Navigator 8.1.2
- Opera 9
- Galeon 1.2.7

Ve všech těchto prohlížečích bylo kompletně otestováno uživatelské rozhraní a aplikace je v nich plně funkční s minimálními odchylkami ve vzhledu.

Na závěr jsem provedl implementaci výkonnostních testů. Jedná se o jednoduchou sadu testovacích případů, jejichž cílem není otestování funkčnosti aplikace, ale porovnání obou verzí aplikace z hlediska rychlosti. K tomuto testování nebyl použitý žádný testovací rámec ani jiný nástroj. Tyto testy jsou běžné programy jazyka Java, jejichž výstup je směrován do konzole. Jejich úkolem je provést otestování tříd, které implementují jednotlivá rozhraní DAO a porovnat rozdíly v rychlosti provedení jednotlivých metod. Výsledkům těchto testů se věnuje následující, dvanáctá kapitola.

11.7 Závěrečné porovnání technologií

V této poslední podkapitole provedeme závěrečné a shrnující porovnání technologií JDBC a Hibernate z hlediska programátora a z hlediska jejich dopadů na vývoj projektu. Dále zde ještě jednou stručně shrneme rozdílnost implementací obou verzí aplikací a způsob, jakým lze minimalizovat dopad na aplikaci způsobený přechodem od jedné technologie pro perzistenci objektů ke druhé.

11.7.1 Hibernate vs. JDBC

Provést nějakým způsobem korektní porovnání těchto technologií je poměrně složité. Stejně jako nelze zodpovědět otázku, který rámec pro perzistenci je nejlepší, nelze zodpovědět ani to, zda je lepší použití Hibernate nebo JDBC. Vždy, když si tuto otázku položíme, je třeba se řádně zamyslet nad celým kontextem situace, ve které potřebuje řešit perzistenci objektů a podle ní se rozhodnout, který způsob pro nás v dané situaci bude výhodnější.

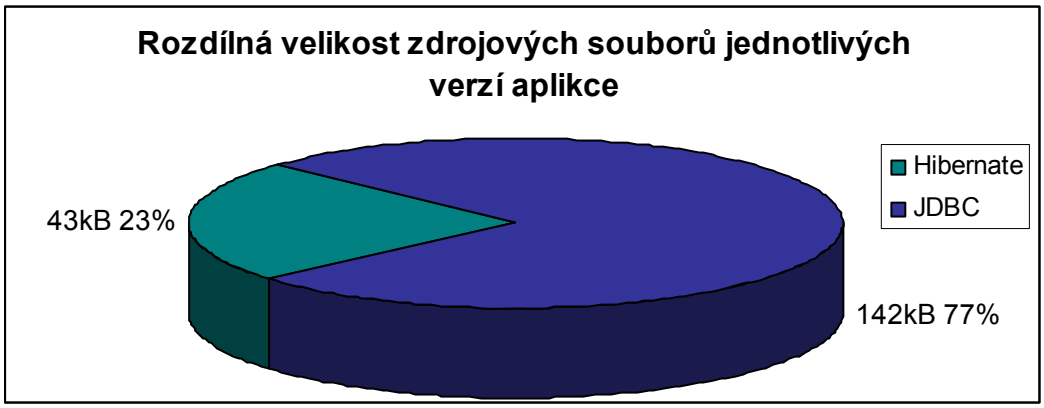
Pokud se například nacházíme v situaci, kdy ve velmi krátké době potřebujeme naimplementovat nějakou velice jednoduchou aplikaci, která bude přistupovat k datům uloženým v databázi, a neumíme pracovat ani s jednou z těchto technologií, bude pro nás lepší volbou JDBC. Důvodem je to, že naučení se základní práce s JDBC je otázkou maximálně několika desítek minut. K tomu, abychom dosáhli pomocí JDBC základní funkčnosti práce s databází, je zapotřebí znalosti pouze několika málo objektů, naučení se procházení ResultSetu a znalosti jazyka SQL, která by měla být pro programátora informačních systémů samozřejmostí. Oproti tomu naučení se základní práce s Hibernate je otázkou

minimálně několika dní a proniknutí více do hlubin jeho možností spíše několika týdnů, které si mnohdy nemůžeme dovolit.

Na druhou stranu, pokud budeme implementovat rozsáhlý informační systém obsahující stovky případů použití, složitou hierarchii tříd a odpovídajícím způsobem rozsáhlé schéma databáze, mělo by být Hibernate, nebo případně jiný rámec pro perzistenci, jasnou volbou. V tomto případě se zajisté vyplatí investovat několik dní nebo týdnů času do jeho studia, neboť se nám tento čas zcela jistě vrátí během implementace perzistence, jak jsem se přesvědčil i při implementaci této demonstrační aplikace.

Hlavní výhodou Hibernate a obecně všech ORM řešení je (kromě výkonnosti popsané v příští kapitole) to, že nás zbavuje „otrocké“ práce s perzistencí objektů, kdy je musíme při ukládání „rozbíjet“ na jednotlivé atributy a naopak při jejich získávání znovu, často velmi pracně díky asociacím a rozsáhlé hierarchii, skládat. Pokud použijeme Hibernate, tak většina běžných operací (ať už se jedná o dotaz v HQL, uložení nebo aktualizaci objektu) nezabírá zpravidla více než jeden řádek zdrojového kódu a jejich implementace je proto velice rychlá. Díky tomu nemusíme strávit tolik času prací na perzistentní vrstvě a můžeme se věnovat částem aplikace, které jsou pro potencionálního zákazníka zajímavější. Bohužel jsem si nevedl žádné záznamy o tom, jak dlouho mi trvalo naimplementování jednotlivých tříd vrstvy DAO. Troufám si však odhadnout, že implementace pomocí Hibernate zabrala zhruba 25% času implementace v JDBC.

Další výhodou, kterou tímto řešením získáme, je větší míra udržovatelnosti. Je vcelku zřejmé, že se lépe hledají a odstraňují chyby v metodě, která má 20 řádků, z nichž většina je získávání session, zahajování transakce a podobné záležitosti, kde chybu v podstatě udělat nelze, než v metodě, která má řádku 100 a obsahuje procházení ResultSetu, který má desítky sloupců. Pro ilustraci rozdílu počtu řádků zdrojového kódu v obou implementacích jsem vytvořil následující graf. Tento graf zobrazuje velikost všech tříd, které v jednotlivých verzích implementují rozhraní DAO.



obrázek 11.3

Rozdílná velikost zdrojových souborů

Zatímco v případě Hibernate má těchto 11 souborů v součtu 43kB, v případě JDBC je to 142kB, tedy více než trojnásobek. Z toho tedy vyplývá, že i počet řádků zdrojového kódu na naprogramování stejné funkčnosti je v této aplikaci při použití JDBC v průměru více než trojnásobný oproti Hibernate.

11.7.2 Shrnutí změn nutných k přechodu mezi technologiemi pro perzistenci

Díky použití MVC, návrhového vzoru DAO a důslednému dodržování principu „*vysoká koheze, nízké propojení*“ byl přechod mezi jednotlivými způsoby perzistence velice jednoduchý. Přesto, že jsem již v průběhu kapitoly upozorňoval v příslušných vrstvách na jednotlivé odlišnosti, provedu zde ještě jejich shrnutí. Jako první jsem naimplementoval aplikaci pomocí Hibernate a následně jsem provedl přechod na JDBC verzi. V praxi jsou obvyklé spíše opačné postupy, kdy se stávající aplikace předělávají z JDBC na nějaký rámec pro perzistenci. Z našeho pohledu však není příliš podstatné, která verze byla první. Rozdíly v jednotlivých vrstvách jsou tedy následující:

- **Vrstva Model – perzistentní objekty (sekce DTO)** – v JDBC verzi bylo pouze z těchto tříd provedeno odstranění anotací. Pokud bychom použili pro mapování XML, mohly by zůstat naprosto beze změny.
- **Vrstva Model – sekce DAO** – jednotlivá rozhraní zůstala samozřejmě naprosto nezměněna. Naopak třídy, které je implementují je třeba kompletně znovu naprogramovat.
- **Vrstva Controller – servlety** – v této vrstvě byly provedeny pouze „kosmetické“ úpravy v místech instancování DAO objektů, jak uvádí následující příklad.
    - **Hibernate:** `CenikDAO cenikDAO = new HibernateCenikDAO()`
    - **JDBC:** `CenikDAO cenikDAO = new HibernateCenikDAO()`

  Úprava tříd této vrstvy je tedy velmi jednoduchá a lze ji do značné míry zautomatizovat nástroji, kterými disponuje většina vývojových prostředí.
- **Vrstva View – stránky JSP** – v této vrstvě nebylo potřeba provést vůbec žádné změny, neboť je od vrstvy starající se o perzistenci zcela oddělena vrstvou Controller a přechod na jinou technologii perzistence se jí tím pádem vůbec nedotkne.

Jak je vidět, tak díky tomuto návrhu, který je značně flexibilní, není změna technologie pro perzistenci vůbec žádným problémem a jediná část aplikace, kterou je třeba znovu naimplementovat, jsou metody, které, ať už jakýmkoliv způsobem, komunikují s databázovým serverem.

Změna technologie pro perzistenci však není jedinou, která je v tomto směru možná. Díky tomu, že řídicí vrstva „nic neví“ o samotné prezentaci dat, nepředstavovala by žádný problém ani změna prezentační vrstvy na jinou technologii než jsou JSP.

12 Výkonnostní testy

V této předposlední kapitole se seznámíme s výsledky některých výkonnostních testů, které byly zmíněny v minulé kapitole. Jak již bylo uvedeno, jedná se o sadu jednotlivých testů, která je naimplementována jako klasická konzolová aplikace. Výstupem této aplikace je vždy čas potřebný na provedení příslušného testu.

Testování jednotlivých metod bylo prováděno pomocí cyklu, který měl 100, 500 nebo 1000 opakování (bylo tedy provedeno 100, 500 nebo 1000 dotazů). Každý test byl pro vyšší vypovídací hodnotu pětkrát zopakován a z naměřených časů byla vypočítána průměrná hodnota. Z těchto hodnot byly dále vytvořeny grafy. Z důvodů úspory místem zde nebudeme uvádět tabulky naměřených hodnot a zaměříme se pouze na grafy, které z nich byly vytvořeny.

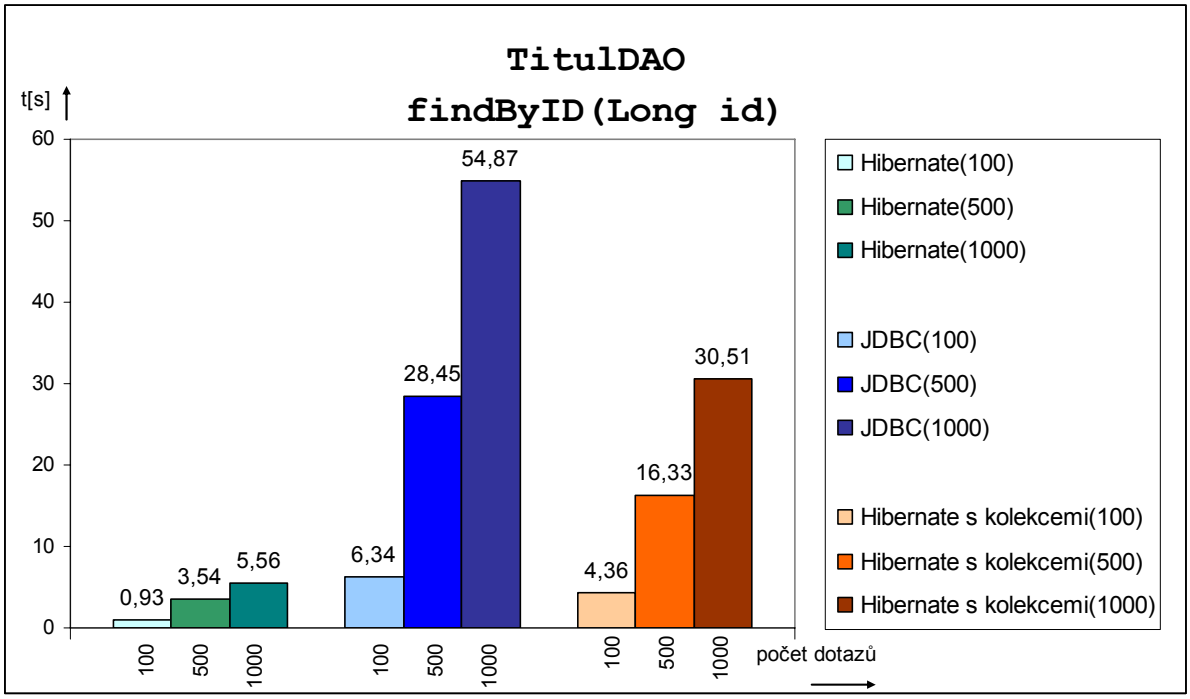
Testy byly rozděleny podle druhů dotazů jazyka SQL (`SELECT, INSERT,…`).

12.1 Testy Select

V této podkapitole se podíváme na výsledky testů několika metod z rozhraní DAO, jejichž úkolem je získat nějaká data z databáze (provádějí tedy dotaz SELECT resp. jeho obdobu v HQL).

12.1.1 Test metody findById(Long id) rozhraní TitulDAO

V následujícím obrázku, je výsledek testu nám již dobře známé metody `findById(Long id)`.



obrázek 12.1

Test metody findById(Long id) rozhraní TitulDAO

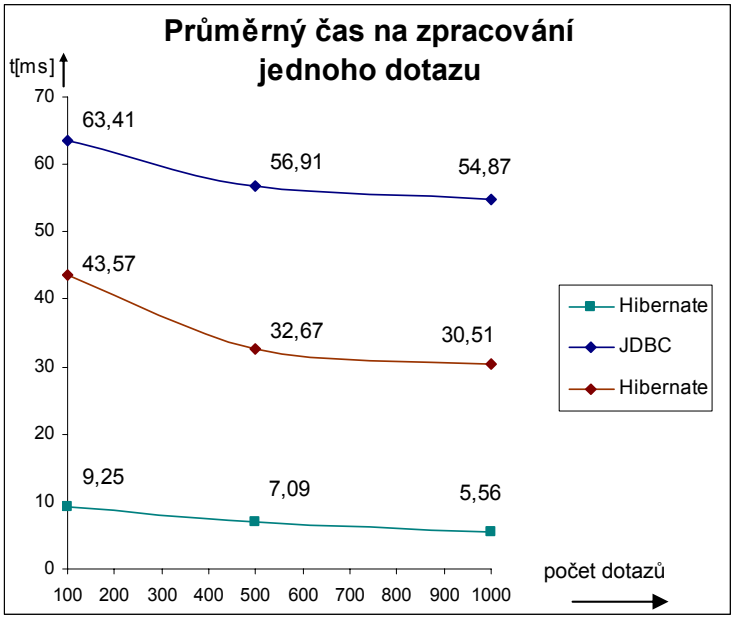
K předchozímu grafu je potřeba několik vysvětlujících komentářů. Jak jsme si uvedli dříve, tak úkolem metody `findById(Long id)` rozhraní `TitulDAO` je získání konkrétního titulu podle jeho `id`. Identifikátor, který jsem během testu této metodě předával, bylo náhorně vygenerované číslo v rozsahu 1..počet záznamů v tabulce *titul*. Dále jsme si uváděli, že Hibernate umožňuje použití „líného“ načítání kolekcí. Toho je v tomto případě využito, neboť získaný objekt *titul* obsahuje kolekci objektů *exemplář*, která v sobě u exemplářů *dvd* dále obsahuje další dvě kolekce (viz. kapitola 11.1).

Graf je tedy rozdělen na tyto tři části:

1) Výsledek testů implementace metody v Hibernate (bez vynuceného načtení kolekcí) – zelené sloupce.
2) Výsledek testů implementace metody pomocí JDBC – modré sloupce.
3) Výsledek testů implementace metody v Hibernate s vynucením načtení kolekcí. Toto načtení můžeme vynutit tím, že do příslušné kolekce vstoupíme – oranžové sloupce.

Každá část grafu obsahuje sloupce reprezentující časy provádění pro 100, 500 a 1000 dotazů. Z grafu je dobře patrné, že pokud neprovedeme přístup do kolekcí, je verze Hibernate zhruba desetinásobně rychlejší. Zpravidla však do kolekcí při použití této metody v aplikaci přistupujeme a proto je v tomto směru objektivnější pro porovnání třetí část grafu, kde je tento přístup a tím pádem načtení kolekcí provedeno. I zde je však vidět, že je Hibernate oproti JDBC výrazně rychlejší, což je zásluha zejména toho, že si vytváří cache pro načtené perzistentní objekty.

Důkazem toho je i následující graf, ve kterém je ilustrována sestupná tendence průměrného času na jeden dotaz, při zvyšování jejich počtu.



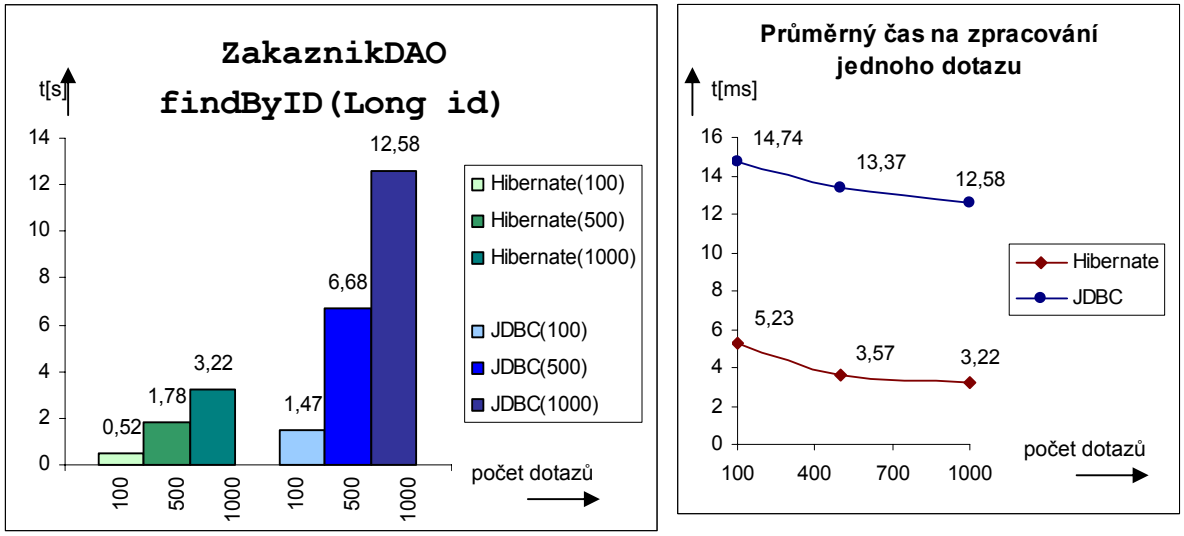
obrázek 12.2

Průměrný čas na zpracování jednoho dotazu

V tomto grafu je vidět, že u verze Hibernate bez načítání kolekcí klesl průměrný čas na jeden dotaz z 9,25ms u 100 dotazů na 5,56ms při provádění tisíce dotazů. Ve verzi s nuceným načítáním dotazů není tento rozdíl tak patrný. Příčinou by mohla být velikost cache, neboť objekty *titul* se všemi kolekcemi jsou podstatně větší.

12.1.2 Test metody findById(Long id) rozhraní ZakaznikDAO

Další metodou, jejíž výsledek testu zde uvedeme je opět metoda `findById()`, ale tentokráte z rozhraní `ZakaznikDAO`, jejímž úkolem je získat objekt zákazníka.



obrázek 12.3

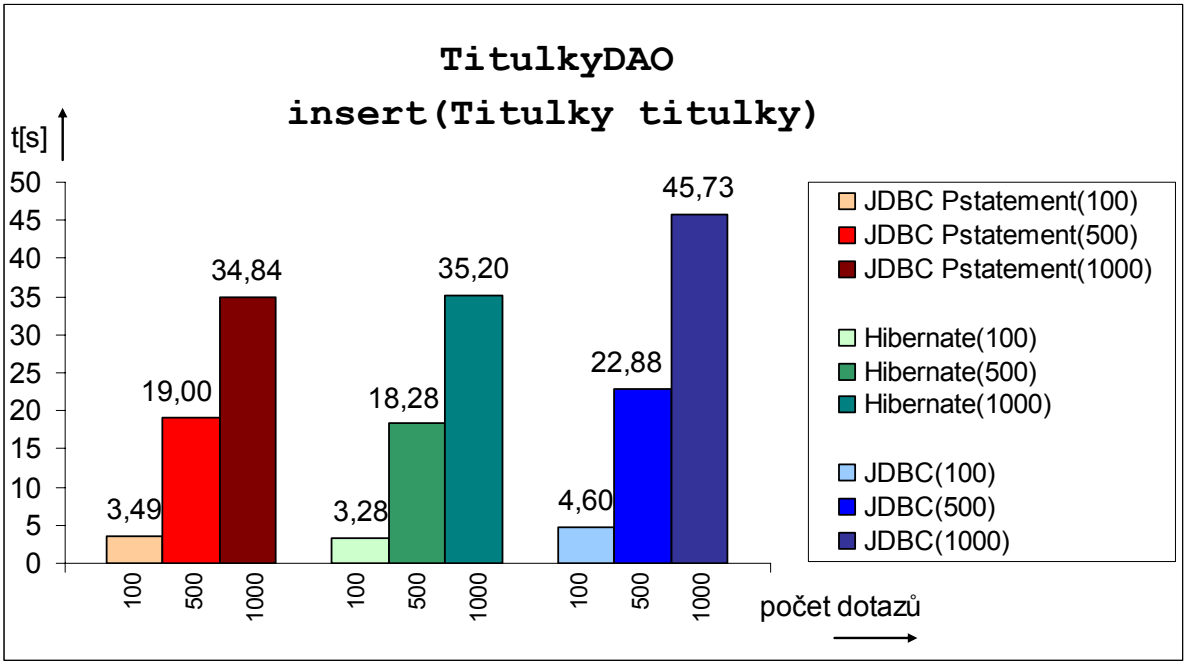
Test metody findById(Long id) rozhraní ZakaznikDAO a průměrný čas na zpracování jednoho dotazu

Z tohoto grafu je jasně vidět vyšší rychlost ve zpracování pomocí Hibernate oproti JDBC, která výrazně vzrůstá s počtem dotazů. Zatímco při zpracování sta dotazů je tento rozdíl zhruba 200%, u tisíce dotazů již činní zhruba 300%. Za tento výkonnostní nárůst je opět zodpovědná cache perzistentních objektů.

12.2 Testy Insert

Na následujícím grafu je zobrazen výsledek testů metody `insert(Titulky titulky)` rozhraní `TitulkyDAO`. Tento graf je opět rozdělen na tři části:

1) Výsledek testů implementace metody v JDBC s použitím předpřipravených dotazů pomocí objektu `PreparedStatement` – oranžové sloupce.
2) Výsledek testů implementace metody pomocí Hibernate – zelené sloupce.
3) Výsledek testů implementace metody v JDBC pomocí „obyčejného“ objektu `Statement` – modré sloupce.



obrázek 12.4

Test metody insert(Titulky titulky) rozhraní TitulkyDAO

Z tohoto grafu vyplývá několik zajímavých skutečností. První z nich je ta, že díky faktu, že nám v tomto případě cache objektů není ničím užitečná, neklesá s počtem dotazů čas na jejich zpracování, ale po přepočtu času na jeden dotaz zůstává téměř konstantní. Z tohoto důvodu zde nebudeme uvádět ani příslušný graf.

Další zajímavou skutečností je porovnání implementace metody v Hibernate a v JDBC pomocí předpřipravených dotazů. Jak je jasně z grafu vidět, tak jejich časy jsou v podstatě naprosto shodné. Je to mimo jiné důkazem toho, že jádro Hibernate důsledně dodržuje používání těchto dotazů. Pokud v JDBC použijeme „obyčejný“ objekt `Statement`, jsou výsledky testů o několik procent horší.

12.3 Ostatní testy

Dalšími testy, které jsem v aplikaci prováděl byly testy provádějící aktualizaci a odstraňování objektů. Z důvodů úspory místem zde již nebudou jejich výsledky uvedeny a budou pouze stručně popsány.

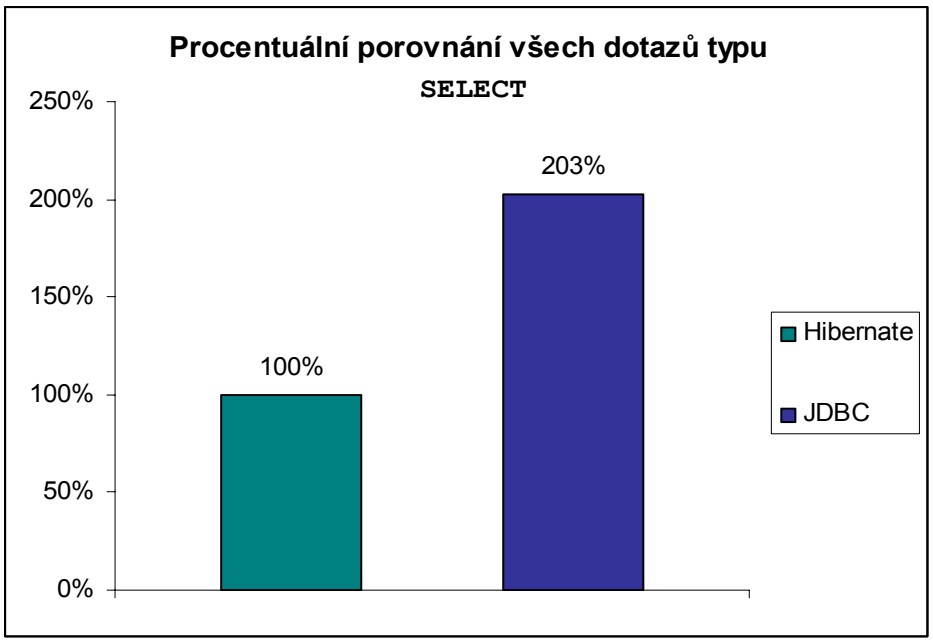
Výsledky testů update dopadly podobně jako tomu bylo u uvedeného výsledku testu insert (viz. 12.2). Pokud použijeme předpřipravené dotazy, obě verze metod jsou zhruba stejně rychlé. V případě nepoužití předpřipraveného dotazů, je JDBC verze vždy o několik procent pomalejší.

Dále jsem provedl výsledky testů metod, které provádějí odstraňování objektů. Vzhledem k tomu, že odstranění je vždy velmi jednoduchá operace (pokud se provede smazání např. objektu

titul, odstraňování dat ze souvisejících tabulek provede samotný databázový server díky nastavení `ON DELETE CASCADE` u cizích klíčů), jsou výsledky těchto testů shodné pro obě verze aplikace.

12.4 Závěr

Jak lze předpokládat, tak rychlost Hibernate je oproti JDBC výrazně vyšší při použití dotazů typu SELECT. Je to způsobeno především používáním cache perzistentních objektů. Na následujícím grafu je uvedeno procentuální porovnání všech provedených SELECT testů.



obrázek 12.5

Procentuální porovnání všech SELECT testů

Z výsledného grafu je patrné, že Hibernate je při dotazech typu `SELECT` v průměru zhruba dvakrát rychlejší než JDBC. Samozřejmě tento výsledek není v žádném případě nějakým vzorem, ke kterému vždy dochází. Pomocí velkého množství různých optimalizací a nastavení lze obě verze těchto aplikace vyladit a dosáhnout tak vyššího výkonu.

Hibernate je podstatně výkonnější, zejména pokud se provádí velké množství dotazů na stejnou tabulku databáze, kde může jeho cache velmi výrazně ovlivnit výsledný čas. Další velmi užitečnou vlastností je líné načítání kolekcí, kde v případě, že k nim nepotřebujeme přistoupit, Hibernate nebude provádět jejich načítání a výsledný rozdíl ve výkonnosti může být i desetinásobný, jak jsme se přesvědčili v obrázku 12.1.

U ostatních typů dotazů není výkonnostní rozdíl tak patrný. Důvodem je zřejmě to, že zdaleka nejvíce času při dotazech `INSERT` a `UPDATE` zabere samotné zpracování databázovým serverem a pevným diskem a není zde tolik prostoru pro optimalizace, jako je tomu u SELECT dotazů, kde nám velmi výkonnostně pomáhají cache perzistentních objektů.

13 Závěr

Diplomová práce se zabývá moderními technologiemi pro perzistenci objektů v jazyce Java. Toto téma jsem si zvolil proto, abych se s jednotlivými technologiemi blíže seznámil, protože jazyk Java se stává stále populárnějším a technologie pro perzistenci jeho objektů jsou jednou z oblastí, která prochází velice dynamickým vývojem.

Nejdříve byly v této práci ukázány možnosti objektové serializace, která je nejjednodušším způsobem perzistence objektů, vhodným však zejména pro jednoduché aplikace a aplikace, u kterých je ukládání dat do souborů samozřejmé, jako například textový nebo grafický editor.

V dalších kapitolách se již práce věnuje technologiím, které nám umožňují ukládat perzistentní data do databázových systémů. Přestože při programování v Javě by bylo nejpřirozenějším přístupem ukládání objektů do objektově orientovaných databázových systémů, tak se jim tato práce nevěnuje. Důvodem je jejich stále poměrně malé rozšíření a v mnoha případech i značné výkonnostní nedostatky. V současné době se v drtivé většině aplikací používají pro ukládání perzistentních objektů relační databázové systémy, které jsou po mnoha desetiletích vývoje velmi vyspělé a poskytují velmi propracované a výkonné techniky pro správu, ukládání a získávání dat, která jsou v nich uložena. V práci je poměrně podrobným způsobem popsáno aplikační rozhraní JDBC, pomocí něhož lze velmi jednoduchým způsobem přistupovat k datům uloženým v relační databázi.

Nejbouřlivější vývoj v technologiích pro perzistenci objektů v současné době probíhá u nástrojů zajišťujících objektově-relační mapování (ORM). Z tohoto důvodu je právě jim věnována i většina této práce. Úkolem těchto nástrojů je překonat tzv. *paradigm mismatch*, což je termín, který označuje nesoulad paradigmat, ke kterému dochází při použití objektově orientovaného jazyka a relační databáze. Úkolem těchto nástrojů je provádět automatický převod mezi objektovou a relační formou dat a zajistit tak pro programátora aplikace transparentní přístup pro práci s těmito daty. V práci je popsána poměrně nová technologie Enterprise JavaBeans 3.0, jejíž specifikace pro perzistenci dat se stala standardním API pro perzistenci v jazyce Java. Dále je zde popsán rámec pro objektově-relační mapování Hibernate, který zmíněnou specifikaci implementuje a je díky jeho flexibilitě v současné době nejpoužívanějším a nejoblíbenějším nástrojem tohoto typu. V další kapitole je popsáno standardizované aplikační rozhraní JDO od společnosti Sun. V poslední kapitole týkající se teorie jsou ještě stručně popsány další technologie pro perzistenci objektů. První z nich je SQLJ, což je technologie založená na hostitelské verzi jazyka SQL. Práce s SQLJ se do značné míry podobá práci s JDBC API, ale přináší oproti němu několik podstatných výhod, na které je popis zaměřen. Dále je zde uveden základní popis rámce ObJectRelationalBridge (OJB), což je další volně dostupný rámec pro ORM, který implementuje ODMG a částečně i JDO API. Posledním popsaným nástrojem je TopLink, který je velmi vyspělým nástrojem pro perzistenci, patřícím mezi placené

produkty. V samotném závěru této kapitoly je ještě přehledovým způsobem uvedeno několik dalších nástrojů.

Popis všech těchto nástrojů a technologií pro perzistenci byl zaměřen zejména na rozbor jejich výhod, nevýhod a na vzájemné porovnání. Vždy jsem se snažil upozornit na vhodnost použití dané technologie pro konkrétní druhy aplikací a problémů a vyzdvihovat vlastnosti, kterými jedna druhou převyšuje. V tomto směru bylo mým hlavním cílem poskytnou čtenáři dostatek informací k tomu, aby se po přečtení této práce byl schopen rozhodnout, kterou z nich vybrat pro řešení konkrétního problému.

V druhé části této práce (počínaje osmou kapitolou) je uveden popis vývoje aplikace, realizující informační systém pro potřeby videopůjčovny, na které byla provedena demonstrace dvou zcela rozdílných technologií. První z nich je technologie JDBC a druhou rámec pro perzistenci Hibernate. Toto vzájemné porovnání je učiněno tím, že vrstva aplikace zajišťující perzistenci dat byla naimplementována oběma technologiemi. Jako hlavní cíl, který jsem si při práci na této aplikaci vytkl, bylo jejich vzájemné porovnání z různých hledisek. Dalším cílem, který vyplynul na povrch v době počátečních návrhů aplikace, bylo její rozdělení do vrstev takovým způsobem, aby při přechodu z jedné technologie perzistence na druhou nedošlo k ovlivnění dalších vrstev aplikace. K tomuto účelu se jako nejvhodnější ukázalo použití architektonické rámce Model-View-Controller (MVC) a návrhového vzoru Data Access Objects (DAO), pomocí kterého lze zajistit jednotný přístup k perzistentním objektům, nezávisle na použité technologii pro perzistenci.

Samotná práce na systému se skládala z jednotlivých fází životního cyklu softwarového produktu. Nejprve jsem provedl podrobnou analýzu požadavků, které jsou na něj kladeny a zachytil je pomocí digramu případu použití a jeho podrobného popisu. V další fázi jsem provedl detailní návrh systému podle rámce MVC zasazeného do kontextu J2EE aplikace a vzoru DAO. V této fázi jsem na základě požadavků na systém provedl návrh perzistentních tříd. Při jeho návrhu jsem se snažil kromě splnění požadavků na systém použít pokud možná co nejvíce různých druhů vztahů, které můžeme v objektově orientované aplikaci použít, abych mohl provést objektivní porovnání složitosti implementací obou technologií pro perzistenci. Z tohoto důvodu se zde kromě obvyklých asociací s kardinalitou 1:n vyskytují vztahy dědičnosti, kompozice, a obousměrné m:n asociace. Dále je zde proveden návrh rozhraní DAO, které definuje metody pro práci s perzistentními objekty a návrh tříd, které toto rozhraní implementují pomocí JDBC a Hibernate. V další části je uveden návrh tříd vrstvy Controller a v závěru některé zásady použité při implementaci prezentační vrstvy. Poté se projekt přesunul do fáze implementace, ve které byly jednotlivé vrstvy aplikace naimplementovány pomocí příslušných technologií. Konkrétně ve vrstvě Model jde o běžné třídy jazyka Java, vrstva Controller obsahující logiku aplikace je složena ze servletů a v prezentační vrstvě byly použity technologie JSP, JSTL, HTML, CSS a JavaScript. Poslední část práce se týká výkonnostních testů jednotlivých technologií pro perzistenci objektů.

Výsledná aplikace je plně funkční v obou zmíněných technologiích a pokrývá veškeré případy

použití. Díky velice flexibilnímu návrhu je její vývoj možný v jakémkoliv směru. Jak jsem se přesvědčil, tak návrh pomocí vzoru DAO velice zjednodušuje přechod mezi technologiemi pro perzistenci a nebylo by problémem do budoucna v případě potřeby provést přechod na některou z dalších nebo přejít k používání některého z objektově orientovaných databázových systémů. Dalším směrem budoucího vývoje aplikace by mohl být přechod prezentační vrstvy na některou z novějších technologií, jako je například JavaServer Faces (JSF).

Celkově byla tvorba této diplomové práce velice zajímavým seznámením se s nejmodernějšími technologiemi, které pro mě bylo do budoucna zcela jistě i velmi přínosnou záležitostí. Jazyk Java a nejrůznější technologie pro perzistenci objektů procházejí velmi rychlým vývojem a jejich používání v oblasti tvorby informačních systému neustále narůstá a vše nasvědčuje tomu, že tento trend bude i nadále pokračovat.

Literatura

[1] Pokorný, J.: Databázové systémy a jejich použití v informačních systémech, Academia Praha, 2001

[2] Sperko, R.: Java Persistence for Relational Databases, Apress, 2003

[3] DuBois, P.: MySQL, New Riders Publishing, 1999

[4] Matthews, M., Cole, J., Gradecki, J.: MySQL and Java Developer's Guide, Wiley Publishing, Inc., 2003

[5] O`Donahue, J.: Java Database Programming Bible, John Wiley & Sons, 2002

[6] Hibernate Reference Documentation, 2006, URL: http://www.hibernate.org/hib_docs/v3/reference/en/html/

[7] Hibernate Annotations Reference Guide, 2006, URL: http://www.hibernate.org/hib_docs/annotations/reference/en/html/

[8] Bauer, Ch., King, G.: Java Persistence with Hibernate, Manning, 2006

[9] Peak, P., Heudecker, N.: Hibernate Quickly, Manning, 2006

[10] Sriganesh, R., Brose, G., Silverman, M.: Mastering Enterprise JavaBeans 3.0, Wiley Publishing, Inc., 2006

[11] Jendrock, E., Ball, J., Carson, D., Evans, I., Fordin, S., Haase, K.: The Java EE 5 Tutorial, 2006, URL: http://java.sun.com/javaee/5/docs/tutorial/doc/

[12] Šubčík, L.: Moderní způsoby perzistence dat u J2EE aplikací, diplomová práce, FI MUNI, 2006

[13] Burget, R.: Seriál článků o JDO, 2005, URL: http://interval.cz/autori/burget-radek/

[14] JDO Specification, 2003, URL: http://jcp.org/aboutJava/communityprocess/final/jsr012/

[15] Roos, R.: Java Data Objects, Addison-Wesley, 2003

[16] Zendulka, J., Rudolfová, I.: Databázové systémy, studijní opora, FIT VUT v Brně, 2006

[17] Roh, Š.: Oracle a Java, 2003, URL: http://www.srnet.cz/~stepan/st/oracle_java.html

[18] Price, J.: Java Programming with Oracle SQLJ, O'Reilly, 2001

[19] The Apache Software Fundation: Apache ObJectRelationalBridge, 2006, Dokumenty dostupné na URL: http://db.apache.org/ojb/

[20] Core J2EE Patterns, 2006, URL: http://java.sun.com/blueprints/corej2eepatterns/

[21] Oracle: Oracle TopLink Developer's Guide 10g, 2006, URL: http://www.oracle.com

[22] THOUGHT Inc.: CocoBase, 2006, Dokumenty dostupné na URL: http://www.thoughtinc.com

[23] The Apache Software Fundation: Torque, 2006, URL: http://db.apache.org/torque/

[24] ObjectStyle: Apache Cayenne, 2006, URL: http://cayenne.apache.org/

[25] Zendulka, J., Bartík, V., Květoňová, Š.: Analýza informačních systémů, studijní opora, FIT VUT v Brně, 2006

Seznam použitých zkratek a symbolů

API	Application Programming Interface
BMP	Bean-Managed Persistence
CMP	Container-Managed Persistence
CSS	Cascading Style Sheets
DAO	Data Access Objects
DTO	Data Transfer Object
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
EJB QL	Enterprise JavaBeans Query Language
HQL	Hibernate Query Language
HTML	HyperText Markup Language
IMS	Information Management System
J2C	J2EE Connector
J2EE	Java 2, Enterprise Edition
J2ME	Java 2, Micro Edition
J2SE	Java 2, Standard Edition
JAXB	Java Architecture for XML Binding
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JDOQL	Java Data Objects Query Language
JIT	Just In Time
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JSTL	JavaServer Pages Standard Tag Library
JTA	Java Transaction API
JVM	Java Virtual Machine
MVC	Model-View-Controller
ODMG	Object Data Management Group
OJB	ObJectRelationalBridge

ORM	Object-Relational Mapping
OTN	Oracle Technology Network
POJO	Plain Old Java Object
QBE	Query-By-Example
SQL	Structured Query Language
SQLJ	Structured Query Language for Java
SŘBD	Systém Řízení Báze Dat
UML	Unified Modeling Language
XML	eXtensible Markup Language

Seznam příloh

Příloha 1: Instalace aplikace

V této příloze se nachází kompletní popis instalace aplikace. Před tím, než provedeme instalaci aplikace samotné, která je velmi jednoduchá, musíme nainstalovat a nakonfigurovat všechny potřebné technologie.

Předinstalační příprava

Všechny produkty, které budeme instalovat, jsou umístěny na přiloženém disku CD v adresáři `Install`. Instalaci jednotlivých technologií budeme provádět na disk `d:` do adresáře `app`, tedy `d:\app`. Pokud budete produkty instalovat do jiného umístění, nahraďte si ve zbylé části této přílohy cestu Vaším umístěním.

Instalace MySQL

Samotná instalace MySQL serveru je velmi jednoduchá. Je však třeba provést několik jednoduchých nastavení. Prvním je vybrat formu použití serveru. Zde je třeba vybrat buď „*multifunkční*“ nebo „*transakční*“. Důvodem je dostupnost enginu *InnoDB*. Druhým je zaškrtnutí možnosti přidat adresář `bin` do systémové proměnné `PATH` a posledním nastavení hesla pro účet administrátora.

Instalace JDK

Instalace JDK je opět velmi jednoduchá a kromě nastavení instalačního adresáře není zapotřebí nic speciálního nastavovat. Po úspěšném nainstalování vytvoříme v systému proměnnou `JAVA_HOME`, která nám usnadní práci s dalšími nástroji a nastavíme ji na adresář, ve kterém je nainstalována Java. Tedy v našem případě na `d:\app\Java\jdk1.6.0`. Posledním krokem je přidání `bin` adresáře Javy do systémové proměnné `PATH`. Př.: `%JAVA_HOME%\bin`. Pro ověření úspěšnosti nainstalování, můžeme zkusit spustit z příkazové řádky překladač Javy příkazem `javac`.

Instalace serveru JBoss

Pro instalaci aplikačního serveru JBoss postačuje rozbalit připravený zip archiv například do adresáře `d:\app\jboss-4.0.5.GA`. Dále stačí již pouze vytvořit systémovou proměnnou JBOSS_HOME a nastavit ji na `d:\app\jboss-4.0.5.GA`. Posledním krokem je opět přidání adresáře `bin` do proměnné `PATH`. Př.: `%JBOSS_HOME%\bin`. Úspěšnost instalace lze ověřit spuštěním serveru pomocí příkazu `run`, který spustí dávku `run.bat` z adresáře `bin`.

Instalace nástroje Ant

Posledním, co je potřeba nainstalovat, je nástroj Ant. Jeho instalace je stejně jednoduchá jako u serveru JBoss. Opět pouze provedeme rozbalení archivu do adresáře `d:\app\ant-1.7.0`,

vytvoříme příslušnou proměnnou `ANT_HOME`, nastavíme ji na `d:\app\ant-1.7.0` a na závěr přidáme do proměnné `PATH` příslušný adresář `bin` přidáním `%ANT_HOME%\bin`. Úspěšnost instalace lze opět velmi jednoduše ověřit pomocí příkazu `ant`, který spustí dávku `ant.bat` z příslušného adresáře. Pokud je vše v pořádku a v adresáři, ze kterého jsme dávku spustili, se nenachází soubor `build.xml`, vypíše Ant příslušné chybové hlášení.

Instalace databáze a nasazení aplikace samotné

Po úspěšném nainstalování všech potřebných nástrojů, můžeme přistoupit k vytvoření databáze a k instalaci samotné aplikace. Pro tento účel jsem vytvořil instalační skripty, které celé proces velice zjednoduší.

1) **Instalace a naplnění databáze** – pro vytvoření a naplnění databáze jsou vytvořeny následující dva skripty, které jsou umístěny na přiloženém CD v adresáři `db`.
    - `install.sql` – vytvoří databázi `videopujcovna` včetně všech jejích tabulek. Posledním příkazem skriptu je založení účtu `pujcovna` s heslem `pujcovna123`, pomocí kterého bude aplikace k databázi přistupovat.
    - `fill.sql` – naplnění databáze demonstračními daty.

2) **Nasazení aplikace** – nasazení aplikace na server JBoss je prováděno pouhým zkopírováním webových archivů do příslušného adresáře serveru. Zde máme dvě možnosti, jak toto nasazení provést.
    - **Kopie archivů** – na instalačním CD v adresáři `Aplikace` jsou připraveny hotové archivy `VideopujcovnaJDBC.war` a `VideopujcovnaHibernate.war`. Pro nasazení aplikace na server stačí tyto dva soubory jednoduše zkopírovat do adresáře `%JBOSS_HOME%\server\default\deploy`.
    - **Skriptem Antu** – druhou možností, jak nasadit aplikaci na server, je použití nástroje Ant. K tomu slouží dva skripty `build.xml`, které jsou umístěny spolu s kompletními zdrojovými kódy aplikace na přiloženém CD v podadresářích:
        - `Aplikace\VideopujcovnaJDBC`
        - `Aplikace\VideopujcovnaHibernate`

      Vzhledem k tomu, že tyto sestavovací skripty neprovádějí pouze nasazení aplikace, ale nejprve kompilaci všech zdrojových souborů a následně sestavení kompletní struktury webového archivu, musejí být příslušné adresáře překopírovány nejprve z CD na pevný disk, aby skripty měly možnost vytvářet soubory a adresáře. Pro překlad a nasazení aplikace postačí v příslušném adresáři spustit Ant příkazem `ant`.

Tímto nasazením aplikace na server je instalace dokončena. Je třeba upozornit na to, že server JBoss běží (pokud nenastavíme jinak) na portu 8080.

Příloha 2: Skript pro nasazení aplikace

Vzhledem k tomu, že Ant je skutečně komplexním nástrojem určeným pro sestavování především podstatně rozsáhlejších aplikací než je tato demonstrační, je v této příloze uveden výpis souboru `build.xml`, který provádí kompletní sestavení aplikace. V tomto výpisu nejsou ilustrovány zdaleka všechny jeho možnosti a měl by posloužit čtenáři spíše k prvnímu seznámení se způsobem práce s ním.

```
<?xml version="1.0"?>

<project name="VideopujcovnaJDBC" default="all" basedir=".">

    <!-- Inicializace promennych -->
    <property name="app.name" value="${ant.project.name}"/>
    <property name="source.dir" value="src"/>
    <property name="build.dir" value="build"/>
    <property name="classes.dir" value="${build.dir}/classes"/>
    <property   name="distribution.dir"
                value="${build.dir}/distribution"/>

    <!-- soubory v lib.dir budou vlozeny do vysledneho WAR -->
    <!--soubory v compile.lib.dir neboudou vlozeny do WAR souboru-->
    <property name="lib.dir" value="lib"/>
    <property name="compile.lib.dir" value="compile-lib"/>

    <property name="war.name" value="${app.name}.war"/>
    <property name="web.dir" value="webapp"/>
    <property name="web.inf.dir" value="${web.dir}/WEB-INF"/>

    <property environment="env" />

    <path id="compile.classpath">

        <fileset dir="${compile.lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
```

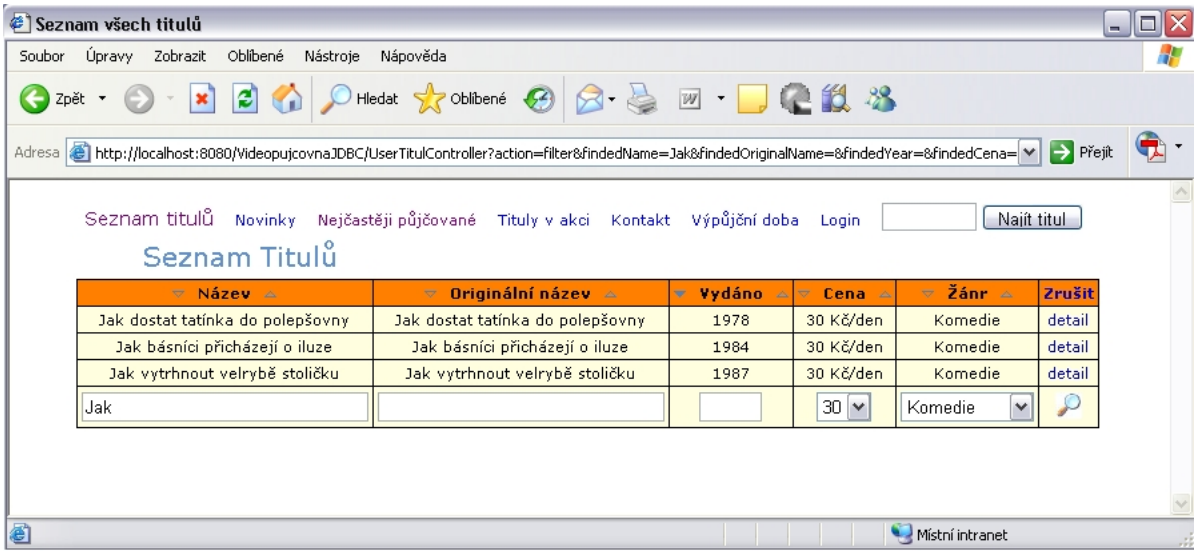
```
    <fileset dir="${lib.dir}">
        <include name="**/*.jar"/>
    </fileset>
</path>


<!-- cil, ktery provede vsechno -->
<target     name="all" depends="clean,war,deploy"
            description="Provedeni vseho"/>

<!-- smazani zkompilovanych souboru -->
<target name="clean" description="Smazani adresare build">
    <delete dir="${build.dir}"/>
</target>

<!-- kompilace souboru -->
<target name="compile" description="Kompilace kodu">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${classes.dir}" />

    <javac  srcdir="${source.dir}" destdir="${classes.dir}"
            debug="on" deprecation="on">
         <classpath refid="compile.classpath"/>
     </javac>

    <copy todir="${classes.dir}">
        <fileset dir="${source.dir}">
            <include name="**/*.properties" />
            <include name="**/*.xml" />
        </fileset>
    </copy>
</target>

<!-- tvorba WAR archivu -->
<target     name="war" depends="compile"
            description="Tvorba WAR archivu">
    <mkdir dir="${distribution.dir}" />
```

```
        <war    destFile="${distribution.dir}/${war.name}"
                webxml="${web.inf.dir}/web.xml">
            <fileset dir="${web.dir}" excludes="WEB-INF/web.xml" />
            <classes dir="${classes.dir}" />
            <lib dir="${lib.dir}" />
            <webinf dir="${web.inf.dir}" excludes="web.xml" />
        </war>
    </target>

    <!-- nasazeni aplikace na JBoss -->
    <target name="deploy" description="Deploy WAR souboru">
        <copy   file="${distribution.dir}/${war.name}"
                todir="${env.JBOSS_HOME}/server/default/deploy"/>
    </target>

</project>
```

Příloha 3: Ukázky uživatelského rozhraní

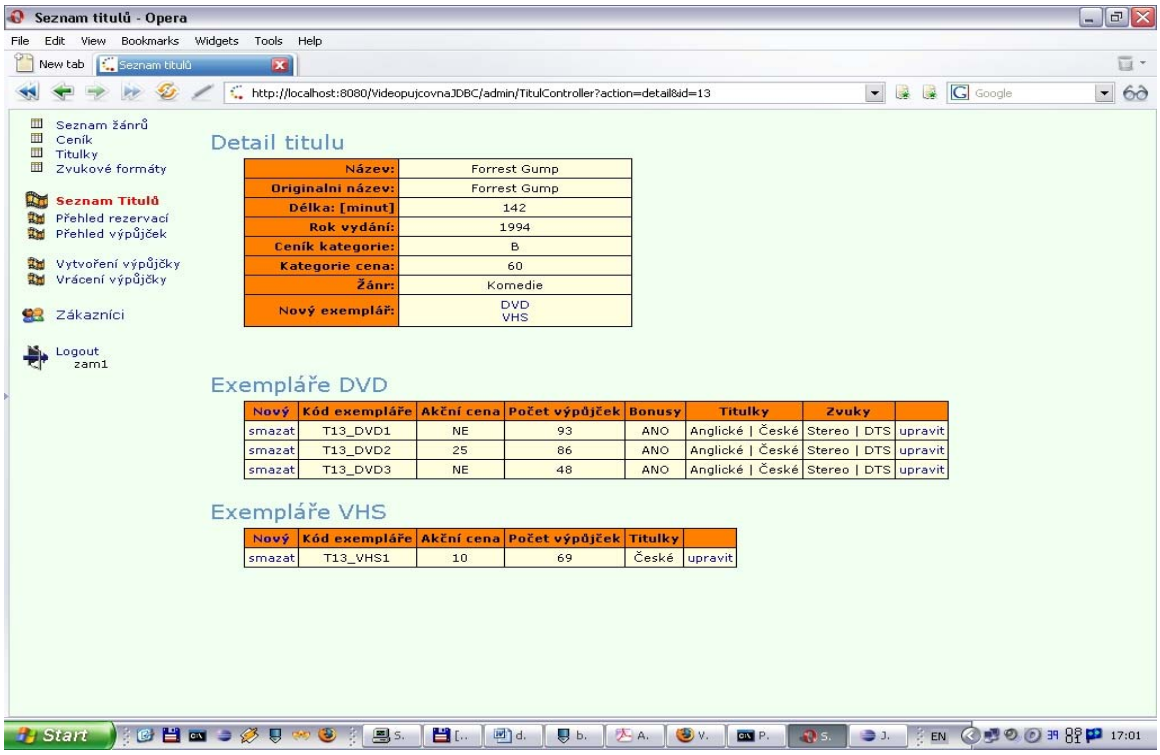
obrázek P3.1

Historie výpůjček (Uživatelská část aplikace)



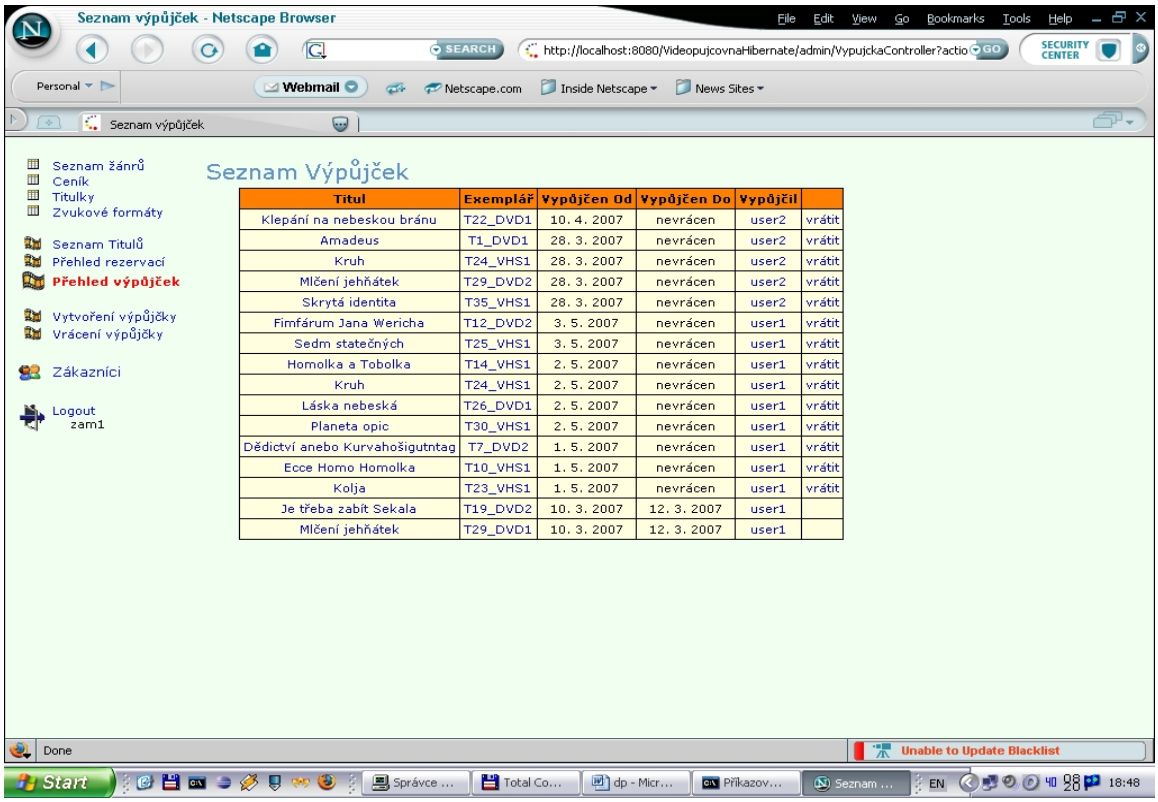
obrázek P3.2

Filtrování titulů (Uživatelská část aplikace)



obrázek P3.3

Detail titulu (Administrátorská část aplikace)



obrázek P3.4

Přehled výpůjček (Administrátorská část aplikace)