VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
EKONOMICKÁ FAKULTA

KATEDRA APLIKOVANÉ INFORMATIKY

Návrh a implementace logovacího frameworku

Design and Implementation of the Logging Framework

Student: Bc. Miroslav Zapletal

Vedoucí diplomové práce: Ing. Vítězslav Novák, PhD.

VŠB - Technická univerzita Ostrava
Ekonomická fakulta
Katedra aplikované informatiky

Zadání diplomové práce

Student: **Bc. Miroslav Zapletal**

Studijní program: N6209 Systémové inženýrství a informatika

Studijní obor: 1802T001 Aplikovaná informatika

Téma: Návrh a implementace logovacího frameworku
Design and Implementation of the Logging Framework

Zásady pro vypracování:

1. Úvod
2. Teoretická východiska a výběr metodiky
3. Analýza a návrh aplikace
4. Implementace a zhodnocení
5. Závěr

Seznam použité literatury
Seznam zkratek
Prohlášení o využití výsledků diplomové práce
Seznam příloh
Přílohy

Seznam doporučené odborné literatury:

ARLOW, Jim a Ila NEUSTADT. *UML 2 and the unified process: practical object-oriented analysis and design.* 2nd ed. Boston: Addison-Wesley, 2005. ISBN 03-213-2127-8.
BECK, Kent. *Test-driven development: by example.* Boston: Addison-Wesley, 2003. ISBN 03-211-4653-0.
SCHILDT, Herbert. *Java 7: výukový kurz.* Brno: Computer Press, 2012. ISBN 978-80-251-3748-2.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Vítězslav Novák, Ph.D.**

Datum zadání: 23.11.2012
Datum odevzdání: 26.04.2013

Ing. Petr Rozehnal, Ph.D.
vedoucí katedry

prof. Dr. Ing. Dana Dluhošová
děkanka fakulty

„Místopřísežně prohlašuji, že jsem celou práci, včetně všech příloh, vypracoval samostatně."

V Ostravě dne 26. dubna 2013

.....

Bc. Miroslav Zapletal

Rád bych poděkoval panu Ing. Vítězslavu Novákovi, PhD. za ochotu, poskytnuté rady a především trpělivost. Dále děkuji týmu ERDC ve společnosti ON Semiconductor za poskytnutí výborného zázemí.

Obsah

1 Úvod

Současný svět se ubírá směrem k čím dál většímu pronikání počítačů do každodenní lidské činnosti. Moderní společnosti nedokážou bez podpory strategických systémů přežít na globálním trhu, jednotlivci zase nedokážou bez počítače využívat ani bankovních služeb. Fakt je, že jsme počítači natolik obklopeni, že už v dnešní době do veliké míry zásadně ovlivňují směr našich životů, ať už jen z hlediska vzdělání a zaměstnání.

Zkusme se zaměřit spíše na firmy a uvažujme nad potřebou výrobních (či obchodních) informačních systémů. Výrobu řídí stroje a lidé, kteří využívají počítače ke konfiguraci strojů vyrábějících produkty. Pro obyčejného zaměstnance nepředstavuje taková skutečnost nic zvláštního a pozoruhodného. Zdání klame. Za vývojem informačního systému, i toho z uživatelského hlediska velice jednoduchého, stojí mnoho hodin práce zkušených odborníků, ať už programátorů, analytiků či projektových lídrů. Bohužel (možná i bohudík) je práce na informačních systémech nikdy nekončící. Uživatelské požadavky na funkčnost se neustále zpřesňují, přidávají se nové funkcionality, roste tlak na neustálé zvyšování výkonu systému a tak dále.

Právě proto stojí vývoj softwaru mnoho finančních prostředků. Na rozdíl od jiných investic, jsou investice do vývoje softwaru na první pohled maximálně nevýhodné. Až čas totiž ukáže, na kolik zlepšil daný software produktivitu zaměstnanců, o kolik se snížily výrobní náklady zavedením nového výrobního systému a tak bychom mohli dlouho pokračovat. Rizikovost investice do vývoje softwaru zvyšuje i nevhodné používání nových technologií nebo rychlé morální zastarání výpočetní techniky. Tyto aspekty doslova otevírají firmám dveře k promrhání kapitálu společnosti v nerentabilních softwarových projektech.

Otázkou je, jak snížit rizikovost vývoje informačních systémů, případně jak zlepšovat kvalitu softwaru. Ideální je mít hmatatelný důkaz o používání daného softwaru uživateli. Jestli software používají efektivně, jestli jeho používání vede k tvorbě přidané hodnoty atd. Tento důkaz je eso v rukávu vývojového týmu při vyjednávání s managementem společnosti o uvolnění finančních prostředků na vývoj.

A právě tento fakt je cílem diplomové práce. Vývoj systému, který by dokázal zaznamenávat používání softwaru uživateli a jednotlivé výsledky zobrazit v agregované formě pro potřeby vedení.

V rámci práce je představen průřez vývoje systému od formulace zadání, až po finální implementaci a nasazení systému. Celý vývoj probíhal v souladu s osvědčenou metodikou Unified Process. Před samotným popisem vývoje jednotlivých aspektů systému jsou nejdříve rozebrána teoretická východiska vývoje informačních systémů a popsány technologie, které byly při vývoji použity.

2 Teoretická východiska a výběr metodiky

2.1 Problematika perzistence

Téměř všechny aplikace vyžadují perzistentní data. Persistence je jedním ze základních koncepcí v oblasti vývoje aplikací. Pokud informační systém nezachová data v případě vypnutí, měl by velice malou použitelnost. Když mluvíme o perzistenci v Javě, obvykle mluvíme o ukládání údajů v relační databázi pomocí SQL. V rámci objektově orientovaných systémů nám perzistence dává možnost vytváření objektů, které nezaniknou při ukončení aplikace. Jednoduše řečeno, tyto objekty přežijí proces, který je vytvořil. Stavy (hodnoty) objektů mohou být uloženy na pevný disk a v budoucnu mohou být opětovně načteny. Pokud bychom chtěli být důslední, tak nemůžeme říci, že objekt nepřežije aplikaci, ale že je uložen mimo aplikační kontext a v případě potřeby je vytvořen objekt nový s příslušnými atributy. Tento mechanismus není pouze spjat s jednotlivými objekty, umožňuje nám uchovávat také celé grafy objektů. Abychom byli úplní, tak opakem perzistentních objektů jsou objekty tranzitivní (přechodné). Prakticky veškeré enterprise aplikace vytváří při svém běhu jak tranzitivní, tak perzistentní objekty. Proto je výhodné využívat různé subsystémy (frameworky) pro správu perzistentních objektů.

Moderní databáze nám poskytují strukturovaný pohled na perzistentní data, který nám umožňuje snadné třídění, vyhledávání, manipulaci a seskupování dat. Systémy řízení báze dat (integrované v databázových produktech) jsou pak odpovědné za konzistenci a sdílení dat (ať už mezi uživateli, či aplikacemi). V případě perzistence musíme tyto aspekty zohlednit a vyzdvihnout především:

- uchovávání, organizaci a získávání dat,
- souběžnost při práci s daty, datovou integritu,
- sdílení dat.

V rámci aplikace ovšem pracujeme v objektově orientovaném kontextu a data uchováváme zcela jinak. Zatímco na straně databáze jsou data strukturovaná do řádků a sloupců, na straně aplikace data uchováváme jako vlastnosti objektů. Navíc, a to především, musíme spravovat životní cyklus

těchto objektů a umožnit přenášení dat mezi těmito zcela odlišnými datovými modely (Bauer a King. 2007).

2.1.1 Relační databáze

Většina vývojářů používá relační databáze každý den. Relační technologie je dobře známá, a to samo o sobě je dostatečným důvodem pro používání právě těchto databází. Relační databáze se používají kvůli své obrovské flexibilitě a robustní správě dat. Ta nám umožňuje, mimo jiné, chránit integritu dat v databázích a zaručit konzistenci systému přímo na úrovni datové vrstvy. Dodnes je relační model celosvětově používaná koncepce, i když je „veteránem“ na poli IT, jelikož první zmínka o této koncepci byla vyřčena panem E. F. Coddem již roce 1970 (v článku **A Relational Model of Data for Large Shared Data Banks**). Jednou z nejdůležitějších vlastností nejen relačních databází je tzv. nezávislost dat. Tento pojem představuje skutečnost, že data jsou uchovávána naprosto nezávisle na aplikačních vrstvách (aplikacích) a je možné k nim přistupovat naprosto stejně z různých prostředí (Java, .NET, apod.). Podstatný je také fakt, že data žijí déle než aplikace (což platí dvojnásob v dnešní době datových skladů). V návaznosti na nezávislost dat je důležité zmínit, že relační databáze neposkytují možnost sdílení dat nejen mezi aplikacemi samotnými, ale i mezi částmi jednotlivých aplikací, tzv. APIs (aplikační programová rozhraní).

Pro správu a řízení dat u relačních databází se stará speciální aplikační programové rozhraní, založené používání standardu SQL pro práci s daty. V této souvislosti často narazíme na pojem „SQL databáze“ místo správného označení „relační databáze“ (Bauer a King, 2007).

Relační databáze využívají pro ukládání dat relačního modelu, ve kterém jsou data reprezentována uspořádanými n-ticemi, seřazených do relací. Relace je vztah mezi prvky jedné nebo více množin a zobrazením takové relace, v případě že se jedná o relaci binární (vztah prvků jedné množiny k prvkům množiny druhé), může být tabulka. Proto se často říká (i když nepřesně), že se data uchovávají v tabulkách (Halpin a Morgan, 2008).

2.1.2 Přístup k relačním databázím v Javě

Při práci s relační databází na platformě Java používáme JDBC API (Java Database Connectivity). JDBC je pouze specifikace přístupu k databázi, implementací jsou tzv. JDBC ovladače vyvíjené výrobci jednotlivých databází. Při práci s JDBC využíváme SQL dotazování. Tyto SQL

dotazy (příkazy) máme buď předem hotové, předpřipravené (parametrizované) nebo je vytváříme od základu při běhu programu. Typický scénář funguje následovně:

1. vytvoříme příslušný SQL dotaz (nejčastěji dosadíme parametry do předpřipravené šablony),
2. tento dotaz spustíme pomocí JDBC API,
3. JDBC API nám vrátí množinu výsledků dotazu (ResultSet),
4. procházíme řádek po řádku množinu výsledků a ukládáme jednotlivé atributy do objektů.

Tento přístup k datům se nazývá **nízkoúrovňový**. Věnujme především pozornost čtvrtému bodu. Již na první pohled je z něj patrná nesourodost objektových a relačních struktur.

Jedná se o velice pracný přístup z důvodu psaní velkého množství kódu (který se navíc často opakuje). Logicky z toho plyne, že je nízkoúrovňový přístup velice náchylný k chybám. Díky těmto aspektům se vyvinula u programátorů potřeba odbourat tyto nedostatky a vytvořit abstraktní vrstvu nad nízkoúrovňovým přístupem k datům.

Mohlo by se zdát, že je na vině relační datová struktura databází a že stačí tuto strukturu vyměnit za vhodnější pro objektové prostředí aplikací. Ovšem relační struktura je natolik rozšířená a léty ověřená, že v praxi nenajdeme de facto jinou (ano, existují výjimky, jako např. Adresářové služby, key-value databáze apod.). Takže pesimismus v oblasti relačních struktur není na místě (Bauer a King, 2007).

Tato obtížná slučitelnost objektových a relačních prostředí označujeme pojmem **paradigm mismatch** (neshoda paradigmat). Tato neshoda se netýká pouze modelování, ale konkrétnímu psaní kódu (objektově orientované programování versus SQL, imperativní versus deklarativní). Právě díky této neshodě se setkáváme u vývojářů enterprise projektů s obrovským úsilím, které je třeba vynaložit právě na řešení persistence a překonání neshody paradigmat (Seddighi, 2009).

2.1.3 Neshoda paradigmat

Pro ideální pochopení, o čem pojem paradigm mismatch je, uvedeme jednoduchý příklad. Přepokládejme, že vyvíjíme jednoduchý obchodní systém (např. e-shop). Tento systém obsahuje dvě entity – UŽIVATEL a FAKTURA, kdy uživateli může být uživateli vystaveno více faktur. Na straně aplikace si vytvoříme dvě třídy s příslušnými vlastnostmi. Relace mezi třídami může být obousměrná,

to nám objektový přístup umožňuje. Na straně databáze si vytvoříme dvě tabulky, u kterých relaci implementujeme pomocí cizího klíče. Taková relace je ovšem jednosměrná. Pokračujme v našem příkladě dále a uvažujme, že bychom chtěli uchovávat informaci o adrese uživatele. Mohli bychom přidat do třídy Uživatel vlastnost, která bude představovat řetězcovou reprezentaci adresy. Ovšem vhodnější bude vytvořit novou třídu Adresa, která bude obsahovat atomické hodnoty (ulice, číslo popisné, poštovní směrovací číslo, město). Na straně databáze nám ale stačí přidat příslušné sloupce do tabulky UŽIVATEL. Je otázka názoru, zdali je to správný přístup, ale jisté je, že se vyhneme spojování tabulek v mnoha případech (což je pro výkon databáze určitě výhodnější). Zde už je velice patrný nesoulad paradigmat. Zkusme uvažovat ještě dále a využijme tzv. objektového rozšíření relačních databází. Vytvoříme si uživatelský datový typ, který bude představovat adresu a bude obsahovat příslušné části adresy. A do tabulky UŽIVATEL následně přidáme sloupec tohoto datového typu. Takže nakonec máme možnost přidat několik sloupců nebo jeden jediný sloupec vlastního datového typu. Tento jev vychází z nesouladu paradigmat a nazývá se **granularita** (Bauer a King, 2007).

2.1.4 Granularita datový typů

Granularita spočívá v různé velikosti datových typů. K tomuto jevu nemusí dojít jen při přechodu z jednoho databázového systému k druhému (např. ze systému Oracle k MS SQL), ale i při používání uživatelských datových typů (UDT). UDT patří k tzv. objektovému rozšíření relačního konceptu, které v dnešní době podporují všechny významné databázové systémy, resp. jejich systémy řízení báze dat. Problém je ten, že UDT není přenositelný mezi jednotlivými systémy. Tím granularita dále nabývá na rozměrech (a významnosti). Nezapomínejme, že k dotazování se do databáze používáme SQL. SQL je ovšem standard, který do jisté míry implementují výrobci databází. Sám o sobě poskytuje velmi malou podporu pro UDT, což ve výsledku vyústilo ve velké rozdíly mezi verzemi SQL jednotlivých výrobců. Stává se, že jeden SQL dotaz nelze spustit proti různým databázovým systémům (Bauer a King, 2007).

2.1.5 Objektově relační mapování

Objektivně relační mapování (ORM) nám umožňuje elegantní řešení problémů představených v minulých kapitolách, konkrétně:

- nízkoúrovňového přístupu,
- neshody paradigmat,

- granularity datových typů.

ORM by se jednoduše dalo definovat jako automatizovaná, transparentní cesta k perzistenci objektů v aplikaci, kdy pomocí metadat mapujeme jednotlivé vlastnosti objektů na sloupce v tabulkách. Mapování představuje transformaci dat z jedné datové koncepce do druhé a naopak. Lze namítnout, že z podstaty věci můžeme ihned zmínit výkonnostní propad oproti nízkoúrovňovému přístupu. To je za jistých okolností pravda, ale nezapomínejme, že i v případě přímého dotazování do databáze data selektujeme a ukládáme. Navíc náklady na správu a budoucí vývoj aplikací jsou nižší v případě ORM.

Řešení přístupu k databázi pomocí ORM poskytuje:

- API pro CRUD (create, read, update, delete) operace,
- dotazovací jazyk nebo API pro specifikování dotazů,
- služby pro definici mapovacích metadat,
- podporu pro transakční zpracování, líné načítání (lazy fetching), případně jiné funkce.

Mark Fussel, výzkumník problematiky ORM, definoval čtyři následující úrovně implementace objektově relačního mapování.

Čistě relační úroveň

Na této úrovni není de facto ORM mechanismus implementován. Celá aplikace, včetně uživatelského rozhraní, plně využívá relačního datového modelu a operace s daty se provádí pomocí konkrétního SQL kódu. Tento přístup není příliš vhodný pro větší systémy, avšak pro velice jednoduché aplikace je plně dostačující, i za cenu nepřenositelnosti a horší udržovatelnosti, nemluvě o rozšíření aplikace dalším vývojem. Pro dlouhodobý vývoj je tato úroveň nevhodná.

Lehké mapování objektů

Lehké mapování objektů představuje stav, kdy jsou entity (třídy na straně aplikace) ručně mapovány na databázové tabulky. S využitím návrhových vzorů jsou ručně psané SQL dotazy ukryty před business logikou aplikace. Můžeme v tomto případě mluvit o základní úrovni abstrakce. Mnoho vývojářů používá právě tento mechanismus, aniž by si uvědomovali, že se již jedná o mechanismus objektově relačního mapování. Pro systémy s nižším počtem entit je tato úroveň dostačující.

Střední mapování objektů

Aplikace již respektuje objektové paradigma. Stále se používá SQL pro získávání dat a jejich manipulaci, avšak takový kód je již generován, buď v době kompilace, nebo přímo za běhu. Asociace mezi objekty je již podporována perzistenčním mechanismem a dotazy již mohou obsahovat objektově orientované výrazy (objektové rozšíření relačních databází). Objekty jsou ukládány do paměti perzistentní vrstvou.

Úplné mapování objektů

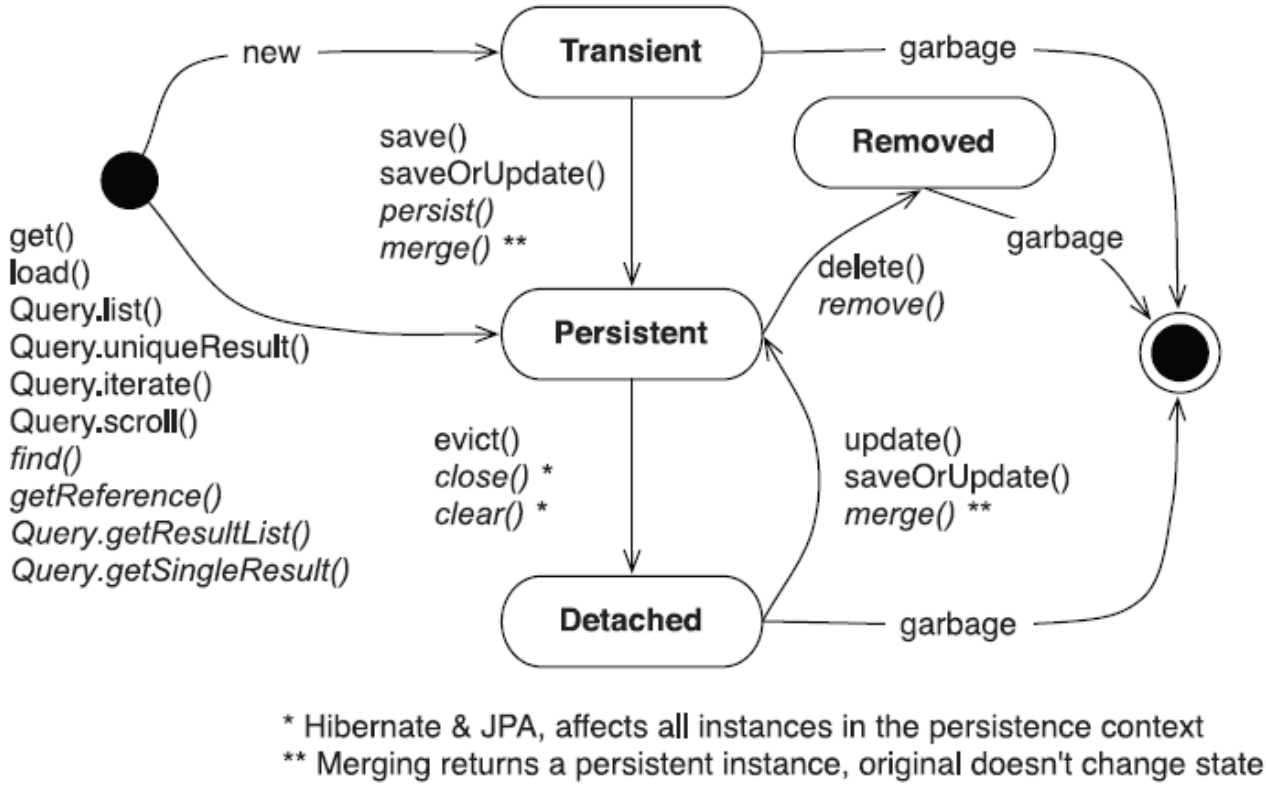
Je poslední, nejvyšší úrovní implementace ORM. V aplikaci existuje perzistentní vrstva s plnou podporou sofistikovaného objektového modelování, např. dědičnost, polymorfismus, kompozice apod. Perzistence je na této úrovni transparentní, tzn., že perzistentní objekty nemusí dědit (či přímo nedědí) od jiného objektu, případně implementovat nějaká rozhraní. Je také umožněno efektivní načítání dat pro různé případy (odložené, okamžité). Tato úroveň je velice těžce dosažitelná vlastním vývojem, jelikož takový vývoj je otázkou mnoha let. Existují ovšem různé frameworky, které nám právě tyto principy umožňují. Na platformě Java je zdaleka nepoužívanější **Hibernate**, jenž je implementací specifikace **Java Persistence API** (Bauer a King, 2007).

2.1.6 Hibernate

Hibernate je framework pro objektivně relační mapování, který umožňuje nejvyšší úroveň implementace ORM, tzn. úplné mapování objektů, se všemi výhodami. Hibernate je vyvíjen komunitou JBoss a je zároveň implementací Java Persistence API (JPA), což nám umožňuje dynamický vývoj enterprise aplikací, při kterém je možno zaměňovat jednotlivé perzistenční moduly v případě potřeby. Jádro frameworku je základní službou pro perzistenci a obsahuje engine pro zpracování jazyka HQL (Hibernate Query Language). Jádro je nezávislé na používané Java platformě, čili jej můžeme využít u Java EE aplikací, stejně jako u Java SE aplikací. Pro mapování jednotlivých tříd můžeme používat XML soubory a v případě, že používáme JDK 5.0 a vyšší, tak je velice moudré využít moderního přístupu a mapovat třídy pomocí anotací. Hibernate jako takový obsahuje vlastní anotace, avšak je možné využít Java Persistence API anotací (díky implementaci specifikace). Používání anotací má výhodu především v typové bezpečnosti a vyšší přehlednosti.

Aby Hibernate vyhovoval specifikaci JPA, obsahuje i vlastního správce entit (Entity Manager), který řídí životní cyklus perzistentních objektů. Entity manager je zastřešení nad jádrem, aby byla implementace JPA proveditelná (Bauer a King, 2007).

2.1.6.1 Životní cyklus perzistentních objektů



Obr. 2.1. Životní cyklus perzistentních objektů (Bauer a King, 2007)

2.1.6.2 Deklarace perzistentního objektu

Pro deklaraci perzistentního objektu (třídy), můžeme využít anotací z balíku JPA. V praxi často odpovídá jedna entita tabulce v databázi, proto definujeme název tabulky v kontextu celé entity. Jednotlivým vlastnostem přidělíme sloupce v příslušné tabulce, ze kterých se budou do globálních proměnných objektu (skrze settery) ukládat příslušné hodnoty. Proměnnou odpovídající primárnímu klíči označíme anotací **@Id** (Red Hat., ©2004).

```
@Entity
@Table(name = "TABLE_NAME")
public class User {

    @Id
```

```
    @Column(name = "PRIMARY_KEY_COLUMN")
    private int id;
    @Column(name = "USER_NAME")
    private String userName;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return userName;
    }

    public void setName(String name) {
        this.userName = name;
    }
}
```

2.1.6.3 Vytvoření perzistentního objektu z tranzitivního

Vytvořením instance entitní třídy se objekt nestává automaticky perzistentním. Pokud bychom vyžadovali trvalost objektu, je nutné jej synchronizovat s datovým úložištěm. Tuto synchronizaci provedeme na příslušném session objektu frameworku Hibernate. Operace uložení je ve své podstatě zápisem do databáze, proto je nutné počítat s potřebou vytvoření příslušné transakce a jejím potvrzením. V opačném případě by k synchronizaci nedošlo (Red Hat, ©2004).

```
Item item = new Item();
item.setName("Playstation3 incl. all accessories");
item.setEndDate( ... );
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
Serializable itemId = session.save(item);
tx.commit();
```

2.1.6.4 Vytvoření tranzitivního objektu z perzistentního

Pokud bychom vyžadovali odstranění objektu z perzistentní vrstvy (tzn. jeho přeměnu na tranzitivní objekt), můžeme to udělat následujícím způsobem. Tento způsob využívá tzv. proxy objekt, který získáme zavoláním metody **load()** příslušného objektu session. Mějme prosím na paměti, že

zavoláním metody **delete()** objektu session nedojde k destrukci instance příslušného objektu (Red Hat, ©2004).

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
Item item = (Item) session.load(Item.class, new Long(1234));
session.delete(item);
tx.commit();
session.close();
```

2.1.6.5 Modifikace perzistentního objektu

Při požadavku na modifikaci objektu stačí, abychom v rámci jedné transakce získali instanci objektu, které pozměníme určité vlastnosti. Potvrzením této transakce se objekt automaticky synchronizuje s perzistentní vrstvou (Red Hat, ©2004).

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
Item item = (Item) session.get(Item.class, new Long(1234));
item.setDescription("This Playstation is as good as new!");
tx.commit();
session.close();
```

2.1.6.6 Získání seznamu perzistentních objektů pomocí HQL dotazu

Jednotlivé objekty můžeme získávat i za pomoci HQL dotazu. Jedná se o objektové dotazování, které připomíná klasické SQL dotazy. V následujícím kódu vidíme vytvoření seznamu obsahující všechny instance třídy User, které perzistentní vrstva dokáže poskytnout. Výsledky dotazů můžeme omezovat podmínkami, podobně jako u SQL kódu. Je ale důležité mít na paměti, že pracujeme s objekty a ne s tabulkami a jejími sloupci (Red Hat, ©2004).

```
Session session = sessionFactory.openSession();
List list = session.createQuery("from User").list();
session.close();
```

2.1.6.7 Vytvoření proxy objektu

V některých případech není nutné pro vytvoření objektu spouštět příslušný dotaz do databáze. Pokud nám získaný objekt slouží např. pouze jako uspokojení závislosti, můžeme v takovém případě vytvořit instanci na základě jednoznačného identifikátoru (definovaného anotací @Id) bez komunikace s databází. Tento objekt je nazýván jako **proxy objekt** a je podsunutý frameworkem. Jelikož fakticky obsahuje pouze hodnotu identifikátoru, tak v případě vyžádání hodnoty jakékoli jeho vlastnosti je

vyhozena výjimka. Proxy objekt získáme zavoláním metody **load()** příslušného objektu session (Red Hat, ©2004).

```
Session session = sessionFactory.openSession();
Item item = (Item) session.load(Item.class, new Long(1234));
session.close();
```

2.1.6.8 Vytváření relací

Na rozdíl od tabulek v databázi nám Hibernate poskytuje možnost vytvářet tzv. obousměrné relace (bidirectional relations). V rámci databáze existují relace jednosměrné – z jedné tabulky se pomocí cizího klíče dostaneme k tabulce druhé. U objektů můžeme mít odkaz mezi dvěma objekty současně. To znamená, že objekt číslo jedna bude držet ukazatel na objekt číslo dvě a obráceně v jednom okamžiku. Vytvoření takové relace je velmi intuitivní a respektuje objektové principy, jak vidíme na ukázce níže (Red Hat, ©2004).

```
@Entity
@Table(name="ZAMESTNANEC")
public class Zamestnanec {
    @Id
    @Column(name="ID");
    private int     id;
    @Column(name="JMENO");
    private String  jmeno;
    @ManyToOne
    @JoinColumn(name = "ID_ODDELENI")
    private Oddeleni   oddeleni;

    //Getters & Setters
}

@Entity
@Table(name="ODDELENI")
public class Oddeleni {
    @Id
    @Column(name="ID");
    private int         id;
    @Column(name="NAZEV");
    private String      nazev;
    @OneToMany(mappedBy="oddeleni")
    private List<User>  users;

    //Getters & Setters
}
```

Hibernate nám také poskytuje podporu pro vytváření relací typu M:N. Princip je stejný jako u obousměrných relací, akorát oba objekty mají v držení množinu referencí. Jelikož se v relačních databázích modeluje M:N za pomoci tzv. vazební tabulky, musíme tuto tabulku definovat u příslušné vlastnosti anotací **@JoinTable**, jak je uvedeno níže (Red Hat, ©2004).

```
@Entity
@Table(name="ZAMESTNANEC")
public class Zamestnanec {
    @Id
    @Column(name="ID");
    private int     id;
    @Column(name="JMENO");
    private String  jmeno;
    @ManyToMany
    @JoinTable(name = "ODDELENI_ZAMESTNANEC", joinColumns = {
        @JoinColumn(name = "ID_ODDELENI")},
                    inverseJoinColumns = {
        @JoinColumn(name = "ID_ZAMESTNANCE")})
    private Oddeleni   oddeleni;

    //Getters & Setters
}
```

2.1.7 Kritika objektivně relačního mapování

Pro zachování objektivity je důležité zmínit i kritiku ORM přístupu. ORM je určené především pro práci s plnými entitami. Relační databáze ovšem nabízí velice výkonné funkce pro kombinování, třídění, filtrování a přeměnu entit a jejich atributů. Ačkoliv s těmito funkcemi drží nejmodernější ORM frameworky krok v zájmu zachování kompatibility, je často poměrně obtížné tyto funkce implementovat. Proto je nutné klást veliký důraz na vzdělávání se v používání jednotlivých frameworků, což má za následek možné prodloužení doby vývoje a zpočátku i velikou náchylnost k chybám. Obecně se ale při perfektním osvojení těchto technik považuje objektivně relační mapování za přínos při vývoji enterprise systému (Apache Software Foundation, ©2008-2012).

2.2 Spring framework

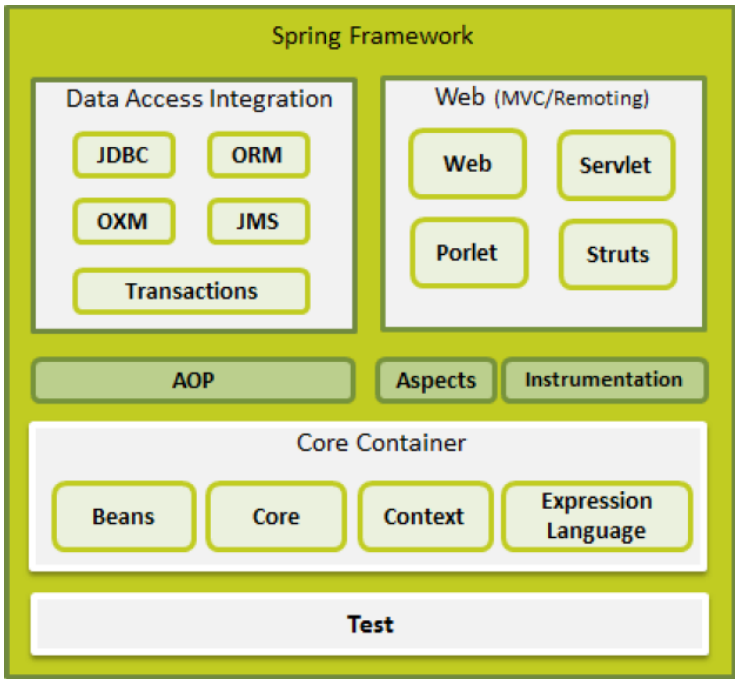
Spring je framework pro vývoj enterprise aplikací na platformě Java. Hlavní filozofií je vytváření znovupoužitelného kódu a poskytnutí manažera životního cyklu business objektů. Na rozdíl od technologie EJB, je Spring Framework od začátku vyvíjen jako lightweight a není omezen pouze na platformu Java EE, nýbrž je možno jej plně využívat i na platformě Java SE, díky používání POJOs

(plain old java objects). Proto není k běhu Springu potřeba EJB kontejner. Mezi další výhody používání Springu patří (SpringSource, ©2004-2013):

- modulární architektura (používáme pouze ty moduly, které potřebujeme),
- podpora pro integraci ORM frameworků, logovacích frameworků apod.,
- lightweight Inversion of Control kontejner,
- jednodušší testování enterprise aplikací,
- API pro aspektově orientované programování.

2.2.1 Moduly

Spring Framework je relativně obsáhlá sada knihoven. Pro zjednodušení byl celý framework rozdělen do několika (SpringSource, ©2004-2013).



Obr 2.2. Schéma Spring Frameworku (SpringSource, ©2004-2013)

- Core container (základní kontejner)
    - Core (jádro) – základní část frameworku poskytující IoC kontejner.
    - Beans – implementace návrhového vzoru factory pro dynamické vytváření instancí.

    - Context – zastřešení nad moduly core a beans, poskytuje přístup k definovaným objektům.
    - Expression language – podpora pro vytváření dotazování a manipulaci s objekty během běhu aplikace.
- Data access/integration (modul datové vrstvy)
    - JDBC vrstva – abstrakce nad nízkoúrovňovým přístupem k datovým zdrojům, poskytující znovupoužitelný kód pro přístup k databázím.
    - ORM modul – integrace ORM frameworků.
    - OXM modul – abstrakce nad XML mapováním objektů, např. JAXB.
    - JMS modul – podpora pro předávání zpráv (java messaging services).
- Transakční modul – poskytováni podpory pro správu transakcí v rámci POJO objektů.
- Web
    - Webový modul – poskytování základních webových funkcionalit (např. podpora pro upload více souborů současně), včetně inicializace IoC kontejneru servlety.
    - Web-servlet modul – integrovaný MVC (model-view-controller) framework.
    - Web-struts modul – integrace Struts frameworku.
    - Web-portlet modul – podpora MVC filozofie v prostředí porletů.
- Ostatní moduly
    - Podpora pro aspektově orientované programování (včetně AspectJ).
    - Podpora pro Junit a jiné frameworky pro unit testing.
    - Implementace class loaderu pro některé aplikační servery.

2.2.2 Inversion of Control a Dependency injection

Hlavním znakem Spring frameworku je tzv. dependency injection (DI), česky injekce závislostí. Dependency Injection je konkrétní uplatnění filozofie Inversion of Control, která nám předkládá vzor pro vytváření relací mezi objekty. Podle této filozofie by třídy měly být maximálně na sobě nezávislé a

tato závislost by měla být vytvořena až při běhu aplikace, nikoli v době její kompilace. Tento přístup si klade za cíl nechat vytváření a správu objektů na externím procesu, mimo programový kód vlastní aplikace. Právě kvůli mechanismu spočívající v podsouvání jednotlivých objektů za běhu, přejmenoval Martin Fowler Inversion of Control na mnohem přesnější Dependency Injection (Harrop a Ho, 2012).

2.3 Aspektově orientované programování

Aspektově orientované programování (AOP) je filozofie psaní programového kódu, která umožňuje rozbít programovou logiku na jednotlivé samostatné části (concerns). Tyto části slouží k modularizaci vyvíjeného systému, které můžeme samostatně (nezávisle na ostatních) spravovat. Některé funkce pokrývají celý programový kód aplikace, nejznámějším případem je určitě obyčejné logování. Takovým funkcím říkáme **cross-cutting concerns** a většinou nesouvisí s business logikou aplikace. To už je důvod pro jejich odstínění. Pro nejjednodušší možné vysvětlení AOP si uveďme analogii s OOP (objektově orientované programování). Základním stavebním kamenem OOP je třída, zatímco základním stavebním kamenem AOP je aspekt. AOP ale není konkurentem OOP, nýbrž jeho doplněním o možnost nového přístupu k psaní kódu. Z toho vyplývá, že aspekt je s největší pravděpodobností implementován právě třídou. Pokud nám slouží injekce závislostí (Dependency Injection) k odstranění závislostí mezi objekty samotnými, tak AOP slouží k odstranění závislostí mezi objekty a programovým kódem, který ovlivňuje jejich chování (Laddad, 2010).

Pro uplatnění AOP musíme pochopit následující pojmy (SpringSource, ©2004-2013):

- **Aspekt** – modul, který představuje uplatnění požadavků cross-cutting koncernů. Aplikace může mít libovolný počet aspektů pro uspokojení požadavků. Například logovací modul nazýváme v terminologii AOP aspektem, který spravuje logování.
- **Joint point** – místo připojení je oblast programu, na které, jak již název napovídá, navážeme jednotlivé aspekty. V kontextu OOP se nejčastěji jedná o metody aplikace.
- **Advice** – konkrétní akce provedená aspektem neboli aplikovaný (navázaný) kód na příslušném joint pointu. Advices dělíme do následujících kategorií:
    - **Before advice** – tato akce je vykonána před samotným joint pointem. Při použití můžeme snadno ovlivňovat hodnoty parametrů metod, či zablokovat volání metody.

- **After advice** – vykonání akce až za joint pointem, tzn. až po obdržení návratové hodnoty metody. Návratovou hodnotu můžeme upravit a předat dál, čímž přidáme k původní metodě novou funkcionalitu.
- **Around advice** – kombinace obou předchozích případů. Akce obalí celou metodu, čímž umožní kompletně změnit povahové vlastnosti metody. Můžeme upravovat hodnoty vstupních parametrů i hodnotu návratového typu. Danou „obalenou" metodu můžeme i nemusíme provést, což bývá někdy výhodné.

- **Pointcut** – množina přípojných bodů, ve kterých by měl být spouštěn advice. Specifikace této množiny se provádí pomocí různých výrazů pro popis objektů.
- **Introduction** – umožnění přidávání nových metod či vlastností do již existujících tříd.

2.4 JavaServer Faces

JavaServer Faces (JSF) je v podstatě klasický Java framework pro vývoj webových aplikací. Je ovšem zaměřen na zjednodušení vývoje uživatelského rozhraní, které bývá mnohdy kamenem úrazu webových projektů. Pro tvorbu uživatelského rozhraní se může sice využívat jiných Java technologií, jako JSP (JavaServer Pages) či přímo Java Servlets, ale vzhledem k úzké vazbě na business logiku těchto řešení se často vývojáři dostávají do problémů při rozšiřování aplikace. Navíc mnoho aplikací využívá stejných principů a mechanismů, které bychom i tak museli uplatňovat při využití JSP či Servletů. JSF nám nabízí robustní základ obsahující tyto mechanismy, stejně jako implementaci osvědčených návrhových vzorů používaných při vývoji webových aplikací na platformě Java (Burns a Schalk, 2010).

JSF je striktně založená technologie na architektuře Model-View-Controller (MVC), jenž rozděluje aplikaci do tří, na sobě co nejméně závislých komponent:

- **Model** – reprezentace informací, se kterými aplikace pracuje. Model poskytuje data, které zobrazuje vrstva **View.**
- **View** – vrstva poskytující uživatelské rozhraní uživateli. Tato vrstva zobrazuje data poskytovaná modelem a umožňuje uživateli s těmito daty interaktivně nakládat.

- **Controller** – komponenta zajišťující obsluhu událostí vrstvy **View**. Na základě těchto událostí následně **Controller** upravuje příslušný model, což tranzitivně vyvolá změny ve vrstvě **View**, nebo přímo samotnou vrstva **View** bez úpravy modelu (v případech, kdy to není zapotřebí).

Poptávka po oficiálním MVC frameworku na platformě Java dala za vznik **požadavku na specifikaci** s označením **JSR127** v roce 2001. Od této doby je technologie JavaServer Faces oficiálně součástí Java EE platformy (Oracle Corporation, ©2013).

Vývoj specifikace JSR127 byl zaměřen především na dosažení následujících cílů (Burns a Schalk, 2010):

1. Vytvoření frameworku pro snadné a efektivní vytváření vysoce kvalitního uživatelského rozhraní s využitím znovupoužitelných grafických komponent.
2. Definování množiny jednoduchých tříd pro obsluhu uživatelských událostí. Tyto třídy by řídily životní cyklus jednotlivých uživatelských grafických komponent.
3. Grafické komponenty by měly být založeny na standardních komponentách HTML. Tyto komponenty by měly být použitelné jako základ nových, složitějších komponent.
4. Řízení událostí mezi grafickým rozhraním a server-side aplikační vrstvou by mělo být umožněno použitím klasických JavaBeans.
5. Podpora pro validaci uživatelských vstupů, včetně validace na straně klienta, by měla být nedílnou součástí frameworku.
6. Framework by měl umožňovat jednoduchou lokalizaci aplikace do různých jazyků.
7. Poskytování automatického mechanismu zjišťování klientských informací, např. verze webového prohlížeče.
8. Poskytnutí automatického mechanismu pro generování výstupu splňující podmínky přístupnosti, především pro osoby zdravotně postižené, tak jak to stanoví iniciativa WAI konsorcia W3C.

Od verze 2.0 využívá JSF pro zobrazování technologii **Facelets. Facelets** je systém šablon založený na jazyku XML a ruší závislost JSF na JSP kontejneru, respektive na zastaralé JSP technologii vůbec, protože plně zastupuje vrstvu **View** (Hookom, 2005).

2.4.1 PrimeFaces

PrimeFaces je knihovna obsahující sadu uživatelských komponent pro použití ve webových aplikacích využívajících JSF. Jedná se o open source, lightweight řešení, které slouží především k zobrazování vizuálních komponent. Obsahuje více než 100 komponent snadných k použití včetně podpory tematických stylů, AJAXu a specifikace HTML5 (PrimeFaces, ©2011).

2.5 RESTful webové služby

Všechny webové služby využívají protokol HTTP pro svůj provoz. Ne všechny jej ale využívají totožným způsobem. Většina webových služeb používá pro komunikaci protokol SOAP, který slouží k zabalení zprávy do XML formátu. Tento způsob sice přináší mnohem lepší typovou bezpečnost a procedurální rozhraní pro webovou službu, ovšem klade relativně vysoké nároky na zpracování a síťový provoz. Pro zasílání jednoduše strukturovaných, maloobjemových dat skrze webovou službu poslouží mnohem lépe tzv. **RESTful** webová služba. Tento typ webových služeb využívá pouze metod protokolu HTTP, nejčastěji **PUT** a **GET,** pro předávání dat. To může mít za následek nižší systémové nároky a menší síťový provoz. Taková webová služba ovšem nemá k dispozici mechanismus pro jasný a definitivní popis svého rozhraní (Richardson a Ruby, 2007).

Ne vždy je jednoduché rozhodnout, zda použít SOAP nebo REST pro implementaci webových služeb. Implementaci webových služeb na principu REST bychom měli zvážit v následujících případech (Tyagi, 2006):

1. Webová služba je kompletně bezstavová. To můžeme snadno ověřit úvahou, zdali má interakce webové služby přežít restart serveru.
2. Pokud odpověď webové služby není dynamicky generována. V tomto případě stojí za zvážení implementace tzv. cache pro zvýšení výkonnosti systému.
3. Poskytovatel i příjemce služby kompletně znají podstatu a kontext obsahu přenášeného skrze webovou službu. Jelikož RESTful webová služba neobsahuje možnost pro jasný popis rozhraní (na rozdíl od SOAP webové služby), je nutné, aby obě strany byly předem dohodnuté na způsobu komunikace.
4. V případě nedostatku šířky pásma pro komunikaci, například na mobilních zařízeních.

5. Při možnosti budoucího rozšíření současných webových stránek aplikace o nové chování neinvazivním způsobem.

2.6 XML Binding

V rámci Java aplikací často používáme XML pro výměnu dat. Není to náhoda, ale následek uznání XML jako standardu pro přenos informací napříč různými systémy. Přenosy informací ve struktuře XML bývají základním stavebním kamenem webových služeb. V rámci JDK máme několik možností, jak můžeme XML soubory zpracovávat. Můžeme využít například SAX (Simple API for XML) či DOM (Document Object Model). Nebudeme rozebírat jednotlivé vlastnosti těchto technologií, ale rovnou si řekneme, že ani jeden způsob zpracování XML se příliš nehodí pro implementaci přístupu zvaného **XML Binding**. Tento přístup spočívá v reprezentaci Java objektů pomocí XML formátu. Pro tyto případy bylo vytvořeno vedle SAX a DOM nové aplikační rozhraní s **názvem Java Architecture for XML Binding (JAXB)**. JAXB zjednodušuje přístup k XML dokumentu z Java aplikace podsunutím dat ve formátu jazyka Java. Jednoduše řečeno – schéma XML dokumentu navážeme (binding) na Java třídy, které toto schéma reprezentují. Jednotlivé objekty poté můžeme jednoduše serializovat do XML souborů a v případě potřeby je z XML souborů deserializovat. Odpadá pracné procházení XML souboru a ukládání jednotlivých hodnot do vlastností objektů, jako v případě použití SAX či DOM. XML Binding má obrovské využití např. při zajištění jednoduché perzistence, či posílání objektů napříč webovými službami, především typu REST (Mehta a Ort, 2003).

2.7 Metodika vývoje softwaru

Metodika vývoje softwaru je standardní proces probíhající v organizaci, který realizuje všechny kroky potřebné k analýze, návrhu, implementaci a udržování informačního systému. Je souhrnem vzájemně konzistentních postupů, metod, technik a nástrojů.

U volby metodiky hrají v současné době klíčovou roli dva faktory – **iterativnost** a **agilita**.

Iterativnost znamená opakovatelnost jednotlivých etap vývoje softwaru. Na rozdíl od vodopádového vývoje, kdy každá následující etapa začíná až po úplném ukončení etapy předcházející (což má za následek prodloužení doby vývoje a předání méně kvalitního produktu), jsou jednotlivé etapy provedeny a následně se celý proces opakuje, než projekt dostane finální podobu. Iterativnost je důležitá vlastnost vývojových metodik, kvůli proměnnému vnějšímu prostředí. Pro kvalitu výsledného

produktu je zapotřebí veškeré uživatelské požadavky neustále konzultovat a upřesňovat se zadavatelem (uživatelem), čímž se často mění směr vývoje projektu.

Agilitou rozumíme vlastnost umožňující pružně zasahovat do vývoje systému. V případě agilních metodik se systémovost přístupu odsouvá až na druhou kolej a vývoj se ubírá směrem k tzv. prototypování, kdy je za intenzivní konzultace s uživateli vytvořen prototyp systému, který se následně pružně vyvíjí podle dalších uživatelských požadavků. Agilita je dominantou malých vývojových týmů, kde jednotliví členové nejsou úzce zaměřeni na jedinou oblast vývoje (programování, analýza), ale podílejí se (více nebo méně) na všech etapách vývojového procesu (Kaluža, 2010).

Moderní metodiky vývoje systémů nejsou ovšem pouze iterativní, případně agilní. V poslední době se neustále zvětšuje význam refactoringu a testování. Testováním ale nemáme na mysli testování finálního produktu nebo přírůstku, nýbrž tzv. unit testing (jednotkové testování). Unit testing spočívá v automatizovaném testování jednotlivých stavebních jednotek aplikace (v kontextu OOP jsou to třídy). Některé metodiky, jako extrémní programování či Test Driven Development, přikládají unit testingu naprosto zásadní význam při vývoje softwaru (Beck, 2003).

2.7.1 Unified Process

Unified Process je metodika vývoje softwaru od autorů jazyka UML. Zatímco UML je grafický aparát sloužící k modelování systému, UP je procesní část při které UML používáme. UP je metodika založená na metodách Ericsson a Rational a na dalších postupech tvorby softwaru, ze kterých si vybírá pragmatičnost a ověřené postupy. Za osobu stojící za vznikem UP se často považuje Ivar Jacobson, který sehrál ve vývoji metodiky klíčovou roli (Arlow a Neustadt, 2005).

2.7.1.1 Axiomy

Základem metodiky UP jsou tři axiomy, které je třeba mít neustále při vývoji softwaru na paměti (Arlow a Neustadt, 2005):

1. řízení případem užití a rizikem – celá tvorba softwaru se odvíjí od uživatelských požadavků, které je třeba neustále upřesňovat a třídit,
2. soustředění na architekturu – metodika je orientovaná na tvorbu robustní architektury pomocí návrhu a postupného vývoje,

3. iterace a přírůstky – celá metodika je iterativní, důvodem je neustálá komunikace se zadavatelem a následné upřesňování jeho požadavků, díky čemuž je větší pravděpodobnost vytvoření kvalitního produktu.

2.7.1.2 Projektové řízení a milníky

Celá metodika respektuje principy projektového řízení. Projekt je rozdělen do čtyř částí:

1. **zahájení** – tato fáze zahrnuje studii proveditelnosti (feasibility study), představení podnikatelského záměru, definice kritických rizik a zachycení hlavních požadavků na tvorbu systému,
2. **rozpracování** – cílem je vytvoření spustitelného základu systému, přesnější definice rizik, definice metrik pro hodnocení kvality, zachycení případů užití pro 80% funkčních požadavků,
3. **konstrukce** – zde se již utváří systém do jeho finální podoby, ve které bude nasazen, s důrazem na zachování integrity systému, která může být narušena nedostatkem času,
4. **zavedení** – tato fáze začíná po skončení testování a při finálním zavádění produktu, je důležité odstranit dosud neopravené chyby systému a připravit uživatele k používání nového systému (školení).

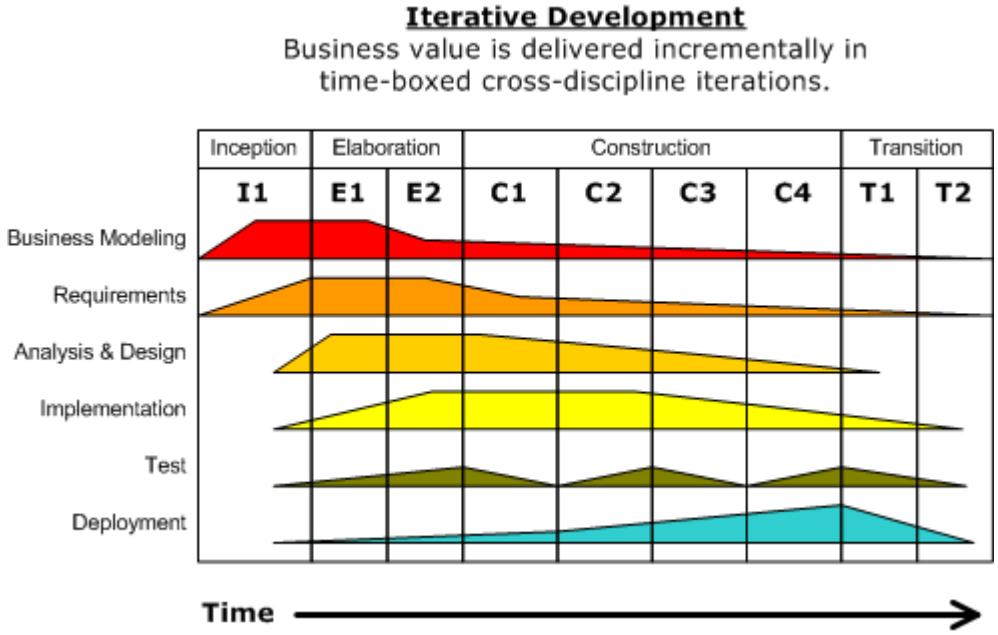
Každá fáze projektu je zakončená milníkem neboli hmatatelným důkazem o ukončení etapy, který můžeme předložit jak zadavateli, tak vedení. Každý milník má definované podmínky (metriky), které je třeba splnit, abychom mohli považovat etapu za ukončenou. Milníkem fáze zahájení je rozsah systému, pro fázi rozpracování je milníkem architektura. U konstrukce milník představuje počáteční provozní způsobilost, kdy je možno systém testovat na uživatelských počítačích. Milníkem zavedení je systém, který úspěšně prošel beta-testy a je možno jej nasadit do ostrého provozu (Arlow a Neustadt, 2005).

2.7.1.3 Iterace

V každé fázi projektu se provede nejméně jedna iterace. Počet iterací není striktně daný, je třeba respektovat zkušenosti vývojového týmu. Iterace se cyklují tak dlouho, dokud nejsou splněny podmínky milníku, potom můžeme postoupit do další fáze projektu. V rámci každé iterace se uplatňuje pět hlavních aktivit, tzv. **RADIT** (Arlow a Neustadt, 2005):

- **requirements** (požadavky) – aktivita spojená se získáváním uživatelských požadavků,

- **analysis** (analýza) – vybroušení uživatelských požadavků a jejich strukturování,
- **design** (návrh) – v rámci návrhu už neřešíme co by měl systém dělat, ale jak to má systém dělat; je to realizace požadavků v rámci architektury,
- **implementation** (implementace) – programování,
- **testing** (testování) – ověření, zda implementace odpovídá požadavkům.



Obr. 2.3. Objem prací v jednotlivých fázích vývoje (Arlow a Neustadt, 2005)

2.7.2 Test-driven development

Programování řízené testy (TDD) je způsob psaní programového kódu založený na neustálém testování. Tento přístup k vývoji nám stanoví následující vývojový cyklus:

1. **psaní testů** – nejdříve napíšeme testy, které budou kontrolovat funkcionalitu, tzv. unit testy (viz kapitola Unit testing),
2. **spuštění testů** – testy spustíme a ujistíme se, že všechny testy neprojdou, pokud ano, tak jsou špatně napsány (obsahují chyby),
3. **psaní vlastního kódu** – po selhání všech testů se teprve pouštíme do psaní vlastního výkonného kódu; držíme se pravidla, aby byl kód co nejrychleji napsán, tak aby prošel testem, bez ohledu na efektivitu,

4. **refactoring** – zdrojový kód podrobíme refactoringu.

V praxi se TDD odráží na kvalitě psaného kódu a jeho eleganci. Programátor je vždy jištěn sadou testů, což výrazně snižuje riziko neúspěchu a eliminuje se strach z velkých změn v aplikačním kódu. Dále mají programátoři důkaz o funkcionalitě aplikace, což je při komunikaci s některými uživateli obrovská výhoda. Často se stává, že uživatelé si vymýšlí skutečnosti o fungování aplikace a staví programátory do nevýhodné pozice.

Někteří odpůrci TDD jsou toho názoru, že neúměrně prodlužuje dobu vývoje, jelikož nepíšeme jen výkonný kód, ale i kód, který není přímo v aplikaci použit. Zde je ovšem nutné podotknout, že záleží na zkušenostech programátora. TDD lze kombinovat s ostatními vývojovými metodikami, např. s UP, kde lze snadno aplikovat princip testování na jednotlivé uživatelské požadavky, či případy užití (Beck, 2003).

2.7.3 Unit testing

Unit testing neboli jednotkové testování je testování aplikačních jednotek (většinou tříd). Princip vychází z předpokladu, že každá jednotka by měla být samostatně testovatelná. V současné době, kdy je dominantní objektově orientované programování, je tento předpoklad samozřejmostí. Pomocí unit testů si programátor neustále ověřuje zdrojový kód, zdali má správnou funkcionalitu. V praxi se unit testing provádí psaním testovacích tříd. Každá tato třída testuje pouze jedinou třídu výkonného zdrojového kódu aplikace. Pro jednotkové testování na platformě Java existují v dnešní době speciální frameworky, z nichž je nejznámější JUnit, který má výbornou podporu v prostředí NetBeans i Eclipse (Beck, 2003).

2.8 Unified Modelling Language

Unified Modelling Language (UML) je modelovací jazyk sloužící k popsání systému. Obecně modelovací jazyk obsahuje pseudokód, diagramy, obrázky, zdrojový kód či slovní popis. Popravdě může obsahovat cokoli, čím jsme schopni popsat systém. Každý modelovací jazyk má svou notaci, která představuje určitý popis systému. Této vlastnosti říkáme **sémantika**. Existuje mnoho modelovacích jazyků, které můžeme použít, ale UML se vyznačuje následujícími výhodami oproti konkurenci (Hamilton a Miles, 2006):

1. Každý element má přesně definovaný svůj význam, takže si můžeme být jisti, že nedojde k celkovému nepochopení koncepce navrhovaného systému.
2. Celá množina prvků jazyka UML je vytvořena na základě jednouchých notací.
3. UML dokáže popsat veškeré důležité aspekty systému.
4. UML je dostatečně robustní na modelování celého systému. V případě potřeby ovšem poskytuje možnost systémové dekompozice, při které se zaměřujeme na modelování jednotlivých subsystémů.
5. Jazyk UML prošel dlouhým vývojem, který ovlivnily nejlepší návyky při vývoji aplikací za posledních 20 let.
6. Poslední výhoda je pro někoho možná pouze formalita, ale přesto ji zmíníme. UML je otevřený standard, který (často) bývá upraven různými firmami či skupinami. Díky tomu nejsme svázáni pouze s jediným produktem.

2.8.1 Struktura

Struktura jazyka UML obsahuje tři základní prvky:

1. **stavební bloky** – základní prvky jazyka (entity, relace mezi nimi a celé diagramy),
2. **společné mechanismy** – způsoby, kterými lze dosahovat specifických cílů,
3. **architekturu** – řešení architektury systému.

Jednotlivé prvky si rozebereme v kapitolách níže.

2.8.1.1 Stavební bloky

V rámci stavebních bloků existují tři entity (prvky) – **předměty, vztahy a diagramy.**

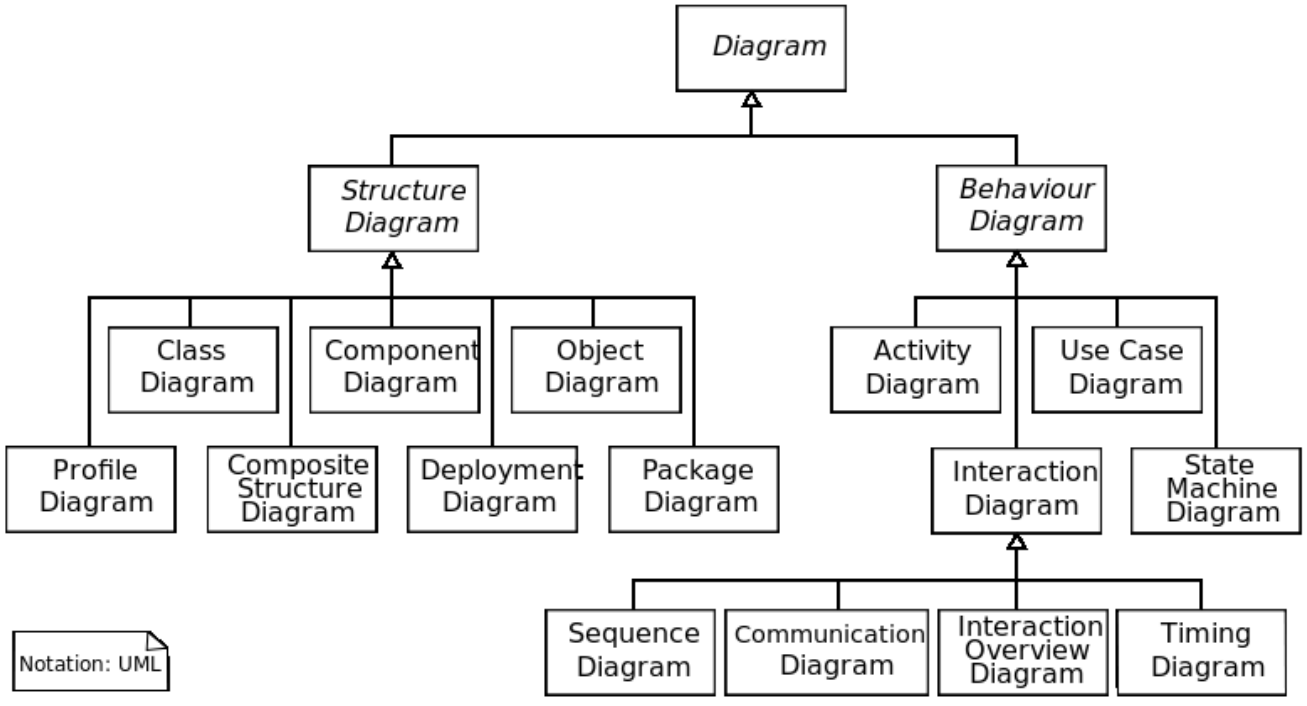
Předměty dělíme na:

1. **strukturní abstrakce** – třída, rozhraní, komponenta, případ užití,
2. **chování** – interakce, stav,
3. **seskupení** – balíček,
4. **poznámky** (anotace).

Mezi jednotlivými předměty mohou existovat jisté relace, které umožňují zachytit sémantický vztah. UML nám poskytuje aparát pro modelování celkem 7 druhů vztahů:

1. **závislost** – změna jedné entity vyvolá změnu druhé entity,
2. **asociace** – spojení mezi objekty,
3. **agregace** – prvek je součástí jiného prvku,
4. **kompozice** – silnější verze agregace,
5. **ochranná nádoba** – zdrojový prvek obsahuje cílový prvek,
6. **zobecnění** – generalizace, je možno prvek nahradit obecnějším prvkem,
7. **realizace** – asociace mezi klasifikátory.

Vztahy a předměty využíváme pro vytváření diagramů. V UML existují dvě kategorie diagramů – **diagramy struktury a diagramy chování** (Arlow a Neustadt, 2005).



Obr. 2.4. Diagramy jazyka UML2 (Arlow a Neustadt, 2005)

2.8.1.2 Společné mechanismy

Společné mechanismy představují strategie modelování objektů, které se používají napříč celým jazykem UML. Modely mají vždy alespoň grafický rozměr (modelování pomocí diagramů a symbolů), a textový rozměr (specifikace prvků modelu). Právě **specifikace** představují první společný mechanismus jazyka UML. Druhým společným mechanismem jsou **ornamenty**, které umožňují obohacení jednotlivých prvků UML. Při modelování složitého systému můžeme začít na velice málo objektech, které časem obohatíme o různé ornamenty. Tato vlastnost ukazuje velikou dynamičnost celého jazyka. **Podskupiny** jsou třetím společným mechanismem jazyka. V rámci UML se podskupiny dělí na skupinu klasifikátorů a instancí (abstraktní vyjádření typu předmětu a konkrétní výskyty takových vyjádření) a skupinu rozhraní a implementací (co je předmětem vykonávání a jakým způsobem je toho dosaženo). Čtvrtým společným mechanismem je **rozšiřitelnost**. Ta je obzvláště důležitá vzhledem k faktu, že UML je univerzální modelovací jazyk. Rozšiřitelnost byla dosažena uplatněním třech principů – možnost přidávat k prvkům omezení, možnost definování nových prvků založených na stávajících prvcích a v poslední řadě možnost rozšiřovat prvky o nové informace (Arlow a Neustadt, 2005).

2.8.1.3 Architektura

Architekturou se myslí organizační struktura systému, včetně jeho rozkladu na jednotlivé součásti, jeho propojitelností, interakce, mechanismů a směrných zásad, která proniká do návrhu systému (Booch, Jacobson a Rumbaugh, 2005).

Abychom byli schopni pomocí UML modelovat architekturu systému, je definováno pět různých pohledů na systém (Arlow a Neustadt, 2005):

1. **Logický pohled** – důraz je kladen na způsob zobrazení, jakým třídy a objekty implementují chování daného systému.
2. **Pohled procesů** – procesově orientovaná varianta logického pohledu mající stejné artefakty.
3. **Pohled implementace** – v rámci tohoto pohledu se modelují soubory a komponenty, které tvoří hotový kód (respektive kódový základ) systému. Zaprvé slouží k znázornění závislostí komponent, zadruhé zobrazuje způsob správy a konfigurace modelovaného systému.

4. **Pohled nasazení** – představuje nasazení systému na fyzické výpočetní uzly (osobní počítače, servery, mobilní zařízení…). Umožňuje modelovat způsob distribuce systému na jednotlivé uzly.
5. **Pohled případů užití –** základní pohled UML, od kterého se ostatní odvíjí. Symbolizuje zachycení uživatelských požadavků na funkčnost systému.

3 Analýza a návrh aplikace

3.1 Formulace zadání projektu

Pro potřeby společnosti ON Semiconductor, s. r. o. v Rožnově pod Radhoštěm byl vznesen požadavek na vývoj systému, který byl umožňoval logování (protokolování) různých událostí vzniklých v interních aplikacích a informačních systémech společnosti. Co si představit pod pojmem logování různých událostí? Především se jedná o zaznamenávání uživatelských akcí. Vývojový tým mající v rukávu přesné informace o používání softwaru uživateli může upravit v příštích verzích příslušný software, tak aby lépe odpovídal představě o komfortu užívání. Držme se pravidla – spokojenější uživatel systému = výkonnější uživatel systému. V druhé řadě se jedná o protokolování objemu přenášených dat mezi poskytovateli webových služeb a jejich klienty. Webové služby se ve společnosti používají pro přenos dat mezi desktopovými aplikacemi a datovými sklady. Pro budování síťové infrastruktury je důležité mít přesné informace o objemu přenášených dat, které využijeme při návrhu optimalizace počítačové sítě. Samozřejmě zde hraje roli i aspekt financování. Vývojový tým drží v ruce čísla o využití aplikací, kterými si může naklonit vedení společnosti, při vyjednávání o uvolnění finančních prostředků na rozvoj infrastruktury.

V neposlední řadě budou sloužit jednotlivé záznamy pro interní potřeby vývojového týmu. Jedná se o např. o měření doby zpracování časově náročných úloh, evidence běhových prostředí aplikací či fyzických počítačů, na kterých se aplikace provozují.

Logovací systém by měl být dostatečně flexibilní, aby bylo umožněno jeho využití v širokém spektru aplikací, a zároveň dostatečně jednoduchý, aby nebylo časově náročné zakomponování logování do jednotlivých aplikací.

Přesně definované požadavky pro funkčnost systému jsou popsány v následující kapitole.

3.2 Specifikace požadavků

Zachycení požadavků probíhalo formou skupinových rozhovorů, při kterých se projednávaly jednotlivé funkční i nefunkční aspekty systému. Požadavky byly v průběhu vývoje neustále upřesňovány a upravovány, aby bylo dosaženo dodání informačního systému nejvyšší možné kvality, což je v naprostém souladu s iterativností metodiky Unified Process.

Jednotlivé požadavky jsou strukturovány do dvou kategorií – funkční a nefunkční. Funkční požadavky přímo souvisejí s funkcionalitou systému, to znamená., že ovlivňují funkčnost (chování) informačního systému či jednoduše aplikace. Nefunkční požadavky chování systému přímo neovlivňují, ale jsou stále spjaté se systémem a je třeba s nimi při vývoji počítat a respektovat je.

ID	Popis	Druh
R1	Protokolování událostí v aplikacích	funkční
R2	Protokolování síťového provozu webových služeb	funkční
R3	Statistiky fyzických počítačů, na kterých aplikace běží	funkční
R4	Statistiky běhových prostředí zaznamenaných aplikací	funkční
R5	Statistiky uživatelů používajících aplikaci	funkční
R6	Centrální zapínání/vypínání protokolování pro jednotlivé aplikace (typy událostí)	funkční
R7	Vyhledávání jednotlivých záznamů podle kritérií	funkční
R8	Správa uživatelů s možností přidělovat práva správce	funkční
R9	Přihlašování do aplikace na základě autentizace uživatele v Adresářových službách	funkční
R10	Vytváření grafických přehledů z pohledu jednotlivých atributů záznamů	funkční
R11	Při ukončení logované aplikace nepřijít o zaznamenané události	funkční
R12	Lightweight řešení logovací části systému	nefunkční
R13	Při nedostupnosti služby neomezí aplikaci	nefunkční
R14	Nenáročnost na síťový provoz	nefunkční
R15	Použití databáze Oracle pro ukládání záznamů	nefunkční

Tab. 3.1. Uživatelské požadavky

3.2.1 R1 – protokolování událostí v aplikacích

Tento požadavek spočívá v zaprotokolování událostí v aplikacích. Prakticky se jedná o klasickou činnost loggeru, avšak na jednotlivé záznamy je třeba nahlížet z jiných úhlů, např. zda se jedná o událost vyvolanou uživatelem.

3.2.2 R2 – Protokolování síťového provozu webových služeb

Protokolování síťového provozu webových služeb spočívá v měření objemu dat, přenášených mezi klienty a poskytovali webových služeb ve společnosti. Přes webové služby se posílají poměrně objemné soubory dat, proto je důležité tyto přenosy měřit a na základě výsledků případně posílit síťovou infrastrukturu.

3.2.3 R3 – Statistiky fyzických počítačů, na kterých aplikace běží

Vývojáři interních aplikací nemají přesně přehled, na jakých počítačích jejich aplikace fungují. Proto je třeba zaznamenat fyzické počítače, aby bylo možné zjistit, na jakém hardwaru aplikace fungují.

3.2.4 R4 – Statistiky běhových prostředí zaznamenaných aplikací

Pro budoucí vývoj logovaných aplikací, je nutné zjistit verze běhového prostředí, které je nainstalováno na pracovních stanicích společnosti. Jedná se především o operační systém a verzi Java Runtime Environment.

3.2.5 R5 – Statistiky uživatelů používajících aplikaci

Tento požadavek spočívá v protokolování uživatelů, kteří spouštějí logované aplikace, případně jakým způsobem tyto aplikace obsluhují, např. podle jakých parametrů vyhledávají apod.

3.2.6 R6 – Centrální zapínání/vypínání protokolování pro jednotlivé aplikace (typy událostí)

Informační systém by měl mít možnost zapínat či vypínat logování jednotlivých aplikací, případně určitých typů událostí, a to z jediného místa.

3.2.7 R7 – Vyhledávání jednotlivých záznamů podle kritérií

Uživatel musí mít možnost jednotlivé záznamy vyhledávat podle aplikace, typu události, případně podle umístění výrobního závodu, kde aplikace běží. Rovněž je nutné časově omezit vyhledávání – vyhledávat záznamy je možné pouze od určitého kvartálu určitého roku.

3.2.8 R8 – Správa uživatelů s možností přidělovat práva správce

V rámci aplikace musí být umožněno správci přidělovat, případně odebírat, práva pro zapínání/vypínání logování aplikací jednotlivým uživatelům.

3.2.9 R9 – Přihlašování do aplikace na základě autentizace uživatele v Adresářových službách

Autentizace uživatelů přihlašujících se do systému, bude ověřena v rámci Adresářových služeb společnosti. Každý zaměstnanec má v adresářových službách záznam, pomocí kterého se přihlašuje do firemní doménové sítě.

3.2.10 R10 – Vytváření grafických přehledů z pohledu jednotlivých atributů záznamů

Aplikace by měla umožňovat vytvářet grafické přehledy jednotlivých záznamů. Tyto přehledy by byly z pohledu jednotlivých atributů, jako např. typ události, aplikace, výrobní závod apod.

3.2.11 R11 – Při ukončení logované aplikace nepřijít o zaznamenané události

Při ukončení logované aplikace nesmí dojít ke ztrátě jednotlivých záznamů. Musí být zajištěn mechanismus umožňující, aby veškeré logy, které nebyly dosud uloženy v databázi, byly do databáze uloženy.

3.2.12 R12 – Lightweight řešení logovací části systému

Řešení logovacího frameworku musí být lightweight, tzn. jednoduché k použití, málo paměťově náročné a nezvětšující velikost samotných aplikací. Poslední bod je důležitý, protože mnoho aplikací se ve společnosti spouští přes rozhraní Java Web Start, což by při nárůstu velikosti distribuce vedlo k pomalejšímu spouštění a většímu síťovému provozu.

3.2.13 R13 – Při nedostupnosti služby neomezí aplikaci

Pokud není dostupná konektivita do databáze, logger nesmí jakkoli omezit činnost aplikace.

3.2.14 R14 – Nenáročnost na síťový provoz

Odesílání jednotlivých záznamů z loggeru musí být co nejméně náročné na síťový provoz.

3.2.15 R15 – Použití databáze Oracle pro ukládání záznamů

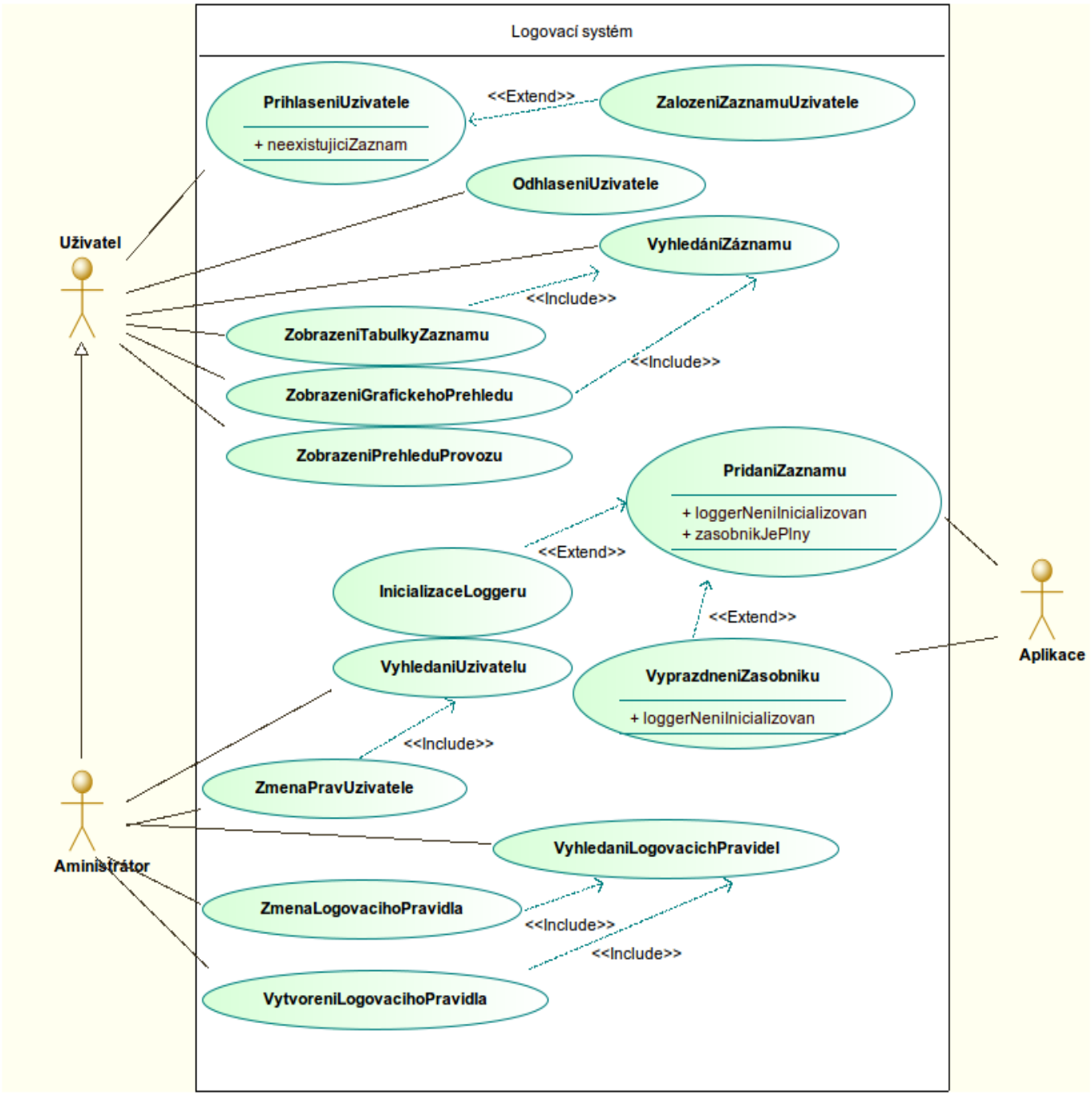
K ukládání záznamů bude sloužit firemní relační databáze Oracle Database 10g.

3.3 Diagram případů užití

Na základě uživatelských požadavků jsou modelovány jednotlivé případy užití. Případ užití je „specifikace posloupností činností, včetně proměnných posloupností a chybován posloupností, které systém, podsystém nebo třída může vykonat prostřednictvím interakce s vnějšími (externími) aktéry“ (Booch, Jacobson a Rumbaugh, 2005).

Je důležité uvést, že případy užití jsou **vždy** modelovány z pohledu aktéra, nikoli z pohledu systémového inženýra (kniha UML). Z toho vyplývá, že se maximálně vyhýbáme funkční dekompozici jednotlivých případů užití, protože se musíme orientovat na zachycení požadavků a ne na umělé strukturování požadavků. Tento způsob často vede k chybné definici jednotlivých případů užití a následně chybnému návrhu. Navíc je tento způsob modelování v rozporu s moderním objektovým přístupem (Arlow a Neustadt, 2005).

Diagram případů užití zobrazený níže, nám dává jasnou představu obecné funkcionalitě modelovaného systému. Jak je vidět na obrázku č. 3.1, jednotlivé případy užití mohou zahrnovat jiné (relace include) či mohou rozšiřovat jiné případy užití o nové chování (relace extend). Pro lepší přehlednost můžeme využít i generických vztahů mezi jednotlivými aktéry (spouštěči případů užití).



Obr. 3.1. Diagram případů užití

3.4 Specifikace jednotlivých případů užití

Ačkoliv nám diagram případů užití poskytuje perfektní procesní pohled na celý systém, jednotlivé případy je důležité blíže specifikovat. Ačkoliv existují jednoduché lineární procesy, u

kterých je tok práce a výstup vždy stejný, modelovaná podmnožina reality nás často staví k vytváření alternativních toků – scénářů. Často je zapotřebí u případu užití vzít v potaz skutečnost, že ne vždy stačí k popisu jeden scénář. Musíme pokrýt pokud možno veškeré možné stavy tvorbou scénářů alternativních. Pro nejvyšší možnou přehlednost popisu případů užití, včetně alternativních scénářů, využíváme oddělenou dokumentaci.

Následující tabulky představují pokročilé modelování případů užití, včetně rozšiřujících případů užití, či popisu alternativních scénářů.

Případ užití: PřihlášeníUživatele
ID: UC1
Stručný popis
Uživatel se přihlásí do systému
Hlavní aktéři
Uživatel
Vedlejší aktéři
Žádní
Vstupní podmínky
1. Uživatel existuje v Adresářové službě a je ověřen
Hlavní scénář
1. Případ užití začíná, když Uživatel žádá o vstup do systému 2. Dokud není Uživatel ověřen v Adresářové službě: 2.1. Uživatel je zažádán o vyplnění svého uživatelského jména a hesla 2.2. Systém ověří správnost zadaných údajů dotazem do Adresářové služby 3. Systém vyhledá záznam o uživateli a jeho právech místo rozšíření: neexistující záznam 4. Uživateli je povolen vstup do systému
Výstupní podmínky
Uživatel byl přihlášen do systému
Alternativní scénáře
NeplatnéÚdaje

Tab. 3.2. Dokumentace případu užití UC1

Alternativní scénář: PřihlášeníUživatele:NeplatnéÚdaje
ID: UC1.2
Stručný popis:
Uživatel se přihlásí do systému
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:

1. Uživatel zadal neplatné údaje
Alternativní scénář:
1. Případ užití začíná krokem 2.2. hlavního scénáře 2. Systém informuje uživatele o neplatných přihlašovacích údajích
Výstupní podmínky:
Žádné

Tab. 3.3. Dokumentace případu užití UC1.2

Rozšiřující případ užití: ZaloženíZáznamuUživatele
ID: UC2
Stručný popis:
Systém nenašel záznam o uživateli a jeho právech
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Uživatel je ověřen, ale nemá v systému záznam s jeho právy
Hlavní scénář:
1. Alternativní scénář začíná krokem 3 hlavního scénáře
2. V systému je založen záznam o uživateli a jsou mu přidělena práva pouze pro čtení
Výstupní podmínky:
1. Uživateli byl založen záznam s právy
Alternativní scénáře:
Žádný

Tab. 3.4. Dokumentace případu užití UC2

Případ užití: OdhlášeníUživatele
ID: UC3
Stručný popis:
Uživatel se odhlásí ze systému
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Uživatel je přihlášen v systému
Hlavní scénář:
1. Případ užití začíná, když Uživatel provede akci odhlášení
2. sezení (session) je terminováno a uživatel je odhlášen
Výstupní podmínky:
1. Uživatel byl odhlášen
Alternativní scénáře:
Žádné

Tab. 3.5. Dokumentace případu užití UC3

Případ užití: VyhledáníZáznamů
ID: UC4
Stručný popis:
Uživatel hledá záznamy
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Uživatel je přihlášen v systému
Hlavní scénář:
1. Uživatel zadá kritéria hledání
2. Systém vyhledá záznamy
Výstupní podmínky:
1. Systém nalezl příslušné záznamy
Alternativní scénáře:
ŽádnýZáznamNenalezen

Tab. 3.6. Dokumentace případu užití UC4

Alternativní scénář: VyhledáníZáznamů:ŽádnýZáznamNenalezen
ID: UC4.2
Stručný popis:
Systém nenalezl záznamy odpovídající kritériím
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Neexistující záznamy s příslušnými atributy
Alternativní scénář:
1.Případ užití začíná krokem 2. hlavního scénáře
2. Systém informuje uživatele o tom, že nenalezl žádný záznam
Výstupní podmínky:
Žádné

Tab. 3.7. Dokumentace případu užití UC4.2

Případ užití: ZobrazeníTabulkyZáznamů
ID: UC5
Stručný popis:
Uživatel chce zobrazit tabulku záznamů
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:

1. Uživatel je přihlášen v systému
Hlavní scénář:
1. Zahrnout (VyhledáníZáznamů)
2. Uživatel zvolí zobrazení výsledků v tabulce
3. Systém zobrazí vyhledané záznamy v tabulce
Výstupní podmínky:
1. Systém zobrazil tabulku se záznamy
Alternativní scénáře:
Žádné

Tab. 3.8. Dokumentace případu užití UC5

Případ užití: ZobrazeníGrafickéhoPřehledu
ID: UC6
Stručný popis:
Uživatel chce zobrazit grafický přehled záznamů
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Uživatel je přihlášen v systému
Hlavní scénář:
1. Zahrnout (VyhledáníZáznamů)
2. Uživatel zvolí zobrazení výsledků v grafickém přehledu
3. Systém zobrazí grafický přehled vyhledaných záznamů
Výstupní podmínky:
1. Systém zobrazil grafický přehled
Alternativní scénáře:
Žádné

Tab. 3.9. Dokumentace případu užití UC6

Případ užití: ZobrazeníPřehleduSíťovéhoProvozu
ID: UC7
Stručný popis:
Uživatel chce zobrazit grafický přehled síťového provozu
Hlavní aktéři:
Uživatel
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Uživatel je přihlášen v systému
Hlavní scénář:
1. Zahrnout (VyhledáníZáznamů)
2. Uživatel zvolí zobrazení síťového provozu webových služeb
2. Systém zobrazí grafický výstup o síťovém provozu webových služeb

Výstupní podmínky:
1. Systém zobrazil grafický přehled
Alternativní scénáře:
Žádné

Tab. 3.10. Dokumentace případu užití UC7

Případ užití: VyhledáníUživatelů
ID: UC8
Stručný popis:
Administrátor hledá záznamy o uživatelích a jejich právech
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Administrátor je přihlášen v systému
Hlavní scénář:
1. Administrátor vyhledá uživatelské záznamy s informacemi o právech
2. Systém zobrazí záznamy v tabulce
Výstupní podmínky:
1. Systém zobrazil záznamy
Alternativní scénáře:
Žádné

Tab. 3.11. Dokumentace případu užití UC8

Případ užití: ZměnaPrávUživatele
ID: UC9
Stručný popis:
Administrátor změní práva u uživatelského záznamu
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Administrátor je přihlášen v systému
Hlavní scénář:
1. Zahrnout (VyhledáníUživatelů)
2. Administrátor přiřadí/odebere práva pro administraci příslušnému uživateli
Výstupní podmínky:
1. U příslušného záznamu jsou přiřazena/odebrána administrátorská práva
Alternativní scénáře:
Žádné

Tab. 3.12. Dokumentace případu užití UC9

Případ užití: VyhledáníLogovacíchPravidel
ID: UC10

Stručný popis:
Administrátor hledá pravidla pro logování
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Administrátor je přihlášen v systému
Hlavní scénář:
1. Uživatel vyhledá pravidla logování
2. Uživateli jsou zobrazeny záznamy
Výstupní podmínky:
1. Systém zobrazil záznamy
Alternativní scénáře:
ŽádnéPravidloNenalezeno

Tab. 3.13. Dokumentace případu užití UC10

Alternativní scénář: VyhledáníLogovacíchPravidel:ŽádnéPravidloNenalezeno
ID: UC10.2
Stručný popis:
Systém nenalezl žádné logovací pravidla
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Neexistující pravidla logování
Alternativní scénář:
1.Případ užití začíná krokem 1. hlavního scénáře
2. Systém informuje uživatele o tom, že nenalezl žádný záznam
Výstupní podmínky:
Žádné

Tab. 3.14. Dokumentace případu užití UC10.2

Případ užití: ZměnaLogovacíhoPravidla
ID: UC11
Stručný popis:
administrátor změní pravidlo logování
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Administrátor je přihlášen v systému
Hlavní scénář:

1. Zahrnout (VyhledáníLogovacíchPravidel)
2. Administrátor zapne/vypne logování pro záznamy odpovídající pravidlu
Výstupní podmínky:
1. Pravidlo je změněno
Alternativní scénáře:
Žádné

Tab. 3.15. Dokumentace případu užití UC11

Případ užití: VytvořeníLogovacíhoPravidla
ID: UC12
Stručný popis:
Administrátor vytvoří nové pravidlo logování
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Administrátor je přihlášen v systému
Hlavní scénář:
1. Zahrnout (VyhledáníLogovacíchPravidel)
2. Administrátor vytvoří nové logovací pravidlo
Výstupní podmínky:
1. je vytvořeno nové pravidlo logování
Alternativní scénáře:
DuplicitníPravidlo

Tab. 3.16. Dokumentace případu užití UC12

Alternativní scénář: VytvořeníLogovacíhoPravidla:DuplicitníPravidlo
ID: UC12.2
Stručný popis:
Pravidlo již existuje
Hlavní aktéři:
Administrátor
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Vytvořené pravidlo již existuje
Alternativní scénář:
1.Případ užití začíná krokem 2. hlavního scénáře
2. Systém informuje uživatele o tom, že pravidlo již existuje
Výstupní podmínky:
Žádné

Tab. 3.17. Dokumentace případu užití UC12.2

Případ užití: Přidání záznamu
ID: UC13

Stručný popis:
Aplikace zaloguje událost
Hlavní aktéři:
Logovaná aplikace
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Logger je inicializován
Hlavní scénář:
1. Aplikace zažádá logger o zaprotokolování události
Místo rozšíření: loggerNeníInicializován
2. KDYŽ je povoleno logování příslušným pravidlem:
2.1 Událost se přidá do zásobníku
Místo rozšíření: zásobníkJePlný
Výstupní podmínky:
1. Záznam je přidán
Alternativní scénáře:
Žádné

Tab. 3.18. Dokumentace případu užití UC13

Rozšiřující případ užití: Vyprázdnění zásobníku
ID: UC14
Stručný popis:
Zásobník se záznamů se uloží a vyprázdní
Hlavní aktéři:
Logovaná aplikace
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Logger je inicializován
2. Zásobník je plný
Hlavní scénář:
1. Zásobník je plný nebo si aplikace vyžádá vyprázdnění
Místo rozšíření: loggerNeníInicializován
2. Jednotlivé záznamy ze zásobníku jsou nahrány do databáze
3. Zásobník se promaže
Výstupní podmínky:
1. Zásobník je vyprázdněn
Alternativní scénáře:
Žádné

Tab. 3.19. Dokumentace případu užití UC14

Rozšiřující případ užití: InicializaceLoggeru
ID: UC15
Stručný popis:

Logger se inicializuje
Hlavní aktéři:
Logovaná aplikace
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Logger není inicializován
Hlavní scénář:
1. Logger si ověří konektivitu
2. Je připraveno rozhraní pro zasílání protokolů událostí
3. Zaeviduje se název PC
4. Zaeviduje se běhové prostředí
5. Jsou načtena pravidla pro logování
Výstupní podmínky:
1. Logger je inicializován
Alternativní scénáře:
NeníKonektivita

Tab. 3.20. Dokumentace případu užití UC15

Alternativní scénář: InicializaceLoggeru:NeníKonektivita
UC15.2
Stručný popis:
Konektivita neexistuje
Hlavní aktéři:
Logovaná aplikace
Vedlejší aktéři:
Žádní
Vstupní podmínky:
1. Konektivita pro získání konfigurace neexistuje
Alternativní scénář:
1.Případ užití začíná krokem 1. hlavního scénáře
2. Logger není inicializován a logování neprobíhá
Výstupní podmínky:
Žádné

Tab. 3.21. Dokumentace případu užití UC15.2

3.5 Matice sledovatelnosti požadavků

Při modelování případů užití je důležité si uvědomit, že u obou množin (požadavků a případů užití) je neustále zapotřebí uchovávat jistý vztah. Tento vztah nám říká, že jednotlivé požadavky musí mít přiřazen minimálně jeden případ užití, aby došlo uplatnění požadavku při návrhu systému. Platí to i obráceně – každý případ užití musí mít přiřazen minimálně jeden požadavek. Pokud nastane situace, kdy požadavek nemá přiřazen případ užití, je nezbytné takový případ užití vytvořit. Pokud případ užití nemá přirazen žádný požadavek, může to znamenat, že takový případ užití je nadbytečný. Může to

ovšem znamenat i fakt, že jsme nezaznamenali příslušný požadavek, který musíme do zadání přidat. Matice sledovatelnosti požadavků nám ve formě tabulky přehledně zobrazuje vztahy mezi funkčními požadavky a jednotlivými případy užití (Arlow a Neustadt, 2005).

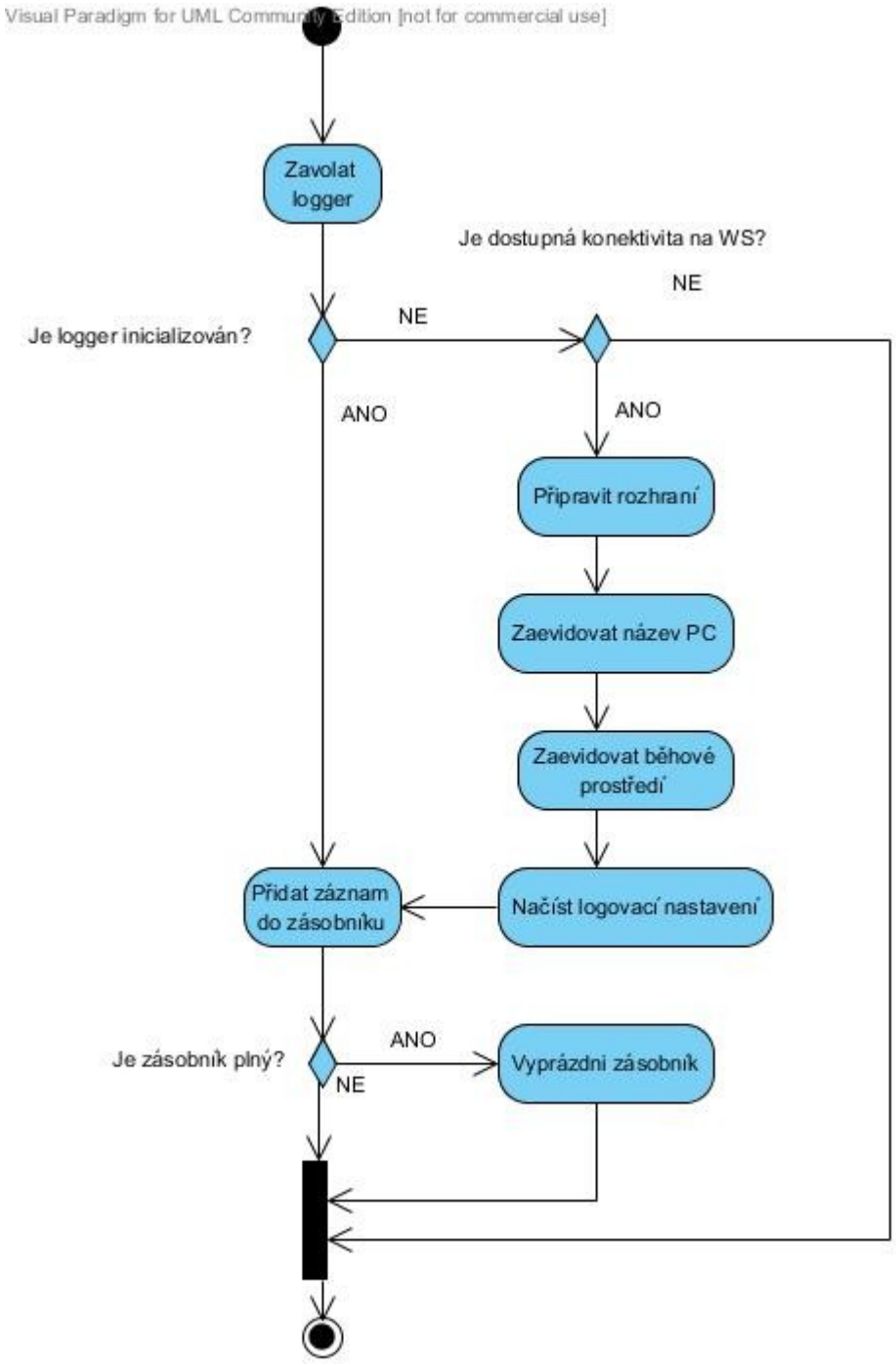
	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10	UC11	UC12	UC13	UC14	UC15
R1													X		
R2													X		
R3															X
R4															X
R5													X		
R6										X	X	X			
R7				X											
R8								X	X						
R9	X	X	X												
R10					X	X	X								
R11														X	

Tab. 3.22. Matice sledovatelnosti požadavků

3.6 Realizace případů užití

3.6.1 Diagram aktivit pro přidání nového záznamu

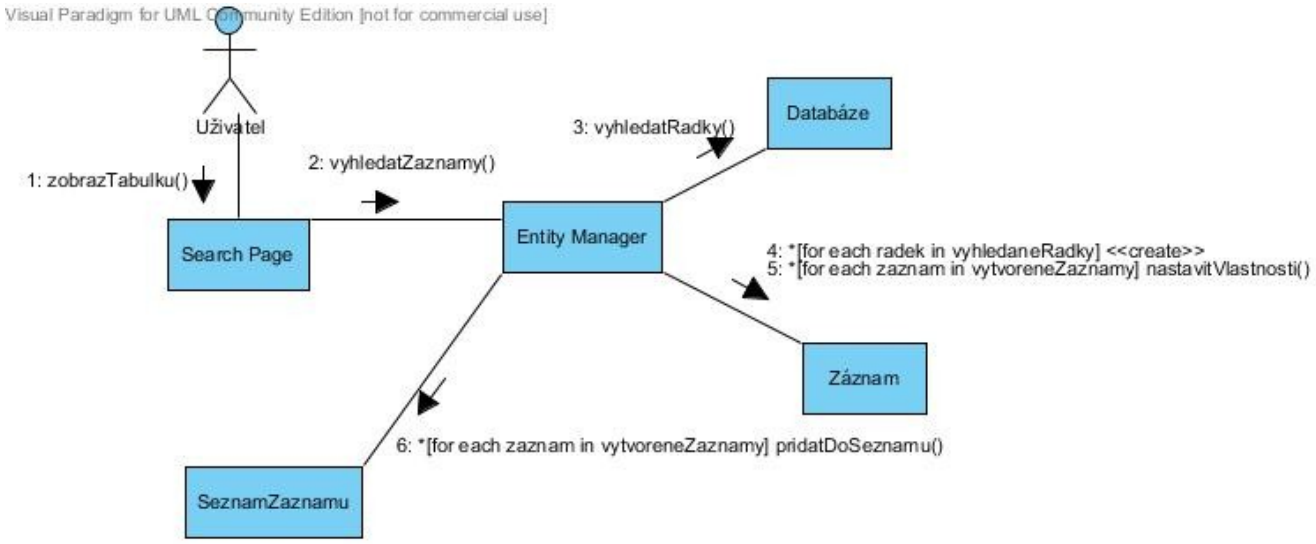
Diagram aktivit umožňuje modelovat procesy navrhovaného systému. Ve specifikaci UML2 je založen na Petriho sítích což přináší výhody ve větší přizpůsobivosti při modelování typů cest a odlišení diagramů aktivit od stavových diagramů. Výhodou využití diagramů aktivit je jeho univerzálnost, která nám poskytuje aparát k modelování chování případů užití, tříd, rozhraní, komponent apod. Diagram aktivit můžeme využít při obecném modelování procesů případů užití bez nutnosti definice tříd a objektů realizujících daný proces (Arlow a Neustadt, 2005).



Obr. 3.2. Diagram aktivit pro UC13

3.6.2 Diagram spolupráce

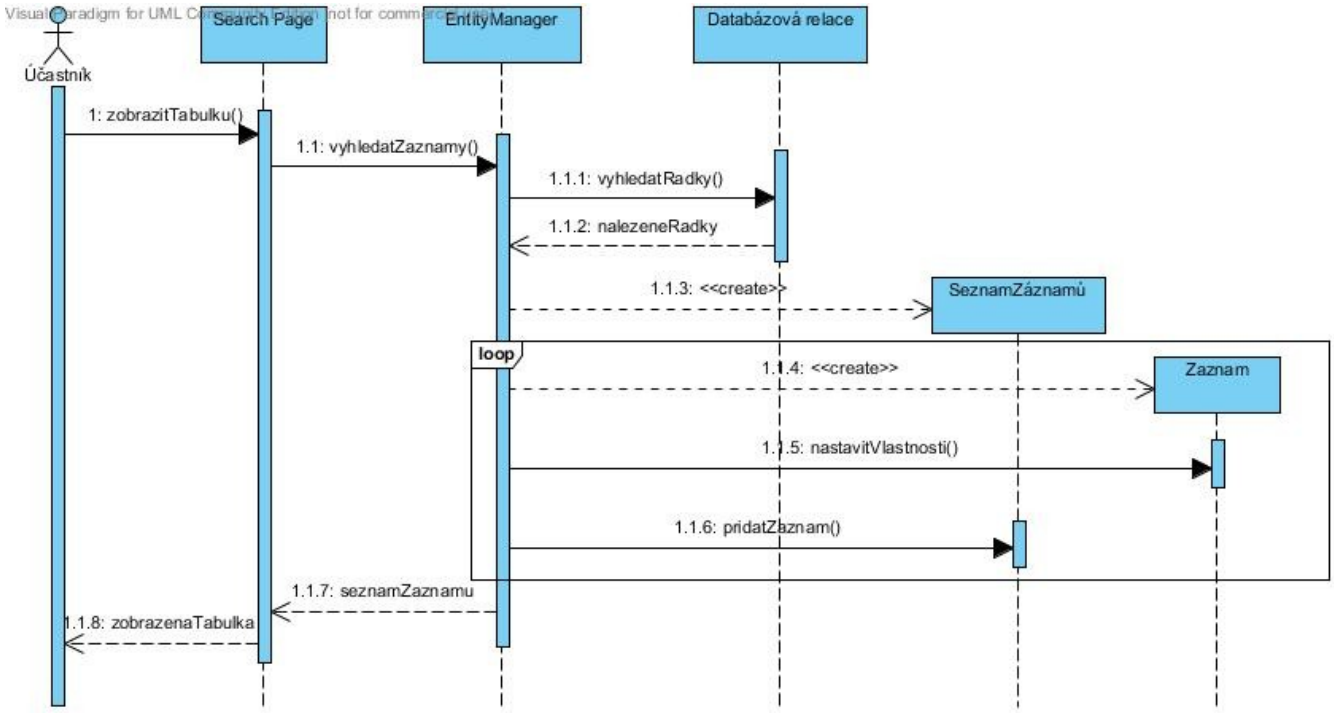
Diagramy spolupráce slouží k modelování strukturálních vztahů mezi objekty. Objekty jsou spojeny linkami, které symbolizují komunikační kanály sloužící pro přenos zpráv. Jednotlivé zprávy jsou hierarchicky strukturovány, aby byla vyjádřena posloupnost akcí (Arlow a Neustadt, 2005).



Obr. 3.3. Diagram spolupráce pro UC5

3.6.3 Sekvenční diagram pro zobrazení výsledků vyhledávání

Sekvenční diagramy jsou velmi podobné komunikačním diagramům, avšak na rozdíl od nich se zaměřují na časovou posloupnost jednotlivých zpráv. Tyto diagramy bývají sice často nejobsáhlejší ze všech diagramů interakcí, které nabízí specifikace UML2, ale zároveň i nejoblíbenější mezi uživateli díky jednoduchosti konceptu diagramu. U sekvenčních diagramů je časová osa vertikální (tok času plyne shora dolů) a jednotlivé čáry života (modelované objekty) jsou zde umístěny v horizontální linii. Toto uskupení minimalizuje překřížení čar v diagramu (Arlow a Neustadt, 2005).



Obr. 3.4. Sekvenční diagram pro UC5

3.7 Objektový a relační model

3.7.1 Analýza podstatných jmen a sloves

Analýza podstatných jmen a sloves je velice jednoduchý nástroj pro nalezení tříd, atributů a odpovědností. Používá se dlouhá léta a je postavena na přímé analýze jazyka problémové domény. Musíme však dávat pozor na synonyma a homonyma v popisu, neboť mohou vést k nesprávné definici tříd. Důležité je plně rozumět problémové doméně, jinak je zapotřebí shromáždit co nejvíce možných informací, které bychom mohli využít. Problémem analýzy podstatných jmen a slovek jsou skryté třídy, které nejsou v textu zachytitelné, avšak v doméně existují. Záleží tedy na zkušenostech analytika, jakým způsobem využije analýzu podstatných jmen a sloves k modelování tříd a jejich atributů (Arlow a Neustadt, 2005).

Jednotlivé **aplikace** využívající systém, budou zaznamenávat **události**, přičemž se jednotlivé záznamy uloží do databáze pod svým jedinečným **identifikačním číslem. Uživatel** má možnost *vyhledávat* jednotlivé **záznamy** na základě **aplikace, typu události**, **výrobního závodu** a **data zaznamenání**. **Systém** *nezaznamenává* pouze **události**, ale také zpracovává statistiky **počítačů**, na kterých *jsou* **aplikace** *spouštěny*, **běhových prostředí**, které představuje **operační systém** a **verze JRE**,

a **uživatelích** *používajících* logované aplikace. **Uživatelé** logovacího systému budou mít možnost, na základě **uživatelských práv**, *nastavovat* **logování** pro jednotlivé **aplikace** či **typy událostí**.

Kandidát na třídu	**Kandidát na atribut**	**Odpovědnost**
LogRecord (záznam)	ID Výrobní závod Logovaná aplikace Typ události Zpráva události Počítač Běhové prostředí Uživatel Čas záznamu	Evidence záznamu
Computer (počítač)	Název počítače	Evidence počítačů, na kterých běží logované aplikace
LogUser (uživatel logované aplikace)	ID uživatele	Evidence uživatelů používajících protokolované aplikace
ApplicationUser (uživatel IS)	ID uživatele Práva	Vyhledávání záznamů Správa uživatelů Nastavení logování
Environment (běhové prostředí)	Operační systém Verze JRE	Evidence běhového prostředí aplikací
FAB (výrobní závod)	Název závodu	Evidence výrobních závodů, ve kterých se provozují logované

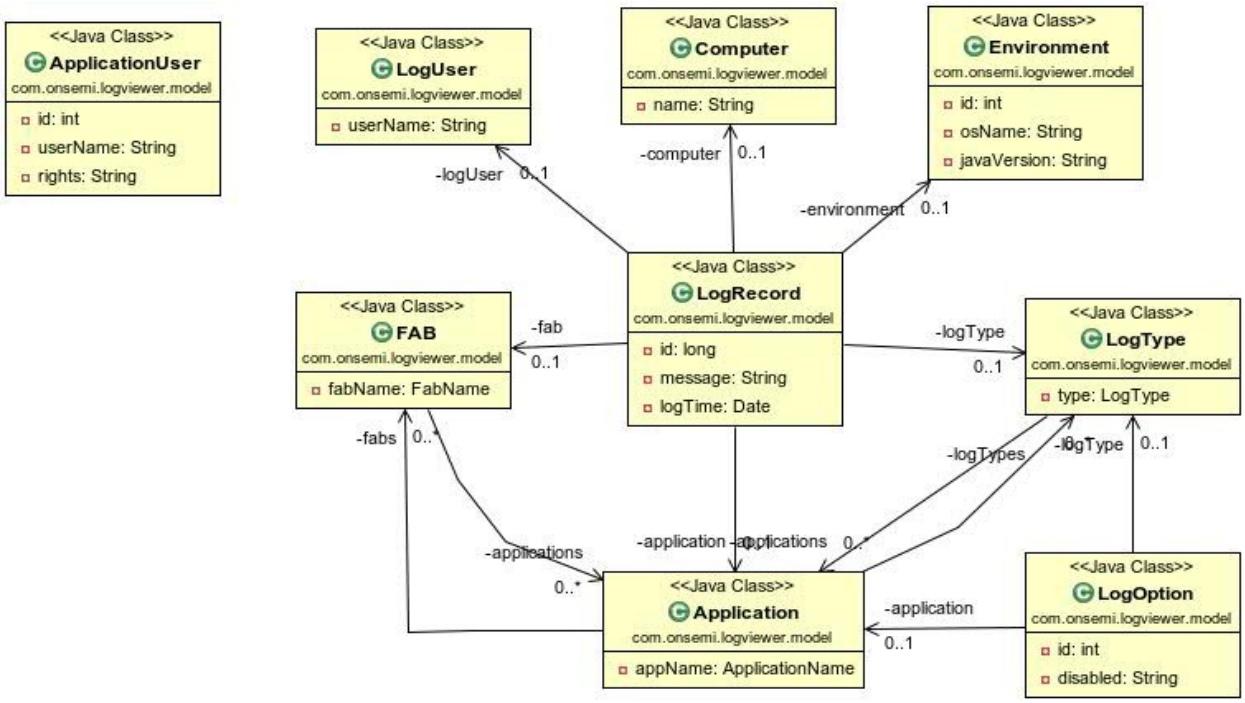
		aplikace
Application (aplikace)	Název aplikace	Evidence logovaných aplikací
LogOption (nastavení logování)	Aplikace Typ události Nastavení logování	Evidence jednotlivých nastavení.
LogType (typ události)	Typ události	Evidence typu událostí

Tab. 3.23. Analytické třídy

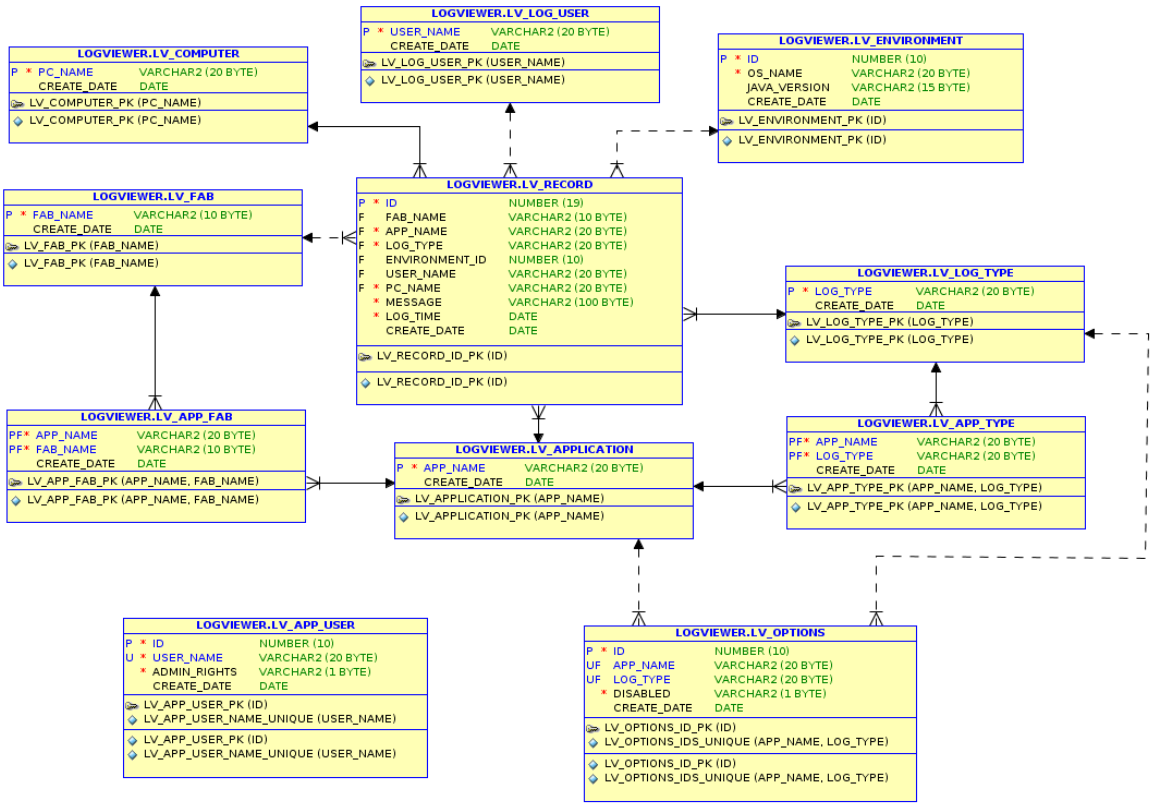
3.7.2 Neshoda paradigmat v praxi

Při pohledu na objektový a relační model aplikace okamžitě vidíme problém neshody paradigmat. Jednotlivé třídy nemusí zdaleka přesně reflektovat tabulky v databázi, navíc se zcela jinak modelují jednotlivé relace, především relace typu M:N. U relačního modelu je nutné vytvořit tzv. slabou entitu, u modelu objektového ovšem vytváříme relaci mezi třídami bez slabé entity.

Třídy zobrazené v diagramu představují perzistentní třídy v aplikaci, které spravuje Entity Manager (součást frameworku Hibernate).



Obr. 3.5. Objektový model aplikace



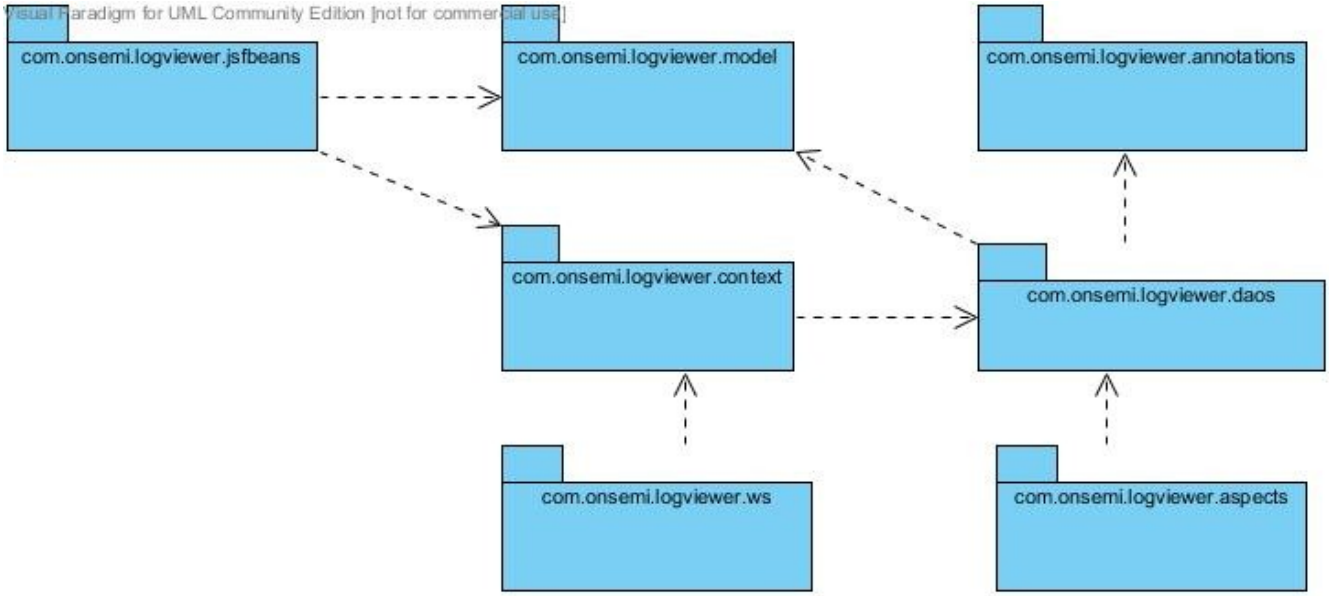
Obr. 3.6. Relační model aplikace

3.8 Návrh balíčků aplikace

Při návrhu byla aplikace rozdělena do následujících balíčků:

- **com.onsemi.logviewer.annotations** – anotace používané v aplikaci,
- **com.onsemi.logviewer.aspects** – balíček obsahující aspektové třídy,
- **com.onsemi.logviewer.context –** třídy pro přístup ke kontejneru Spring Frameworku,
- **com.onsemi.logviewer.daos –** třídy pro přístup k databázi,
- **com.onsemi.logviewer.jsfbeans –** implementace technologie JSF,
- **com.onsemi.logviewer.model** – balíček obsahující perzistentní třídy,
- **com.onsemi.logviewer.ws** – tento balíček obsahuje webové služby pro komunikaci s logovacím podsystémem.

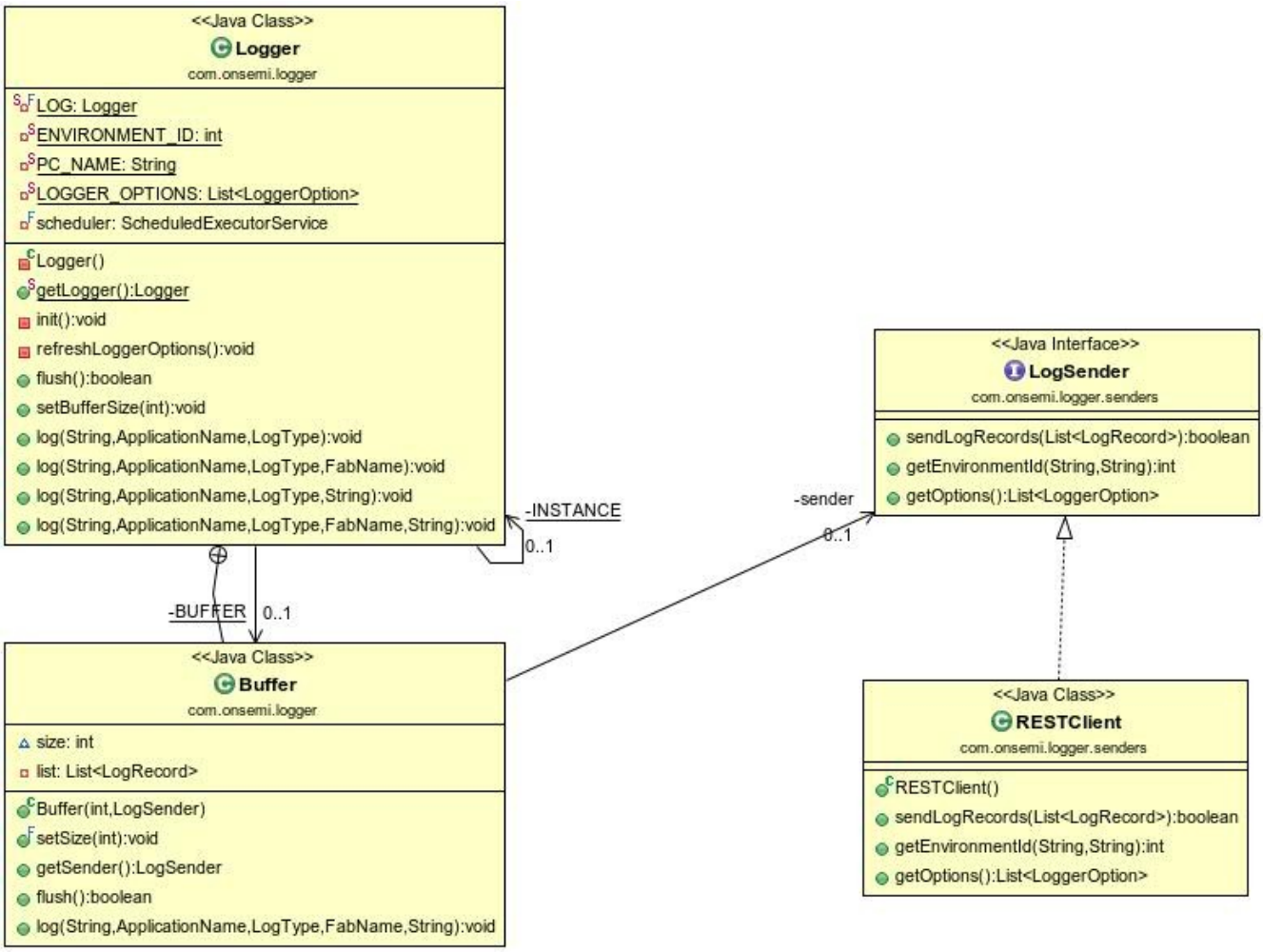
Závislosti mezi jednotlivými balíčky jsou patrné z diagramu balíčků. V diagramu balíčků se nevyskytuje cyklická závislost, která by znamenala důvod pro sloučení balíčku do jednoho. Balíčky proto budou implementovány podle tohoto návrhu.



Obr. 3.7. Diagram balíčků

3.9 Návrh logovací částí systému

Logovací část systému (logger) je mechanismus pro přímý sběr protokolovaných událostí. Obsahuje (zatím) pouze čtyři jednoduché třídy, čímž je splněn požadavek na lightweight koncepci. Třída Logger je singleton obsahující metody pro záznam událostí a metodu pro explicitní vyprázdnění bufferu. Buffer je vnořená třída, která udržuje seznam jednotlivých záznamů. Při dosažení stanovené velikost seznamu dojde samovolně k jeho vyprázdnění přes rozhraní LogSender. V současné době je k dispozici pouze implementace tohoto rozhraní na základě RESTful webových služeb. Do budoucna se uvažuje o rozšíření zasílání pomocí emailů. To byl jeden z mnoha důvodů, proč bylo odesílání řešeno pomocí rozhraní.



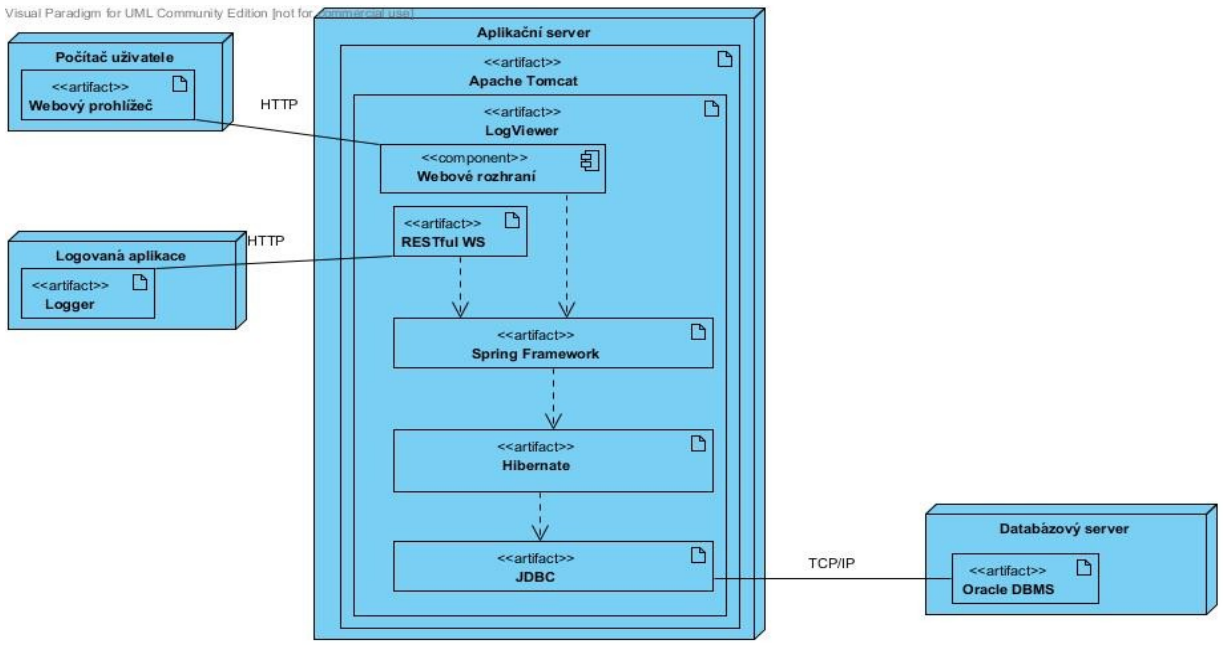
Obr. 3.8. Diagram návrhových tříd logovací části

4 Implementace a zhodnocení

V této kapitole je popsána problematika implementace navrženého systému. Je zde popsáno programování business logiky a zdůvodnění architektury systému. Uvedený zdrojový kód slouží k snadnému popsání problematiky programování tohoto systému a k ukázce používání moderních technologií jako například Hibernate či Spring Framework. Při psaní programového kódu byla použita metodika programování řízeného testy, tzv. Test Driven Development(TDD). V malé podkapitole ke konci je konkrétní ukázka praktikování TDD ve vývojovém prostředí NetBeans IDE.

4.1 Diagram nasazení

Logovací framework je rozdělen do dvou částí – logger a logviewer. Logger je knihovna, která poskytuje aplikacím API pro logování požadovaných událostí. LogViewer je webová aplikace sloužící jako prostředník mezi loggerem a databází. Tato aplikace obsahuje modul webových služeb, který zajišťuje komunikaci mezi oběma subsystémy, a modul webového uživatelského rozhraní. V rámci webové aplikace navíc funguje správa uživatelů a jejich práv pro editaci logovacích nastavení. Oběma modulům poskytuje Spring Framework instance servisních objektů, které zprostředkovávají komunikaci s perzistenční vrstvou aplikace. Životní cyklus perzistentních objektů zajišťuje Hibernate, který skrze nízkoúrovňové rozhraní JDBC komunikuje s databází.



Obr. 4.1. Diagram nasazení

4.2 Příprava databáze

V rámci databáze Oracle byl vytvořen vlastní tabulkový prostor pro aplikaci. Jednotlivé tabulky odpovídají relačnímu návrhovému modelu.

4.2.1 Tabulky

Pro vytvoření tabulek byl klasicky použit Data Definition Language standardu SQL. Pro ukázku je přiložen kód pro vytvoření tabulky, ve které se uchovávají jednotlivé záznamy. Každý záznam obsahuje své ID (systémové), kterému orientačně odpovídá délka datového typu long v jazyce Java. Jednotlivé řádky tabulky jsou pro potřeby vývojového týmu opatřeny datem vložení řádku do tabulky, nikoli jen datem vzniku záznamu.

```
CREATE TABLE LV_RECORD
  (
    ID             NUMBER(19) NOT NULL,
    FAB_NAME       VARCHAR2(10),
    APP_NAME       VARCHAR2(20) NOT NULL,
    LOG_TYPE       VARCHAR2(20) NOT NULL,
    ENVIRONMENT_ID NUMBER(10),
    USER_NAME      VARCHAR2(20),
    PC_NAME        VARCHAR2(20) NOT NULL,
    MESSAGE        VARCHAR2(100) NOT NULL,
    LOG_TIME       DATE NOT NULL,
    CREATE_DATE    DATE,
    CONSTRAINT LV_RECORD_ID_PK PRIMARY KEY (ID) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_RECORD_FAB_ID_FK FOREIGN KEY (FAB_NAME) REFERENCES
LV_FAB(FAB_NAME),
    CONSTRAINT LV_RECORD_APP_ID_FK FOREIGN KEY (APP_NAME) REFERENCES
LV_APPLICATION(APP_NAME),
    CONSTRAINT LV_RECORD_LOG_TYPE_ID_FK FOREIGN KEY (LOG_TYPE) REFERENCES
LV_LOG_TYPE(LOG_TYPE),
    CONSTRAINT LV_RECORD_ENVIRONMENT_ID_FK FOREIGN KEY (ENVIRONMENT_ID)
REFERENCES LV_ENVIRONMENT(ID),
    CONSTRAINT LV_RECORD_USER_ID_FK FOREIGN KEY (USER_NAME) REFERENCES
LV_LOG_USER(USER_NAME),
    CONSTRAINT LV_RECORD_COMPUTER_ID_FK FOREIGN KEY (PC_NAME) REFERENCES
LV_COMPUTER(PC_NAME)
  );
```

Hodnoty do sloupce ID vytváří definovaná sekvence.

```
CREATE SEQUENCE LV_RECORD_SEQ MINVALUE 1 MAXVALUE 9999999999999999999 START
WITH 1 INCREMENT BY 1;
```

V rámci databáze jsou na některé tabulky aplikovány triggery typu before insert. Ty mají za úkol ověřovat, zda vkládaný řádek obsahuje hodnotu sloupce ID a datum vzniku řádku. V případě neexistence se hodnota sloupce ID se vezme z příslušné sekvence. Pro definování data vytvoření řádku je použit aktuální čas systému.

```
CREATE OR REPLACE TRIGGER BI_LV_RECORD BEFORE
  INSERT ON LV_RECORD REFERENCING NEW AS NEW FOR EACH ROW DECLARE id
NUMBER;
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
    IF :new.id IS NULL THEN
      SELECT LV_RECORD_SEQ.NEXTVAL INTO id FROM dual;
      :new.id:=id;
    END IF;
  END;
  /
```

4.3 Logger

Aplikace, kterou je zapotřebí protokolovat, musí mít vytvořenou kompilační závislost na knihovně **logger.jar.** Tato knihovna obsahuje tu část logovacího frameworku, která poskytuje API pro protokolování požadovaných událostí.

4.3.1 Kontrolování nastavení logování

Mezi požadavky bylo zařazeno i centrální vypínání (či zapínání), logování jednotlivých aplikací, případně typů událostí. Toto je velice jednoduché vyřešit, pokud při inicializaci loggeru (např. spuštění aplikace) získáme nějakým způsobem nastavení. Inicializace ovšem probíhá pouze jednou v rámci běhu aplikace. U desktopových aplikací to nepředstavuje problém, ale u serverových webových aplikací je požadován non-stop provoz. Vypínání či zapínání logování by tudíž bylo naprosto nepoužitelné. Řešením problému je cyklické kontrolování logovacích pravidel.

Pro opožděné, či periodicky volané příkazy můžeme od verze JDK 1.5 využít třídu **ScheduledExecutorService** z balíčku **java.util.concurrent**. Objekt vrácený zpracovanou úlohou může být použit pro kontrolu nebo i pro stornování volaných úloh. V rámci této třídy nemusíme pracovat pouze s jediným vláknem, nýbrž můžeme použít libovolný počet vláken určených ke zpracování úloh. To je velmi výhodné při náročných paralelních, cyklicky volaných operacích.

Kód níže reflektuje práci s touto třídou. Instanci třídy získáme zavoláním statické metody **newScheduledThreadPool(int corePoolSize)** třídy **java.util.concurrent.Executors**. Za parametr metody dosadíme počet vláken, která se mají v poolu uchovávat i v době jejich neaktivity. Metoda **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)** instance třídy **ScheduledExecutorService** nám zajistí opakované spouštění příslušného objektu Runnable (implementace rozhraní **java.lang.Runnable**) dle zadaných parametrů (doba zpoždění, doba periody a zvolení příslušné časové jednotky). V kódu vidíme, že se metoda **refreshLoggerOptions()** objektu **Logger** bude volat každých patnáct minut.

```
private final ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(1);

    private Logger() {
    }

    public static synchronized Logger getLogger() {
        if (INSTANCE == null) {
            INSTANCE = new Logger();
            BUFFER = new Buffer(100, new RESTClient());
            INSTANCE.init();

            final Runnable check = new Runnable() {
                @Override
                public void run() {
                    INSTANCE.refreshLoggerOptions();
                }
            };

            INSTANCE.scheduler.scheduleAtFixedRate(check, 15, 15,
TimeUnit.MINUTES);
        }
        return INSTANCE;
    }
```

4.3.2 Logování událostí

Samotné logování událostí je v principu jednoduché, což odpovídá požadavku na snadnost použití. Pro logování slouží metoda **log()** třídy **Logger**. Tato metoda je několikrát přetížená v závislosti na parametrech záznamu. Ve výsledku se ovšem vždy volá konkrétní metoda níže. Při zavolání se ověří, zda je možné logovat dotyčnou aplikaci a typ události. Následně se záznam přidá do zásobníku, který je

reprezentován třídou Buffer. Jedná se o malou třídu obsahující kolekci záznamů a metody pro přidání záznamu a vyprázdnění zásobníku.

```
public void log(String message, ApplicationName appName, LogType logType,
FabName fabName, String userName) {
        boolean canLog = true;

        for(LoggerOption o : LOGGER_OPTIONS){
            if(appName.equals(o.getAppName()) &&
logType.equals(o.getLogType()) && !o.isEnabled()){
                canLog = false;
                break;
            }
        }

        if (canLog) {
            BUFFER.log(message, appName, logType, fabName, userName);
        }
}
```

4.4 Logviewer

Logviewer je druhý subsystém logovacího frameworku. Jedná se plnohodnotnou webovou aplikaci, která poskytuje rozhraní pro příjem záznamů z jednotlivých aplikací a poskytování nastavení.

4.4.1 Deklarace business logiky

Životní cyklus objektů business logiky aplikace spravuje Spring Framework. Díky Inversion of Control mechanismu odstraníme kompilační závislosti jednotlivých business objektů. Tím docílíme modularizace celé aplikace s možností snadných úprav při budoucím vývoji. Pro vývojáře se jedná o obrovskou výhodu, jelikož veškeré konfigurační údaje se nachází na jednom místě, mimo zdrojový kód aplikace.

4.4.1.1 Database access objects v režii Spring Frameworku

Správným návykem pro přístup k databázi je vytvoření tzv. DAOs – database access objects. Tyto objekty používáme pro operace s perzistentními objekty aplikace. Jedná o služby, které, nám přes určité rozhraní poskytují metody pro databázové operace (např. klasické create, read, update & delete operace). Vzhledem k tomu, že se jedná o služby (v logickém kontextu), je důležité svěřit jejich životní cyklus IoC kontejneru a nestarat se o vytváření jejich instancí (Fisher a Murphy, 2010).

Níže uvedený úryvek z konfiguračního XML souboru Spring Frameworku ukazuje deklaraci database access objektů v aplikaci. Všimněme si, že je důležité vytvářet instanci třídy implementující

rozhraní, ne rozhraní samotné. V aplikaci ovšem přistupujeme k objektu přes jeho rozhraní (viz kapitola o implementaci DAO).

```
<bean id="applicationDao"
class="com.onsemi.logviewer.daos.ApplicationDAOImpl" />
<bean id="applicationUserDao"
class="com.onsemi.logviewer.daos.ApplicationUserDAOImpl" />
<bean id="computerDao" class="com.onsemi.logviewer.daos.ComputerDAOImpl" />
<bean id="environmentDao"
class="com.onsemi.logviewer.daos.EnvironmentDAOImpl" />
<bean id="fabDao" class="com.onsemi.logviewer.daos.FABDAOImpl" />
<bean id="logRecordDao" class="com.onsemi.logviewer.daos.LogRecordDAOImpl"
/>
<bean id="logTypeDao" class="com.onsemi.logviewer.daos.LogTypeDAOImpl" />
<bean id="logUserDao" class="com.onsemi.logviewer.daos.LogUserDAOImpl" />
```

4.4.1.2 Connection pooling

U některých objektů je velice náročné (z hlediska zdrojů) vytváření nových instancí a jejich následná likvidace. Proto se naskýtá myšlenka znuvupoužitelnosti objektů. V takovém případě objekt vytvoříme a trvale jej uchováme v paměti počítače, často při startu aplikace. Při zažádání o objekt nevytváříme novou instanci, ale poskytneme odkaz na instanci již vytvořenou. Problém nastává v okamžiku, kdy chceme použít více než jednu instanci. V takovém případě musíme využít mechanismus zvaný pooling. Pooling představuje filosofii správy objektů. Při potřebě objektu si příslušnou instanci vytáhneme z poolu (kolekce objektů). Jakmile skončíme práci, objekt neničíme, ale vrátíme jej zpět do kolekce. Takové objekty můžeme využívat napříč aplikací a tím snížíme náročnost na paměť a procesorový čas (Sridhar, 2009).

Obecně vytváření a uzavírání konektivit stojí mnoho času. Proto se v praxi používá právě na spravování jednotlivých konektivit pooling. Bavíme se v tomto případě o pojmu **Connection Pooling.** Otázkou zůstává jakým způsobem vytvořit takový pool. Správu celého životního cyklu poolu bychom měli svěřit IoC kontejneru. V takovém případě nebudeme mít na příslušný connection pool kompilační závislost a můžeme jednotlivé implementace poolu zaměňovat.

To může být v některých případech obrovská výhoda. Jednotlivé implementace vykazují různou výkonnost na různých platformách. V případě nízkého výkonu (nebo jiných důvodů) můžeme, bez větších zásahů do aplikace, celý pool vyměnit za jinou implementaci (Hanik, 2010).

Kód níže ilustruje deklaraci connection poolu (konkrétně Apache JDBC-Pool) v rámci Spring Frameworku. Beanu deklarativně nastavíme příslušné vlastnosti, konkrétně výchozí velikost poolu, maximální počet aktivních konektivit a přístupové údaje k databázi.

```
<bean id="myDataSource" class="org.apache.tomcat.jdbc.pool.DataSource"
destroy-method="close">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
        <property name="initialSize" value="10" />
        <property name="maxActive" value="50" />
</bean>
```

4.4.1.3 Hibernate

Při konfiguraci ORM frameworku Hibernate názorně vidíme výhody IoC přístupu. Spring je naprosto ideální pro spravování životního cyklu SessionFactory objektu, který se stará o poskytování instancí session (sezení) pro komunikaci s databází. Tomuto objektu deklarativně nastavíme námi vytvořený datový zdroj (datasource), aniž bychom napsali jediný řádek zdrojového kódu. V budoucnu můžeme vyměnit datový zdroj za jiný bez zásahu do zdrojového kódu aplikace, stačí změnit XML konfigurační soubor (JavaBeat, 2007).

Kromě datového zdroje je zapotřebí nastavit tzv. dialekt. Jelikož Hibernate používá objektový dotazovací jazyk HQL, musíme jasně definovat, pro jakou relační databázi se má z příslušného HQL dotazu generovat SQL dotaz. Pro usnadnění vývoje aplikace je výhodné zaznamenávat generovaný SQL kód do logu aplikačního serveru, což umožníme nastavením vlastnosti **hibernace.show_sql** na **true**.

I když Hibernate umožňuje mapování jednotlivých perzistentních tříd pomocí XML mapovacích souborů, je určitě modernější a vhodnější způsob mapování pomocí anotací. Použití anotací vede k mnohem vyšší přehlednosti zdrojového kódu a v neposlední řadě se jedná o čistě „Java“ přístup. Anotace existují v JDK od verze 1.5. Aby se předcházelo neefektivního prohledávání celého CLASSPATH kvůli perzistentním třídám, sdělíme frameworku Hibernate konkrétní třídy, které bude spravovat.

```
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
```

```
        <property name="hibernateProperties">
            <value>
                hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
                hibernate.show_sql=true
            </value>
        </property>
        <property name="annotatedClasses">
            <list>
                <value>com.onsemi.logviewer.model.Application</value>
                <value>com.onsemi.logviewer.model.ApplicationUser</value>
                <value>com.onsemi.logviewer.model.Computer</value>
                <value>com.onsemi.logviewer.model.Environment</value>
                <value>com.onsemi.logviewer.model.FAB</value>
                <value>com.onsemi.logviewer.model.LogRecord</value>
                <value>com.onsemi.logviewer.model.LogType</value>
                <value>com.onsemi.logviewer.model.LogUser</value>
                <value>com.onsemi.logviewer.model.LogOption</value>
            </list>
        </property>
    </bean>
```

4.4.1.4 Deklarace aspektu

Deklarování aspektu v kontextu Spring Frameworku se provádí de facto stejně jako deklarace Spring Beans. Vytvoříme bean, který bude aspekt představovat, to znamená, že bude obsahovat alespoň jednu metodu s kódem, který se bude injektovat na příslušný **join point** (bod připojení kódu). Pro tento mechanismus existuje v angličtině výraz „**aspect weaving**“. Značkou **<aop:aspect>** sdělíme Spring kontejneru, že daný bean představuje aspekt a definujeme u něj volanou metodu, **advice** a **pointcut**. Advice představuje implementaci injektovaného kódu a přesný okamžik jeho připojení. Poincut slouží k výběru množiny prvků, které chceme aspektem ovlivnit. V ukázce kódu vidíme deklarovaný aspekt, jehož metoda **manageSession()** „obalí“ metody označené anotací **@DBOperation**, bez ohledu na návratový typ či parametry.

```
<bean id="daoAspectBean" class="com.onsemi.logviewer.aspects.DAOAspect">
        <property name="sessionFactory" ref="mySessionFactory" />
    </bean>

    <aop:config>
        <aop:aspect id="daoAspect" ref="daoAspectBean">
            <aop:around method="manageSession" pointcut="execution
(@com.onsemi.logviewer.annotations.DBOperation * *(..))" />
        </aop:aspect>
    </aop:config>
```

4.4.2 Implementace business logiky

V minulé kapitole jsme se věnovali deklaraci business logiky, včetně deklarace aspektu v rámci kontextu Spring Frameworku. Nyní si ukážeme konkrétní implementaci aspektu. Aspekt je klasická třída obsahující metody a případně vlastnosti. V rámci tohoto aspektu jsme deklarovali advice typu around, to znamená, že příslušná metoda „obalí“ metodu ovlivněnou. Tato metoda obsahuje parametr typu **ProceedingJoinPoint**, jehož instanci nám automaticky podsune Spring Framework. Z tohoto objektu můžeme získat instanci třídy obsahující ovlivněný kód. Vlastní vykonání metody se provádí zavoláním **proceed()** metody objektu **ProceedingJoinPoint**. Tato metoda nám vrátí typ **Object**, ve kterém je uložena návratová hodnota ovlivněné metody. V případě naší aplikace aspekt zajišťuje otevírání a uzavírání sezení, včetně správy transakcí jednotlivých metod, vyžadujících přístup do databáze. Po získání návratového objektu s klidem potvrdíme transakci a synchronizujeme a uzavřeme session.

```
public class DAOAspect {

    private SessionFactory sessionFactory;

    public Object manageSession(ProceedingJoinPoint jp) throws Throwable {
        DAO dao = (DAO) jp.getThis();

        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        dao.setSession(session);

        Object retVal = jp.proceed();

        dao.setSession(null);
        if (tx.isActive()) {
            tx.commit();
        }

        if(session.getTransaction().isActive()){
            session.getTransaction().commit();
        }
        session.flush();
        session.close();

        return retVal;
    }

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
```

```
    }
}
```

Zdrojový kód třídy se díky přesunutí opakovaného kódu do aspektu krásně zeštíhlí. Všimněme si především, že se vůbec nestaráme o vytváření instance session objektu, pouze jej využíváme. Veškeré transakce nám v rámci sezení spravuje také aspekt. Tento přístup je krásnou ukázkou vytváření znovupoužitelného kódu, který explicitně nevoláme. To vede k zeštíhlení zdrojového kódu a mnohem větší odolnosti proti chybám – stačí jednou dobře napsat aspekt.

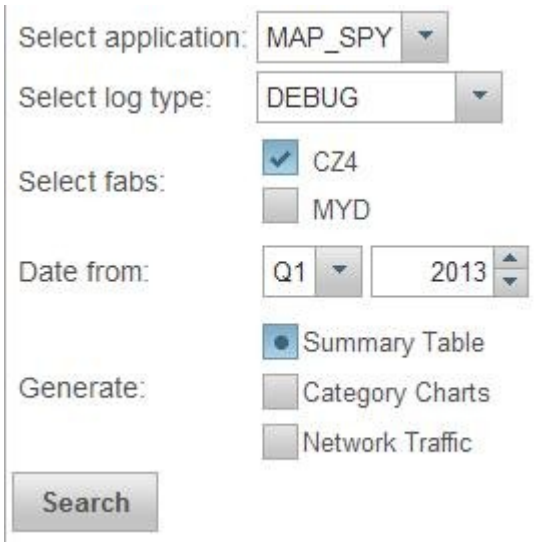
```
public class ApplicationDAOImpl implements ApplicationDAO {

    private Session session;

    @Override
    public void setSession(Session session) {
        this.session = session;
    }

    @DBOperation
    @Override
    public List<Application> allApplications() {
        return (List<Application>) session.createQuery("from
Application").list();
    }

    @DBOperation
    @Override
    public void saveApplication(Application app) {
        session.save(app);
    }
}
```

4.4.3 GUI

Grafické uživatelské rozhraní bylo implementování pomocí technologie JavaServer Faces. Vzhledem k nutnosti využití složitějších komponent byl k vývoji využit framework PrimeFaces.

4.4.3.1 Vyhledávací formulář

Uživatelé využívají tento formulář pro vyhledávání jednotlivých záznamů. Vyhledávání probíhá na základě aplikace a typu události. Výsledky hledání jsou omezeny časem a výrobním závodem. Uživatel si zvolí, jakou formu výstupu hodlá zobrazit (tabulku, grafy či přehled provozu webových služeb). K vytvoření formuláře bylo využito komponent z balíčku PrimeFaces, které rozšiřují klasické JSF komponenty.



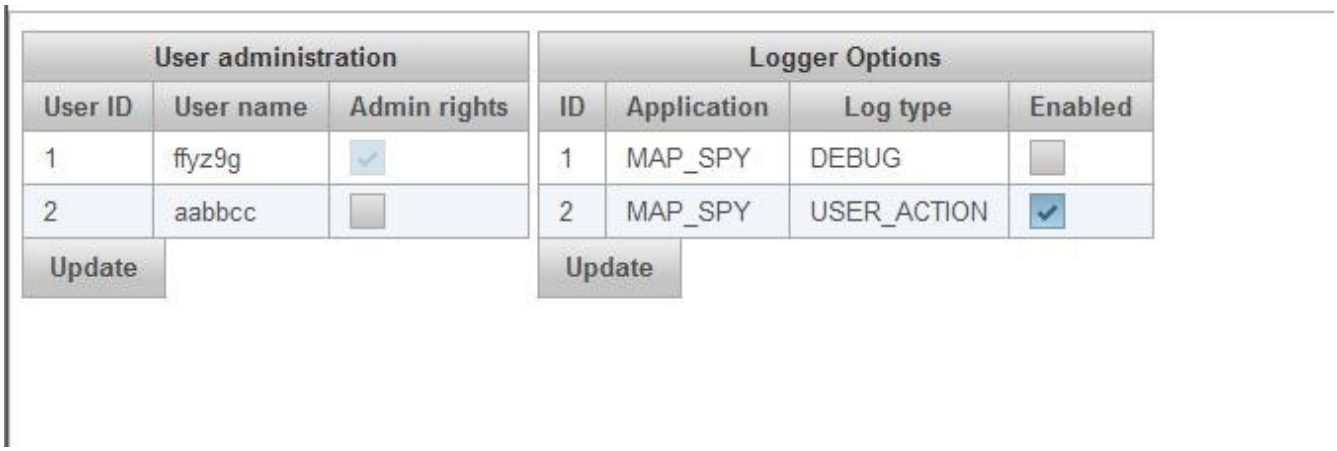
Obr. 4.2. Formulář pro vyhledávání

```
<h:form id="searchForm">
                    <h:messages />
                    <h:panelGrid columns="2">
                        <p:outputLabel value="Select application:" />
                        <p:selectOneMenu
value="#{searchBean.selectedApplication}">
                            <f:selectItems
value="#{searchBean.allApplications}" var="_app"
itemValue="#{_app.appName}" itemLabel="#{_app.appName}" />
                        </p:selectOneMenu>
                        <p:outputLabel value="Select log type:" />
                        <p:selectOneMenu
value="#{searchBean.selectedLogType}">
                            <f:selectItems value="#{searchBean.logTypes}"
var="_type" itemValue="#{_type.type}" itemLabel="#{_type.type}" />
                        </p:selectOneMenu>
                        <p:outputLabel value="Select fabs:" />
                        <p:selectManyCheckbox
value="#{searchBean.selectedFabs}" layout="pageDirection">
                            <f:selectItems value="#{searchBean.allFabs}"
var="_fab" itemValue="#{_fab.fabName}" itemLabel="#{_fab.fabName}"/>
                        </p:selectManyCheckbox>

                        <p:outputLabel value="Date from:" />
                        <h:panelGrid columns="2">
                            <p:selectOneMenu
value="#{searchBean.selectedQuarter}" >
                                <f:selectItem itemValue="1" itemLabel="Q1"
/>
                                <f:selectItem itemValue="2" itemLabel="Q2"
```

```
/>
                                    <f:selectItem itemValue="3" itemLabel="Q3"
/>
                                    <f:selectItem itemValue="4" itemLabel="Q4"
/>
                                </p:selectOneMenu>
                                <p:spinner value="#{searchBean.selectedYear}"
size="4" min="1900" max="#{searchBean.actualYear()}" maxlength="4" />
                            </h:panelGrid>
                            <p:outputLabel value="Generate: " />
                            <p:selectOneRadio
value="#{searchBean.generationSelected}" layout="pageDirection">
                                <f:selectItem itemValue="1" itemLabel="Summary
Table" />
                                <f:selectItem itemValue="2" itemLabel="Category
Charts" />
                                <f:selectItem itemValue="3" itemLabel="Network
Traffic" />
                            </p:selectOneRadio>
                        </h:panelGrid>

                        <p:commandButton ajax="false" value="Search"
action="#{searchBean.search()}"/>
                    </h:form>
```

4.4.3.2 Nastavení loggeru a správa uživatelů

Následující obrázek ukazuje možnosti pro správu uživatelů a nastavení loggeru.



Obr. 4.3. Formulář pro správu uživatelů a nastavení

V ukázce kódu si všimněme konkrétní realizaci formulářů.

```
<h:form>
  <p:dataTable  value="#{administrationBean.allAppUsers}" var="appUser" >
      <f:facet name="header">User administration</f:facet>
```

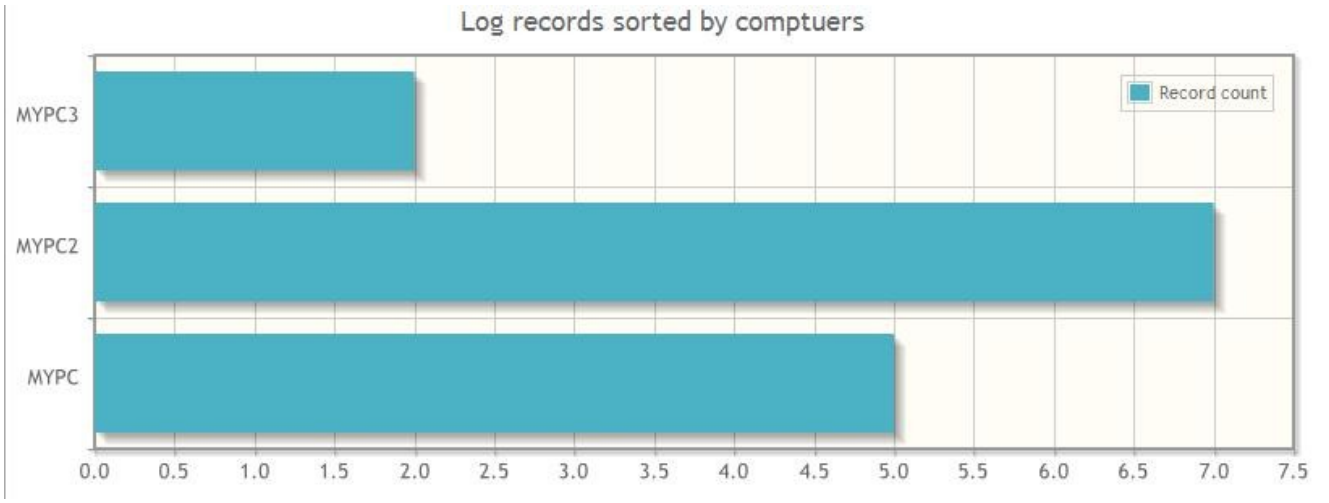
```
          <p:column headerText="User ID">#{appUser.id}</p:column>
          <p:column headerText="User name">#{appUser.userName}</p:column>
          <p:column headerText="Admin rights">
          <p:selectBooleanCheckbox
valueChangeListener="#{administrationBean.setActualUser(appUser)}"
value="#{appUser.admin}"
disabled="#{administrationBean.enableAdminCheckBox(appUser)}" />
          </p:column>
  </p:dataTable>
  <p:commandButton action="#{administrationBean.updateUsers()}"
value="Update" ajax="false"/>
</h:form>
```

4.4.3.3 Výsledky vyhledávání

Výsledky vyhledávání si uživatel může nechat zobrazit v tabulce či jako graf. V obou případech se o vykreslení stará framework PrimeFaces a to konkrétně komponenta **DataTable** a **BarChart**. Komponenta DataTable je rozšířením klasické JSF komponenty, avšak nabízí možnost filtrování, třídění, vkládání součtových řádků apod. BarChart vykresluje do webové stránky klasický sloupcový graf na základě modelu, který komponentě předáme.

ID	Message	Log type	Application	PC Name	FAB	User	Date
25	debug log	DEBUG	MAP_SPY	MYPC	CZ4		2013-04-10
34	debug pc3	DEBUG	MAP_SPY	MYPC3	CZ4		2013-04-10
35	debug pc3	DEBUG	MAP_SPY	MYPC3	CZ4		2013-04-10
36	pc3 debug log	DEBUG	MAP_SPY	MYPC3	CZ4		2013-04-10

Obr. 4.4. Tabulka záznamů



Obr. 4.5. Graf agregovaných dat

V ukázce níže vidíme vytváření modelu pro sloupcový graf.

```
public CartesianChartModel getSortedByFabsModel() {
        CartesianChartModel model = new CartesianChartModel();
        ChartSeries s = new ChartSeries("Record count");
        for (String fabName : selectedFabs) {

            int count = 0;
            for (LogRecord r : recordsFound) {
              if(r.getFab().getFabName().equals(FabName.valueOf(fabName))){
                    count += 1;
                }
            }

            s.set(fabName, count);
        }
        model.addSeries(s);
        return model;
    }
```

4.4.3.4 Správa přihlášených uživatelů

Vzhledem k požadavku o přihlašování do aplikace pod uživatelským jménem a heslem bylo nutné do aplikace zakomponovat mechanismus, který by spravoval aktuálně přihlášené uživatele a nepřihlášeným by zamezil přístup. Za tímto účelem byl vytvořen JSF Managed Bean s rozsahem sezení (session scoped), který drží informace o přihlášeném uživateli pro každou session. V případě odhlášení uživatele stačí terminovat příslušné sezení a přesměrovat uživatele na přihlašovací stránku. Při nečinnosti uživatele je ve výchozím nastavení délka trvání sezení 30 minut.

```
public String logout() {
    if (user != null) {
        LOG.log(Level.INFO, "USER {0} logged out.", user.getUserName());
    }

FacesContext.getCurrentInstance().getExternalContext().invalidateSession();
return "login?faces-redirect=true";
}
```

4.5 Unit testing v NetBeans IDE

Programový kód byl psán za pomoci metodiky Test Driven Development. V praxi to znamenalo, že ke každé třídě byl nejdříve napsán test, než byl napsán vlastní kód třídy. Na základě výsledků testů se kód upravoval do podoby tak dlouho, dokud nebyly všechny testy úspěšné. V ukázce kódu vidíme test metody **getUser()**. Testovací metoda je označená anotací **@Test** z důvodu dynamického volání všech metod testovací třídy. V tomto konkrétním případě test projde, pokud bude porovnání objektů **userId** a **result.getUserName()** pravdivé. Toto porovnání se provádí v metodě **assertEquals(Object**

expected, Object result), která v případě nonekvivalence vyhodí chybu **AssertionFailedError.** Na základě této chyby stanovujeme výsledek testu.

```
@Test
public void testGetUser() {
    System.out.println("getUser");
    String userId = "ffyz9g";
    ApplicationUserDAOImpl instance = new ApplicationUserDAOImpl();
    Session session = sessionFactory.openSession();
    instance.setSession(session);

    ApplicationUser result = instance.getUser(userId);
    assertEquals(userId, result.getUserName());

    session.close();
    }
```

Vývojové prostředí NetBeans poskytuje vynikající podporu pro unit testing. V grafickém přehledu se dozvíme statistiku spuštěných testů a případné důvody selhání některých testů. Psaní programového kódu se stává bezpečnější, jelikož můžeme neustále programový kód ověřovat spouštěním testů.

Obr. 4.6. Test Driven Development v NetBeans IDE

4.6 Zhodnocení

Po nasazení logovacího frameworku má nyní vývojový tým v ruce informace o reálném využití interních aplikací společnosti. Na základě těchto informací se mohou kvalitněji rozhodovat o budoucím vývoji či změnách IT infrastruktury, které by pomohly ke zvýšení výkonu informačních systémů. Počítá se i se zvýšením uživatelského komfortu při práci s aplikacemi, jelikož má tým k dispozici

přesná data o užívání softwaru zaměstnanci společnosti, podle kterých se může následně ubírat vývoj grafického uživatelského rozhraní.

5 Závěr

Cílem práce bylo osvětlení problematiky vývoje informačních systému na platformě Java. Během mé stáže ve společnosti ON Semiconductor jsem dostal možnost vyvíjet informační systém od samého začátku. Využil jsem této příležitosti, abych se zdokonalil v analytickém myšlení a zjistil, jak se v praxi používá metodika Unified Process. Osvojil jsem si také programování řízené testy, které se ukázalo být obrovským pomocníkem při programování jednotlivých částí systému.

Celá práce byla kvůli co největší přehlednosti strukturována do tří kapitol tak, jak aby co nejvíce reflektovaly skutečný vývoj systému – od teorie směrem k praxi.

V teoretické části byly vyzdviženy hlavní problémy při navrhování systému, jako například problematika perzistence či přenosu dat mezi webovými službami. Pochopení těchto problematik bylo zcela zásadní pro budoucí vývoj aplikace. Dále byly vyjmenovány nejdůležitější moderní technologie, které při vývoji sehrály zásadní roli.

V druhé kapitole byly důsledně popsány uživatelské požadavky, které představují startovní pozici celé metodiky Unified Process. To posloužilo k následné analýze celé problémové domény a návrhu dvou aplikací – logger a logviewer.

Třetí kapitola završuje průřez vývojem systému. Názorně bylo ukázáno používání jednotlivých technologií v praxi, včetně zmínění dopadů jejich použití na aplikační logiku. Na závěr bylo přestaveno konkrétní použití metodiky Test Driven Development v prostředí NetBeans IDE.

Seznam použité literatury

APACHE SOFTWARE FOUNDATION, ©2008-2012. What's wrong with Hibernate and JPA. *Apache.org* [online]. [cit. 2013-02-25]. Dostupné z: http://empire-db.apache.org/empiredb/hibernate.htm

ARLOW, Jim a Ila NEUSTADT, 2005. *UML 2 and the unified process: practical object-oriented analysis and design.* 2nd ed. Boston: Addison-Wesley. ISBN 03-213-2127-8.

BAUER, Christian a Gavin KING, 2007. *Java Persistence with Hibernate*. Greenwich: Manning Publications. ISBN 1-932394-88-5.

BECK, Kent, 2003. *Test-driven development: by example*. Boston: Addison-Wesley. ISBN 03-211-4653-0.

BOOCH, Grady, Ivar JACOBSON a James RUMBAUGH, 2005. *The Unified Modeling Language Reference Manual*. 2nd ed. Boston: Addison-Wesley. ISBN 978-0321718952.

BURNS, Ed a Chris SCHALK, 2010. *JavaServer Faces 2.0, The Complete Reference*. New York: McGraw Hill. ISBN 978-0-07-162509-8.

FISHER, Paul T. a Brian D. MURPHY, 2010. *Spring Persistence with Hibernate*. New York: Apress. ISBN 978-1-4302-2632-1.

HALPIN, Terry a Tony MORGAN, 2008. *Information Modeling and Relational Databases.* 2nd ed. Burlington: Morgan Kaufmann Publishers. ISBN 978-0-12-373568-3.

HAMILTON, Kim a Russ MILES, 2006. *Learning UML 2.0*.Sebastopol: O'Reilly. ISBN 978-0596009823.

HANIK, Filip, 2010. Understanding the JDBC Pool Performance Improvements. In: *TomcatExpert.com* [online]. Mar 24, 2010 [cit. 2013-03-16]. Dostupné z: http://www.tomcatexpert.com/blog/2010/03/22/understanding-jdbc-pool-performance-improvements

HARROP, Rob a Clarence HO, 2012. *Pro Spring 3*. New York: Apress. ISBN 978-1-4302-4107-2.

HOOKOM, Jacob, 2005. Inside Facelets Part 1: An Introduction. In: *JSF Central* [online]. Aug 17, 2005 [cit. 2013-04-10]. Dostupné z: http://www.jsfcentral.com/articles/facelets_1.html

JAVABEAT, 2007. Spring and Hibernate ORM Framework Integration. *Javabeat.net* [online]. Oct 16, 2007 [cit. 2013-02-13]. Dostupné z: http://www.javabeat.net/2007/10/integrating-spring-framework-with-hibernate-orm-framework/

KALUŽA, Jindřich, 2010. *Informační systémy pro strategické řízení*. Ostrava: VŠB-TU Ostrava. ISBN 978-80-248-2280-8.

LADDAD, Ramnivas, 2010. *AspectJ in Action: Enterprise AOP with Spring Applications*. Greenwich: Manning Publications. ISBN 978-1-933988-05-4.

MEHTA, Bhakti a Ed ORT, 2003. Java Architecture for XML Binding (JAXB). In: *Oracle Technology Network* [online]. Mar, 2003 [cit. 2013-03-22]. Dostupné z: http://www.oracle.com/technetwork/articles/javase/index-140168.html

ORACLE CORPORATION, ©2013. JSR 127: JavaServer Faces. *Jcp.org* [online]. [cit. 2013-04-11]. Dostupné z: http://www.jcp.org/en/jsr/detail?id=127

PRIMEFACES, ©2011. Ultimate JSF Component Suite. *PrimeFaces.org* [online]. [cit. 2013-04-11]. Dostupné z: http://primefaces.org/

RED HAT, ©2004. Relational Persistence for Idiomatic Java. *Hibernate.org* [online]. [cit. 2013-02-25]. Dostupné z: http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/

RICHARDSON, Leonard a Sam RUBY, 2007. *Restful Web Services*. Sebastopol: O'Reilly. ISBN 978-0-596-52926-0.

SEDDIGHI, Ahmad, 2009. *Spring Persistence with Hibernate*. Birmingham: Packt Publishing. ISBN 978-1-849510-56-1.

SPRINGSOURCE, ©2004-2013. Spring Framework Reference Documentation. *SpringSource.org* [online]. [cit. 2013-02-18]. Dostupné z: http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/

SRIDHAR, M. S., 2009. Implement Java Connection Pooling with JDBC. In: *Developer.com* [online]. Nov 11, 2009 [cit. 2013-04-05]. Dostupné z: http://www.developer.com/java/data/article.php/3847901/Implement-Java-Connection-Pooling-with-JDBC.htm

TYAGI, Sameer, 2006. RESTful Web Services. In: *Oracle Technology Network* [online]. Aug, 2006 [cit. 2013-03-18]. Dostupné z: http://www.oracle.com/technetwork/articles/javase/index-137171.html

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
AOP	aspektově orientované programování
API	Application Programming Interface
apod.	a podobně
atd.	a tak dále
CRUD	Create, Read, Update, Delete
DAO	Database Access Object
DI	Dependency Injection
DOM	Document Object Model
EJB	Enterprise JavaBeans
HQL	Hibernate Query Language
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoC	Inversion of Control
IT	informační technologie
Java EE	Java Enterprise Edition
Java SE	Java Standard Edition
JAXB	Java Architecture for XML Binding
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Messaging Services
JPA	Java Persistence API
JRE	Java Runtime Environment
JSF	JavaServer Faces
JSP	JavaServer Pages
MS SQL	Microsoft SQL
MVC	Model-View-Controller

OOP	objektově orientované programování
ORM	objektivně-relační mapování
POJO	Plain Old Java Object
RADIT	Requirements, Analysis, Design, Implementation, Testing
REST	Representational State Transfer
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
tzn.	to znamená
tzv.	takzvaně
TDD	Test Driven Development
UDT	User Data Type
UML	Unified Modeling Language
UP	Unified Process
XML	Extensible Markup Language

Prohlášení o využití výsledků diplomové práce

Prohlašuji, že

- jsem byl seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. – autorský zákon, zejména § 35 – užití díla v rámci občanských a náboženských obřadů, v rámci školních představení a užití díla školního a § 60 – školní dílo;
- beru na vědomí, že Vysoká škola báňská – Technická univerzita Ostrava (dále jen VŠB-TUO) má právo nevýdělečně, ke své vnitřní potřebě, diplomovou práci užít (§ 35 odst. 3);
- souhlasím s tím, že diplomová práce bude v elektronické podobě archivována v Ústřední knihovně VŠB-TUO a jeden výtisk bude uložen u vedoucího diplomové práce. Souhlasím s tím, že bibliografické údaje o diplomové práci budou zveřejněny v informačním systému VŠB-TUO;
- bylo sjednáno, že s VŠB-TUO, v případě zájmu z její strany, uzavřu licenční smlouvu s oprávněním užít dílo v rozsahu § 12 odst. 4 autorského zákona;
- bylo sjednáno, že užít své dílo, diplomovou práci, nebo poskytnout licenci k jejímu využití mohu jen se souhlasem VŠB-TUO, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly VŠB-TUO na vytvoření díla vynaloženy (až do jejich skutečné výše).

V Ostravě dne 26. dubna 2013

.....

Bc. Miroslav Zapletal

Seznam příloh

Příloha č. 1 SQL kód pro vytvoření tabulek, sekvencí a triggerů

```
CREATE TABLE LV_APP_USER
  (
    ID           NUMBER(10) NOT NULL,
    USER_NAME    VARCHAR2(20) NOT NULL,
    ADMIN_RIGHTS VARCHAR2(1) NOT NULL,
    CREATE_DATE  DATE,
    CONSTRAINT LV_APP_USER_PK PRIMARY KEY (ID) USING INDEX STORAGE (INITIAL
1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_APP_USER_NAME_UNIQUE UNIQUE(USER_NAME)
  );
CREATE TABLE LV_LOG_USER
  (
    USER_NAME   VARCHAR2(20) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_LOG_USER_PK PRIMARY KEY (USER_NAME) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
  );
CREATE TABLE LV_COMPUTER
  (
    PC_NAME     VARCHAR2(20) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_COMPUTER_PK PRIMARY KEY (PC_NAME) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
  );
CREATE TABLE LV_ENVIRONMENT
  (
    ID           NUMBER(10) NOT NULL,
    OS_NAME      VARCHAR2(20) NOT NULL,
    JAVA_VERSION VARCHAR2(15),
    CREATE_DATE  DATE,
    CONSTRAINT LV_ENVIRONMENT_PK PRIMARY KEY (ID) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
```

```
  );
CREATE TABLE LV_FAB
  (
    FAB_NAME    VARCHAR2(10) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_FAB_PK PRIMARY KEY (FAB_NAME) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
  );
CREATE TABLE LV_LOG_TYPE
  (
    LOG_TYPE    VARCHAR2(20) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_LOG_TYPE_PK PRIMARY KEY (LOG_TYPE) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
  );
CREATE TABLE LV_APPLICATION
  (
    APP_NAME    VARCHAR2(20) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_APPLICATION_PK PRIMARY KEY (APP_NAME) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts
  );
CREATE TABLE LV_APP_FAB
  (
    APP_NAME    VARCHAR2(20) NOT NULL,
    FAB_NAME    VARCHAR2(10) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_APP_FAB_PK PRIMARY KEY (APP_NAME, FAB_NAME) USING INDEX
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_APP_FAB_APP_ID_FK FOREIGN KEY (APP_NAME) REFERENCES
LV_APPLICATION(APP_NAME),
    CONSTRAINT LV_APP_FAB_FAB_ID_FK FOREIGN KEY (FAB_NAME) REFERENCES
LV_FAB(FAB_NAME)
  );
CREATE TABLE LV_APP_TYPE
```

```
  (
    APP_NAME    VARCHAR2(20) NOT NULL,
    LOG_TYPE    VARCHAR2(20) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_APP_TYPE_PK PRIMARY KEY (APP_NAME, LOG_TYPE) USING INDEX
STORAGE (INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_APP_TYPE_APP_ID_FK FOREIGN KEY (APP_NAME) REFERENCES
LV_APPLICATION(APP_NAME),
    CONSTRAINT LV_APP_TYPE_LOG_TYPE_ID_FK FOREIGN KEY (LOG_TYPE) REFERENCES
LV_LOG_TYPE(LOG_TYPE)
  );
CREATE TABLE LV_RECORD
  (
    ID             NUMBER(19) NOT NULL,
    FAB_NAME       VARCHAR2(10),
    APP_NAME       VARCHAR2(20) NOT NULL,
    LOG_TYPE       VARCHAR2(20) NOT NULL,
    ENVIRONMENT_ID NUMBER(10),
    USER_NAME      VARCHAR2(20),
    PC_NAME        VARCHAR2(20) NOT NULL,
    MESSAGE        VARCHAR2(100) NOT NULL,
    LOG_TIME       DATE NOT NULL,
    CREATE_DATE    DATE,
    CONSTRAINT LV_RECORD_ID_PK PRIMARY KEY (ID) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_RECORD_FAB_ID_FK FOREIGN KEY (FAB_NAME) REFERENCES
LV_FAB(FAB_NAME),
    CONSTRAINT LV_RECORD_APP_ID_FK FOREIGN KEY (APP_NAME) REFERENCES
LV_APPLICATION(APP_NAME),
    CONSTRAINT LV_RECORD_LOG_TYPE_ID_FK FOREIGN KEY (LOG_TYPE) REFERENCES
LV_LOG_TYPE(LOG_TYPE),
    CONSTRAINT LV_RECORD_ENVIRONMENT_ID_FK FOREIGN KEY (ENVIRONMENT_ID)
REFERENCES LV_ENVIRONMENT(ID),
    CONSTRAINT LV_RECORD_USER_ID_FK FOREIGN KEY (USER_NAME) REFERENCES
LV_LOG_USER(USER_NAME),
    CONSTRAINT LV_RECORD_COMPUTER_ID_FK FOREIGN KEY (PC_NAME) REFERENCES
LV_COMPUTER(PC_NAME)
```

```
  );
CREATE TABLE LV_OPTIONS
  (
    ID          NUMBER(10) NOT NULL,
    APP_NAME    VARCHAR2(20),
    LOG_TYPE    VARCHAR2(20),
    DISABLED    VARCHAR2(1) NOT NULL,
    CREATE_DATE DATE,
    CONSTRAINT LV_OPTIONS_ID_PK PRIMARY KEY (ID) USING INDEX STORAGE
(INITIAL 1M NEXT 1M PCTINCREASE 0) TABLESPACE logviewer_ind_ts,
    CONSTRAINT LV_OPTIONS_APP_ID_FK FOREIGN KEY (APP_NAME) REFERENCES
LV_APPLICATION(APP_NAME),
    CONSTRAINT LV_OPTIONS_LOG_TYPE_ID_FK FOREIGN KEY (LOG_TYPE) REFERENCES
LV_LOG_TYPE(LOG_TYPE),
    CONSTRAINT LV_OPTIONS_IDS_UNIQUE UNIQUE (APP_NAME, LOG_TYPE)
  );


CREATE SEQUENCE LV_APP_USER_SEQ MINVALUE 1 MAXVALUE 9999999999 START WITH 1
INCREMENT BY 1;
CREATE SEQUENCE LV_ENVIRONMENT_SEQ MINVALUE 1 MAXVALUE 9999999999 START
WITH 1 INCREMENT BY 1;
CREATE SEQUENCE LV_RECORD_SEQ MINVALUE 1 MAXVALUE 9999999999999999999 START
WITH 1 INCREMENT BY 1;
CREATE SEQUENCE LV_OPTIONS_SEQ MINVALUE 1 MAXVALUE 9999999999 START WITH 1
INCREMENT BY 1;


CREATE OR REPLACE TRIGGER BI_LV_APP_USER BEFORE
  INSERT ON LV_APP_USER REFERENCING NEW AS NEW FOR EACH ROW DECLARE id
NUMBER;
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  := sysdate();
    END IF;
    IF :new.id IS NULL THEN
      SELECT LV_APP_USER_SEQ.NEXTVAL INTO id FROM dual;
      :new.id:=id;
```

```
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_LOG_USER BEFORE
  INSERT ON LV_LOG_USER FOR EACH ROW
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  := sysdate();
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_COMPUTER BEFORE
  INSERT ON LV_COMPUTER FOR EACH ROW
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
  END;
  /


CREATE OR REPLACE TRIGGER BI_LV_ENVIRONMENT BEFORE
  INSERT ON LV_ENVIRONMENT REFERENCING NEW AS NEW FOR EACH ROW DECLARE id
NUMBER;
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
    IF :new.id IS NULL THEN
      SELECT LV_ENVIRONMENT_SEQ.NEXTVAL INTO id FROM dual;
      :new.id:=id;
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_FAB BEFORE
```

```
  INSERT ON LV_FAB FOR EACH ROW
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_APPLICATION BEFORE
  INSERT ON LV_APPLICATION FOR EACH ROW
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_RECORD BEFORE
  INSERT ON LV_RECORD REFERENCING NEW AS NEW FOR EACH ROW DECLARE id
NUMBER;
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
    END IF;
    IF :new.id IS NULL THEN
      SELECT LV_RECORD_SEQ.NEXTVAL INTO id FROM dual;
      :new.id:=id;
    END IF;
  END;
  /
CREATE OR REPLACE TRIGGER BI_LV_OPTIONS BEFORE
  INSERT ON LV_OPTIONS REFERENCING NEW AS NEW FOR EACH ROW DECLARE id
NUMBER;
  BEGIN
    IF :new.create_date IS NULL THEN
      :new.create_date  :=sysdate();
```

```
    END IF;
    IF :new.id IS NULL THEN
      SELECT LV_OPTIONS_SEQ.NEXTVAL INTO id FROM dual;
      :new.id:=id;
    END IF;
  END;
  /
```