VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

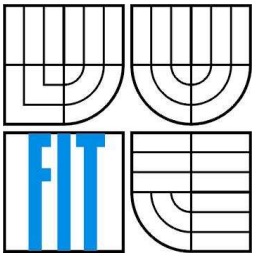
VNITŘNÍ INFORMAČNÍ SYSTÉM MALÉ FIRMY

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE MICHAL HUVAR
AUTHOR

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VNITŘNÍ INFORMAČNÍ SYSTÉM MALÉ FIRMY

INTERNAL INFORMATION SYSTEM FOR SMALL COMPANY

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — MICHAL HUVAR
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR JAŠA
SUPERVISOR

BRNO 2008

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat vnitřní informační systém malé firmy, který eviduje docházku zaměstnanců, umožňuje efektivní správu objednávek, služeb, projektů a poskytuje přehledně prezentované výsledky. Práce popisuje některé moderní populární prostředky pro vývoj informačních systémů. Věnuje se podrobné analýze, návrhu a implementaci celé aplikace.

Klíčová slova

Informační systém, databáze, JEE, aplikační rámec Spring, Spring MVC, Hibernate, Sitemesh, Acegi Security, JSP, HTML

Abstract

The purpose of this bachelor's thesis is to design and implement internal information system of small company, which files and attendance of employees, offers effective administration of orders, services, projects and provides well presented results. This work describes some of modern popular tools for development of information systems. It follows detailed analysis, design and implementation of whole application.

Keywords

Information system, database, JEE, Spring Framework, Spring MVC, Hibernate, Sitemesh, Acegi Security, JSP, HTML

Citace

Michal Huvar: Vnitřní informační systém malé firmy, bakalářská práce, Brno, FIT VUT v Brně, 2008

Vnitřní informační systém malé firmy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Jaši.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Huvar
9.5.2008

Poděkování

Chtěl bych poděkovat Ing. Petru Jašovi za poskytnutí podrobných požadavků na systém a za odborné vedení a konzultace, které mi poskytl v průběhu vypracovávání bakalářské práce.

Obsah

Úvod

S mohutným rozvojem informačních technologií v posledních desetiletích vzrostlo i rozšíření informačních systému mezi menší firmy. Prakticky každá firma disponuje v dnešní době moderním počítačovým vybavením, a tudíž roste poptávka po softwaru, který usnadní a zefektivní pracovní výsledky. Současně s tímto trendem se také zlepšují nástroje pro vývoj těchto systémů.

Cílem této bakalářské práce je návrh a implementace vnitřního informačního systému malé firmy na platformě Java Enterprise Edition. V první kapitole Vás seznámím s technologiemi, které jsem použil a které jsou zároveň jedny z nejpopulárnějších v této oblasti v posledních letech. Popíši zde již zmíněnou platformu Java Enterprise Edition s důrazem na aplikační rámec Spring Framework a nástroj pro objektově-relační mapovaní Hibernate. Dále zde zmíním databázový systém PostgreSQL, aplikační server Apache Tomcat, nástroj Apache Maven a v neposlední řadě také skvělé vývojové prostředí Eclipse IDE.

Druhá kapitola podrobně rozebírá návrh řešení tohoto informačního systému. Přestože tato aplikace nebude nikdy skutečně použita, chtěl jsem, aby dostala reálné rozměry. Proto jsem v kontaktu s firmou, která mi poskytla hrubý přehled o tom, jak to u nich chodí. Konkrétně se jedná o firmu zabývající se webhostingem, tvorbou webu, internetových obchodů a podobných systémů. V této kapitole je tedy uvedena analýza a sběr požadavků, detailní popis případů použití včetně příslušných diagramů a rozvržení objektů do databáze. Je zde také rozebrán důraz na rozdělení aplikace do tří vrstev a dodržování některých návrhových vzorů, které jsou trendem ve většině dnešních JEE aplikacích.

Třetí kapitola se zabývá implementací celého systému. Za pomoci ukázek zdrojových kódů zde podrobně rozebírám všechny vrstvy aplikace a použití výše zmíněných technologií. V popisu první vrstvy se zabývám nastavením a prací s rámcem Hibernate. V druhé vrstvě popisuji aplikační logiku. V popisu třetí vrstvy se věnuji všemu, co se týká uživatelského rozhraní. Tato kapitola také obsahuje ukázky konfigurace aplikace pomocí XML konfiguračních souborů, které jsem použil.

Poslední kapitolou je závěr, který tvoří shrnutí a přínos celé práce.

1 Technologie

V současné době existuje velké množství nástrojů pro tvorbu informačních systémů. Jejich popsání však není v možnostech ani cílech moji práce. Proto v této kapitole zmíním pouze ty, které jsem si vybral pro vytvoření této aplikace a uvedu jejich výhody a nevýhody. Veškeré nástroje, které jsem použil jsou volně dostupné.

1.1 Jazyk Java

Jazyk Java je vyspělý objektově orientovaný programovací jazyk, který byl vyvinut firmou Sun Microsystems a byl představen v roce 1995. Vychází z jazyka C, C++ a její aktuální verze je 1.6 s kódovým označením Tiger. Základní verzi tvoří Java Standard Edition (Java SE), která se používá pro běžné desktopové aplikace. V posledních letech se ovšem rozšířila i do jiných oblastí a vznikla tak Java Micro Edition (Java ME) používaná v mobilních zařízeních, JavaCard pro čipové karty a Java Enterprise Edition (Java EE) umožňující vytvářet škálovatelné výkonné aplikace běžící na serverech. Jazyk Java se stal jedním z nejpoužívanějších programovacích jazyků, a to z důvodů, které uvedu v následující kapitole.

1.1.1 Vlastnosti jazyka Java

Přenositelnost

Jedním s hlavních rysů tohoto jazyka je jeho přenositelnost. Programy v Javě jsou nejprve převáděny do tzv. mezikódu, který je nezávislý na cílové architektuře a teprve poté interpretovány pomocí Java Virtual Machine (JVM). JVM je jakýsi virtuální stroj, který zajišťuje správné zpracování mezikódu na dané architektuře. Nevýhodou tohoto řešení je pomalejší start programů, proto jsou v poslední době využívány technologie, které často prováděné nebo neefektivní části kódu překládají přímo do nativního kódu cílové architektury.

Jednoduchost a čitelnost

Oproti jazyku C++ je syntaxe Javy zjednodušena a upravena, což má za následek i lepší čitelnost.

Robustnost a spolehlivost

Díky tomu, že Java neumožňuje vytváření některých choulostivých konstrukcí, je možné v ní psát velmi spolehlivé aplikace. Přímo vyžaduje ošetření výjimek a poskytuje statickou typovou kontrolu. Pro správu paměti využívá tzv. Garbage Collector, který vyhledává a automaticky odstraňuje již nepoužívané objekty.

Distribuovanost

Jazyk Java umožňuje vytvářet rozsáhlé distribuované systémy – klientské aplikace a servery. Tuto vlastnost nejvíce využívá Java EE, které se budu věnovat v následující kapitole.

1.1.2 Java Enterprise Edition

Java EE (dále pouze JEE) je velmi moderní a rychle se rozvíjející platforma a spolu s technologií .NET od firmy Microsoft tvoří současnou špičku pro vývoj informačních a podnikových systémů. Je založena na Java SE a označuje celý balík technologií. V současné době existuje velké množství placených či volně dostupných nástrojů pro tvorbu JEE aplikací, které tyto technologie v různé míře využívají.

1.2 Aplikační rámec Spring

Pro tvorbu tohoto informačního systému jsem si vybral aplikační rámec Spring, který je současným hitem v oblasti vývoje JEE aplikací. Jedná se o open-source projekt, který byl založen v roce 2003 Rodem Johnsonem na základě kódu, který byl publikován v knize o vývoji bez EJB [1]. Cílem tohoto rámce je zefektivnění, zjednodušení a snížení ceny návrhu a vývoje JEE programů. Je to modulární neinvazivní odlehčený kontejner umožňující tvorbu jak webových, tak desktopových aplikací. V této práci jsem použil současnou nejaktuálnější verzi 2.5.2, která je stažitelná z webových stránek věnujících se tomuto rámci [2].

1.2.1 Základní rysy rámce Spring

Konzistentní podpora více vrstev

V dnešní době je dobrou zvyklostí rozdělovat enterprise aplikace do několika navzájem nezávislých vrstev. Většina aplikačních rámců se proto zaměřuje pouze na jednu z těchto vrstev. Narozdíl od nich Spring podporuje konzistentním způsobem všechny vrstvy aplikace – tzn. datovou, aplikační a prezentační. Zároveň však umožňuje integraci velkého množství kvalitních a časem prověřených nástrojů a tudíž je možné nahradit jakoukoli vrstvu jiným nástrojem, např. Pro prezentační vrstvu použít rámec Struts nebo pro datovou nástroj Hibernate.

Inversion of Control

Inversion of Control (u Springu označován jako Dependency Injection) je návrhový vzor, který odstraňuje těsné programové vazby mezi jednotlivými objekty a vrstvami. Provázání objektů řeší dvěma možnými způsoby. První způsob označovaný jako Setter Injection provazuje objekty pomocí setovacích metod. Druhý způsob využívá standardních konstruktorů jazyka Java.

POJO objekty

Rámec Spring podporuje využití jak klasických Enterprise Java Beans (EJB), tak i POJO objektů. Termín POJO (Plain Old Java Object) označuje klasické Java objekty obsahující pouze atributy, get/set metody a připadně nějaké další základní metody. Programátor tedy není nijak svázán technologicky specifickým rozhraním.

Snadná testovatelnost

Jelikož testování je nedílnou součástí vývoje každého softwaru, tak i Spring zde poskytuje velkou podporu. Díky možnosti oddělení jednotlivých vrstev a objektů umožňuje klasické jednotkové testování.

Aspektově orientované programování

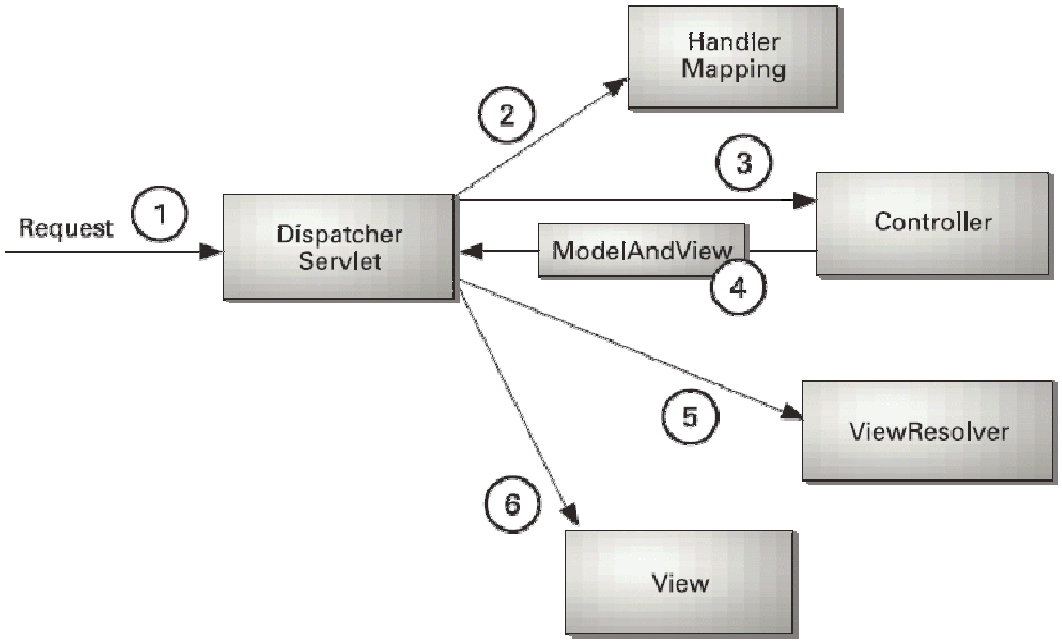
Pricip AOP (Aspect-oriented programming) spočívá ve vyčlenění některých částí stále se opakujícího kódu do tzv. aspektů. Ty mohou být poté volány před nebo po vykonání metody. V rámci Spring se jedná o jednu z jeho nejsilnějších vlastností, která se prolíná celým rámcem a zejména je využívána pro poskytnutí deklarativních služeb a pro tvorbu uživatelských aspektů.

1.2.2 Spring MVC

Pro prezentační vrstvu poskytuje Spring svůj vlastní rámec – Spring MVC, který využívá návrhového vzoru Model-View-Controller. Tento návrhový vzor rozděluje prezentační vrstvu na další tři části:

- **Model** – je tvořen doménovými objekty nesoucími data
- **View** – je jakýsi uživatelem definovaný pohled, který bude aplikován na model, zobrazuje tedy model
- **Controller** – zpracovává požadavky a na jejich základě vytvoří model, na který potom aplikuje odpovídající pohled

Přestože existuje několik velmi oblíbených prezentačních rámců (např. Struts), vybral jsem si Spring MVC, jelikož poskytuje vysokou flexibilitu, mapování hodnot z formulářů na POJO objekty, JSP značkovací knihovnu pro snadné psaní formulářů a je přirozenou součástí rámce Spring, a proto může využívat všechny jeho vymoženosti. Pracuje na principu přijímání a zasílaní požadavků. Na obrázku 1.1 je zobrazen životní cyklus požadavku.



Obázek 1.1 životní cyklus požadavku

1) Základní jednotkou pro komunikaci se servletovým kontejnerem, na kterém aplikace běží, je *DispatcherSerlvet*. Při příchodu požadavku musí rozhodnout, který kontroler bude požadavek obsluhovat.
2) To, který kontroler se vybere, určují objekty implementující rozhraní *HandlerMapping*. Tyto objekty obsahují mapovaní jednotlivých stránek na kontrolery.
3) Kontrolery jsou třídy, které provádějí skutečné zpracování požadavku. Ve správně implementované třívrstvé aplikaci by však neměly obsahovat žádnou větší logiku a měly by volat metody objektů z aplikační vrstvy. Všechny kontrolery implementují rozhraní *Controller*.
4) Po dokončení své činnosti vrací kontroler *DispatcherServletu* objekt typu *ModelAndView*, podle nehož se bude generovat odpověď na požadavek. Tento objekt obsahuje pohled (*View*), který se má aplikovat a mapu objektů, které budou v tomto pohledu uživatelovi dostupné.
5) *View* je rozhraní určující uživatelem definovaný pohled, který se použije na odpověď požadavku. Ve většině případů tvoří pohled JSP stránka, ale může to také být například pdf dokument nebo jiný formát.
6) Objekty implementující rohraní *ViewResolver* umožňují *DispatcherServletu* rozhodnout o tom, který pohled se bude na odpověď aplikovat.

Více o tomto rámci a celém Springu se můžete dočíst v referenční dokumentaci, která je součástí tohoto projektu nebo v knize o Springu [3].

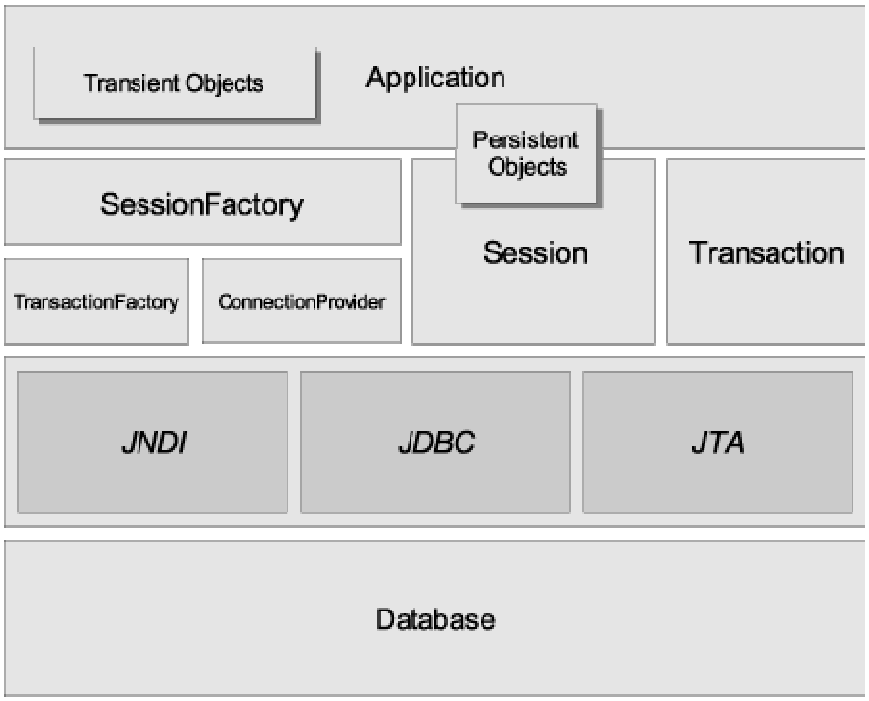
1.3 Hibernate

Většina dneších JEE aplikací používá objektově-orientovaný přístup a relační databáze. Tyto dvě techniky jsou však značně rozdílné, a tak programátoři stráví velkou část času psaním kódu pro manipulaci s databází. Proto je v poslední době trendem používat objektově-relační mapovaní (ORM), které umožňuje mapovat objekty do tabulek relační databáze. Jedním z nepopulárnějších nástrojů, který ORM poskytuje, je právě Hibernate. Je přímo podporován rámcem Spring a poskytuje jednotné API pro manupulaci s daty nad databází a tím velmi usnadňuje implementaci datové vrstvy.

1.3.1 Architektura Hibernate

Hibernate architektura umožňuje několik různých způsobů přístupu k databázi. Uvedu zde pouze ty dva nejkrajnější a to tzv. odlehčený přístup, kdy je Hibernate API využíváno minimálně a aplikace si spravuje vlastní JDBC spojení a transakce.

Opakem je tzv. Full-cream architektura, což je plné využití možností Hibernate a maximální odstínění aplikace od JDBC. Tento přístup jsem zvolil ve své práci, a proto ho dále detalněji popíši. Na obrázku 1.2 je znázorněna Full-cream architektura Hibernate.



Obrázek 1.2 Full-cream architektura Hibernate

Nejdůležitější součástí Hibernate je *Session*. *Session* obaluje JDBC spojení a vytváří konverzi mezi relační a objektovou reprezentací dat. Vytváří také transakce a slouží jako cache první úrovně. Objekty typu *Session* jsou vytvářeny velmi často.

Oproti tomu objekt typu *SessionFactory* je zpravidla vytvořen jednou za běh programu. Slouží jako konfigurace objektově-relačního mapování a udržuje cache druhé úrovně.

Transaction jsou transakce, které jsou vytvářeny v *TransactionFactory*. Odstiňují aplikaci od JDBC a JTA transakcí a zvyšují tak její přenositelnost.

Persistent Objects jsou objekty, které jsou v oblasti působnosti aktuální *Session* a jakákoli jejich změna je provedena také nad databází.

Opakem jsou *Transient Objects*, což jsou objekty, které nikdy nebyly v působnosti s žádnou *Session* a tudíž nemají odpovídající zápis v databázi. Většinou to jsou nově vytvořené objekty.

1.3.2 Mapování objektů a práce s nimi

Hibernate potřebuje vědět jakým způsobem má ukládat javovské objekty do databáze. K tomuto účelu slouží tzv. mapování. Objekty lze do databázových tabulek mapovat dvěma možnými způsoby. Prvním z nich je mapování pomocí XML souborů, který je starší a vyžaduje mít většinou pro každou třídu vlastní XML soubor. Druhý způsob je dle mého názoru pohodlnější a přehlednější. Jedná se o mapování pomocí Java 5 anotací, které se píše přímo v kódu dané třídy. Hibernate poté pomocí mapování vygeneruje příslušné sql schéma pro vytvoření tabulek a jejich provázání.

Samotná práce s objekty probíhá v aplikačním rámci Spring velice jednoduše. Nejprve je nutné získat přístup k továrně sezení. To zajistím tím, že každá třída datové vrstvy bude rozšiřovat třídu *HibernateDaoSupport*. Poté mohu pro veškerou manipulaci s daty použít *HibernateTemplate*, které za mě zajistí korektní otevření a uzavření Hibernate *Session*, zachytávání výjimek a správu transakcí. Pro získávání objektů z databáze poskytuje Hibernate tzv. Hibernate Query Language (HQL), což je plně objektově-orientovaný jazyk podobný svou syntaxí jazyku SQL. Výběr více propojených objektů z databáze je možný dvěma způsoby. První z nich je použití hladové strategie (Eager fetching strategy), kde jsou vybrány všechny objekty při prvním přístupu do databáze. Pro větší efektivnost aplikace se však většinou používá líná strategie (Lazy fetching Strategy), kdy je vybrán pouze jeden objekt a ostatní asociované objekty jsou vybrány az při prvním přístupu k nim. Více informací o Hibernate a jeho poslední verzi si můžete stáhnout z oficiálních webových stránek tohoto nástroje [4].

1.4 PostgreSQL

Nedílnou součástí většiny informačních systémů je databáze. Já jsem si pro svoji aplikaci vybral PostgreSQL, což je plnohodnotný databázový systém s volně otevřeným zdrojovým kódem. Důvodem mého výběru je široká použitelnost, vynikající stabilita, snadná konfigurace a přívětivé uživatelské rozhraní pgAdminIII. V současné době je nejaktuálnější verze 8.3.1 dostupná na webových stránkách [5], nicméně já jsem použil verzi 8.2.

1.5 Apache Tomcat

Apache Tomcat je odlehčený JSP/Servlet kontejner vyvíjený společností Apache Software Foundation. JSP/Servlet kontejner je označení pro jakýsi webový server, který obsahuje servlety pracující s JSP stránkami. Servlet je program napsaný v jazyce Java, který zpracovává požadavky HTTP protokolu a umí na ně také odpovídat. JSP je technologie umožňující vytvářet HTML stránky s dynamicky generovaným obsahem. V moji práci tedy Tomcat tvoří úlohu serveru, na kterém aplikace běží. Aktuální verze 6.x implementující Servlet 2.5 a JSP 2.1 je stažitelná z webových stránek zabývajících se tímto projektem [6].

1.6 Apache Maven

Maven je další nástroj od společnosti Apache Software Foundation, který představuje pokročilejší náhradu nástroje Ant. Slouží ke kompilaci, distribuci a dokumentaci projektů v Javě, umožňuje ale také použít antovské úkoly. Ve své aplikaci jsem použil verzi 2.0.8 pro kompilaci, generování databáze, instalaci do Tomcatu, stahování potřebných závislostí a generování javadoc dokumentace. Apache Maven je možno stáhnout z webových stránek tohoto projektu [7].

1.7 Eclipse IDE

V závěru výčtu použitých technologií musím také zmínit vývojové prostředí Eclipse, ve kterém jsem celou aplikaci tvořil. Eclipse je projekt od společnosti Eclipse Foundation s otevřeným zdrojovým kódem stažitelný na webových stránkách [8]. Je to vynikající vývojové prostředí určené pro programování v jazyce Java oblíbené především mezi JEE vyvojáři. Mezi jeho hlavní přednosti patří propracovaná kontrola syntaxe v průběhu psaní zdrojového kódů a podpora téměř neomezeného množství pluginů. Díky těmto pluginům nemusí programátor externě používat ostatní nástroje, ale může si je integrovat přímo do vývojového prostředí a všechny operace provádět v něm. Mezi mé oblíbené pluginy patří Subversive, který integruje populární systém pro správu a verzování

zdrojových kódů Subversion a také Spring IDE, který zjednodušuje správu springovských beanů a umožňuje vytvářet grafy s jejich zavislostmi, což může někdy pomoci při orientaci ve velkých projektech.

2 Návrh řešení

Stejně jako u každého jiného informačního systému, tak i u toho mého je velice důležité vytvořit kvalitní návrh, který může velmi usnadnit a urychlit další fáze jeho vývoje.

Zadáním mé práce bylo navrhnout a implementovat systém, který bude evidovat docházku zaměstnanců a bude umožňovat efektivní správu objednávek, služeb a projektů s možností sledovat plán projektu oproti jeho skutečnému stavu. Měl by umět automatickou analýzu projektů, upozorňovaní na případné nedodržení plánu a přehledně prezentovat výsledky získané z analýzy s možností exportu do alespoň jednoho z formátů XML, pdf nebo xls. Na závěr bych měl systém otestovat na nějakém vzorku dat a zhodnotit výsledky.

2.1 Analýza a sběr požadavků

Jak jsem již zmínil v úvodu, tento systém nebude nikdy reálně použit. Nicméně nechal jsem se inspirovat skutečnou firmou a na základě diskuse s jedním z jejich zaměstnanců jsem definoval základní požadavky na tento systém.

Aplikace bude rozlišovat čtyři základní typy uživatelů:

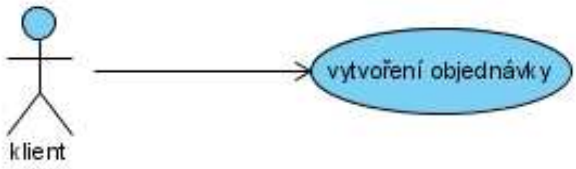
- Klienti
- Zaměstnanci
- Projektoví manažeři
- Šéf

V následující kapitole jsou všechny výše zmíněné přístupy k aplikaci detailněji popsány a vymodelovány pomocí diagramů případů užití. Pro jejich tvorbu jsem použil Visual Paradigm for UML Community Edition ve verzi 6.2, který je po registraci volně stažitelný z webových stránek tohoto nástroje [9].

2.1.1 Diagramy případů užití

2.1.1.1 Klienti

Klientská část aplikace je přistupná pro všechny uživatele prostřednictvým WWW. Obsahuje pouze jediný formulář pro objednávku nového projektu. Při objednávce je dostupný seznam již registrovaných zákazníků. Klient se tedy může vybrat ze seznamu nebo si vyplní informace o sobě do objednávkového formuláře a po odeslání objednávky se automaticky přidá do databáze klientů.



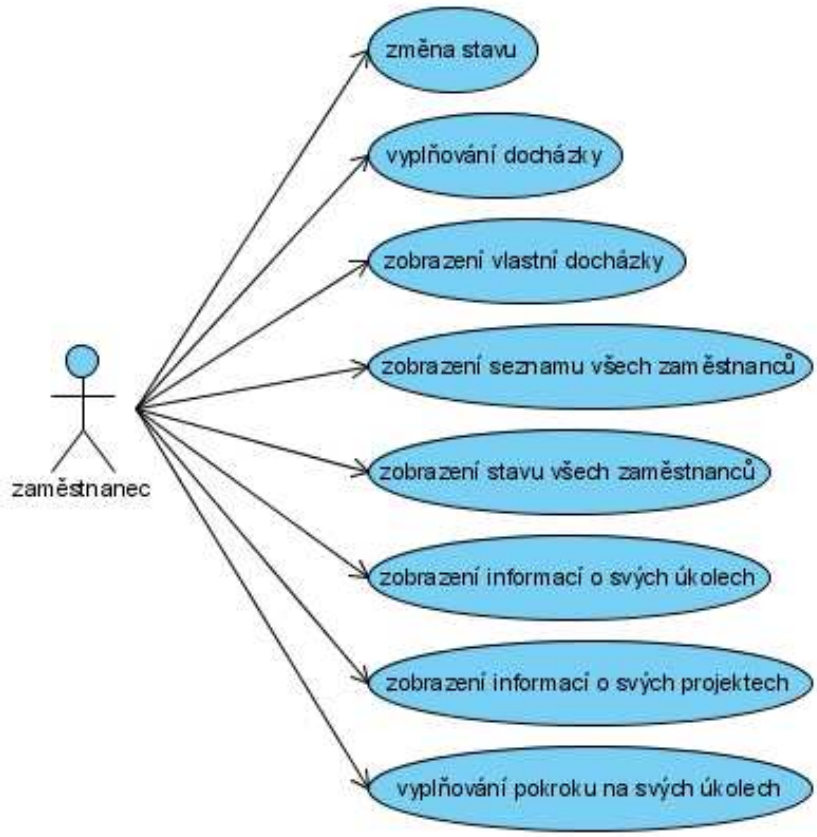
Obrázek 2.1 případ užití – klient

2.1.1.2 Zaměstnanci

Zaměstnanecká část je stěžejní částí aplikace. Je přístupná pouze pro přihlášení. Každému zaměstnanci je umožněno měnit svůj stav, zobrazovat stavy ostatních zaměstnanců a zobrazovat svoji docházku za určité období. Dostupné stavy jsou:

- V práci
- Mimo práci
- Na obědě
- Pracovně mimo
- Soukromě mimo
- Dovolená
- Nemoc

Dále si mohou zaměstnanci prohlížet detailní informace o všech projektech a úkolech, na kterých se podílí. Během práce na úkolech každý zaměstnanec vyplňuje informace o stavu průběhu svého úkolu, tedy v kolika procentech zhruba je a co již provedl. Každý úkol má nastaveno datum a čas dokončení. Pokud se tento termín překročí, bude automaticky odeslán upozorňovací email odpovědnému řešiteli projektu.



Obrázek 2.2 případ užití – zaměstnanci

2.1.1.3 Projektoví manažeři

Projektoví manažeři jsou speciální skupinou uživatelů, která bude mít stejná práva jako zaměstnanci. Navíc je jim však umožněno vytvářet aktivní služby, projekty, spravovat klienty a urgovat zaměstnance při práci na jejich úkolech. Manažeři mohou také vyplňovat objednávkový formulář (např. Pokud bude klient provádět objednávku přes telefon). V tomto systému je celkem pět typů služeb:

- Webová prezentace
- Publikační systém
- Internetový obchod
- Registrace domény
- Webhosting na dané doméně

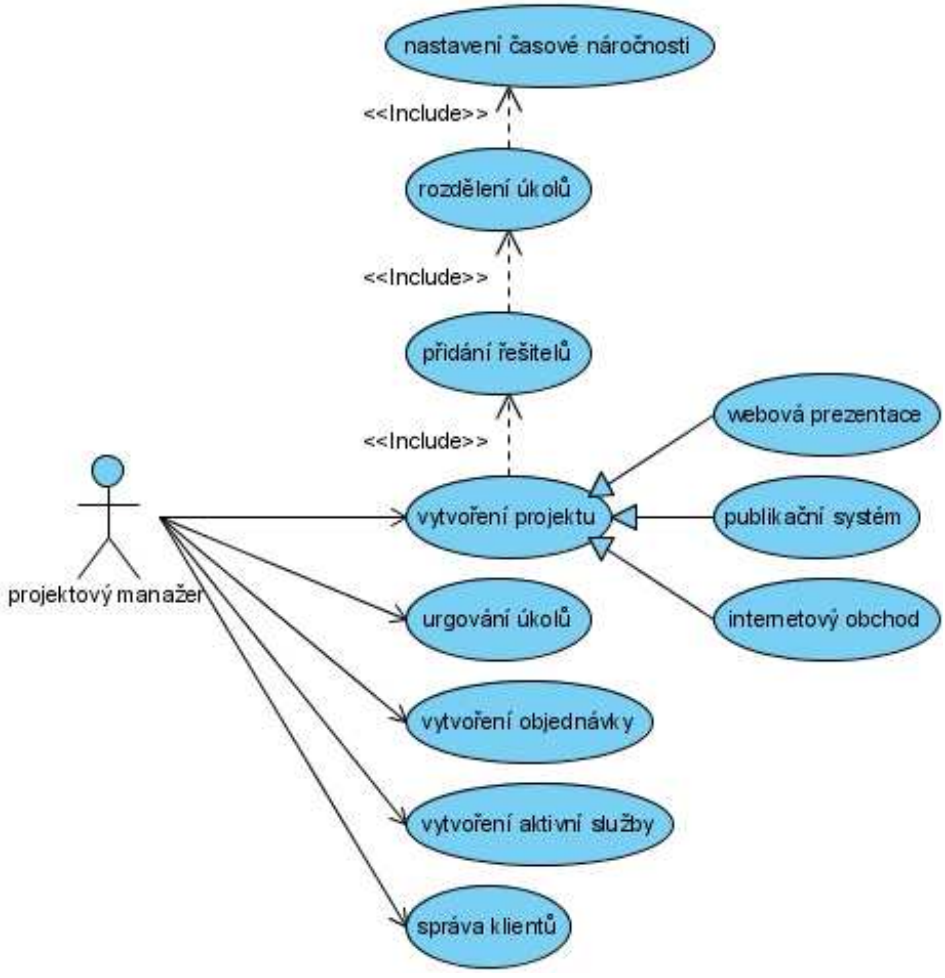
První tři výjmenované služby jsou provozované pouze na serverech firmy, takže zákazník si k nim musí také objednat jeden z webhostingových programů, který bude běžet buď na jeho vlastní

doméně nebo si objedná registraci nové domény. Z každé objednávky manažer vytvoří aktivní službu, která bude uchovávat informace o stavu služby, paušálu za webhosting, doménu a datum posledního zaplacení. Stavy aktivních služeb jsou:

- Před spuštěním
- Aktivní
- Pozastavená
- Zrušená

Pokud si klient objednal službu jako je internetový obchod nebo webová prezentace, tak manažer po vytvoření aktivní služby může založit projekt pro tuto službu. Při vytváření nového projektu je nutné mu přidělit řešitele, rozdělit mezi ně jednotlivé úkoly a také u nich určit přibližnou časovou náročnost. Každý projekt musí mít odpovědného řešitele, který má na starost kontrolu dokončeného projektu a řádné předání zákazníkovi nebo projektovému manažerovi.

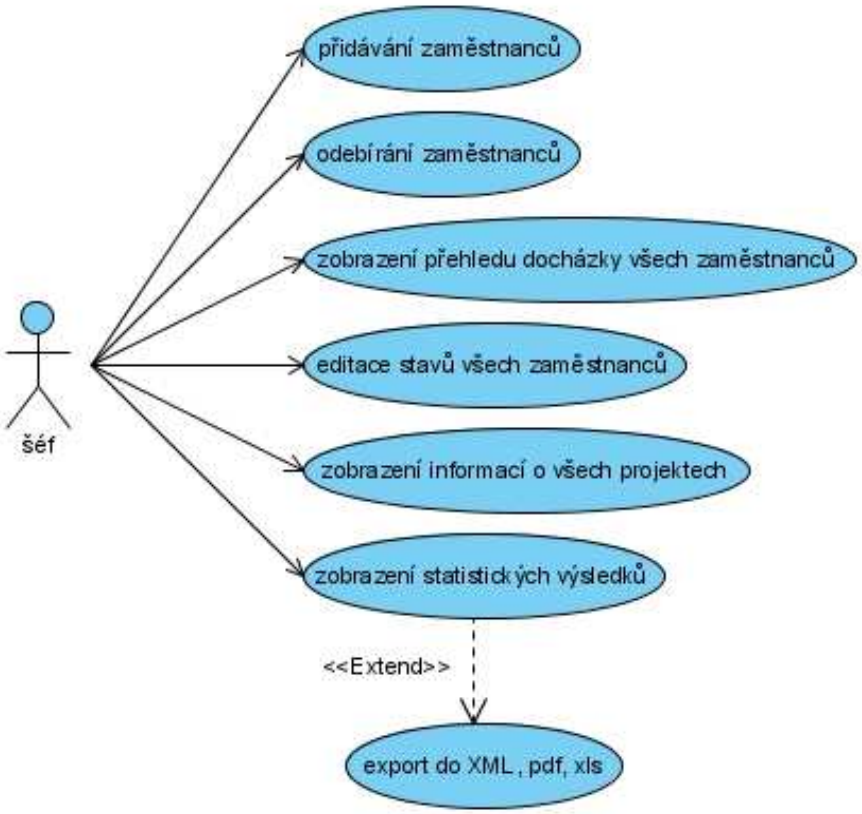
Urgování úkolů může projektový manažer prodádět zasláním emailu příslušnému pracovníkovi.



Obrázek 2.3 případ užití – projektový manažeři

2.1.1.4 Šéf

Uživatel s touto rolí má navíc oproti manažerům možnost editovat stav všech zaměstnanců a nechat si zobrazit jejich docházku za určité období. Má také kompletní přehled o všech projektech a jejich statistikách. Statistiky sledují zejména dodržování dokončení úkolů a projektů v zadaných termínech. Šéf může tyto statistiky exportovat do formátu (XML, cvs, pdf nebo xls).



Obrázek 2.4 případ užití – šéf

2.2 Dodržování návrhových vzorů

Vývoj kvalitních moderních informačních systémů je spojen s dodržováním určitých návhových vzorů. Návrhový vzor představuje obecné řešení stále se opakujících netriviálních problémů. Psaní aplikací v rámci Spring přímo vyžaduje dodržování některých návrhových vzorů. Ty, které jsem použil nyní podrobně popíši, čerpal jsem z knihy o Springu [3].

2.2.1 Kódování do rozhraní

Kódování do rozhraní umožňuje oddělení deklarace metod, které budou objekty používat od jejich implementace. Tím získá programátor vysokou flexibilitu. Rozhraní se používají především pro komunikaci mezi jednotlivými vrstvami v aplikaci, aby se zamezilo svazování kódu a bylo možné například zaměnit databázový systém nebo objektově-relační přístup k databázi za klasické JDBC spojení.

2.2.2 Rozdělení aplikace do tří vrstev

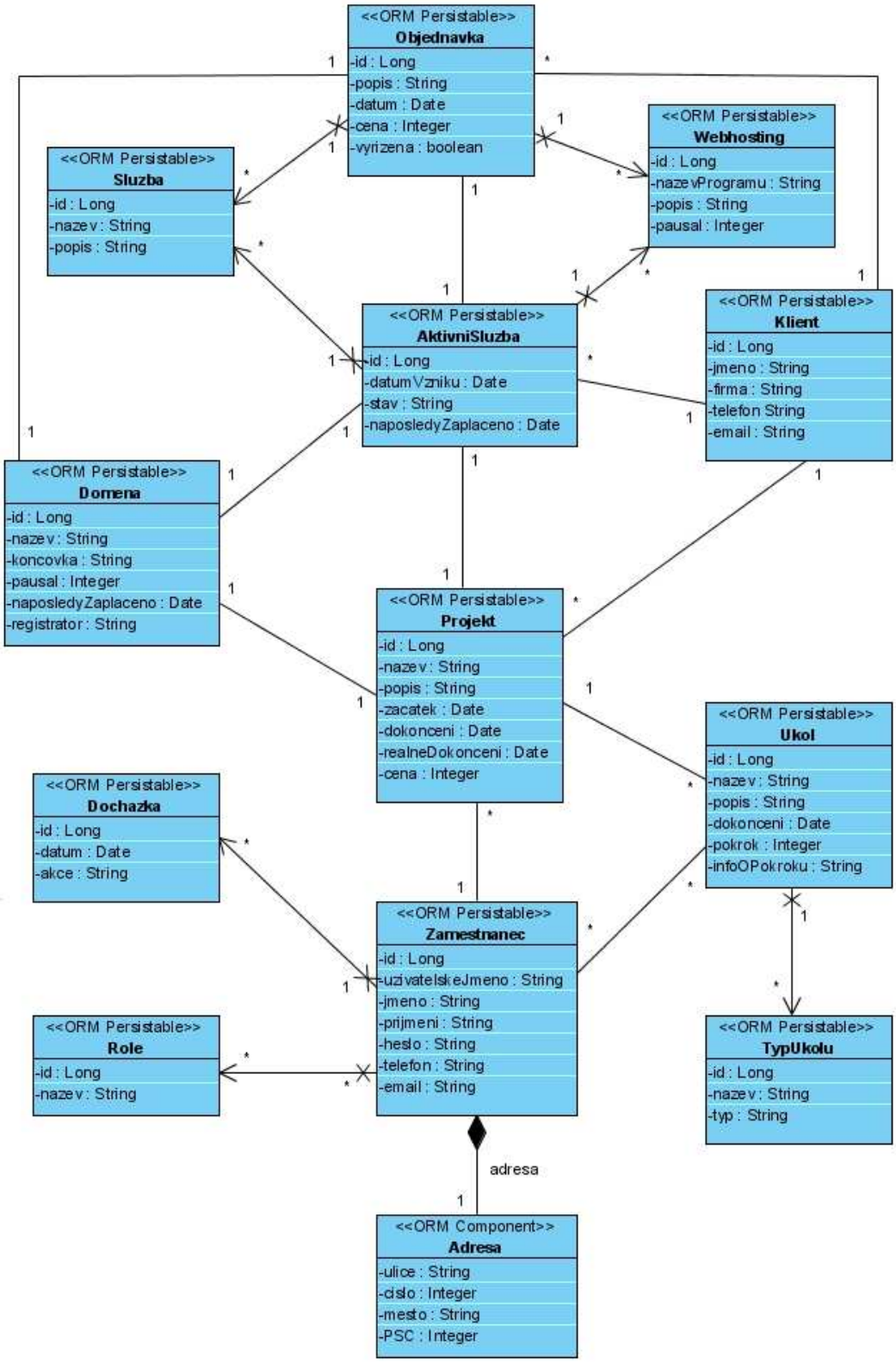
Typické JEE aplikace se skládají z několika na sobě nezávislých vrstev. Spolu s kódováním do rozhraní by pak měla vzniknout snadno udržovatelná a rozšiřitelná aplikace. Já jsem použil rozdělení do tří vrstev:

1) **Datová vrstva DAO (Data Access Object)** – třídy, které komunikují pouze z databází
2) **Aplikační vrstva** – třídy, které poskytují aplikační logiku, tvoří most mezi uživatelským rozhraním a datovou vrstvou
3) **Prezentační vrstva** – třídy a pohledy, které vytvářejí webové rozhraní pro komunikaci s uživatelem

Prezentační vrstva je ve Springu MVC dále rozdělena na Model, Pohled a Kontroler.

2.3 Vrstva Model – diagram perzistentních tříd

Vrstva Model je tvořena objekty nesoucími data, a tudíž je nutné je ukládat do databáze. Proto se také Model může řadit i do datové vrstvy. Po analýze a sběru požadavků jsem tedy navrhl diagram perzistentních tříd. Jednotlivé třídy na tomto diagramu a jejich provázání dále podrobněji popíši.



Obrázek 2.5 diagram perzistentních tříd

2.3.1 Třída Objednavka

Tato třída uchovává informace o objednávkách v systému. Obsahuje atributy *id*, *popis*, *datum*, *cena* a *vyrizena*. *Id* je unikátní číslo, které tvoří primární klíč pro jednoznačnou identifikaci objektu v databázi. Tento klíč obsahuje každá samostatná třída kromě *Adresy*, proto se o něm v popisu dalších tříd již nebudu zmiňovat. Do *popisu* se ukládají podrobnosti o objednávce, *cena* je orientační nebo maximální cena, kterou je klient ochoten nabídnout. *Datum* je datum vytvoření objednávky a *vyrizena* slouží k rozlišení nově podaných od již vyřízených objednávek.

2.3.2 Třída Sluzba

Představuje jednotlivé služby, které firma nabízí. V případě mé aplikace bude tedy obsahovat pouze tří záznamy (webová prezentace, internetový obchod a publikační systém). Není však problém doplnit jakoukoli novou službu. Její atributy jsou *nazev* a *popis*.

2.3.3 Třída Webhosting

Třída *Webhosting* má podobný charakter jako třída *Sluzba*. Slouží k uchovávání informací o webhostingových programech, které firma nabízí. Obsahuje atributy *nazevProgramu*, *popis* a *pausal*. *Pausal* představuje roční paušál za daný program.

2.3.4 Třída Domena

Ukládá informace jak o klientových, tak o nově registrovaných doménách. Její atributy jsou *nazev*, *koncovka*, *pausal*, *naposledyZaplaceno* a *registrator*. *Název* a *koncovka* tvoří celou doménu (např. seznam.cz). *Pausal* je roční cena za doménu a jeho poslední zaplacení se ukládá do *naposledyZaplaceno*. *Registrator* slouží k identifikaci společnosti, u které je doména registrována, pokud není registrována u „naší firmy“.

2.3.5 Třída Klient

Tato třída uchovává informace o všech klientech v systému. Její atributy jsou *jmeno*, *firma*, *telefon*, *email*. *Jmeno* symbolizuje kontaktní osobu.

2.3.6 Třída AktivniSluzba

Aktivní služba představuje jakousi základní jednotku v systému. Z každé objednávky se vytváří aktivní služba a většina projektů se zakladá právě také z aktivní služby. Proto nese odkazy na všechny důležité objekty jakou jsou *Objednavka*, *Sluzba*, *Webhosting*, *Klient* a *Domena*. Samotný objekt *AktivniSluzba* pak obsahuje pouze atributy *datumVzniku*, *stav* (např. před spuštěním) a

naposledyZaplaceno, kam se ukládá datum posledního zaplacení paušálu za daný webhostingový program.

2.3.7 Třída Projekt

Třída *Projekt* uchovává informace o všech projektech v systému. Pokud se projekt vytváří z aktivní služby, tak obsahuje pouze odkaz na onu aktivní službu, zaměstnance a úkoly. Všechny důležité informace o klientovi, doméně atd. jsou pak dostupné přes aktivní službu. Pokud by se jednalo o nějaký specialní projekt, který se nebude vytvářent z aktivní služby, tak může projekt obsahovat odkaz přímo na klienta a doménu. Vztah projekt – zaměstnanec zde představuje pouze vazbu zakladatele nebo odpovědného řešitele k danému projektu. Zaměstnanci, kteří řeší jednotlivé úkoly projektu se vážou přímo na úkoly. *Projekt* obsahuje atributy *nazev*, *popis*, *zacatek*, *dokonceni*, *realneDokonceni*, *cena*. *Zacatek* značí datum založení projektu, *dokonceni* je datum plánovaného dokončení, které zadává projektový manažer a *realneDokonceni* je skutečné datum dokončení projektu. *Cena* zde představuje definitivní cenu projektu, kterou zadává projektový manažer na základě náročnosti projektu, či klientem stanovené požadované ceny v objednávce.

2.3.8 Třída Ukol

Tato třída nese informace o jednotlivých úkolech v projektu. Obsahuje atributy *nazev*, *popis*, *dokonceni*, *pokrok* a *infoOPokroku*. *Dokoncení* je datum požadovaného dokončení zadané projektovým manažerem. *Pokrok* značí procentuální stav úkolu a *infoOPokroku* je popis práce, kterou zaměstnanec zatím na úkolu provedl.

2.3.9 Třída TypUkolu

TypUkolu slouží pouze k uložení předdefinovaných typů úkolů. Nese dva atributy *nazev* a *typ*.

2.3.10 Třída Zamestnanec

Zde se uchovávají všechny informace o zaměstnancích v systému. Atributy *uzivatelskeJmeno*, *jmeno*, *prijmeni*, *heslo*, *telefon* a *email* není třeba nijak popisovat.

2.3.11 Třída Adresa

Třída *Adresa* je komponenta třídy *Zamestnanec*. Proto také jako jediná nemá vlastní identifikační číslo. Její atributy *ulice*, *cislo*, *mesto*, *PSC* také netřeba dále vyvětlovat.

2.3.12 Třída Role

Tato třída uchovává role, kterými může zaměstnanec disponovat. Jelikož má moje aplikace celkem čtyři druhy přístupů, tak bude obsahovat také čtyři typy rolí. Názvy rolí jsou ROLE_ANONYMOUS pro přístup klientů nebo zaměstnanců před přihlášením, ROLE_USER, ROLE_MANAGER a ROLE_ADMIN, kterou bude disponovat šéf.

2.3.13 Třída Dochazka

Poslední třídou, jejíž data jsou ukládána do databáze je *Dochazka*. Atributy této třídy jsou *datum* a *akce*. *Akce* může být příchod či odchod z práce.

3 Implementace

V této kapitole budu popisovat poslední fázi vývoje mé aplikace. Jelikož není v možnostech této práce detailně popsat implementaci každé funkcionality, pokusím se zde rozebrat alespoň ty z mého pohledu nejpodstatnější věci.

3.1 Datová vrstva

V popisu této vrstvy se budu zabývat zejména nastavením a prací s rámcem Hibernate.

3.1.1 Nastavení Hibernate v rámci Spring

Nastavení aplikačního rámce Spring je možné provést pomocí XML konfiguračních souborů a od verze 2.5 také pomocí anotací. V tomto případě jsem zvolil použití klasických XML souborů, protože poskytuje nastavení všech beanů podobné funkcionality v jednom souboru a tím se mi jeví jako přehlednější.

Nejprve je třeba nakonfigurovat aplikační kontext, který se postará o spuštění továrny tříd (bean factory) a o správné provázání objektů (dependency injection). V mé aplikaci představuje tuto konfiguraci soubor *applicationContext.xml*. Pro správný chod Hibernate jsem v tomto souboru definoval instanci tříd *BasicDataSource* a *AnnotationSessionFactoryBean*.

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName"
    value="${hibernate.connection.driver_class}"/>
    <property name="url" value="${hibernate.connection.url}"/>
    <property name="username" value="${hibernate.connection.username}"/>
    <property name="password" value="${hibernate.connection.password}"/>
</bean>

<bean id="sessionFactory"
     class="org.springframework.orm.hibernate3.annotation.AnnotationSession
     FactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="configLocation" value="classpath:hibernate.cfg.xml"/>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=${hibernate.dialect}
        </value>
    </property>
```

```
</bean>
```

Kvůli stručnosti jsem uvedl pouze jejich základní konfiguraci. Vlastnosti těchto beanů, které mají proměnné hodnoty, definuji v konfiguračním souboru *config.properties*. V moji aplikaci jsem tyto hodnoty nastavil pro databázi PostgreSQL. Objekt třídy *AnnotationSessionFactoryBean* je zodpovědný za vytváření továrny sezení. Jeho vlastnost *configLocation* určuje cestu k souboru *hibernate.cfg.xml*, který obsahuje seznam tříd namapovaných na tabulky relační databáze. Pro správné fungování transakcí je ještě nutné deklarovat instanci třídy *HibernateTransactionManager*. Tento bean by měl obsahovat odkaz na bean *sessionFactory*, ale jelikož mám v hlavičce všech konfiguračních souborů uvedenou vlastnost `default-autowire="byName"`, tak se mi toto provázání provede automaticky. O této vlastnosti se ještě zmíním později.

```
<bean id="transactionManager"
  class="org.springframework.orm.hibernate3.HibernateTransactionManager"/>
```

3.1.2 Mapování objektů do tabulek relační databáze

Jak jsem již zmínil v kapitole 1.3.2 pro mapování objektů jsem použil Java 5 anotace. Nyní zde tedy ukážu, jak jsem namapoval objekty do tabulek relační databáze podle navrženého diagramu perzistentních tříd (obrazek 2.5).

Každá třída se váže na jednu tabulku v databázi. Následuje zjednodušená ukázka z třídy *Zamestnanec*:

```
@Entity
@Table(name="zamestnanec")
public class Zamestnanec {

    private Long id;
    private String jmeno;

    @Id
    @SequenceGenerator(name="zamestnanci_sequence",
    sequenceName="zamestnanci_sequence", allocationSize=4)
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
    generator="zamestnanci_sequence")
    public Long getId() {
        return id;
    }

    @Column(length=255, nullable=false)
    public String getJmeno() {
        return jmeno;
    }
```

Anotace `@Entity` označuje, že tato třída bude perzistentní, `@Table` představuje tabulku v databázi, kam se bude ukládat. `@Id` označuje primární klíč objektu. V tomto případě bude Hibernate k jeho ukládání používat vlastní sekvenci s názvem *zamestnanci_sequence*. Ostatní atributy se mapují anotací `@Column`, která může obsahovat několik dalších vlastností. Já jsem zde použil vlastnost `length`, která určuje maximální možnou velikost a `nullable=false`, která nedovoluje, aby tento sloupec byl prázdný. Hibernate si zde sám určuje, jaký datový typ bude mít sloupec v databázi podle javovského typu.

Pro ukázku mapování vztahů mezi jednotlivými třídami zde uvedu obousměrný vztah *Zamestnanec-Ukol* s kardinalitou m:n, kde vlastníkem je *Ukol*. Nejprve namapování z třídy *Ukol*:

```
@ManyToMany(fetch=FetchType.EAGER)
@JoinTable(
     name="ukol_ma_zamestnance",
     joinColumns=@JoinColumn(name="ukol_id"),
     inverseJoinColumns=@JoinColumn(name="zamestnanec_id"))
public Set<Zamestnanec> getZamestnanci() {
     return zamestnanci;
}
```

Vztah m:n se řeší anotací `@ManyToMany`. `Fetch=FetchType.EAGER` zaručuje načtení kolekce zaměstnanců ihned po načtení úkolu z databáze, jejím opakem je tzn. líné načítání (*FetchType.LAZY*). `@JoinTable` označuje pomocnou tabulku obsahující dva sloupce, které jsou definované anotací `@JoinColumn` a jejich názvy jsou tvořeny konkatenací názvu odkazované entity a jejího primárního klíče. Protože je asociace obousměrná, tak jsem její druhý konec musel uvést také v třídě *Zamestnanec*:

```
@ManyToMany(fetch=FetchType.EAGER, mappedBy="zamestnanci")
public Set<Ukol> getUkoly() {
     return ukoly;
}
```

Zde se uvádí pouze vlastnost `mappedBy`, která odkazuje na atribut vlastnící vztah. V aplikaci jsem použil také všechny ostatní druhy vztahů jako jsou *@OneToOne*, *@OneToMany*, *@ManyToOne* a *@Embedded*. Jejich implementaci je možné nalézt v konkrétních třídách. Každou třídu, která bude pracovat s databází, je třeba uvést do konfiguračního souboru *hibernate.cfg.xml*.

3.1.3 Práce s perzistentními objekty

Práce s perzistentními objekty se provádí výhradně ve vrstvě DAO. Pro každou perzistentní třídu existuje odpovídající třída DAO. K manipulaci s databází používám metodu *getHibernateTemplate()* z třídy *HibernateDaoSupport*. Proto musí všechny DAO třídy tuto třídu rozšiřovat. Získání seznamu všech zaměstnanců z databáze pak vypadá následovně:

```
public List<Zamestnanec> getZamestnanci() {
     return getHibernateTemplate().find("from Zamestnanec z order by
     upper(z.uzivatelskeJmeno)");
}
```

Spring se sám stará o otevření, uzavření sezení a jelikož mám pro továrnu sezení nakonfigurovaného transakčního manažera, tak jsou všechny operace prováděny v transakci. Aby vše fungovalo správně, definoval jsem každou implementaci DAO třídy v *applicationContext.xml*:

```
<bean id="zamestnanecDao"
    class="informacniSystem.dao.hibernate.ZamestnanecDaoHibernate"/>
```

V každé takovéto definici bych měl ještě uvést odkaz na továrnu sezení (sessionFactory). Nyní však vysvětlím vlastnost `default-autowire="byName"`, kterou mám uvedenou v hlavičce každého konfiguračního souboru. Díky tomuto nastavení analyzuje Spring všechny JavaBeans vlastnosti každého beanu a pro každou z nich se mezi ostatními deklaracemi pokusí najít ten bean, jehož id se shoduje s názvem vlastnosti a tento bean mu nastavit jako závislost. Jelikož mám továrnu sezení definovanou v beanu s `id="sessionFactory"`, tak se každému beanu, který ji vyžaduje, přidělí automaticky. Velikost konfiguračních XML souborů se tak podstatně zmenší.

3.2 Aplikační vrstva

Tato vrstva obsahuje tzv. manažery, což jsou třídy, které vykonávají veškerou aplikační logiku. V mém systému se zde např. počítají hodiny zaměstnancovi docházky, odesílají emaily nebo vybírají projekty či úkoly, které se mají zobrazit. Manažery většinou volají metody z datové vrstvy a naopak metody manažerů jsou volány z vrstvy prezentační. Nikdy by se však nemělo dít naopak. Definice manažerů v *applicationContext.xml* je stejně jednoduchá jako u tříd DAO:

```
<bean id="zamestnanecManager"
    class="informacniSystem.service.ZamestnanecManagerImpl"/>
```

3.3 Prezentační vrstva

V této podkapitole se zmíním o poslední a v případě mé aplikace také nejsložitější vrstvě. Jak jsem již psal dříve, naimplementoval jsem ji v rámci Spring MVC.

3.3.1 Základní konfigurace aplikace

Pro nastartování webové aplikace jsem v konfiguračním souboru *web.xml* musel správně nadefinovat instanci třídy *org.springframework.web.servlet.DispatcherServlet*, která představuje základní jednotku pro komunikaci se servletovým kontejnerem.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>
    org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
```

Moje instance třídy *DispatcherServlet* se jmenuje *dispatcher* a je mapována na všechny požadavky s koncovkou html. Její nastavení se nachází v konfiguračním souboru *dispatcher-servlet.xml*. Tento soubor obsahuje veškeré definice objektů prezentační vrstvy aplikace a je inicializován při startu servletu. Inicializaci souboru *applicationContext.xml* zajistí třída *org.springframework.web.context.ContextLoaderListener*, kterou definuji také v souboru *web.xml*.

3.3.2 Zabezpečení

Zabezpečení mého systému jsem zajistil na úrovni řízení přístupu uživatelů k jednotlivým webovým zdrojům, a to pomocí rámce Acegi Security, který je stažitelný na webových stránkách [10]. Tento rámec využívá AOP možností rámce Spring, a proto je velmi populární. V nedávné době se stal dokonce přímo součástí Springu a vznikl tak Spring Security 2.0. Jelikož však zatím není k dispozici kvalitní dokumentace, použil jsem osvědčenou verzi Acegi Security 1.0.5. Veškerá konfigurace zabezpečení, kterou nyní budu popisovat, je uložena v souboru *security.xml*.

Každá webová stránka v mém systému má definované přístupové role. Jakmile se uživatel poprvé přihlásí do systému, tak se vytvoří objekt *Authentication* reprezentován rozhraním *org.acegisecurity.Authentication*, který v sobě nese uživatelské jméno a heslo. Poté je volána metoda *authenticate* objektu rozhraní *org.acegisecurity.AuthenticationManager*, je jí předán objekt *Authentication* a z databáze ověří, zda je heslo pro zadané uživatelské jméno správné. Pokud ano, tak do objektu *Authentication* priloží informaci o tom, že tento uživatel byl již autentizován. Následuje rozhodnutí o autorizaci.

To má na starosti objekt rozhraní *org.acegisecurity.AccessDecisionManager*, jehož metoda *decide* porovná role uživatele s povolenými rolemi k dané stránce a rozhodne, zda stránku zpřístupní nebo zda zobrazí stránku definovanou pro zakázaný přístup.

Aby se při uživatelově pokusu o přístup k další stránce nemusela celá autentizace opakovat, tak se jeho objekt *Authentication* ukládá do objektu *org.acegisecurity.context.SecurityContextHolder*, který se sváže s aktuálním vláknem programu. Informace o momentálně přihlášeném uživateli pak

mohu získat voláním metody *securityContextHolder.getContext().getAuthentication()*. Uživatelská hesla mám šifrována pomocí algoritmu SHA.

3.3.3 Vrstva Kontroler

To jak funguje zasílání požadavků a odpovědí ve Springu MVC jsem již ukázal v kapitole 1.2.2. na obrazku 1.1. Nyní zde popíši implementaci kontrolerů. V moji aplikaci jsem použil celkem dva typy, a to klasický kontroler zobrazení a formulářový kontroler.

Kontrolery zobrazení slouží k zobrazování dat z aplikační a datové vrstvy, rozšiřují třídu *org.springframework.web.servlet.mvc.ParameterizableViewController* a mají povinnou metodu *handleRequestInternal*, která vrací objekt třídy *org.springframework.web.servlet.ModelAndView*. Tento objekt v sobě obsahuje model a pohled. Model jsou data v instanci třídy *java.util.Map*, která se budou zobrazovat a pohled je logický název pohledu. O tom, jaká JSP stránka se vybere pro tento pohled, rozhodne objekt třídy *org.springframework.web.servlet.view.InternalResourceViewResolver*. Ten mám nastavený tak, že pro každý logický název pohledu se vybere stránka se stejným názvem a příponou .jsp. Definice kontroleru zobrazeni v konfiguračním souboru *dispatcher-servlet.xml* vypadá následovně:

```
<bean name="zamestnanecController"
    class="informacniSystem.webapp.controller.ZamestnanecController">
    <property name="viewName" value="zamestnanci"/>
</bean>
```

Zde mám definovaný logický pohled *zamestnanci*, který se předává objektu *ModelAndView*. Pro tento kontroler se tedy vybere stránka *zamestnanci.jsp*. Nyní však kontroler ještě neví, na kterou požadovanou stránku se má aplikovat. To zajistím přidáním příslušné definice do objektu rozhraní *org.springframework.web.servlet.handler.SimpleUrlHandlerMapping*:

```
<property name="mappings">
    <props>
        <prop key="/zamestnanci.html">zamestnanecController</prop>
    </props>
</property
```

Tato definice říká, že požadavek na stránku *zamestnanci.html* bude zpracovávat *zamestnanecController* a jako odpověď vrátí stránku *zamestnanci.jsp* s příslušným modelem.

Druhým typem je formulářový kontroler, který zajišťuje zpracovávání dat z HTTP formulářů. Každý takovýto kontroler rošiřuje třídu *org.springframework.web.servlet.mvc.SimpleFormController* a obsahuje několik základních metod. První z nich je metoda *formBackingObject*, která vrací tzv. command objekt zapouzdřující data manipulovaná ve formuláři. Může to být buď objekt prázdný nebo získaný z databáze k předvyplnění dat ve formuláři. Další metodou je *referenceData*, ta vrací data, která potřebuji k zobrazení formulářové stránky. Poslední metodou je *onSubmit*, která se vykoná

po potvrzení formuláře. Většinou obsahuje ukládání dat do databáze. Definice formulářového kontroleru v *dispatcher-servlet.xml* je obdobná jako u klasického kontroleru, navíc se musí uvést atribut *commandName*, který označuje název command objektu, *commandClass*, jež představuje třídu, na kterou se budou vázat formulářová data a *successView* obsahující pohled, který se bude aplikovat po úspěšném vykonání formuláře

```
<bean name="zamestnanecFormController"
    class="informacniSystem.webapp.controller.ZamestnanecFormController">
    <property name="formView" value="admin/zamestnanecForm" />
    <property name="successView" value="redirect:../zamestnanci.html" />
    <property name="commandName" value="zamestnanec"/>
    <property name="commandClass"
    value="informacniSystem.model.Zamestnanec"/>
</bean>
```

3.3.4 Vrstva View

Vrstvu View tvoří v mojí aplikaci výhradně jsp stránky. Ty obsahují HTML kód v kombinaci s různými značkami. Já jsem použil Spring knihovnu značek pro práci s formuláři, knihovnu Struts Menu pro realizaci menu a knihovnu JSTL, která umožňuje logické akce, iterování a práci s URL. Jelikož velká část JSP stránek v mém systému obsahuje nějakou tabulku, tak bylo vhodné využít skvělou knihovnu Display tag, která zjednodušuje psaní tabulek a disponuje velkým množstvím funkcí jako například třídění sloupců nebo export tabulek do formátů cvs, XML, pdf a xls. V některých případech (např. kontrola zadávaných dat do formulářů) jsem použil také JavaScript.

3.3.5 Sitemesh

Nedílnou součástí vývoje aplikací v rámci Spring je použití nástroje Sitemesh. Je to aplikační rámec usnadňující tvorbu vzhledu webových aplikací. Využívá se především pro HTML stránky a funguje na principu filtrů. Každý požadavek na HTML stránku je zpracováván sitemesh filtrem, který zjišťuje, zda je pro danou stránku definovaný nějaký dekorátor. Pokud ano, tak tímto dekorátorem stránku odekoruje. Můj systém obsahuje dva dekorátory. Prvním z nich je *default.jsp*. Tento dekorátor se používá na většinu stránek v aplikaci a obsahuje základní HTML značky, hlavičky, menu a zápatí. Každá další stránka pak už nemusí tyto značky obsahovat a tím se stává přehlednější, rychlejší na implementaci a dosahuje jednotného vzhledu s ostatními stránkami. Druhý dekorátor se nazývá *nezalogovany.jsp* a aplikuje se na všechny stránky, kam lze přistoupit bez přihlášení. Nastavení rámce Sitemesh je v souboru *sitemesh.xml*, nastavení dekorátorů pak v souboru *decorators.xml*. Více informací o rámci Sitemesh je možné nalézt na webových stránkách tohoto nástroje [11].

4 Závěr

V této bakalářské práci jsem se zabýval realizací vnitřního informačního systému malé firmy. Nejprve jsem představil některé technologie používané pro vývoj informačních systémů v jazyce Java. Poté jsem podrobně popsal analýzu požadavků za pomoci diagramů případů užití a návrh systému. Nakonec jsem se snažil nastínit implementaci celé aplikace.

Díky podrobnému návrhu a dodržování některých návrhových vzorů se mi podařilo vytvořit robustní a rozšiřitelnou aplikaci, která eviduje docházku zaměstnanců, umožňuje vytváření objednávek, správu zaměstnanců, klientů, všech poskytovaných služeb a zejména projektů. U projektů automaticky sleduje plán oproti skutečnému stavu. V budoucnu bych tento systém mohl rozšířit o vystavování faktur za poskytované služby a evidenci finančních zisků či ztrát firmy. Poté a po doladění některých detailů s konkrétní firmou by tento informační systém mohl být reálně použitelný.

Toto téma jsem si zvolil zejména proto, že jsem se chtěl podrobněji seznámit s moderními technologiemi v oblasti Java Enterprise Edition. To se mi podařilo a naučil jsem se pracovat s velmi populární kombinací aplikačního rámce Spring a rámce pro objektově-relační mapování Hibernate. Seznámil jsem se také s nástroji jako je Sitemesh nebo Acegi Security a zároveň jsem získal další zkušenosti s návrhem informačních systémů.

Všechny tyto znalosti mi jsou velkým přínosem a vývoji informačních systémů v těchto technologiích bych se chtěl věnovat i v budoucnu.

Literatura

[1] Johnson, R., Hoeller, J.: Expert One-on-One: J2EE Development Without EJB. Willey Publishing, Inc., 2004.

[2] Spring Framework, 2008, URL: http://www.springframework.org/download/

[3] Raible, M.: Spring Live. SourceBeat, LLC, 2005.

[4] Hibernate – Relational Persistence for Java and .NET, 2008, URL: http://www.hibernate.org/

[5] PostgreSQL, 2008, URL: http://www.postgresql.org/download/

[6] Apache Tomcat, 2008, URL: http://tomcat.apache.org/

[7] Apache Maven Project, 2008, URL: http://maven.apache.org/download.html

[8] Eclipse, 2008, URL: http://www.eclipse.org/downloads/

[9] Visual Paradigm for UML 6.2 Community Edition, 2008, URL: http://www.visual-paradigm.com/product/vpuml/vpumldownload.jsp?edition=ce

[10] Acegi Security, 2008, URL: http://www.acegisecurity.org/

[11] Sitemesh, 2008, URL: http://www.opensymphony.com/sitemesh/

Seznam příloh

Příloha 1: Instalace aplikace

Tato příloha popisuje kompletní postup instalace aplikace pro operační systém Windows XP. Před samotnou instalací je však potřeba nainstalovat a nakonfigurovat všechny nástroje potřebné pro správný chod aplikace.

Instalace PostgreSQL

Pro instalaci databázového systému PostgreSQL je potřeba spustit soubor *postgresql-8.2.msi*, který se nachází ve složce *install/postgresql*. Při instalaci zaškrtněte volbu *install as a service* a vyplňte *Account name* a *Account password*.

Instalace JDK

Instalace JDK se spustí souborem *jdk-1_5_0_14-windows-i586-p.exe* ve složce *install*. Pro správný běh vývojového prostředí Javy je nutné přidat cestu k adřesáři *bin* do systémové proměnné *PATH*.

Instalace Mavenu

Přesuňte složku *apache-maven-2.0.8* z adresáře *install* na pevný disk. Stejně jako u JDK i zde je potřeba přidat cestu k adřesáři *bin* do systémové proměnné *PATH*.

Instalace Tomcatu

Pro instalaci Tomcatu spusťte instalační soubor *jakarta-tomcat-5.0.28.exe* z adresáře *install*.

Instalace a spuštění aplikace

Přesuňte složku *informacniSystem* na pevný disk. Otevřete tuto složku a editujte mavenovský popisovač projektu *pom.xml*. Na řádku 105 změňte cestu k Vaší instalaci Tomcatu.

Editujte konfigurační souboru *config.properties* ve složce *src/main/resources* a nastavte správně atritubuty *hibernate.connection.username* a *hibernate.connection.password* tak, jak jste nastavili *Account name* a *Account password* při instalaci PostgreSQL. Zde můžete také změnit SMTP server a časový interval pro kontrolu termínu dokončení úkolů a projektů.

V PostgreSQL vytvořte databázi s názvem *informacniSystem* v kódování *UTF8*. To je možné provést buď z psql konzole příkazem `CREATE DATABASE informacniSystem WITH ENCODING 'UTF8';` nebo v grafickém nástroji pgAdmin III.

Nyní je nutné spustit vygenerování tabulek příkazem `mvn compile antrun:run sql:execute -Drecreate.db.skip=false` z příkazové řádky v adresáří informacniSystem.

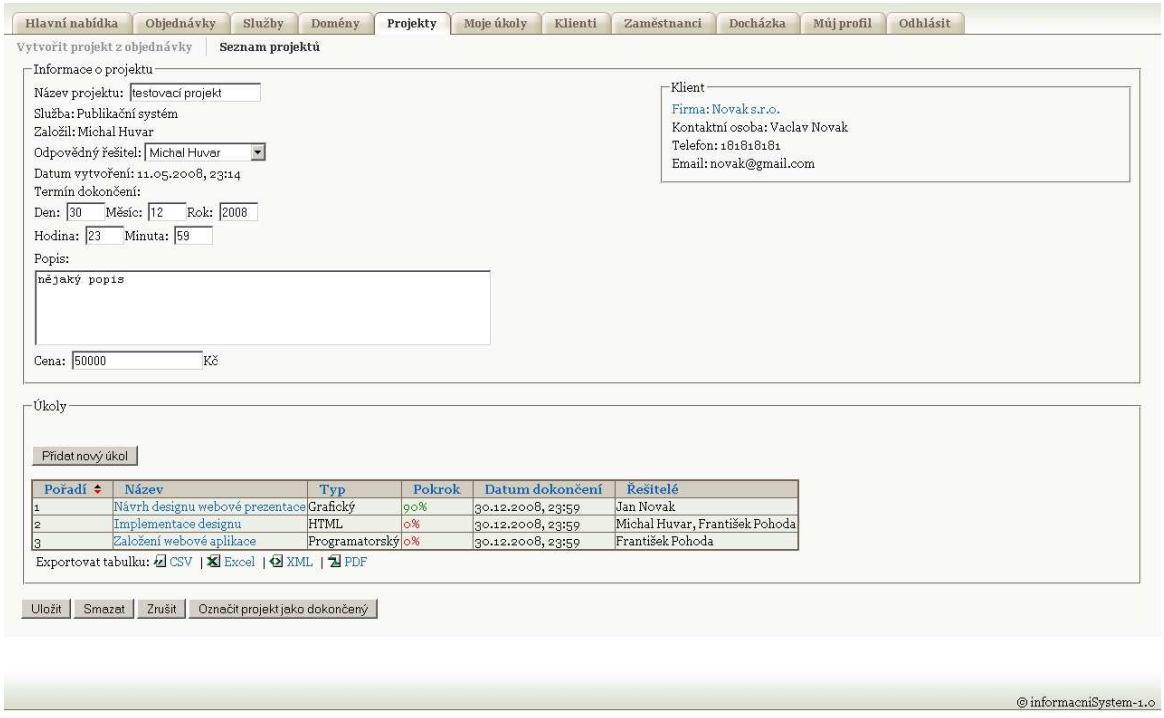
V posledním kroku spusťte příkaz `mvn package cargo:start` pro nasazení aplikace do Tomcatu a její zpřístupnění na adrese *http://localhost:8080/informacniSystem-1.0/*. V aplikaci jsou tři základní uživatelé (v závorkách je uvedeno přihlašovací jméno a heslo):

1) Šéf (admin/admin)
2) Projektový manažer (manager/manager)
3) Běžný zaměstnanec (user/user)

Aplikaci zastavíte kombinací kláves *CTRL+C* v příkazovém řádku. Programovou dokumentaci je možno nalézt v adresáři *apidocs*.

Příloha 2: Ukázka uživatelského rozhraní

Obrázek P2.1 Vytvoření objednávky (klientská část aplikace)



Obrázek P2.2 Detail projektu (zaměstnanecká část aplikace)

Obrázek P2.3 Seznam projektů (zaměstnanecká část aplikace)