VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

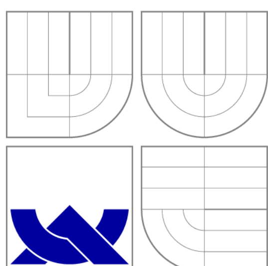
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODERNÍ TECHNOLOGIE PRO VÝVOJ J2EE APLIKACÍ

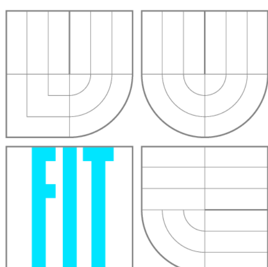
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE | LIBOR ONDRUŠEK
AUTHOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

MODERNÍ TECHNOLOGIE PRO VÝVOJ J2EE APLIKACÍ

THESIS TITLE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LIBOR ONDRUŠEK
AUTHOR

VEDOUCÍ PRÁCE — Mgr. ROMAN TRCHALÍK
SUPERVISOR

BRNO 2009

Abstrakt

Tato bakalářská práce by měla jednoduchým způsobem ukázat využitelnost moderních technologií pro vývoj rozsáhlých podnikových aplikací a názorným způsobem přiblížit vývoj takovýchto aplikací na praktickém příkladu. Jádrem práce je teoretický rozbor technologií *Java*, zejména vývojových rámců *Spring* a *Hibernate*. Teoretická část je názorně doplněna praktickou implementací.

Abstract

This bachelor thesis should be a simple way to show the usefulness of modern technologies for the development process of advanced enterprise applications. This work wants to introduce simple using of this technologies in practical example. The kernel of this work is theoretical analysis of *Java* technology, especially development frameworks like *Spring* and *Hibernate*. The theoretical part is illustrated by practical implementation.

Klíčová slova

J2EE, Spring, Hibernate, Java, MVC, ORM, jetnotkové testování, třívrstvá architektura, analýza aplikace, návrh aplikace

Keywords

J2EE, Spring, Hibernate, Java, MVC, ORM, unit testing, three-tier architecture, application analyse, application concept

Citace

Libor Ondrušek: Moderní technologie pro vývoj J2EE
aplikací, bakalářská práce, Brno, FIT VUT v Brně, 2009

Moderní technologie pro vývoj J2EE aplikací

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Mgr. Romana Trchalíka

. . . .
Libor Ondrušek
19. května 2009

Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu práce za odborné konzultace a vedení práce, Ing. Petru Adámkovi z fakulty informatiky Masarykovy univerzity za otevření světa *J2EE* technologií a všem mým kantorům z fakulty za to, že jsem mohl dojít až sem. Veliké díky taky v neposlední řadě patří mé přítelkyni Martině, jelikož mě po celou dobu práce podporovala.

© Libor Ondrušek, 2009.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

I Základy J2EE 6

1 Úvod 7
1.1 Stručný popis práce . . . 7
1.2 Konvence . . . 8

2 Základní pojmy 9
2.1 Jazyk Java . . . 9
2.2 Pomocné nástroje . . . 10
2.3 Databáze . . . 10
2.4 Vývojová prostředí . . . 10
2.4.1 Eclipse . . . 10
2.4.2 NetBeans . . . 11
2.4.3 IntelliJ Idea . . . 11
2.5 Verzování vyvíjené aplikace . . . 11

II Popis rámců 13

3 Spring Framework 14
3.1 Java EE . . . 14
3.1.1 Komplexní server versus odlehčený aplikační rámec . . . 14
3.1.2 Komplexní Java EE serverová architektura . . . 15
3.1.3 Odlehčený přístup rámce Spring . . . 16
3.2 Architektura . . . 17
3.2.1 Jádro . . . 19
3.2.2 Dependency Injection (Vkládání závislostí) . . . 23
3.2.3 AOP – Programování zaměřené na aspekty . . . 26
3.2.4 Bezpečnostní rámec . . . 32
3.3 MVC Framework . . . 39
3.3.1 Architektura . . . 39
3.3.2 Postup požadavku skrze MVC . . . 39
3.3.3 Vytvoření kontroleru . . . 40

4 Hibernate Framework 43
4.1 Architektura . . . 43
4.1.1 Továrna relace . . . 44
4.1.2 Transakční továrna . . . 45
4.1.3 Poskytovatel spojení . . . 45

4.1.4 Relace, transakce a dotaz 45
4.1.5 Perzistentní objekty 46
4.1.6 Dočasné objekty 47
4.2 Nastavení rámce Hibernate pro použití 47
4.3 Nastavení mapování objektů 48
4.3.1 Mapování pomocí *XML* souboru 48
4.3.2 Mapování pomocí anotací 49
4.4 Nastavení aplikace 52

III Praktická část **54**

5 Analýza a návrh aplikace **55**
5.1 Analýza 55
5.1.1 Základní údaje o projektu 55
5.1.2 Neformální analýza systému 56
5.1.3 Hardwarové technologie 57
5.2 Návrh 57
5.2.1 Implementační technologie 57
5.2.2 Případy užití 57
5.2.3 Aplikační a prezentační vrstva 59
5.2.4 Návrh databáze 59

6 Implementace a testování **61**
6.1 Implementace 61
6.1.1 Rozbor zadání 62
6.1.2 Vlastní vytváření aplikace 65
6.1.3 Problémy při implementaci 65
6.2 Testování 66
6.2.1 Jednotkové testování 66
6.2.2 Integrační testování 67
6.2.3 Akceptační testování 68

IV Zhodnocení **69**

7 Závěr **70**

A Ukázky kompletních kódů **72**
A.1 Anotovaný perzistentní POJO objekt User 72
A.2 Zaváděcí soubor `web.xml` 76
A.3 Soubor deklarace aplikačního kontextu 78
A.4 Anotovaný kontrolér webové vrstvy 80

Seznam obrázků

3.1 Komplexní *Java EE* serverová architektura 15
3.2 Moduly rámce Spring 17
3.3 Životní cyklus beanu ve *Spring* kontejneru 23
3.4 Modulární napříč–jdoucí řešení systému. 27
3.5 Funkce aspektů v aplikaci. 28
3.6 Aplikace proxy objektu. 30
3.7 Bezpečnostní rámec – filtry. 32

4.1 Umístění *Hibernate* v aplikaci. 44
4.2 Vnitřní architektura rámce *Hibernate* 45

5.1 Diagram případů užití. 58
5.2 ER-diagram databáze 60

6.1 Schéma funkce SVN repozitáře. 62
6.2 Schéma třívrstvé webové aplikace. 62
6.3 Základní schema datové a aplikační vrstvy. 64

Seznam ukázek kódu

1.1 Ukázka zdrojového kódu 8
3.4 Vytvoření prototypu. 21
3.5 Programové vytvoření aplikačního kontextu. 22
3.6 Získání *JavaBeanu* z aplikačního kontextu. 22
3.7 Deklarace číselných datových typů. 24
3.8 Deklarace řetězcových datových typů. 24
3.9 Deklarace datového typu `List<T>`. 24
3.10 Deklarace datového typu `Set<T>`. 24
3.11 Deklarace datového typu `Map<K,V>`. 25
3.12 Deklarace `*.properties` souboru. 25
3.13 Deklarace objektu prázdné hodnoty `NULL`. 25
3.14 Implementace editoru vlastností. 26
3.15 Implementace pokynu aspektu. 27
3.16 Implementace definice cíle aspektu. 28
3.17 Deklarace aspektu v aplikačním kontextu. 29
3.18 Deklarace aspektu v aplikačním kontextu pomocí regulárních výrazů. 29
3.19 Deklarace aspektu pomocí anotací. 30
3.20 Deklarace aspektu v aplikačním kontextu pomocí tagů. 31
3.21 Deklarace aspektu pomocí *AspectJ*. 32
3.23 Přímé použití tagů bezpečnostního rámce. 33
3.24 Nepřímé použití standardních tagů. 33
3.26 Nepřímé použití tagů bezpečnostního rámce. 34
3.28 Minimální http konfigurace 35
3.29 Rozšířená http konfigurace 35
3.30 Deklarace uživatelů v aplikačním kontextu. 36
3.31 Autentizace z databáze. 36
3.32 Přidání podpory pro JSR-250 anotace do kontextu. 37
3.33 Zabezpečení metod servisního objektu 37
3.34 Zabezpečení přímo v kontextu. 38
3.35 Zabezpečení v kontextu pomocí intercept-methods 38
3.36 Deklarace rozhodovacího manažera pro web 38
3.37 Deklarace rozhodovacího manažera pro metody 38
3.38 Deklarace umístění anotovaných kontrolérů 40
3.39 Anotovaný multipožadavkový kontrolér 40
3.40 Anotovaný formulářový kontrolér 41
4.1 Perzistentní třída 46
4.2 Deklarace továrny relace. 47
4.3 Deklarace datového zdroje 48

4.4 Mapování pomocí *XML* souboru 48
4.5 Anotace `@Entity` a `@Table` 49
4.6 Příklad anotace perzistentního objektu 50
4.7 Anotace atributů třídy 51
4.8 Nastavení anotovaných perzistentních objektů. 52
6.1 Test pomocí *mock* objektu 67
A.1 Kompletní třída User. 72
A.2 Kompletní zaváděcí soubor web.xml 76
A.3 Kompletní soubor deklarace aplikačního kontextu. 78
A.4 Kompletní anotovaná formulářový kontrolér. 80

Část I

Základy J2EE

Kapitola 1

Úvod

Cíle:

- Uvedení do tématu.
- Seznámení se stylem práce.

Tato práce se zabývá moderními technologiemi, které usnadňují programátorům vyvíjet složité, robustní a škálovatelné podnikové aplikace, pomocí speciálních rámců (*frameworks*). Podrobněji popisuje technologii *Java*[7] a rámce *Spring*[5] a *Hibernate*[8].

1.1 Stručný popis práce

Výše jmenovanné rámce se v současné době těší velké oblibě u mnoha programátorů. *Spring* poskytuje jednoduché řešení (light-weight) pro podnikové aplikace, nabízí možnost použití deklarativního transakčního řízení, vzdáleného přístupu do logiky aplikace a mnoho možností pro perzistenci (ukládání) dat v databázích. Poskytuje také plnohodnotný *MVC – Model-View-Controller (model-pohled-kontrolér)* rámec a jednoduché možnosti pro použití *AOP* (aspect oriented programming) funkcionality.

Druhá kapitola představuje programovací jazyk *Java*, který je jedním možným řešením pro vývoj podnikových (enterprise) aplikací a některé podpůrné programy, které pomáhají vývojářům spravovat a udržovat jejich projekt. Další možností pro vývoj podnikových (enterprise) aplikací je Microsoft platforma .NET, která se Javě do jisté míry podobá, ale nepřináší takovou přenositelnost a další výhody. Na druhou stranu je o něco rychlejší než *Java*. V této kapitole si taky vysvětlíme pojem *EJB – Enterprise JavaBeans Technology*, jenž je nutné znát pro správné pochopení principů tvorby enterprise aplikací technologií *Java*.

Ve třetí kapitole se podíváme zblízka na rámec *Spring* a jeho výhody při programování složitější třívrstvé aplikace. Zejména se zaměříme na aplikační a prezentační vrstvu této aplikace a různé metody konfigurace a to pomocí *XML – eXtensible Markup Language (rozšiřitelný značkovací jazyk)* souborů nebo anotací. Letmo si taky popíšeme jiné možnosti tvorby prezenční vrstvy za pomocí dalších dostupných rámců.

Čtvrtá kapitola nám doplní třetí perzistentní vrstvu aplikace, pro jejíž vývoj se dá s výhodou použít rámec *Hibernate* místo perzistence rámce *Spring*, což nám dává další možnosti a využití. Tato kombinace (*Spring* + *Hibernate*) je jednou z možností hojně používanou a oblíbenou.

Dále práce obsahuje popis aplikace která bude demonstrovat výše zmíněné nástroje a předvedeme její funkčnost. Tato aplikace slouží k ulehčení činností souvisejících se statistikou a vyhodnocováním dat z výrobních procesů firmy, která se zabývá výrobou betonových dílců. Jedná se o jednoduchou třívrstvou aplikaci typu klient-server a má za úkol sbírat data z různých pracovišť a posléze je statisticky vyhodnocovat.

Nakonec práci uzavřeme shrnutím nových poznatků a vědomostí.

1.2 Konvence

U této práce se můžeme setkat s různými styly textů a grafických úprav. Každý tento styl má určitý význam a popisuje jiný typ informace. Jedná se o tyto:

- Krátké a věcné poznatky

Zdrojové kódy se mohou vyskytnout v několika formách. Pokud je krátký kód začleněn přímo do textu, bude vypadat takto: `int i = 0;`

Větší logické celky a části kódu jsou zvýrazněny v samostatném rámečku:

```
// komentar zdrojoveho kodu
int a = 15;
if(a < 5){
  a = a + 1;
}
```

Ukázka 1.1: Ukázka zdrojového kódu

Typy pro jednoduší pochopení jsou uvedeny odsazenou kurzívou. Každá kapitola má určitou skladbu:

- Nadpis.
- Stanovení cílů.
- Obsah.
- Zhodnocení dosaženosti cílů.

Doufám že Vám tyto konvence usnadní orientaci v této práci a pomohou k rychlejšímu zapamatování si o čem pojednává.

Zhodnocení: Čtenář by měl mít představu o tom, čím se tato práce zabývá a jakým stylem je psaná.

Kapitola 2

Základní pojmy

Cíle:

- Seznámení se ze základními pojmy kolem Java technologií.
- Pochopení funkcí nástrojů pro snazší vývoj aplikace.
- Výběr vhodných nástrojů.

2.1 Jazyk Java

Java je moderní objektově orientovaný programovací jazyk, který vyvinula a v roce 1995 představila firma Sun Microsystems. Pro svou přenositelnost je ve světě velmi oblíbený a používaný. Tento jazyk je šířen jako open source (software s otevřeným zdrojovým kódem) a existuje prakticky ve třech technologiích, které se jako celek nazývají *platforma Java*.

- Java ME (Micro Edition) je složena ze dvou konfigurací pro zařízení s omezeným výkonem a používána pro vývoj softwaru mobilních telefonů, PDA, a jiných, většinou přenosných zařízení.
- Java SE (Standard Edition) je prakticky jazyk *Java*, který byl vyvíjen od první verze a rozšiřován. Používá se v největší míře pro desktopové aplikace.
- Java EE (Enterprise Edition) je rozšířením *Java* SE určeným pro vývoj a provoz informačních systémů a podnikových aplikací.

Syntaxe Javy v podstatě vychází ze syntaxe C a C++, ale byla výrazně zjednodušena a usnadněna tím že byly vypuštěny složité konstrukce a přibyla spousta nových rozšíření. Je to platformově nezávislý jazyk. Na systému, kde má aplikace běžet musí být nainstalován pouze virtuální stroj *JVM – Java Virtual Machine*. *Java* je totiž interpretovaný jazyk, kde se při překladu nevytváří strojový kód ale speciální mezikód, tzv. *bytecode*, který poté interpretuje JVM.

V pozdějších verzích jazyka Java byl použit *JIT – Just In Time* (až je třeba) kompilátor, který před spuštěním aplikace přeložil program dynamicky do strojového kódu konkrétního počítače, což vedlo k výraznému zrychlení výsledného programu, ale ke zpomalení startu aplikace. V současné době se s výhodou používají technologie *HotSpot*, které zpočátku mezikód interpretují a dle statistiky získané za běhu aplikace později kompilují části kódu do strojového jazyka počítače tak jako JIT.

2.2 Pomocné nástroje

Pro vývoj Java programů se s velkou výhodou dá používat spousta dalších podpůrných nástrojů, většinou vyvíjených kolem *The Apache Software Foundation* http://www.apache.org Jedním z nich je nástroj *Maven*. V podstatě jde o hodně sofistikovaný nástroj pro správu a údržbu projektů. Původně to byl pokus o zjednodušení kompilovacího a linkovacího procesu v *Jakarta Turbine*. Ten obsahoval několik projektů a každý měl svůj vlastní a od ostatních odlišný sestavovací *Ant* soubor a knihovny byly ukládány v *CVS – Concurrent Version System* (systém pro správu verzí), repozitářích. *Ant* je nástroj pro sestavování Java programů a *Jakarta Turbine* je pro změnu framework pro tvorbu webů. http://turbine.apache.org.

Ale bylo potřeba vytvořit standardní způsob, k sestavování projektů za pomocí transparentní definice toho, z čeho se projekt skládal a publikování sestaveného projektu pro další použití, buďto jako finální produkt, nebo jako *JAR – Java ARchive* knihovny. Výsledkem tohoto snažení byl nástroj, který umožňuje jednoduchou správu, testování, sestavování a jiné operace nad *Java* projektem.

2.3 Databáze

2.4 Vývojová prostředí

V dnešní době je používání vývojových prostředí běžnou praxí a tyto nástroje využívá spousta vývojářů, buďto samostatně, nebo v rámci vývojových týmů. Jejich výhoda spočívá hlavně v komplexnosti a rychlé pomoci při programování, kdy je vývojové prostředí schopno vám napovídat které atributy, nebo metody můžete na daném místě použít, tzv. *Object browser* nebo *Intelli sense* funkce.

Bývá většinou zaměřený na jeden programovací jazyk a obsahuje, editor, kompilátor a často i debugger konkrétního jazyka. Některé vývojové prostředí obsahují i grafické nástroje pro snadnou tvorbu *grafických uživatelských rozhraní*.

U vývojových prostředí pro *Java* technologie bývá paradoxně podpora pro velkou škálu programovacích jazyků a velká spousta textových editorů. Standardem je integrovaný debugger schopný procházet několik vláken a trasovat cestu aplikací.

2.4.1 Eclipse

Eclipse je vývojové prostředí založené na principu vysoké flexibility a modulovatelnosti. Jeho návrh na bázi vložených modulů tzv. *pluginů* dovoluje používat Eclipse pro širokou škálu programovacích jazyků, jako jsou například hlavně *Java*, *C/C++*, *Python*, TEX, *PHP* a jiné.

Prakticky se jedná o *open source* vývojovou platformu, na které se dají vyvíjet další různé rozšíření. Pod hlavním projektem Eclipse se skrývá ještě několik dalších dílčích podprojektů, které vyvíjejí právě výše zmíněné rozšíření. V současnosti je Eclipse vyvíjen na standardu *OSGi R4*, což z něj dělá atraktivní vývojovou platformu.

Výhodou tohoto řešení je nahrávání rozšíření dynamicky, jakmile je toto potřebné, což zaručuje snížení náročnosti na systémové prostředky a zvýšení rychlosti startu aplikace.

2.4.2 NetBeans

Vývojové multiplatformní prostředí podporované firmou *Sun Microsystems* pod *CDDL – Common Development and Distribution License*[1] licencí, které má v oblibě široká masa vývojářů i studentů. Vývoj tohoto produktů probíhá z převážné části v Praze, ve společnosti *Sun Microsystems Inc.*.

Toto prostředí je napsáno v jazyce *Java*, jako ostatně většina jemu podobných, a podporuje jazyky *C++*, *PHP*, *Ruby* a jiné. Umí výhodně zjednodušit práci s prezentačními rámci jako je *Struts* nebo *RoR* a dovoluje vývoj *SOAP – Simple Object Access Protocol*[2] aplikací a webových služeb.

Jeho výhodou je integrovaný *builder* pro grafické uživatelské rozhraní, který ovšem produkuje kód, jenž nelze ručně upravovat. I přesto je to vynikající prostředí hlavně pro začínající vývojáře nad platformou *Java*.

2.4.3 IntelliJ Idea

Komerční nástroj vyvíjený českou firmou JetBrains. Toto prostředí je sině orientováno na vysokou produktivitu a efektivnost kódu. Dovoluje vysokou integrovatelnost s běžnými vývojovými nástrojemi a samozřejmě podporuje, jako dnes už snad všechny vývojové prostředí, práci v týmu na rozsáhlých projektech.

2.5 Verzování vyvíjené aplikace

Pro každou firmu i samostatného programátora je, nebo by alespoň měl být, verzovací *SCM – Source Code Management nebo také Source Configuration Management.* software jedním z nejdůležitějších podpůrných nástrojů při vývoji aplikací. Někomu se může zdát, že je to důležité pouze u velkých a drahých projektů, ale já jsem se přesvědčil, že tyto nástroje umí ušetřit mnoho hodin hledání a usilovné práce, vyskytne-li se nějaký problém. Také se ukázalo, jak výborný pomocník to umí být, pokud potřebujete pracovat na několika pracovištích a nestojíte o stálé hlídání aktuálnosti vašeho zdrojového kódu na dílčích pracovištích.

V podstatě jde v nástroje typu klient – server, které běží někde na serveru, kde se vytvoří zabezpečené úložiště – repozitář a sem se dá posléze zdrojový kód importovat, hierarchicky členit do stromové struktury a následně stahovat, či jinak zpracovávat.

Největší výhodou je zachování a synchronizace změn provedených v kódu, což znamená, že lze získat z historie verzi, na které jsme pracovali dejme tomu před měsícem a můžeme vysledovat co se měnilo a kdo to měnil. Pokud jeden člen týmu změní a posléze nahraje do repozitářů něco, na čem současně pracoval také někdo další, ohlásí se konflikt a poslední nahrávající musí tento řešit. k tomu jsou součástí těchto nástrojů utility, které najdou v textu úseky, které si neodpovídají a značí je. Je na vývojáři, jak tyto konflikty vyřeší.

Při každém nahrávání do repozitáře je nutné dobře a věcně popisovat změny.

jako hlavní zástupce můžeme uvést např. *Subversion (SVN)*, *Concurrent Version System (CVS)*, *SourceSave* a jiné.

[1] Softwarová open source licence používaná převážně firmou Sun Microsystems Inc.

[2] Protokol pro výměnu zpráv skrze síť založených na XML.

Zhodnocení: Seznámili jsme se ze základy Java technologie a podpůrnými nástroji pro pohodlný vývoj složitějších podnikových aplikací a popsali jsme si některé základní principy těchto nástrojů a technologií.

Část II

Popis rámců

Kapitola 3

Spring Framework

Cíle:

- Srovnání různých přístupů k podnikovým aplikacím.
- Základní architektura rámce *Spring*.
- Pochopení důležitých funkcí rámce.

Na tomto místě se nabízí jedna ze základních otázek, kterou si pokládají vývojáři nad platformou *Java EE*. A sice v čem se liší vývoj nad *Spring* framework od běžného vývoje často používaných *Java EE* aplikací?

3.1 Java EE

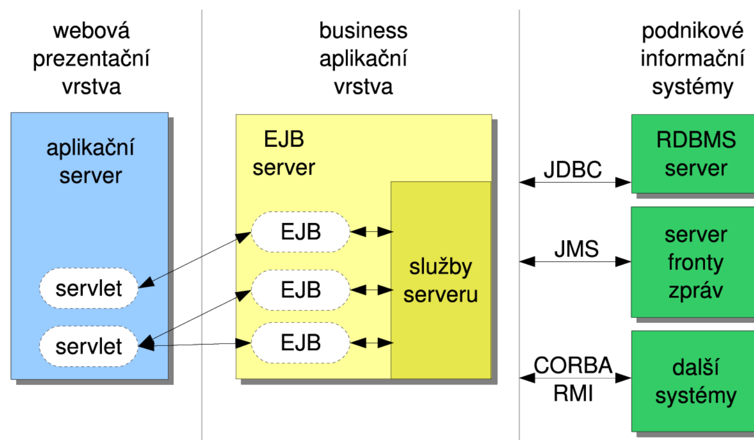
Rozdíly lze najít v architektuře, návrhu a hlavně ve filosofii těchto dvou přístupů k vývoji. Ve skutečnosti byl v pozadí fakt, že se Rod Johnson a Juergen Hoeller (Dva hlavní vývojáři a zakladatelé aplikačního rámce *Spring*.) snažili dokázat, že může existovat lepší přístup k vývoji podnikových *Java* aplikací.

Jelikož použitím kterékoliv z těchto dvou technologií lze dosáhnout stejného výsledku, vyplývá z toho, že se tyto v jádru zásadně neliší. Obě tyto technologie se budou dále bok po boku rozvíjet společně, tak jak tomu bylo doposud, protože se vzájemně doplňují a je pouze na systémových architektech a vývojářích, kterou z nich zvolí pro stavbu daného systému.

Ve stručnosti si přiblížíme hlavní a nejdůležitější rozdíly těchto technologií, které jsou nezbytné k tomu porozumět způsobu výběru vhodné technologie pro stavbu určitého druhu systému.

3.1.1 Komplexní server versus odlehčený aplikační rámec

Základní rozdíl je v architektuře těchto dvou řešení. *Java EE* používá techniku centrálního serveru (tzv. hevyweight řešení). Toto řešení využila spousta velkých výrobců software a vznikly tak komplexní a drahá serverová řešení. Mnoho složitých, převážně komerčních a důležitých aplikací dnes běží na těchto serverech. Na druhé straně *Spring* framework používá odlehčené řešení spojením pouze potřebných komponent namísto jednoho monolitického komplexního serveru (lightweight řešení). Ku příkladu, pokud nepotřebujeme u webového systému, který používá *Spring* řešení, transakce, nebudeme vůbec konfigurovat transakční komponentu.



Obrázek 3.1: Komplexní *Java EE* serverová architektura

3.1.2 Komplexní Java EE serverová architektura

Základem je existence jednoho, nebo více aplikačních *Java* serverů, které vykonávají činnost aplikace na straně serveru. Tyto aplikační servery se často nazývají kontejnery, protože obsahují (*contain*) a vykonávají činnost komponent vytvořených vývojáři. Jak takový server může vypadat vidíme na obrázku 3.1.

Jedná se o typický třívrstvý model *Java EE* systému. Servlety jsou zde webové aplikace, které běží v prezentační, neboli webové vrstvě. Jsou to softwarové komponenty, které běží v *servlet kontejneru* (webovém kontejneru) jako může být např. Tomcat nebo Jetty.

EJB jsou komponenty, které běží v aplikační, neboli business vrstvě a mají na starosti vykonávat veškerou hlavní logiku dané aplikace.

Nejdůležitější body této architektury:

- Aplikační servery musí být nainstalovány a udržovány profesionálním způsobem.
- Softwarové komponenty jsou vytvářeny vývojáři a umisťovány do prostředí těchto serverů
- Komponenty musí být vytvořeny speciálně pro *Java* EE implementace - novější verze nemusí splňovat zpětnou kompatibilitu
- Protože komponenty musí patřit do rodiny *Java* EE, je nutné, aby podporovali implementaci specifických rozhraní a dodržovali hierarchii dědičnosti mezi komponentami
- Nově vytvořené komponenty musí být testovány uvnitř EJB kontejneru, protože tento je svázán s externími zdroji
- Cyklus vývoje může být a bývá velmi časově náročný díky úzké vazbě mezi komponentami a kontejnerem, které se mnohdy ještě hodně komplikují.
- Jelikož jsou servery většinou plnohodnotné, neboli komplexní (odtud název heavyweight) mohou komponenty využívat bohatou množinu kontejnerových služeb dostupných v běžných *Java* EE prostředích

- Ve velkých běhových prostředích, kde je obvykle použito několik aplikačních serverů, může *Java* EE poskytnout lepší škálovatelnost aplikací na straně serveru a distribuovatelnost API standardů jako např. distribuované transakce (XA)

3.1.3 Odlehčený přístup rámce Spring

Jak uvádí kniha *Spring in action*[9]

> Spring is an open source framework, created by Rod Johnson and described in his book *Expert One-on-One: J2EE Design and Development.*

neboli *Spring* je otevřený rámec, vytvořený Rodem Johnsonem a popsaný v jeho knize *Expert One-on-One: J2EE Design and Development* [4]. Spring vznikl jako odezva na pomalý a těžkopádný vývoj klasických *J2EE* aplikací. Umožnil použití jednoduchého a pro programátory lehce pochopitelného přístupu k vývoji těchto aplikací a to vše za uchování všech výhod a propracovaných přístupů z původní *J2EE* technologie, jako například možnost jednoduchého testování pomocí *JUnit*[1], viz. 6.2 a nezávislost na specifických knihovnách a komponentách.

Spring řadíme do skupiny tzv. odlehčených modulárních rámců. Tuto technologii charakterizují výstižně následující pojmy, které jsou s ní těsně spjaty:

Lightweight (Odlehčený) Odlehčenost zde platí opravdu doslova a to co se týká velikosti i režijních nároků. Celý *Spring* rámec je distribuováno jako `*.jar` knihovna v jednom souboru, který má nyní ve verzi 2.5.6 pouhých 5,8 MB, což je na jeho možnosti úctyhodně malá velikost. Pokud se jedná o režie, tyto jsou prakticky zanedbatelné, protože aplikace založené na *Spring* rámci může běžet i v jednoduchých servlet kontejnerech.

Dependency Injection (Vkládání závislostí) Původně se tato technika vyvinula z návrhového vzoru *Inversion of Control*(Obrácení řízení), který dokáže předat řízení běhu z programátorova kódu na podpůrný aplikační rámec. U rámce *Spring* se význam tohoto pojmu posunul natolik, že se odborníci, zabývající se touto problematikou, rozhodli zavést nový pojem, a sice *Dependency Injection*. Princip této techniky je jednoduchý a je popsán v 3.2.2.

Aspect-oriented (Orientace na aspekty) viz. 3.2.3

Container (Kontejner) Lze hovořit o kontejneru, protože obsahuje a spravuje objekty, které jsou vytvořeny uvnitř. Můžeme deklarovat, jak budou objekty vytvářeny a nastaveny a hlavně můžeme nastavovat závislosti mezi nimi za pomocí *Dependency Injection*.

Framework (Rámec) Jelikož tato technologie vytváří komplexní a obsáhlé aplikace z jednoduchých komponent, lze hovořit o aplikačním rámci. Objekty jsou vytvářeny deklarativně a skládány do aplikace, nejčastěji deklarací v `*.xml` souboru. Také tento rámec poskytuje mnoho vnitřních funkcí, jako například transakční zpracování nebo podporu perzistentní vrstvy.

JavaBeans Třídy programovacího jazyka Java, které dodržují konvence zavedené firmou *Sun Microsystems Inc.*. Tj. přístup k atributům třídy je zásadně a pouze přes `get...`

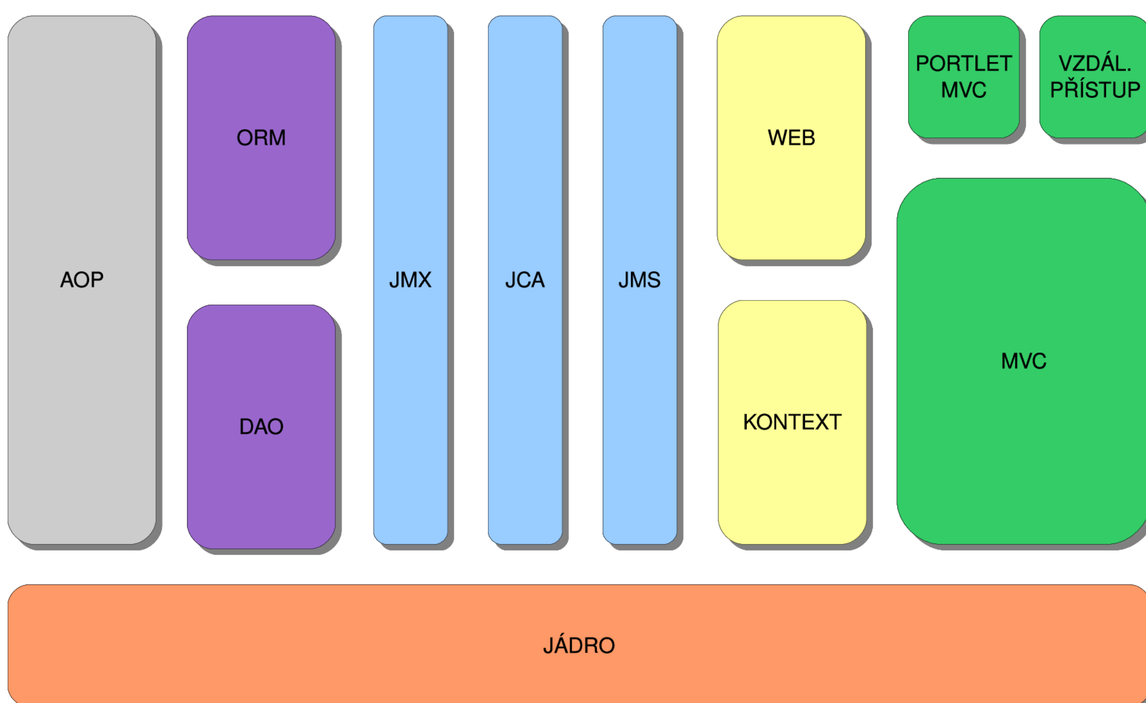
[1] JUnit je knihovna pro vytváření jednotkových testů, dnes hojně používané technologie při vývoji softwaru

a `set...` metody, jak uvidíme v ukázkách kódů. Více na stránkách české Wikipedie [6] Třída `PlantDaoHibernate` na straně **??** je toho příkladem.

V podstatě nám *Spring* poskytuje výborný a jednoduchý přístup k rychlému a efektivnímu vytváření aplikací *J2EE* stylu se zachováním všech dobrých a potřebných vlastností. Poskytuje Hodně vnitřních modulů, které využívají nejmodernější techniky vývoje softwaru. Toto můžeme uplatnit při vývoji složitých webových podnikových aplikací, ale i pro obyčejné a jednoduché programy, používající grafické uživatelské rozhraní, jako je například *Java Swing – knihovna pro grafické uživatelské rozhraní, součást Java Foundation Classes.*.

3.2 Architektura

Rámec se skládá z několika modulů, vystavěných nad hlavním jádrem, viz. obrázek 3.2. Tyto moduly zajišťují vše potřebné pro tvorbu podnikových aplikací, ale na druhou stranu



Obrázek 3.2: Moduly rámce Spring

neuvazují k použití pouze těchto, protože *Spring* umožňuje integraci s mnoha dalšími populárními rámci a knihovnami pro vývoj.

Moduly dohromady tvoří základ rámce a poskytují různé funkce a služby.

Jádro Jak lze vidět z obrázku 3.2, jádro tvoří největší a hlavní modul rámce. Poskytuje základní funkcionalitu a určuje, jak budou vytvářeny, konfigurovány a spravovány *beany*. To má na starost hlavní a základní objekt *BeanFactory*, nebo-li továrna na *beany*. Tento modul je podrobně popsán v kapitole 3.2.1.

AOP Nebo-li programování zaměřené na aspekty. Modul slouží jako základ pro psaní vlastních aspektů aplikace a nabízí k tomuto účelu mnoho přístupů. Nabízí rovněž podporu

pro *AspectJ* což je rozšíření pro snadnější a příjemnější používání *AOP*. Podrobněji se s modulem seznámíme v kapitole 3.2.3.

DAO Data Access Objects. Pomáhá při perzistenci dat a dovoluje udržovat kód pro databáze čistý a jednoduchý se snahou vyvarovat se chybám a problémům. Vytváří vrstvu smysluplných a jednotných výjimek a hlášení při používaní různých databázových řešení. Využívá také AOP modul pro poskytování transakčního zpracování operací mezi databází a aplikací.

ORM Objektově relační mapování. Lze použít místo čistého *JDBC – Java Database Connectivity* (přístup k databázím pomocí jazyka *Java*) přístupu. Podpora je založena na přístupu skrze DAO modul pro velké množství databázových řešení. *Spring* nemá vlastní ORM nástroj, ale nabízí propojení s mnoha populárními ORM nástroji, jako jsou např. *Hibernate*, *JPA, iBATIS, JDO, . . .* Transakční manažer podporuje každý z nich stejně jako JDBC. Rámci *Hibernate* se budeme podrobně věnovat v kapitole 4

JMX *Java Management Extensions.* Nástroj pro tvoření webových, dynamických a modulárních řešení. Dovoluje vytvářet bezpečný přístup pro správu a konfiguraci spuštěné aplikace.

JMS *Java Message Service.* Poskytuje asynchroní komunikaci mezi aplikacemi, které nejsou spolu navzájem spojeny. Jedná se o spojení typu *peer-to-peer*, ve kterém jsou si obě strany na stejné úrovni. Výhodou je vysoká rychlost, bezpečnost a nezávislost na okolních systémech.

JCA *Java EE Connector API.* Poskytuje standardní způsob pro integraci *Java* aplikací s různými druhy podnikových informačních systémů, obsahujících databáze a centralizované řešení. V podstatě je tento modul podobný JDBC přístupu, pouze s tím, že se jedná o vyšší abstrakci připojování, kdy můžeme připojovat celé systémy.

Kontext Též nazývaný aplikační kontext, tvoří páteř a osu aplikace. Definuje, většinou v *XML* souboru, jak bude vytvořena a poskládána aplikace. Určuje závislosti jednotlivých „beanů“. Tento modul podporuje mnoho služeb, jako je posílaní emailů, *JNDI – Java Naming and Directory Interface* (rozhraní pro přístup ke službám Naming service a Directory service), integrace EJB, vzdálené volání služeb a plánování. Je zde přítomna také podpora pro integraci s „šablonovacími“ nástroji, jako je např. *Velocity*, nebo *FreeMaker*.

WEB Podpora speciálních tříd pro *Spring MVC* a *Spring Portlet MVC*. Poskytuje taktéž podporu pro další požadavky typycké pro *WEB*, jako je nahrávání souborů nebo přístup k parametrům *HTTP* požadavku. Navíc obsahuje podporu pro další používané prezentační rámce.

MVC Model-pohled-kontrolér přístup pro prezentační vrstvu aplikace. Odděluje aplikační logiku od šablony pohledu. *Spring* poskytuje vlastní řešení MVC přístupu, ale zároveň nabízí propojení na oblíbené specializované rámce jako jsou *Struts, JSF, WebWork, Tapestry, . . .* Více v kapitole 3.3.

Portlet MVC Podpora pro psaní prezentačních vrstev v „portlet“ stylu. Tzn. zobrazování několika různých a na sobě nezávislých aplikací na jedné stránce.

Vzdálený přístup Přístup do *API* dalších alikací, potřebných pro funkci. Převádí a propojuje vzdálené objekty, jako by byly v lokální aplikaci.

3.2.1 Jádro

Jádrem technologie je kontejner popsaný v 3.1.3. Používá *Dependency Injection* pro správu a konfiguraci komponent, které jsou deklarovány uvnitř. Díky tomuto přístupu jsou komponenty jednoduché, snadno pochopitelné, znovupoužitelné a dají se testovat za pomocí jednotkového testování, které je v současné době považováno za standard v oblasti testovacích nástrojů pro programovací jazyk Java i jiné. Více k testování v kapitole 6.2.

Spring nepřichází pouze s jednou implementací tohoto kontejneru, ale poskytuje jich několik. Můžeme je rozdělit do dvou větších skupin:

1. Továrny tříd – *BeanFactory* se používají pro jednodušší aplikace.

2. Aplikační kontexty – *Application context* jsou výhodné pro složité modulární systémy se velkým množstvím komponent.

Továrna tříd

Jednoduchý kontejner podporující *DI*. Je reprezentován rozhraním

```
org.springframework.beans.factory.BeanFactory
```

Jak jméno napovídá, jedná se o implementaci návrhového vzoru „Továrna“ (*Factory pattern*), který má tu výbornou vlastnost, že umí rozhodnout o vytvoření instance třídy až při běhu programu. Navíc s ním lze dosáhnout lepší efektivity programu, jelikož rozhoduje, zda se vytvoří nový objekt nebo se použije již vytvořený objekt, který není v současné chvíli používán. Toto je s výhodou použito, protože Spring kontejner v sobě potřebuje uchovávat spoustu různých typů objektů.

Po startu systému, obdrží továrna tříd definice objektů, které je potřeba vytvořit a jak mají být na sobě závislé. Tyto bývají většinou definovány v konfiguračním souboru ve formátu *XML*, nebo tzv. *properties* souboru. První z výše jmenovaných se používá nejčastěji, s druhým typem se lze setkat v některých starších aplikacích.

Výsledkem činnosti továrny tříd jsou potom nakonfigurované a vzájemně propojené *JavaBeans*, které jsou připravené k použití. Továrna tříd ale také pečuje o životní cyklus těchto objektů. Stará se o jejich volání, znovupoužití a odstraňování z paměti, pokud jsou takovéto akce nadefinovány.

V aplikačním rámci *Spring* existuje několik implementací továrny tříd, ale nejčastěji se používá

```
org.springframework.beans.factory.xml.XmlBeamFactory
```

která používá právě výše zmiňovaný *XML* definiční soubor. Názorným příkladem takovéhoto souboru by mohl být následující:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">

  <!--Objekty pro praci s~datovou vrstvou-->
  <bean id="plantDao" class="cz.dao.PlantDaoHibernate">
    <constructor-arg value="cz.pojo.Plant" />
    <property name="name" value="NazevDaoObjektu" />
  </bean>

  <!--Manazery pro styk mezi webovou a datovou vrstvou-->
  <bean id="plantManager" class="cz.managers.PlantManagerImpl">
    <property name="plantDao" ref="plantDao" />
  </bean>
</beans>
```

Ukázka 3.1: Fragment definice aplikačního kontextu.

Pro funkčnost této definice musí systém obsahovat tyto rozhraní a třídy:

```java
public interface IPlantDao {
}
public interface IPlantManager {
}

public class Plant {
}
public class PlantDaoHibernate implements IPlantDao {
  private Class<Plant> type;
  private String name;
  public PlantDaoHibernate(Class<Plant> type) {
    this.type = type;
    }
  public void setName(String name) {
    this.name = name;
  }
}
public class PlantManagerImpl implements IPlantManager {
  private plantDao;
  public IPlantDao getPlantDao(){
    return this.plantDao;
  }
  public void setPlantDao(IPlantDao plantDao){
    this.plantDao = plantDao;
  }
}
```

Ukázka 3.2: Objekty JavaBean.

Pokud dostane aplikační rámec tyto definice, provede za pomocí *DI* na pozadí následující kód:

```java
IPlantDao plantDao = new PlantDaoHibernate(new Plant());
plantDao.setName = "NazevDaoObjektu";
IPlantManager plantManager = new PlantManagerImpl();
plantManger.setPlantDao = plantDao;
```

Ukázka 3.3: Kód na pozadí.

Spring by dle tohoto kódu vytvořil objekt `plantManager`, který by byl závislý na objektu `plantDao` a nastavil by jejich atributy. Tyto objekty by v systému vystupovali jako jediné instance, jelikož by byly vytvořeny podle návrhového vzoru *Singleton* (jedináček). Pokud by jsme požadovali po továrně tříd instanci tohoto objektu, vždy bychom dostali jeden a ten stejný objekt. To platí samozřejmě i pro volání ze všech objektů, které jsou na tomto jedináčkovi závislé.

Pokud bychom si přáli, aby nebyly objekty takto vytvořeny, museli bychom v elementu `bean` použít atribut `singleton=,,false‘‘`. V takovémto případě by se inkriminovaný objekt vytvořil jako tzv. *prototyp* a při každém požadavku na továrnu tříd o tento *Bean* by se vytvořila nová instance.

Příklad:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans">

  <!--Objekty pro praci s~datovou vrstvou-->
  <bean id="plantDao" class="cz.dao.PlantDaoHibernate" singleton="false">
    <constructor-arg value="cz.pojo.Plant" />
    <property name="name" value="NazevDaoObjektu" />
  </bean>
```

Ukázka 3.4: Vytvoření prototypu.

Jelikož továrna tříd nepředstavuje hlavní výhody rámce *Spring*, ale pouze základní funkcionalitu správy komponent, lze ji v tomto stavu používat pouze pro jednoduché aplikace (nejčastěji pro mobilní zařízení).

Pro plné využití všech funkcí rámce se dále budeme zabývat výhradně aplikačními kontexty.

Aplikační kontext

Zatímco továrnu tříd je nutno inicializovat programově, aplikační kontext lze inicializovat a spustit také deklarativně. Toho se využívá při spouštění webových aplikací, běžících v některém webovém kontejneru, jako je např. *Tomcat* nebo *JBoss*.

Tento typ kontejneru je reprezentován rozhraním

`org.springframework.context.ApplicationContext`

a vytváří se na základě požadavku továrny tříd pro podporu služeb poskytovaných rámcem, jako je třeba rozpoznávání textových řetězců dle národního prostředí nebo schopnost zpřístupnit události zpracovatelům událostí. Z hlediska objektového návrhu prakticky rozšiřuje továrnu tříd. Toto rozšíření dovoluje použití ve velkých modulárních systémech a představuje plnou sílu rámce *Spring*.

Proti továrně tříd zapouzdřuje mnohem více funkčnosti, jako například:

- Internacionalizaci pomocí definic textových zpráv (I18N).
- Genericitu pro nahrávání souborových zdrojů (např. obrázků).
- Zachytávání událostí objektů za pomocí tzv. *listeners – posluchačů.*

Aplikační kontexty jsou v rámci *Spring* implementovány několika nejběžněji používanými způsoby:

`ClassPathXmlApplicationContext` nahraje definice z *XML* souboru umístěném v *classpath (cesta v aplikaci, kde jsou umístěny zkompilované objekty).*

`FileSystemXmlApplicationContext` nahraje definice z *XML* souboru, který je umístěn v souborovém systému.

`XmlWebApplicationContext` nahraje definice z *XML* souboru ve struktuře webové aplikace.

Tudíž se tyto implementace v podstatě liší pouze tím, kde se snaží najít definiční soubory pro sestavení aplikace.

Jako ukázka může sloužit například druhá uvedená implementace. Při deklaracích aplikačního kontextu se pro určení názvů a cest používá tzv. *Ant hvězdičková notace.*

```
ApplicationContext ctx = new FileSystemXmlApplicationContext("foo.xml");
```

Ukázka 3.5: Programové vytvoření aplikačního kontextu.

Ostatní implementace se používají podobným způsobem.

Z takto vytvořeného kontextu, ale i továrny tříd, lze získat vytvořené objekty (*beany*) následujícím způsobem.

```
MyBean myBean = (MyBean) ctx.getBean("myBean");
```

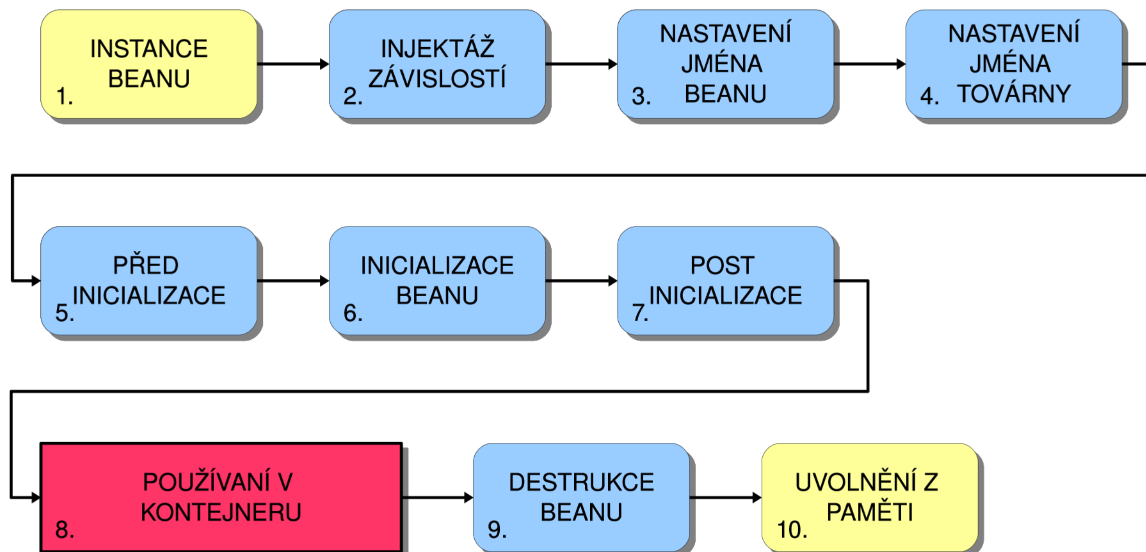
Ukázka 3.6: Získání *JavaBeanu* z aplikačního kontextu.

Životní cyklus beanu

V klasické *Java* aplikaci je životní cyklus objektu docela jednoduchý. Instance objektu se vytvoří klíčovým slovem `new` a objekt je ihned použitelný. Pokud se objekt určitou dobu nepoužívá, *Garbage Collector* se postará o jeho uvolnění z paměti, je-li to nutné.

Ve *Spring* kontejneru je tento životní cyklus propracovanější. Jak je vidět na obrázku 3.3, skládá se z více operací.

1. Na pozadí ze použije klíčové slovo `new` pro vytvoření instance *beanu.*
2. Použití *DI* pro injektáž závislostí.
3. Pokud *bean* implementuje `BeanNameAware`, použije *Spring* metodu `setBeanName()` pro uložení jména z identifikátoru.
4. Pokud *bean* implementuje `BeanFactoryAware`, použije se metoda `setBeanFactory()` pro identifikaci továrny tříd, která *bean* vytvořila.



Obrázek 3.3: Životní cyklus beanu ve *Spring* kontejneru

5. Spuštění metody `postProcessBeforeInitialization()`.
6. Pokud *bean* implementuje rozhraní `InitializingBean`, zavolá se metoda `afterProperitiesSet()`, nebo speciálně deklarovaná metoda.
7. Spuštění metody `postProcessAfterInitialization()`.
8. *Bean* je připraven k použití v kontejneru.
9. Pokud *bean* implementuje rozhraní `DisposableBean`, zavolá se metoda `destroy()`, nebo speciálně deklarovaná metoda.
10. Objekt je uvolněn z paměti pomocí *Garbage Collectoru*

3.2.2 Dependency Injection (Vkládání závislostí)

V předchozích ukázkách kódu na byly použity dva způsoby vložení závislosti mezi objekty. První byl u třídy `PlantDaoHibernate` a byl to tzv. *Constructor Injection*. Tato technika používá konstruktoru třídy, pro vytvoření závislosti definovaného objektu. V mnoha případech je to jediný způsob jak předat argumenty a používá se hlavně tam, kde potřebujeme v aplikaci použít *genericitu*.

Naopak druhý způsob (třída `PlantManagerImpl`) ukazuje použití *Setter Injection*. Tato technika si klade za podmínku použití přesné specifikace *JavaBeans* u spravované třídy. Pokud budeme používat *Beany* třetích stran, které tuto specifikaci nedodržují, musíme použít první metodu. U atributu `name` je ukázáno, jak *Spring* dokáže vkládat hodnoty atributům při vytváření.

Inicializace atributů objektů

JavaBean atributy jsou privátní (z vnějšku nedostupné) a bývají navázány na veřejné metody k získaní hodnoty a nastavení hodnoty. Nastavovací metodu, neboli *setter*, používá

Spring pro inicializaci hodnot atributu. *Spring* disponuje pokročilými konverzními schopnostmi, proto je inicializace atributů objektů velmi jednoduchá a intuitivní. Lze snadno deklarativně přiřazovat všechny primitivní typy jazyka *Java*, kolekce a často používaných tříd.

Nejprve ukázka deklarativní inicializace některých datových typů.

1. Primitivní datové typy, např. `int` a `float`.

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="celeCislo" value="34"/>
  <property name="desetineCislo" value="45.3"/>
</bean>
```

Ukázka 3.7: Deklarace číselných datových typů.

2. Pole řetězců `String[]`

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="pole" value="prvni,druhy,treti,ctvrty"/>
</bean>
```

Ukázka 3.8: Deklarace řetězcových datových typů.

3. Seznam celých čísel `List<int>`

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="seznam">
    <list>
      <value>145</value>
      <value>267</value>
      <value>32</value>
    </list>
  </property>
</bean>
```

Ukázka 3.9: Deklarace datového typu `List<T>`.

4. Množina desetinných čísel `Set<double>`

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="mnozina">
    <set>
      <value>34.6</value>
      <value>23.8</value>
      <value>20</value>
    </set>
  </property>
</bean>
```

Ukázka 3.10: Deklarace datového typu `Set<T>`.

5. Mapa klíčů (`int`) a řetězců `Map<int, String>`

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="mapa">
    <map>
      <entry key="34">
        <value>zili</value>
      </entry>
      <entry key="6">
        <value>byli</value>
      </entry>
      <entry key="345">
        <value>ryli</value>
      </entry>
    </map>
  </property>
</bean>
```

Ukázka 3.11: Deklarace datového typu `Map<K,V>`.

6. Položky tzv. *properties* souboru. Tvar záznamu je: `key=value`.

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="mapa">
    <props>
      <prop key="adresa">Bozetechova</prop>
      <prop key="cislo">23</prop>
      <prop key="nadmVyska">216.58</prop>
    </props>
  </property>
</bean>
```

Ukázka 3.12: Deklarace `*.properties` souboru.

7. Speciální objekt – `NULL`

```
<bean id="Objekt" class="org.ukazka.Objekt">
  <property name="nullObjekt"><null/></property>
</bean>
```

Ukázka 3.13: Deklarace objektu prázdné hodnoty `NULL`.

Konverze funguje na principu *javovské reflexe* a tzv. *editorů vlastností – Property Editors*, definovaných ve standardu *JavaBeans*. Nejprve se *Spring* pomocí reflexe snaží určit o jaký datový typ se jedná a v závislosti na výsledku vyhledává vhodný editor vlastností.

Spring používá pro primitivní datové typy editory z balíků *Javy* a také editory implementované přímo v rámci *Spring*. Pro vlastní třídy je pak nutné napsat vlastní editor vlastností a zaregistrovat ho v aplikačním kontextu. Tyto editory je vhodné implementovat jako rozšíření třídy `PropertyEditorSupport` z balíku `java.beans.PropertyEditorSupport` viz. následující ukázka

```java
public class ConcreteClassEditor extends PropertyEditorSupport {

    private IConcreteClassManager concreteClassManager;

    public ConcreteClassEditor(IConcreteClassManager concreteClassManager) {
  this.concreteClassManager = concreteClassManager;
    }

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
  ConcreteClass concreteClass = concreteClassManager.getByName(text);
  setValue(concreteClass);
    }
}
```

Ukázka 3.14: Implementace editoru vlastností.

Autowiring

Autowiring, neboli automatická inicializace atributů je velmi výhodné rozšíření rámce *Spring* pro velké aplikace, kde je vidina mnoha objektů a tudíž i rozsáhlá deklarace aplikačního kontextu. *Spring* dokáže rozeznat jak automaticky spojovat *beany* dohromady, dle nastavení typu automatické inicializace. Mohou to být tyto:

`byName` Podle jména. Pokusí se najít *bean* v kontejneru, jehož jméno nebo ID je stejné jako jméno atributu. Pokud nenajde vhodného kandidáta, atribut zůstane neinicializovaný.

`byType` Podle datového typu. Pokusí se v kontejneru najít jeden *bean*, jehož typ se shoduje s typem inicializovaného atributu. Pokud nenajde, platí to co v předchozím případě. Pokud najde více vhodných kandidátů, bude vyhozena výjimka.

`constructor` Podle konstruktoru. Hledá jeden nebo více *beanů* s parametry jednoho z konstruktorů inicializovaného *beanu*. V případě nejednoznačného výběru se opět vyhodí výjimka.

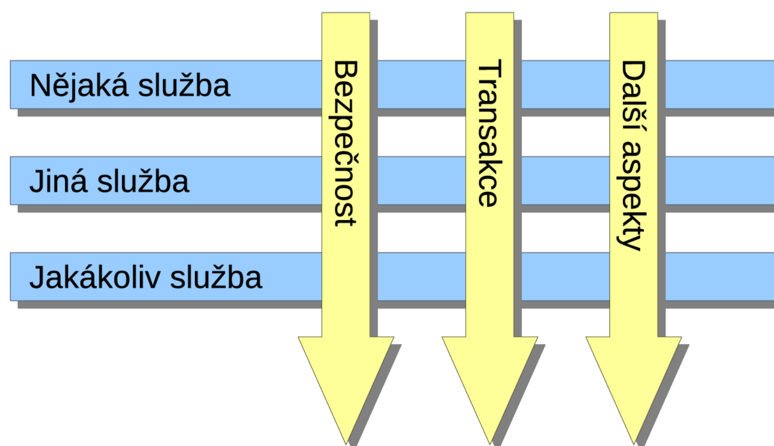
`autodetect` Autodetekce. Nejprve se pokusí inicializovat atribut podle konstruktoru a poté podle typu. Nejednoznačnost bude obsloužena stejně jako u automatické inicializace typu `constructor` nebo `byType`.

Jak je vidět, každý z těchto typů je něčím specifický a ne vždy je vhodné ho použít. Nejvíce jistoty správné inicializace je u typu `byName`, jelikož pokud budeme aplikaci tvořit systematicky a promyšleně, dostaneme v každém případě správě inicializované atributy.

3.2.3 AOP – Programování zaměřené na aspekty

Při vývoji rozsáhlých systémů se setkáme s často se opakující funkcionalitou, jako je logování (zaznamenávání důležitých událostí), autorizace nebo správa transakcí Tato funkcionalita se většinou dotýká spousty částí sytému a vytváří tzv. *napříč–jdoucí koncerny*, tj. je do aplikace navázána ve více částech a opakuje se. Více informací lze najít na Wikipedii[6] pod heslem „Cross-cutting“.

AOP se prakticky snaží oddělit tyto koncerny od *business* logiky aplikace a rozebrat ji na jasné části – moduly, do tzv. pokynů – *advices*. Výhoda tohoto řešení je, že potom můžeme deklarativně určit, kde všude se bude jaký pokyn volat a aplikovat. Místa volání těchto pokynů se označují *point-cuts*. *Spring* sám potom zajistí, aby se daný pokyn zavolal přesně v tomto místě a vykonal jemu příslušnou činnost. Aspekty poskytují čistější alter-



Obrázek 3.4: Modulární napříč–jdoucí řešení systému.

nativu k dědičnosti a delegování u objektově orientovaného programování. Díky tomu, že se funkcionalita vyskytuje pouze na jednom místě a do aplikace se přidává deklarativně, vytváříme čistější a jednodušeji udržovatelný kód.
Spring podporuje většiny typů pokynů. Hlavně jsou to pak pokyny–interceptory (*Interception Around Advice*). Mohou být aplikovány před i po volání metody, pouze před (*Before Advice*), po vyhození výjimky (*Throws Advice*) a po úspěšném návratu z volané metody (*After Returning Advice*).

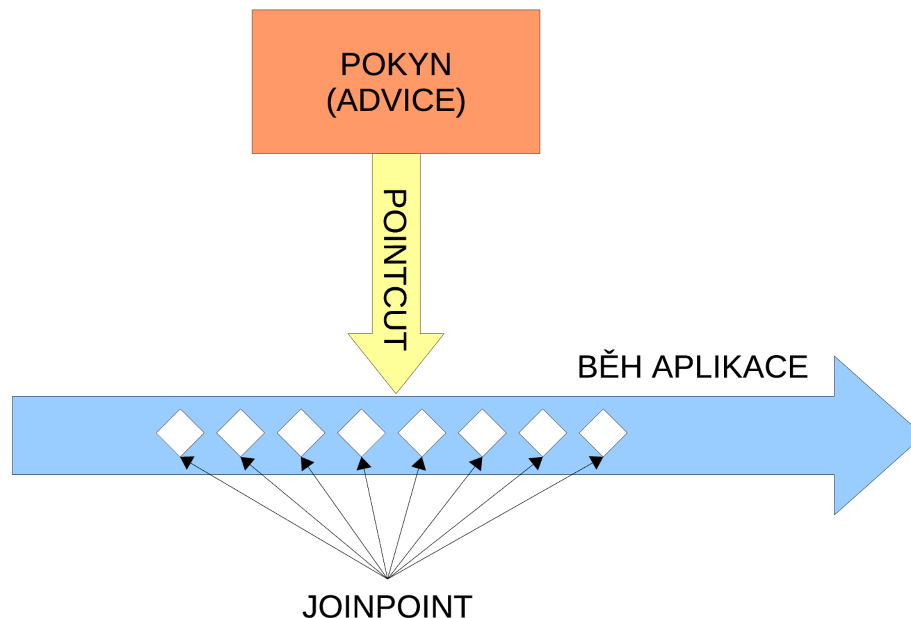
Pro lepší představu funkčnosti viz. obrázek 3.5. Pro důkladné pochopení této technologie je nutné znát význam pojmů, spojených s *AOP*. Příklady jsou převzaty a mírně upraveny podle knihy Pro Spring[3] a ukazují nejčastější typ pokynů – *interceptor*.

Advice – pokyn Činnost, která má být vykonávána. Zapouzdřuje logiku celého aspektu. Definuje co a kdy se vykonává.

```
public class SampleInterceptor implements MethodInterceptor {
  public Object invoke(MethodInvocation invocation) throws Throwable
  {
    System.out.print("Startuji pokyn ...");
    Object retVal = invocation.proceed();
    System.out.println("Koncim pokyn ...");
    return retVal;
  }
}
```

Ukázka 3.15: Implementace pokynu aspektu.

Jak je vidět, pokyn je implementací rozhraní `MethodInterceptor`. Musí implementovat metodu `invoke`, ve které volá metodu, na kterou se bude pokyn aplikovat a vrací



Obrázek 3.5: Funkce aspektů v aplikaci.

její návratovou hodnotu.

Joinpoint – vstupní bod Je místo v aplikaci, kde lze zavolat pokyn k vykonání. Přesněji je to místo, kde může být aspekt zapojen do činnosti aplikace.

Pointcut – definice cílů Definuje kdy má být pokyn vykonán. Je to přesné určení objektu a jeho metody, na kterou bude pokyn aplikován. V rámci *Spring* jsou tyto definice reprezentovány rozhraním `org.springframework.aop.Pointcut`.

```
  public class SamplePointcut implements Pointcut {
  public ClassFilter getClassFilter() {
    return new ClassFilter() {
      public boolean matches(Class clazz) {}
    };}
  public MethodMatcher getMethodMatcher() {
    return new MethodMatcher() {
      // vraci true pro dynamicke definice cilu, false pro staticke
      public boolean isRuntime() {
      }
      // vraci true pro vyhovujici tridy a metody
      public boolean matches(Method m, Class targetClass) {
      }
      // vraci true pro vyhovujici tridy, metody jejich parametry
      public boolean matches(Method m, Class targetClass, Object[]
      args) {}
    };}
}
```

Ukázka 3.16: Implementace definice cíle aspektu.

Tato třída je schopna určit, zda vyhovuje definici pro jakoukoliv trojici (třída, metoda, parametry metody) a rozhodnout, jedná-li se o statické, či dynamické definice cílů.

Aspect – Advisor V podstatě jde o kombinaci pokynu a definice cíle (*Advice* a *Pointcut*). Tím je definováno vše co aplikace potřebuje vědět o aspektu, čili co a kdy se má vykonat. V rámci *Spring* jsou reprezentovány implementacemi rozhraní `org.springframework.aop.Advi` Nejvíce se používá implementace `DefaultPointcutAdvisor`, podporující všechny hlavní důležité typy pokynů. V ukázce lze vidět, jak se deklarativně konfigurují aspekty v aplikačním kontextu.

```
<bean id="sampleInterceptor" class="SampleInterceptor" />
<bean id="samplePointcut" class="SamplePointcut" />
<bean id="sampleAdvisor"
  class="org.springframework.aop.support.DefaultPointcutAdvisor" >
  <property name="advice" ref="sampleInterceptor" />
  <property name="pointcut" ref="samplePointcut" />
</bean>
```

Ukázka 3.17: Deklarace aspektu v aplikačním kontextu.

Z pohledu programátora je velmi výhodné používat další implementace aspektu využívající regulárních výrazů pro definici cílů. Jedná se o třídu `RegexpMethodPointcutAdvisor`.

```
  <bean id="setAndSendAdvisor"
  class="org.springframework.aop.support.RegexpMethodPointcutAdvisor"
  >
  <property name="advice" ref="sampleInterceptor"/>
  <property name="patterns">
      <list>
      <value>.*set.*</value>
      <value>.*send.*</value>
    </list>
  </property>
</bean>
```

Ukázka 3.18: Deklarace aspektu v aplikačním kontextu pomocí regulárních výrazů.

Introducion Dovoluje přidávat nové metody nebo atributy do již existujících tříd, které jsou aspektem obsluhovány. Např. pokud máme pokyn pro auditování objektů, který má na starost udržovat informaci o poslední editaci objektu, můžeme do tohoto objektu pomocí aspektu přidat metodu `setModified(Date)` a atribut `modified`, které nám budou zajišťovat uchování posledního data úpravy tohoto objektu.

Target – cíl Je to objekt, na který se bude aplikovat pokyn.

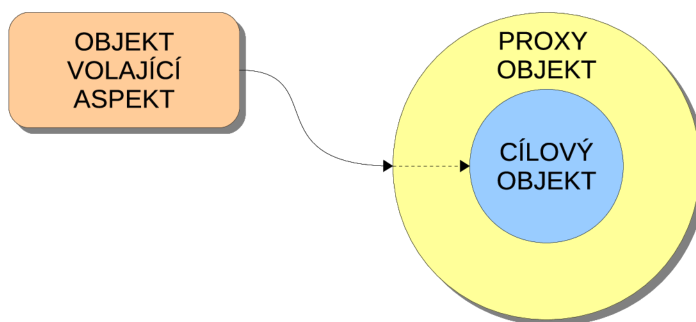
Proxy Objekt vytvořený po aplikaci pokynu. Je to nový objekt, který obaluje cílový objekt pro vykonání pokynu a vystupuje jako původní – cílový objekt viz. obrázek 3.6. Pro vytváření se používají standardní dynamické proxy jazyka Java, nebo knihovna *CGLIB*. První z uváděných způsobů je značně limitován rozhraními, jelikož vytvořený proxy objekt je možné přetypovat pouze na rozhraní, které implementuje cílový objekt. V rámci *Spring* je ten to způsob realizován pomocí třídy `ProxyFactoryBean`.

Druhá uváděná knihovna umožňuje aplikaci aspektů i na konkrétní třídy, ale musíme ji začlenit do aplikace, což přináší další závislosti.

Co se týká klientských objektů, mohou být *Target* a *Proxy* totožné objekty.

Weaving Je proces aplikace aspektu na cílový objekt, za vytvoření nového *proxy* objektu. Aspekt je aplikován na cílový objekt ve specifikovaném místě (*joinpoint*). *Weaving* může odstartovat v několika bodech během životního cyklu objektu.

- V čase kompilace. Toto využívá rámec *AspectJ*.
- Během zavádění třídy *Class Loaderem* do *JVM*.
- Za běhu. Kdykoliv během provádění aplikace pomocí výše zmiňovaných dynamických proxy objektů.



Obrázek 3.6: Aplikace proxy objektu.

Spring AOP patří mezi tři nejrozšířenější *AOP* rámce s tím že poskytuje podporu i pro další níže jmenované. Těmi jsou *AspectJ* http://www.eclipse.org/aspectj a *JBoss AOP* http://www.jboss.org/jbossaop.

Podpora *AOP* v rámci *Spring* je implementována několika způsoby. Příklady jsou převzaty z knihy *Spring in Action*[9] a pro lepší pochopení přeloženy.

- Klasické výše popsané proxy objekty.

- Anotace *@AspectJ* (pouze ve verzi 2.0).

```
  @Aspect
public class Obecenstvo {

  public Obecenstvo() {
  }

  @Pointcut("execution(* *.perform(..))")
  public void performance() {
  }

  @Before("performance()")
  public void vzitMisto() {
    System.out.println("Obecenstvo se posadilo.");
  }
```

```
    @Before("performance()")
    public void vypnoutTelefony() {
      System.out.println("Obecenstvo vypnulo sve telefony.");
    }

    @AfterReturning("performance()")
    public void potlesk() {
      System.out.println("TLESK TLESK TLESK TLESK");
    }

    @AfterThrowing("performance()")
    public void nespokojenost() {
      System.out.println("Vratte nam nase penize!");
    }
  }
```

Ukázka 3.19: Deklarace aspektu pomocí anotací.

- Standardní *POJO* objekty (pouze ve verzi 2.0). Deklarují se v aplikačním kontextu.

```
    <bean id="audience"
    class="com.springinaction.springidol.Audience" />
  <aop:config>
    <aop:aspect ref="audience">
      <aop:before
        method="takeSeats"
        pointcut="execution(* *.perform(..))" />

      <aop:before
        method="turnOffCellPhones"
        pointcut="execution(* *.perform(..))" />

      <aop:after-returning
        method="applaud"
        pointcut="execution(* *.perform(..))" />

      <aop:after-throwing
        method="demandRefund"
        pointcut="execution(* *.perform(..))" />
    </aop:aspect>
  </aop:config>
  </beans>
```

Ukázka 3.20: Deklarace aspektu v aplikačním kontextu pomocí tagů.

- Propojení na *AspectJ* rámec.

```
  public aspect JudgeAspect {
  public JudgeAspect() {
  }

  pointcut performance() : execution(* perform(..));
  after() returning() : performance() {
    System.out.println(criticismEngine.getCriticism());
  }
  // propojeni
  private CriticismEngine criticismEngine;
  public void setCriticismEngine(CriticismEngine criticismEngine) {
    this.criticismEngine = criticismEngine;
  }
}
```

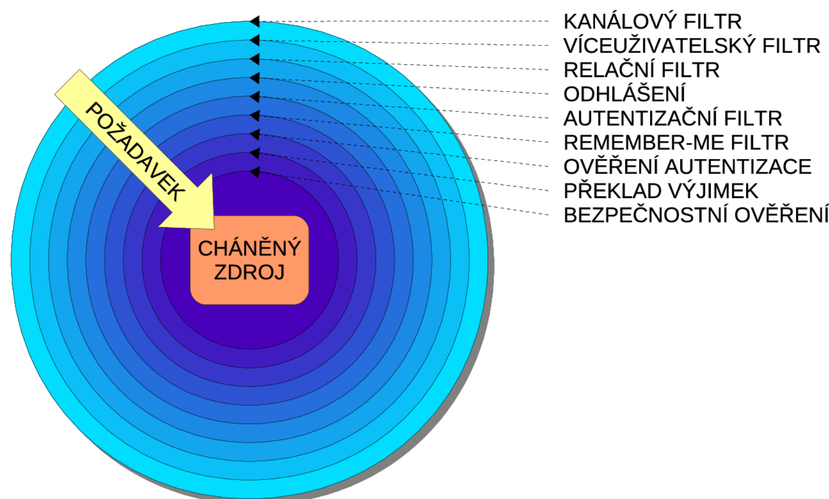
Ukázka 3.21: Deklarace aspektu pomocí *AspectJ*.

AOP zde dovoluje definici cílů pouze v metodách a to jako přímé volání z kódu, nebo deklarativně.

3.2.4 Bezpečnostní rámec

Každá aplikace, ke které má mít přístup více uživatelů, musí řešit autorizaci a autentizaci uživatelů. Zde se s výhodou dá využít výše popsaná *AOP* technika, jelikož zabezpečení patří mezi napříč-jdoucí koncerny. V aplikačním rámci *Spring* je bezpečnost implementována jako vložený rámec *Spring Security* (http://static.springsource.org/spring-security/site/index.html), který vznikl převzetím rámce *Acegi Security* v roce 2008 a zakomponováním přímo pod hlavičkou rámce *Spring*. I přesto se dá tento zabezpečovací rámec používat samostatně a dokonce i pro jiné než webové aplikace.

Bezpečnostní rámec funguje v podstatě jako sada nastavovatelných filtrů, které prověřují požadavek a buďto ho propustí, nebo vyhodí výjimku. Přesně jako na obrázku 3.7.



Obrázek 3.7: Bezpečnostní rámec – filtry.

Ve verzi rámce *Spring* 2.5 došlo k zjednodušení aplikace bezpečnosti, díky zjednodušenému zápisu automatické konfigurace do aplikačního kontextu a anotacím.

Způsobů, jak správně nakonfigurovat *Spring Security* rámec je hned několik. Díky rychlému rozvoji celého rámce *Spring*, je v současné době otázkou několika řádků v kódu. Dříve se používaly složitější deklarace bezpečnostních filtrů. Nyní lze díky anotování a konfiguraci pomocí prostoru jmen (*namespace*) toto podstatně zjednodušit.

V dalším textu bude ukázán a vysvětlen jeden ze způsobů konfigurace, který je použit u aplikace, jež je součástí této práce a několik příkladů z webových stránek *Spring Security*[2], kde lze také najít další užitečná nastavení a informace.

Jak bylo výše zmíněno, je výhodné použít zjednodušené konfigurace za pomocí jmenných prostorů, což v podstatě není nic jiného než definované *XML* schéma, dle standardu. Odkaz na toto schéma musíme nejprve vložit do deklarace aplikačního kontextu. Díky dobré modularitě rámce *Spring*, použijeme s výhodou samostatný *XML* soubor, v našem případě `applicationContext-security.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security-2.0.1.xsd">
  ...
</beans:beans>
```

Ukázka 3.22: Import bezpečnostního prostoru jmen.

Tato definice dovoluje přímé používání tagů bezpečnostního rámce ve tvaru

```
<authentication-provider atribut="hodnota"/>
```

Ukázka 3.23: Přímé použití tagů bezpečnostního rámce.

a standardních *bean* tagů ve tvaru

```
<beans:bean atribut="hodnota"/>
```

Ukázka 3.24: Nepřímé použití standardních tagů.

Pokud bychom nepoužívali samostatný soubor *XML*, museli bychom respektovat upřednostnění standardních tagů pro deklaraci *beanů*. Pro ilustraci malá ukázka. Při definici prostoru jmen tímto způsobem:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-2.0.4.xsd
    ">
    ...
</beans>
```

Ukázka 3.25: Nemodulární import bezpečnostního prostoru jmen.

bychom pak museli deklarační tagy používat ve tvaru

```
<security:authentication-provider atribut="hodnota"/>
```

Ukázka 3.26: Nepřímé použití tagů bezpečnostního rámce.

a další *beany* by potom byly deklarovány obvyklým způsobem.

Prostor jmen je navržen tak, aby se dali jednoduchým způsobem nakonfigurovat nejběžnější používané akce pro zabezpečení aplikace. Tento návrh je založen na spojení s aplikačním rámcem *Spring*. Jmenný prostor lze rozdělit na tyto základní části:

Web/HTTP Security Zabezpečení různých oddělených částí webové aplikace. Nastavují se zde filtry a služby pro autentifikaci uživatele podle přihlašovacího jména.

Business Object (Method) Security Zabezpečení metod objektů servisní vrstvy.

AuthenticationManager Obsluhuje autentizační požadavky z dalších částí rámce. Základní je již registrován přímo v prostoru jmen.

AccessDecisionManager Rozhodovací manažer pro zabezpečení částí webu a metod. Základní manažer je registrován automaticky, ale můžeme použít a registrovat vlastní.

AuthenticationProviders Poskytovatel autentizace definuje způsob, jakým jsou uživatelé autentizovaní. Jmenný prostor poskytuje mnoho běžných těchto poskytovatelů a také možnost zaregistrovat svůj vlastní.

UserDetailsService Detaily autentizovaného objektu. Podobný poskytovateli autentizace, často používaný ostatními *beany*.

Zabezpečení částí webové aplikace

Nyní k vlastní konfiguraci zabezpečení. Zde uvedený způsob odráží nastavení v přiložené aplikaci. Není to minimální nastavení, ale jsou zde použity některé zajímavé možnosti bezpečnostního rámce.

Jako první musíme nadeklarovat filtry, které zapojují bezpečnostní rámec do činnosti. Toto provedeme v základním startovacím deklaračním souboru webové aplikace, umístěném v jejím hlavním adresáři `WEB-INF/web.xml` a z kterého bere servletový kontejner nastavení spouštěné aplikace.

```
<!-- Deklarace zakladniho filtru ramce Spring Security -->
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter
  -class>
  </filter>

<!-- Deklarace mapovani filtru pro Spring Security -->
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Ukázka 3.27: Nastavení filtrů pro bezpečnost.

Nyní jsem deklarovali aplikačnímu rámci *Spring* filtr, který se bude aplikovat pomocí *AOP* na každou volanou stránku v aplikaci pomocí *AOP proxy* objektu `DelegatingFilterProxy`. *Bean* s názvem „springSecurityFilterChain" je definován ve jmenném prostoru pro podporu zabezpečení webových aplikací.

Minimální konfigurace pro přístup k částem webové aplikace zapsaná do deklaračního *XML* souboru aplikačního kontextu vypadá takto:

```
<http auto-config='true'>
  <intercept-url pattern="/**" access="ROLE_USER" />
</http>
```

Ukázka 3.28: Minimální http konfigurace

Zde je deklarováno, že všechny *URL* jsou dostupné uživateli s implementovaným rozhraním `UserDetailsService` a přístupovou rolí `ROLE_USER`. Takto minimální konfigurace, ale asi nebude dostačovat většině aplikací. proto ji rozšíříme.

```
<http auto-config="true" lowercase-comparisons="true">
  <intercept-url pattern="/secure/admin/**" access="ROLE_ADMIN" requires-
  channel="https"/>
  <intercept-url pattern="/secure/client/**" access="ROLE_ADMIN, ROLE_USER"
  />
  <intercept-url pattern="/*" access="ROLE_ANONYMOUS,ROLE_ADMIN, ROLE_USER"
  />

  <form-login login-page="/login.html" authentication-failure-url="/login.
  html?login_error=1" />

  <!-- Nastaveni viceuzivatelskeho pristupu -->
  <concurrent-session-control max-sessions="1" expired-url="/session-expired
  .html" />
  <remember-me key="dfsdfs154gf654d68fg54fd2" />
  <anonymous username="-" />
</http>
```

Ukázka 3.29: Rozšířená http konfigurace

V tomto případě říkáme, že porovnávání má být *case-insensitive*, deklarujeme části webu a jim příslušné úrovně přístupu. Deklarace se procházejí postupně za sebou a bezpečnostní rámec se zachová podle první shody. Proto je nutné na první místa dávat co nejspecifičtější deklarace a postupovat směrem k větší abstrakci. U umístění „secure/admin“ navíc říkáme že je požadováno šifrované spojení *SSL*.

Dále je uvedeno, kde se bude hledat a jak se jmenuje stránka pro přihlášení uživatele a kam přesměrovat v případě nezdařené autentifikace. V úplně spodní části je řečeno jak se bude aplikace stavět k víceuživatelskému přístupu. Zde konkrétně určujeme jednomu uživateli jednu relaci. Pokud se stejný uživatel přihlásí odjinud během existující relace, bude původní autentifikace a relace zneplatněna a aplikace nás přesměruje na „expired-url“.

Deklarace „remember-me key“ definuje konkrétní řetězec, který bude použit pro vytvoření *cookie* k zapamatování přihlášeného uživatele. A konečně poslední řádek říká, jaká je textová reprezentace nepřihlášeného uživatele.

Aby mohl bezpečnostní rámec podle něčeho autentizovat, musí vědět, jakým způsobem a odkud ověřovat uživatele. Proto musíme deklarovat poskytovatele autentizace – `authentication-provider`.

Nejjednodušší způsob, jak toho dosáhnout je, uvést přímo v deklaraci jména, hesla a role uživatelů.

```
<authentication-provider>
  <user-service>
    <user name="jimi" password="jimispassword"
              authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="bobspassword"
              authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
```

Ukázka 3.30: Deklarace uživatelů v aplikačním kontextu.

Tento způsob je samozřejmě pro reálné aplikace zcela nevhodný, ale je zase nejvýhodnější pro testování aplikace, protože nepotřebujeme řešit žádné jiné napojení a zdroje dat. Během autentizace se provede tzv. *in-memory authentication*.

Poskytovatelé autentizace tedy mohou být různých druhů a technologií. *Spring Security* podporuje autentizaci se serveru *LDAP*, *JDBC* autentizaci, *OpenID* a konečně autentizaci z databáze, kterou používá i přiložená aplikace.

```
<!-- Poskytovatel autentizace -->
<authentication-provider user-service-ref="userDao">
  <password-encoder ref="passwordEncoder">
    <salt-source user-property="username" />
  </password-encoder>
</authentication-provider>

<!-- Autentizacni manager -->
<bean id="daoAuthenticationProvider"
      class="org.springframework.security.providers.dao.
      DaoAuthenticationProvider">
  <property name="userDetailsService" ref="userDao" />
  <property name="passwordEncoder" ref="passwordEncoder" />
  <property name="saltSource" ref="saltSource" />
</bean>
```

```
<!-- Sifrovaci algoritmus -->
<bean id="passwordEncoder"
      class="org.springframework.security.providers.encoding.
      ShaPasswordEncoder">
  <constructor-arg value="256"></constructor-arg>
</bean>

<!-- Kodovaci sul (unikatni kodovaci klic) -->
<bean id="saltSource"
      class="org.springframework.security.providers.dao.salt.
      ReflectionSaltSource">
  <property name="userPropertyToUse" value="username"></property>
</bean>
```

Ukázka 3.31: Autentizace z databáze.

Na prvním řádku je vidět typ poskytovatele autentizace. Je jím datový objekt „userDao“, který má díky rámci *Hibernate* skrze transakční management přístup do databáze. Více v kapitole 4. Dále je zde deklarován šifrovací algoritmus *SHA-256* a unikátní kódovací klíč, který před zašifrováním přidá k ověřovanému řetězci uživatelův login. Tím „přisolí“ tento řetězec a sváže ho s objektem.

Zabezpečení metod servisních objektů

Co se týká zabezpečení metod servisních objektů, lze říci, že se díky *Java* anotacím značně zjednodušil proces vývoje. Pokud tedy používáme *Javu* 5 a vyšší, která podporuje *JSR-250* bezpečnostní anotace, můžeme toto zjednodušení výhodně použít.

Nejprve je nutné uvést v aplikačním kontextu, že budeme tento typ anotací používat, aby mohl aplikační rámec vyhledat, které metody jsou zabezpečeny.

```
<global-method-security secured-annotations="enabled"
                        jsr250-annotations="enabled"/>
```

Ukázka 3.32: Přidání podpory pro JSR-250 anotace do kontextu.

Pak už nám nic nebrání v jednoduchém použití anotací před metodou v rozhraní, které deklaruje objekt, jehož metody chceme zabezpečit.

```
public interface BankService {

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
  }
```

Ukázka 3.33: Zabezpečení metod servisního objektu

Metody servisních objektů lze taky jednoduše zabezpečovat deklarací přímo v aplikačním kontextu.

```
<global-method-security>
  <protect-pointcut
      expression="execution(* com.mycompany.*Service.*(..))"
      access="ROLE_USER"/>
</global-method-security>
```

Ukázka 3.34: Zabezpečení přímo v kontextu.

Tato deklarace zabezpečuje všechny objekty z balíku `com.mycompany`, jejichž jméno končí na `Service` a mohou je spouštět pouze uživatelé s rolí `ROLE_USER`.

Alternativní zápis může vypadat například takto.

```
<bean:bean id="target" class="com.mycompany.myapp.MyBean">
  <intercept-methods>
    <protect method="set*" access="ROLE_ADMIN" />
    <protect method="get*" access="ROLE_ADMIN,ROLE_USER" />
    <protect method="doSomething" access="ROLE_USER" />
  </intercept-methods>
</bean:bean>
```

Ukázka 3.35: Zabezpečení v kontextu pomocí intercept-methods

Tento zápis povoluje spouštět metody objektu `MyBean` omezeně. Uživatelům s rolí `ROLE_ADMIN` metody s prefixem `get` a `set` a uživatelům s rolí `ROLE_USER` metody s prefixem `get` a metodu `doSomething`.

Vlastní rozhodovací manažer

Jak bylo výše zmíněno jmenný prostor již registruje základního rozhodovacího manažera. Pokud by situace vyžadovala použít jiného, můžeme ho vytvořit a následně zaregistrovat do aplikačního kontextu.

Pro zabezpečení částí webové aplikace by mohla být použita následující deklarace.

```
<http access-decision-manager-ref="myAccessDecisionManagerBean">
  ...
</http>
```

Ukázka 3.36: Deklarace rozhodovacího manažera pro web

Pro zabezpečení metod servisních objektů by to bylo podobně.

```
<global-method-security access-decision-manager-ref="
myAccessDecisionManagerBean">
  ...
</global-method-security>
```

Ukázka 3.37: Deklarace rozhodovacího manažera pro metody

V předchozím textu bylo předvedeno jednoduché nastavení aplikace pro použití zabezpečovacího rámce *Spring Security.* Způsob jakým to bylo demonstrováno patří k novému přístupu zabezpečení. Dříve bylo nutné deklarovat řádově několik filtrů, což vnášelo větší možnosti chyb a zpomalovalo vývoj. Tento „starý“ způsob jde díky zpětné kompatibilitě používat dále, ale je výhodnější a přehlednější použít způsob nový.

3.3 MVC Framework

Jelikož je tato práce zaměřena hlavně na aplikační a datovou vrstvu, následující kapitola bude velmi stučně popisovat vrstvu prezentační. MOžností, které lze vyžít při tvorbě této vrstvy je mnoho a v této práci není prostor na to, aby se všechny možnosti demostrovaly.

Rámec *Spring* poskytuje základní prostředky pro vytváření „prezentační“ vrstvy. Dovoluje připojení speciálních rámců, které se touto problematikou zabývají, jak bylo popsáno výše, a nabízí vlastní řešení architektury *MVC – Model-View-Controller*, které má několik výhod oproti externímu řešení.

Hlavní výhodou je nesporně plná stoprocentní integrace do rámce *Spring* a další novou od verze 2.5 je to možnost vytváření anotovaných kontrolérů, což zrychluje práci a zpřehledňujě aplikaci jako takovou.

3.3.1 Architektura

Tento rámec je implemantací návrhového vzoru *MVC*, což ukazuje na princip komunikace dotaz-odpověď.

Model je část, ve které jsou zapouzdřena data, která se budou zobrazovat jako naformátovaný pohled (nejčastěji se používá šablonovací techologie *JSP – Java Server Page*), za pomoci kontroléru, ječ celý tento proces řídí.

Hlavním objektem, který řídí požadavky je `DispatcherServlet`, ten rozhoduje, který kontrolér bude zpracovávat dotaz od uživatele. Proto musí být správně zaregistrován a deklarován ve spouštěcím souboru `web.xml`.

3.3.2 Postup požadavku skrze MVC

Požadavek je instancí třídy `HttpServletRequest` a je zpracován v několika bodech.

- Do objektu, který představuje výše zmíněný hlavní objekt *MVC* se uloží aplikační kontext daného servletu, lokalizátor národního prostředí, detektor motivu, popřípadě požadavek typu `MultipartHttpSertvletRequest`.
- V aplikačním kontextu se vyhledají všechny objekty typu `HandlerMapping` a rozhodne se který kontroler zpracuje požadavek.
- Zavolají se všechny definované předzpracovávající interceptory.
- Požadavek se předá vybranému kontroléru a ten vygeneruje model a určí nazev pohledu.
- Zavolají se všechny definované postzpracovávající interceptory.
- V aplikačním kontextu se vyhledají všechny objekty typu `ViewResolver`, které převedou název pohledu na cestu k souboru se šablonou stránky.
- Vytvořený model se zobrazí skrze vybraný pohled.

3.3.3 Vytvoření kontroleru

Jelikož je to důležitý objekt, který řídí zpracování už konkretizovaného požadavku, ukážeme si, jak vypadá a jaké jsou základní možnosti a funkce.

Prakticky existují dva způsoby vytváření těchto pobjektů. Jeden je deklarace v aplikačním kontextu, kde se určí mapování požadavků na kotroléry. To vyžaduje, aby byly všechny takové kontrolery potomky třídy `Controller`, nebo tříd od ní odvozených. Dalším způsobem, a novým trendem, je deklarace kontrolerů přímo v jeho třídě pomocí anotování.

Druhým případem se budeme zabývat trochu více, jelikož je použit v přiložené aplikaci a je jednodušší na používání.

Anotované kontroléry

V první řadě je nutné sdělit aplikačnímu rámci *Spring*, kde má dané kontrolery hledat. To zajistíme deklarací tohoto místa v aplikačním kontextu.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http
://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p" xmlns:context="http://
  www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www
  .springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context http://www.
        springframework.org/schema/context/spring-context-2.5.xsd">

  <!-- Detektory pohledu -->
  <bean class="org.springframework.web.servlet.view.
  InternalResourceViewResolver"
    p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />

  <!-- Vyhledani kontroleru -->
  <context:component-scan base-package="cz.zpsv.databaseZQ.controllers" />
</beans>
```

Ukázka 3.38: Deklarace umístění anotovaných kontrolérů

V tomto souboru je určeno, kde hledat kontroléry a kde hledat pohledy. Konkrétní kontroler musí být anotovaný nasledujícím způsobem. Existují prakticky dva základní druhy konrolérů. První z nich je klasický multipožadavkový kontrolér, který nahrazuje původní třídu `MultiActionController`. Ten může fungovat pro několik různých mapování.

```
@Controller
public class SpecimensListController {

    private ISpecimenManager specimenManager;

    @Autowired
    public void setSpecimenManager(ISpecimenManager specimenManager) {
  this.specimenManager = specimenManager;
    }
```

```
    /* Vytvori model obsahujici stranku vzorku.
     * @return Model {@link ModelMap} pro webovou sablonu.*/
    @RequestMapping("/secure/client/specimens-list")
    public String specimensListHandler(
      @RequestParam(required = false, value = "action") String action,
      @RequestParam(required = false, value = "id") Long id,
      ModelMap model, HttpServletRequest request) {

  // pozadavek na mazani vzorku
  if (id != null && action.equalsIgnoreCase("delete")) {
      this.getSpecimenManager().deleteByPK(id, request);
  }

  ValueList valueList = getSpecimenManager().getValueListHandlerHelper()
    .getValueList("specimensList",
      new ValueListInfo("id", ValueListInfo.DESCENDING));
  model.addAttribute("list", valueList);
  return "/secure/client/specimens-list";
    }
}
```

Ukázka 3.39: Anotovaný multipožadavkový kontrolér

Kde anotace `@RequestMapping` určuje mapování požadavku, dle kterého se tento kontrolér vybral. Jelikož jde o multipožadavkový kontrolér, může se tato anotace vyskytovat i vícekrát v jedné třídě, ale pouze vždy u jedné metody, která zmíněný požadavek obslouží. tyto metody moho vracet několik různých datových typů, z nichž nejčastějším bývá `String` (určuje název pohledu), a podle tohoto typu se rohoduje další postup určení pohledu.

Další typ, formulářový kontrolér, dovoluje zobrazovat klasické webové formuláře a zapouzdřuje jejich funkcionalitu. Zde se doplňuje anotace `@RequestMapping` o parametr `method`, který určuje o jakou metodu odesílání požadavku se anotovaná metoda bude starat.

```
@Controller
@RequestMapping("/secure/client/specimen-detail")
@SessionAttributes("specimen")
public class SpecimenDetailFormController {

    private IConsistenceTypeManager consistenceTypeManager;

    @InitBinder
    public void initBinder(WebDataBinder binder) {
  SimpleDateFormat dateFormat = new SimpleDateFormat("dd.MM.yyyy");
  dateFormat.setLenient(true);
  binder.registerCustomEditor(Date.class, new CustomDateEditor(
    dateFormat, false));
  binder.registerCustomEditor(ConsistenceType.class,
    new ConsistenceTypeEditor(consistenceTypeManager));
    }

    // Vlozi do atributu modelu vsechny typy konzistenci
    @ModelAttribute("consistenceTypes")
    public Collection<ConsistenceType> populateConsistenceTypes() {
  return consistenceTypeManager.getAll();
    }
```

```
     /**
      * Odesle data do databaze
      */
     @RequestMapping(method = RequestMethod.POST)
     public String prosessSubmit(@ModelAttribute Specimen specimen,
       BindingResult result, SessionStatus status, ModelMap model) {
   new SpecimenValidator().validate(specimen, result);
   if (!result.hasErrors()) {
     // zde se pokracuje pokud na formulari nebyly chyby
       return "redirect:specimen-detail.html?id=" + savedSpecimen.getId()
         + "&action=detail";
   }
   model.addAttribute("hasErrors", new Boolean(true));
   model.addAttribute("formEditable", true);
   return "/secure/client/specimen-detail";
     }

     @Autowired
     public void setConsistenceTypeManager(
       IConsistenceTypeManager consistenceTypeManager) {
   this.consistenceTypeManager = consistenceTypeManager;
     }

   /**
      * Vytvori a predvyplni formular.
      * @param id Id vzorku.
      * @param model Model {@link ModelMap} pro zobrazeni ve formulari.
      */
     @RequestMapping(method = RequestMethod.GET)
     public void setupForm(
       @RequestParam(required = false, value = "id") Long id,
       @RequestParam(required = false, value = "action") String action,
       ModelMap model) {
     ...
     }
     }
}
```

Ukázka 3.40: Anotovaný formulářový kontrolér

Oba dva tyto příklady jsou okomentovány v kódu, proto není potřeba podrobnějšího vysvětlení. Podrobný popis tohoto způsobu vytváření lze najít v referenční dokumentaci rámce *Spring*[5], nebo na stránkách http://blog.springsource.com, kde jsou poznámky a návody vlastního autora anotovaných kontrolerů Juergean Hoellera. Ovšem zde je nutné již rozumět funkcím rámce *MVC* a klasickému způsoby vytváření kontrolérů.

Zhodnocení: Tento rámec prináší velké usnadnění a zachovává potřebnou funkcionalitu technologie *J2EE*, což ho posunuje na přední místa v trendu vývoje složitých podnikových aplikací. Jedná se o dosti rozsáhlou problematiku, proto bylo poukazováno hlavně na nejdůležitější části a aspekty tohoto rámce.

Kapitola 4

Hibernate Framework

Cíle:

- Pochopení objektově-relačního mapování.
- Ukázka deklarace databázových tabulek v kódu.

Perzistentní datové úložiště je v dnešní době jedna z nejdůležitějších částí aplikací. Dále v textu budeme datová pezrzistentní úložiště nazývat zkráceně databáze. Jsou na něj kladeny zvláštní nároky na spolehlivost rychlost, snadnou obsluhu atd. Existuje mnoho produktů, které jsou schopny tyto požadavky naplnit. Většina z nich implementuje dotazovací jazyk SQL a rozšiřuje ho vlastním způsobem. Takové rozdíly mohou být zdrojem zvýšených pracovních nákladů, jestliže jsme nuceni aplikaci předělat na podporu jiné databáze.

Dnešní databáze jsou ve většině případů relační a moderní aplikace zase používají objektový model návrhu. Díky tomuto rozdílu potřebujeme nástroj, který bude jednoduše a pro programátora transparetně převádět data z objektové formy do relačního modelu a naopak a bude schopný jednotným způsobem přistupovat k různým databázovým produktům.

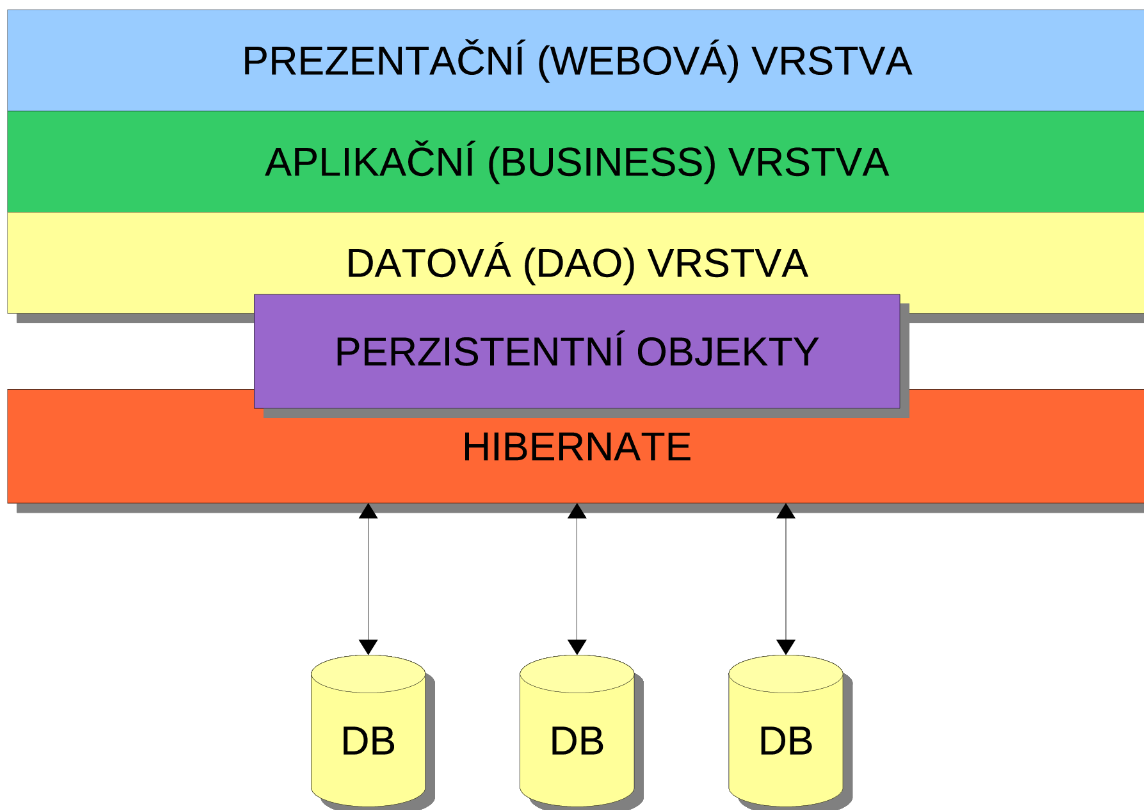
Výše popsané požadavky přesně splňuje nástroj *Hibernate*, vyvíjený firmou Red Hat jako součást produktu *JEMS – JBoss Enterprise Middleware System*. Jedná se *open-source* knihovnu poskytující rámec pro *ORM* (objektově-relační mapování). Zároveň řeší rozdíly mezi různými *SQL* dialekty podporovaných databází.

Jelikož aplikační rámec *Spring* podporuje přímou konektivitu pro rámec *Hibernate*, je to vhodné řešení implementace datové vrstvy do *J2EE* aplikací používajících tento rámec.

Jak se dá od takto zaměřeného produktu očekávat, *Hibernate* poskytuje řešení transakčního zpracování, podporu vyrovnávací paměti (*cache*), databázové datové typy, polymorfismus, dědičnost, oběktový jazyk pro sestavování objektových dotazů a další. Jak bylo již výše zmíněno, najdeme zde i jednotné *API* pro práci s různými typy databází, což značně rozšiřuje možnosti výsledného produktu.

4.1 Architektura

Rámec *Hibernate* vytváří vrstvu mezi datovou vrstvou aplikace a databázovým strojem (viz. obrázek 4.1), nastavenou pomocí `*.properties` souborů a `*.xml` souborů definujících mapování objektů do databáze. Tyto soubory lze od verze *Springu* 2.5 nahradit anotacemi přímo ve třídách perzistentních objektů. Tuto techniku používá také přiložená aplikace a budeme se jí zabývat dále v této práci. Co se týká hlavní funkčnosti a skladby rámce, lze říci



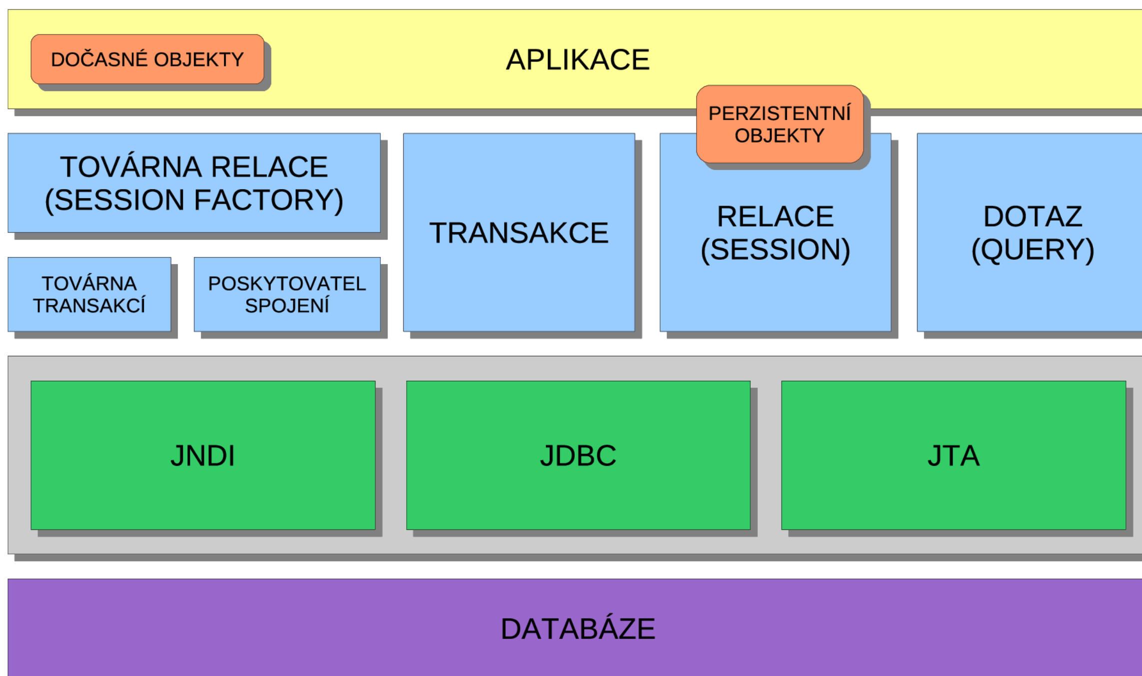
Obrázek 4.1: Umístění *Hibernate* v aplikaci.

že vše se odehrává kolem relace (sezení, *session*). To je implementací rozhraní `Session` a obaluje spojení s databází pomocí technologie *JDBC – Java Database Conectivity* a konvertuje objekty do a z relační databáze. Pomocí objektu relace se vytváří a používá transakční zpracování. Taktéž udržuje vyrovnávací pamět první úrovně. Jelikož je z pohledu náročnosti na systémové prostředky jednoduché vytvářet tyto objekty a děje se tak prakticky pokaždé, když je v aplikaci vyvolán požadavek na databázi (nejčastěji *HTTP*), nemusí vznikat obava z přetížení systému při velké frekvenci dotazů.

Na dalším obrázku (4.2) je znázorněna vnitřní architektura rámce *Hibernate* s návazností na okolní objekty a technologie.

4.1.1 Továrna relace

Nejdůležitějším prvkem v této architektuře je továrna relace *Session Factory*, která je implementací návrhového vzoru továrna – *Factory* pro zajištění vytváření relace spojení. Jedná se standardní *JavaBean* objekt, konfigurovaný nejčastěji pomocí `*.properties` souboru umístěném v aplikaci. To pomáhá udžovat konfiguraci centralizovanou a přehlednou. Tento objekt je nutné vytvořit jako jedináčka *singleton* a pokud aplikace používá více databází, pro každou databázi zvlášť. To zajišťuje většinou rámec *Spring* v aplikačním kontextu.



Obrázek 4.2: Vnitřní architektura rámce *Hibernate*

4.1.2 Transakční továrna

Transakční továrna *Transaction Factory* implementuje rozhraní `TransactionFactory` a vytváří transakční objekty, které jsou nižší urovní implementace předurčeny pro dobrou přenositelnost aplikace. Nejčastěji se jedná o *JDBC* a *JTA – Java Transaction API* transakce. Takto vytvořené objekty jsou dosažitelné díky výše popsaným objektům relace. Jen pro doplnění, tato část není povinná, ale je z důvodu jasnější funkčnosti doporučená.

4.1.3 Poskytovatel spojení

Poskytovatel spojení vytváří *JDBC* spojení s databází a organizuje je v množinách pro znovupoužití *Pool*. Pokud se tedy spojení jednou vytvoří a ukončí, tak se přidá do této množiny a později, při dalším požadavku na spojení se zkontrolují parametry a pokud odpovídají, tzn. je požadováno znovu stejné spojení, použije se toto spojení z množiny. To dělá aplikaci výkonější a šetrnější k systémovým prostředkům. Takto vytvořené množiny spojení jsou následně poskytovány továrně relace k používání.

4.1.4 Relace, transakce a dotaz

Pro zjištění základních operací nad databázemi jsou zde prvky relace, transakce a dotaz implementující rozhraní `Session`, `Transaction` a `Query`. Těmi operacemi jsou základní *CRUD – Create Read Update Delete* operace, čili vytváření, čtení, změna a mazání dat v databázi. Rozhraní dotaz zde slouží speciálně pro získávání dat z databáze.

4.1.5 Perzistentní objekty

V podstatě jimi mohou být běžné *Java* objekty implementující rozhraní `Serializable`, obsahující bezparametrický konstruktor. V dokumntaci je ještě doporučeno, aby tato třída obsahovala jeden atribut pro primární klíč. Další atributy této třídy se mapují jako sloupce databázové tabulky. Pro ilustraci třída z přiložené aplikace, reprezentující výrobní směnu.

```
public class Relay implements Comparable<Relay>, Serializable {

  private static final long serialVersionUID = -2765475613857275459L;

  private int id;
  private String relayName;
  private Set<Specimen> specimens = new HashSet<Specimen>();

  /** Zakladni bezparametricky konstruktor */
  public Relay() {
    super();
  }

  // get a set metody pro atributy
  ...

  public int compareTo(final Relay other) {
  return new CompareToBuilder().append(relayName, other.relayName)
    .toComparison();
  }

  @Override
  public boolean equals(final Object other) {
  if (!(other instanceof Relay))
    return false;
  Relay castOther = (Relay) other;
  return new EqualsBuilder().append(relayName, castOther.relayName)
    .isEquals();
  }

  @Override
  public int hashCode() {
  return new HashCodeBuilder(1309009611, -2071585405).append(relayName)
    .toHashCode();
  }

  @Override
  public String toString() {
  return relayName;
  }
}
```

Ukázka 4.1: Perzistentní třída

Pro tyto třídy je doporučeno, jak vidíme z ukázky, přepsat metody `equals`, `hashCode`, `toString` a implementovat metodu `compareTo` rozhraní `Comparable`<T>. To je z důvodů unikátnosti objektů a správnosti porovnávání dle uživatelem definovaných vlastností. Zde konkrétně vidíme že se bude porovnávat podle názvu výrobní směny a textová reprezentace objektu bude taktéž ve formě tohoto názvu. Pro vytváření těchto doporučených metod lze

s výhodou použít nástrojů skupiny *Apache Software Foundation* pro snadnější a jednotné vytváření výše uvedených metod. Navíc existují v *IDE* prostředích nástroje pro generování těchto metod, což práci ještě více usnadňuje a zrychluje.

4.1.6 Dočasné objekty

Jsou objekty, které se v aplikaci změnili, ale zatím se tyto změny nepřenesly do databáze.

4.2 Nastavení rámce Hibernate pro použití

Jelikož používáme *Hibernate* spolu s rámcem *Spring*, je předáno nastavení na něj. Pokud bychom ho používali samostatně, konfigurovali bychom ho taktéž samostatně a přímo.

V aplikačním kontextu musíme nakonfigurovat několik důležitých *JavaBean* objektů. S výhodou můžeme opět využít modularitu a tato nastavení umístit do samostatného souboru. V přiložené aplikaci se jmenuje `applicationContext-hibernate.xml`. Jako první to bude výše zmiňovaná továrna relace.

```
<!-- Tovarna relace -->
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
annotation.AnnotationSessionFactoryBean">
  <!-- Umisteni konfigurace mapovani -->
  <property name="configLocation" value="classpath:/hibernate.cfg.xml" />
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      <prop key="hibernate.cache.use_second_level_cache">true</prop>
      <prop key="hibernate.cache.provider_class">org.hibernate.cache.
      EhCacheProvider</prop>
    </props>
  </property>
  <property name="dataSource" ref="dataSource" />
</bean>
```

Ukázka 4.2: Deklarace továrny relace.

V našem případě deklarujeme jako továrnu realce implementaci, která používá anotace k určení mapování objektů do databáze. Pokud bychom chtěli využít starší, a v některých případech bohatší[1], způsob mapování pomocí *XML* souborů, museli bychom uvést třídu `LocalSessionFactoryBean`.U ní se nastavuje název a umístění mapovacích deklarací, dialekt databáze (proměnná odkazující na `database.properties` soubor), vyrovnávací paměť a konečně zdroj dat. Ten je dalším objektem, který se musí objevit v deklaraci aplikačního kontextu.

[1]Jedním z omezení anotovaného mapování je, že nelze mapovat rozhraní. To se dá obejít použitím abstraktních tříd.

```
<!-- Zdroje dat z~databaze -->
<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>
```

Ukázka 4.3: Deklarace datového zdroje

Zde se nastavují parametry *JDBC* připojení k databázi. Tuto komponentu bude reprezentovat třída `BasicDataSource`, která tvoří základ datových zdrojů. Aby bylo nastavení úplné je nutné ještě publikovat obsah souboru `database.properties`, odkud se berou výše uvedené proměnné zapsané jako `${...}`.

```
hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
hibernate.connection.driver_class=org.postgresql.Driver
hibernate.connection.url=jdbc:postgresql:/localhost:5432/databaseZQ
hibernate.connection.username=zq
hibernate.connection.password=zq
```

Jména proměnných jsou voleny tak, aby byl jasný jejich význam, proto je nebudu více popisovat.

4.3 Nastavení mapování objektů

Aby byl *Hibernate* schopný převádět objekty na databázové záznamy a opačně, musí se nějak dozvědět, co kam přiřadit a jak. K tomu slouží definice mapování. Jak bylo výše vysvětleno, existují dva hlavní způsoby, jak toho dosáhnout. Prvním je nastavení mapování z *XML* souboru.

4.3.1 Mapování pomocí *XML* souboru

```
<hibernate-mapping>
  <class name="events.Event" table="EVENTS">
  <id name="id" column="EVENT_ID">
    <generator class="native"/>
  </id>
  <property name="date" type="timestamp" column="EVENT_DATE"/>
  <property name="title"/>
</class>
</hibernate-mapping>
```

Ukázka 4.4: Mapování pomocí *XML* souboru

Tento způsob je ale složitější a proto se budeme věnovat nastavení mapování pomocí anotací, což urychluje práci a zpřehledňuje celou aplikaci. Vývojáři rámce *Hibernate* tento způsob

dále zefektňují a propracovávají, jelikož se ukázal jako velmi kvalitní a výhodný.

4.3.2 Mapování pomocí anotací

Anotované mapování je definováno standardem *EJB 3.0/JPA – Enterprise JavaBeans/Java Persistence API*, ze kterého anotace v *Hibernate* vycházejí. Jedná se převážně o anotace z balíku `javax.persistence`. Vývojáři *Hibernate* nad rámec těchto anotací vyvinuli ještě další speciální a některé aleternativní, pro využití všech specialit poskytovaných rámcem.

Uváděné příklady jsou z přiložené aplikace a referenční dokumentace k rámci *Hibernate*[8].

Objekty mapované na tabulky

Hlavní komponentou, kterou musíme namapovat je třída, která v databázi reprezentuje doménu databázové tabulky. Její instance je pak jedním řádkem v této tabulce, neboli zkráceně záznamem. Proto při deklaraci třídy uvedeme anotaci `@Entity`, která říká, že tato třída bude perzistentní. Dále je vhodné použít anotaci `@Table`, která blíže určuje vlastnosti databázové tabulky, která bude odpovídat mapované třídě. Ta má několik nepoviných parametrů, mezi nimi je `name`, neboli jméno tabulky, dále `schema`, který představuje schema v databázovém serveru a nebo `uniqueConstraints`, jež říká které sloupce v tabulce musí splňovat integritní omezení na jedinečnost záznamu. Poslední jmenovanou vlastnost lze nastavovat také přímo u atributu třídy.

```
@Entity
@Table(name="tbl_sky", uniqueConstraints = {@UniqueConstraint(columnNames={"
month", "day"})}
public class Sky implements Serializable {
  ...
}
```

Ukázka 4.5: Anotace `@Entity` a `@Table`

Atributy

To by samozřejmě nestačilo, proto musíme rámci *Hibernate* ještě sdělit, jak převádět atributy třídy. Pokud neuvedeme u *getter* metod příslušných danným atributům žádnou anotaci, pokusí se rámec *Hibernate* spojit tento atribut se sloupcem, který má stejné jméno. Z těchto důvodů se důraně doporučuje anotace explicitně definovat pro každý atribut. Na výběr máme z několika možností jak to udělat.

`@Basic` Základní anotace atribut-sloupec. Je vyžitelná pro deklaraci způsobu načítáni atributu z databáze horlivě (implicitní) vs. líně (načte se až je atribut potřeba). Lze ji použít na všechny typy, které implementují rozhraní `Serializable`.

`@Temporal` Jediný význam této anotace je že deklaruje dočasné uchování dat u atributu typu `Date`, `Time` nebo `Timestamp`.

`@Version` Takto označený atribut může být využíván pro optimistické zamykání při víceuživatelském přístupu, nebo aktualizaci objektu. Atribut by měl být číselného typu.

`@Lob` Využitelné pro velké objekty typu `Blob` nebo `Clob`, známé z *Oracle* databází.

`@Column` Nejčastěji používaná univerzální anotace atributu. Popisuje sloupec databáze a má hodně nepoviných (rozšiřujících) atributů.

- `name="columnName"` – jméno sloupce
- `unique` – unikátnost hodnot (implicitně vypnuto)
- `nullable` – možnost použití hodnoty `NULL` (implicitně vypnuto)
- `insertable` – povolení vytváření nových hodnot (implicitně zapnuto)
- `updatable` – povolení změny hodnot (implicitně zapnuto)
- `columnDefinition=""` – popis pro generátor struktury tabulky
- `table=""` – jméno tabulky, do které atribut patří (implicitně je to tabulka v anotaci třídy)
- `length` – délka sloupce (implicitně 255)
- `precision` – počet desetinných míst u desetinných číselných hodnot (implicitně 0)
- `scale` – měřítko desetinných čísel (implicitně 0)

`@Id` Anotce používaná pro určení primárního klíče tabulky. Je sice bezparametrická, ale používá se zároveň s dalšími anotacemi pro určení bližžších informací o primárním klíči, např. o způsobu generování tohoto klíče. Těmito anotacemi jsou `@GeneratedValue (strategy = GenerationType.AUTO)`, která říká že *Hibernate* vybere nejvhodnější strategii generování nové unikátní hodnoty dle typu databáze. V příkladu je znázorněno nastavení tohoto generování speciálně pro databáze *PostgreSQL*, ve které se používají pojmenované sekvence generovaných čísel pro každou tabulku zvlášť nebo společně pro všechny. Tuto variantu dostaneme při nastaveném automatickém výběru.

```
...
@Id
@SequenceGenerator(name = "test_types_id_seq", sequenceName = "
test_types_id_seq")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "
test_types_id_seq")
public int getId() {
return id;
 }
...
@Column(name = "test_type_name", length = 50, nullable = false, unique =
true)
public String getTestTypeName() {
return testTypeName;
 }
...
```

Ukázka 4.6: Příklad anotace perzistentního objektu

Pokud potřebujeme dosáhnout toho aby se daný atribut nemapoval, můžeme použít u deklarace atributu klíčové slovo `static` nebo `transient`. Další možnost je anotovat příslušnou `get...` metodu anotací `@Transient`.

```
@Entity
@Table(name = "users")
public class User implements Comparable<User>, Serializable, UserDetails {

  private transient String confirmPassword;
  private static boolean enabled;
  ...

  @Transient
  @Override
  public boolean isAccountNonLocked() {
  return enabled;
  }
}
```

Ukázka 4.7: Anotace atributů třídy

Závislosti mezi objekty

Dalším důležitým nastavením jsou závislosti mezi perzistetními objekty. Jedná se o integritní omezení databázových tabulek, kdy jedna tabulka odkazuje na druhou a naopak pomocí primárního a cizího klíče. *Hibernate* při mapování takovéhoto vztahu používá obousměrnou asociaci, čili závislé objekty si drží odkazy na sebe vzájemně. Je to z důvodu jednoduššího řešení některých operací.

Při mapování mohou nastávat prakticky dva druhy závislostí:

- 1:1 `@OneToOne`, neboli jedna ku jedné.
- M:1 `@ManyToOne`, neboli mnoho ku jedné.

V prvním případě jde o závislost, kdy jeden objekt odkazuje právě na jeden další objekt v jednom atributu a naopak. Toto jde řešit třemi způsoby

1. Odkaz jednoho objektu na druhý pomocí cizího klíče. Parametr `name` anotace `@JoinColumn` určuje sloupec s cizím klíčem.
2. Pomocí primárního klíče řídícího objektu. `@PrimaryKey JoinColumn` deklaruje asociaci shodným primárním klíčem závislých objektů.
3. Spojovací tabulkou anotací `@JoinTable`, používanou pouze není-li možnost měnit tabulky v databázi (např. při předělávání existujícího systému).

V druhém případě může objekt odkazovat na více objektů daného typu, tak že si objekt s kardinalitou *many* (více) drží množinu odkazů na odkazující pomocí anotace `@OneToMany` s parametrem `targetEntity` a `mappedBy`.

Mapování kolekcí

Pro vyčerpání všech způsobů závislostí v databázích ještě schází mapování kolekcí, které tvoří protějšek výše uvedeného vztahu M:1 a poslední druh, kterým je závislost M:M. Tyto závislosti se mapují do standardních *Java* kontejnerů, jako jsou `List<T>`, `Set<T>` a `map<T>`.

První uvedené mapování se anotuje jako `@OneToMany` a je na druhé straně závislosti mapované jako `@ManyToOne`. Může být jednostrané, kdy se používá spojení cizím klíčem nebo asociativní tabulkou, a obousměrné, které je doporučováno a proto používáno častěji.

U druhého scénáře se používá anotace `@ManyToMany` a `@JoinTable` pro určení názvu a parametrů spojovací tabulky.

Pro mapování závislostí existuje mnoho scénářů a nastavení, které je zbytečné uvádět v této práci, jelikož je nutné pochopit pouze základní styl a syntaxi pro práci s tímto rámcem. Více informací a nastavení je mozžno najít v referenční příručce[8], nebo na stránkách věnovaných přímo anotacím perzistentních objektů rámce *Hibernate* http://docs.jboss.org/hibernate/stable/annotations/reference/en/html.

4.4 Nastavení aplikace

Pro správnou funkčnost je ještě nutné říct aplikačnímu rámci *Spring*, jak a kde má hledat mapování objektů. U suborů *XML*, je jasně dána cesta v hlavním zaváděcím souboru `web.xml`. Pro anotované mapování, stačí rámci pouze sdělit, které třídy mají být mapovány. Oba druhy nastavení bývájí tradičně umístěné v souboru `hibernate.cfg.xml`. Protože v přiložené aplikaci jsou použity anotované perzistentní objekty, příklad ukazuje výše zmíněný soubor pro takovýto případ.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Mapovani trid hibernate anotacemi -->
<hibernate-configuration>
  <session-factory>
    <mapping class="cz.zpsv.databaseZQ.pojo.Authority" />
    <mapping class="cz.zpsv.databaseZQ.pojo.ConcreteClass" />
    <mapping class="cz.zpsv.databaseZQ.pojo.ConcreteEnvironment" />
    <mapping class="cz.zpsv.databaseZQ.pojo.ConsistenceType" />
    <mapping class="cz.zpsv.databaseZQ.pojo.Plant" />
    <mapping class="cz.zpsv.databaseZQ.pojo.Prescriptions" />
    <mapping class="cz.zpsv.databaseZQ.pojo.Relay" />
    <mapping class="cz.zpsv.databaseZQ.pojo.Specimen" />
    <mapping class="cz.zpsv.databaseZQ.pojo.TestType" />
    <mapping class="cz.zpsv.databaseZQ.pojo.User" />
  </session-factory>
</hibernate-configuration>
```

Ukázka 4.8: Nastavení anotovaných perzistentních objektů.

Je zde prakticky pouze deklarováno, v kterých balících a jaké třídy má rámec hledat. Po nalezení těchto tříd rámec sám zaregistruje do aplikačního kontextu potřebné mapování pro další použití.

Zhodnocení: Nástroj popsaný v této kapitole, přináší programátorovi *Java EE* aplikací výrazné zrychlení práce a malé výdaje. Díky několika způsobům nastavení, lze říci že se hodí pro všechny případy, kdy je potřeba efektivní a transparentní přístup do databáze.

Navíc k rámci *Hibernate* existují další podpůrné nástroje, které dokáží kupříkladu vygenerovat z nastavení mapování konkrétní schéma databáze, podle zvolého dialektu a naopak konvertovat databízi na *Java* třídy s potřebnými nastaveními.

Část III

Praktická část

Kapitola 5

Analýza a návrh aplikace

Cíle:

- Ukázat postup vznikání aplikace v praxi.

Pro ukázku použití těchto technologií budeme uvažovat projekt zadaný zákazníkem, který musí splňovat definované parametry a musí být použitelný na stávajících technologiích zákazníka.

Tento projekt vychází z mé dosavadní praxe ve stavebnictví, kde je potřeba sledovat určité ukazatele výroby a provádět statistiku a sledovat odchylky v kvalitě výrobků. Jako zákazník zde bude vystupovat smyšlená firma *PrefaXY*, která vyrábí betonové výrobky a je certifikována pro normu ISO 9001 *Management kvality*, což ji zavazuje dodržovat českou normu ČSN EN 206-1[1], která pojednává o betonu a požadavcích na něj.

Následující analýza a návrh je komplexnější, než je implementovaná aplikace, protože pro předvedení popisovaných technologií to bezpečně dostačuje.

5.1 Analýza

Cíle:

5.1.1 Základní údaje o projektu

Zákazník, firma PrefaXY požaduje vytvořit samostatný informační systém typu klient–server s víceuživatelským přístupem, který bude běžet na centrálním serveru v sídle společnosti a uživatelé se k tomuto systému budou připojovat pomocí *tenkého* webového klienta.

Systém bude schopný sbírat blíže specifikovaná výrobní data, která vznikají na různých pobočkách této společnosti, umístěných různě po celé České republice a tyto data statisticky vyhodnocovat podle ČSN EN 206-1, což je primární cíl aplikace. Tyto data se budou ukládat na centrálním úložišti v databázi *PostgreSQL*, ale je nutné uvažovat případnou změnu poskytovatele databáze, například *Oracle*, nebo *MS SQL Server*.

Data do systému zadávají pracovníci technické kontroly, kteří k tomuto budou řádně proškoleni a prozkoušeni z používání systému. Jedná se spíše o běžné uživatele informačních technologií na úrovni kancelářských aplikací typu MS Word a MS Excel, proto bude nutné tomuto stavu přizpůsobit uživatelskou dokumentaci a nápovědu.

Jako jednotku dat budeme uvažovat jeden vzorek, což bude ve většině případů betonová krychle o straně 150 mm, ale systém musí být schopen přijmout i data o vzorku jiných rozměrů. Jelikož norma ČSN EN 206-1 mluví pouze o statistice krychelné pevnosti, bude se vždy jednat o tělesa ve tvaru krychle. To aplikaci značně zjednodušuje, ale nelze vyloučit, že zákazník bude v budoucnu požadovat rozšíření o jiné tvary vzorků.

Systém bude dále schopen tyto nasbírané data, dle požadavků obsluhy, exportovat do tabulek formátů `*.pdf`, `*.xml`, `*.html`, popřípadě dalších. Dále musí umožňovat přidávání hodnot parametrů zkoušených vzorků, jako je třeba třída betonu a jeho prostředí, nebo číslo receptury.

5.1.2 Neformální analýza systému

Nejdůležitějším modulem systému bude statistické vyhodnocení nasbíraných výrobních dat, který bude umožňovat zadání rozsahu datumů vyhodnocované části dat, číslo receptury, typ zkoušky, druh vyhodnocení a jiné. Výstupem bude protokol, obsahující hlavičku, vyhodnocení dat a zápatí, kde budou náležitosti popsané ve výše uvedené normě. Zde je nutné uvažovat dva typy dat, a sice počáteční data z výroby a data z průběžné výroby. Přesné definice a četnost získávání dat jsou podrobně popsány ve výše zmíněné normě.

Požadované prvky systému a jejich vlastnosti jsou shrnuty v následující tabulce:

	VLASTNOSTI	
PRVEK	**ZADÁVANÉ**	**VYBÍRANÉ**
Vzorek	datum výroby, datum zkoušky, obsah vzduchu, protokol CHRL, hodnota CHRL, rozměry, hmotnost, protokol HP, hodnota HP, pařený beton?, specifikovaný V/C, skutečný V/C, objemová hmotnost ČB, zatížení, konzistence	typ konzistence, receptura, typ zkoušky, výrobní směna, výrobna
Výrobna	název, číslo	
Typ konzistence	název, jednotky rozměru	
Typ zkoušky	název	
Receptura	číslo	třída betonu
Třída betonu	název, charakteristická pevnost	stupeň prostředí
Stupeň prostředí	název	
Výrobní směna	název	
Uživatel	jméno, příjmení, email, heslo, aktivní?	

Tabulka 5.1: Přehled vlastností prvků systému.

Dále musí mít systém správu uživatelů, s kategorizovanými právy, pro různou úroveň řízení přístupu do systému. Bude zde hlavní administrátor, jeden nebo více, kteří budou mít práva přidávat a modifikovat uživatele.

U prvku vzorek musí být ze systému dohledatelné, kdo a kdy ho vytvořil a poslední uživatel, který ho modifikoval a v jaký datum a čas. Nejlepší by bylo, kdyby se tyto údaje zobrazovali při zadávání vzorku. Vzorek může přidávat jakýkoliv uživatel, ale modifikovat

ho může pouze administrátor, nebo jeho původní zadavatel. Mazat vzorky může pouze administrátor, kvůli bezpečnosti a důvěryhodnosti dat. Tímto bude některý vedoucí pracovník, který rozhodne zda vzorek smazat, nebo ne.

Systém by měl mít jednoduché uživatelské rozhraní, které bude hlídat, zda jsou vkládané hodnoty v pořádku, či nikoliv a v negativním případě na to vhodným způsobem upozorňovat. Měla by zde být dosažitelná nápověda, která vhodně pomůže uživateli s ovládáním aplikace.

5.1.3 Hardwarové technologie

Jelikož zákazník v současné době používá UNIX servery, které mají dostatečnou rezervu výkonu, budou pro běh aplikace použity tyto servery a datová úložiště. Zákazník také používá databázový systém PostgreSQL, proto se pro databáze využije rovněž tento systém. Zákazník do budoucna uvažuje o rozšíření svých vnitřních systémů, proto musí být aplikace jednoduše upravitelná pro použití v *SOA (Servisně orientovaná architektura)* technologiích.

Uživatelé používají na svých stanicích systémy Windows, na což se musí brát zřetel z hlediska kódování. K hlavnímu centrálnímu úložišti dat se uživatelé připojují transparentně pomocí hardwarových VPN tunelů. Tyto jsou navíc zajištěny firewallem.

5.2 Návrh

Celý systém bude vytvořen a zkonstruován jako třívrstvá webová aplikace typu klient–server. Datová a aplikační vrstva poběží na serveru v centrále zákazníka, klienty budou webové prohlížeče na uživatelských stanicích.

K serveru bude mít přístup pouze proškolený a poučený administrátor, aby se zaručila bezpečnost a integrita dat. Do databáze bude mít přístup pouze servisní technik výrobce a to pouze z vnitřní sítě zákazníka.

Aplikace musí být schopna provádět všechny funkce specifikované v diagramu případů užití, musí splňovat bezpečnostní podmínky a musí být stabilní a spolehlivá.

5.2.1 Implementační technologie

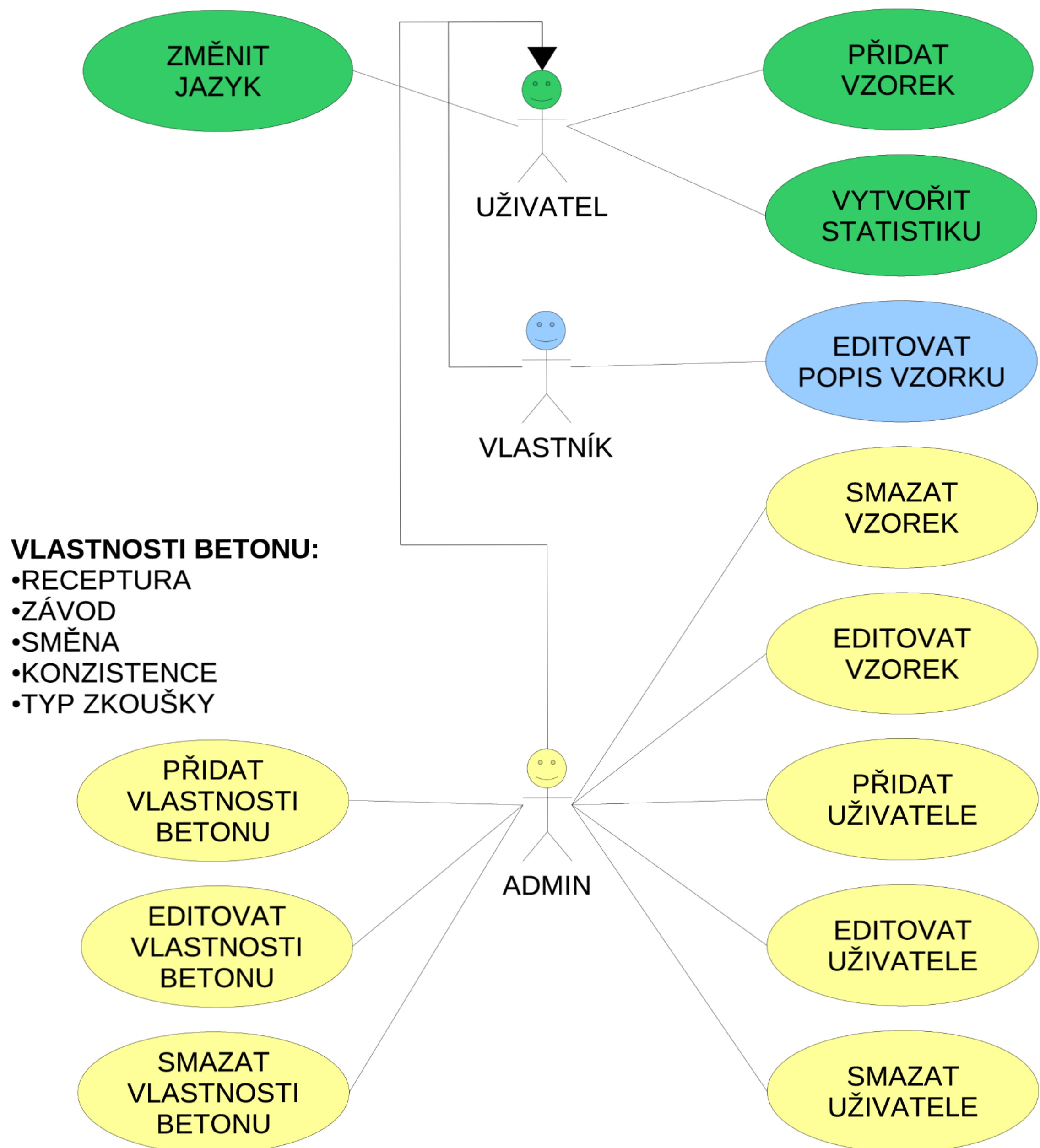
K vůli jednoduchosti a rychlosti řešení, které požaduje zákazník, bude použita technologie Java, přesněji J2EE. Jako aplikační rámec bude použit *Spring* a rámec pro perzistenci dat *Hibernate*. Tyto technologie zaručují přesný, rychlý a spolehlivý model informačního systému, přesně podle požadavků zákazníka.

Jako aplikační server bude použit Tomcat, jelikož patří do skupiny odlehčených aplikačních serverů, což v současné situaci zákazníkovi dostačuje a vyhovuje. Pokud by vznikla potřeba tento systém doplnit a překoncipovat na servisně orientovanou architekturu, bude velmi snadno vyměněn za jiný aplikační server, např. JBoss, nebo WebLogic.

Jak bylo zmíněno v analýze 5.1.1 zákazník používá databázový systém *PostgreSQL*, proto bude prověřen a posléze využit k perzistenci dat i pro tento projekt.

5.2.2 Případy užití

Případy užití se skládají z několika jednoduchých akcí, rozdělených podle role uživatele, které musí být v systému vytvořeny a nebudou editovatelné. Podrobněji v diagramu případů užití.



Obrázek 5.1: Diagram případů užití.

Vlastnosti betonu se skládají z několika položek, pro které platí stejné akce jako pro ostatní prvky, tj.

- Přidání.
- Editace.
- Smazání.

5.2.3 Aplikační a prezentační vrstva

Aplikační vrstva se bude skládat z několika modulů, které budou vzájemně provázány. Nejdůležitější část aplikace je statistika, ta bude proto v samostatném modulu.

K zabezpečení aplikace budou využity prostředky rámce *Spring Security* a zákazníkem definovaná politika práv uživatelů. Systém může mít mnoho uživatelů a několik správců – administrátorů, kteří budou zodpovědní za celý systém.

Prezentační vrstva bude vytvořena technologií *MVC – Modul-View-Controller* za použití prezentační části rámce *Spring*. Grafika aplikace bude dodána externí dodavatelskou firmou předem ve statickém provedení. Návrhy vzhledu si obstarává zákazník sám ve spolupráci se zhotovitelem. Centrála zákazníka má v současnosti plnou konektivitu s pracovišti, proto se využije stávající síť a systém zabezpečení.

Aplikace musí být odladěna pouze pro prohlížeč *Mozilla Firefox*, ale je vhodné pracovat tak, aby vyhověla *W3 – World Wide Web Consorcium* standardu.

5.2.4 Návrh databáze

Databázové schema aplikace bude vytvořeno s ohledem na další možnosti rozšíření této aplikace a bude dodržovat několik důležitých zásad, mimo běžné zásady databázových úložišť, a to:

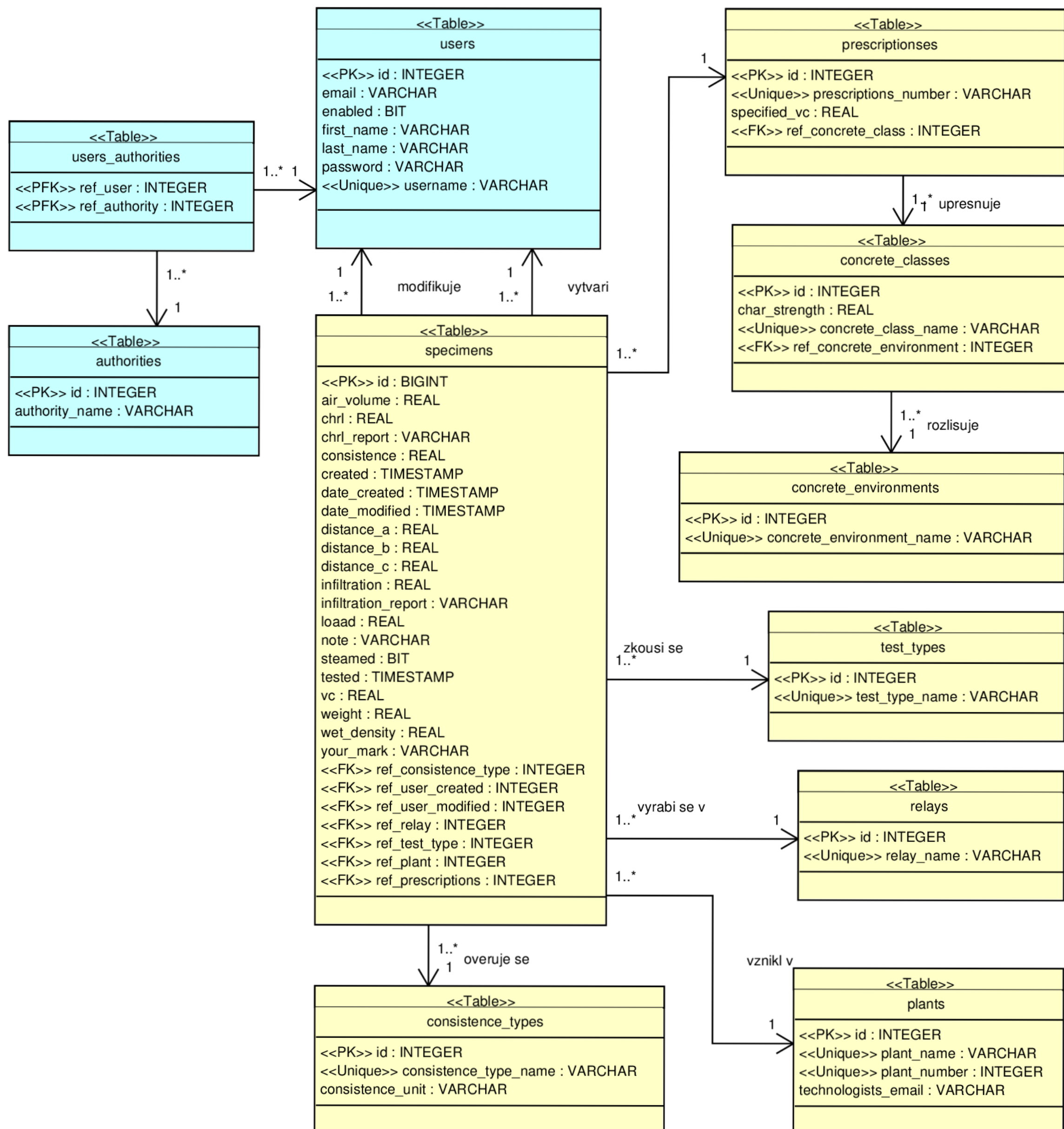
- Primární klíč každé relace (tabulky) bude tvořen celým číslem a to přesně typem `Long` nebo `Integer`.
- Primární klíč nebude naturální hodnotou relace. Tj. bude mít pouze význam číselného identifikátoru.
- Cizí klíče ponesou jednotný prefix `ref_` aby byly jednoduše identifikovatelné

Datová vrstva aplikace bude implementována za pomocí rámce *Hibernate* s propojením na aplikační vrstvu viz. 5.2.3. Bude nutné vytvořit skript pro definici databázových tabulek. K této činnosti se využije nástroj *Hibernate Tools*, který vytváří strukturu databáze přímo z anotovaných tříd aplikace a dovede ho přímo aplikovat na zvolený databázový stroj.

Je nutné dodržet co největší jednoduchost a plnou modularitu, pro pozdější snadné úpravy aplikace.

Zhodnocení: Tato kapitola ukázala, jak důležitá je podrobná analýza u zákazníka. Uděláme-li slabou analýzu, budeme se o to více museti ptát zákazníka během práce, což není ideální.

Výše popisovaná fáze vývoje projektu se řídí známými metodologiemi, které nejsou předmětem této práce, proto se jim kapitola nevěnuje podrobně. Zde měla sloužit pro představu, co celý projekt zahrnuje.



Obrázek 5.2: ER-diagram databáze

Kapitola 6

Implementace a testování

Cíle:

- Ukázat jak se popisovaná aplikace implementuje
- Přiblížit nutnost testování během vývoje.

Hlavním problémem při implementaci takovéhoto složitého systému je přesné plánování práce, protože se určitě budeme muset zabývat množstvím problémů a detailů. Důležitou součástí práce je mít kvalitní software a zařízení, na kterém budeme systém vyvíjet. Je taky docela dobré dodržovat přesně stanovené zásady a návyky, jako pravidelné ukládání práce do repozitářů a zálohovaní již vytvořené aplikace.

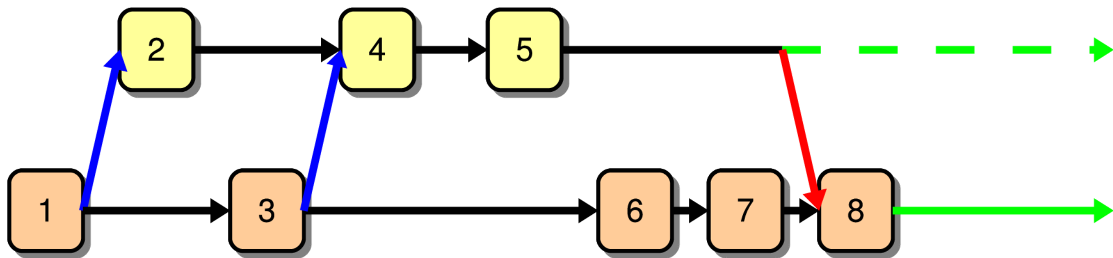
6.1 Implementace

Při implementaci jsem vycházel z vytvořeného návrhu z kapitoly 5.2, který je ovšem celkový a komplexní. Z tohoto důvodu se hodně věcí opakuje a mělo by být implementováno stejným stylem. Proto není implementace kompletní, ale zvolil jsem ji tak, aby ukázala všechny důležité části této práce na praktickém příkladě. Veškeré manipulace s objekty a pohledy demonstruje objekt `Specimen` a funkčnost jeho se týkající. Bezpečnost je demonstrována na správě uživatelů a taktéž závislosti objektů.

S vývojem aplikace jsem postupoval tak, že jsem si stanovil milníky, které jsem se v průběhu práce snažil dodržovat. Pokud jsem narazil na nějaký problém, vše jsem si pečlivě zapsal a využil výhod verzovacího systému *SVN*, který dokáže vytvořit novou „větev“ (*branch*) aplikace, kde můžeme problém nějakým způsobem řešit a pokud se nám to povede, sloučíme tuto větev s hlavní vývojovou větví *trunk* aplikace. V případě, že se rozhodneme danou situaci řešit jiným způsobem, lze se pohodlně vrátit k předchozí práci před vznikem nové „větve“ a pokračovat jinak. Ještě máme možnost použít pevné zarážky tzv. *tag* pro statickou zálohu některé z větví. Názorně to lze vidět na obrázku 6.1.

Černé šipky tady představují normální změnu kódu, modré zase rozvětvení nebo spojení hlavní větve do vedlejší vývojové větve. Červená šipka pak znamená sloučení do hlavní větvě. Zelené šipky úplně vpravo ukazují možnost vydávání různých úrovní aplikace, jako je třeba *SNAPSHOOT* nebo stabilní verze.

Čísla vyjadřují verze ukládaného kódu. Jak je patrné je toto číslovaní pro lepší orientaci inkrementováno s každou změnou v repozitáři.

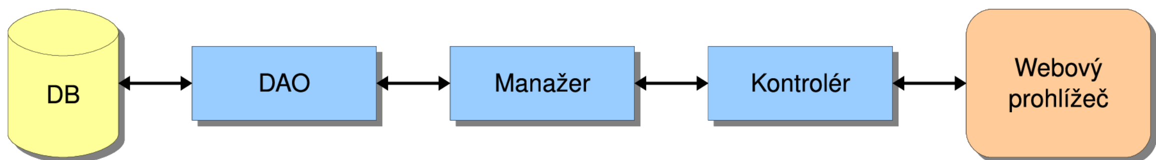


Obrázek 6.1: Schéma funkce SVN repozitáře.

6.1.1 Rozbor zadání

Před vlastní implementací je důležité promyslet pořádně základy aplikace, protože se tím vyhneme problémům s přepisováním špatně znovupoužitelného kódu.

U třívrstvých aplikací je důležité oddělit od sebe jednotlivé vrstvy tak aby na sobě byly co nejméně závislé a zároveň aby jejich provázanost byla co nejvíce univerzální viz. obrázek 6.2. Já jsem se rozhodl využít genericity, kterou jazyk *Java* poskytuje.



Obrázek 6.2: Schéma třívrstvé webové aplikace.

DB je databázový stroj.

DAO *Data Access Object* je objekt, pomocí něhož se přistupuje do databáze.

Manažer je *business* objekt aplikační vrstvy, který řídí přístup do databáze skrze *DAO* objekt.

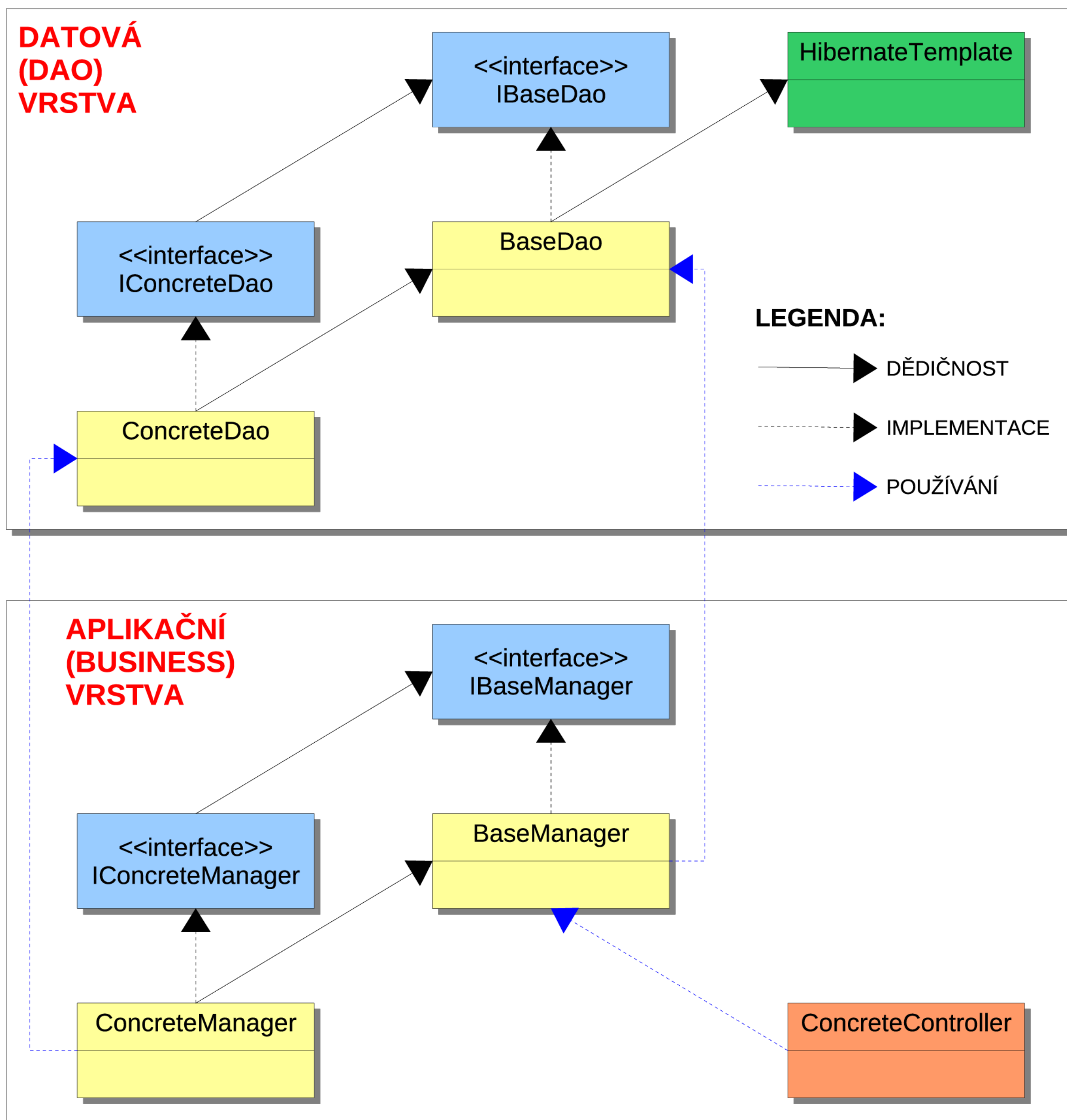
Kontrolér je objekt který definuje, jaké prvky budou ve výsledném pohledu a získává je přes manažerské objekty.

Při požadavku aplikace o data je postup následující:

1. Uživatel ve webovém prohlížeči požádá o zobrazení stránky s daty.
2. Webový prohlížeč vznese požadavek do aplikace na objekt *Controller* (kontroler).
3. Kontroler si řekne o data manažerovi a čeká až mu je manažer vrátí.
4. Manažer po požadavku o data tento přenese na *DAO* a čeká.
5. *DAO* přistoupí do databáze a výsledek předá manažerovi.
6. Manažer předá výsledek do kontroléru a ten ho zobrazí na stránce.

Díky této skladbě jsou vrstvy mezi sebou vzájemně nezávislé na samotné implementaci a lze je „jednoduše" vyměnit. Například je častá výměna implementace *DAO* vrstvy. (Implementce *Hibernate* se vymění například za implementaci v *IBatis*).

Po analýze zadání bylo jasné, že většina objektů bude používat stejné metody pro přístup do databáze a pro získávání dat z databáze. Mimoto se nabízely ještě další společné rysy. Proto jsem se rozhodl v datové i aplikační vrstvě vytvořit *Base – základní* objekty, které budou zapouzdřovat společnou funkcionalitu a ostatní objekty budou od těchto zděděny a předají si přesný typ objektu, pro který budou fungovat. Genericita se pak už postará o vracení správných typů objektů. Použil jsem samozřejmě techniku programování do rozhraní, kvůli přehlednosti a testovatelnosti aplikace. Výsledné základní schema je na obrázku 6.3. Z tohoto obrázku je jasné jak se budou objekty vytvářet. Podobné metody budou definovány v `Base...` objektech a speciální v konkrétních objektech.



Obrázek 6.3: Základní schema datové a aplikační vrstvy.

6.1.2 Vlastní vytváření aplikace

Při vlastním psaní aplikace jsem postupoval od pohledu do databáze, tedy zprava doleva z pozice obrázku 6.2.

1. Nejprve jsem vytvořil statickou stránku v prohlížeči za pomocí jazyka *HTML* a uživatelského *JavaScriptu*, která bude jako šablona pro generovaný dynamický pohled.
2. Vytvořil jsem třídu popisující objekt kontroléru, např. `SpecimensListController`. (Používal jsem anotované kontroléry)
3. Vytvořenou třídu kontroléru jsem zapsal do *XML* souboru definující aplikační kontext.
4. *HTML* prvky v šabloně stránky jsem upravil všechny prvky, které měli být dynamické za pomocí *JSP – Java Server Pages* značek.
5. Dále jsem napsal všechny konkrétní manažery aplikační vrstvy, přes které jsem potřeboval přistupovat k perzistentním datům a jejich metody. Pro ilustraci např.: `SpecimenManagerImpl` a `PlantManagerImpl`.
6. Na tyto manažery jsem navázal konkrétní *DAO – Data Access Object* objekty pro přístup do databáze a metody k tomu potřebné. Pro tento případ to budou: `SpecimenDaoHibernate` a `PlantDaoHibernate`.
7. Nakonec jsem k takto vytvořenému řetězu objektů napsal základní jednotkové testy k ověření správné funkčnosti.

Po provedení tohoto koloběhu pro všechny požadované stránky systému jsem měl vytvořenou funkční kostru aplikace, která byla schopná zobrazovat záznamy z databáze a ukládat do ní záznamy nové.

6.1.3 Problémy při implementaci

Při vytváření seznamů záznamů jsem se setkal s malým problémem, který se musí řešit u každé aplikace, která spravuje větší objemy dat. Tím problémem je stránkování při výpisu záznamů.

Modelový případ je pokud je v systému mnoho dat a my je potřebujeme procházet. Nastává několik možností řešení:

1. Pokud by se data načítala po jednom záznamu, bylo by to nepohodlné a neefektivní.
2. Jakmile se pokusíme načíst data všechny, můžeme se dostat do problémů, protože operační paměť není nekonečná.
3. Ideální je načítat data z databáze po přesně definovaném počtu a držet si odkazy na poslední a první načtený záznam.

Třetí případ je samozřejmě řešení tohoto problému a pokud využijeme všech možností, bude výsledek dobře fungovat, dobře vypadat a bud se sním výhodně pracovat.

Já jsem podle hesla nevynalézat kolo vyzkoušel dva možné rámce, které se pro tuto problematiku nabízely. Prvním z nich byl rámec *HDPagination – High Data Pagination*(http://www.hdpagination.org) a druhý *ValueList* http://valuelist.sourceforge.net. Oba tyto rámce implementují návrhový vzor definovaný pro *J2EE* aplikace *Value List Handler*

firmou *Sun Microsystems Inc.* (Definice je dostupná na stránkách http://java.sun.com/blueprints/corej2eepatterns/Patterns/ValueListHandler.html).

Rámec *Spring* sice poskytuje techniku pro stránkování záznamů, ale funguje tak, že nejprve načte všechny data do paměti a ty potom rozstránkuje Toto řešení je vhodné pro malé objemy dat, proto se pro tento projekt nehodilo.

Pro otestování obou těchto rámců jsem použil metodu popsanou v 6.1. První rámec *HDPaginate* se zpočátku tvářil velmi dobře, ale nakonec mě nepřesvědčil o svých schopnostech, proto jsem sáhl k dalšímu rámci. Tento už byl o poznání propracovanější a fungoval bezchybně. Velikou výhodou pro něj je, že umí data jednoduše formátovat do tabulek za použití *JSP* značek a poskytuje mnoho rozšíření vhodných pro zobrazování větších objemů dat.

Teď bylo na řadě doplnit funkčnost pro úpravy, mazání záznamů, podle požadavků zadání.

6.2 Testování

Testování patří k vývoji softwaru, jelikož dává programátorovi jakousi jistotu, že to co vytváří funguje podle očekávání. U rozsáhlých podnikových aplikací je to ještě o to rapidnější, jelikož je zde dost vysoká pravděpodobnost, že zákazník projeví zájem o rozšíření aplikace.

Proto platí: „Čím více testů, tím lépe.“ Samozřejmě že musí být tyto testy smysluplné a o něčem vypovídající, jinak by se přidaná hodnota, která testováním vzniká, znehodnotila. Z toho důvodu vnikla celá řada metodik testování a programování, které se tím zabývají.

XP programming neboli extrémní programování udává jako základ vývoje testy, které se spouští při každé významější změně kódu.

TDD – Test Driven Development je programování řízené testy. Nejprve se vytváří sada smysluplných testů a posléze se podle nich vytváří samotná aplikace.

U velkých webovýach a portálových podnikových aplikací se používají v podstatě tři druhy testovacích technik.

Jednotkové testování. Ověřování samostatných částí zdrojového kódu. Je to běžná technika u mnoha programovacích jazyků.

Integrační testování. Prověřování správného volání metod a funkčnosti mezi závislými komponentami.

Akcecptační testování. Celkové ověření aplikace, zda splňuje to co zákazník požadoval a zaplatil.

6.2.1 Jednotkové testování

Pro jednotkové testy se *Java* aplikacích nejvíce používá knihovna `JUnit`, díky které se dají pohodlně vytvářet a nastavovat různé druhy testů. Tato technika je dostatečně známá a proto se budeme raději věnovat dalším technikám.

6.2.2 Integrační testování

Tato technika nám dovoluje testovat volání metod mezi komponentami a ověřit tak správnou návaznost. Pokud vytvoříme sadu dobrých a odpovídajících testů, pomůže nám to k „doladění“ struktury aplikace. Ovšem je nutné striktně dodržovat programování do rozhraní, kdy všechny objekty implementují známá rozhraní. Toto hlídá i aplikační rámec *Spring*. Nejen proto se integrační testování hodí výborně pro vývoj aplikací postavených nad tímto rámcem.

Při tomto způsobu testování se používá techniky tzv. *mockování* objektů. V podstatě jde o nahrazení známeho rozhraní „podtrčeným“ objektem, který se tváří jako implememtace tohoto rozhraní. Potom v testu nastavíme u tohoto *mock* objektu očekávání volání metod a vyvoláme akci, která by tuto metodu měla používat.

Jako výsledek dostaneme buďto správný průchod testem, nebo definice míst, kde test selhal. To je další výhoda této techniky, jelikož nám pomáhá přesně odhalovat místo vzniku chyby. Pro lepší integrovatelnost se často používá knihovna `jMock`.

Používání takovýchto technik vede k lepší struktuře kódu a efektivnějšímu stylu kódování.

Techniku *mockování* objektů používá k testování i přiložená aplikace, proto malá ukázka nastavení testu.

```
@RunWith(JMock.class)
public class SpecimenManagerImplTest {

  private Mockery context;
  private ISpecimenDao specimenDao;
  private SpecimenManagerImpl specimenManager;

  // Nastaveni objektu pro testovani.
  @Before
  public void setUp() throws Exception {
  context = new JUnit4Mockery();
  specimenDao = context.mock(ISpecimenDao.class);
  specimenManager = new SpecimenManagerImpl();
  specimenManager.setSpecimenDao(specimenDao);
  }

   // Test pro metody
  @Test
  public void testGetByStatisticCriteria() {

  final StatisticCriteria criteria = new StatisticCriteria();

  // nastaveni ocekavanych reakci
  context.checking(new Expectations() {
    {oneOf(specimenDao).getByStatisticCriteria(criteria);}
  });

  // provedeni testu
  specimenManager.getByStatisticCriteria(criteria);
  }
}
```

Ukázka 6.1: Test pomocí *mock* objektu

6.2.3 Akceptační testování

Je to vývojový proces, kdy se aplikace ukáže buďto jako připravená pro nasazení do produkce, nebo ne. Jde vlastně o seznam případů užití s nadefinovanými způsoby reakcí na tyto případy. Pro webové aplikace se může použití tohoto druhu testu zdát jako test webové „prezentační" vrstvy, ale ve skutečnosti tomu tak zdaleka není. Testuje se celá aplikace, jelikož jsou všechny tři vrstvy propojeny. Pro programátorsky přívětivější použití se tento druh testování implemetuje za použití podpůrných knihoven, jako může býr např. `JWebUnit`.

Zhodnocení: Při implemantaci a testování aplikace se v praxi používá celá řada pomocných technik a knihoven. Proto je při vývoji velkých aplikací efektivní, pokud se testování věnuje zvláštní tým, jež vytváří automatizované testy.

Část IV

Zhodnocení

Kapitola 7

Závěr

Tato práce shrnula stručným způsobem použití moderních technologií, které se využívají při vytváření složitých a rozsáhlých podnikových aplikací.

Jelikož se v dnešní době velkým firmám jeví jako nutnost mít informační systém, protože dobře chápou jeho přínos co do efektivity práce, stává se velkým trendem tyto aplikace vytvářet. Na trhu bychom našli hodně firem, které nabízí zhotovení takové aplikace na klíč, přičemž jde většinou o upravený, již vyvinutý a ověřený, základ. To vede k zrychlení práce při vývoji a snížení ceny pro koncového zákazníka. Integrovatelnost s dalšími rámci a technologiemi zaručuje splnění jakýchkoliv zákazníkových přání.

A právě pro takovýto způsob tvorby aplikací jsou technologie popsané v této práci zcela ideální. Kdyby jsme udělali průzkum, kolik softwarových firem používá konkrétně tyto moderní rámce, dostali bychom se k vysokým číslům přesahujícím 40 %. Z tohoto trendu lze pochopit, že se v těchto moderních technologiích nalézá budoucnost informačních systémů.

V porovnání s dalšími technikami, kde konkuruje hlavně techologie*.Net* firmy *Microsoft*, najdeme *J2EE* aplikace v odvětvích, kde je kladen důraz hlavně na stabilitu a bezpečnost. Jedná se většinou o obrovské finanční instituce a státní úřady.

Pro podporu aplikačního rámce *Spring* vzniká mnoho aplikací, které se snaží vývojový proces, ještě zjednodušit a zpřehlednit. Tyto aplikace jsou ve většině případů *open-source*, což nahrává i menším firmám, které nemusí dávat velké finanční prostředky do nákupu vývojových technologií. Navíc probíhá ze strany vývojářů popisovaných technologií intenzivní výzkum a začleňování nových technik do aplikačních rámců. Obrazkem může být například rámec *Spring*, kdy v každé uvolněné verzi najdeme mnoho nových vylepšení a užitečných pomůcek. Dá se říci, že každá uvolněná vezre je takovou malou revolucí mezi aplikačními rámci.

Proto jsem si taky tyto technologie vybral pro svoji bakalářskou práci, neboť v nich spatřuji velkou budoucnost na poli informačních systémů a hodlám se jim ve spojení se *SOA – Service Oriented Architecture* technologiemi věnovat intenzivně dále.

Literatura

[1] *ČSN EN 206-1 se změnou Z3: Beton*. 2001.

[2] Alex, B.; Taylor, L.: *Spring Security: Reference Documentation*. [online]. Dostupné na URL: http://static.springframework.org/spring-security/site/reference/html/springsecurity.html.

[3] Harrop, R.; Machacek, J.: *Pro Spring*. Berkeley, CA 94705: Apress, první vydání, 2005, ISBN 1-59059-461-4, 978-1-59059-461-2.

[4] Johnson, R.: *Expert One-on-One: J2EE Design and Development*. Indianapolis, IN 46256: Wiley Publishing, Inc., první vydání, 2003, ISBN 0-7645-4385-7.

[5] Johnson, R.; et al.: *The Spring Framework:Reference Documentation*. [online]. [rev. 2008-11-03]. Dostupné na URL: http://static.springframework.org/spring/docs/2.5.x/reference.

[6] Různí autoři: *Wikipedie: Otevřená encyklopedie*. [online]. Dostupné na URL: http://wikipedia.org.

[7] Sun Microsystems, Inc.: *Sun Developer Network*. [online]. Dostupné na URL: http://java.sun.com/.

[8] The hibernate.org Team: *Hibernate Framework:Reference Documentation*. [online]. Dostupné na URL: http://www.hibernate.org.

[9] Walls, C.: *Spring in Action*. Greenwitch, CT 06890: Manning Publications Co., druhé vydání, 2008, ISBN 1-933988-13-4.

Dodatek A

Ukázky kompletních kódů

A.1 Anotovaný perzistentní POJO objekt User

```
// ===========================================================================
// Aplikace: databaseZQ
// Autor: Libor Ondrusek
// Spolecnost: FIT VUT Brno
// Rok: 2008
// Kodovani: UTF-8
// Soubor: User.java
// ===========================================================================

package cz.zpsv.databaseZQ.pojo;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;
import javax.persistence.OrderBy;
import javax.persistence.Table;
import javax.persistence.Transient;

import org.apache.commons.lang.builder.CompareToBuilder;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.springframework.security.GrantedAuthority;
import org.springframework.security.userdetails.UserDetails;

/**
 * Modeluje uzivatele systemu. (UserDetail)
 * @author Libor Ondrusek
 * @version 1.0.0
```

```java
 */

@Entity
@Table(name = "users")
public class User implements Comparable<User>, Serializable, UserDetails {

  private static final long serialVersionUID = 7871099877416233147L;

  private Set<Authority> authoritySet = new HashSet<Authority>();
  transient private String confirmPassword;
  private String email;
  private boolean enabled;
  // admin ma moje 'obvykle heslo' a test ma heslo 'test'
  private String firstName;
  private int id; // generovany
  private String lastName;
  private String password; // povinny
  private Set<Specimen> specimens = new HashSet<Specimen>();

  private String username; // povinny

  /** Zakladni bezparametricky konstruktor */
  public User() {
  super();
  }

  public User(int id, String username, String firstName, String lastName) {
  super();
  this.id = id;
  this.username = username;
  this.firstName = firstName;
  this.lastName = lastName;
  }

  @Override
  public int compareTo(final User other) {
  return new CompareToBuilder().append(username, other.username)
    .toComparison();
  }

  @Override
  public boolean equals(final Object other) {
  if (!(other instanceof User))
    return false;
  User castOther = (User) other;
  return new EqualsBuilder().append(username, castOther.username).append(
    id, castOther.id).isEquals();
  }

  @Transient
  @Override
  public GrantedAuthority[] getAuthorities() {
  GrantedAuthority[] authorities = (GrantedAuthority[]) authoritySet
    .toArray(new GrantedAuthority[authoritySet.size()]);
  return authorities;
  }

  @ManyToMany(targetEntity = Authority.class, fetch = FetchType.EAGER)
```

```
@JoinTable(name = "users_authorities", joinColumns = @JoinColumn(name = "
ref_user"), inverseJoinColumns = @JoinColumn(name = "ref_authority"))
public Set<Authority> getAuthoritySet() {
return authoritySet;
}

@Transient
public String getConfirmPassword() {
return confirmPassword;
}

@Column(name = "email", length = 100)
public String getEmail() {
return this.email;
}

@Column(name = "first_name", length = 50)
public String getFirstName() {
return this.firstName;
}

@Id
@SequenceGenerator(name = "users_id_seq", sequenceName = "users_id_seq")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "
users_id_seq")
public int getId() {
return this.id;
}

@Column(name = "last_name", length = 50)
public String getLastName() {
return this.lastName;
}

@Column(name = "password", length = 80, nullable = false)
public String getPassword() {
return this.password;
}

@OneToMany(targetEntity = Specimen.class, mappedBy = "userCreated")
@OrderBy("id")
public Set<Specimen> getSpecimens() {
return specimens;
}

@Column(name = "username", length = 20, unique = true, nullable = false)
public String getUsername() {
return username;
}

@Override
public int hashCode() {
return new HashCodeBuilder(920821029, -259222617).append(username)
  .append(id).toHashCode();
}

@Transient
@Override
public boolean isAccountNonExpired() {
```

```
return enabled;
}

@Transient
@Override
public boolean isAccountNonLocked() {
return enabled;
}

@Transient
@Override
public boolean isCredentialsNonExpired() {
return enabled;
}

@Column(name = "enabled")
public boolean isEnabled() {
return this.enabled;
}

public void setAuthoritySet(Set<Authority> authoritySet) {
this.authoritySet = authoritySet;
}

public void setConfirmPassword(String confirmPassword) {
this.confirmPassword = confirmPassword;
}

public void setEmail(String email) {
this.email = email;
}

public void setEnabled(boolean enabled) {
this.enabled = enabled;
}

public void setFirstName(String firstName) {
this.firstName = firstName;
}

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

public void setLastName(String lastName) {
this.lastName = lastName;
}

public void setPassword(String pass) {
this.password = pass;
}

public void setSpecimens(Set<Specimen> specimens) {
this.specimens = specimens;
}

public void setUsername(String username) {
this.username = username;
}
```

```
  @Override
  public String toString() {
  return getFirstName() + " " + getLastName();
  }

}
```

Ukázka A.1: Kompletní třída User.

A.2 Zaváděcí soubor web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
     http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <!-- Zakladni informace k aplikaci -->
  <display-name>DatabazeZQ - PrefaXY</display-name>
  <description>Aplikace k ukazce pouziti ramce Spring a Hibernate</
  description>

  <!-- Umisteni korenoveho aplikacniho kontextu -->
  <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value> /WEB-INF/spring/applicationContext*.xml </param-value>
  </context-param>

  <!-- Umisteni nastaveni logovani -->
  <context-param>
  <param-name>log4jConfigLocation</param-name>
  <param-value>classpath:/log4j.xml</param-value>
  </context-param>

  <!-- Vytvoreni korenoveho aplikacniho kontextu -->
  <listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</
  listener-class>
  </listener>
  <listener>
  <listener-class>org.springframework.security.ui.session.
  HttpSessionEventPublisher</listener-class>
  </listener>
  <listener>
  <listener-class>org.springframework.web.util.Log4jConfigListener</listener
  -class>
  </listener>

  <!-- Vstupni servlet aplikace -->
  <servlet>
  <servlet-name>databaseZQ</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
  class>
  <init-param>
```

```
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/webApplicationContext*.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Mapovani pozadvku -->
  <servlet-mapping>
  <servlet-name>databaseZQ</servlet-name>
  <url-pattern>*.html</url-pattern>
  </servlet-mapping>

  <!-- Vstupni soubor webovych stranek -->
  <welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <!-- Knihovna tagu do JSP sablon -->
  <jsp-config>
  <taglib>
    <taglib-uri>/spring</taglib-uri>
    <taglib-location>/WEB-INF/tld/spring-form.tld</taglib-location>
  </taglib>
  </jsp-config>

  <!-- Filtr pro kodovani znaku -->
  <filter>
  <filter-name>encoding-filter</filter-name>
  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</
  filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>UTF-8</param-value>
  </init-param>
  </filter>

  <filter-mapping>
  <filter-name>encoding-filter</filter-name>
  <url-pattern>*</url-pattern>
  </filter-mapping>

  <!-- Deklarace zakladniho filtru ramce Spring Security -->
  <filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter
  -class>
  </filter>

  <!-- Deklarace mapovani filtru pro Spring Security -->
  <filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>*</url-pattern>
  </filter-mapping>

  <!-- Zobrazeni chybovych stranek -->
  <error-page>
  <error-code>403</error-code>
  <location>/403.jsp</location>
  </error-page>
```

```
  <error-page>
  <error-code>404</error-code>
  <location>/404.jsp</location>
  </error-page>

  <!-- Ukonceni session po 15-ti minutach -->
  <session-config>
  <session-timeout>15</session-timeout>
  </session-config>

</web-app>
```

Ukázka A.2: Kompletní zaváděcí soubor web.xml

A.3 Soubor deklarace aplikačního kontextu

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http
://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p" xmlns:context="http://
  www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www
  .springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context http://www.
        springframework.org/schema/context/spring-context-2.5.xsd">

  <!-- Nastaveni konfiguracnich souboru -->
  <bean id="propertyPlaceholderConfigurer" class="org.springframework.beans.
  factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:/*.properties" />
  <property name="ignoreUnresolvablePlaceholders" value="true" />
  </bean>

  <!-- Zdroje zprav pro aplikaci s nastavenim kodovani souboru -->
  <bean id="messageSource" class="org.springframework.context.support.
  ReloadableResourceBundleMessageSource"
  p:basename="WEB-INF/spring/messages/messages" p:cacheSeconds="20">
  <property name="fileEncodings">
    <props>
    <prop key="WEB-INF/spring/messages/messages_cs">UTF-8</prop>
    <prop key="WEB-INF/spring/messages/messages_en">UTF-8</prop>
    <prop key="WEB-INF/spring/messages/messages">UTF-8</prop>
    </props>
  </property>
  </bean>

  <!-- Utilita pro ziskavani aplikacniho kontextu a "beanu" -->
  <bean id="springApplicationContext" class="cz.zpsv.databaseZQ.utils.
  SpringApplicationContext" />

  <!-- Manazery pro styk mezi webovou a perzistentni vrstvou -->
  <bean id="concreteClassManager" class="cz.zpsv.databaseZQ.managers.impl.
  ConcreteClassManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.ConcreteClass" />
  <constructor-arg>
```

```
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.ConcreteClass" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="concreteClassDao" ref="concreteClassDao" />
  </bean>
  <bean id="consistenceTypeManager" class="cz.zpsv.databaseZQ.managers.impl.
  ConsistenceTypeMangerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.ConsistenceType" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.ConsistenceType" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="consistenceTypeDao" ref="consistenceTypeDao" />
  </bean>
  <bean id="plantManager" class="cz.zpsv.databaseZQ.managers.impl.
  PlantManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.Plant" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.Plant" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="plantDao" ref="plantDao" />
  </bean>
  <bean id="prescriptionsManager" class="cz.zpsv.databaseZQ.managers.impl.
  PrescriptionsManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.Prescriptions" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.Prescriptions" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="prescriptionsDao" ref="prescriptionsDao" />
  </bean>
  <bean id="relayManager" class="cz.zpsv.databaseZQ.managers.impl.
  RelayManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.Relay" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.Relay" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="relayDao" ref="relayDao" />
  </bean>
  <bean id="specimenManager" class="cz.zpsv.databaseZQ.managers.impl.
  SpecimenManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.Specimen" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.Specimen" />
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
```

```
  </constructor-arg>
  <property name="specimenDao" ref="specimenDao" />
  </bean>
  <bean id="testTypeManager" class="cz.zpsv.databaseZQ.managers.impl.
  TestTypemanagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.TestType" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.TestType"></constructor-
    arg>
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="testTypeDao" ref="testTypeDao" />
  </bean>
  <bean id="userManager" class="cz.zpsv.databaseZQ.managers.impl.
  UserManagerImpl">
  <constructor-arg value="cz.zpsv.databaseZQ.pojo.User" />
  <constructor-arg>
    <bean class="cz.zpsv.databaseZQ.dao.hibernate.BaseDaoHibernate">
    <constructor-arg value="cz.zpsv.databaseZQ.pojo.User"></constructor-arg>
    <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
  </constructor-arg>
  <property name="userDao" ref="userDao" />
  <property name="passwordEncoder" ref="passwordEncoder"></property>
  <property name="saltSource" ref="saltSource"></property>
  </bean>
  <bean id="authorityManager" class="cz.zpsv.databaseZQ.managers.impl.
  AuthorityManagerImpl">
  <property name="authorityDao" ref="authorityDao" />
  </bean>

  <!-- Manazer transakci -->
  <bean id="transactionManager" class="org.springframework.orm.hibernate3.
  HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
  </bean>
</beans>
```

Ukázka A.3: Kompletní soubor deklarace aplikačního kontextu.

A.4 Anotovaný kontrolér webové vrstvy

```
// ==========================================================================
// Aplikace: databaseZQ
// Autor: Libor Ondrusek
// Spolecnost: FIT VUT Brno
// Rok: 2008
// Kadovani: UTF-8
// Soubor: SpecimenDetailFormController.java
// ==========================================================================

package cz.zpsv.databaseZQ.controllers;
```

```java
import java.sql.Timestamp;
import java.text.SimpleDateFormat;
import java.util.Collection;
import java.util.Date;

import javax.servlet.http.HttpServletRequest;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.beans.propertyeditors.StringTrimmerEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import cz.zpsv.databaseZQ.managers.IConsistenceTypeManager;
import cz.zpsv.databaseZQ.managers.IPlantManager;
import cz.zpsv.databaseZQ.managers.IPrescriptionsManager;
import cz.zpsv.databaseZQ.managers.IRelayManager;
import cz.zpsv.databaseZQ.managers.ISpecimenManager;
import cz.zpsv.databaseZQ.managers.ITestTypeManager;
import cz.zpsv.databaseZQ.pojo.ConsistenceType;
import cz.zpsv.databaseZQ.pojo.Plant;
import cz.zpsv.databaseZQ.pojo.Prescriptions;
import cz.zpsv.databaseZQ.pojo.Relay;
import cz.zpsv.databaseZQ.pojo.Specimen;
import cz.zpsv.databaseZQ.pojo.TestType;
import cz.zpsv.databaseZQ.pojo.editors.ConsistenceTypeEditor;
import cz.zpsv.databaseZQ.pojo.editors.PlantEditor;
import cz.zpsv.databaseZQ.pojo.editors.PrescriptionsEditor;
import cz.zpsv.databaseZQ.pojo.editors.RelayEditor;
import cz.zpsv.databaseZQ.pojo.editors.TestTypeEditor;
import cz.zpsv.databaseZQ.pojo.validators.SpecimenValidator;

/**
 * Formularovy kontroler MVC modulu ramce Spring, ktery se stara o zobrazeni
 * fomulare obsahujici detaily o vzorku
 * @author Libor Ondrusek
 * @version 1.0.0
 */
@Controller
@RequestMapping("/secure/client/specimen-detail")
@SessionAttributes("specimen")
public class SpecimenDetailFormController {

private IConsistenceTypeManager consistenceTypeManager;
/** Logovaci objekt */
private final Log logger = LogFactory.getLog(getClass());
private IPlantManager plantManager;
private IPrescriptionsManager prescriptionsManager;
```

```java
private IRelayManager relayManager;
private ISpecimenManager specimenManager;

private ITestTypeManager testTypeManager;

public IConsistenceTypeManager getConsistenceTypeManager() {
  return consistenceTypeManager;
}

public IPlantManager getPlantManager() {
  return plantManager;
}

public IPrescriptionsManager getPrescriptionsManager() {
  return prescriptionsManager;
}

public IRelayManager getRelayManager() {
  return relayManager;
}

public ISpecimenManager getSpecimenManager() {
  return specimenManager;
}

public ITestTypeManager getTestTypeManager() {
  return testTypeManager;
}

@InitBinder
public void initBinder(WebDataBinder binder) {
  SimpleDateFormat dateFormat = new SimpleDateFormat("dd.MM.yyyy");
  dateFormat.setLenient(true);
  binder.registerCustomEditor(Date.class, new CustomDateEditor(
    dateFormat, false));
  binder.registerCustomEditor(Plant.class, new PlantEditor(plantManager));
  binder.registerCustomEditor(Relay.class, new RelayEditor(relayManager));
  binder.registerCustomEditor(ConsistenceType.class,
    new ConsistenceTypeEditor(consistenceTypeManager));
  binder.registerCustomEditor(Prescriptions.class,
    new PrescriptionsEditor(prescriptionsManager));
  binder.registerCustomEditor(TestType.class, new TestTypeEditor(
    testTypeManager));
  binder
    .registerCustomEditor(String.class, new StringTrimmerEditor(
      true));
}

// Vlozi do atributu modelu vsechny typy konzistenci
@ModelAttribute("consistenceTypes")
public Collection<ConsistenceType> populateConsistenceTypes() {
  return consistenceTypeManager.getAll();
}

// Vlozi do atributu modelu adresu soucasne stranky
@ModelAttribute("currentURL")
public String populateCurrentURL(HttpServletRequest request) {
  return request.getRequestURI();
}
```

```
// Vlozi do atributu modelu vsechny vyrobni zavody
@ModelAttribute("plants")
public Collection<Plant> populatePlants() {
  return plantManager.getAll();
}

// Vlozi do atributu modelu vsechny receptury
@ModelAttribute("prescriptionses")
public Collection<Prescriptions> populatePrescriptionses() {
  return prescriptionsManager.getAll();
}

// Vlozi do atributu modelu vsechny vyrobni smeny
@ModelAttribute("relays")
public Collection<Relay> populateRelays() {
  return relayManager.getAll();
}

// Vlozi do atributu modelu vsechny typy zkousek
@ModelAttribute("testTypes")
public Collection<TestType> populateTestTypes() {
  return testTypeManager.getAll();
}

/**
 * Odesle data do databaze
 */
@RequestMapping(method = RequestMethod.POST)
public String prosessSubmit(@ModelAttribute Specimen specimen,
  BindingResult result, SessionStatus status, ModelMap model) {
  new SpecimenValidator().validate(specimen, result);
  if (!result.hasErrors()) {
  // doplneni udaju o uzivatelich
  // zaznam je zadavan jako novy
  if (specimen.getUserCreated() == null
    || specimen.getDateCreated() == null) {
    specimen.setUserCreated(this.specimenManager.getActualUser());
    specimen.setDateCreated(new Timestamp(this.specimenManager
      .getActualSystemTime().getTime()));
  }
  specimen.setUserModified(this.specimenManager.getActualUser());
  specimen.setDateModified(new Timestamp(this.specimenManager
    .getActualSystemTime().getTime()));
  // ulozeni vzorku do databaze
  Specimen savedSpecimen = this.specimenManager.save(specimen);
  status.setComplete();
  return "redirect:specimen-detail.html?id=" + savedSpecimen.getId()
    + "&action=detail";
  }
  model.addAttribute("hasErrors", new Boolean(true));
  model.addAttribute("formEditable", true);
  return "/secure/client/specimen-detail";
}

@Autowired
public void setConsistenceTypeManager(
  IConsistenceTypeManager consistenceTypeManager) {
  this.consistenceTypeManager = consistenceTypeManager;
```

```
}

@Autowired
public void setPlantManager(IPlantManager plantManager) {
  this.plantManager = plantManager;
}

@Autowired
public void setPrescriptionsManager(
  IPrescriptionsManager prescriptionsManager) {
  this.prescriptionsManager = prescriptionsManager;
}

@Autowired
public void setRelayManager(IRelayManager relayManager) {
  this.relayManager = relayManager;
}

@Autowired
public void setSpecimenManager(ISpecimenManager specimenManager) {
  this.specimenManager = specimenManager;
}

@Autowired
public void setTestTypeManager(ITestTypeManager testTypeManager) {
  this.testTypeManager = testTypeManager;
}

/**
 * Vytvori a predvyplni formular.
 * @param id Id vzorku.
 * @param model Model {@link ModelMap} pro zobrazeni ve formulari.
 */
@RequestMapping(method = RequestMethod.GET)
public void setupForm(
  @RequestParam(required = false, value = "id") Long id,
  @RequestParam(required = false, value = "action") String action,
  ModelMap model) {

  // pokud neni zadano id vzorku, nebo je v akci pridat, bude se pridavat
  if (id == null || id == 0
    || (action != null && action.equalsIgnoreCase("add") == true)) {
  logger.info("Formular detailu vzorku nacita vzorek s ID = " + id
    + " pro akci: " + action);

  model.addAttribute("specimen", new Specimen());
  model.addAttribute("formEditable", new Boolean(true));
  }
  // je pozadovana editace, nebo detail
  else if (id != null) {
  logger.info("Formular detailu vzorku nacita vzorek s ID = " + id
    + " pro akci: " + action);

  model.addAttribute("id", id);

  Specimen specimen = getSpecimenManager().getByPK(id);
  model.addAttribute("specimen", specimen);
  // Prida do modelu stari vzorku
  model.addAttribute("age", specimenManager.calcAgeInDays(specimen
```

```
    .getTested(), specimen.getCreated()));
  // Prida do modelu objemovou hmotnost ztvrdleho vzorku
  model.addAttribute("solidDensity", specimenManager
    .calcSolidDensity(specimen.getDistanceA(), specimen
      .getDistanceB(), specimen.getDistanceC(), specimen
      .getWeight()));
  // Prida do modelu zatizeni
  model.addAttribute("strength", specimenManager.calcStrength(
    specimen.getDistanceA(), specimen.getDistanceB(), specimen
      .getLoad()));
  // pokud je pozadovan detail, bez editace
  if (action.equalsIgnoreCase("detail") == true) {
    model.addAttribute("formEditable", false);
  }
  // pokud je pozadovana editace
  else if (action.equalsIgnoreCase("edit") == true) {
    model.addAttribute("formEditable", true);
  }
  }
}
}
```

Ukázka A.4: Kompletní anotovaná formulářový kontrolér.