VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

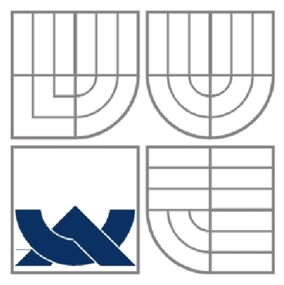
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRHOVÉ VZORY PRO WEBOVÉ APLIKACE

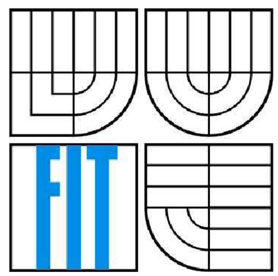
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAN DUDEK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRHOVÉ VZORY PRO WEBOVÉ APLIKACE
DESIGN PATTERNS FOR WEB APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE Bc. JAN DUDEK
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. JAROSLAV ZENDULKA, CSc.
SUPERVISOR

BRNO 2008

Abstrakt

S nástupem webu se začal objevovat zcela nový typ klient-server aplikací založených na standardních webových technologiích. Tyto technologie jsou dostupné na drtivé většině klientských stanic a není tak nutná instalace dalšího softwaru. Z tohoto důvodu si získaly webové aplikace mezi uživateli i vývojáři značnou oblibu a jsou nasazovány nejen na Internet, ale i na firemní intranety. Vzhledem k současnému rozšíření těchto aplikací je důležité sumarizovat vyzkoušené a ověřené postupy využitelné při jejich návrhu. A právě tato práce popisuje vybrané návrhové vzory aplikovatelné při vývoji webových aplikací a demonstruje jejich použití na konkrétních existujících systémech stejně jako na vlastní ukázkové aplikaci.

Klíčová slova

web, webová aplikace, analýza, návrh, návrhový vzor, UML, MVC, model, pohled, řadič

Abstract

With the coming of the web a brand new type of client-server applications based on standard web technologies has started to appear. These technologies are available on an overwhelming majority of client stations and don't need any additional software to be installed. As a result web applications have gained popularity among both users and developers and are currently deployed not only on the Internet, but also on company intranets. Considering present expansions of such applications it is important to summarize tested and approved approaches usable in their design. And it is this particular work that describes the chosen design patterns applicable in web applications development and demonstrates their usage on both concrete existent systems and a custom sample application.

Keywords

web, web application, analysis, design, design pattern, UML, MVC, model, view, controller

Citace

Dudek Jan: Návrhové vzory pro webové aplikace. Brno, 2008, diplomová práce, FIT VUT v Brně.

Návrhové vzory pro webové aplikace

Prohlášení

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

……………………
Jan Dudek
Datum

Poděkování

Na tomto místě bych chtěl poděkovat doc. Ing. Jaroslavu Zendulkovi, CSc. za ochotu k vedení této práce a za cenné připomínky k vznikajícímu textu. Dále bych chtěl poděkovat Davidu Lengyelovi za rady při návrhu grafického konceptu ukázkové aplikace.

© Jan Dudek, 2008.
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

OBSAH 1

1 ÚVOD 4

2 SPECIFIKA WEBOVÝCH APLIKACÍ 6

2.1 PROTOKOL HTTP 6
2.2 PROSTŘEDKY PRO TVORBU WEBOVÝCH STRÁNEK 7
2.2.1 Webové formuláře 7
2.3 WEBOVÝ PROHLÍŽEČ 7
2.4 WEBOVÝ SERVER 8
2.5 ARCHITEKTURA WEBOVÉ APLIKACE 8
2.6 VÝHODY A NEVÝHODY WEBOVÝCH APLIKACÍ 9
2.7 AJAX 9
2.8 WEBOVÉ SLUŽBY 10

3 VZORY PRO WEBOVÉ APLIKACE 11

3.1 POJEM NÁVRHOVÉHO VZORU 11
3.2 PROČ ZVLÁŠTNÍ VZORY PRO WEBOVÉ APLIKACE? 11
3.3 CHARAKTERISTIKA POPISOVANÝCH VZORŮ 11
3.4 POUŽITÁ UML NOTACE 12

4 VZOR WEB MVC 13

4.1 ÚČEL 13
4.2 PROBLÉM 13
4.3 ŘEŠENÍ 15
4.3.2 Service to Worker 16
4.3.3 Dispatcher View 16
4.4 IMPLEMENTACE 17
4.4.1 Rámec Struts 1.3.8 17
4.4.2 Ruby on Rails 17
4.4.3 Specifikace JavaServer Faces 1.2 19
4.4.4 Rámec ASP.NET 2.0 20

5 VZOR FRONT CONTROLLER 22

5.1 ÚČEL 22
5.2 PROBLÉM 22
5.3 ŘEŠENÍ 22
5.3.1 Architektura front controller komponenty 23
5.4 IMPLEMENTACE 25
5.4.1 Rámec Struts 1.3.8 25
5.4.2 Rámec Zend Framework 1.0.0 28
5.4.3 Specifikace JavaServer Faces 1.2 30
5.4.4 Rámec ASP.NET 2.0 31

6 VZOR DATA TRANSFER OBJECT 32

6.1 ÚČEL 32
6.2 PROBLÉM 32
6.3 ŘEŠENÍ 32
6.4 IMPLEMENTACE 34

6.4.1 *Prostředí JSP 2.0* ... 34
6.4.2 *Zend Framework 1.0.0* ... 35

7 VZOR INTERCEPTING FILTER ... 36

7.1 ÚČEL ... 36
7.2 PROBLÉM ... 36
7.3 ŘEŠENÍ ... 36
7.4 IMPLEMENTACE ... 37
7.4.1 *Prostředí Java EE* ... 37
7.4.2 *Rámec Struts2* ... 39
7.4.3 *Rámec ASP.NET 2.0* ... 40

8 VZOR VIEW HELPER ... 41

8.1 ÚČEL ... 41
8.2 PROBLÉM ... 41
8.3 ŘEŠENÍ ... 41
8.4 IMPLEMENTACE ... 42
8.4.1 *Prostředí JSP* ... 42
8.4.2 *Knihovna JSTL* ... 44
8.4.3 *Zend Framework 1.0.0* ... 44

9 VZOR COMPOSITE VIEW ... 46

9.1 ÚČEL ... 46
9.2 PROBLÉM ... 46
9.3 ŘEŠENÍ ... 46
9.3.2 *Pohled AtomicView* ... 47
9.3.3 *Pohled CompositeView* ... 47
9.3.4 *Definice rozložení Template* ... 47
9.3.5 *Správce vykreslení ViewManager* ... 48
9.4 IMPLEMENTACE ... 50
9.4.1 *Knihovna Tiles* ... 50
9.4.2 *ASP.NET 2.0* ... 51

10 NÁVRH UKÁZKOVÉ APLIKACE ... 52

10.1 MODEL POŽADAVKŮ APLIKACE ... 53
10.1.1 *Uživatelé systému* ... 53
10.1.2 *Přihlášení do systému* ... 53
10.1.3 *Odhlášení ze systému* ... 53
10.1.4 *Případy užití prováděné studentem* ... 54
10.1.5 *Zobrazit seznam zapsaných předmětů* ... 54
10.1.6 *Zobrazit projekty předmětu* ... 54
10.1.7 *Zobrazit varianty projektu* ... 54
10.1.8 *Zobrazit týmy projektu* ... 54
10.1.9 *Přihlásit k týmu* ... 55
10.1.10 *Odhlásit z týmu* ... 55
10.1.11 *Případy užití prováděné vyučujícím* ... 56
10.1.12 *Zobrazit seznam vyučovaných předmětů* ... 56
10.1.13 *Vytvořit projekt předmětu* ... 56
10.1.14 *Zobrazit projekt předmětu* ... 57
10.1.15 *Smazat projekt* ... 57
10.1.16 *Upravit projekt* ... 57

10.1.17 Vytvořit variantu projektu ... *57*
10.1.18 Zobrazit variantu projektu ... *58*
10.1.19 Upravit variantu ... *58*
10.1.20 Smazat variantu ... *58*
10.1.21 Hodnotit tým ... *58*
10.1.22 Hodnotit studenta ... *59*
10.2 VÝBĚR IMPLEMENTAČNÍ PLATFORMY ... 59
10.3 PŘÍNOS VZORU WEB MVC ... 59
10.3.2 Vrstva modelu ... *60*
10.3.3 Vrstva pohledu ... *62*
10.3.4 Vrstva řadiče ... *62*
10.3.5 Integrace vrstev aplikace ... *62*
10.3.6 Komunikace v rámci MVC architektury ... *63*
10.4 PŘÍNOS VZORU FRONT CONTROLLER ... 65
10.5 PŘÍNOS VZORU DATA TRANSFER OBJECT ... 66
10.6 PŘÍNOS VZORU INTERCEPTING FILTER ... 68
10.7 PŘÍNOS VZORU VIEW HELPER ... 69
10.7.1 Řídící struktury prezentační logiky ... *69*
10.7.2 Lokalizace aplikace ... *70*
10.7.3 Pokročilá prezentační logika ... *71*
10.8 PŘÍNOS VZORU COMPOSITE VIEW ... 73
11 ZÁVĚR ... **77**
BIBLIOGRAFIE ... **78**
ELEKTRONICKÉ ČLÁNKY ... 78
SEZNAM POUŽITÝCH ZKRATEK ... **80**
SEZNAM OBRÁZKŮ ... **81**
SEZNAM PŘÍLOH ... **82**

1 Úvod

S masivním rozšířením sítě Internet směrem k běžným uživatelům se začal objevovat nový typ aplikací typu klient-server, které pro komunikaci využívají standardní webové protokoly a technologie. Na místě původních statických stránek tak vyrůstají první jednoduché aplikace těžící z vysoké přívětivosti pro uživatele. Ten již nemusí instalovat a udržovat lokální kopii celého softwaru, ale komunikuje s ním vzdáleně pouze pomocí prohlížeče, který má standardně k dispozici a jehož prostředí důvěrně zná. Takto je postupem času připravena živná půda pro mnohem komplexnější systémy, jak je známe dnes. Zároveň si softwaroví inženýři mnohých firem začínají uvědomovat nesporné výhody tohoto architektonického řešení a uplatňují jej i v prostředí firemních intranetů.

Takto vytvářené aplikace byly a jsou limitovány webovými prostředky původně určenými pro přenos a prezentaci statického, resp. minimálně dynamického obsahu. Jak však jejich význam narůstal, začaly zpětně ovlivňovat prostředí, odkud vzešly. Např. po uvolnění webového mailového klienta *Gmail* společnosti *Google* byla vydána nová verze prohlížeče *Opera*, jejíž hlavním úkolem bylo vyřešit podporu této konkrétní aplikace[1]. Zároveň jsou navrhovány nové programovací a komunikační principy, které jdou *proti* původnímu zaměření webových technologií[2]. Lze tedy říci, že webové aplikace jsou dnes hlavní hybnou silou vývoje celého webu.

Ve kterékoliv inženýrské disciplíně je životně důležitá existence nějakého prostředku k opětovnému využití vyzkoušených a prověřených myšlenek a idejí. Jeden takový koncept využívaný v softwarovém inženýrství představují návrhové vzory umožňující, aby se při návrhu softwarového systému mohly recyklovat úspěšně zrealizované a ověřené nápady. Množina takových vzorů byla postupem času vytvořena i v prostředí webových aplikací. Bohužel však pro ně platí, že jsou často příliš závislé na konkrétním vývojovém a běhovém nástroji, např. existuje popsaná sada vzorů pro aplikace psané v prostředí *Java EE* apod.

Cílem této diplomové práce je prozkoumat zdroje s těmito vzory a pokusit se je zevšeobecnit natolik, aby byly aplikovatelné na libovolné vývojové prostředí. Chtěl bych tak vytvořit katalog několika detailně popsaných vzorů včetně ukázek jejich použití v konkrétních reálných systémech. Součástí této práce je i ukázková webová aplikace, na které se budu snažit demonstrovat výhody jejich aplikování.

V první kapitole práce se zaměřím na obecnou problematiku webových aplikací. Budu se snažit vysvětlit, proč si tato forma získala tak vysokou oblibu jak mezi softwarovými inženýry, tak uživateli, a poukážu na hlavní úskalí, kterým je v tomto prostředí nutno čelit, a na některé nové trendy ve vývoji webu. V následující kapitole se letmo dotknu tématiky návrhových vzorů a pokusím se legitimizovat důvod vzniku speciální sady vzorů pro webové aplikace.

V hlavní části práce se zabývám detailním popisem vybraných webových vzorů. Každému věnuji samostatnou kapitolu, ve které se snažím detailně osvětlit, jaká je konkrétní motivace pro jeho zavedení. K popisu struktury a principů daného vzoru využívám diagramů jazyka UML 2. Důležité je pro mě i implementační hledisko, na reálných systémech se snažím demonstrovat praktické využití.

---

[1] http://www.opera.com/docs/changelogs/windows/800/

[2] Např. principy *Ajax* a *Comet*, které se snaží vypořádat s omezeními komunikace typu požadavek-odpověď, která je vlastní právě webovému protokolu http (viz 2.7 a pozn. 9).

Výše zmíněná ukázková aplikace je tématem následující kapitoly. Ta popisuje její analýzu a návrh s důrazem na využití popisovaných vzorů. V závěru této práce pak hodnotím dosažené výsledky a navrhuji další možný vývoj.

2 Specifika webových aplikací

Webovou aplikací obvykle rozumíme softwarový systém umístěný na jednom či více serverech, se kterým uživatel komunikuje pomocí webových technologií. Skrz protokol **http** odesílá své požadavky a na oplátku získává grafické rozhraní, které je popsáno pomocí standardních prostředků používaných pro tvorbu **webových stránek**. GUI webové aplikace se tedy skládá z množiny stránek, které jsou interpretovány uvnitř **webového prohlížeče** nabízejícího omezené výpočetní prostředky, jde tedy o obdobu tenkého klienta. V následujícím textu se krátce zmíním o těchto základních technologiích.

2.1 Protokol http

Aplikační protokol http lze právem považovat za samotný základ webu. Původně byl navrhnut pro přenos hypertextových dokumentů. Jde o relativně jednoduchý protokol, který sice nemá definován transportní prostředek, ale zdaleka nejčastěji je přenášen přes protokol TCP. Následující vlastnosti jsou stěžejní z pohledu webových aplikací:

- Komunikace probíhá dle principu požadavek-odpověď. Vždy ji začíná klient svým **http požadavkem** na daný zdroj, jehož výsledek mu server odešle v rámci **http odpovědi**. Důležité je, že server nemůže klientu odesílat nějaká data bez toho, aby od něj předtím neobdržel požadavek.
- Součástí http požadavku je specifikace *metody*, což je operace, která se má vykonat s daným zdrojem. V prostředí webových aplikací se zdaleka nejčastěji využije metoda GET (v odpovědi je vrácena reprezentace zdroje) a POST (danému zdroji jsou odeslána data ke zpracování), případně ještě HEAD (vrátí pouze hlavičky, které by byly zaslány v odpovědi na GET). Existují i další metody (PUT, DELETE, TRACE, …), které jsou ale podporovány pouze minimálně.
- Protokol http je **nestavový**. Poté, co server odešle klientu http odpověď, nemá již o této transakci žádnou informaci. Při provádění následující transakce proto nemůže využít žádná data z té předchozí. To má mj. za následek, že udržování stavu spadá do zodpovědnosti klienta.
- Požadavek i odpověď jsou tvořeny textovými dokumenty. Pokud jsou proto v aplikaci přenášena citlivá data, je nutno použít bezpečnou alternativu **https**.

Následující výpis představuje možnou ukázku http požadavku:

```
GET /index.html HTTP/1.1
Host: example.com
Connection: close
Accept-Encoding: gzip
Accept: text/html, application/xml;q=0.9, application/xhtml+xml,
image/png, image/jpeg, image/gif, image/x-xbitmap, */*;q=0.1
Accept-Language: en-US,en;q=0.9
Accept-Charset: iso-8859-1, utf-8, utf-16, *;q=0.1
```

V tomto případě je požadován zdroj `example.com/index.htm` pomocí metody GET. Následují nepovinné hlavičky popisující dodatečné informace o klientském prohlížeči. Na takový požadavek může server zareagovat následující http odpovědí:

```
HTTP/1.1 200 OK
Date: Thu, 17 Jan 2008 13:24:17 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT
Content-Length: 438
```

```
Connection: close
Content-Type: text/html; charset=UTF-8

<HTML>
  <HEAD>
    <TITLE>Example Web Page</TITLE>
  </HEAD>
  <BODY>
  …
  </BODY>
</HTML>
```

Protože byla zvolena metoda GET, server odešle v rámci http odpovědi reprezentaci zdroje, zde v jazyce HTML. První řádek obsahuje stavový kód popisující výsledek operace. Následují další hlavičky obsahující dodatečné informace a na závěr samotné tělo odpovědi.

2.2 Prostředky pro tvorbu webových stránek

Pro definici obsahu webové stránky se využívá značkovací jazyk HTML postavený nad metajazykem SGML, případně některé jeho odnože[3]. Poskytuje prostředky pro popis sekcí, odstavců, seznamů, tabulek a odkazů na další stránky, pro vkládání obrázků a multimediálního obsahu a pro základní sémantické značení textu. Způsob formátování jednotlivých elementů se popisuje pomocí jazyka CSS, který umožňuje definovat kompletní rozvržení a grafický návrh. Interaktivní část rozhraní má na starost klientské skriptování nejčastěji v podobě určité verze jazyka *JavaScript*. Ten mj. umožňuje využití prostředků prohlížeče, přístup k obsahu stránky pomocí standardu DOM a událostmi řízené programování.

2.2.1 Webové formuláře

Základní kámen každé interaktivní aplikace představují uživatelské prvky, kterými lze zadat aplikaci pokyny a vstupní data. Mezi součásti jazyka HTML patří klasické prvky GUI známé z *desktop* aplikací jako různá textová pole, zaškrtávací pole, rádiová tlačítka, rozbalovací nabídky, rolovací seznamy a některé další [HAX, 287]. Tyto prvky jsou sdružovány do jednotlivých formulářů, jejichž obsah je zabalen do http požadavku a odeslán na server.

2.3 Webový prohlížeč

Webový prohlížeč představuje pro webové aplikace tenkého klienta, který interpretuje a zobrazuje uživateli právě obdrženou stránku grafické rozhraní. Pomocí prostředků daného počítače umožňuje http komunikaci se serverem. Prohlížeč se stal neodmyslitelnou součástí uživatelských PC a lze jej považovat za standardně dostupnou aplikaci.

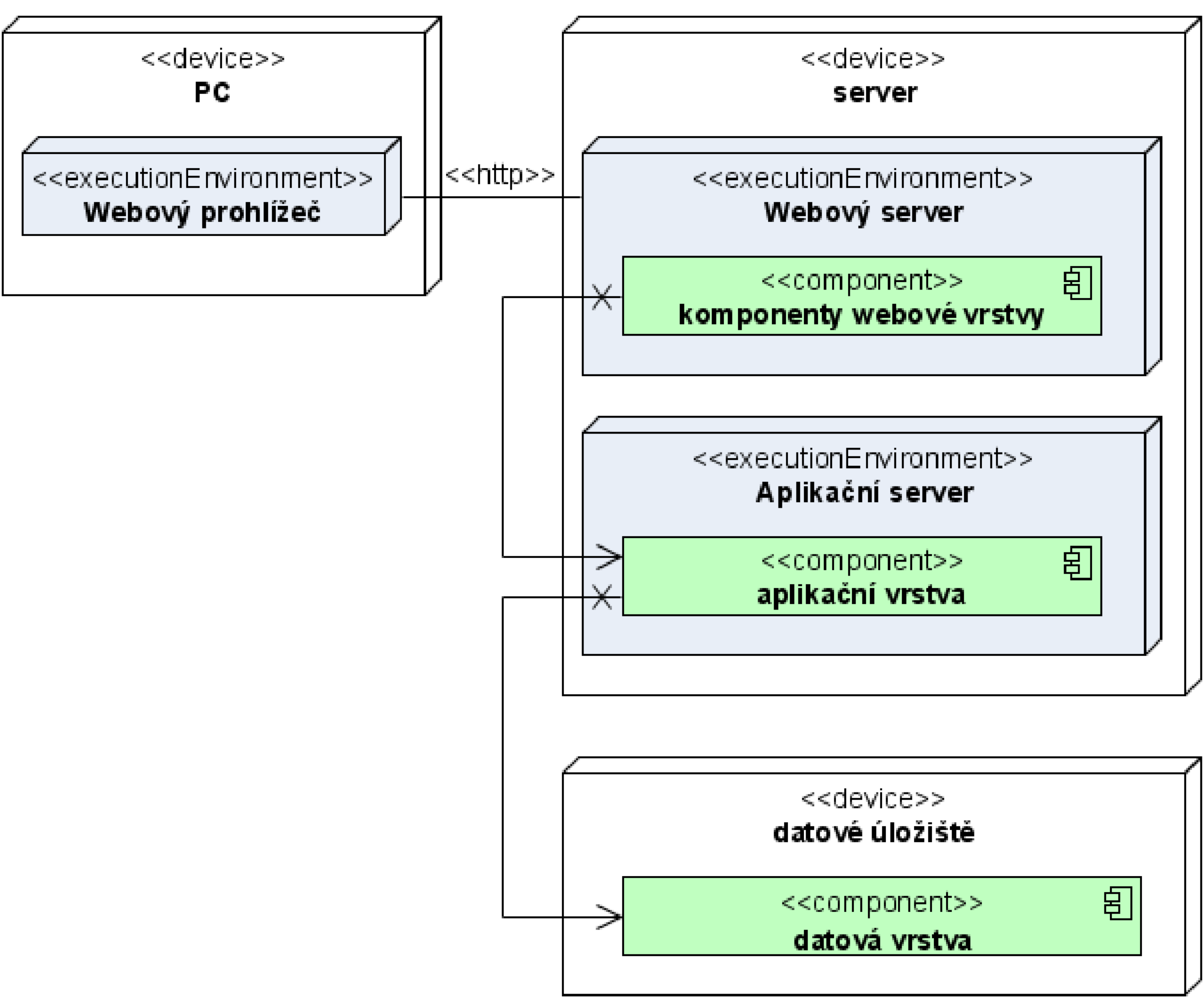
[3] Např.jazyk XHTML dodržující pravidla metajazyka XML.

2.4 Webový server

Webový server je program[4] naslouchající na daném portu příchozí http komunikaci. Přijaté požadavky je schopen předat dále webové aplikaci. Ta na jejich základě provede patřičné operace a vygeneruje nové uživatelské rozhraní, které pomocí prostředků serveru odešle ve formě http odpovědi zpět prohlížeči.

2.5 Architektura webové aplikace

Následující obrázek popisuje generickou webovou aplikaci se všemi jejími částmi:



Obrázek 2.1: Diagram nasazení popisující architekturu webové aplikace

Architekturu webové aplikace lze dělit dle daných kritérií na různé logické a fyzické vrstvy. Jedno takové dělení je naznačeno na uvedeném diagramu. Střední, aplikační vrstva zapouzdřuje logiku prováděnou nad daty získávanými z datové vrstvy. Co však činí webovou aplikaci odlišnou oproti jiným typům, je právě přítomnost webové vrstvy (u *desktop* aplikace se na tomto místě obvykle nachází vrstva grafického rozhraní). Ta se musí vypořádat s nástrahami komunikace přes protokol

[4] V jiném kontextu může být jako webový server označován počítač, na kterém je spuštěn tento program.

http, vhodným způsobem transformovat příchozí požadavky z GUI na volání příslušné aplikační logiky, na základě získaných dat vytvořit HTML stránku a odeslat ji zpět prohlížeči.

2.6 Výhody a nevýhody webových aplikací

Hlavní a nejdůležitější výhodou formy webové aplikace je jednoduchost její distribuce a údržby. Odpadá totiž starost s instalováním u zákazníka, webový prohlížeč a připojení k Internetu nebo firemnímu intranetu je v dnešní době považováno za standardní výbavu. Zákazník pouze potřebuje znalost příslušné URL adresy. Vzhledem k tomu, že je celá aplikace umístěna centrálně na serveru, se v podstatné míře usnadňuje její údržba. Zatímco při opravách klasických *desktop* aplikací je nutno záplatovat každou instalaci zvlášť a potýkat se tak s jejich případnými různými verzemi, webová aplikace existuje v jedné nebo několika málo instancích.

Webové aplikace skýtají ještě další podstatnou výhodu, kterou se odlišují od jiných síťových aplikací typu klient-server. Zatímco obecná síťová komunikace může být na různých úrovních blokována (např. na uživatelské stanici pomocí *firewallu* nebo v rámci celkového zabezpečení podnikové sítě), standardní http spojení bývá obvykle prostupné.

Neduhy webových aplikací pramení především z interpretování grafického rozhraní webovým prohlížečem. Přestože by teoreticky měla být dodaná data chápána totožným způsobem, praxe ukazuje velké množství nekompatibilit. Proto je obecně nutno rozhraní ladit pro předem vybranou množinu prohlížečů. Tento problém je však eliminován, pokud vyvíjíme aplikaci pro prostředí firemního intranetu s daným typem prohlížeče.

Další nevýhoda tkví v použití protokolu http, který je koncipován jako protokol typu požadavek-odpověď určený především pro prostředí webových stránek, méně však již webových aplikací[5]. Při použití standardních prostředků se uživatelovy akce přetváří v http požadavek, na který server reaguje zasláním odpovědi ve formě nového stavu rozhraní. Z toho vyplývá, že i při triviálním požadavku je nutno překreslovat celé rozhraní jeho aktuální verzí, ačkoliv by z povahy dané uživatelské akce možná stačilo zaměnit jeho malou část. Tímto jsou síťové prostředky zatěžovány mnohem víc, než by teoreticky bylo nutné, a navíc se GUI aplikace stává pro uživatele méně přívětivé.

2.7 Ajax

Techniky vývoje souhrnně nazývané jako **Ajax** (*Asynchronous Javascript And XML*) se snaží vyřešit výše nastíněný problém tím, že poskytují stránce možnost asynchronně komunikovat s http serverem a dynamicky od něj získávat data. Prostředníkem komunikace mezi klientskými skripty nejčastěji napsanými v jazyce *JavaScript* a serverem bývá objekt typu `XMLHttpRequest`, který je schopen odeslat http požadavek a následně přijmout odpověď [AJX, 39]. Tu opět převezme klientský skript a dynamicky provede změny ve zobrazené stránce.

Výhody použití této techniky spočívají především v uživatelské přívětivosti takto vytvořeného GUI. To dokáže mnohem rychleji reagovat, nemusí být totiž kompletně překreslováno dle obdržené http odpovědi, ale pouze aktualizuje svou určitou část. Nevýhody *Ajaxu* nalézáme v závislosti na podpoře klientského skriptování a na nabourání webového paradigmatu, že každý stav stránky má vlastní adresu. Je však nutno říct, že tento princip není pro webové aplikace natolik důležitý jako

[5] Samotný název protokolu, *HyperText Transfer Protocol*, prozrazuje jeho orientaci na přenos hypertextových dokumentů. Webová aplikace však toto původní určení rozšiřuje na práci v dynamickém prostředí.

v případě klasických stránek. Přes tyto nedostatky se AJAX stal důležitým prvkem při implementaci GUI webových aplikací a pro usnadnění jeho využití existují mnohé rámce a knihovny[6].

2.8 Webové služby

Webové aplikace jsou ze své podstaty uzpůsobeny lidskému uživateli, kterému poskytují GUI ve formě množiny HTML stránek. Toto může být překážkou, pokud chceme nastíněných výhod webové komunikace využívat i pro spolupráci dvou či více systémů. Pro tento případ jsou vhodnější tzv. **webové služby** (*web services*), což jsou komponenty aplikace dostupné prostřednictvím webového API. Podobně jako u webových aplikací probíhá komunikace pomocí protokolu http, kterým se ale v tomto případě přenášejí XML dokumenty se strukturou definovanou určitým protokolem. Typickým představitelem takového protokolu je SOAP (*Simple Object Access Protocol*).

Webová aplikace může zvýšit svou interoperabilitu tím, že bude poskytovat nejen klasické GUI popsané v jazyce HTML, ale i API k některým svým službám ve formě webových služeb.

[6] Např. řešení *Google Web Toolkit* (http://code.google.com/webtoolkit/).

3 Vzory pro webové aplikace

3.1 Pojem návrhového vzoru

Pod pojmem návrhový vzor si můžeme představit koncept, který *systematicky nazývá, vysvětluje a vyhodnocuje důležitý a v objektově orientovaných systémech se opakující návrh* [GoF, 21]. Jde tedy o prostředek generalizace opakovaných a úspěšných myšlenkových postupů. Návrháři tak nemusí vytvářet své artefakty na zelené louce, ale mají po ruce katalog hotových a prověřených řešení včetně jednoznačné terminologie, neboť každý vzor má svůj obecně akceptovaný název.

Základním materiálem pro studium návrhových vzorů se stala kniha *Design Patterns – Elements of Reusable Object Oriented Software* čtveřice autorů *Erich Gamma*, *Richard Helm*, *Ralph Johnson* a *John Vlissides*, která zavedla dělení návrhových vzorů na následující tři kategorie:

- **Tvořivé vzory**: přenášejí zodpovědnost za vytváření instancí tříd na jinou komponentu. Umožňují tak vybudovat systém, ve kterém lze mnohem pružněji nahrazovat různé části za jiné.
- **Strukturální vzory**: základním úkolem těchto vzorů je poskytnout návod pro skládání jednotlivých objektů do složitějších struktur tak, aby mohly uskutečňovat nové funkce.
- **Vzory chování**: zabývají se implementací a zapouzdřováním algoritmů do objektové struktury. Zaměřují se tedy na popis dynamického aspektu a komunikace.

Vzory lze pochopitelně využít i v jiných fázích vývoje softwaru než je návrh. Existují analytické vzory používané ve fázi analýzy nebo architektonické vzory užitečné při vysokoúrovňovém návrhu struktury celého systému.

Tato práce se zabývá podmnožinou vzorů využívaných při vývoji webových aplikací. Obecně se tyto vzory pohybují v rozmezí návrhových a architektonických vzorů.

3.2 Proč zvláštní vzory pro webové aplikace?

Webové aplikace jsou velkým krokem stranou od klasických *desktop* řešení a zároveň svébytnou formou distribuované síťové aplikace. Jsou však velmi rozšířeny a nasazovány nejen na Internetu, ale i na firemních intranetech, kde těží z mnohých nezanedbatelných výhod. Proto jsou vývojáři čím dál tím častěji zaměstnáni jejich vývojem, který může být poznamenán zvláštnostmi specifickými právě pro web. Specializované vzory pro webové aplikace by jim měly usnadnit práci a standardizovat vyzkoušená řešení v této oblasti.

Z hlediska možného rozvrstvení popsaného na obrázku 2.1 jde o vzory řešící problémy vyskytující se na webové vrstvě. Právě tato vrstva je unikátní webovým aplikacím, proto je přirozené, že se k ní vztahuje zvláštní sada vzorů.

3.3 Charakteristika popisovaných vzorů

V této práci popisuji šest webových vzorů, které považuji za důležité při vývoji každé webové aplikace. Architektonický vzor *Web MVC* aplikuje MVC paradigma na webové distribuované prostředí. Řadič takové architektury může být navržen pomocí vzoru *Front Controller*, zatímco *Data Transfer Object* ukazuje jeden ze způsobů předávání dat mezi jejími komponentami. Vzor *Intercepting Filter* představuje prostředek pro jednoduché přidávání zodpovědností celé webové

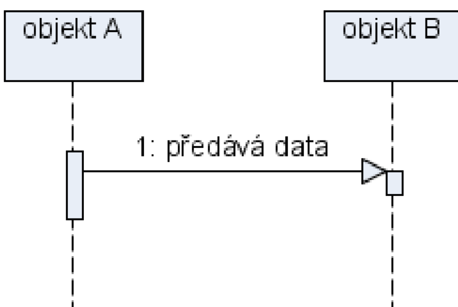
aplikaci. Lze tedy říci, že tyto čtyři vzory se zabývají architekturou webové vrstvy a komunikací v rámci vzoru *Web MVC*.

Dva zbývající vzory, *View Helper* a *Composite View*, poskytují řešení některých problémů, které se objevují při sestavování celého výstupu aplikace, který je zdaleka nejčastěji popsán v textovém dokumentu pomocí značkovacího jazyka.

Jak jsem již uvedl v úvodu této práce, všechny tyto vzory jsou již v literatuře obvykle popsány ve spojitosti s konkrétní vývojovou platformou. V následující části práce se je snažím zevšeobecnit natolik, aby byly využitelné v libovolném prostředí. Každému je věnována samostatná kapitola, jejíž struktura je inspirována členěním použitým v [GoF]. Každá kapitola začíná krátkou částí **Účel**, ve které je daný vzor uveden v několika větách. **Problém** uvádí důvod jeho vzniku nastíněním konkrétního problému. **Řešení** popisuje samotný vzor mj. pomocí vhodných UML diagramů. V poslední části, **Implementace**, popisuji použití vzoru v různých komerčních i *open-source* rámcích, případně uvádím vlastní fragment kódu z něj vycházející.

3.4 Použitá UML notace

Při popisování vzorů se snažím v maximální míře využít diagramů jazyka UML 2.0. V případě sekvenčních diagramů demonstrujících komunikaci vrstev nebo komponent z vyšší úrovně abstrakce značí text přiřazený k dané šipce zprávy nikoliv její název, ale význam v rámci celého scénáře. Např. následující ukázku zprávy je potřeba číst jako *objekt A předává data objektu B*:



Obrázek 3.1: Ukázka UML notace sekvenčních diagramů

V tomto případě bude `objekt B` pravděpodobně poskytovat nějakou metodu typu `setData(data)`, kterou přijímá data od dalšího objektu. Použitím této notace jsem se snažil, aby abstraktnější komunikační scénáře popsané pomocí sekvenčních diagramů byly dobře srozumitelné a čitelné.

4 Vzor Web MVC

4.1 Účel

Adaptuje klasický návrhový vzor *Model-View-Controller* na specifika prostředí webových aplikací.

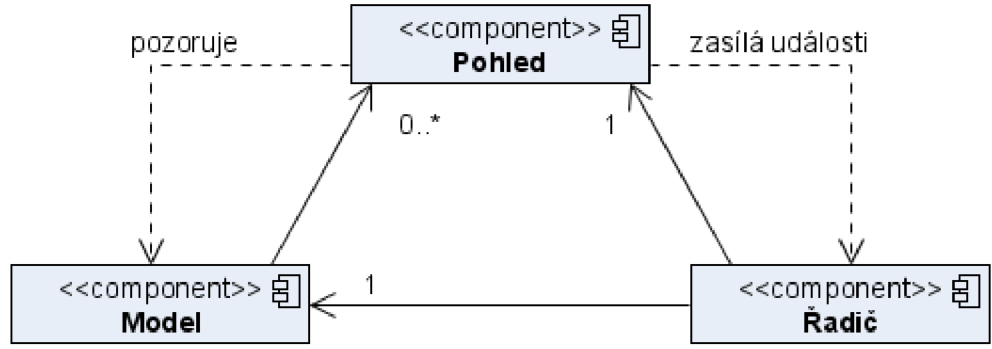
4.2 Problém

Architektonický vzor MVC obecně řeší problematiku dat a jejich prezentace rozdělením dané architektury na tři komponenty. **Model** obaluje stav aplikace a informace o něm dává k dispozici dalším součástem. Na jednotlivých úrovních granularity získává různou sémantiku. U jednoduché grafické komponenty, např. textového vstupu, může jít o instanci třídy zapouzdřující zadaný řetězec. Na úrovni systému pak model představuje celou aplikační logiku nebo rozhraní k ní.

Pohled realizuje zobrazení dat, resp. stavu modelu. Obvykle je ve formě grafického uživatelského rozhraní, které data nejen zobrazuje, ale také poskytuje uživateli prostředí pro zadávání požadavků. Asociace mezi pohledem a modelem má kardinalitu N:1.

Řadič hraje mezi modelem a pohledem roli prostředníka. Z pohledu obdrží informaci o uživatelově akci, tu zpracuje a zavolá příslušnou aplikační logiku uloženou v modelu.

Toto rozdělení zajistí, že model je pouze volně svázán se svými pohledy a je tedy znovupoužitelný. Následující diagram popisuje obecnou MVC architekturu včetně možných relací mezi jejími součástmi:



Obrázek 4.1: Diagram komponent obecné MVC architektury

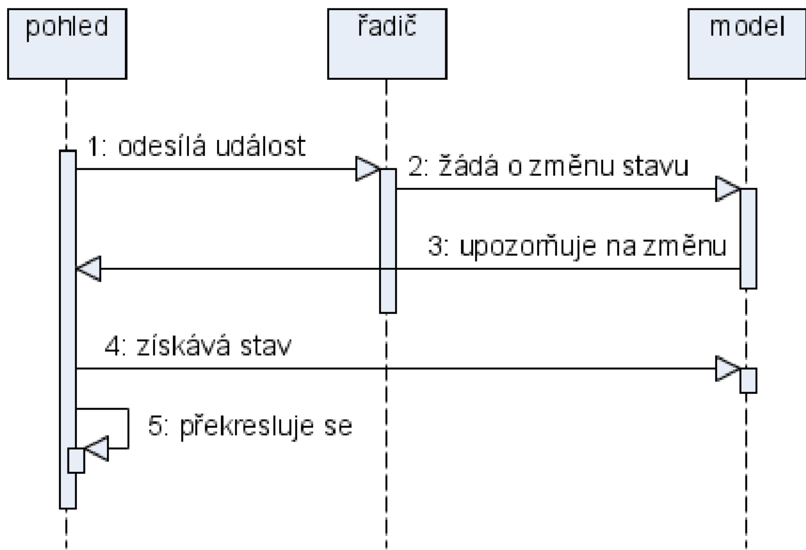
Je zřejmé, že jednotlivé části MVC musí mít mezi sebou jasně definované komunikační kanály. Mezi pohledem a modelem lze s úspěchem využít návrhového vzoru *Observer* [7]. Jeho implementace zajistí, že model registruje všechny své prezentace a posílá jim upozornění na změnu ve spravovaných datech. Jednotlivé pohledy na toto reagují zpětným doptáním se modelu na změny a překreslením.

Další komunikační cestou, kterou musíme řešit, je předávání uživatelských podnětů vznikajících na prezentační vrstvě příslušným řadičům. Zde lze využít návrhového vzoru *Event-*

[7] *„Definuje meziobjektovou závislost ‚1 ku n'. Změní-li jeden objekt svůj stav, všechny závislé objekty jsou automaticky upozorněny a zaktualizovány."* [GoF, 280]

Listener[8]. Data o uživatelově akci (např. souřadnice a příznak zmáčknutých tlačítek u události vyvolané myší) se zabalí do objektu reprezentujícího tuto událost a odešlou ke zpracování příslušnému řadiči.

Následující sekvenční diagram demonstruje zpracování uživatelského podnětu v rámci generické MVC architektury:



Obrázek 4.2: Sekvenční diagram komunikace v klasické MVC architektuře

1. Pohled odesílá údaje o uživatelově akci zapouzdřené do instance dané třídy události.
2. Řadič zpracuje příchozí událost a posílá modelu zprávu s žádostí o změnu jeho stavu.
3. Model posílá pohledu jednoduchou zprávu, že se změnil jeho stav.
4. Pohled si vyžádá od modelu informace o novém stavu.
5. Pohled se na základě těchto informací překreslí.

Komunikace probíhající podle uvedeného scénáře se obvykle nazývá *Push*. Pohled zůstává v klidu, dokud není nějakým způsobem modelem upozorněn na změnu stavu. Model je tedy zodpovědný za aktualizaci svých pohledů.

Tato strategie však není ve webových aplikacích ze samotné podstaty http protokolu realizovatelná, neboť každá komunikace musí začínat požadavkem klientské strany (viz 2.1). Model, umístěný na serveru, proto v tomto případě nemá žádné efektivní prostředky[9], jak notifikovat pohled na klientské straně.

---

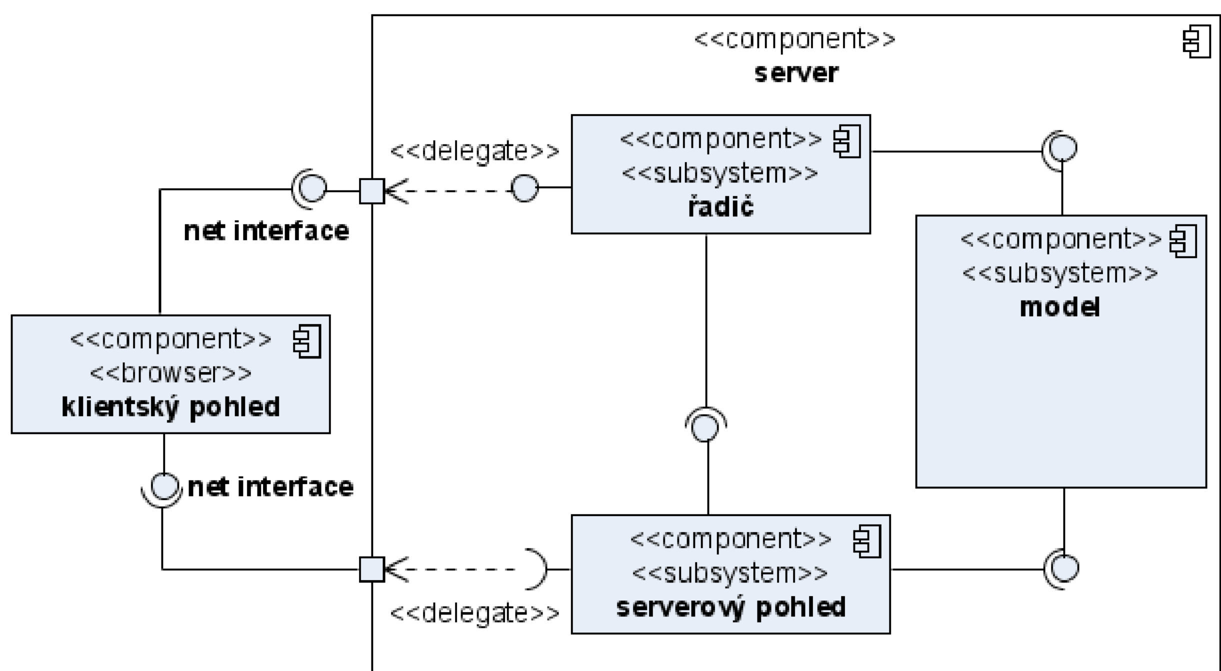
[8] *Event-Listener* (také označovaný jako *Publisher-Subscriber*) je vzor vycházející *z Observer*. Na rozdíl od něj však komponenta své pozorovatele (zde posluchače) neinformuje pouze o změně, ale přímo jim zasílá potřebná data, v tomto případě informace o akci uživatele. Velmi časté využití našel např. v API *Swing* a AWT, které jsou součástí platformy *Java SE*. [JPP, 127]

[9] Existuje několik technologií, které se pokoušejí o realizaci *push* v rámci http protokolu. Jde např. o řešení souhrnně nazývaná *Comet* používající neuzavřená http spojení a zpracování klientským skriptováním [KPR], využití `multipart/x-mixed-replace` *MIME typu* navrženého původně firmou *Netscape*, případně použití *Server-Sent Events* prohlížeče *Opera 9* implementující *server push* definovaný v návrhu HTML5 [ESW]. Žádné z těchto řešení však není dle mého názoru zatím příliš použitelné pro nasazení do komerčních aplikací.

4.3 Řešení

Chceme-li využít výhod vzoru MVC i pro webové aplikace, je potřeba jej adaptovat na zvláštnosti tohoto prostředí. Model v tomto případě totiž představuje všechny vrstvy umístěné *pod* webovou vrstvou (viz obrázek 2.1). Navíc musíme zavést novou komponentu, která se na serveru stará o vytvoření pohledu a o jeho distribuci zpět klientské straně v rámci http odpovědi. Tato součást bude dále označována jako **serverový pohled**.

Také je nutno revidovat komunikační kanály mezi jednotlivými součástmi. To v tomto případě znamená přizpůsobit tento vzor komunikační strategii označované jako *Pull*, která bere v potaz výše nastíněná specifika http protokolu. V tomto případě si pohled musí sám aktivně vyžadovat nová data od pasivního modelu, jde tedy o opak *Push*. Následující obrázek demonstruje architekturu vzoru *Web MVC*:



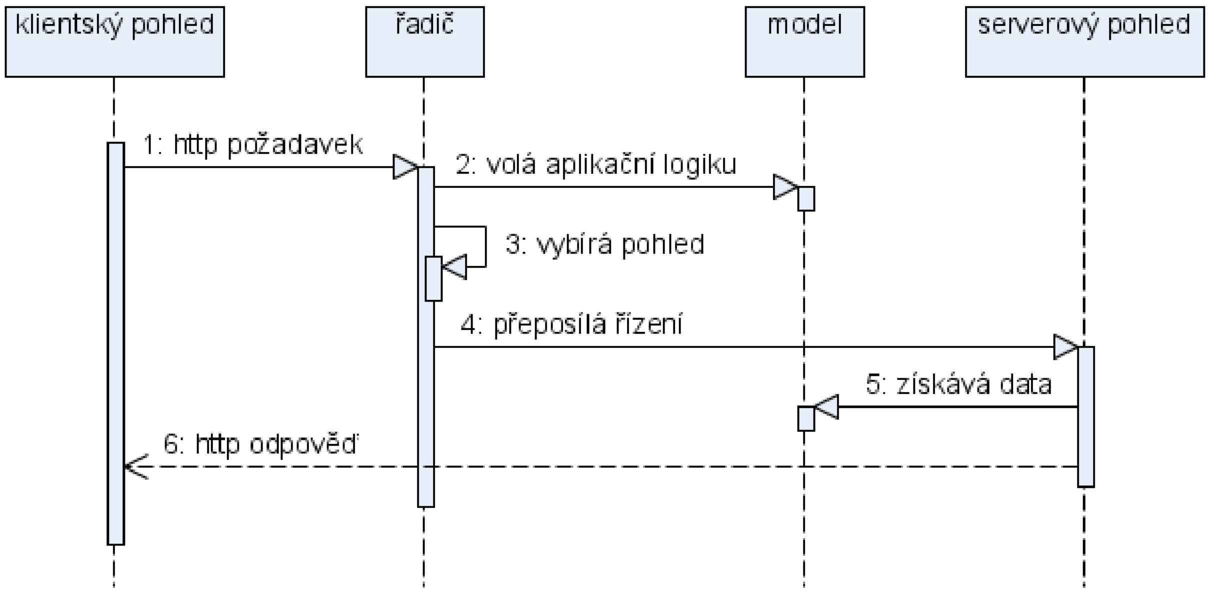
Obrázek 4.3: Architektura MVC adaptovaná na webové prostředí

Klientská část zasílá události ve formě http požadavků, které zachycuje řadič. Serverový pohled má za úkol vytvořit nový klientský pohled obsahující vhodně naformátovaná data modelu a odeslat jej zpět v podobě http odpovědi. Z toho vyplývá také další rozdíl oproti klasickému MVC. Pokud chce uživatel získat aktuální data (např. použitím funkce prohlížeče *refresh*), nekomunikuje klientská část pohledu přímo s modelem, ale opět přes řadič.

Model v takto široce pojaté MVC architektuře představuje celou aplikační logiku. Otázkou zůstává čas jejího volání. To může proběhnout před předáním řízení serverovému pohledu, nebo až posléze. Tyto strategie komunikace jsou v literatuře nazývány jako **Service to Worker**, resp. **Dispatcher View**. V prvním případě je volání blíže řadiči, v druhém blíže serverového pohledu. Obě tak definují kontinuum pro jeho umístění [CJP, 278].

4.3.2 Service to Worker

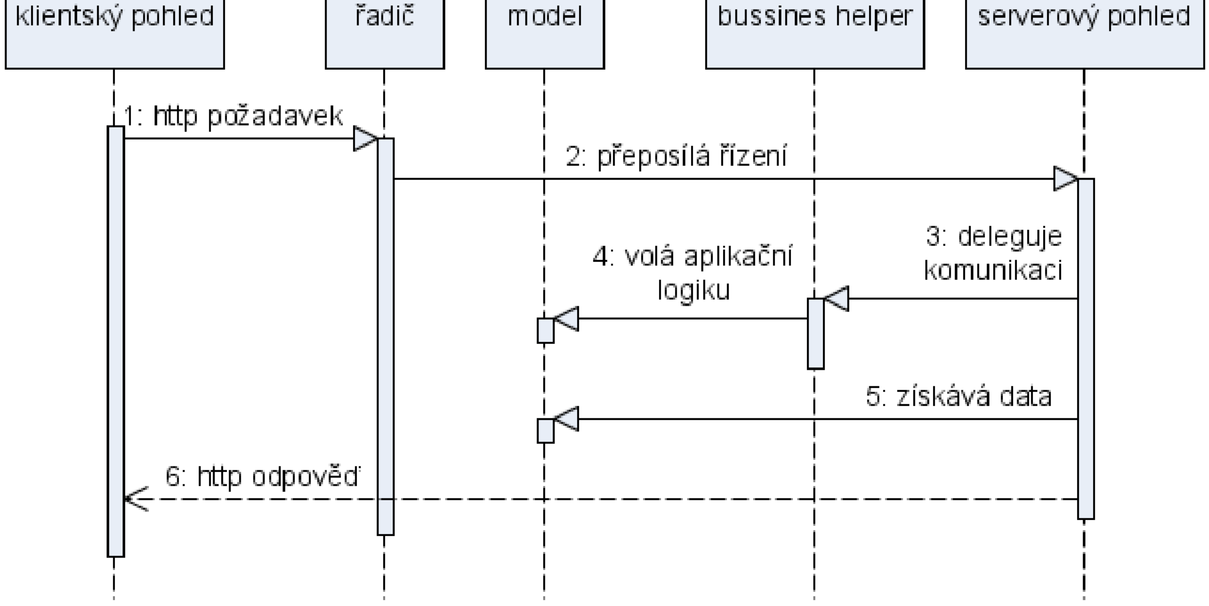
V této komunikační strategii[10] je volání aplikační logiky umístěno v řadiči. Aplikace, resp. rámce postavené na jejím základě, se také nazývají *Požadavkem řízené* [WFJ], protože příchozí http požadavek zde představuje spouštěcí mechanismus komunikace s modelem:



Obrázek 4.4: Sekvenční diagram komunikační strategie Service to Worker

4.3.3 Dispatcher View

Strategie *Dispatcher View* se vyznačuje voláním aplikační logiky až z pohledu. Jedinou úlohou řadiče je v tomto případě předat řízení vrstvě pohledu:



Obrázek 4.5: Sekvenční diagram komunikační strategie Dispatcher View

[10] V kontextu *Java EE* se toto řešení také často nazývá *Model 2* [PJS, 7].

V tomto případě obvykle pohled nekomunikuje s modelem přímo, ale deleguje k tomu komponenty dále označované jako **bussines helper**. Řešení využívající tuto komunikační strategii jsou označovaná jako *Komponentově orientovaná* [WFJ]. Díky odloženému volání se totiž serverový pohled může skládat z jednotlivých komponent představujících generický obraz klientských uživatelských prvků. Ty se mohou samy starat o svůj stav a aktivně komunikovat s modelem. Takové pojetí se tak více blíží tvorbě klasického *desktopového* GUI (např. napsaného v prostředí *Swing* nebo MFC).

4.4 Implementace

Architektura MVC se stala výchozím kamenem pro velké množství webových rámců napříč různými jazyky a platformami. Jejich základním úkolem je ulehčit vývoj a správu rozsáhlých webových aplikací s mnoha obrazovkami a formuláři. MVC rámec obvykle poskytuje tyto služby:

- **Centralizovaná správa navigace po jednotlivých obrazovkách aplikace**. Na jednom místě (obvykle v konfiguračním XML dokumentu) lze definovat posloupnost jednotlivých stránek včetně alternativních cest (např. návrat zpět na stránku s formulářem jako reakce na nesprávně zadaná data)
- **Již hotový řadič**. Ten je obvykle implementován dle vzoru *Front Controller* (viz 5.1).
- **Centralizovaná správa chybových stavů včetně jejich logování**. Dle dané konfigurace aplikace jednotně reaguje na zachycené chybové stavy.
- **Komplexní správa formulářů a prvotní zpracování jejich dat**. Rámec obvykle přebírá data z formuláře a ukládá je do připravené struktury, se kterou se manipuluje jednodušeji než se surovými textovými daty.
- **Validace přijatých dat**. Pokud některá data neprojdou konfigurovatelnou validací, rámec automaticky zobrazí zdrojový formulář s vyplněnými daty a chybovou zprávou.
- **Dodržování best practices používaných pro tvorbu webových aplikací**. Většina rámců např. umožňuje přechod na další stránku po zpracování formuláře pomocí *GET after POST* strategie[11].

4.4.1 Rámec Struts 1.3.8

MVC rámec *Struts*[12] slouží k vyvíjení webových aplikací v prostředí *Java EE*. Komunikace uvnitř jeho architektury probíhá dle strategie *Service to Worker*, veškeré řízení aplikace je tedy zapouzdřeno v dodaném řadiči (viz 5.4.1, kde je chování tohoto rámce popsáno podrobněji). Pro implementaci serverového pohledu se zde využívá technologie JSP, která představuje nedílnou součást *Java EE* platformy. Co se týče modelu, *Struts* nepředepisují žádnou konkrétní technologii, lze tedy využít např. rámec *Spring*, *Java EE* technologii EJB nebo vlastní, jednodušší řešení modelu.

4.4.2 Ruby on Rails

Webový rámec *Ruby on Rails*[13] je z hlediska této práce zajímavý tím, že architekturu MVC komunikující pomocí strategie *Service to Worker* využívá pro řešení, jehož hlavním cílem je zrychlit

---

[11] Jako odpověď na http POST požadavek je odesláno prohlížeči přesměrování pomocí stavového kódu 302, resp. 303 na stránku, kterou již klient získá pomocí nového požadavku GET. Je takto zamezeno opakovanému odeslání POST dat.

[12] http://struts.apache.org/1.x/

[13] http://www.rubyonrails.com/

a zautomatizovat vývoj především CRUD (*Create-Read-Update-Delete*) webových aplikací. Jako programovací jazyk je zde použit *Ruby*. Na rozdíl od dalších popisovaných rámců poskytuje podporu pro všechny tři složky MVC architektury.

4.4.2.1 Model

Model se v případě *Ruby on Rails* skládá z doménových entit, jejichž třídy se generují přímo na základě tabulek v dané databázi. Pokud v ní např. existuje tabulka s názvem `students`, definice příslušné třídy je prostá:

```
class Student < ActiveRecord::Base
end
```

Základem je zde technologie *ActiveRecord*, která je schopna ze struktury záznamů v dané tabulce vygenerovat potřebné atributy třídy a vzájemně je provázat, jde tedy o řešení na principu objektově-relačního mapování. Zároveň bude takto vytvořená třída poskytovat základní statické metody pro CRUD operace jako `new()`, `find()` nebo `find_all()`. V tomto případě tedy bude třída `Student` obsahovat kompletní logiku persistence prostým děděním z `ActiveRecord::Base` [ROR, 181].

4.4.2.2 Řadič

Řadič rámce *Ruby on Rails* je navržen dle vzoru *Front Controller*, prochází jím tedy všechny http požadavky. Rozšiřovat jej lze pomocí specializací třídy `ApplicationController`, které v architektuře tohoto vzoru představují typický příklad *helper* komponent (viz 5.3.1).

Výběr příslušné *helper* součásti závisí na příchozím http požadavku, požadavek na zdroj `/MyHelper/myMethod` bude vyhodnocen tak, že se má použít instance třídy `MyHelper` a zavolat její metoda `myMethod()`. O celou tuto funkčnost se rámec stará automaticky.

V kódu řadiče se nejčastěji využije některé ze statických metod třídy entity a získaný objekt se pak uloží pro zobrazení v serverovém pohledu:

```
class StudentAction < ApplicationController
  def list
    @students = Student.find_all
  end

  def find
    @student = Student.find(@params["id"])
  end
end
```

V tomto případě poskytuje řadič dvě akce, `list` a `find` (dostupné skrz URL http://example.com/StudentAction/list, resp. http://example.com/StudentAction/find). První z nich získá z modelu kolekci všech studentů, druhá pak jednoho konkrétního studenta na základě dodaného identifikátoru. V obou případech budou nalezené entity dostupné i v pohledu.

4.4.2.3 Pohled

Jednotlivé stránky pohledu jsou podobně jako např. v technologii JSP směsicí statického HTML kódu a speciálních syntaktických konstruktů, ve kterých lze přistupovat k entitám předaným z řadiče a vypisovat jejich vlastnosti.

4.4.3 Specifikace JavaServer Faces 1.2

JavaServer Faces představuje oficiální specifikaci[14] společnosti *Sun* pro webové rámce platformy *Java EE*. Jde tedy o sadu pravidel a rozhraní, kterými se musí řídit výrobci kompatibilních rámců. Sun zároveň poskytuje i vzorovou implementaci JSF RI[15], existují však i další, např. *open-source* řešení *myFaces*[16].

JSF specifikace popisuje pokročilou MVC architekturu inspirovanou strategií *Dispatcher View*. Snaží se o komponentně orientovaný přístup k tvorbě uživatelského rozhraní. Zpracování http požadavku se řídí tzv. *JSF životním cyklem* [JE5, 169] skládajícím se ze šesti fází:

1. **Restore View** – na základě http požadavku je identifikován klientský pohled, odkud byl odeslán. Ten je na serveru reprezentován stromem komponent skládajícím se z obrazů jednotlivých uživatelských prvků (pohled obsahuje formuláře, které se skládají z jednotlivých komponent, proto stromová struktura) a k nim dle konfigurace přidaným *validátorům* (objekty validující data dané komponenty) a *event handlerům* zpracovávajícím události přidružené k prvku. Pokud jde o první požadavek uživatele na tento pohled, vytvoří se nový strom neobsahující žádná data. V opačném případě (při *postback*[17]) se obnoví již předtím vytvořený strom popisující stav pohledu při minulém požadavku.
2. **Apply Request Values** – tato fáze je určena pro aktualizování stavu již vytvořeného stromu komponent. Každý prvek nastaví svůj stav dle hodnoty získané z http požadavku. Protože mají prvky zároveň informaci o svém minulém stavu, mohou generovat události informující o jeho změně.
3. **Process Validation** – nyní se spouští všechny validátory přiřazené komponentám v první fázi. Pokud je alespoň jedna hodnota komponenty shledána nevalidní, pokračuje se šestou fází, tj. vykreslí se zdrojový pohled společně s chybovými zprávami upozorňujícími na špatně zadané hodnoty.
4. **Update Model Values** – v této fázi již máme zvalidovaná data na úrovni formuláře (ale stále ještě nemusí být validní z hlediska *business* úrovně, např. neplatné heslo), proto je lze odeslat do příslušného objektu připojeného k danému požadavku v konfiguraci (tzv. *backing bean*), který představuje z hlediska MVC architektury komponentu *bussines helper*.
5. **Invoke Application** – komponenta *backing bean* nyní v sobě obsahuje zvalidovaná a zkonvertovaná data, lze tedy volat její příslušnou metodu přiřazenou k danému formuláři, která obsluhuje komunikaci s modelem. Tato metoda vrací řetězec popisující výsledek operace. V konfiguračním souboru aplikace se pak určí, který následný pohled je svázán se kterým konkrétním řetězcem.
6. **Render Response** – aplikační logika byla zavolána a výsledek převzat. Řízení se předá komponentě `ViewHandler`, která řídí vytvoření pohledu a jeho odeslání v rámci http odpovědi.

Z hlediska MVC architektury nejdříve http požadavek převezme řadič navržený dle vzoru *Front Controller*. Ten má relativně málo pravomocí (viz 5.4.3), pouze spustí instanci životního cyklu, která začne provádět v kontextu uživatele jednotlivé výše uvedené fáze. Řízení je tedy předáno

---

[14] http://java.sun.com/javaee/javaserverfaces/

[15] https://javaserverfaces.dev.java.net

[16] http://myfaces.apache.org

[17] Odeslání formuláře pomocí metody POST na adresu stránky, na které je umístěn.

pohledu ve formě stromu komponent tak, jak je to popsáno ve strategii *Dispatcher View*. Z pohledu je také volána aplikační logika pomocí *bussines helper* komponent, zde ve formě *backing beans*[18].

Díky značné modularitě je poslední fáze, vykreslení odpovědi, nezávislá na konkrétní technologii pohledu. Zmíněný strom komponent představuje generický pohled společný všem technologiím, zatímco samotné generování dat pro klienta zajišťuje třída rozšiřující abstraktní třídu `ViewHandler`. Lze tak použít libovolnou technologii, vedle JSP např. *Facelets*[19] nebo generování pohledu pomocí XSLT transformace. Podobně jako *Struts* je JSF nezávislé na technologii implementace modelu.

4.4.4 Rámec ASP.NET 2.0

Rámec ASP.NET představuje komplexní webové řešení[20] firmy *Microsoft* pro platformu .NET. Z uvedených rámců je nejpodobnější specifikaci JSF, se kterou sdílí inspiraci komunikační strategií *Dispatcher View*, komponentně orientovaný přístup a použití událostně řízeného programování. I zde sestává zpracování příchozího požadavku z několika etap [ASC, 110], pro které platí, že ve svém průběhu vytvářejí události řídící chod celé aplikace:

1. **Page Initialization** – rámec na základě příchozího požadavku vytváří na serveru hrubý obraz stránky ve formě množiny ovládacích prvků[21] a získává základní objekty nutné pro zpracování požadavku.
2. **View State Loading** – jednotlivé ovládací prvky získají svůj stav z minulého požadavku, pokud jde o *postback*. Informace o tomto stavu lze stejně jako v případě JSF ukládat u klienta nebo na serveru [AVS].
3. **Postback Data Processing** – v této fázi si ovládací prvky vyžádají svůj nový stav z http požadavku. Zároveň tak získají informaci, zdali se tento stav změnil od minula.
4. **Page Loading** – ovládací prvky jsou rámcem umisťovány do stromové struktury, podobně jako v případě JSF, čímž je dokončeno vytváření kompletního obrazu stránky na serveru.
5. **Postback Change Notification** – rámec hledá v celém stromu ovládací prvky, u kterých došlo ke změně stavu zjištěné v třetí etapě, a vyvolává příslušné upozorňující události.
6. **Postback Event Handling** – v této etapě nastane hlavní část volání aplikační logiky. Je vyvolána událost oznamující stisknutí daného odesílacího prvku na klientu (nejčastěji *submit* tlačítko formuláře) a zachycena v kódu, který dále komunikuje s modelem.
7. **Page Pre-rendering Phase** – realizují se finální akce, které musí proběhnout ještě před samotným vytvářením odpovědi klientu.
8. **View State Saving** – z jednotlivých ovládacích prvků se vytvoří textový řetězec obsahující serializované informace o jejich stavu a je uložen pro získání v dalším případném *postback* požadavku.
9. **Page Rendering** – jednotlivé ovládací prvky zapisují své tělo ve formě HTML elementů do http odpovědi.

---

[18] Zajímavým architektonickým řešením se prezentuje rámec *Seam* (http://www.jboss.com/products/seam), který umožňuje v JSF jako *backing bean* komponenty použít přímo objekty modelu implementovaného v EJB3. V tomto případě jsou tedy úplně vynechány *bussines helper* komponenty a pohled komunikuje přímo s modelem.

[19] https://facelets.dev.java.net

[20] http://www.asp.net

[21] Ovládacím prvkem může být v ASP.NET libovolná komponenta stránky. Může jít o jednoduchou třídu pouze generující HTML kód stejně jako složitější prvek s vlastní logikou.

10. **Page Unloading** – provádějí se poslední úklidové operace a ničí se všechny objekty, které reprezentovaly daný pohled na serveru.

Každý požadavek je při vstupu do rámce nejdříve zpracován příslušným *http handlerem*, který představuje jednoduchou *front controller* komponentu, jejíž zodpovědností je spustit životní cyklus zpracování stránky (viz 5.4.4). V tuto chvíli je tedy aktivován serverový pohled ve formě stromu ovládacích prvků. V etapě *Postback Event Handling* se pak předá řízení ovladači definovanému v tzv. *code behind* souboru, což je množina ovladačů událostí přiřazených dané stránce. Odtud je volána aplikační logika, proto tuto část rámce můžeme považovat za *bussines helper* prvek architektury. Samotný model může být implementován libovolnou technologií. Na závěr se pohled rekonstruovaný na serveru vypíše do http odpovědi a zruší se. V tomto ohledu se MVC architektura ASP.NET liší od specifikace JSF, která nabízí navíc připojení libovolné technologie pro vytvoření výsledné http odpovědi.

5 Vzor Front Controller

5.1 Účel

Centralizuje základní zpracování http požadavků do jediné komponenty řadiče, která může mít dle zvolené komunikační strategie i další zodpovědnosti.

5.2 Problém

V prostředí webových aplikací existuje množina úkonů, které se na serveru musí provést hned po přijetí http požadavku. Jde vesměs o operace související s jeho základním zpracováním jako správa http sezení, převzetí parametrů požadavku, dat z *cookies*, základní autorizace a autentikace apod. Tyto úkony jsou obvykle pro celou webovou aplikaci jednotné nebo alespoň velmi podobné, proto je vhodné, aby byly všechny implementovány jednou a na jediném místě.

Při použití komunikační strategie *Service to Worker* (viz 4.3.2) jsou na řadič webové aplikace kladeny mnohem vyšší nároky. Po zpracování http požadavku je dále obecně zodpovědný za komunikaci s modelem a za výběr adekvátního pohledu, kterému nakonec předá řízení. Závislosti výběru operace na příchozím http požadavku a volby příslušného pohledu na výsledku zmíněné operace tak vlastně definují celé chování webové aplikace. Správa tohoto chování by měla být také centralizovaná a tím pádem mnohem jednodušeji konfigurovatelná.

5.3 Řešení

Z nastíněného problému vyplývá, že pro prostředí rozsáhlých webových aplikací se jeví jako nejlepší strategie použít jediný centrální řadič, resp. několik málo řadičů zodpovědných za řízení jednotlivých logických částí aplikace. Tato komponenta se pak označuje **front controller** a plní obvykle následující úlohy:

- **Přijímá všechny http požadavky.** V daném prostředí proto musí existovat možnost přesměrování příchozích http požadavků na jedinou komponentu.
- **Spravuje http sezení.** Je zodpovědná za udržování kontextu uživatele na serveru a odstiňuje tak vývojáře od nízkoúrovňové manipulace s identifikátory sezení.
- **Zodpovídá za základní bezpečnost aplikace.** Vzhledem k tomu, že webová aplikace má často jediný model zabezpečení, je vhodné jej implementovat právě do této komponenty.

S posunem zodpovědnosti za řízení aplikace směrem k řadiči získává navíc *front controller* další zodpovědnosti:

- **Provádí základní validaci přijatých dat.** Pokud je schopen i bez volání aplikační logiky rozpoznat, že přijatá data jsou nekorektní (např. nevyplněná povinná hodnota, neplatná celočíselná hodnota apod.), rovnou deleguje příslušný serverový pohled, aby to uživateli vhodně oznámil (např. opětovným zobrazením formuláře se zvýrazněnými chybnými daty).
- **Komunikuje s modelem.** Na základě zpracovaného požadavku volá příslušnou aplikační logiku.
- **Deleguje řízení na pohled.** Na základě dostupné logiky obvykle získané z konfiguračního souboru vybírá serverový pohled, kterému vhodným způsobem zpřístupní model (resp. data modelu) a předá řízení.

5.3.1 Architektura front controller komponenty

Z předchozího textu je patrné, že *front controller* může být zodpovědný za relativně rozsáhlý počet úloh. Proto je obvyklé, že bývá složen z více částí s vlastními zodpovědnostmi. Obvykle rozpoznáváme tyto součásti:

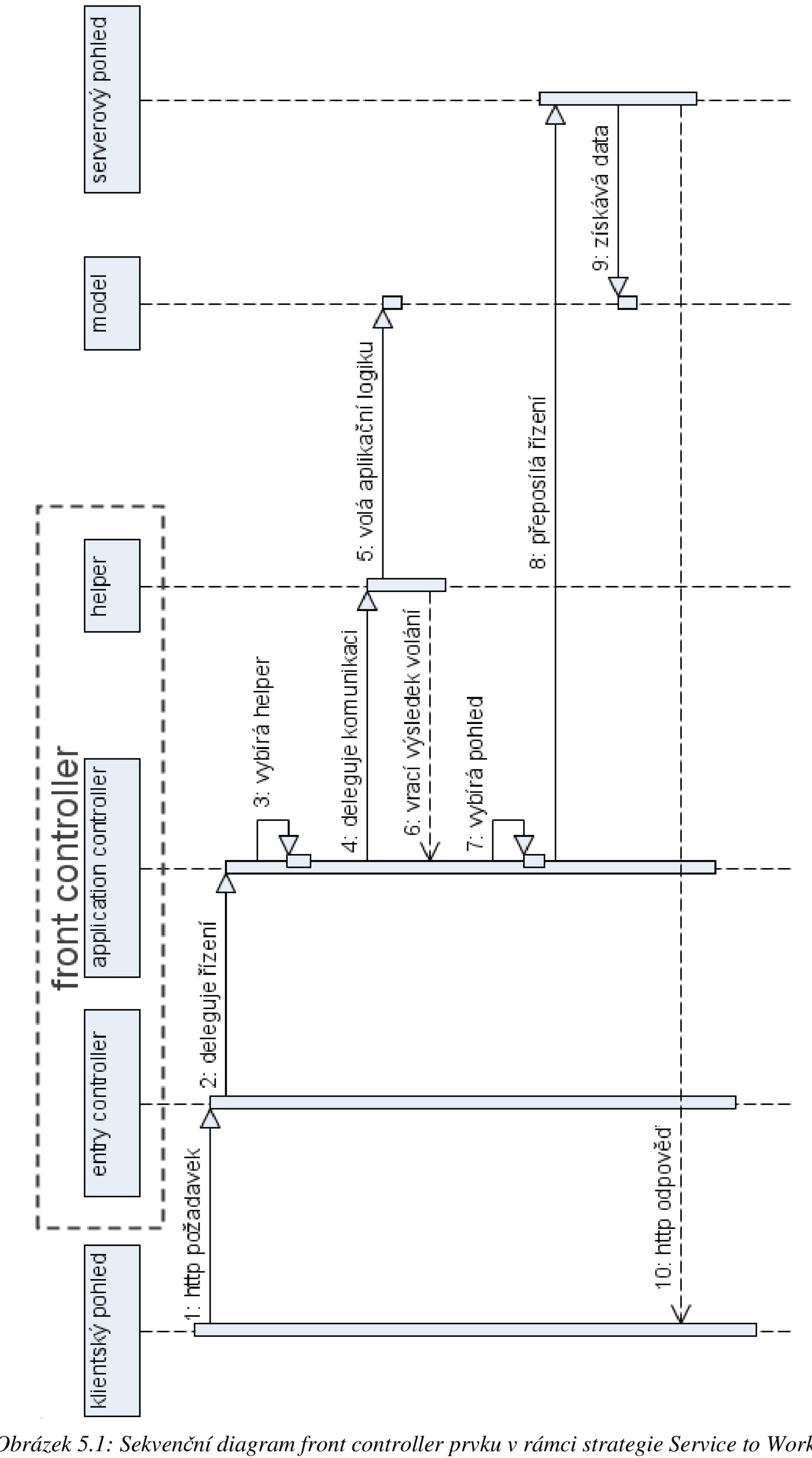
- **Entry controller** – představuje místo, do kterého jsou směřovány všechny http požadavky. Je zodpovědný za jejich převzetí a prvotní zpracování.
- **Application controller** – často také označován jako *dispatcher*. Obvykle je přítomen ve strategii *Service to Worker* a je v něm centralizována veškerá navigace aplikace. *Entry controller* mu předá předzpracovaný http požadavek, na základě kterého vybere správnou *helper* komponentu a spustí ji. Dle výsledku této operace pak vybere následný serverový pohled.
- **Helper** – pomocná komponenta, která zapouzdřuje komunikaci s modelem. Často je implementována pomocí návrhového vzoru *Command* [22], tedy více *helperů* sdílí stejné rozhraní a lze tak kód v nich obsažený spouštět stejným způsobem.

Jak bylo uvedeno u vzoru *Web MVC*, strategie komunikace *Service to Worker* a *Dispatcher View* vytvářejí kontinuum pro umístění volání aplikační logiky. Na tom pak samozřejmě závisí i rozsah zodpovědnosti *application controller* komponenty. Obecně se tato zodpovědnost zvětšuje, čím *blíže* je volání řadiči.

5.3.1.1 Strategie Service to Worker

V případě tohoto typu komunikace musí volání aplikační logiky proběhnout ještě před předáním řízení pohledu. *Application contoller* proto musí dle přijatého http požadavku nalézt správnou *helper* komponentu, spustit její operaci a na základě výsledku delegovat serverový pohled:

[22] *„Zapouzdří žádost do objektu a tím umožní parametrizaci klientů pomocí různých žádostí…“* [GoF, 228]



Obrázek 5.1: Sekvenční diagram front controller prvku v rámci strategie Service to Worker

5.3.1.2 Strategie Dispatcher View

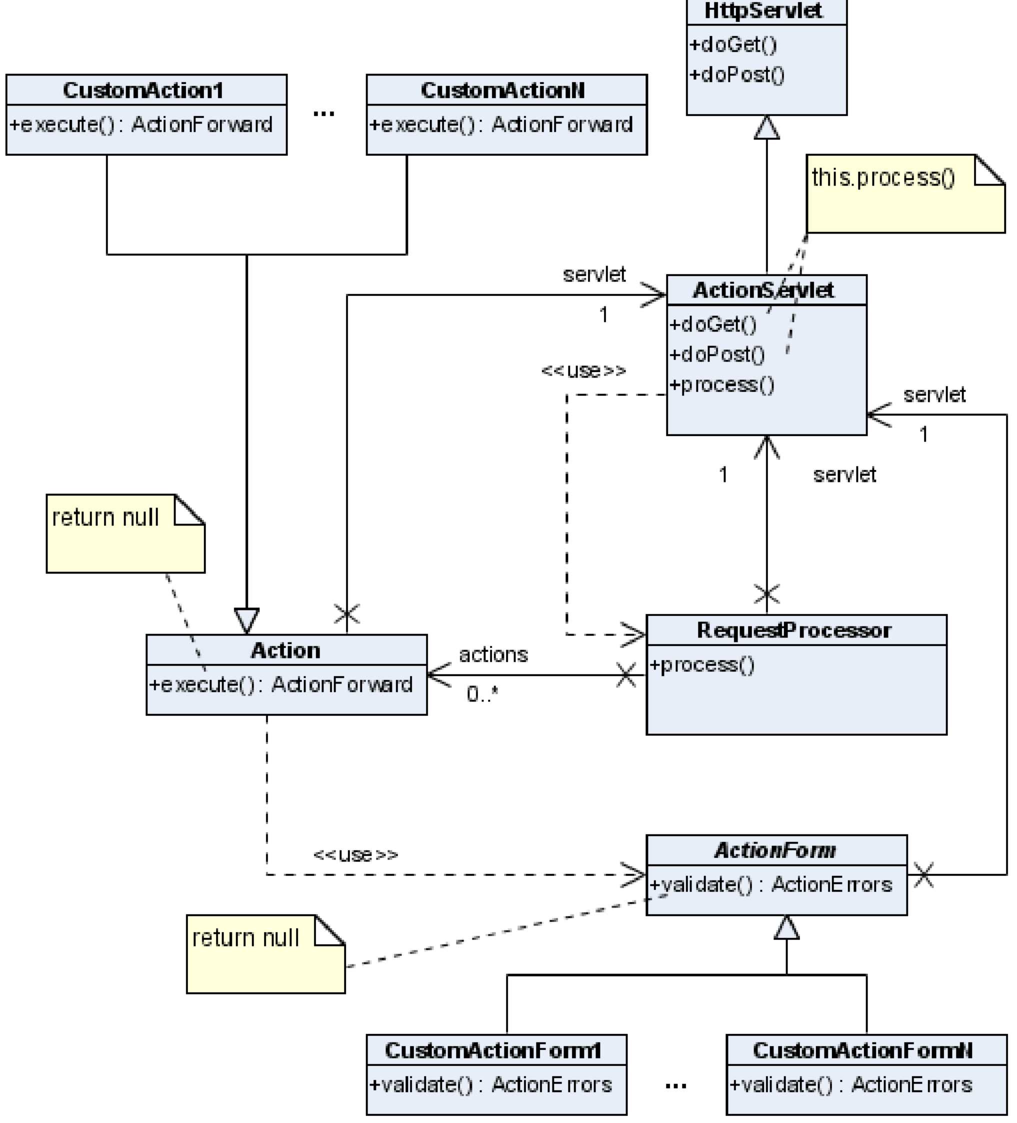
V této strategii hraje *front controller* pouze malou roli, protože aplikační logika je volána až z pohledu (viz obrázek 4.5). Proto nemusí být tak strukturovaný jako v předešlém případě a splývá do jediné malé komponenty, jejíž úlohou je základní zpracování http požadavků a předání řízení serverovému pohledu. V extrémním případě pak může být *front controller* jako takový úplně vynechán a jeho úlohu převezme přímo web server, resp. kontejner.

5.4 Implementace

Vzor *Front Controller* je v určité formě k nalezení ve většině rozšířených webových rámcích. Lze jej však samozřejmě využít i při psaní vlastního řadiče.

5.4.1 Rámec Struts 1.3.8

V MVC rámci *Struts* je řadič implementován pomocí návrhového vzoru *Front Controller* a komunikace s modelem probíhá prostřednictvím strategie *Service to Worker*. Strukturu řadiče [PJS, 110] demonstruje následující diagram:



Obrázek 5.2: Diagram tříd front controller komponenty rámce Struts

5.4.1.2 ActionServlet

Třída `ActionServlet` je specializací *Java EE* třídy `HttpServlet`, která představuje bázovou třídu pro všechny servlety zpracovávající http požadavky. V architektuře *Struts* plní roli vstupního bodu. Metody `doGet()` i `doPost()` pouze volají metodu `process()`, jejíž jedinou úlohou je získat referenci na instanci třídy `RequestProcessor` a předat jí řízení. Třída `ActionServlet` tak koresponduje s komponentou *entry controller*. Všechny http požadavky jsou do ní směřovány deklarativně v souboru *web.xml*, který je součástí specifikace *Java EE*:

```
…
<servlet>
  <!-- definice instance servletu -->
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
…
</servlet>

<servlet-mapping>
```

```
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
…
```

V tomto případě jsou všechny http požadavky na zdroj odpovídající masce `*.do` přesměrovány na instanci servletu `ActionServlet` spravovanou webovým kontejnerem.

5.4.1.3 Action

Do podtříd bázové třídy `Action` vývojář zapouzdřuje volání aplikační logiky modelu. Chování je vloženo do metody `execute()` vracející objekt třídy `ActionForward`, který popisuje následný pohled. Specializace `Action` jsou tak klasickým představitelem *helper* komponent navržených dle návrhového vzoru *Command*.

5.4.1.4 ActionForm

`ActionForm` je bázová abstraktní třída pro třídy určené k přejímání dat od klienta. Rámec do jejich instancí automaticky ukládá hodnoty parametrů http požadavku. Metoda `validate()` slouží k validaci přijatých dat a je volána hned poté, co je ukládání dokončeno.

Podtřídy `ActionForm` tedy slouží jako obálka pro data získaná především z formulářů. Příslušná specializace `Action` ji má k dispozici a získaná data dále zpracovává a používá při komunikaci s modelem.

5.4.1.5 RequestProcessor

Na instanci třídy `RequestProcessor` je delegován výběr správné podtřídy `Action` a volání její metody `execute()`, převzetí výsledku v podobě instance `ActionForward` a předání řízení správnému pohledu. `RequestProcessor` tak v sobě zapouzdřuje chování *application controlleru.* Příslušná konfigurace logiky výběru akce a pohledu se nachází v hlavním konfiguračním souboru, obvykle nazývaném *struts-config.xml*. Následující okomentovaný fragment kódu pochází z metody `process()` této třídy a popisuje způsob, jakým probíhá zpracování http požadavku:

```
public void process(HttpServletRequest request,
  HttpServletResponse response)
  throws IOException, ServletException {
…

  //získej pravidla pro výběr akce a pohledu
  ActionMapping mapping = processMapping(request, response, path);
  if (mapping == null) {
    return;
  }
…

  //získej ActionForm pro tento požadavek, naplň jej daty a zvaliduj
  ActionForm form = processActionForm(request, response, mapping);
  processPopulate(request, response, form, mapping);
  try {
    if (!processValidate(request, response, form, mapping)) {
      return;
    }
…

  //získej akci pro tento požadavek
  Action action = processActionCreate(request, response, mapping);
```

```
  if (action == null) {
    return;
  }

  //proveď akci a převezmi její výsledek
  ActionForward forward =
    processActionPerform(request, response, action, form, mapping);

  //předej řízení pohledu na základě obdrženého výsledku
  processForwardConfig(request, response, forward);
}
```

5.4.2 Rámec Zend Framework 1.0.0

Rámec *Zend Framework* je určen pro zjednodušení implementace systémů s architekturou MVC v prostředí PHP[23]. Protože uvnitř komunikuje dle strategie *Service to Worker*, je hlavní těžiště zpracování požadavku umístěno do řadiče navrženého dle vzoru *Front Controller*.

5.4.2.1 Směrování požadavků do jednoho řadiče

Vzhledem k absenci pokročilé webové platformy umožňující konfiguraci aplikace na jednom místě (např. již zmíněný soubor *web.xml* v architektuře *Java EE*) je nutno vyřešit problém směřování všech požadavků do řadiče. V prostředí PHP proto musíme využít prostředky, které nám nabízí samotný http server. Při použití oblíbeného serveru *Apache* lze dosáhnout přesměrování pomocí jeho modulu *URL Rewrite*[24]. V následujícím pravidle se všechny požadavky kromě těch na statické soubory (obrázky, kaskádové styly apod.) přesměrují ke zpracování do skriptu *index.php*:

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

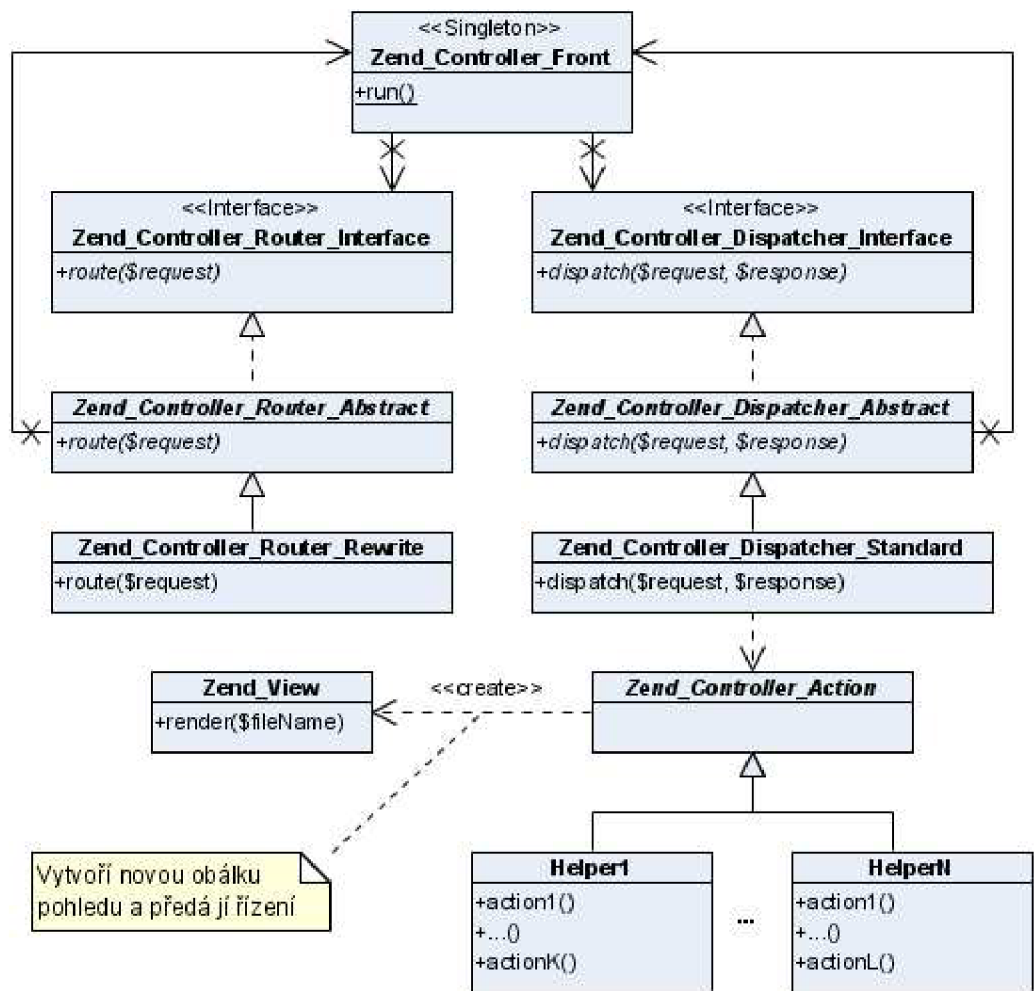
Při použití jiného http serveru by samozřejmě bylo nutno využít jiného obdobného prostředku. V uvedeném skriptu je pak nastartován celý proces zpracování tak, že se předá řízení řadiči. Doporučená podoba tohoto skriptu je minimalistická:

```
<?php
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('cesta/k/helper/komponentam');
```

Nyní je již konečně předáno řízení samotnému rámci, resp. řadiči. Jeho strukturu popisuje následující diagram:

[23] http://www.zendframework.com/

[24] http://httpd.apache.org/docs/2.0/mod/mod_rewrite.html



Obrázek 5.3: Diagram tříd front controller komponenty rámce Zend Framework

5.4.2.2 Zend_Controller_Front

Tato třída představuje vstupní místo pro všechny příchozí požadavky. Je implementována dle návrhového vzoru *Singleton*[25], v jednu chvíli tedy může existovat nejvýše jedna její instance. Ta převezme požadavek, vytvoří základní objekty potřebné při jeho zpracování a následně jej přepošle dalším komponentám řadiče. V kontextu vzoru *Front Controller* jde tedy o klasický příklad části *entry controller*.

5.4.2.3 Zend_Controller_Action

Helper komponenty řadiče jsou reprezentovány specializacemi abstraktní třídy `Zend_Controller_Action`. V tomto rámci mají navíc zodpovědnost za výběr správného pohledu, který je obalen do instance třídy `Zend_View`. *Helper* tedy na základě výsledku komunikace s modelem vybere dle naprogramované logiky příslušný pohled a pomocí zmíněné instance mu předá řízení:

```
//volání aplikační logiky
…
$view = new Zend_View();
if ($userExists){
  echo $view -> render('userLogged.phtml');
```

[25] *„Zajišťuje, aby jedna třída měla pouze jednu instanci, a poskytuje k ní globální přístupový bod“* [GoF, 134]

```
}
else {
  echo $view -> render('badLogin.phtml');
}
…
```

5.4.2.4 Logika pro výběr příslušného helperu

Pro volbu správného *helperu* zodpovědného za komunikaci s modelem musí existovat určitá logika, v rámci *Zend Framework* zapouzdřená v tzv. *routeru*. Jde o třídu implementující rozhraní `Zend_Controller_Router_Interface`, jejíž úkolem je na základě informací z požadavku vybrat správnou *helper* třídu a její konkrétní metodu, která bude volána.

Standardní dodávaná implementace tohoto rozhraní `Zend_Controller_Router_Rewrite` provádí výběr na základě informací poskytnutých v URL adrese. Přijde-li na aplikaci sídlící na http://example.com požadavek http://example.com/helper1/action2/param/value, bude použita třída `Helper1` a volána její metoda `action2()`, která bude mít k dispozici parametr `param` s hodnotou `value`. Rámec poskytuje i další *routovací* algoritmy, případně lze použít vlastní řešení.

Jako poslední se dostává ke slovu *dispatcher*, standardně implementovaný dodanou třídou `Zend_Controller_Dispatcher_Standard`. Jeho úlohou je převzít od *routeru* informace o zvolené třídě *helperu*, instanciovat ji a zavolat její vybranou metodu.

Obě dvě popsané komponenty tedy plní funkci *application controller* prvku, neboť obsluhují celou navigaci po aplikaci.

5.4.3 Specifikace JavaServer Faces 1.2

Implementace *MyFaces 1.1.5* specifikace JSF používá k základnímu zpracování http požadavku *front controller* v podobě servletu `FacesServlet`. Jak již bylo řečeno v kapitole 4.4.3, tato specifikace vychází ze strategie *Dispatcher View* a řadič má tedy relativně malou zodpovědnost. To se samozřejmě promítlo i do rozsahu metody `service()` přijímající http požadavky:

```
public void service(ServletRequest request,
  ServletResponse response)
  throws IOException, ServletException {
…

  //identifikace zdrojového pohledu
  String pathInfo = httpRequest.getPathInfo();
  if (pathInfo != null
    && (pathInfo.startsWith("/WEB-INF") ||
      pathInfo.startsWith("/META-INF"))) {
…

    //soubory z adresářů WEB-INF a META-INF jsou nepřístupné
    ((HttpServletResponse) response).
      sendError(HttpServletResponse.SC_NOT_FOUND);
    return;
  }
…

  //získej kontext zpracování požadavku
  FacesContext facesContext =
    _facesContextFactory.getFacesContext(
      _servletConfig.getServletContext(),
      request,
      response,
```

```
        _lifecycle);

    //spusť jednotlivé fáze životního cyklu
    try {
      _lifecycle.execute(facesContext);
      _lifecycle.render(facesContext);
    }
…

}
```

Stejně jako u rámce *Struts* jsou i v tomto případě http požadavky směrovány do řadiče pomocí deklarace v souboru *web.xml*.

5.4.4 Rámec ASP.NET 2.0

Podobně jako v předchozím případě i zde hraje *front controller* pouze malou úlohu, neboť rámec ASP.NET 2.0 také vychází z komunikační strategie *Dispatcher View* (viz 4.4.4). Příchozí požadavek je zde převzat podle typu cílového zdroje příslušnou *http handler* komponentou, která v případě požadavku na stránku webové aplikace předá řízení serverovému pohledu [ASC, 227]. Následující fragment standardní konfigurace rámce popisuje mapování komponent na požadavky:

```
<httpHandlers>
  …
  <add verb="*" path="*.java" type="System.Web.HttpForbiddenHandler" />
  <add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory" />
  <add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory" />
  …
</httpHandlers>
```

Přijde-li např. požadavek na zdroj s příponou `.aspx`, bude zpracován pomocí komponenty vytvořené továrnou `PageHandlerFactory` a podobně pro další typy. Pokud z nějakého důvodu nevyhovují *handler* prvky dodávané s rámcem, je možno vytvořit vlastní. Každá taková komponenta musí implementovat následující rozhraní:

```
public interface IHttpHandler {
  bool IsReusable { get; }
  void ProcessRequest(HttpContext context);
}
```

Metoda `IsReusable()` vrací logickou hodnotu podle toho, zda je možno využít jednu instanci třídy pro zpracování více požadavků současně. Hlavní práci pak odvádí metoda `ProcessRequest()`, která převezme kontext zpracování zapouzdřující http komunikaci a provede příslušné přípravné operace, než předá požadavek dále. Takto vytvořený *handler* je pak nutno přiřadit k odpovídajícímu typu požadavku, jak bylo demonstrováno výše.

6 Vzor Data Transfer Object

6.1 Účel

Poskytuje mechanismus pro bezpečné vystavení dat modelu, které se mají prezentovat daným serverovým pohledem.

6.2 Problém

Ve webové MVC architektuře komunikující pomocí strategie *Service to Worker* (viz 4.3.2) musíme řešit otázku, jakým způsobem po volání aplikační logiky zpřístupnit data z modelu serverovému pohledu zejména z hlediska zachování co nejvolnějšího spojení těchto dvou částí aplikace. Zatímco v klasickém MVC se pohled po notifikaci doptá modelu na jeho nový stav, ve webovém MVC je kvůli značně volné vazbě mezi modelem a serverovým pohledem obtížné zachovat tuto komunikační cestu. Navíc dvojí komunikace s modelem webové aplikace (poprvé při volání aplikační logiky, podruhé při získávání dat) může vést k redundantním operacím a ke zpomalení celého systému.

6.3 Řešení

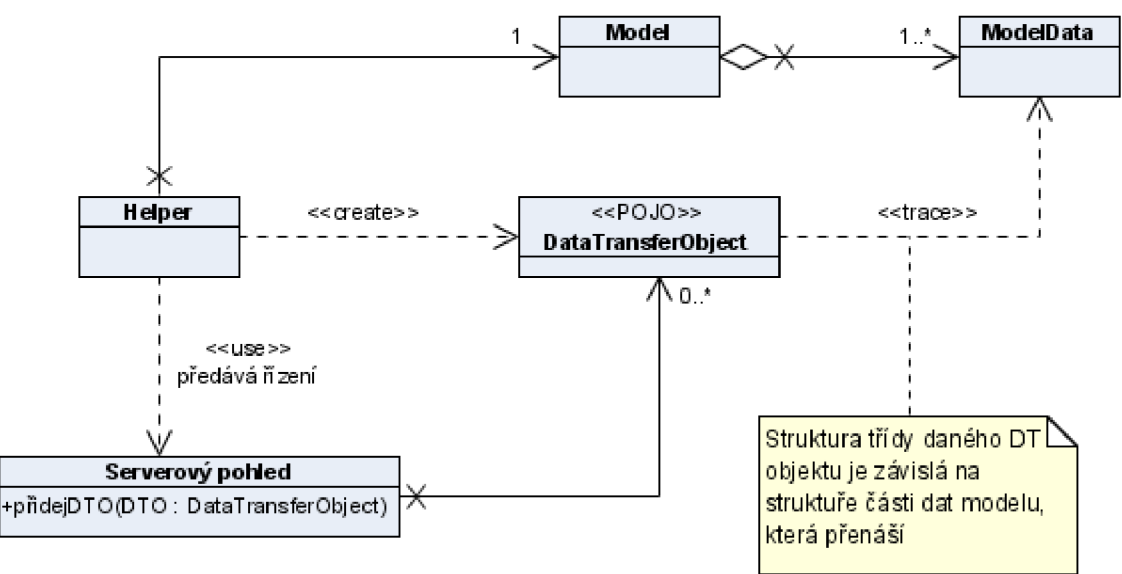
Vhodným řešením nastíněného problému se jeví obrácení komunikace, pohled nezískává data z modelu aktivně, ale jsou mu poskytnuta (*injected*) třetí stranou. Toto paradigma se v softwarovém inženýrství nazývá **Inversion of Control** [IOC].

Použití této obrácené komunikace má mj. za následek, že serverový pohled disponuje při sestavování odpovědi všemi potřebnými daty a nemusí již komunikovat s modelem, takto se vazba mezi těmito komponentami stane ještě volnější.

K návrhu prostředku, jakým dopravit data do pohledu, slouží vzor *Data Transfer Object*. Ten navrhuje pro transfer dat použít DT objekty mající určité vlastnosti:

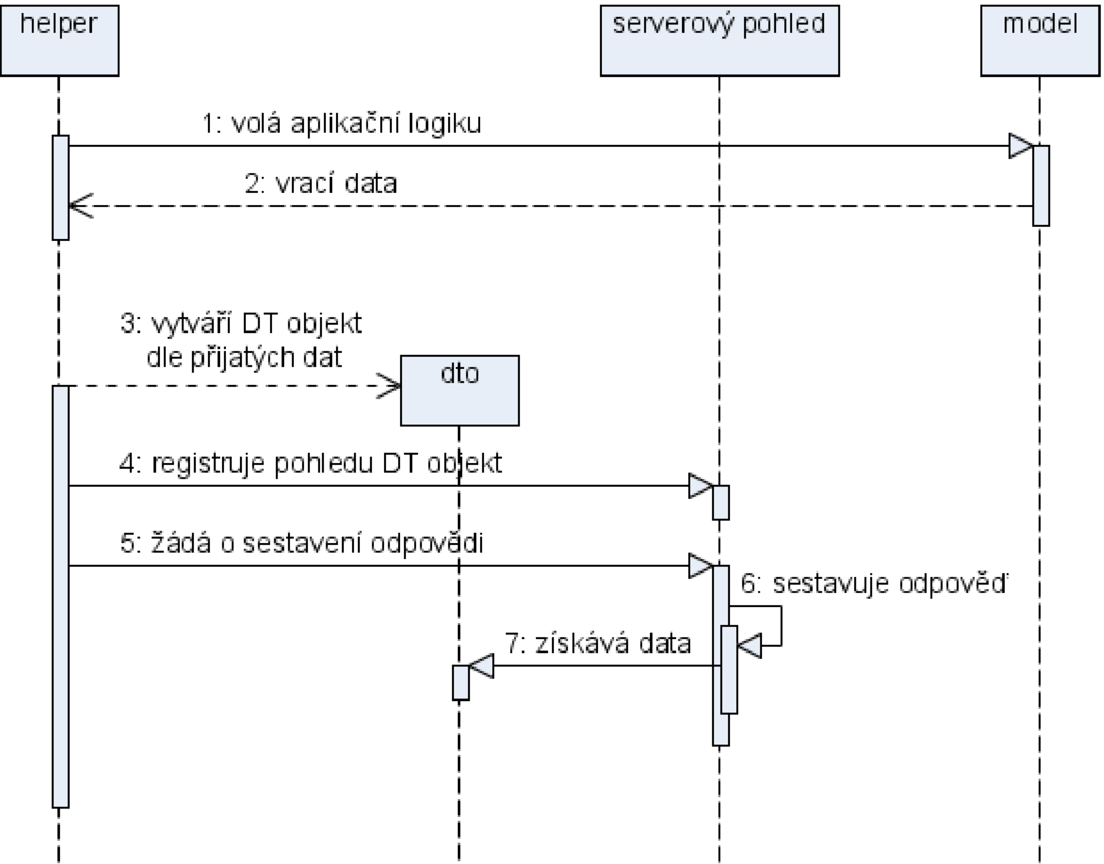
- Data uložená v DT objektu jsou duplikovaná a jsou určena pouze pro účely prezentace pohledem.
- Uložená data nemusí mít nutně stejnou strukturu, jakou mají uvnitř modelu. Informace o nějaké entitě nemusí být úplné, nejsou-li daným pohledem vyžadovány. Naopak v rámci jednoho DT objektu mohou být sdruženy informace z více entit, pokud k sobě logicky náleží.
- Pokud je povolena změna přenášených dat v rámci DT objektu, nemá žádný vliv na zdrojová data v modelu.
- Konkrétní technologie serverového pohledu musí poskytovat rozhraní pro injektáž DT objektů a jednoduchý přístup k nim.

Na vzoru *Data Transfer Object* tedy participují tyto třídy:



Obrázek 6.1: Diagram tříd statické struktury vzoru Data Transfer Object

Z dynamického hlediska pak komunikace mezi jednotlivými prvky probíhá následovně:



Obrázek 6.2: Sekvenční diagram vytvoření a předání DT objektu

6.4 Implementace

6.4.1 Prostředí JSP 2.0

Stránka JSP realizující konkrétní pohled může přistupovat k datům uloženým v různých *rozsazích* (*scope*), což jsou úložiště s definovanou dobou trvanlivosti dat. Např. objekty uložené ve *rozsahu request* jsou vymazány hned po dokončení zpracování požadavku, zatímco *rozsah session* zajišťuje dostupnost dat po celou dobu trvání konkrétního http sezení [JSJ, 311].

Helper komponenta zodpovědná za volání aplikační logiky vytvoří příslušné DT objekty, zaregistruje je do vhodných rozsahů a pak předá řízení vybrané JSP stránce. Ta si je pomocí zjednodušené syntaxe přebere a dále použije při generování pohledu pro klienta.

Následující příklad vychází z rámce *Struts*, který využívá JSP jako technologii serverových pohledů. Zjednodušená akce řadiče (viz 5.4.1.3) provede komunikaci s modelem, vytvoří DT objekt, zaregistruje jej do *rozsahu request* pod názvem `data1` a pak za pomoci rámce předá řízení zvolenému pohledu:

```
public class MyAction1 extends Action {

  private Model model;

  public ActionForward execute(
    ActionMapping mapping, ActionForm  form,
    HttpServletRequest request, HttpServletResponse response)
    throws Exception {
        //vytvoř DT objekt na základe dat z modelu
      UserDTO dto = new UserDTO(model.getUserInfo());
        //ulož je do rozsahu požadavku
      request.setAttribute("data1", dto);
        //vrací popis pohledu, kterému bude předáno řízení
    return mapping.findForward("pohled1");
  }
}
```

V JSP pohledu lze využít několik syntaktických konstruktů pro zjednodušené získání dat předaných skrz daný *rozsah*. V následující ukázce je využit jazyk EL[26]. Zápis `#{data1.surname}` zapříčiní, že JSP stránka bude prohledávat v určeném pořadí existující *rozsahy*, dokud nenarazí na objekt s identifikátorem `data1`. Následně zavolá jeho metodu `getSurname()` a výsledek vypíše do vytvářené odpovědi klientu:

```
…
  <tr>
    <th>Příjmení: </th>
    <td>#{data1.surname}</td>
  </tr>
…
```

[26] Jazyk EL představený v JSP 2.0 umožňuje zjednodušený netypový přístup k datům uloženým v různých rozsazích, provádění logiky nad nimi a výpis do těla odpovědi. [JE5, 107] Již od JSP 1.0 lze využít těžkopádnějších instrukcí `jsp:useBean`, `jsp:setProperty` a `jsp:getProperty`. [JSP, 166]

6.4.2 Zend Framework 1.0.0

Tento rámec pro prostředí PHP standardně využívá třídu `Zend_View` jako obálku serverového pohledu (viz 5.4.2.3). Ta poskytuje metodu `assign()`, která převezme libovolná data a zaregistruje je pohledu pro další zpracování.

I v tomto případě *helper* komponenta vytvoří DT objekt na základě získaných dat, předá jej pohledu a přepošle mu řízení:

```
…
public class Helper1 extends Zend_Controller_Action {
  private $model;

  public function action1 {
      //vytvoř DT objekt na základě dat z modelu
    $dto = new UserDTO($model -> getUserInfo());
    $view = new Zend_View();
      //předání dat pohledu
    $view -> assign('data1', $dto);
    $view -> render('view.phtml');
  }
}
```

Ve vybraném skriptu *view.phtml* lze pak k objektu s daty přistupovat jednoduchým způsobem:

```
…
<table>
  <tr>
    <td>Příjmení: </td>
    <td><?php echo $this -> data1 -> getSurname(); ?></td>
  </tr>
…
```

7 Vzor Intercepting Filter

7.1 Účel

Umožňuje přidávat webové aplikaci další zodpovědnosti při zpracování http komunikace ve formě konfigurovatelného řetězce setříděných modulů.

7.2 Problém

Každá webová aplikace obecně zpracovává http požadavky obdržené od klienta a odesílá mu data v podobě http odpovědí. Problém nastane, pokud ji potřebujeme obohatit o nízkoúrovňovou funkčnost, která pracuje s oběma fázemi http komunikace, zvláště pak využíváme-li služeb některého hotového MVC rámce. V takovém případě musíme zasáhnout do aplikace na více místech (do řadiče i serverového pohledu). To může mít za následek větší nepřehlednost a narušení její struktury.

7.3 Řešení

Pro účely dalšího zpracování http komunikace je vhodné zavést do systému komponentu, která se umístí *před* celou aplikaci. Touto komponentou pak protékají jak http požadavky proudící k aplikaci (k řadiči), tak odpovědi zpět ke klientu (vygenerované serverovým pohledem). Měla by být maximálně modulární, tj. co nejjednodušším způsobem umožňovat konfiguraci (přidávání, odebírání) modulů obsahujících určitou funkčnost. K této příležitosti lze využít kombinaci dvou klasických návrhových vzorů *Decorator*[27] a *Chain of Responsibility*[28].

Prvně jmenovaný popisuje, jak dodávat konkrétnímu objektu funkce pomocí *dekorací* sdílejících s ním totožné rozhraní. Klient pak komunikuje s vnější *dekorací*, která vykoná svou část operace a předá řízení další *dekoraci* v řetězci, resp. přímo *dekorovanému* objektu. Zároveň je možno tento řetězec konfigurovat za běhu.

Vzor chování *Chain of Responsibility* přináší možnost zpracovávat určitou žádost pomocí více objektů usazených v řetězcové struktuře. Žádost postupně prochází jednotlivými objekty, dokud ji některý z nich nezpracuje.

Uvedené vlastnosti dohromady popisují, co požadujeme od vzoru *Intercepting Filter*:

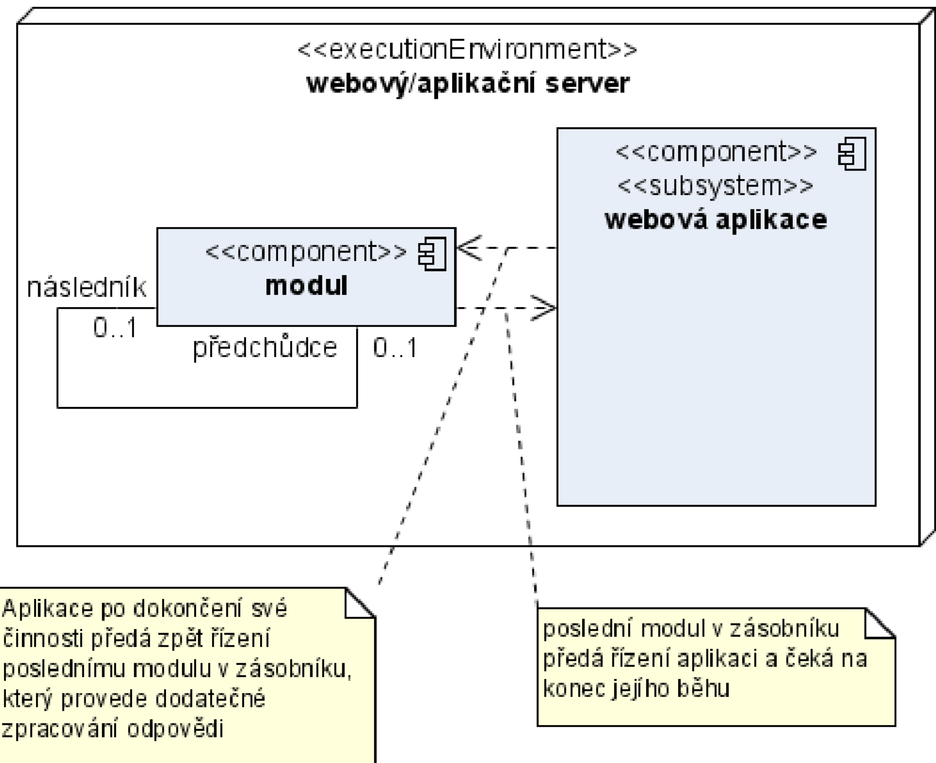
- Možnost přidávat další funkčnost pro zpracování http komunikace ve formě řetězce jednoduchých modulů.
- Jednoduchou konfiguraci tohoto řetězce, např. pomocí zvláštního konfiguračního souboru.
- Každý modul se dle naprogramované logiky může rozhodnout, zdali předá komunikaci dále, což znamená, že v určitém případě se http požadavek vůbec nemusí dostat k samotné aplikaci.
- Klient nemá a nepotřebuje znalost přiřazených modulů, s aplikací komunikuje stále stejným způsobem.

---

[27] *„Dynamicky připojí k objektu další povinnosti. Dekorátoři poskytují při rozšiřování funkcí tvárnou alternativu k tvorbě podtříd.“* [GoF, 176]

[28] *„Vyhýbá se spojení odesilatele žádosti s příjemcem tím, že umožňuje více než jednomu objektu žádost zpracovat. Zřetězí objekty příjemců a předává prostřednictvím řetězu žádost, až ji nějaký objekt zpracuje.“* [GoF, 219]

Namísto definování zvláštního řetězce pro každý směr komunikace budeme požadovat, aby každý modul obsahoval funkčnost pro oba dva tyto směry. Touto obousměrností se vzor *Intercepting Filter* liší od výše uvedených klasických návrhových vzorů:



Obrázek 7.1: Diagram komponent vzoru Intercepting Filter

Návrhový vzor *Intercepting Filter* tedy popisuje moduly seřazené do řetězce, ve kterém každý modul zná svého následníka a předchůdce. Když na server dorazí http požadavek, převezme jej první modul, předzpracuje a odešle svému následníku. Takto se pokračuje až do posledního prvku řetězce, který předá komunikaci aplikaci. Http odpověď obsahující popis pohledu projde dodatečným zpracováním moduly v opačném pořadí. Jsou tedy seřazeny v zásobníkové LIFO struktuře. Z uvedeného diagramu také plyne, že v daném běhovém prostředí musí existovat možnost pro přidávání komponent *před* aplikací.

7.4 Implementace

7.4.1 Prostředí Java EE

V prostředí *Java EE* existuje možnost deklarativně v hlavním konfiguračním souboru *web.xml* přiřadit aplikaci tzv. filtry. Obecně jde o třídy implementující rozhraní `Filter`, které obvykle mají následující strukturu [JE5, 280]:

```
public class MyFilter implements Filter {
…
  private void doBeforeProcessing(
    ServletRequest request, ServletResponse response)
    throws IOException, ServletException {
```

```
    //akce provedená před předáním požadavku dalšímu filtru/aplikaci
  }

  private void doAfterProcessing(
    ServletRequest request, ServletResponse response)
    throws IOException, ServletException {

    //akce provedená po odeslání odpovědi aplikací
  }

  public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    doBeforeProcessing(request, response);
      chain.doFilter(request, response);
    doAfterProcessing(request, response);
  }

  public void destroy() {…}

  public void init(FilterConfig filterConfig) {…}
}
```

Z rozhraní `Filter` pocházejí metody `init()`, `destroy()` a nejdůležitější `doFilter()`, do které se implementuje samotná funkčnost filtru. V uvedeném obecném příkladě se nejdříve zavolá metoda `doBeforeProcessing()` obsahující kód předzpracovávající http požadavek klienta a poté se předá řízení následujícímu filtru v řetězci. Takto se postupně dostanou ke slovu všechny filtry a nakonec aplikace. Metody filtrující odpověď (zde `doAfterProcessing()`) se pak volají v opačném pořadí. Pokud je v některém filtru vynecháno volání `chain.doFilter()`, zpracování zde končí a další komponenty řetězce včetně samotné aplikace jsou vynechány.

Máme-li hotové třídy implementující uvedené rozhraní, můžeme je přiřadit dané aplikaci a nakonfigurovat jejich pořadí v již zmíněném souboru *web.xml*:

```
…
<filter>
  <filter-name>logFilter</filter-name>
  <filter-class>filters.LogFilter</filter-class>
</filter>
<filter>
  <filter-name>cacheFilter</filter-name>
  <filter-class>filters.CacheFilter</filter-class>
</filter>

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

<filter-mapping>
  <filter-name>cacheFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
…
```

V tomto případě máme definovány dva filtry, které jsou instancemi tříd `LogFilter` a `CacheFilter`. Oba dva se budou aplikovat při každém http požadavku na aplikaci díky mapování

v elementu `url-pattern`. Jejich pořadí v řetězci je určeno posloupností příslušných elementů `filter-mapping`.

Toto řešení vzoru *Intercepting Filter* lze využít pro libovolné aplikace či rámce, které jsou postaveny nad platformou *Java EE*.

7.4.2 Rámec Struts2

Tento následník[29] původních *Struts* je zajímavý tím, že ač je postaven nad technologií *Java EE*, volí vlastní řešení [SS2, 25] pro realizaci vzoru *Intercepting Filter* nezávislé na výše uvedené technologii filtrů [30]. Místo nich používá tzv. *interceptory*, což jsou z hlediska Javy instance tříd implementujících rozhraní `Interceptor`:

```
public interface Interceptor extends Serializable {
  void destroy();
  void init();
  String intercept(ActionInvocation invocation) throws Exception;
}
```

Toto rozhraní je velmi podobné zmíněnému rozhraní `Filter` a implementuje se totožným způsobem, v metodě `intercept()` je umístěn příslušný kód, následující prvek řetězce se aktivuje voláním `invocation.invoke()`.

Jednotlivé *interceptory*, resp. jejich zásobníky se pak aplikaci přiřazují v konfiguračním souboru. *Struts2* také obsahuje množství předpřipravených *interceptorů* obsahujících samotnou funkčnost rámce. Jde např. o zpracování výjimek a chyb validace, *upload* souborů, internacionalizaci a další. Následující příklad představuje fragment konfiguračního souboru definujícího dva nové *interceptory* a jeden zásobník:

```
…
<package … >

  <interceptors>
    <interceptor name="myInt1" class="interceptors.MyInterceptor1"/>
    <interceptor name="myInt2" class="interceptors.MyInterceptor2"/>
  </interceptors>

  <interceptor-stack name="customStack">
    <interceptor-ref name="basicStack"/>
    <interceptor-ref name="myInt1"/>
    <interceptor-ref name="myInt2"/>
  </interceptor-stack>

</package>
…
```

V tomto případě je vytvořen zásobník s názvem `customStack` obsahující v sobě základní zásobník `basicStack` dodávaný přímo s rámcem a dva dodefinované *interceptory* `myInt1` a `myInt2`. I zde platí, že budou setříděny podle pořadí svých elementů `interceptor-ref`. Takto vytvořeným zásobníkem pak lze *dekorovat* celou aplikaci nebo její logické části.

---

[29] http://struts.apache.org/2.x/

[30] Další zajímavostí je, že řadič není implementován pomocí *Java EE* servletu jako u jiných webových rámců postavených nad touto technologií (viz např. 5.4.1 nebo 5.4.3), ale jako jeden filtr [SS2, 14].

7.4.3 Rámec ASP.NET 2.0

Jak již bylo řečeno v kapitole 5.4.4, každý požadavek zpracovávaný rámcem ASP.NET prochází přiřazeným http *handlerem*. Zároveň však může být ještě předzpracován několika http moduly [ASC, 228], což jsou třídy implementující rozhraní `IHttpModule`, které vyžaduje metody `Init()` a ukončovací `Dispose()`. Úlohou prvně jmenované je přiřadit aplikaci dodatečnou obsluhu událostí, které vznikají při prvotním zpracování http požadavku, resp. při odesílání odpovědi. Pro implementaci vzoru *Intercepting Filter* pak lze využít např. tyto události:

- `BeginRequest` – http požadavek dorazil do rámce ke zpracování..
- `PreRequestHandlerExecute` – událost signalizující, že v následujícím kroku bude spuštěn příslušný http *handler*.
- `PostRequestHandlerExecute` – činnost http *handleru* byla ukončena.
- `EndRequest` – http odpověď bude odeslána klientu.

Dvě prvně zmíněné události se hodí pro práci s požadavkem směrem k aplikaci, zbývající pak pro dodatečné zpracování odpovědi. V následujícím příkladě definujeme modul obsahující metody `preProcess()` a `postProcess()`, které budou spuštěny před předáním řízení http *handleru*, resp. po ukončení jeho činnosti:

```
public class MyModule: IHttpModule {
  public void Dispose() {/* úklid po daném modulu */}

  public void Init(HttpApplication application) {
    application.PreRequestHandlerExecute +=
      new EventHandler(this.preProcess);

    application.PostRequestHandlerExecute +=
      new EventHandler(this.postProcess);
  }

  public void preProcess(object source, EventArgs eventArgs) {
    //kód spuštěný před předáním řízení http handleru
  }

  public void postProcess(object source, EventArgs eventArgs) {
    //kód spuštěný po ukončení práce http handleru
  }
```

Pokud se některý z modulů rozhodne, že se požadavek nemá předávat samotné aplikaci, má možnost zavolat metodu `CompleteRequest()` definovanou ve třídě `HttpApplication`.

Takto lze vytvořit více modulů, každý s oddělenou funkcionalitou. Webové aplikaci je pak přiřadíme v jejím konfiguračním souboru:

```
…
<system.web>
  <httpModules>
    <add name="Module1" type="MyModule">
    …
  <httpModules>
</system.web>
…
```

8 Vzor View Helper

8.1 Účel

Minimalizuje množství kódu realizujícího v pohledu prezentační logiku pro adaptaci dat tím, že ji zapouzdřuje do jednoduše přístupných komponent. Zároveň může poskytovat generická data, takže pohled lze vyvíjet nezávisle na modelu.

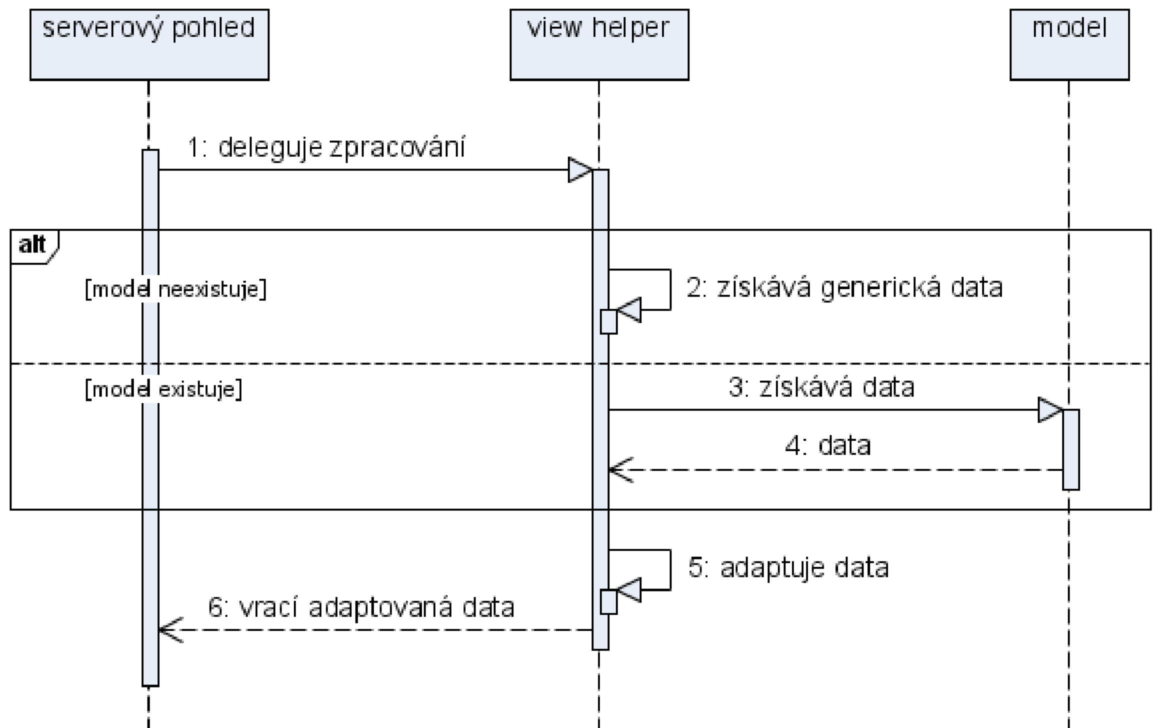
8.2 Problém

Technologie používané pro implementaci serverových pohledů často využívají jazyky se speciální zjednodušenou syntaxí (občas nazývané jako *šablonovací* jazyky), které mají odstínit vývojáře těchto prvků od využívání plnohodnotného jazyka. Tato syntaxe obvykle umožňuje přejímat data z modelu a vypisovat je do výstupního dokumentu a poskytuje základní konstrukce pro realizaci jednoduché logiky (iterování nad daty, podmíněný příkaz apod.).

Formátování surových dat získaných z modelu do požadované podoby však může být natolik složité, že na něj uvedené syntaktické prostředky nestačí. V tomto případě musí vývojáři sáhnout zpět po silnějším jazyku.

8.3 Řešení

Řešení nastíněného problému představuje zapouzdření adaptace dat modelu do komponent, které jsou z dané technologie serverového pohledu jednoduše přístupné, např. pomocí zvláštních syntaktických prostředků. Pohled je pak využívá namísto toho, aby se pokoušel sám o realizaci prezentační logiky. Tyto prvky se nazývají **view helper** a jejich úkol obecně spočívá v převzetí dat z modelu, adaptaci pro účely prezentace a jejich navrácení pohledu. Také mohou obsahovat další zodpovědnost. Pokud dle určité logiky zjistí, že model zatím neexistuje, může pohledu nabídnout vlastní generická data. Ta samozřejmě musí sdílet strukturu s daty modelu:



Obrázek 8.1: Sekvenční diagram zpracování dat modelu pomocí view helper komponenty

Vývojáři vrstvy pohledu nejsou v tomto případě nijak vázáni existencí modelu. Je ale nutno dbát na synchronizaci struktury dat v modelu a *view helper* komponentě, což nemusí být obecně jednoduché[31].

8.4 Implementace

8.4.1 Prostředí JSP

Technologii JSP lze využít k implementaci serverových pohledů v rámci platformy *Java EE*. Daná JSP stránka se obvykle skládá z kombinace HTML elementů a speciálních syntaktických prvků. Ty lze využít pro přístup k datům a jejich vypsání do těla odesílaného dokumentu, ale při realizaci složitější adaptační logiky již mohou být nedostačující. V tomto případě lze sáhnout po *scriptletech*, což jsou kusy kódu napsaného přímo v jazyce *Java* vložené do bloku ohraničeného `<% %>`. Této situaci se však chceme vyhnout, k čemuž nám specifikace JSP od verze 1.1 nabízí tzv. *Custom Tags*.

Tato technologie umožňuje definovat vlastní XML elementy, které představují rozhraní pro přístup ke kódu zapouzdřenému v třídách dodržujících předepsaná pravidla [JSP, 287]. Z toho vyplývá, že daný prostředek je v JSP přímo určen pro implementaci komponent *view helper*.

Následující příklad demonstruje vytvoření takové jednoduché vzorové komponenty. Její úlohou je pro účely prezentace zaokrouhlit a zarovnat číselné hodnoty získané z modelu na daný počet desetinných míst. Pokud jí není předána žádná hodnota, nahradí ji vlastní, generickou. Nejdříve vytvoříme *tag* implementující tuto funkcionalitu:

---

[31] Jiný přístup eliminující naznačený problém představuje uložení generických dat jako součást modelu. V tomto případě je synchronizace mnohem jednodušší, pohled však musí mít přístup k modelu hned od začátku vývoje. [VHP]

```java
public class NumberHelper extends BodyTagSupport {

  //počet desetinných míst
  private int fractionSize;
…

  public int doAfterBody() throws JspException {
    BodyContent bc = this.getBodyContent();
    JspWriter out = bc.getEnclosingWriter();

    double value;
    //převezmi řetězec z těla tohoto tagu
    String strValue = bc.getString().trim();

    if (strValue.length() == 0) {
      //nebyla dodána žádná hodnota, použij generickou
      value = 13.5678;
    }
    else {
      try {
        value = Double.parseDouble(strValue);
      }
      catch (NumberFormatException e) {…}
    }

    //naformátuj hodnotu a vypiš ji do dokumentu
    DecimalFormat formatter = new DecimalFormat();
    format.setMinimumFractionDigits(this.fractionSize);
    format.setMaximumFractionDigits(this.fractionSize);
    try {
      out.println(formatter.format(value));
    }
    catch (IOException e) {…}
  }

  //převezme z atributu elementu fractionSize požadovaný počet
  //desetinných míst
  public void setFractionSize(int value)
  {
    this.fractionSize = value;
  }
}
```

Takto zapouzdřenou funkčnost pak můžeme jednoduše využívat v JSP dokumentech:

```jsp
<%-- zaregistruj knihovnu obsahující použitý tag --%>
<%@ taglib uri="/WEB-INF/tlds/helpers" prefix="helpers" %>
…
<table>
  <tr>
    <td>hodnota #1</td>
    <td><helpers:NumberHelper>
      ${model.value1}
    </helpers:NumberHelper></td>
  </tr>
  <tr>
    <td>hodnota #2</td>
    <td><helpers:NumberHelper fractionSize="2">
      ${model.value2}
    </helpers:NumberHelper></td>
  </tr>
```

```
</table>
…
```

V prvním případě je hodnota vypsána s implicitním počtem desetinných míst, v druhém o počtu míst rozhoduje nepovinný atribut `fractionSize`. V reálném příkladě by daná komponenta pochopitelně implementovala složitější adaptační funkčnost.

8.4.2 Knihovna JSTL

Při psaní aplikací s pomocí JSP lze využít různé předpřipravené knihovny *tagů*. Jednou z nich je i knihovna JSTL, jejíž specifikace je přímo součástí prostředí *Java EE*. Obsahuje mj. podknihovnu *Internationalization-capable formatting* sdružující *tagy* určené pro formátování dat s ohledem na specifika národního prostředí [JE5, 158]. Elementy `formatDate` a `formatNumber` představují komplexní *view helper* komponenty pro adaptaci čísel a datumů. Následující příklad demonstruje řešení výše uvedeného problému pomocí knihovny JSTL:

```
<%-- zaregistruj knihovnu JSTL formatting  --%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
…
<tr>
  <td>hodnota #2</td>
  <td>
    <fmt:formatNumber value="${model.value2}"
      maxFractionDigits="2" minFractionDigits="2" />
  </td>
</tr>
…
```

Jak v předchozím, tak i v tomto případě se povedlo pomocí komponent napsaných dle vzoru *View Helper* odstranit z pohledu kód v jazyce *Java* ve prospěch alternativní syntaxe ve formě XML elementů. Kód dané JSP stránky je tak přehlednější a lépe spravovatelný.

8.4.3 Zend Framework 1.0.0

Třída `Zend_View` (viz 5.4.2.3) plnící funkci obálky serverového pohledu poskytuje možnost implementace *view helper* komponent. Ty jsou zde reprezentovány obyčejnou PHP třídou, která musí dodržovat určité pojmenovací konvence, především metoda obsahující samotnou adaptační logiku odvozuje svůj identifikátor od názvu celé třídy. V pohledu pak *view helper* použijeme pomocí zjednodušené syntaxe `$this -> helperName();`. Na pozadí se vytvoří instance třídy, případně se použije již dříve vytvořená, a zavolá se její adaptační metoda. Od všech těchto úkonů je uživatel kompletně odstíněn.

Následující *view helper* implementuje stejnou funkčnost jako v předešlých příkladech:

```
<?php

public class View_Helper_NumberHelper {
  function numberHelper($value, $fractionSize = 2) {
    //kontrola parametru metody
…
    //vytvoření formátovacího řetězce
    $fstr = '%01.' . $fractionSize . 'f';
    return sprintf($fstr, $value);
  }
}
```

V pohledu zapouzdřeném v instanci `Zend_View` jej pak můžeme využívat již nastíněným způsobem:

```
…
<tr>
  <td>hodnota #1</td>
  <td>
    <?php $this -> numberHelper($this -> modelValue); ?>
  </td>
</tr>
…
```

I v tomto případě se tedy povedlo pohled odstínit od složitějšího kódu a zapouzdřit adaptační funkčnost do jednoduše přístupné komponenty.

9 Vzor Composite View

9.1 Účel

Umožňuje sestavit výslednou http odpověď za pomoci více serverových sub-pohledů umístěných do stromové struktury, která podporuje znovupoužitelnost jednotlivých uzlů. Zároveň uvádí mechanismus pro oddělení definice obsahu stránky od jeho rozložení na ní.

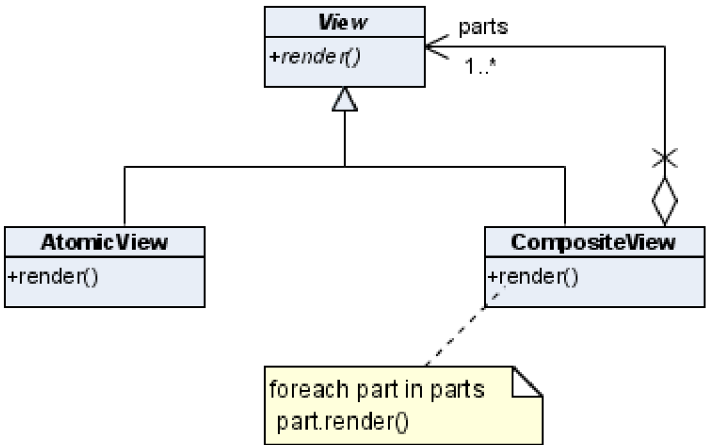
9.2 Problém

Uživatelské rozhraní webových aplikací je nejčastěji tvořeno množinou HTML dokumentů obvykle obsahujících společné prvky jako záhlaví, zápatí, přihlašovací nebo vyhledávací formuláře apod. Pokud tyto části vložíme do kódu každé webové stránky aplikace, bude v nich obtížné provádět změny, protože je budeme muset realizovat ve všech duplikátech.

Tyto prvky můžeme umístit do různých souborů a skrze platformou nabízeného *include* mechanismu vkládat do hlavního pohledu. Vyřešíme tím sice otázku jejich znovupoužitelnosti, ale vyvstane nový problém. Takový hlavní pohled bude obsahovat jak definici obsahu, tak i rozložení. Tím pádem nebude možno vytvářet generické definice rozložení použitelné ve více aplikacích a nebudeme mít k dispozici jednoduchý mechanismus, jak zaměňovat v rámci celé aplikace jednotlivé sub-pohledy za jiné.

9.3 Řešení

Pro řešení uvedeného problému lze s úspěchem vyjít z návrhového vzoru *Composite*[32]. Ten umožňuje přistupovat k jednoduchým i složeným prvkům pomocí stejného rozhraní. Složený prvek se pak skládá z částí, které obecně mohou být opět složené. Aplikován na serverové pohledy je kategorizuje do následujících skupin:



Obrázek 9.1: Diagram tříd popisující rozdělení pohledů dle návrhového vzoru Composite

[32] *„Skládá objekty do stromových struktur k vyjádření hierarchií typu objekt – celek. Skladba umožňuje klientům jednotně zacházet s jednotlivými objekty i skladbami objektů.“* [GoF, 166]

9.3.2 Pohled AtomicView

Pohled `AtomicView` představuje sub-pohled realizující nějakou část výsledné stránky. Platí pro něj, že je již dále nedělitelný a není závislý na žádném dalším pohledu. Může jít např. o jednoduchý formulář pro fulltextové vyhledávání:

```
<form method="get" action="/search">
  <input type="text" id="fulltextSearch" />
  <input type="submit" value="Hledej" />
</form>
```

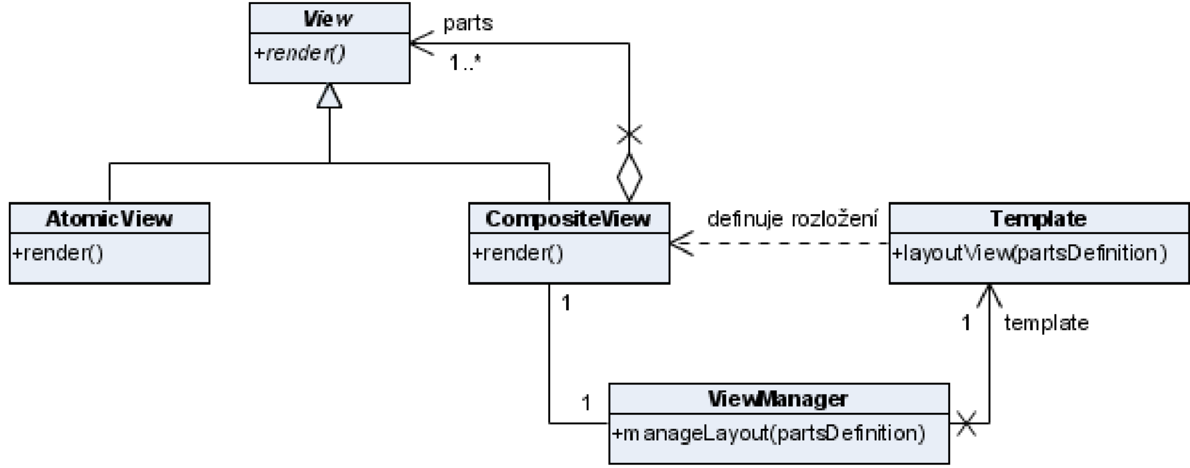
9.3.3 Pohled CompositeView

Naproti tomu se `CompositeView` skládá z více sub-pohledů. Může být popsán různými mechanismy, které umožní definovat jeho jednotlivé části a přiřadit jim jednoznačné identifikátory. V následujícím příkladě je použita forma XML dokumentu:

```
<compositeViewDefinition>
  <subView id="header" src="/header.htm">
  <subView id="footer" src="/footer.htm">
  <subView id="search" src="/search.htm">
  <subView id="content" src="/content.xml">
</compositeViewDefinition>
```

Takto jsme ukázkově vytvořili složený pohled, který obsahuje tři atomické sub-pohledy s identifikátory `header`, `footer` a `search`, a jeden další složený sub-pohled `content`, jehož obsah je definován v souboru *content.xml*.

Jak je vidět, `CompositeView` pouze popisuje, z jakých částí se skládá, ale už nedefinuje jejich rozložení na stránce. Proto musíme výše uvedený diagram doplnit o další prvky:



Obrázek 9.2: Diagram tříd vzoru Composite View

9.3.4 Definice rozložení Template

Šablona představuje obecnou znovupoužitelnou definici rozložení [WAC]. Obsahuje HTML kód společně se speciálními elementy, které se při zpracování nahradí jednotlivými částmi složeného pohledu. Chce-li takový pohled využít určitou šablonu, musí ji k sobě přiřadit ve svém těle. Následující fragment kódu rozšiřuje výše uvedenou XML definici obsahu o atribut odkazující na konkrétní šablonu:

```
<compositeViewDefinition template="/mainTemplate.tpl">
  <subView id="header" src="/header.htm">
…
```

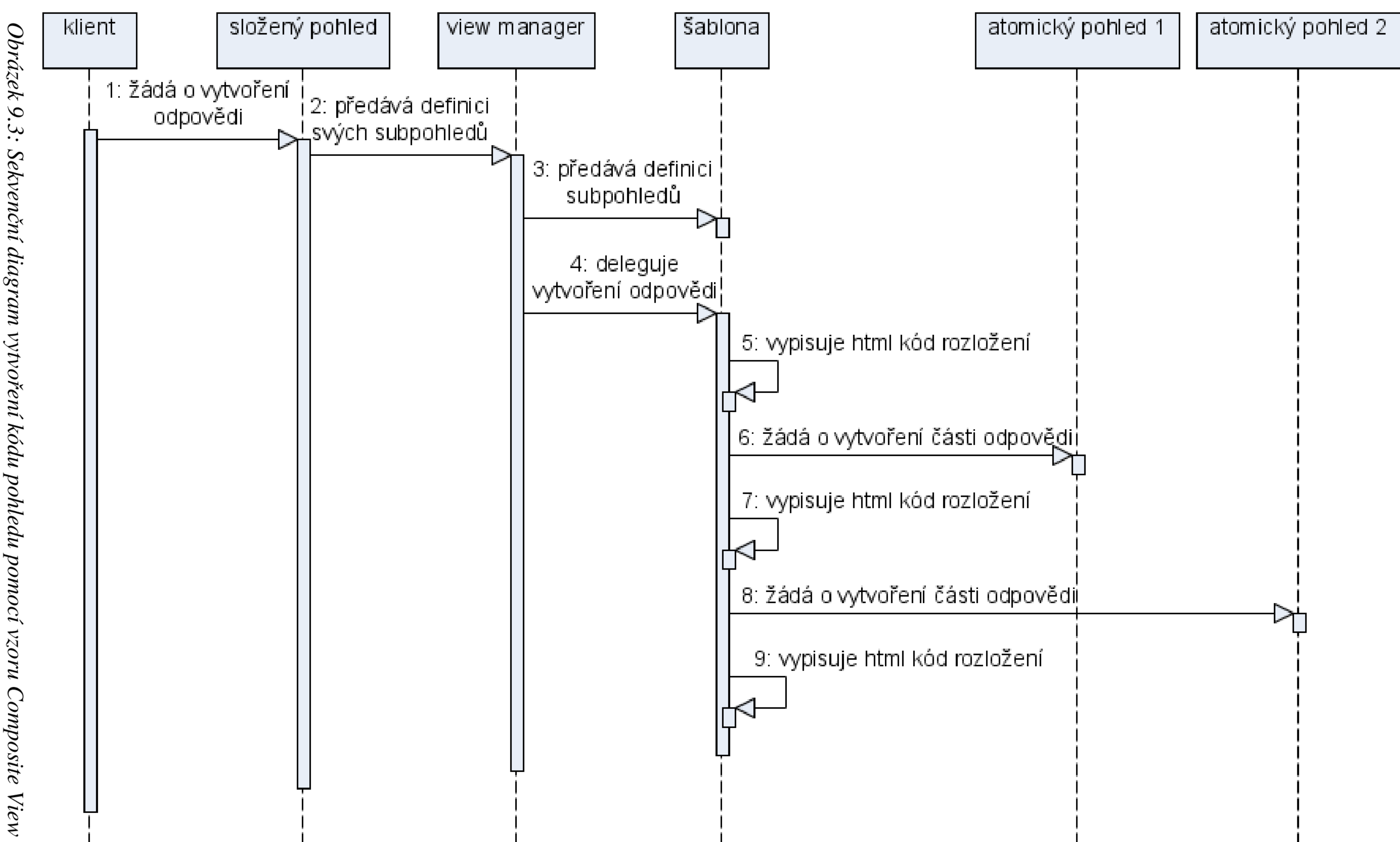
Daná šablona pak může vypadat např. následovně:

```
…
<body>
  <div>
    <template:insert viewId="header" />
    <div id="leftColumn">
      <template:insert viewId="content" />
    </div>

    <div id="rightColumn">
      <template:insert viewId="search" />
    </div>
    <template:insert viewId="footer" />
  <div/>
</body>
…
```

9.3.5 Správce vykreslení ViewManager

Složený pohled využívá komponentu *view manager* k vykreslení svého obsahu pomocí šablony tak, že jí předá informace o svých jednotlivých sub-pohledech (identifikátor a cestu k nim). *View manager* předá tyto informace šabloně, která má v tuto chvíli dostatek informací k tomu, aby se mohla začít vykonávat:



Obrázek 9.3: Sekvenční diagram vytvoření kódu pohledu pomocí vzoru Composite View

Hlavní výhoda takového přístupu především spočívá v pouze volné provázanosti mezi složeným pohledem a jeho šablonou. Šablony jsou znovupoužitelné ve více projektech a lze tak vytvářet celé jejich knihovny. Složený pohled pak může dynamicky měnit své rozložení pouhou volbou nové šablony. Takto je např. možno odeslat uživateli pohled s rozložením dle jeho preferencí.

9.4 Implementace

9.4.1 Knihovna Tiles

Knihovna *Tiles* dodávaná společně s webovým rámcem *Struts* představuje šablonovací řešení pro platformu *Java EE* a je postavena na základě technologie JSP. V její terminologii se stránka skládá z jedné nebo více *dlaždic* (*tiles*), které mohou být zaplněny jednoduchými JSP stránkami nebo opět složenými sub-pohledy. Ke stránce složené z více *dlaždic* je přiřazena šablona, zde nazývaná *rozložení* (*layout*) [PJS, 338].

V následujícím příkladě je stránka *index.jsp* složena z více částí, atomických `header` a `footer` a složené `content`. Předepisuje jim konkrétní soubory obsahující jejich data:

```
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>

<tiles:insert page="/WEB-INF/layout/indexLayout.jsp">
  <tiles:put name="header" value="/WEB-INF/pages/header.jsp"/>
  <tiles:put name="footer" value="/WEB-INF/pages/footer.jsp"/>
  <tiles:put name="content" value="/WEB-INF/pages/content.jsp"/>
</tiles:insert>
```

Zároveň je v jejím těle uveden soubor *indexLayout.jsp* definující rozložení:

```
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>
<html>
…
  <body>
    <div id="headerDiv"><tiles:insert attribute="header"/></div>
    <div id="contentDiv"><tiles:insert attribute="content"/></div>
    <div id="footerDiv"><tiles:insert attribute="footer"/></div>
  </body>
</html>
```

Soubory *header.jsp* a *footer.jsp* zaregistrované pod identifikátory `header`, resp. `footer` jsou atomickými sub-pohledy obsahujícími data pro hlavičku a patičku stránky, zatímco *content.jsp* je opět složenou stránkou (tentokrát ze dvou *dlaždic*) s vlastním rozložením:

```
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>

<tiles:insert page="/WEB-INF/layout/contentLayout.jsp">
  <tiles:put name="content1" value="/WEB-INF/pages/content1.jsp"/>
  <tiles:put name="content2" value="/WEB-INF/pages/content2.jsp"/>
</tiles:insert>
```

Samotný element `tiles:insert` uváděný v těle složených pohledů představuje *view manager* prvek, který předá šabloně definici rozkládaných dat.

9.4.2 ASP.NET 2.0

Rámec ASP.NET 2.0 s sebou přináší vlastní šablonovací mechanismus vycházející ze vzoru *Composite View* nazvaný *Master Pages* [ASC, 610]. Jedna *Master Page* představuje v jeho terminologii šablonu, která kromě všech prvků použitelných na klasických ASP.NET stránkách navíc nabízí element `asp:contentplaceholder` definující místo, kam se bude vkládat obsah:

```
<%@ Master Language="C#" %>
…
  <body>
    <div id="headerDiv">
      <asp:contentplaceholder id="header" runat="server" />
    </div>
    <div id="contentDiv">
      <asp:contentplaceholder id="content" runat="server" />
    </div>
    <div id="footerDiv">
      <asp:contentplaceholder id="footer" runat="server" />
    </div>
  </body>
…
</html>
```

Narozdíl od implementace vzoru *Composite View* v technologii *Tiles* jsou v tomto případě atomické sub-pohledy využívané v dané šabloně vloženy přímo do složeného pohledu pomocí elementů `asp:content`:

```
<% @ Page Language="C#" MasterPageFile="~/Master.master" %>

<asp:content ContentPlaceHolderID="header" …>
  <!-- kód hlavičky stránky -->
</asp:Content>

<asp:content ContentPlaceHolderID="content" …>
   <!-- kód obsahu stránky -->
</asp:content>

<asp:content ContentPlaceHolderID="footer" …>
   <!-- kód patičky stránky -->
</asp:content>
```

Samotný složený pohled si tedy určuje šablonu, která se na něj má použít (zde pomocí atributu `MasterPageFile`), a ve svém těle obsahuje jednotlivé sub-pohledy. To je rozdíl oproti *Tiles*, kde jsou sub-pohledy samostatné a navíc mohou být opět složené.

10 Návrh ukázkové aplikace

K demonstraci použití a přínosu popsaných vzorů jsem vyvinul ukázkovou webovou aplikaci inspirovanou školními systémy pro evidenci projektů a řešitelských týmů. Její základní funkce jsou shrnuty v následujícím seznamu:

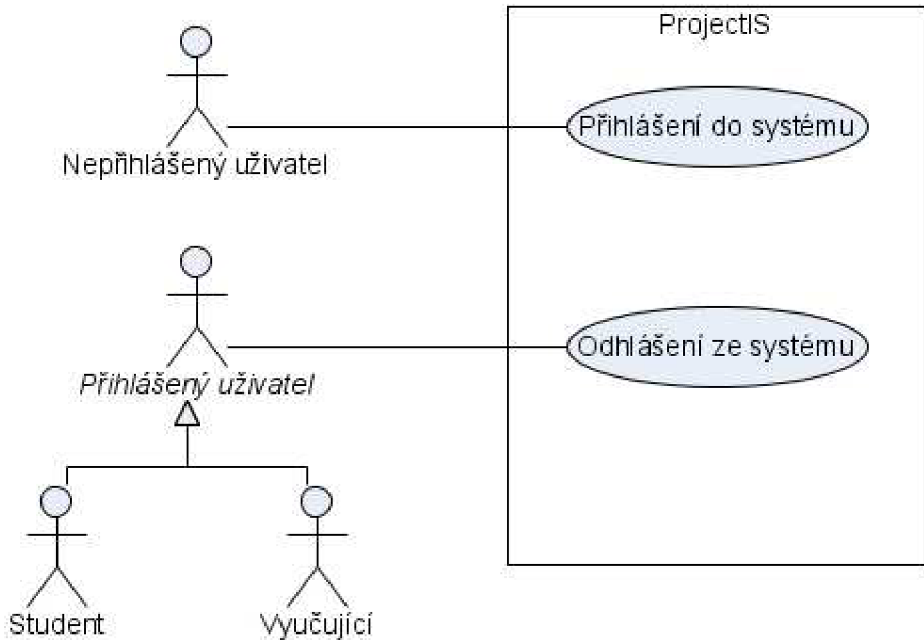
- S aplikací pracují studenti a vyučující, kteří se přihlašují pod svým uživatelským jménem a heslem.
- Vyučující může k předmětům, které vyučuje, přidávat nové projekty, modifikovat je a mazat.
- Vyučující může k vypsaným projektům přiřazovat nové varianty, modifikovat je a mazat. Při vytváření varianty deklaruje, kolik ji může řešit týmů a jejich kapacitu.
- Vyučující může hodnotit celý tým dohromady nebo každého studenta zvlášť určitým počtem bodů.
- Student se může přihlásit do týmu dané varianty, příp. se z ní odhlásit. Byl-li již hodnocen, zobrazí se mu i tento údaj.
- Aplikace neřeší otázku získávání seznamu předmětů, vyučujících, studentů a jejich vazeb. Obsahuje však akci, která databázi naplní ukázkovými hodnotami.

Jde tedy o typickou CRUD webovou aplikaci, na kterou lze dobře a přehledně aplikovat myšlenky obsažené v uvedených vzorech. To bylo hlavní motivací při vymýšlení jejího zaměření.

V následujícím textu uvádím stručný model požadavků popsaný slovně a pomocí *Use Case* diagramů. Tato část nemá za úkol detailně rozebrat funkčnost celé aplikace, což ani není cílem diplomové práce, ale spíše ji nastínit natolik, aby byl srozumitelný následný text. V tom se totiž zaměřím na jednotlivé vzory z hlediska přínosu, který mi poskytly ve fázích návrhu a implementace.

10.1 Model požadavků aplikace

10.1.1 Uživatelé systému



Obrázek 10.1: Diagram popisující typy uživatelů a jejich základní akce

Rozlišujeme dva základní typy uživatelů. Nepřihlášený uživatel se může pouze přihlásit do systému, čímž se z něj stane jeden ze dvou možných přihlášených uživatelů, tedy student nebo vyučující. Jediná akce jim společná je odhlášení ze systému, které z nich opět učiní nepřihlášeného uživatele.

10.1.2 Přihlášení do systému

Nepřihlášenému uživateli se zobrazí úvodní obrazovka systému s formulářem, kde se může přihlásit pomocí svého uživatelského jména a hesla. Po úspěšném přihlášení pokračuje v práci jako student (případ užití *Zobrazit seznam zapsaných předmětů*) nebo vyučující (případ užití *Zobrazit seznam vyučovaných předmětů*).

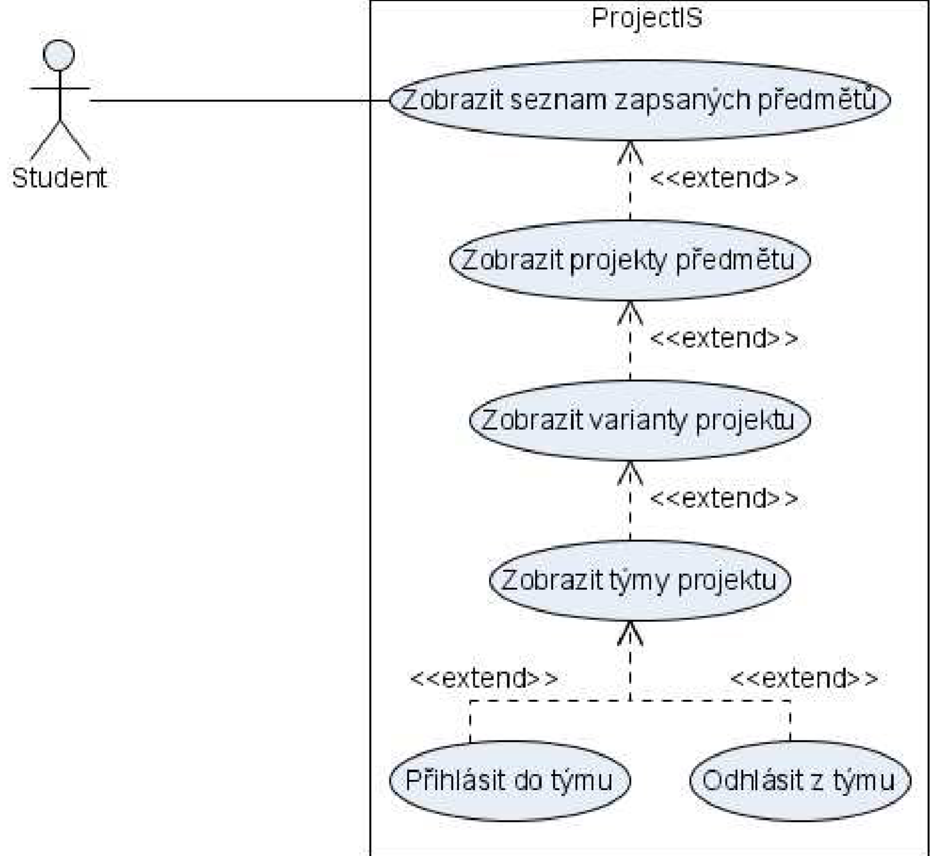
10.1.2.1 Alternativní scénáře

- *Uživatel nezadal uživatelské jméno nebo heslo*: zobrazí se opět formulář s upozorněním, že některý z údajů nebyl vyplněn. Případné zadané uživatelské jméno ve formuláři zůstane.
- *Uživatel zadal neplatné přihlašovací údaje*: zobrazí se opět úvodní obrazovka s upozorněním, že zadaná kombinace je neplatná.

10.1.3 Odhlášení ze systému

Přihlášený uživatel má kdykoliv v průběhu práce se systémem k dispozici volbu odhlášení ze systému. Po jejím zvolení se stane zpět nepřihlášeným uživatelem a pokračuje se případem užití *Přihlášení do systému*.

10.1.4 Případy užití prováděné studentem



Obrázek 10.2: Diagram popisující možné akce prováděné studentem

10.1.5 Zobrazit seznam zapsaných předmětů

Po přihlášení do systému se zobrazí studentovi seznam jeho zapsaných předmětů. Zvolením konkrétního předmětu se aktivuje případ užití *Zobrazit projekty předmětu*.

10.1.6 Zobrazit projekty předmětu

Studentovi se zobrazí seznam projektů vypsaných k danému předmětu. Po zvolení konkrétního projektu se aktivuje případ užití *Zobrazit varianty projektu*.

10.1.7 Zobrazit varianty projektu

Studentovi se zobrazí seznam variant daného projektu. Zvolením konkrétní varianty se aktivuje případ užití *Zobrazit týmy projektu*.

10.1.8 Zobrazit týmy projektu

Zobrazí studentovi seznam týmů řešících daný projekt. Při každém týmu je uveden seznam jeho členů. Pokud je student členem určitého týmu, systém mu nabídne možnost odhlášení. U všech nezaplněných týmů se zobrazí volba pro přihlášení. Pokud byl student již hodnocen, zobrazí se zde i tento údaj. Pokud student zvolí odhlášení, spustí se případ užití *Odhlásit z týmu*. Pokud student zvolí přihlášení, spustí se případ užití *Přihlásit k týmu*.

10.1.9 Přihlásit k týmu

Student je přihlášen k danému týmu. Následně se opět spustí případ užití *Zobrazit týmy projektu* společně s výsledkem operace.

10.1.9.1 Alternativní scénáře

- *Již se nelze přihlašovat*: Mezi zobrazením volby k přihlášení a jejím zvolením vypršel termín k přihlašování. Pokračuje se případem užití *Zobrazit týmy projektu*, kde je zobrazena příslušná chybová hláška.
- *Tým je již plný*: Mezi zobrazením volby k přihlášení a jejím zvolením byl tým již zaplněn. Pokračuje se případem užití *Zobrazit týmy projektu*, kde je zobrazena příslušná chybová hláška.

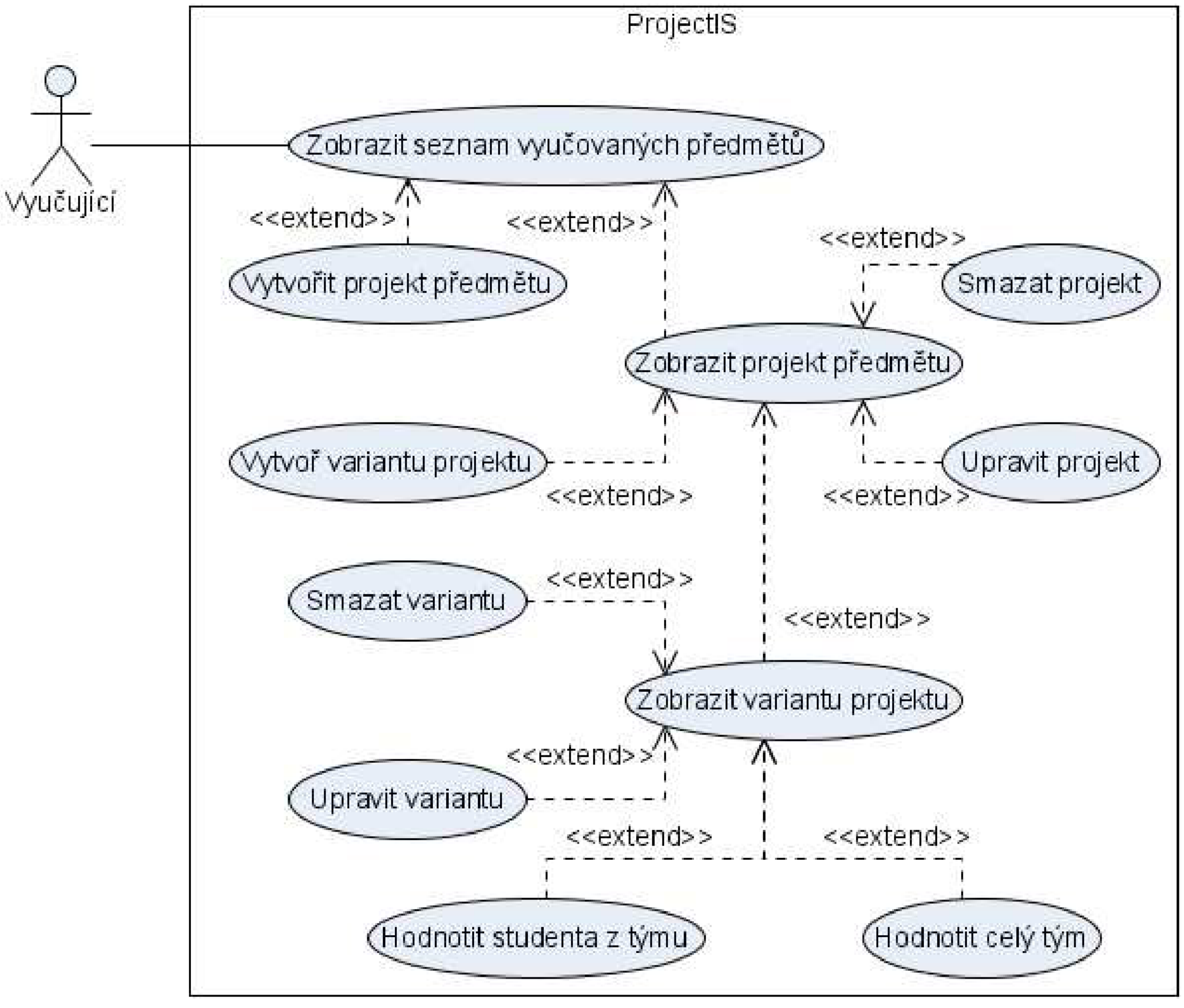
10.1.10 Odhlásit z týmu

Student je odhlášen z daného týmu. Následně se opět spustí případ užití *Zobrazit týmy projektu*, kde je zobrazen výsledek operace.

10.1.10.1 Alternativní scénáře

- *Již se nelze odhlašovat*: Mezi zobrazením volby k odhlášení a jejím zvolením vypršel termín k odhlašování. Pokračuje se případem užití *Zobrazit týmy projektu*, kde je zobrazena příslušná chybová hláška.

10.1.11 Případy užití prováděné vyučujícím



Obrázek 10.3: Diagram popisující možné akce prováděné vyučujícím

10.1.12 Zobrazit seznam vyučovaných předmětů

Po přihlášení do systému se vyučujícímu zobrazí seznam vyučovaných předmětů společně se seznamem vypsaných projektů a volbou pro vytvoření nového projektu ke každému z nich. Tato volba spustí případ užití *Vytvořit projekt předmětu*. Vybráním konkrétního projektu se spustí případ užití *Zobrazit projekt předmětu*.

10.1.13 Vytvořit projekt předmětu

Systém nabídne vyučujícímu formulář pro vytvoření nového projektu. Ten do něj zadá název projektu, nepovinný popis, datum odevzdání projektu, zahájení a ukončení přihlašování a maximální počet bodů. Po odeslání hodnot se spustí případ užití *Zobrazit seznam vyučovaných předmětů* s výsledkem operace.

10.1.13.1 Alternativní scénáře

- *Nebyl zadán některý z povinných parametrů*: zobrazí se zpět formulář s již vyplněnými hodnotami. U povinných nevyplněných polí se vypíše chybová zpráva oznamující, že dané pole musí být vyplněno.
- *Některý parametr je v neplatném formátu*: zobrazí se zpět formulář s již vyplněnými hodnotami. Neplatného formátu mohou nabývat datumová pole (datum odevzdání projektu, zahájení a ukončení přihlašování), resp. maximální počet bodů (ten musí být celé číslo větší rovno nule). U neplatných polí se zobrazí chybová zpráva.
- *Porušena posloupnost datumů*: datum začátku registrace musí být dříve než konec registrace, který musí být dříve než datum odevzdání projektu. Při porušení této podmínky se zobrazí opět formulář s vyplněnými hodnotami a příslušnou chybovou zprávou.

10.1.14 Zobrazit projekt předmětu

Vyučujícímu se zobrazí informace o vybraném projektu včetně seznamu variant. Zvolením dané varianty se spustí případ užití *Zobrazit variantu projektu*. Také je nabídnuta volba pro úpravu (případ užití *Upravit projekt*) a smazání (*Smazat projekt*) daného projektu a volba pro zadání nové varianty projektu (*Vytvořit variantu projektu*).

10.1.15 Smazat projekt

Případ užití se spustí, když vyučující zvolí smazání v rámci případu užití *Zobrazit projekt předmětu*. Projekt je smazán a výsledek operace zobrazen v rámci případu užití *Zobrazit seznam vyučovaných předmětů*.

10.1.15.1 Alternativní scénáře

- *Projekt má alespoň jednu variantu*: Projekt lze smazat jen tehdy, pokud nemá vypsány žádné varianty. V opačném případě je vyučující na tuto skutečnost upozorněn chybovou zprávou.

10.1.16 Upravit projekt

Zobrazí se formulář určený k úpravě daného projektu. Pole jsou vyplněná jeho hodnotami. Vyučující je upraví (lze upravovat všechny) a odešle, čímž se spustí případ užití *Zobrazit seznam vyučovaných předmětů* s výsledkem operace. Alternativní scénáře jsou totožné s případem užití *Vytvořit projekt předmětu*.

10.1.17 Vytvořit variantu projektu

Systém zobrazí formulář určený pro vytvoření nové varianty. Vyučující do něj zadá název, nepovinný popis, maximální počet řešitelských týmů a jejich kapacitu a tato data odešle. Výsledek operace se zobrazí v rámci případu užití *Zobrazit projekt předmětu*.

10.1.17.1 Alternativní scénáře

- *Nebyl zadán některý z povinných parametrů*: zobrazí se zpět formulář s již vyplněnými hodnotami. U povinných nevyplněných polí se vypíše chybová zpráva oznamující, že dané pole musí být vyplněno.
- *Některý parametr je v neplatném formátu*: zobrazí se zpět formulář s již vyplněnými hodnotami. Neplatného formátu může nabývat maximální počet řešitelských týmu (celé číslo větší rovno

jedné) a nebo počet členů jednoho týmu (celé číslo větší rovno jedné). Při neplatných polích se zobrazí chybová zpráva.

10.1.18 Zobrazit variantu projektu

Poskytne vyučujícímu detail dané varianty projektu včetně všech týmů, které ji řeší. Pokud již vyučující zadal nějaká hodnocení, jsou zobrazena i ta. Pro každý tým systém nabídne volbu pro hodnocení všech jeho členů najednou (spustí případ užití *Hodnotit tým*) a podobně volbu hodnocení pro každého studenta zvlášť (*Hodnotit studenta*). Dále nabídne volbu pro smazání této varianty (*Smazat variantu*) a pro její úpravu (*Upravit variantu*).

10.1.19 Upravit variantu

Vyučující má k dispozici formulář s vyplněnými hodnotami získanými z upravované varianty. Má umožněno změnit její název a popis. Po odeslání se pokračuje případem užití *Zobrazit variantu projektu*.

10.1.19.1 Alternativní scénáře

- *Nebyl zadán název varianty*: zobrazí se zpět formulář s již vyplněnými hodnotami společně s chybovou hláškou, že název varianty nebyl vyplněn.

10.1.20 Smazat variantu

Případ užití se spustí, když vyučující zvolí v rámci případu užití *Zobrazit variantu projektu* smazání této varianty. Ta je smazána společně se všemi vygenerovanými týmy a pokračuje se případem užití *Zobrazit projekt předmětu*.

10.1.20.1 Alternativní scénáře

Varianta má alespoň jeden tým, ve kterém je alespoň jeden student: zobrazí se výstraha, že na této variantě již někdo pracuje a žádost o potvrzení smazání. Pokud vyučující smazání potvrdí, odstraní se společně s variantou i všechny její týmy včetně informací o příslušnosti studentů do nich a následuje případ užití *Zobrazit projekt předmětu*. V opačném případě se pokračuje případem užití *Zobrazit variantu předmětu*.

10.1.21 Hodnotit tým

Vyučující zadá bodové hodnocení aplikované na všechny členy týmu a odešle jej. Studenti jsou ohodnoceni a systém pokračuje případem užití *Zobrazit variantu projektu*. Pokud nezadá žádnou hodnotu, je to pokyn k odstranění daných hodnocení.

10.1.21.1 Alternativní scénáře

- *Hodnocení není celé číslo z rozsahu <0, maximumBodůZaProjekt>*: zobrazí se zpět formulář společně s chybovou zprávou oznamující, že zadaná hodnota není celé číslo, resp. není z uvedeného intervalu.

10.1.22 Hodnotit studenta

Vyučující zadá bodové hodnocení konkrétního studenta a odešle jej. Pokračuje se případem užití *Zobrazit variantu projektu*. Pokud nezadá žádnou hodnotu, je to pokyn k odstranění tohoto hodnocení. Alternativní scénáře jsou totožné s případem užití *Hodnotit tým*.

10.2 Výběr implementační platformy

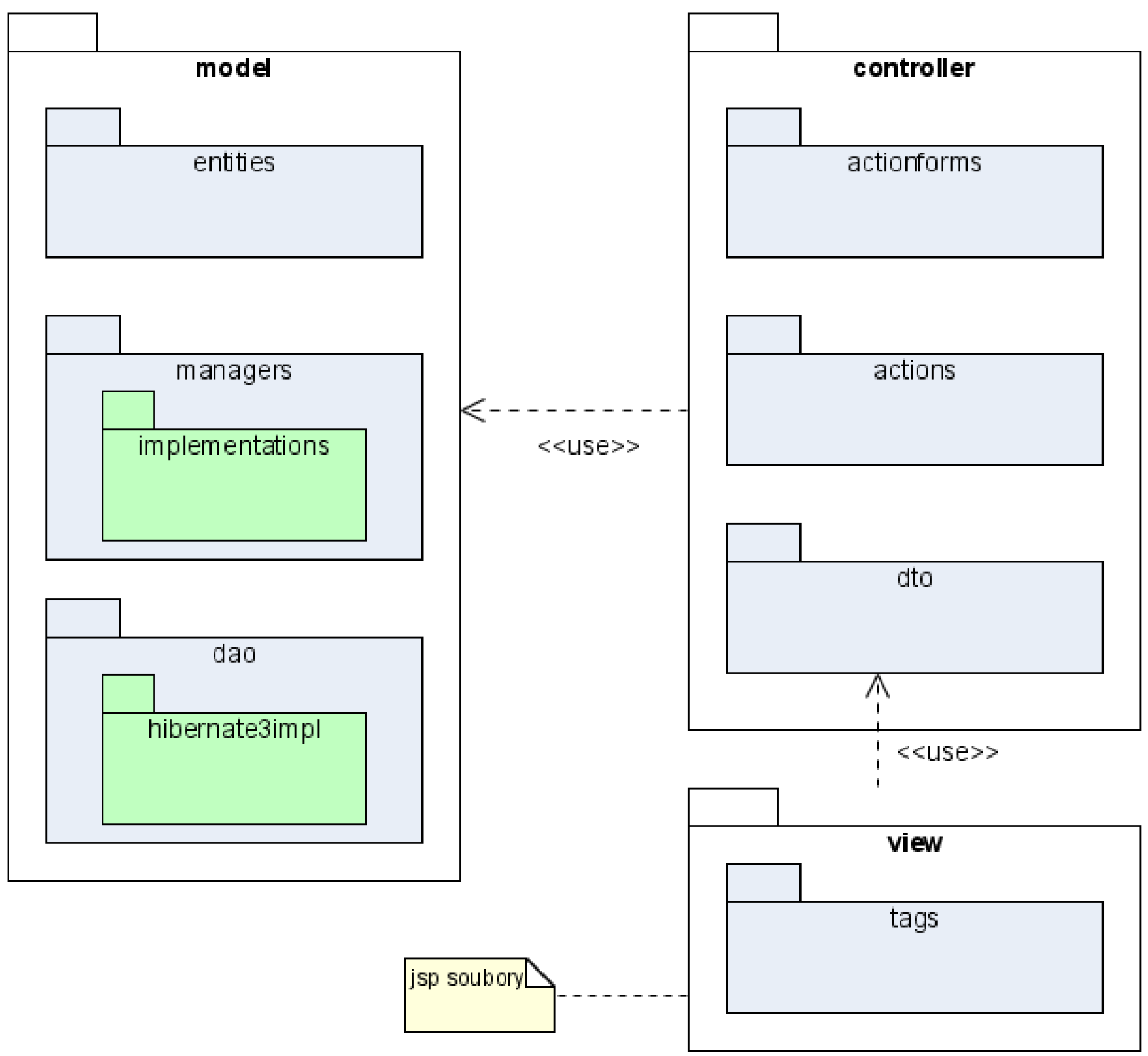
Ještě než se začnu věnovat vývoji ukázkové aplikace, rád bych zdůvodnil své kroky při výběru implementační platformy. Prvním kritériem pro mě byla její vhodnost pro demonstrování popsaných webových vzorů. Z tohoto pohledu mi nejlépe vyšla platforma *Java EE*, která obsahuje velké množství rámců a knihoven aplikujících tyto vzory.

Dále jsem se rozhodoval podle své znalosti dané platformy a zájmu o ni. Platformu *Java EE* považuji za perspektivní prostředí pro vývoj *enterprise* aplikací, a proto jsem uvítal možnost o ní získat prostřednictvím této práce další znalosti.

Z těchto důvodu jsem se nakonec rozhodl pro řešení nad touto platformou, ačkoliv jsem přesvědčen, že podobných výsledků by šlo dosáhnout i na dalších, v této práci zmiňovaných prostředích.

10.3 Přínos vzoru Web MVC

Na začátku návrhu architektury systému jsem se rozhodl pro aplikaci vzoru *Web MVC*. S jeho pomocí se mi podařilo celou aplikaci rozdělit do vrstev obsahujících nahraditelné komponenty s jasně definovanými rozhraními a zodpovědnostmi. Na následujícím diagramu balíčků lze vidět toto rozvrstvení:

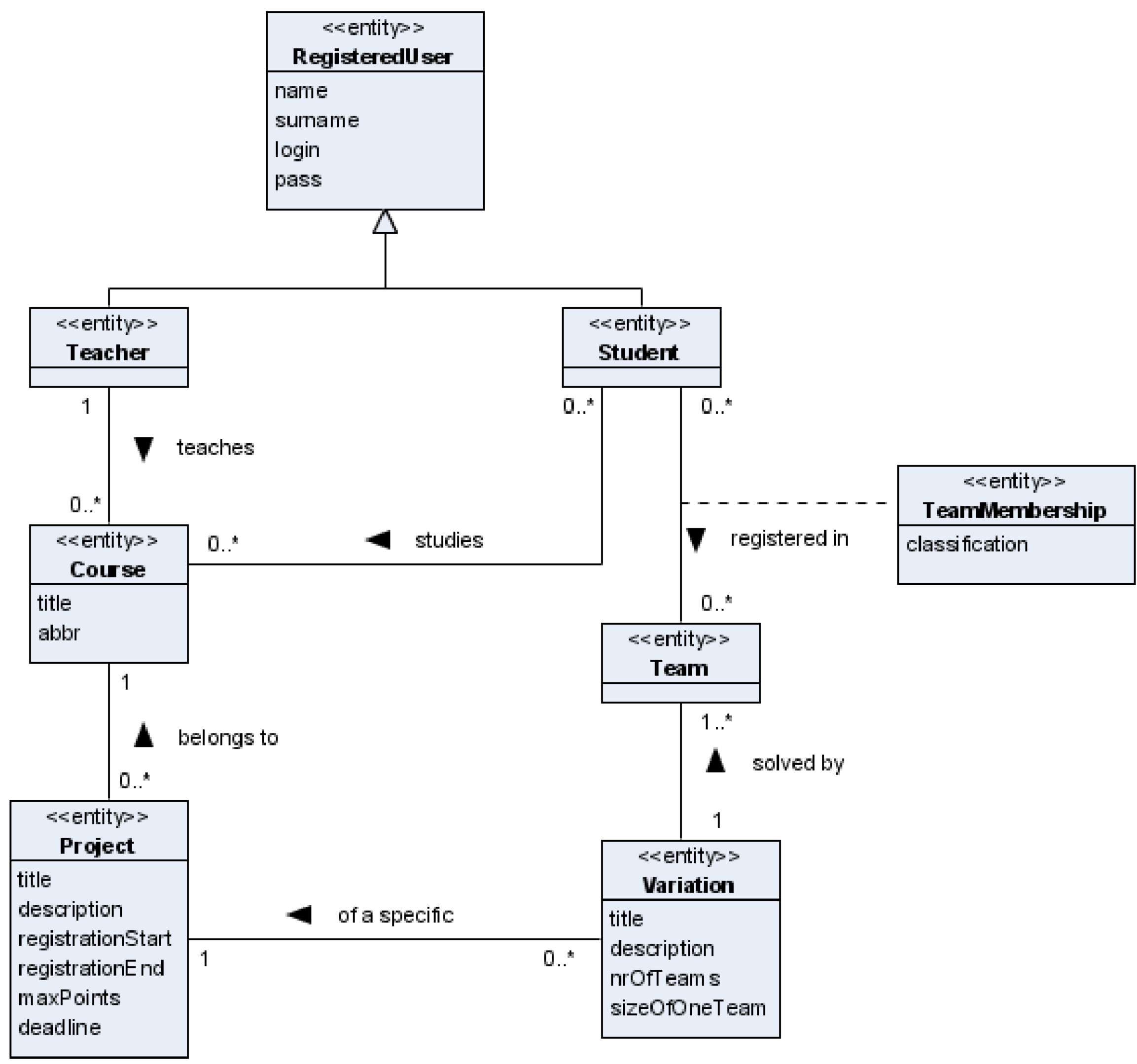


Obrázek 10.4: Diagram balíčků popisující jednotlivé vrstvy aplikace

Z tohoto diagramu vyplývá, že třídy modelu jsou úplně nezávislé na třídách řadiče a podobně serverový pohled využívá pouze DT objekty dodávané řadičem, ale o modelu nemá žádnou znalost.

10.3.2 Vrstva modelu

Model v MVC architektuře zastupuje celou aplikační logiku, zde tvořenou operacemi nad jednotlivými objekty doménových entit. Jejich třídy byly vytvořeny na základě následujícího konceptuálního diagramu:



Obrázek 10.5: Diagram tříd popisující doménové entity z konceptuální úrovně

Tyto entity jsou v modelu reprezentovány svými třídami, jejichž obrazy jsou v databázi automaticky udržovány ve formě tabulek. O tuto úlohu se stará objektově-relační rámec *Hibernate* [33], který zároveň zajišťuje, že všechny změny stavu konkrétního objektu entity budou realizovány i v příslušném záznamu.

Další částí modelu jsou rozhraní definující operace nad jednotlivými entitami. Jde o aplikaci *Java EE* vzoru *Data Access Object*[34]. Tato rozhraní jsou uložena v balíčku `model.dao`, jejich konkrétní implementace využívající rámec *Hibernate* se pak nacházejí v balíčku `model.dao.hibernate3impl`. Pokud by bylo rozhodnuto o jiném způsobu persistence dat než pomocí rámce *Hibernate* (např. pomocí klasického JDBC přístupu), změnila by se pouze implementace těchto DAO tříd, ale rozhraní by zůstala zachována.

Samotnou aplikační logiku ve formě jednotlivých *business* operací definují rozhraní nacházející se v balíčku `model.managers`. Vzhledem k tomu, že aplikace je typu CRUD, obvykle

[33] www.hibernate.org

[34] *„Use a Data Access Object to abstract and encapsulate all access to the persistent store. The Data Access Object manager the connection with the data source to obtain and store data."* [CJP, 463]

metody implementací těchto rozhraní pouze volají své DAO protějšky. Pokud by však vyvstal požadavek na dodatečnou aplikační logiku, byla by implementována právě zde.

Z architektonického hlediska je důležité, že žádná z těchto tříd neobsahuje nějakou specifickou informaci o webové vrstvě, nebo jí není nějak přizpůsobena. Pokud by bylo např. rozhodnuto přetvořit tuto aplikaci na *desktop* řešení, tyto třídy by mohly zůstat beze změny.

10.3.3 Vrstva pohledu

Jednotlivé serverové pohledy jsou implementovány ve formě JSP stránek, jejichž jediným úkolem je zobrazit data dodaná řadičem, přesně dle strategie *Service to Worker*. K tomu mj. využívají dva v této práci popsané vzory, *View Helper* (viz 10.7) a *Composite View* (viz 10.8).

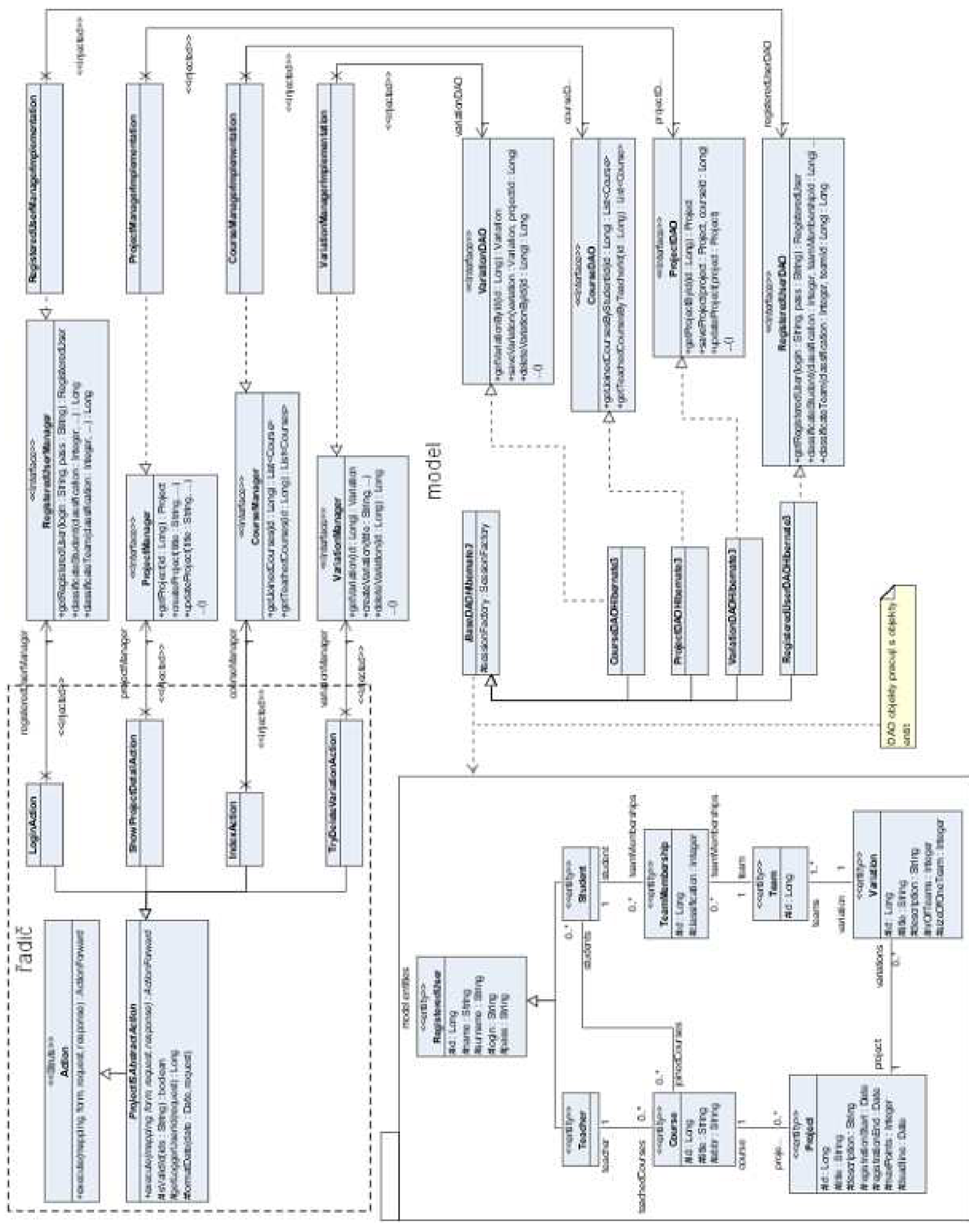
10.3.4 Vrstva řadiče

Řadič jsem se rozhodl neimplementovat od základů, ale využít služeb rámce *Struts*. Ten poskytuje hotový *front controller* řídící celou navigaci po aplikaci. Odezvy na jednotlivé požadavky jsem implementoval do *helper* tříd rozšiřujících *Struts* třídu `Action` uložených v balíčku `controller.actions` (viz 5.4.1). Tyto třídy mají za úkol získat z http požadavku uživatelem zadaná data, zvalidovat je a předat příslušné aplikační logice. Ta provede potřebné operace a vrátí výsledná data, na základě kterých jsou opět v *helperech* vytvořeny DT objekty a předány ke zpracování serverovému pohledu.

10.3.5 Integrace vrstev aplikace

Pro integraci tříd řadiče a modelu jsem se rozhodl využít služeb rámce *Spring*[35]. Ten implementuje paradigma *Inversion Of Control* a zbavuje jednotlivé objekty povinnosti zajišťovat si spojení na další objekty tím, že na sebe bere zodpovědnost za jejich konstrukci, při které jim tato spojení poskytuje automaticky [IOC]. Následující diagram popisuje celou strukturu aplikace z pohledu návrhu. Nejsou v něm uvedeny serverové pohledy, protože jsou implementovány jako JSP stránky. Je-li u asociace uveden stereotyp `injected`, znamená to, že konkrétní spojení mezi objekty je realizováno rámcem *Spring*:

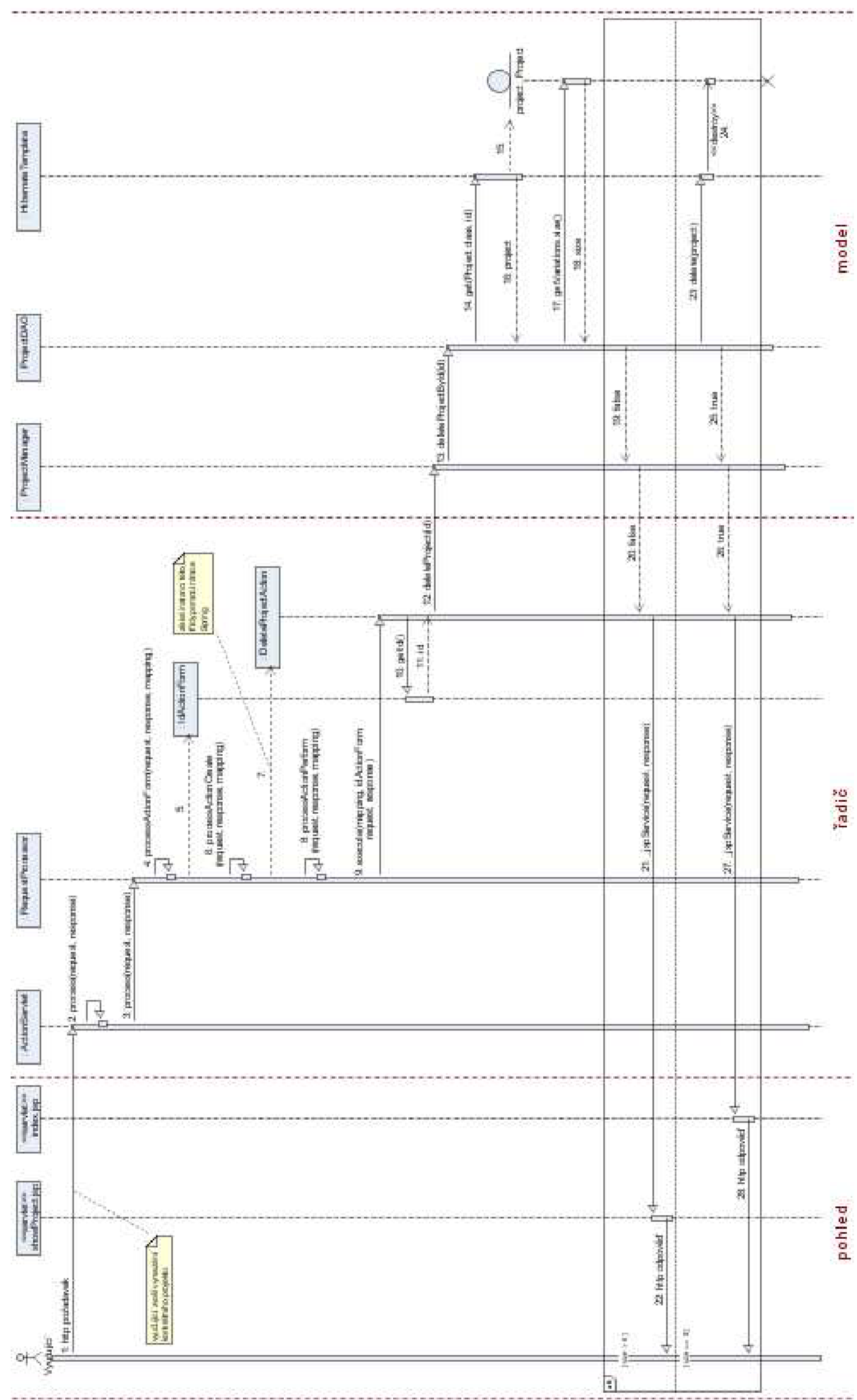
[35] http://www.springframework.org/



Obrázek 10.6: Diagram tříd popisující strukturu aplikace (viz příloha 1)

10.3.6 Komunikace v rámci MVC architektury

Následující sekvenční diagram popisuje realizaci případu užití *Smazat projekt* (viz 10.1.15). Důraz zde kladu na hledisko MVC architektury pomocí konceptu *swimlines*:

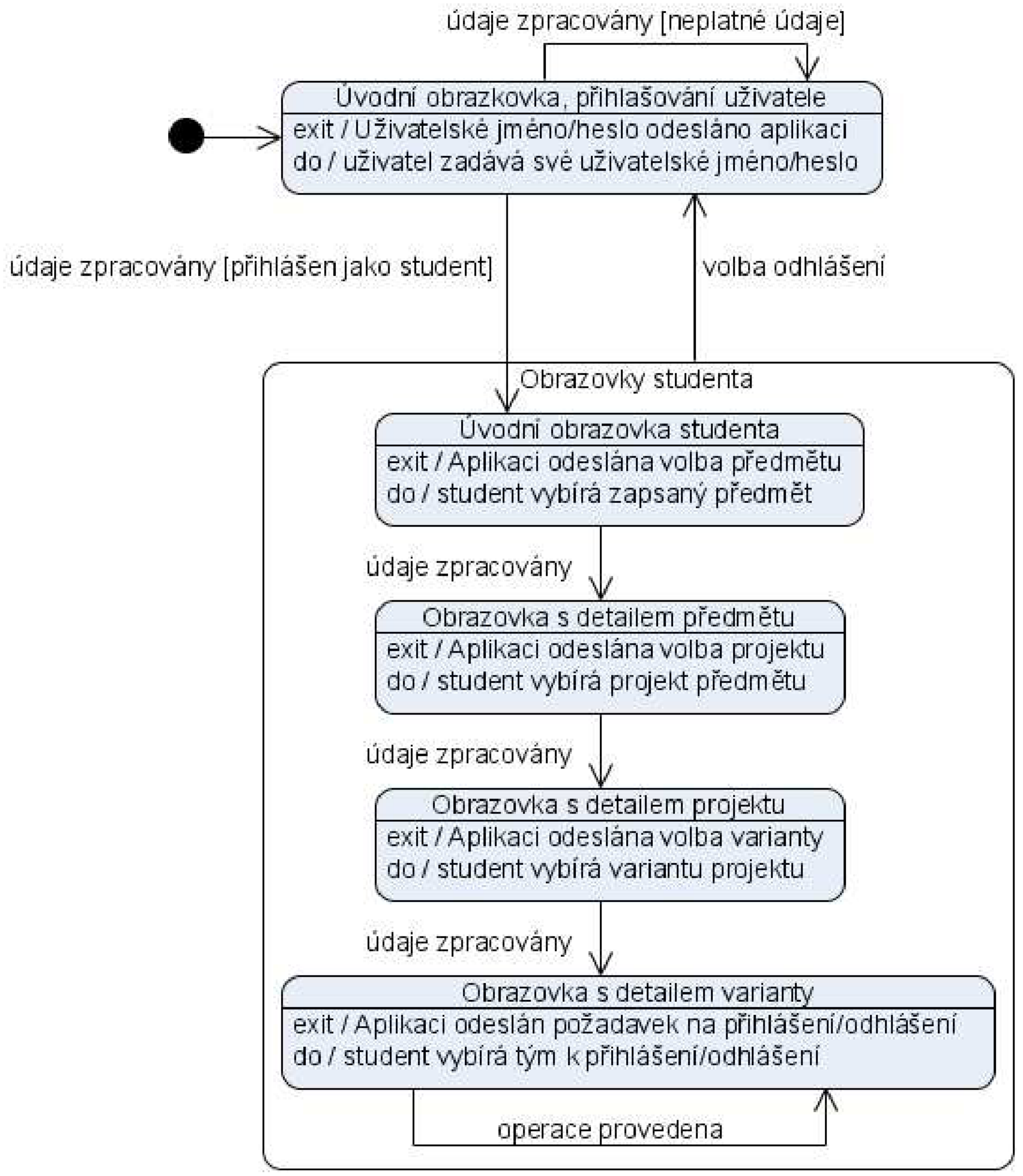


Obrázek 10.7: Sekvenční diagram komunikace v MVC architektuře (viz příloha 2)

Architektonický vzor *Web MVC* mi pomohl při počátečním návrhu architektury celé aplikace. Učinil ji přehlednější a jednotlivé komponenty jen volně svázal s dalšími, což by mělo být cílem každého dobrého návrhu.

10.4 Přínos vzoru Front Controller

Uživatelské rozhraní webové aplikace se obecně skládá z posloupnosti webových stránek, mezi kterými se přechází jako reakce na uživatelské akce. Nejinak je tomu v případě této ukázkové aplikace. Následující stavový diagram vznikl na základě modelu požadavků a popisuje posloupnost obrazovek studenta:



Obrázek 10.8: Stavový diagram popisující posloupnost obrazovek studenta

Uživatel v každém stavu provádí činnost (popsanou aktivitou `do`), která je zakončena použitím některého uživatelského prvku (odkazu, tlačítka formuláře), což má za následek odeslání dat na server a jejich zpracováním tam (výstupní akce `exit`). Přechod mezi stavy je dán výsledkem tohoto zpracování.

Použitý MVC rámec *Struts* se řadí do rodiny požadavkem řízených řešení využívajících pro implementaci svého řadiče vzor *Front Controller* (viz 5.4.1). Centralizuje tedy řízení celé aplikace do

jedné přehledné komponenty. A právě díky tomu bylo možno popsanou navigaci udržet na jednom místě, v tomto případě v konfiguračním souboru rámce:

```
…
<action path="/login" … >
  <forward name="student" path="/student_index.do" … />
  <forward name="failed" path="/WEB-INF/pages/index.jsp"/>
  …
</action>

<action path="/student_showCourseDetail" … >
  <forward name="success" path="/…/student/showCourseDetail.jsp"/>
</action>

<action path="/student_showProjectDetail" … >
  <forward name="success" path="/…/student/showProjectDetail.jsp"/>
</action>
…
```

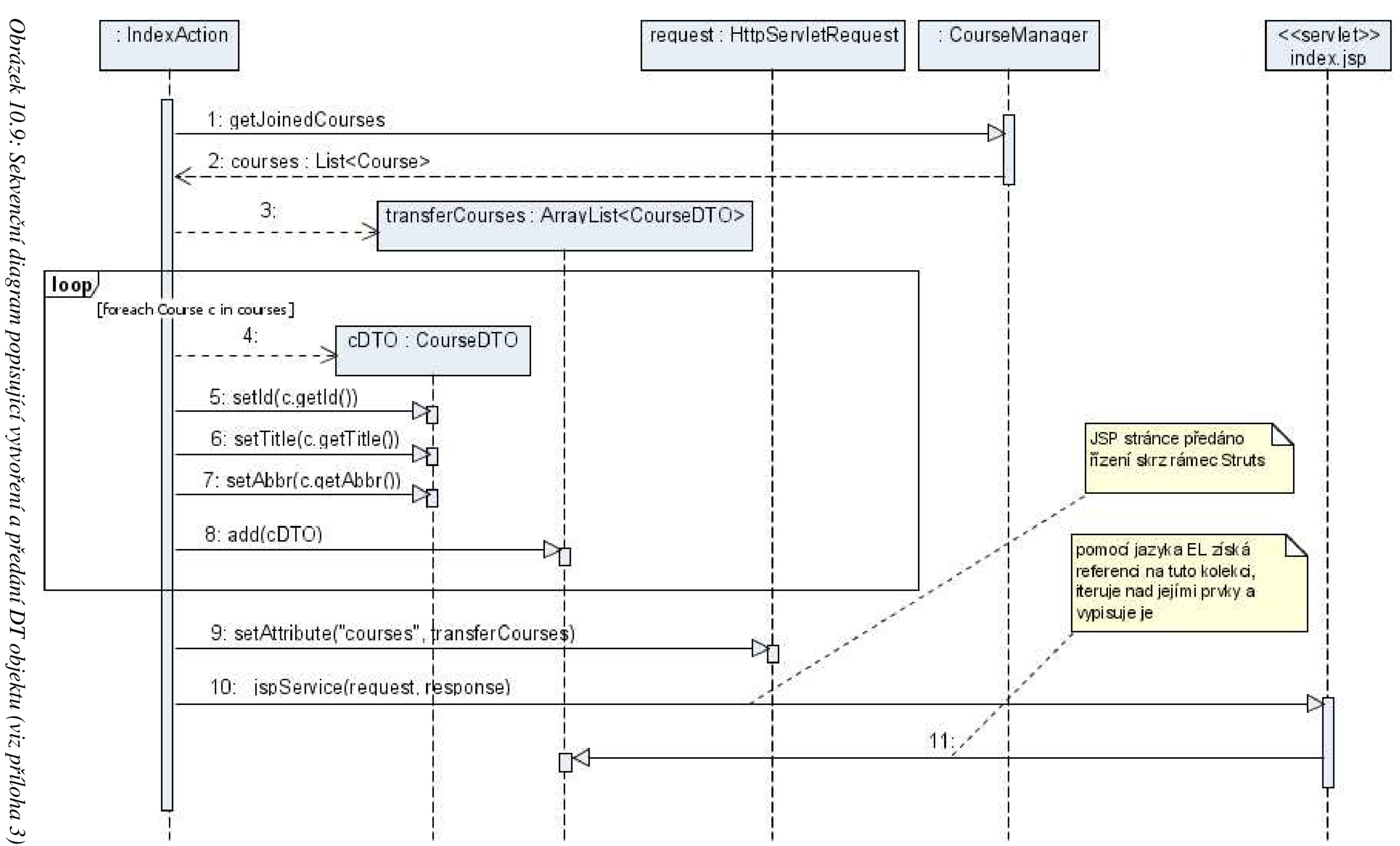
Elementy `action` popisují akce zpracovávající uživatelská data zadaná v současné obrazovce. Jejich potomci `forward` jsou pak vyjádřením jednotlivých možných přechodů, jejichž cílový stav je definován atributem `path` nejčastěji obsahujícím JSP stránku popisující nový stav rozhraní. Např. pokud uživatel zadá a odešle na úvodní stránce své přihlašovací údaje, zpracuje je akce `/login` a pokračuje se do jednoho ze stavů obrazovky popsaného vnořeným elementem `forward`, což koresponduje s uvedeným diagramem. Podobným způsobem byla vyjádřena celá navigace.

Díky aplikaci vzoru *Front Controller* bylo možno umístit navigaci po aplikaci do jednoho místa, odkud je jednoduše spravovatelná.

10.5 Přínos vzoru Data Transfer Object

Protože aplikace využívá komunikační strategii *Service to Worker*, je nutno navrhnout mechanismus, jakým budou data předávána z řadiče do serverového pohledu. K tomuto účelu se nabízí vzor *Data Transfer Object*. Na tomto řešení participuje několik součástí, konkrétně *helper* komponenta řadiče, adresovaná část modelu, serverový pohled a konečně DT objekt, který mezi nimi přenáší data (viz 6.3).

Následující sekvenční diagram popisuje jednu konkrétní komunikaci, která je součástí realizace případu užití *Zobrazit seznam zapsaných předmětů* (viz 10.1.5 nahoře). Jejím úkolem je obdržet z aplikační vrstvy všechny studentovy zapsané předměty a zobrazit je pomocí serverového pohledu:



Obrázek 10.9: Sekvenční diagram popisující vytvoření a předání DT objektu (viz příloha 3)

Jak je z diagramu patrno, většina zodpovědnosti je implementována ve třídě `IndexAction` představující *helper* komponentu řadiče *Struts*. Ta získá data z aplikační logiky, vytvoří pro ně příslušné DT objekty a následně je uloží do *rozsahu* požadavku, odkud jsou přístupné serverovému pohledu.

Struktura třídy `CourseDTO` kopíruje vlastnosti entity `Course`, na rozdíl od ní však slouží pouze k přenosu dat. Je tak zajištěno, že z pohledu nebude možno změnit data v databázi, k čemuž by mohlo dojít, pokud by mu byl vystaven přímo objekt entity.

Pomocí vzoru *Data Transfer Object* se mi povedlo učinit vrstvu serverového pohledu maximálně nezávislou na ostatních komponentách a zajistit, aby neměla prostředky k případné změně stavu objektů entit.

10.6 Přínos vzoru Intercepting Filter

Jak vyplývá z analýzy případů užití, aplikace je rozdělena na tři logické části, na část studentskou, pro vyučujícího a úvodní přihlašovací obrazovku. Aby mohl uživatel vykonávat akce náležející vyučujícímu, musí být jako vyučující i přihlášen, a obdobně pro studenta. Pouze úvodní obrazovky může využívat jako nepřihlášený. V rámci celé aplikace platí, že URL adresy studentské sekce mají prefix `student_`, zatímco vyučující používá adresy začínající `teacher_`. Akce vedoucí k zobrazení úvodní obrazovky a k přihlášení mají URL bez prefixu.

Pro implementaci logiky kontrolující, zdali má uživatel pro danou akci dostatečné oprávnění, jsem se rozhodl využít služeb vzoru *Intercepting Filter*. Každý http požadavek, než dorazí do samotné aplikace (přesněji řečeno, do řadiče rámce *Struts*), projde filtrem třídy `filters.SecurityFilter`, který zkontroluje, zdali je v http sezení uložen objekt popisující správného uživatele:

```
public class SecurityFilter implements Filter {
  …
  public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain chain)
    throws IOException, ServletException {

    HttpServletRequest req = (HttpServletRequest) request;
    HttpServletResponse resp = (HttpServletResponse) response;
    …

      //získej ze sezení objekt popisující přihlášeného uživatele
    HttpSession session = req.getSession();
    RegisteredUserDTO user =
      (RegisteredUserDTO)session.getAttribute(SESSION_LOGGED_USER);

    if (target.equals("index.do") || target.equals("login.do")) {
        //uživatel chce jít na úvodní stránku nebo se přihlásit,
        //odhlaš ho, pokud je přihlášený a pusť dále
      if (user != null) {
        session.removeAttribute(SESSION_LOGGED_USER);
      }
    }
    else if (target.startsWith("student_")) {
      if (user == null || !(user instanceof StudentDTO)) {
          //uživatel se snaží provést akci od studenta, ale není
          //přihlášený nebo ne jako student, pošli jej na
          //úvodní stránku aplikace
        resp.sendRedirect("./index.do?err=log");
```

```
        return;
      }
    }
    else if (target.startsWith("teacher_")) {
      if (user == null || !(user instanceof TeacherDTO)) {
          //uživatel se snaží provést akci od vyučujícího, ale není
          //přihlášený nebo ne jako vyučující, pošli jej na
          //úvodní stránku aplikace
        resp.sendRedirect("./index.do?err=log");
        return;
      }
    }
    chain.doFilter(request, response);
  }
```

Do metody `doFilter()` (viz 7.4.1) jsem implementoval kompletní kontrolu přihlášení. Pokud je detekováno potenciální narušení bezpečnosti, daný http požadavek se vůbec nedostane k aplikaci, ale klientu je zasláno přesměrování na úvodní stránku, kde se následně zobrazí chybová zpráva. V opačném případě dojde k volání metody `chain.doFilter()`, čímž se požadavek odešle ke zpracování samotné aplikaci. Za tímto voláním již nejsou žádné další příkazy, http odpověď proto nechává tento filtr nedotčenu.

Využitím vzoru *Intercepting Filter* se mi povedlo centralizovat správu bezpečnosti do jediné komponenty stojící mimo aplikaci, která tak zůstane zcela odstíněna od neoprávněných požadavků. Navíc, pokud by došlo ke změně bezpečnostní strategie (kontrola existence daného uživatele v databázi při každém požadavku, získání popisu jeho práv od třetí strany apod.), byla by prováděna pouze v tomto filtru a samotná aplikace by zůstala nedotčena.

10.7 Přínos vzoru View Helper

V navrhované aplikaci hraje serverový pohled relativně pasivní úlohu, pouze zobrazuje data dodaná pomocí DT objektů. Přesto se i zde našly problémy, které bylo nutno řešit pomocí prezentační logiky. Protože jsem nechtěl tuto logiku implementovat přímo v pohledu pomocí některého z vyšších jazyku, zde *Javy*, rozhodl jsem se pro řešení založené na vzoru *View Helper*. Následuje popis tématických okruhů společně s jejich řešením.

10.7.1 Řídící struktury prezentační logiky

V rámci vývoje serverových pohledů jsem se často setkal s případem, kdy bylo nutno prostřednictvím jednoduché logiky vypsat různý text podle toho, zdali daná proměnná existuje nebo zdali je daná kolekce prázdná. Za podobnou ukázku prezentační logiky lze považovat i jednoduché iterování nad daty. V těchto případech jsem využil knihovny *custom tagů* JSTL (viz 8.4.2), konkrétně její podknihovny *Core*.

Následující fragment kódu demonstruje řešení využité ve studentské části, které slouží k výpisu variant konkrétního projektu:

```
…
<ul>
  <c:choose>
(1)  <c:when test="${empty project.variations}">
      <li>
        <span class="empty">
(2)       --<bean:message key="empty.variations_project"/>--
```

```
        </span>
      </li>
    </c:when>
(3) <c:otherwise>
(4)   <c:forEach var="variation" items="${project.variations}">
        <li>
(5)       <html:link page="…">
            ${fn:escapeXml(variation.title)}
          </html:link>
        </li>
      </c:forEach>
    </c:otherwise>
  </c:choose>
</ul>
…
```

V tomto případě jsem pomocí JSTL *tagů* realizoval následující logiku:

- pokud projekt nemá žádnou variantu (1), vypiš příslušné upozornění (2)
- v opačném případě (3) pro každou variantu (4) vytvoř odkaz na její detail (5)

Zvýrazněné elementy lze považovat za jednoduché *View Helper* komponenty. Pokud by tento prostředek nebyl k dispozici, bylo by nutno použít *scriptletu* (viz 8.4.1) obsahujícího řídící struktury *Javy*:

```
<ul>
  <%
  List<VariationDTO> variations =
    ((ProjectDTO)pageContext.findAttribute("project"))
      .getVariations();
  if (variations.size() == 0) {
  %>
    <li>
      <span class="empty">…</span>
    </li>
  <%
  }
  else {
    for (VariationDTO variation : variations) {
  %>
    <li>
      <a href="…"><%= variation.getTitle() %></a>
    </li>
  <%}
  }
  %>
</ul>
```

V tomto případě je kód mnohem méně přehledný a míchá v sobě značkovací jazyk společně se strukturami *Javy*, zatímco v předchozí ukázce byly použity pouze značkovací prvky, které jsou vývojářům *frontend* části aplikace obvykle bližší.

10.7.2 Lokalizace aplikace

Pokud má být vyvíjená aplikace lokalizovatelná do různých jazyků, je nutné, aby její serverové pohledy neobsahovaly řetězce ve formě literálů, ale aby byly udržovány na jednom místě v odděleném souboru. Při požadavku na překlad do jiného jazyka se pak tento soubor zamění za jiný.

Použitý webový rámec *Struts* využívá pro tuto příležitost *properties* soubory ve tvaru `klíč=řetězec`. Pro výpis těchto řetězců jsem využil *custom tagu* `message`, který je součástí pod-

knihovny bean dodávané společně s rámcem. V následujícím fragmentu kódu tato *view helper* komponenta načte ze správného[36] souboru řetězce heading.teacher.create_project a text.teacher.create_project a vypíše je na své místo:

```
…
<h2><bean:message key="heading.teacher.create_project"/></h2>
<p><bean:message key="text.teacher.create_project"/>:</p>
…
```

10.7.3 Pokročilá prezentační logika

Ve studentské části aplikace se při výpisu týmů v rámci realizace případu užití *Zobrazit týmy projektu* (viz 10.1.8) objevil případ komplexní prezentační logiky, jejíž řešení by pomocí knihovny JSTL sice bylo možné, ale velmi nepřehledné:

1. Pro každý tým je nutno z jeho DT objektu zjistit, jaké jsou možnosti studenta pro přihlášení do něj.
2. Pokud se do daného týmu nelze přihlašovat, vypíše na obrazovku důvod (tým je plný, ještě nebo již se nelze přihlašovat, resp. student je již členem jiného týmu tohoto projektu).
3. Pokud jsou splněny podmínky pro přihlášení do daného týmu, obrazovka nabídne příslušný uživatelský prvek.
4. Jinak se zobrazí prvek pro odhlášení.
5. Systém vypíše všechny členy týmu na obrazovku.
6. Pokud je přihlášený student právě vypisovaným členem týmu, je nutno zobrazit stav jeho hodnocení.
7. Pokud již vyučující zadal hodnocení, zobraz jej.
8. Jinak vypiš řetězec upozorňující, že hodnocení zatím nebylo zadáno.
9. Pokud je v týmu méně členů než je jeho kapacita, doplň zbývající pozice upozorněním, že tato pozice je prázdná.

Tuto funkčnost jsem se rozhodl zapouzdřit do vlastní *view helper* komponenty, opět v podobě *custom tagu*. Tento postup má z hlediska prezentační vrstvy několik výhod, vytváří pro logiku jednoduché rozhraní v podobě XML elementu a zároveň ji umožňuje implementovat přímo pomocí *Javy*. Následující okomentovaný fragment kódu představuje tento *view helper*:

```
package view.tags;
…
public class PrintTeamTagHandler extends SimpleTagSupport {
    //identifikátor daného DT objektu s daty týmu
  protected String team;
    //maximální velikost týmu
  protected int size;
  …

  public void doTag() throws JspException {
      //získej DT objekt týmu dle předaného identifikátoru
    TeamDTO team =
      (TeamDTO)this.getJspContext().findAttribute(this.team);
    …
    try {
      out.println("<ul>");
```

[36] Soubor je vybrán na základě informací o lokalitě uživatele získané z http požadavku, případně je použit výchozí soubor.

```
(1)   int status = team.getJoinStatus();
      out.println("<li class=\"first\">");

(2)   if (status == REGISTER_LATER_REASON) {
        out.println(res.getMessage("reason.register_later"));
      }
      else if (status == REGISTRATION_MISSED_REASON) {
        out.println(res.getMessage("reason.registration_missed"));
      }
      else if (status == ALREADY_REGISTERED_IN_ANOTHER_TEAM_REASON){
        out.println(
          res.getMessage("reason.already_registered_in_another_team"));
      }
      else if (status == TEAM_FULL_REASON) {
        out.println(res.getMessage("reason.team_full"));
      }
      else {
        …
(3)     if (status == REGISTRATION_AVAILABLE) {
          … //vytvoř tlačítko pro přihlášení
        }
(4)     else {
          … //vytvoř tlačítko pro odhlášení
        }
        …
      }
      out.println("\n</li>");

        //získej ze session DT objekt právě přihlášeného studenta
      StudentDTO sDTO = (StudentDTO)
        this.getJspContext().findAttribute(SESSION_LOGGED_USER);
      …
      int cnt = 0;
(5)   for (StudentDTO student : team.getStudents()) {
        out.print("<li>");
        … //vypiš údaje o studentovi
(6)     if (student.getId().equals(sDTO.getId())) {
(7)       if (student.getClassification() == null) {
            //vypiš řetězec, že student nebyl zatím hodnocen
            …
          }
(8)       else {
            out.print("<strong>");
            out.print(student.getClassification());
            out.print("</strong>");
          }
        }
        out.println("\n</li>");
        cnt++;
      }

(9)   for (;cnt < this.size; cnt++){
        out.print("<li><span class=\"empty\">--");
        out.print(res.getMessage("empty.student_slot"));
        out.println("--</span></li>");
      }
      out.println("</ul>\n");
    }
    catch (java.io.IOException ex) {
```

```
      throw new JspException(ex.getMessage());
    }
  }
}
```

Takto implementovaný *tag* pak popisuje své XML rozhraní v příslušném popisovači *ProjectISTLD.tld*:

```
  <tag>
    <name>printTeam</name>
    <tag-class>view.tags.PrintTeamTagHandler</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>team</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>java.lang.String</type>
    </attribute>
    <attribute>
      <name>size</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>int</type>
    </attribute>
  </tag>
```

Zde je definováno, že daný *tag* bude reprezentován elementem s názvem `printTeam` obsahujícím dva povinné atributy, řetězcový `team` a celočíselný `size`. Použití takto připravené komponenty je v serverovém pohledu implementovaném pomocí JSP jednoduché:

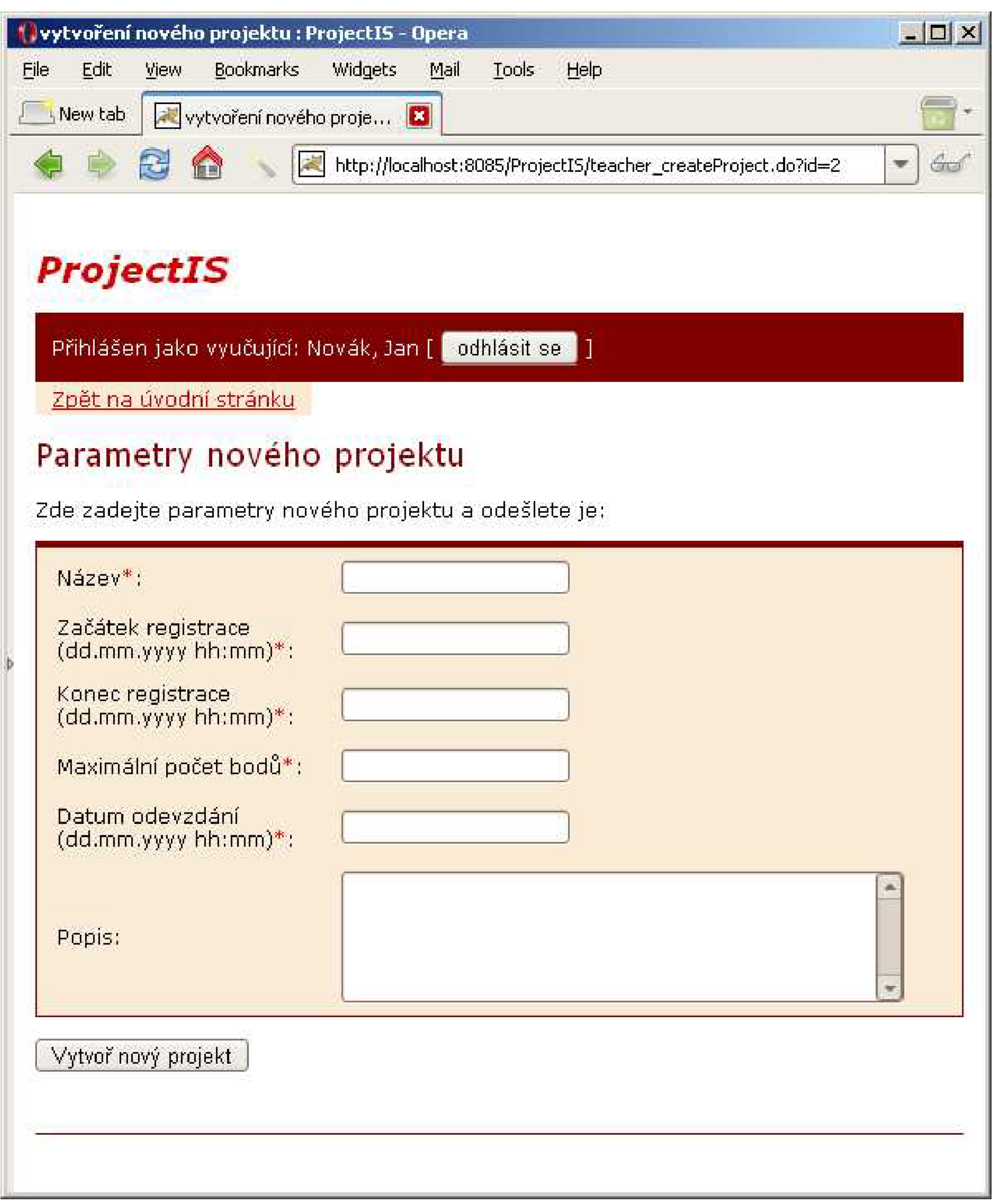
```
<c:forEach var="team" items="${variation.teams}" varStatus="st">
  <h4>${st.count}. <bean:message key="entity.team"/></h4>
  <my:printTeam team="team" size="${variation.sizeOfOneTeam}"/>
</c:forEach>
```

Pomocí JSTL iterace se postupně načítají všechny týmy předané v rámci DT objektu `variation` a předávají *tagu* `printTeam` společně s informací o kapacitě týmu. Ten se již pak postará o správný výpis pomocí implementované logiky.

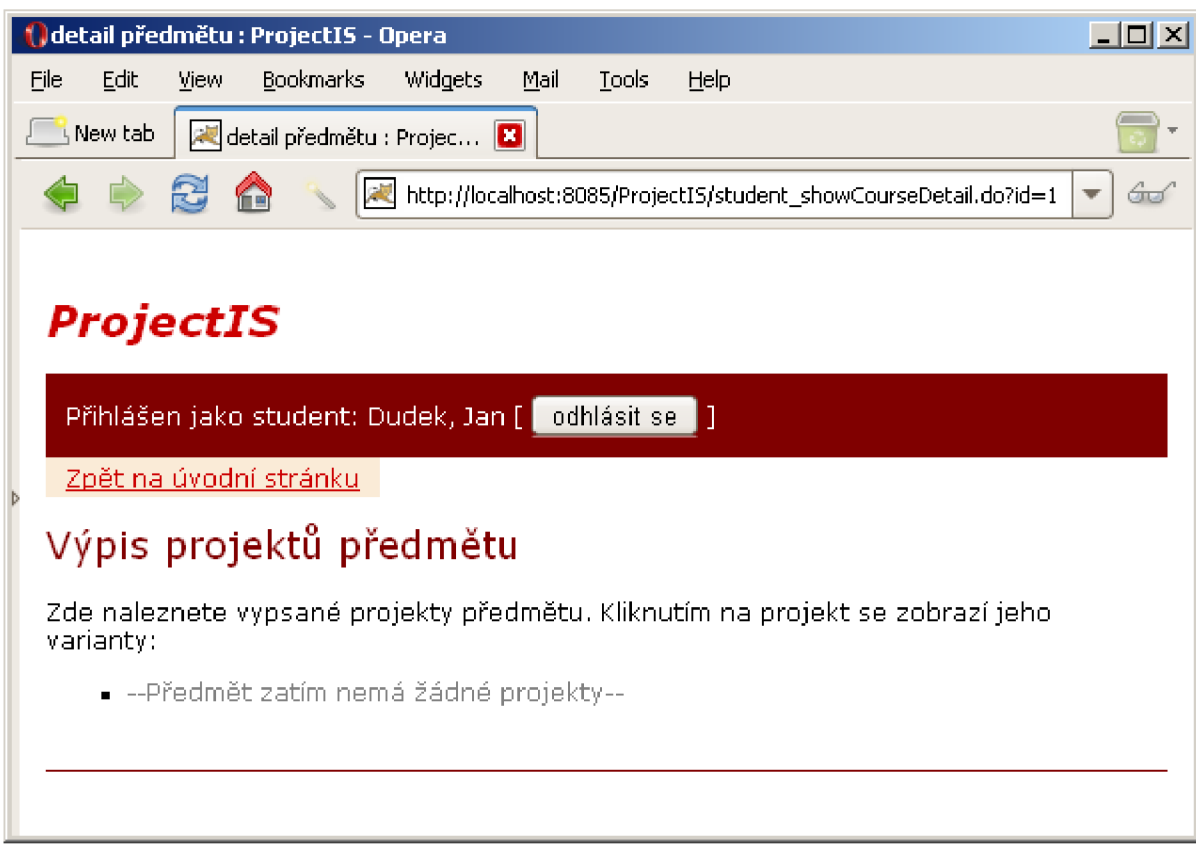
Zmíněné tři ukázky vzoru *View Helper* ukazují, že jeho použití skutečně činí kód serverového pohledu přehlednějším a lépe spravovatelným. Vývojáři mohou i nadále využívat značkovacího jazyka, na který jsou zvyklí, a navíc jsou odstíněni od plnohodnotného programovacího prostředku, jehož prostřednictvím by mohli případně narušit strukturu aplikace, protože by měli zjednodušený přístup k funkcionalitě nižších vrstev.

10.8 Přínos vzoru Composite View

Aby bylo celé GUI aplikace konzistentní a pro uživatele tak příjemné, je nutno zajistit, že všechny obrazovky budou mít stejné rozložení a stejný grafický motiv. Následující dva obrázky zobrazují reálné obrazovky aplikace:



Obrázek 10.10: Obrazovka vyučujícího pro vytvoření nového projektu



Obrázek 10.11: Obrazovka studenta s výpisem projektů předmětu

Jak je vidno, obě obrazovky se sice samozřejmě liší (jiný titulek, stavový řádek, hlavní obsah i odkaz na předchozí stránku), ale rozložení těchto prvků sdílejí. K dosažení takové jednoty se nabízí návrhový vzor *Composite View*. Pro jeho implementaci jsem využil knihovny *Tiles*, která je volnou součástí *Struts* (viz 9.4.1). Nejdříve bylo nutno navrhnout a vytvořit rozložení *projectISLayout.jsp* používané na všech obrazovkách aplikace:

```
...
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<jsp:element name="html">
  <jsp:attribute name="xmlns">http://www.w3.org/1999/xhtml</jsp:attribute>
  <jsp:attribute name="xml:lang">
    <bean:message key="lang"/>
  </jsp:attribute>
  <jsp:attribute name="lang"><bean:message key="lang"/></jsp:attribute>

  <jsp:body>
    <head>
      <meta http-equiv="Content-type" content="text/html; charset=UTF-8"/>
      <link href="${pageContext.request.contextPath}/style.css"
            rel="stylesheet" type="text/css"/>

      <title><tiles:insert attribute="title"/> : ProjectIS</title>
    </head>

    <body>
      <h1>ProjectIS</h1>
```

```
      <div id="main">
        <div id="header">
          <tiles:insert attribute="header"/>
        </div>
        <div id="back">
          <tiles:insert attribute="back"/>
        </div>
        <div id="content">
          <tiles:insert attribute="content"/>
        </div>
      </div>
    </body>
  </jsp:body>
</jsp:element>
```

Každá obrazovka aplikace využívá toto rozložení pro umístění svého obsahu. Např. první zobrazená obrazovka (součást případu užití *Vytvořit projekt předmětu*, 10.1.13) definuje své prvky takto:

```
<%@page pageEncoding="UTF-8"%>
<%@page contentType="text/html"%>
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>

<tiles:insert page="/WEB-INF/pages/layouts/projectISLayout.jsp">

  <tiles:put name="title" type="string">
    <jsp:attribute name="value">
      <bean:message key="tile.teacher.title.create_project"/>
    </jsp:attribute>
  </tiles:put>

  <tiles:put name="header" value="/WEB-INF/…/header.jsp"/>

  <tiles:put name="back" value="/WEB-INF/…/backToIndex.jsp"/>

  <tiles:put name="content"
            value="/WEB-INF/…/content_createProjectForm.jsp"/>

</tiles:insert>
```

Jako titulek se použije lokalizovaný text získaný pomocí *tagu* `bean:message` (viz 10.7.2), za hlavičku obsahu se dosadí sub-pohled ve formě JSP stránky *header.jsp,* za prvek pro skok na předchozí stránku *backToIndex.jsp* a obsah bude zobrazován pomocí stránky *content_createProjectForm.jsp*.

Vzor *Composite View* přinesl do tvorby GUI aplikace zásadní zjednodušení. Protože je vzhled všech stránek definován v jediném dokumentu, je jednoduché jej případně upravit či nahradit za jiný. Zároveň také tento vzor omezil redundanci, části společné více obrazovkám (hlavička, dva možné stavové řádky, volby přechodu na předchozí stránku) jsou definovány pouze jednou a následně opakovaně využity.

11 Závěr

V této diplomové práci bylo mým úkolem prostudovat některé existující vzory využitelné při návrhu a implementaci webových řešení a demonstrovat jejich využití na ukázkové aplikaci. Pro všechny uvedené vzory platilo, že již byly v literatuře popsány v úzké spojitosti s konkrétní vývojovou platformou. Má práce tak sestávala především ze studia těchto pramenů a následného zevšeobecnění popsaných vzorů tak, aby byly využitelné v libovolném prostředí. V tom vidím největší přínos své práce, tyto vzory skutečně charakterizuji co nejobecněji, což dokazuji i zpětným popisem jejich použití v rámcích a knihovnách různých platforem.

Z hlediska dalšího vývoje této práce vidím několik perspektiv. Existují i další v literatuře uváděné vzory, jejichž využití lze nalézt na různých platformách. Zároveň v rámci webu neustále vznikají nové technologie a postupy, z jejichž používání vyplynou nové architektonické i návrhové vzory. Jde např. o v této práci zmíněný *Ajax*, jehož využití je dnes natolik rozšířené, že jistě lze nalézt opakující se vyzkoušené postupy, které půjde zevšeobecnit do nových vzorů. Podobně lze nahlížet i na postupy označované jako *Comet*, i když jejich využití je zatím na svém počátku. Rozhodně lze tedy říci, že vývoj webu bude i nadále poskytovat nová *naleziště* vzorů.

Tvorba této diplomové práce mi pomohla utřídit si své dosavadní znalosti webových aplikací a značným způsobem je rozšířit. Jsem přesvědčený, že je budu schopen aplikovat ve svých budoucích projektech a umožní mi navrhovat efektivní a mnohem elegantnější architektonická řešení, než jsem byl schopen doposud.

Bibliografie

[AJX] ASLESON Ryan, SCHUTTA Nathaniel. *Ajax : Vytváříme vysoce interaktivní webové aplikace*. Jakub Zemánek. 1. vyd. Brno : Computer Press, 2006. 269 s. ISBN 80-251-1285-3.

[ASC] MACDONALD Matthew, SZPUSZTA Mario. *ASP.NET 2.0 a C#*. Jan Pokorný. 1. vyd. Brno : Zoner software, s.r.o., 2006. 1676 s. Tvorba dynamických stránek profesionálně. ISBN 80-86815-38-2.

[CJP] ALUR, Deepak, CRUPI, John, MALKS, Dan. *Core J2EE Patterns : Best Practices and Design Strategies*. 2nd rev. edition. USA, CA : Prentice Hall PTR, 2003. 650 s. ISBN 0-13-142246-4.

[GoF] GAMMA, Erich, HELM, Richard, JOHNSON, Ralph, VLISSIDES, John. *Návrh programů pomocí vzorů : Stavební kameny objektově orientovaných programů*. Pavel Makovec. 1. vyd. Praha : Grada Publishing, 2003. 386 s. Moderní programování. ISBN 80-247-0302-5.

[HAX] MUSCIANO, Chuck, KENNEDY, Bill. *HTML a XHTML : Kompletní průvodce*. David Krásenský. 1.vyd. Praha : Computer Press, 2000. 633 s. Pro webové tvůrce a programátory. ISBN 80-7226-407-9.

[JE5] MUKHAR, Kevin, ZELENAK, Chris. *Beginning Java EE 5 : From Novice to Professional*. 1st edition. Berkeley, CA, USA : Apress, 2005. 641 s. The Expert's Voice In Java. ISBN 1-59059-470-3.

[JSJ] HALL, Marty. *Java servlety a stránky JSP*. Karel Prokeš. 1. vyd. Praha : Neocortex, 2001. 586 s. ISBN 80-86330-06-0.

[JSP] BURD, Barry. *JSP JavaServer Pages : Podrobný průvodce*. Bogdan Kiszka. 1. vyd. Praha : Computer Press, 2003. 381 s. ISBN 80-7226-804-X.

[JPP] SPELL, Brett. *Java Programujeme profesionálně*. Bogdan Kiszka. 1. vyd. Praha : Computer Press, 2002. 1022 s. Programmer to Programmer. ISBN 80-7266-667-5.

[PJS] CAVANESS, Chuck. *Programujeme Jakarta Struts : Tvorba webových aplikací se servlety a stránkami JSP*. Slavoj Písek. 1. vyd. Praha : Grada Publishing, 2003. 446 s. ISBN 80-247-0667-9.

[ROR] HOLZNER, Steven. *Začínáme programovat v Ruby on Rails*. Karel Voráček. 1. vyd. Praha : Computer Press, 2007. 384 s. Programmer to Programmer. ISBN 978-80-251-1630-2-

[SS2] ROUGHLEY, Ian. *Starting Struts 2*. 1st edition. Canada : Lulu.com, 2007. 111 s. Enterprise Software Development Series. Dostupný z WWW: <http://infoq.com/minibooks/starting-struts2>. ISBN 978-1-4303-2033-3.

Elektronické články

[AVS] ESPOSITO, Dino. *The ASP.NET View State* [online]. 2003 [cit. 2007-09-14]. Dostupný z WWW: <http://msdn.microsoft.com/msdnmag/issues/03/02/CuttingEdge/>.

[ESW] BERSVENDSEN, Arve. *Event Streaming to Web Browsers*. *Opera Labs* [online]. 2006 [cit. 2007-06-11]. Dostupný z WWW: <http://labs.opera.com/news/2006/09/01/>.

[IOC] FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern* [online]. 2004 [cit. 2007-9-11]. Dostupný z WWW: <http://martinfowler.com/articles/injection.html>.

[KPR] PICHLÍK, Roman. *Komety přilétají. Dagblog : blog nejen pro kodery* [online]. 2007 [cit. 2007-08-18]. Dostupný z WWW: <http://www.sweb.cz/pichlik/archive/2007_04_15_archive.html#8041729033016633360>.

[VHP] PATZER, Andrew. *Foundations of JSP design patterns: The View Helper pattern* [online]. 2004 [cit. 2007-08-18]. Dostupný z WWW: <http://www.javaworld.com/javaworld/jw-11-2004/jw-1101-viewhelper.html>.

[WAC] GEARY, David. *Web application components made easy with Composite View* [online]. 2001 [cit. 2007-10-01]. Dostupný z WWW: <http://www.javaworld.com/javaworld/jw-12-2001/jw-1228-jsptemplate.html>.

[WFJ] PICHLÍK, Roman. *Web frameworky v Javě. Dagblog : blog nejen pro kodery* [online]. 2006 [cit. 2007-05-31]. Dostupný z WWW: <http://www.sweb.cz/pichlik/archive/2006_12_10_archive.html#116578886450903306>.

Seznam použitých zkratek

- AJAX – Asynchronous JavaScript And XML
- API – Application Programming Interface
- CRUD – Create Read Update Delete
- DAO – Data Access Object
- DT – Data Transfer
- GUI – Graphical User Interface
- LIFO – Last In First Out
- MVC – Model View Controller
- POJO – Plain Old Java Object

Seznam obrázků

Obrázek 2.1 : Diagram nasazení popisující architekturu webové aplikace 8
Obrázek 3.1 : Ukázka UML notace sekvenčních diagramů 12
Obrázek 4.1 : Diagram komponent obecné MVC architektury 13
Obrázek 4.2 : Sekvenční diagram komunikace v klasické MVC architektuře 14
Obrázek 4.3 : Architektura MVC adaptovaná na webové prostředí 15
Obrázek 4.4 : Sekvenční diagram komunikační strategie Service to Worker 16
Obrázek 4.5 : Sekvenční diagram komunikační strategie Dispatcher View 16
Obrázek 5.1 : Sekvenční diagram front controller prvku v rámci strategie Service to Worker 24
Obrázek 5.2 : Diagram tříd front controller komponenty rámce Struts 26
Obrázek 5.3 : Diagram tříd front controller komponenty rámce Zend Framework 29
Obrázek 6.1 : Diagram tříd statické struktury vzoru Data Transfer Object 33
Obrázek 6.2 : Sekvenční diagram vytvoření a předání DT objektu 33
Obrázek 7.1 : Diagram komponent vzoru Intercepting Filter 37
Obrázek 8.1 : Sekvenční diagram zpracování dat modelu pomocí view helper komponenty 42
Obrázek 9.1 : Diagram tříd popisující rozdělení pohledů dle návrhového vzoru Composite 46
Obrázek 9.2 : Diagram tříd vzoru Composite View 47
Obrázek 9.3 : Sekvenční diagram vytvoření kódu pohledu pomocí vzoru Composite View 49
Obrázek 10.1 : Diagram popisující typy uživatelů a jejich základní akce 53
Obrázek 10.2 : Diagram popisující možné akce prováděné studentem 54
Obrázek 10.3 : Diagram popisující možné akce prováděné vyučujícím 56
Obrázek 10.4 : Diagram balíčků popisující jednotlivé vrstvy aplikace 60
Obrázek 10.5 : Diagram tříd popisující doménové entity z konceptuální úrovně 61
Obrázek 10.6 : Diagram tříd popisující strukturu aplikace 63
Obrázek 10.7 : Sekvenční diagram komunikace v MVC architektuře 64
Obrázek 10.8 : Stavový diagram popisující posloupnost obrazovek studenta 65
Obrázek 10.9 : Sekvenční diagram popisující vytvoření a předání DT objektu 67
Obrázek 10.10 : Obrazovka vyučujícího pro vytvoření nového projektu 74
Obrázek 10.11 : Obrazovka studenta s výpisem projektů předmětu 75

Seznam příloh

1. Obrázek 10.6 v plném rozlišení
2. Obrázek 10.7 v plném rozlišení
3. Obrázek 10.9 v plném rozlišení
4. CD-ROM nosič s následujícími adresáři a obsahem:
    - `/text`: text této práce v elektronických formátech
    - `/projectIS/src`: zdrojové soubory projektu společně dokumenty potřebnými k jejich zkompilování (ANT, NetBeans project)
    - `/projectIS/docs`: JavaDoc příručka
    - `/projectIS/readme`: příručka pro nasazení systému na server
    - `/projectIS/bin`: zkompilovaný `.war` soubor určený k nasazení na server