České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačů

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Ondřej Kratochvíl**

Studijní program: Softwarové technologie a management
Obor: Softwarové inženýrství

Název tématu: **Využití byznys pravidel ve formulářích informačních systémů**

Pokyny pro vypracování:

Proveďte rešerši reprezentace byznys pravidel v informačních systémech. Navrhněte možnosti využití byznys pravidel zachycených pomocí aspektově-orientovaného přístupu k návrhu systému. Implementujte knihovnu využívající byznys pravidla ve formulářích pro validaci vstupních hodnot a zlepšení uživatelské přívětivosti. Knihovnu demonstrujte na několika ukázkových příkladech. Součástí práce musí být dokumentace skládající se z popisu architektury řešení a návod k použití. Nakonec proveďte analýzu budoucí rozšiřitelnosti a možností uplatnění navrženého řešení.

Seznam odborné literatury:

[1] BERNARD, Emmanuel; PETERSON, Steve. JSR 303: Bean validation. Bean Validation Expert Group, March, 2009.
[2] CEMUS, Karel; CERNY, Tomas. Aspect-Driven Design of Information Systems. 2014. ISBN 978-3-319-04297-8.
[4] CEMUS, Karel; CERNY, Tomas; DONAHOO, Michael J. Evaluation of approaches to business rules maintenance in enterprise information systems. In: Proceedings of the 2015 Conference on research in adaptive and convergent systems. ACM, 2015. p. 324-329.
[5] CERNY, Tomas, et al. Aspect-driven, data-reflective and context-aware user interfaces design. ACM SIGAPP Applied Computing Review, 2013, 13.4: 53-66.
[6] FOWLER, Martin. Domain-specific languages. Pearson Education, 2010.

Vedoucí: Ing. Karel Čemus

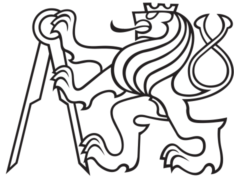
Platnost zadání: do konce letního semestru 2016/2017

prof. Ing. Filip Železný, Ph.D.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 12. 2015

Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Využití byznys pravidel ve formulářích informačních systémů

Ondřej Kratochvíl

Vedoucí: Ing. Karel Čemus
Obor: Softwarové technologie a management
Studijní program: Softwarové inženýrství
Květen 2016

Poděkování

Rád bych poděkoval Ing. Karlovi Čemusovi za odborné vedení bakalářské práce, za jeho cenné rady při konzultacích a za ochotu při poskytování zpětné vazby. Dále bych chtěl také poděkovat Filipovi Klimešovi za spolupráci při návrhu a vývoji knihovny.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 27. května 2016

Ondřej Kratochvíl

Abstrakt

Byznys pravidla tvoří nedílnou součást informačních systémů, která slouží k omezení procesů a operací, ze kterých se systémy skládají. Současný přístup k návrhu aplikací však neumožňuje jejich efektivní reprezentaci, neboť cross-cutting povaha byznys pravidel vynucuje v objektově orientovaném návrhu manuální duplikaci kódu, která výrazně ztěžuje jejich deklaraci i údržbu.

Tato práce se zaměřuje na řešení problémů spojených se zachycením byznys pravidel a jejich interpretaci v uživatelském rozhraní s důrazem na izolaci od výkonného kódu a sdružení do centrální znalostní báze. K tomu je využito aspektově orientované paradigma, jež umožňuje identifikovat cross-cutting problémy jako jednotlivé aspekty a následně je interpretovat do cílové domény. Tento přístup usnadňuje tvorbu byznys pravidel a jejich následnou údržbu, snižuje náchylnost systému k chybám a redukuje množství duplicitního kódu.

Klíčová slova: Aspektově orientované programování, byznys pravidla, uživatelské rozhraní, validace

Vedoucí: Ing. Karel Čemus

Abstract

Business rules constitute a vital part of information systems, that constraint its processes and operations. However, contemporary approach to software design struggles with their effective description, as cross-cutting nature of business rules causes vast code duplication in object oriented design, which increases maintanence efforts.

This thesis aims to solve issues concerning business rules description and their interpretation into user interface, and focuses on untangling them from complex application logic and capturing them into central knowledge base. Aspect-oriented paradigm is used to identify these cross-cutting concerns as individual aspects and subsequently interpret them into target domain. This approach eases the declaration and maintanence of business rules, mitigates the risks of errors and reduces the amount of code duplication.

Keywords: Aspect oriented programming, business rules, user interface, validation

Title translation: Use of Business Rules in Forms in Information Systems

Obsah

1 Úvod 1

2 Analýza 3
2.1 Byznys pravidla . . . 5
2.1.1 Typy byznys pravidel . . . 6
2.2 Využití byznys pravidel . . . 7
2.2.1 Datová vrstva . . . 7
2.2.2 Prezentační vrstva . . . 8
2.2.3 Aplikační vrstva . . . 9
2.3 Problémy . . . 9
2.4 Shrnutí . . . 11

3 Rešerše 13
3.1 Stávající řešení . . . 13
3.1.1 Naivní přístup . . . 13
3.1.2 JSR 303 . . . 14
3.1.3 RichFaces . . . 15
3.2 Aspektově orientované programování . . . 16
3.3 Aspect-driven design approach . . . 18
3.4 Drools . . . 18
3.5 Implementace ADDA konceptu v UI . . . 20
3.5.1 AspectFaces . . . 21
3.5.2 Nutforms . . . 22
3.6 Architektura Nutforms . . . 24
3.6.1 Rich Model . . . 24
3.7 Shrnutí . . . 26

4 Návrh 27
4.1 Validace pomocí AOP . . . 27
4.2 Integrace validace do Nutforms . . . 29
4.2.1 Deklarace pravidel . . . 29
4.2.2 Inspekce pravidel . . . 30
4.2.3 Životní cyklus formuláře . . . 30
4.2.4 Stav validace . . . 32
4.3 Shrnutí . . . 34

5 Implementace 35
5.1 Použité technologie . . . 35
5.1.1 Java . . . 35
5.1.2 JavaScript . . . 36
5.1.3 Drools . . . 36
5.2 Server side . . . 36
5.2.1 Deklarace pravidel . . . 37
5.2.2 Rule parser . . . 38
5.2.3 Rule servlet . . . 38
5.3 Client side . . . 39
5.3.1 Aspect weaver . . . 39
5.3.2 Vyhodnocení pravidel . . . 40
5.3.3 Odeslání formuláře . . . 40

6 Testování 41
6.1 Testy implementace . . . 41
6.1.1 Použité technologie . . . 41
6.1.2 Typy testů . . . 42
6.2 Ukázková aplikace . . . 43
6.2.1 Integrace knihovny . . . 45

7 Závěr 47
7.1 Přínos práce a omezení . . . 48
7.2 Budoucí rozvoj . . . 48
7.3 Shrnutí . . . 49

Literatura 51

A Seznam použitých zkratek 55

B Instalační příručka 57
B.1 Nutforms . . . 57
B.2 Ukázková aplikace . . . 58

C Snímky obrazovky 59

D Obsah přiloženého CD 61

Obrázky

2.1 Schéma třívrstvé architektury . . . 4

3.1 RichModel knihovny Nutforms . 25

4.1 Diagram aktivit při změně hodnoty 31
4.2 Diagram aktivit při odeslání formuláře 32
4.3 Diagram změn stavů formulářových polí 33

5.1 Diagram komponent knihovny a jejich komunikace 37

6.1 Doménový model ukázkové aplikace 44

C.1 Zpětná vazba pro nevalidní hodnoty při vytvoření problému . . 59
C.2 Zpětná vazba pro nevalidní model při editaci zaměstnance 60
C.3 Needitovatelná pole pro kontext *read* 60

Zdrojové kódy

3.1 Ukázka validace pomocí JSR 303 14
3.2 Validace v UI pomocí JSF 15
3.3 Validace v UI pomocí RichFaces 16
3.4 Deklarace pravidla v Drools DSL 19
3.5 Datový model v AspectFaces 21
3.6 Vytvoření formuláře v AspectFaces 22
3.7 Generický layout v AspectFaces 22
4.1 Deklarace pravidla v knihovně Nutforms 30
6.1 Ověření predikátu v knihovně Chai 42
6.2 Deklarace pravidla v ukázkové aplikaci 45

Kapitola 1

Úvod

Informační systémy jsou v dnešní době neodmyslitelnou součástí téměř všech profesí i každodenního života. Usnadňují správu dat a jejich třídění, umožňují automatizaci rutinních operací a celkově zvyšují efektivitu práce. Důležitou součástí těchto aplikací je uživatelské rozhraní, které by mělo působit přehledně a poskytovat snadno dostupné informace, spolu s intuitivní navigací po stránkách. Z pohledu vývoje těchto systémů je právě uživatelskému rozhraní věnována nejvyšší pozornost [25], jelikož uživatelsky nepřívětivé ovládání aplikace výrazně snižuje její konkurenceschopnost.

Součástí většiny systémů jsou byznys pravidla, jež definují omezení nad prováděnými operacemi a následné podmínky po jejich provedení. Ta poskytují mimo jiné kontrolu vstupních dat a zajišťují jejich konzistenci napříč celou aplikací. Zachycení byznys pravidel v informačních systémech je však nelehkým úkolem, jelikož jejich provázanost s jednotlivými částmi aplikace často končí manuální duplicitou kódu. Zejména v systémech s vrstevnatou architekturou [22] je problém interpretace stejných pravidel v různých vrstvách, jelikož jsou často použity rozdílné technologie pro jejich implementaci.

Tato roztříštěnost jednotlivých byznys pravidel v systému a provázanost s výkonným kódem značným způsobem zvyšuje náročnost deklarace pravidel a jejich údržby. Jakékoliv změny musí být provedeny na všech příslušných místech, což je proces velmi náchylný k chybám způsobených opomenutím některých z míst aplikace pravidla, vedoucí k jeho nekonzistentnímu stavu. Navíc, manuální duplicita zvyšuje komplexitu a snižuje čitelnost kódu, tedy celkově dochází k nárůstu času a nákladů na vývoj systému.

Současná řešení usilují o redukci této duplicity, jako například *Hibernate Validator* [17], zaměřující se na znovupoužití validačních pravidel deklarovaných nad datovým modelem prostřednictvím meta-informací v databázi a vrstvě byznys logiky. Bohužel však žádná z knihoven neumožňuje jejich opětovné použití ve všech vrstvách aplikace, případně se soustředí pouze na jeden typ byznys pravidel.

Řešení výše zmíněných problémů lze řešit separací byznys pravidel od výkonného kódu do centrální znalostní báze, odkud budou interpretována do příslušných míst. Jednotlivá pravidla zde budou rozdělena do skupin podle operace, ke které se vztahují, přičemž jedno pravidlo může patřit do více skupin. Tento přístup umožňuje jejich snadnou údržbu, usnadňuje přidávání

nových pravidel a zejména výrazně redukuje množství duplicitního kódu.

Cílem této práce je analýza problematiky reprezentace byznys pravidel a jejich použití v uživatelském rozhraní s důrazem na znovupoužitelnost v ostatních vrstvách. Součástí je také návrh a implementace knihovny, jejímž účelem je snaha o vyřešení zmíněných problémů a přiblížení se výše popsanému ideálnímu stavu. Kapitola 2 se věnuje analýze domény a definuje použité pojmy. V kapitole 3 jsou představeny současné přístupy k řešení problematiky a popsány jejich výhody a nevýhody. 4. kapitola přináší návrh vlastního řešení, přičemž v kapitole 5 je popsána jeho implementace. Ověření funkčnosti navrženého přístupu je věnována kapitola 6, jež se věnuje testování a demonstrace použití knihovny na ukázkové aplikaci. Nakonec jsou v kapitole 7 shrnuty poznatky celé práce a jsou vyhodnoceny přínosy navrženého konceptu, jeho omezení a možnosti budoucí rozšiřitelnosti.

Kapitola 2

Analýza

Webová aplikace (WA) je velice rozsáhlý pojem, který pokrývá širokou škálu produktů, od velice malých jednoúčelových aplikací, jejichž životnost je maximálně pár měsíců, až po velké korporátní informační systémy, které jsou používány mnoho let.

Definice 1. *Webová aplikace je klient-server aplikace, která poskytuje interaktivní služby prostřednictvím Internetu. Na straně klienta je často využíván prohlížeč. [32]*

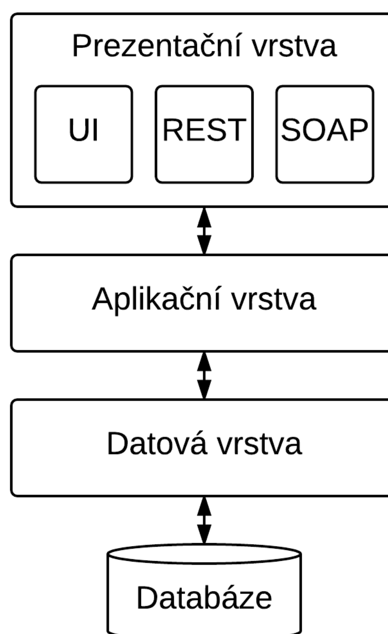
Komunikace mezi klientem a serverem probíhá prostřednictvím HTTP protokolu [18], který je založen na vzájemném odesílání a přijímání požadavků, pomocí kterých dochází k výměně informací. Zásadním rozdílem mezi WA a webovou stránkou je implementace byznys logiky v aplikacích [11]. Ta je místem pro definici byznys procesů a operací, které určují tok aplikace a mění její byznys stav.

S ohledem na typ a velikost aplikace se mění její struktura, která je často založena na vrstevnaté architektuře [32] s případným využitím architektonických vzorů (např. MVC, který je v mnoha případech základem webových UI (*user interface*) frameworků [22]).

Naivní řešení je architektura mající pouze jednu vrstvu, kde jsou všechny prvky aplikace sdruženy na jednom místě a neexistuje žádná hranice, kde bychom mohli oddělit např. prezentaci dat od jejich ukládání. Tento typ architektury má výhodu v jednoduchosti implementace a hodí se spíše pro menší aplikace, ve kterých by komplexní architektura obnášela nepřiměřeně náročnou počáteční režii. Pro větší aplikace s delší dobou vývoje, či s častými změnami, však toto řešení není vhodné, jelikož s rostoucím množstvím kódu se bude z důvodu jeho nestrukturovanosti snižovat přehlednost, což také bude zvyšovat náročnost při rozšiřování takových aplikací.

Vícevrstvá architektura je založena na separaci aplikace do několika nezávislých komponent, které spolu komunikují. Tyto komponenty sdružují funkční celky, které určují účel každé vrstvy. Velmi rozšířeným typem je třívrstvá architektura (obr. 2.1), kde je aplikace rozdělena na tři vrstvy - prezentační, aplikační (logická/byznys) a datová (perzistentní) [22]. Prezentační vrstva je vstupním bodem pro uživatele a zajišťuje zobrazování dat, např. prostřednictvím webové stránky, nebo GUI (*graphical user interface*). Samotná data jsou zpracovávána v aplikační vrstvě, která bývá nazývána

také jako logická, jelikož je zodpovědná za zpracování požadavků uživatele, provádění výpočtů a určování toku aplikace. Datová vrstva slouží k uchování dat a poskytuje rozhraní pro jejich získávání. Komunikace mezi vrstvami by měla probíhat vždy pouze přes nejbližší vrstvu, tedy v případě třívrstvé architektury by měla komunikace mezi prezentační a datovou vrstvou vždy směřovat přes logickou vrstvu. Mezi výhody vrstevnaté architektury patří škálovatelnost a zejména nezávislost jednotlivých vrstev. Je tedy možné vyměnit implementaci kterékoliv z vrstev za jinou [22], bez nutnosti zasahování do kódu v ostatních vrstvách (za předpokladu zachování stejného rozhraní či poskytnutí příslušného adaptéru). S rozdělením do vrstev však také souvisí řada nevýhod - vedle vyšší počáteční režie také integrace změn [22], jelikož z důvodu zapouzdření jednotlivých komponent je třeba změnu provádět ve více vrstvách. Například pro přidání nového atributu je nutné přidat sloupec do databáze, změnit logiku v byznys vrstvě a přidat pole do formuláře v UI.



Obrázek 2.1: Schéma třívrstvé architektury

Duplicitní kód je z hlediska vývoje aplikace neefektivní řešení, které zvyšuje komplexitu systému a způsobuje jeho náchylnost k chybám vytvořeným při údržbě aplikace a integraci změn [1]. Redukce této duplicity přinese nejvíce benefitů pro velké enterprise aplikace, které mají často třívrstvou architekturu, proto se v této práci budu věnovat zejména této architektuře. Menší aplikace využívající MVC, či jiné vzory, sice také obsahují duplicitní kód, nicméně rozdíl po jeho odstranění není ve srovnání s velkými aplikacemi tak markantní. Na obr. 2.1 je naznačena možná dekompozice prezentační vrstvy na jednotlivé komponenty, mezi nimiž může také docházet k duplicitním deklaracím, jejichž

redukce je také zohledněna v této práci.

2.1 Byznys pravidla

Jak je zmíněno výše, specifickým znakem pro WA je byznys logika. Ta se skládá z velkého množství byznys procesů [27], jež popisují akce, které aplikace umožňuje vykonávat. Příkladem takového procesu může být například provedení bankovní transakce. Procesy je možné rozdělit na jednotlivé kroky (operace) [6] - přihlášení do online bankovnictví, vyplnění formuláře pro odeslání platby a potvrzení heslem. Byznys procesy, stejně jako operace, ze kterých se skládají, je možné omezit byznys pravidly.

Definice 1. *Byznys pravidlo je množina předpokladů (precondition) a následných podmínek (post-condition), které se vztahují k byznys procesům, nebo k jejich krokům.*

Před vykonáním každé byznys operace je ověřena platnost předpokladů všech pravidel, které se k této operaci vztahují, a po jejím vykonání jsou zkontrolovány následné podmínky. Pro provedení operace je tedy potřeba získat nejprve seznam pravidel a aktuální stav aplikace pro korektní vyhodnocení podmínek. Pro tyto účely je v [6] definováno několik kontextů[1]:

1. **Application context** obsahuje globální proměnné aplikace, jako např. IP adresu serveru, zátěž, příznaky vývojového/produkčního nastavení, či údržby serveru.

2. **Business context** je seznam předpokladů a následných podmínek [6], které se vztahují k jednotlivým byznys operacím.

3. **User context** obsahuje údaje specifické pro aktuálního uživatele. V tomto kontextu nalezneme proměnné, jako uživatelské jméno, nebo bezpečnostní role. Tento kontext umožňuje mj. personalizaci stránek podle preferencí uživatele, např. zobrazení obsahu v požadovaném jazyce.

4. **Request context** je místem pro získávání dat z aktuálního HTTP požadavku. V rámci tohoto kontextu nalezneme proměnné, jako např. IP adresu uživatele, datum odeslání požadavku, nebo údaje o zařízení, které uživatel používá k zobrazení stránky.

5. **Execution context** sdružuje výše uvedené kontexty, tedy obsahuje všechny potřebné údaje pro korektní provedení byznys operace.

Při vykonávání byznys operace je třeba vyhodnotit proměnné z aktuálního execution kontextu 5. Nejprve musíme před provedeném operace vytvořit instanci business kontextu, ze kterého získáme byznys pravidla obsahující předpoklady a následné podmínky. Jednotlivá pravidla obsahují proměnné, na které navážeme hodnoty z execution kontextu, což umožní reflexi aktuálního

[1] V této práci budu užívat anglických názvů kontextů, jak jsou uvedeny v [6]

stavu aplikace. Poté je možné pravidla vyhodnotit a pokud nedojde k chybě, můžeme aplikovat následné podmínky.

Triviálním příkladem byznys pravidla může být *"Uživatelské jméno nesmí být delší, než 30 znaků"*. Takové pravidlo plyne z procesu *registrace nového uživatele*, které určuje byznys kontext, který bude při provedení této operace a jejích procesů použit. Ten může obsahovat další pravidla, jako například omezení týkající se délky hesla, nebo minimálního věku uživatele. Pro vyhodnocení tohoto pravidla je třeba získat z request kontextu jméno uživatele, které bylo odesláno ve formuláři. Pokud všechna pravidla z business kontextu proběhnou bez chyb, dojde k vytvoření nového uživatele a jeho jméno a další údaje budou uloženy do user kontextu.

Aby bylo byznys pravidlo kompletní, je třeba specifikovat chování při jeho porušení. To je vyjádřeno následnými podmínkami, které se mohou lišit nejen pro různá pravidla, ale také pro stejné pravidlo v závislosti na vrstvě, ve které se pohybujeme. V prezentační vrstvě to může být zobrazení vizuální zpětné vazby, která upozorní uživatele o chybě, v aplikační vrstvě odeslání chybového HTTP statusu a v perzistentní vrstvě vrácení chyby při porušení integritního omezení a neuložení entity. Plná deklarace výše uvedeného pravidla v pseudokódu bude mít tedy následující podobu:

Context: user/new
Entity: cz.cvut.fel.entity.UserEntity
Name: Is username properly long
Precondition: userName.length <= 30
Post-condition: save user

Algoritmus 1: Byznys pravidlo zapsané v pseudokódu

2.1.1 Typy byznys pravidel

Byznys pravidla můžeme podle jejich významu rozdělit do několika základních skupin [35]:

1. **Integritní omezení** – Tento typ pravidel vychází z možností, které jsou k dispozici v perzistentní vrstvě. Databázové systémy umožňují definovat množinu omezení, vázajících se na jednotlivé tabulky a jejich sloupce. Pomocí tohoto typu pravidel můžeme validovat povinnost atributů, minimální/maximální délku řetězce, regulární výraz, číselný rozsah apod.

2. **Validace objektů** – Jelikož integritní omezení nabízejí pouze základní sadu pravidel, pro složitější podmínky můžeme využít validátory nad objekty. Ty definuje například standard *Bean Validation* [4], který používá k deklaraci pravidel meta-instrukce programovacího jazyka, konkrétně v Javě využívá anotace. Nespornou výhodou oproti integritním omezením je možnost definovat vlastní validátory, tedy základní sadu je možno rozšiřovat vytvořením vlastní třídy.

3. **Bezpečnostní pravidla** – Byznys pravidla mohou také sloužit k zabezpečení nebo autorizaci. Pomocí tohoto typu pravidel je možné kontrolovat, zda je uživatel oprávněn k provedení požadované akce, nebo zda má např. dostatečná práva na zobrazení či úpravu vybraných elementů na stránce.

Kromě významu můžeme byznys pravidla také rozdělovat podle jejich povahy. Nejjednodušším typem jsou pravidla, která se vážou pouze na jednu proměnnou z execution kontextu. Takové validační pravidlo je výše uvedený příklad s kontrolou délky uživatelského jména, bezpečnostní pravidlo tohoto typu je např. kontrola, zda je uživatel přihlášen. Tyto podmínky lze spojovat logickými operátory, čímž jsou tvořena komplexnější pravidla, která mohou nejen kontrolovat více proměnných z jednoho kontextu, ale také proměnné napříč různými kontexty. Uvažme například dražební portál, ve kterém není možné upravit položku po začátku dražby. Pokud je však uživatel administrátor, může upravit i právě probíhající dražbu. Před provedením této operace je tedy třeba zjistit, zda má uživatel roli admin (user context), nebo zda je aktuální datum (application context) dříve, než datum začátku upravované dražby (request context).

Dále je třeba také uvážit pravidla, která nelze navázat na jednotlivé atributy, ale souvisí s celým modelem. Takové pravidlo bychom využili například při registraci nového uživatele - ve formuláři bude kromě jména a příjmení také adresa, země a město. Pokud je však uživatel z USA, musí také vybrat stát z nabídky. Rozdělení pravidel podle povahy se budu blíže věnovat v návrhové a implementační části.

2.2 Využití byznys pravidel

Jelikož důležitou součástí webových aplikací je zpracování vstupních dat, je třeba tyto vstupy ošetřit, než nad nimi začneme provádět operace. V první řadě je třeba zkontrolovat, zda splňují požadovaný formát (jméno zákazníka neobsahuje čísla, PSČ je ve správném formátu apod.), abychom se vyvarovali chyb hlouběji v aplikaci. Dále je však také třeba myslet na škodlivé vstupy - zejména *code injection*, kterému může být zabráněno filtrováním vstupu (*input filtering*) a ošetřením výstupu (*output sanitation*) [24].

Právě pro výše zmíněné účely jsou byznys pravidla standardním řešením. Webové aplikace běžně obsahují podle velikosti stovky až tisíce takových pravidel, díky kterým jsou data napříč aplikací konzistentní a jsou splněny základní požadavky na jejich strukturu.

2.2.1 Datová vrstva

Nejnižší úrovní pro definici byznys pravidel je datová vrstva. Základním prvkem datového modelu je entita, jenž reprezentuje jednotlivé byznys objekty, se kterými v aplikaci pracujeme, obsahuje atributy a vazby na další entity. Entity jsou ukládány a čteny z databáze, pro jejich transformaci do objektů v daném programovacím jazyce je často využíváno ORM frameworků (*object-*

relational mapping), v Javě např. Hibernate [2]. Velké množství byznys pravidel se váže právě k entitám, jelikož tvoří velkou část dat v aplikaci, a uživatel je může často modifikovat. V první řadě je třeba zajistit, aby údaje v databázi byly validní a vyhovovaly předem definovaným požadavkům, tedy již na úrovni datové vrstvy je třeba definovat množinu pravidel pro jednotlivé entity. Pokud je při jejím ukládání podmínka některého z pravidel porušena, měla by post-condition daného pravidla zajistit, že nevalidní instance nebude do databáze uložena. Díky tomu můžeme zaručit, že již uložená data budou při čtení z databáze validní.

Integritní omezení v databázi lze definovat v rámci jazyka SQL, který nabízí základní struktury pro tvorbu byznys pravidel [14]. Nejjednodušší pravidla jsou přímo klíčovými slovy - `NOT NULL`, které slouží pro kontrolu prezence hodnoty a `UNIQUE`, pomocí kterého zaručíme, že v rámci daného sloupce tabulky bude hodnota unikátní. Složitější pravidla je možné definovat pomocí klíčového slova `CHECK`, s nímž je možné porovnávat hodnoty jednotlivých sloupců a například omezit číselný rozsah numerického sloupce. V rámci jedné `CHECK` klauzule je možné spojovat několik porovnání logickými výrazy a tvořit složitější podmínky.

Vzhledem k tomu, že integritní omezení neumožňují vytvářet komplexnější pravidla a většinou slouží pouze pro kontrolu jednoduchých podmínek, dochází k jejich duplikaci. Některé ORM frameworky částečně řeší znovupoužitelnost pravidel a redukci duplicity - například Hibernate Validator [3] umožňuje transformaci validátorů nad objekty do integritních omezení databáze v perzistentní vrstvě [6], nicméně znovupoužitelnost těchto pravidel v UI bohužel není možná, tedy v prezentační vrstvě dochází k jejich duplikaci.

2.2.2 Prezentační vrstva

Úpravy dat realizuje uživatel prostřednictvím formulářů. Velice často jsou jednotlivá formulářová pole namapována přímo na atributy entity, eventuálně na atributy DTO (*data transfer object*), objektu, do kterého jsou uložena data z formuláře a v aplikační vrstvě jsou teprve uloženy do entity. Jelikož mnoho pravidel pro danou entitu je možné vyhodnotit na straně klienta, setkáváme se i v prezentační vrstvě s velkým množstvím validačních i jiných pravidel. Díky kontrole dat již u klienta se vyhneme odesílání nevalidního formuláře na server a jeho opětovné vykreslování s příslušnými chybovými hláškami na straně klienta, čímž ušetříme čas na zpracování požadavku. Navíc, z hlediska UX (*user experience*) je pro uživatele lepší, když vidí zpětnou vazbu okamžitě při změně hodnoty pole během vyplňování formuláře, tedy může chyby opravit dříve, než formulář odešle.

Pro klientskou validaci je většinou využíváno funkcí psaných v JavaScriptu, které jsou volány např. při změně hodnoty, nebo při odeslání formuláře. Pokud některá z hodnot neprojde validačním pravidlem, nedojde ani k odeslání formuláře. Kromě JavaScriptu je k validaci také možné využít HTML5 atri-

[2]http://hibernate.org
[3]http://hibernate.org/validator

butů [12], pomocí kterých můžeme kontrolovat povinná pole, datový typ hodnoty, číselný rozsah, či kontrolovat hodnotu regulárním výrazem.

Využití byznys pravidel na straně klienta má však za následek další duplikaci, jelikož stejná pravidla musí být opětovně kontrolována i na serveru. Knihovna *RichFaces* [4] tento problém řeší transformací byznys pravidel deklarovaných pomocí meta-instrukcí nad datovým modelem do validátorů v prezentační vrstvě, k čemuž je využit standard *JSR 303: Bean Validation*. Některá pravidla však pouze na straně klienta bez dat ze serveru není možné vyhodnotit - takovým pravidlem může být kontrola unikátnosti pole, tedy např. zda v systému již není zaregistrován uživatel se zadaným emailem.

2.2.3 Aplikační vrstva

Velice hojně jsou byznys pravidla také využívána v aplikační vrstvě, kde je možné identifikovat jejich různé typy. V případě přijetí formuláře od klienta zde dochází k opětovné kontrole správnosti dat pomocí stejných validačních pravidel navázaných na upravovanou entitu, a to z několika důvodů. Za prvé, validace na straně klienta nemůže být nikdy považována za dostatečnou, jelikož je většinou realizována pomocí JavaScriptu (ten však může být ve většině prohlížečů úplně zakázán), nebo v poslední době také HTML5 (které není podporováno ve všech prohlížečích, zejména v těch starších). Navíc, data přijatá od klienta by měla být validována na serveru za všech okolností, zejména z bezpečnostních důvodů. Útočník může aplikaci podstrčit libovolná data, či úplně obejít kontrolu na straně klienta, například přímým posíláním HTTP požadavků. K reprezentaci validačních pravidel lze využít standard *JSR 303* [4] a jeho implementaci Hibernate Validator, což částečně umožňuje znovupoužitelnost definovaných pravidel.

Kromě validačních pravidel jsou v byznys vrstvě také definována bezpečnostní pravidla, která slouží k autorizaci, tedy ke kontrole, zda má uživatel dostatečná práva na provedení požadované akce. Ta jsou využívána zejména při použití uživatelských rolí ve větších informačních systémech, ale také pro rozlišení přihlášených a nepřihlášených uživatelů. Implementaci bezpečnostních pravidel umožňuje například technologie *Spring Security* [5] [29], nebo *Jave Enterprise Edition* [15].

2.3 Problémy

S rostoucím počtem nových technologií také přibývají nové možnosti, jak reprezentovat byznys pravidla v jednotlivých vrstvách. Vazba na vrstvy je však zásadním omezením pro flexibilní práci s pravidly v informačních systémech. V sekci 2.2 je nastíněn problém s duplikací stejných pravidel využívajících tytéž kontexty napříč vrstvami. Výše zmiňované technologie jako *Hibernate* nebo *RichFaces* tento problém částečně řeší, ale bohužel žádná z nich neumožňuje znovupoužití pravidla v celé aplikaci, ale většinou pouze ve dvou vrstvách. V

[4] http://richfaces.jboss.org/

[5] http://projects.spring.io/spring-security

důsledku toho jsou pravidla duplikována, což zvyšuje náchylnost k chybám při změnách, či při přidávání nových pravidel, neboť lze snadno zapomenou přidat pravidlo na všechna potřebná místa.

Uvažme následující situaci: v jednoduché aplikaci pro vytváření úkolů jsou entity *uživatel*, *úkol* a *kategorie*. Uživatel může vytvářet úkoly a sdružovat je do kategorií, které může taktéž vytvářet, nebo mazat, a ke každé z entit se váže několik byznys pravidel, která definují omezení pro hodnoty jejich atributů (např. text úkolu nesmí být prázdný, název kategorie nesmí být delší, než 50 znaků atp.). V první řadě je třeba vytvořit restrikci na úrovni databáze, kde budeme při každé změně či uložení nové entity kontrolovat, zda jsou všechna integritní omezení splněna. Dále na úrovni byznys logiky definujeme jednotlivé procesy pro provádění CRUD operací (*create, read, update, delete*), ve kterých budeme také kontrolovat, zda při vytváření nové entity či modifikaci již existující není porušeno některé z pravidel. Při mazání kategorie je navíc třeba kontrolovat, zda neexistují úkoly, které mají vazbu na danou kategorii. Pokud ano, je možné situaci vyřešit dvěma způsoby - buď tyto vazby zrušit, tedy některé úkoly nebudou začleněny do žádné kategorie, nebo úkoly smazat. Toto chování je v rámci jazyka SQL implicitně kontrolováno na úrovni databáze. Nakonec je třeba vytvořit v prezentační vrstvě jednotlivé stránky pro vytváření, úpravu a mazání uživatele, kategorie a úkolu a nad formuláři vytvořit funkce pro kontrolu byznys pravidel.

Jak je patrné, pro každou byznys operaci jsou jednotlivá pravidla definována na několika místech, přestože vyjadřují stejné omezení a přistupují k proměnným ze stejných kontextů. Problém nastává ve chvíli, kdy některé z pravidel chceme upravit, smazat, nebo přidat nové. Tuto úpravu je pro zachování integrity pravidel nutné udělat na všech místech, kde se pravidlo vyskytuje, což se s velikostí aplikace stává podstatně náročnější. Vzhledem k tomu, že změny byznys pravidel jsou v informačních systémech poměrně časté, vzniká touto cestou v důsledku opakované definice pravidel velké množství chyb. Navíc, duplicita zde není jen ve směru vertikálním, tedy napříč vrstvami, ale i horizontálním, tedy v rámci jedné vrstvy [6].

Deklarace pravidel tímto způsobem často znemožňuje znovupoužití stejného pravidla na jiném místě, čímž by bylo možné redukovat duplikáty pravidel. Z důvodu rozdílných technologií v různých vrstvách aplikace jsou navíc tatáž pravidla deklarována rozdílným způsobem a často i v jiném programovacím jazyce. Pro co nejsnazší údržbu a vývoj by bylo ideálním řešením mít veškerá byznys pravidla na jednom místě s tím, že pro změny v požadavcích by stačilo upravit definici pouze na tomto jednom místě a změny by se propagovaly na všechna místa, kde je pravidlo použito.

Uvažme ještě různé způsoby zobrazování obsahu, tedy například rozdílný vzhled pro mobilní zařízení, tablety a širokoúhlé displeje. V každém layoutu je třeba duplikovat veškerá byznys pravidla, z čehož je patrné, že s rozšiřováním aplikace se stává údržba těchto pravidel velice náročnou prací, během které lze snadno opomenout upravit pravidlo na některém z míst.

2.4 Shrnutí

Pro většinu webových aplikací jsou byznys pravidla nedílnou součástí. Tvoří důležitou součást procesů v doméně, které definují určité požadavky na stav dat, které by neměly být nikdy porušeny, aby byla zachována integrita a konzistence dat, a umožňují kontrolovat uživatelské vstupy a vyhodnocovat validační podmínky. Pomocí byznys pravidel je ale také možné řídit kontrolu nad procesy v aplikaci, případně omezovat operace. S jejich pomocí můžeme řešit autorizaci v jednotlivých částech aplikace, nebo redukovat množinu dat, které může uživatel s danou rolí upravovat.

Zásadním problémem v současné implementaci byznys pravidel v informačních systémech je rozsáhlá duplicita kódu, týkajícího se těchto pravidel. V aplikacích s dnes standardní třívrstvou architekturou je stejné pravidlo nad jednou entitou definováno v každé ze tří vrstev, navíc často dochází k duplicitám i v rámci jedné vrstvy, kde by mohlo být jedno pravidlo použito pro více různých účelů. Zejména v prezentační vrstvě dochází k duplicitám v různých layoutech rozlišených podle zařízení, nebo rozměru displeje. Podle [25] tvoří UI 48% kódu aplikace a polovina času je věnována jeho implementaci, tudíž neefektivita kódu v této vrstvě může vést k zásadním problémům.

Znovupoužitelnost pravidel by přispěla k čitelnosti kódu, výrazně by zjednodušila úpravy stávajících pravidel i definice nových a redukovala množství chyb spojených s těmito změnami (např. zapomenutí změny pravidla na jednom z míst jeho deklarace). Sdružení pravidel by navíc umožnilo jejich extrahování ze spletité byznys logiky a soustředění na jednom místě. Kód by se pak stal přehlednější a změny mnohem jednodušší. V neposlední řadě by byl tento způsob zvyšoval spolehlivost systému, jelikož jakoukoliv změnu by stačilo provést pouze na tomto jednom místě. Z pohledu managementu softwaru by sdružení pravidel mohlo také sloužit k validaci požadavků a jejich testovatelnost by byla mnohem snazší, jelikož je tímto způsobem zajištěna konzistence pravidel napříč celou aplikací.

Kapitola 3
Rešerše

V kapitole 2 byl zmíněn problém duplikace byznys pravidel napříč aplikací, jenž znemožňuje efektivní vývoj a údržbu těchto pravidel. Cílem této práce je analyzovat stávající možnosti reprezentace byznys pravidel v aplikacích a navrhnout řešení jejich využití v uživatelském rozhraní s důrazem na znovupoužitelnost pravidel.

3.1 Stávající řešení

3.1.1 Naivní přístup

Pro reprezentaci byznys pravidel není vždy nutné nasazovat dodatečné technologie, nebo používat komplexní frameworky. Pro velmi malé aplikace může být výhodné definovat pravidla spolu s byznys logikou pouze pomocí standardních nástrojů jazyka, nejjednodušeji jako `if-else` podmínku před vykonáním dané byznys operace. Toto řešení je velice snadným způsobem pro zachycení omezení procesů domény, navíc nevyžaduje žádnou režii navíc. Pochopitelně je to však za cenu zapletení pravidel do byznys logiky, která sice nemusí být pro malé aplikace složitá, ani rozsáhlá, nicméně jakákoliv budoucí modifikace vyžaduje lokalizaci všech výskytů pravidla, což může být nelehkým úkolem, zejména pro nové programátory, kteří nemají dostatečnou znalost systému.

Takové řešení navíc naprosto znemožňuje jakékoliv znovupoužití deklarovaných pravidel [30], což způsobuje duplicitu nejen ve směru horizontálním, ale zejména vertikálním. Při zaměření na vrstvu uživatelského rozhraní je navíc nebezpečí duplicity mnohem vyšší - pokud je pro zobrazení stránek definována sada layoutů lišící se například rozložením prvků na stránce, je třeba v zájmu zachování integrity byznys pravidel kontrolovat příslušnou podmínku v každém z layoutů. Pro střední a velké aplikace obsahující stovky až tisíce deklarací byznys pravidel je toto řešení z hlediska údržby a vývoje z výše uvedených důvodů velice nevhodné.

3.1.2 JSR 303

JSR 303: Bean Validation je standard, který definuje metadata model a API (*application programming interface*) pro validaci *JavaBean* objektů [4]. Tento standard je součástí *Java Enterprise Edition 6* a jako zdroj informací využívá anotace, jenž je možné rozšiřovat a vytvářet tak vlastní validátory [4]. Validační anotace umožňují přímo rozšířit datový model, což může být základem pro interpretaci byznys pravidel do ostatních vrstev. To umožňuje například *Hibernate Validator*, který je založen na tomto standardu a validační pravidla definovaná nad objekty v aplikační vrstvě transformuje do integritních omezení databázových schémat, které generuje [17]. Pokud například uvedeme anotaci `@NotNull` nad atributem entity, příslušný sloupec v databázi bude vygenerován s integritním omezením `NOT NULL`.

JSR 303 nabízí sadu již implementovaných anotací, které je možné využít, jež sdílejí společné rozhraní. Jednotlivé anotace jsou považovány za omezení, pokud jsou samy definovány s anotací `@Constraint`, jež definuje třídu, která bude sloužit k samotné validaci hodnot [4]. Každá anotace musí také definovat element `message`, který reprezentuje řetězec sloužící k vytvoření zprávy při porušení byznys pravidla. V zobrazovaných zprávách je možné využít proměnné, například při omezení číselného rozsahu lze vypsat horní a dolní hranici intervalu. K transformaci zprávy do zobrazitelného formátu slouží překladové soubory, tedy zpětná vazba je lokalizována podle preferovaného jazyka uživatele v user kontextu.

Zdrojový kód 3.1: Ukázka validace pomocí JSR 303

```
public class Address {
    @NotNull(message="streetName.notNull")
    @Length(max=50)
    private String streetName;

    @Pattern(regexp="[a-zA-Z\s]*")
    private String city;

    @NotNull
    @Valid
    private Country country;
    /*getters, setters*/
}
```

K samotnému vykonání validace a získání zprávy o porušení pravidel slouží instance třídy implementující rozhraní `javax.validation.Validator`. To definuje metodu `validate(T object)`, která aplikuje příslušné validátory na atributy daného objektu, a vrátí množinu objektů třídy `ConstraintViolation`, jež reprezentují jednotlivá porušená pravidla. Pokud je množina prázdná, proběhla validace bez chyb [4].

Výhodou standardu *JSR 303* je způsob obohacení *JavaBean* objektů, jenž je realizován pouze přidáním metadat ve formě anotací, tedy nedochází ke

strukturálnímu zásahu do deklarace třídy. Konkrétní byznys pravidla jsou však stále přimíchána v kódu definující entitu, tedy není zde striktní rozdělení entit a pravidel. Ta musí být získávána prostřednictvím inspekce entity a jejích atributů, čímž pádem dochází k překryvu business kontextu s ostatními kontexty. Tento standard se navíc zabývá pouze validací, tedy pro reprezentaci jiného typu pravidel je třeba využít další technologie, což má za důsledek vyšší komplexitu aplikace. Před vydáním standardu *JSR 349: Bean Validation 1.1* v roce 2013, který je součástí *Java Enterprise Edition 7* navíc nebylo možné validovat vstupní a výstupní argumenty metod, či používat v *JavaBean* objektech vzor Dependency Injection [3], nicméně nová verze standardu tyto techniky již umožňuje.

3.1.3 RichFaces

Knihovna *RichFaces* je komponentární UI framework v jazyce Java, který rozšiřuje implementaci JSF (*Java Server Faces*[1]). Obě knihovny (*RichFaces* i *JSF*) umožňují validaci v uživatelském rozhraní, nicméně *RichFaces* v tomto ohledu výrazně převyšuje *JSF*, neboť umožňuje efektivním způsobem redukovat duplicitu byznys pravidel.

Vestavěná podpora pro validaci v JSF je založena na standardu *JSR 303* a umožňuje definovat validátory v UI, jejichž aplikování je součástí životního cyklu JSF. Samotná validace dat však probíhá na serveru [28], tedy podporu pro čistě *client-side* validaci JSF nenabízí. Tu dodává knihovna *RichFaces*, která navíc výrazně redukuje duplicitu validátorů v rámci UI, jelikož k vytvoření pravidel na straně klienta opětovně používá pravidla datového modelu, deklarovaná pomocí anotací na základě *JSR 303*. Validace části formuláře z příkladu 3.1 v JSF je demonstrována v ukázce 3.2.

Zdrojový kód 3.2: Validace v UI pomocí JSF

```
<h:outputLabel for="streetName" value="Street name"/>
<h:inputText id="streetName" value="#{model.streetName}"
 required="true">
    <f:validateLength maximum="50"/>
</h:inputText>
<h:message for="streetName"/>

<h:outputLabel for="city" value="City"/>
<h:inputText id="city" value="#{model.city}">
    <f:validateRegex pattern="[a-zA-Z\s]*"/>
</h:inputText>
<h:message for="city"/>
```

Jak je patrné, veškerá validační pravidla z modelu jsou duplikována v UI, což výrazně zvyšuje náročnost údržby takového systému. Přístup *RichFaces* se této duplicitě vyhýbá a opětovně využívá pravidla deklarovaná nad datovým

[1] https://javaserverfaces.java.net

modelem. Validaci stejného formuláře na straně klienta v technologii *RichFaces* ilustruje ukázka 3.3.

Zdrojový kód 3.3: Validace v UI pomocí RichFaces

```
<h:outputLabel for="streetName" value="Street name"/>
<h:inputText id="streetName" value="#{model.streetName}">
    <rich:validator/>
</h:inputText>
<h:message for="streetName"/>

<h:outputLabel for="city" value="City"/>
<h:inputText id="city" value="#{model.city}">
    <rich:validator/>
</h:inputText>
<h:message for="city"/>
```

Knihovna *RichFaces* využívá standard *JSR 303* a transformuje pravidla z datového modelu do UI bez nutnosti jejich opětovné deklarace. Pro změnu pravidel stačí modifikovat pouze anotace v modelu a změna se automaticky promítne i do client-side validačních pravidel. *RichFaces* jsou však bohužel omezeny vazbou na JSF, tedy řešení není dostatečně nezávislé a univerzálně aplikovatelné.

3.2 Aspektově orientované programování

Objektově orientované programování (OOP) v současnosti dominuje systémovým návrhům [5], jelikož objektový model velice dobře reprezentuje reálné problémy a umožňuje snadnou vizualizaci. Ukázalo se však, že OOP není dostatečně univerzálním řešením a neumožňuje čistě zachytit některé aspekty návrhu [26], konkrétně již výše zmiňovanou cross-cutting funkcionalitu. OOP nenabízí nástroje pro separaci a zapouzdření těchto problémů, čímž dochází k rozsáhlé duplikaci kódu a propletení s byznys logikou [5], v důsledku čehož je aplikace náročná na údržbu, náchylná k chybám při změnách a celkově vyvíjena neefektivním způsobem, co se týče cross-cutting funkcionality.

Těmto nedostatkům se věnuje *aspektově orientované paradigma* [2] (AOP), které představuje alternativní přístup k řešení těchto problémů. AOP vychází z objektově orientovaného paradigmatu a definuje novou terminologii, jež tento přístup popisuje:

1. **Aspekt** je nový typ komponentu, který slouží k popisu jednotlivých separovaných cross-cutting problémů. Aspekt je základem AOP a umožňuje zapouzdření těchto problémů a jejich izolaci od spletité byznys logiky systému. Cílem této separace je umožnění opětovného použití aspektů, kterého je docíleno označením míst, kde má být daný aspekt využit (*pointcut*). Veškeré změny funkcionality stačí provádět pouze v

[2]Také aspektově orientované programování

těchto aspektech, což výrazně zjednodušuje údržbu systému. Příkladem aspektu mohou být byznys pravidla, nebo například logování.

2. **Join point** označuje místo, na které lze aplikovat logiku zapouzdřenou v aspektech. Tyto body jsou určeny programovacím jazykem a paradigmatem a slouží k instrumentaci kódu funkcionalitou použitých aspektů [35]. V objektově orientovaném programování patří mezi join pointy například třídy, metody, volání metod, čtení a zápis atributu, zpracování výjimky atd. Rozdíl mezi metodou a voláním metody jakožto join pointem je popsán v [35].

3. **Advice** je samotná funkcionalita, která obohacuje kód v jednotlivých join pointech. Tu lze ještě dále rozlišovat podle cíle připojení nové funkcionalitu vzhledem k danému join pointu - před, za, za při (ne)vyhození výjimky, nebo okolo [35]. Příklad advice pro byznys pravidla je verifikace podmínky z množiny pravidel v business kontextu, pro logování je to část kódu, která zapisuje nastalou událost.

4. **Pointcut** je množina join pointů, kde může být daný aspekt aplikován [5]. Pointcuty můžeme rozdělit na statické a dynamické. Statické pointcuty lze určit pomocí statické analýzy, dynamické pouze při běhu programu v konkrétní moment, tedy mohou se měnit v závislosti na aktuálním execution kontextu. Jeden nebo více pointcutů a jedna advice dohromady definují konkrétní aspekt. Například pointcut pro byznys pravidla v UI obsahuje join pointy *inicializace formuláře* pro svázání byznys pravidel s formulářovými poli, a *změna hodnoty formulářového pole* a *odeslání formuláře na server* pro vyhodnocení pravidel.

5. **Aspect weaver** je jednou z nejdůležitějších součástí AOP. Slouží k vytvoření instrumentovaného kódu aplikováním jednotlivých aspektů na nemodifikovaný kód [35]. Tento proces nejprve identifikuje join pointy v nemodifikovaném kódu podle pointcutů jednotlivých aspektů a následně vygeneruje výsledný kód aplikováním příslušných advice. Automatizace tohoto procesu umožňuje transformaci aspektů do různých částí aplikace, včetně distribuce pravidla napříč vrstvami. Toho je možné docílit vytvořením dalšího weaveru, jenž definuje, jakým způsobem bude aspekt transformován do požadované domény.

AOP umožňuje efektivně dekomponovat systém na jednotlivé aspekty a následně je pomocí aspect weaveru transformovat do definovaných míst. Díky tomu je možné pravidla opětovně používat a pohodlně je udržovat, jelikož jednotlivé cross-cutting problémy i jejich pointcuty jsou striktně odděleny. Použití doménově specifických jazyků pro popis aspektů pomáhá unifikovat jazyk, ve kterém budou specifikovány a výsledná implementace v příslušné doméně je určena použitým aspect weaverem. Například byznys pravidla popsaná tímto způsobem mohou být transformována různými weavery podle cílové domény do SQL jako integritní omezení, do byznys vrstvy jako příslušné validátory, a do UI jako validační funkce v JavaScriptu.

3.3 Aspect-driven design approach

Aspect-driven design approach (ADDA) je inovativní přístup k návrhu informačních systémů, který se zaměřuje na identifikaci průřezových problémů (*cross-cutting concerns*) napříč aplikací, jejich separaci do nezávislých aspektů a redukci duplicity. Cross-cutting problém je část kódu, která je duplikována v různých částech aplikace, přičemž slouží ke stejnému účelu. Mezi cross-cutting problémy v UI patří kromě byznys pravidel například také layouty stránek, widgety pro jednotlivá formulářová pole, lokalizace, či samotná struktura modelu.

ADDA přístup se soustředí na identifikaci těchto částí a jejich oddělení od byznys logiky a sdružení na jednom místě (*single focal point*) [6, 7]. Díky tomu je pak snazší jejich automatická transformace do konkrétních míst bez nutnosti opětovné manuální deklarace. Separace cross-cutting problémů a jejich automatické znovupoužití umožňuje výrazně redukovat horizontální i vertikální duplicitu kódu v aplikaci [6].

K popisu jednotlivých aspektů doporučuje ADDA využívat DSL (*Domain-Specific Language*) [6], který bude sloužit konkrétně pro tento účel. Znamená to sice zavedení nového jazyka do aplikace, čímž se zvýší její komplexita a počáteční režie nutná pro konfiguraci, nicméně popis aspektů pomocí DSL je velice praktickým způsobem, který umožňuje rychlý vývoj. Pro korektní interpretaci aspektů a zejména jejich transformaci do různých domén je totiž vyžadován model, který umožňuje inspekci, což při použití aplikačního programovacího jazyka není možné [6]. Vyjádření pomocí meta-instrukcí postrádá typovou bezpečnost, která může být v DSL kontrolována, tedy také není vhodným prostředkem. Deklarace pomocí DSL nabízí velkou volnost, jelikož konkrétní jazyk je zkonstruován právě pro tyto účely a inspekce takto vytvořených aspektů umožňuje pohodlnou transformaci do potřebných míst.

ADDA využívá aspektově orientované paradigma [6] za účelem minimalizace duplikace a náročnosti údržby. Velice důležitou metodou, zejména pro tvorbu distribuovaných uživatelských rozhraní, je *runtime aspect weaving*, tedy transformace aspektů do instrumentovaného kódu, která probíhá za běhu aplikace. To umožňuje reflektovat aktuální stav aplikace, čímž rozšiřuje spektrum možností, podle kterých se můžeme rozhodnout pro výslednou implementaci - díky tomu je například možné deklarovat byznys pravidlo, které bude kontrolovat, zda uživatel používá mobilní zařízení pro zobrazení obsahu. Aspect weaver v ADDA tohoto využívá a při samotném weaving procesu přistupuje k aktuálnímu execution kontextu [7], ze kterého jsou příslušné informace získány.

3.4 Drools

Framework JBoss Drools [3] je nástroj pro tvorbu produkčních systémů (*production system*) pro platformy Java a .NET. Drools slouží zejména k

[3] http://www.drools.org/

tvorbě a údržbě pravidel, která jsou základem těchto systémů, spolu s jejich vyhodnocováním nad konkrétním modelem, k čemuž využívá rozšířenou verzi RETE algoritmu [20]. Součástí projektu je knihovna Drools Expert, jež je samotným jádrem enginu [2] a slouží k tvorbě znalostní báze (*knowledge base*) s deklarovanými pravidly. Tato báze efektivně odděluje byznys pravidla od zbytku aplikace, což značně usnadňuje jejich správu. K popisu pravidel využívá knihovna doménově specifický jazyk Drools DSL, který byl vyvinut speciálně pro tento účel.

Zdrojový kód 3.4: Deklarace pravidla v Drools DSL

```
rule "Is user mature"
    when
        $user : User(age >= 18) // LHS, condition
    then
        System.out.println("User " + $user.name +
        " is mature")   // RHS, consequence
end
```

Pravidla v Drools DSL jsou složeny ze dvou částí - podmínka (*LHS, Left Hand Side*) a následek (*RHS, Right Hand Side*) [2] viz. ukázka 3.4. Syntaxe tohoto jazyka umožňuje používat mj. následující konstrukce:

- **Podmínky** – Ty jsou samotným základem byznys pravidel a slouží k vytvoření LHS. Příkladem deklarace podmínky je `User(age >= 18)` pro výběr dospělých uživatelů, k vytváření složitějších konstrukcí lze využít spojování jednotlivých podmínek logickými operátory.

- **Proměnné** – Pro opětovné použití vybraných hodnot je v rámci pravidel možné deklarovat proměnné, které vždy začínají znakem `$`. Zápis `$account : Account($balance : balance)` vytvoří dvě proměnné, přičemž `$account` je reference na objekt typu `Account`, který je kontrolován pravidlem, a `$balance` je výsledek volání `Account.getBalance()` na dané instanci této třídy.

- **Typy** – Drools nabízí podporu pro veškeré nativní typy v Javě [2]. Kromě číselných typů a znakových řetězců tak umožňuje také pohodlnou práci s daty, enumy, či podporuje využívání regulárních výrazů, např. `User(name matches "[A-Za-z]+")`.

- **Globální proměnné** – Kromě standardních proměnných, které jsou viditelné pouze v rámci jednoho pravidla, je také možné deklarovat globální proměnné, které jsou sdíleny pro aktuální *session* [2].

- **Funkce** – Do jednotlivých souborů s pravidly lze importovat pomocné funkce, které mohou pomoci redukovat duplicitní kód v rámci těchto souborů.

Kromě výše zmíněných konstrukcí umožňuje Drools DSL využívat řadu dalších pokročilých funkcí. Mezi ně patří například volba dialektu mezi výchozím

(Java) a MVEL [10], což je *expression language* založený na syntaxi Javy [2]. Ten umožňuje např. zkrácený přístup k vnořeným atributům objektů pomocí reflexe (`$address.city.state` místo `$address.getCity().getState()`), indexaci v kolekcích pomocí hranatých závorek (seznamy a mapy), nebo inline deklaraci těchto kolekcí a polí [2]. Kromě toho je MVEL typově bezpečný a podporuje vlastní rozšíření [35].

Dále je v Drools DSL možné definovat atributy jednotlivých pravidel. Pro případ konfliktů lze rozlišovat různé priority pravidel, které určí pořadí jejich vyhodnocení. Priorita pravidla nemusí být pouze statická, ale lze využít proměnné, k jejichž vyhodnocení dojde při běhu aplikace. Další z atributů je `no-loop`, který zakazuje opětovné vyhodnocení téhož pravidla z důvodu změny, jež nastala v rámci RHS a která by mohla vést například k nekonečné smyčce. Mezi atributy pravidel patří také výše zmiňovaný dialekt, který se může vztahovat k celému souboru, nebo pouze k jednotlivým pravidlům.

Drools DSL je aktuálně využívaným jazykem pro deklaraci pravidel v implementaci ADDA [6]. Výhodou tohoto jazyka je vlastní kompilátor a parser, který umožňuje inspekci znalostní báze a transformaci pravidel do POJO (*Plain Old Java Object*), díky čemuž lze distribuovat pravidla do jiných vrstev. Drools navíc nabízí velice silné nástroje, z nichž některý byly popsány výše, které umožňují tvorbu i velice komplexních pravidel.

Pro účely ADDA je však tato komplexita spíše nevýhodou, jelikož výrazně ztěžuje distribuci pravidel do jiných vrstev. Primárním cílem knihovny Drools je správa byznys pravidel v aplikační vrstvě, nicméně použití pokročilých nástrojů (import funkcí, operace nad kolekcemi, atributy pravidel apod.) vyžaduje jejich reflexi v aspect weaveru do cílové domény, což je poměrně složitý proces. Dále pak Drools nenabízí podporu pro efektivní sdružovaní pravidel do skupin. To je sice možné pomocí rozdělení do balíků (*package*), nicméně průniky těchto množin, tedy pravidla, která patří do více skupin, způsobují duplicitu, jelikož knihovna neumožňuje import pravidel z jiného souboru. Sdružování lze také realizovat přiřazením atributu *ruleflow-group* k jednotlivým pravidlům, což však vede ke stejnému výsledku, jelikož pravidla mohou mít nejvýše jednu deklarovanou skupinu *ruleflow-group*.

V poslední řadě je také třeba podotknout, že ADDA využívá pouze část nástrojů Drools DSL, jelikož při deklaraci pravidel uvádí pouze LHS podmínku. Následek pravidla je totiž určen samotnou byznys operací [35] a pro různé vrstvy se může lišit. Z předchozího textu je patrné, že Drools DSL je sice dostačujícím nástrojem pro popis aspektu byznys pravidel v ADDA, nikoliv však ideálním. Proto by bylo vhodné navrhnout vlastní doménově specifický jazyk se zjednodušenou syntaxí, jenž se bude zaměřovat na sdružování pravidel do business kontextů s důrazem na jejich znovupoužitelnost a zachycení podmínek těchto pravidel.

3.5 Implementace ADDA konceptu v UI

Duplicita a nepřehlednost kódu v prezentační vrstvě je zapříčiněna projekcí multidimenzionálního prostoru do jednorozměrného kódu [8]. Jednotlivé cross-

cutting problémy si lze totiž představit jako osy n-rozměrného prostoru (struktura modelu, byznys pravidla, lokalizace, layout, atd.), přičemž aktuální stav aplikace může být vnímán jako jeden konkrétní bod v tomto prostoru. Souřadnice tohoto bodu odpovídají hodnotám příslušných proměnných z execution kontextu. Při popisu tohoto prostoru pomocí jednorozměrného jazyka však dochází v důsledku linearizace ke ztrátě ortogonality jednotlivých os [8], tedy některé informace jsou v kódu duplikovány.

3.5.1 AspectFaces

V kapitole 2.4 je uvedeno, že téměř polovina kódu a času je věnována vývoji uživatelského rozhraní. Velká část UI kódu však obsahuje cross-cutting funkcionalitu, která způsobuje značnou duplicitu, pokud nejsou jednotlivé části odděleny v souladu s AOP. Těmto problémům se věnuje knihovna *AspectFaces* [4], která efektivně dekomponuje aplikaci do jednotlivých aspektů a umožňuje automatické generování uživatelského rozhraní. K implementaci UI je použita knihovna JSF, která je v současné době jediná technologie, která umožňuje dynamickou integraci s *AspectFaces*.

Generování UI probíhá v několika fázích. První z nich je inspekce, která přijímá objekt reprezentující data, např. entitu, či DTO, a vytvoří z něj meta-model, jenž bude sloužit jako zdroj informací pro další fáze. Meta-model obsahuje veškerá data, která budou interpretována do uživatelského rozhraní, bez nutnosti jejich opětovné manuální deklarace. Z datového modelu jsou získány informace o atributech a případná meta-data, jako např. validační či vlastní anotace. Knihovna *AspectFaces* je založena na konceptu *Model Driven Development* (MDD) [31], jenž usiluje o zachycení informací v modelu, díky čemuž je následně možné tyto informace sdílet na více místech v aplikaci. Jelikož generování probíhá za běhu aplikace, je uvážen i aktuální execution kontext, tudíž hodnoty jeho proměnných mohou ovlivnit výsledný vzhled stránky. Ukázka datového modelu obohaceného o meta-data je znázorněna na příkladu 3.5.

Zdrojový kód 3.5: Datový model v AspectFaces

```
@Entity
public class User {
    @UiOrder(1) // pořadí pole v UI
    @NotBlank   // validace povinného pole
    private String email;

    @UiOrder(2)
    @UiPassword // použití komponenty pro zadání hesla
    @NotBlank
    private String password;
}
```

[4]http://www.aspectfaces.com

Další fází je samotná transformace, která aplikuje prezentační pravidla pro jednotlivé atributy, jež určují typ komponentu, který pro ně bude ve výsledném kódu použit. Tato pravidla lze libovolně upravovat, stejně jako vytvářet vlastní widgety pro jednotlivá pole. Výsledkem aplikování prezentačních pravidel je výběr šablony, která bude použita k vykreslení stránky. Rozložení jednotlivých komponent na stránce určuje layout, jenž je jedním z atributů komponenty generující UI. Vykreslení formuláře je docíleno jediným komponentem `af:ui`, jak je vidět v ukázce 3.6.

Zdrojový kód 3.6: Vytvoření formuláře v AspectFaces

```
<af:ui instance="#{bean.user}" layout="user-form"/>
```

Zdrojový kód 3.7: Generický layout v AspectFaces

```
<table>
    <tr>
        <td colspan="2">$af:email$</td>
    </tr>
    <af:iteration-part maxOccurs="100">
        <tr>
            <td>$af:next$</td>
            <td>$af:next$</td>
        </tr>
    </af:iteration-part>
</table>
```

Posledním krokem je vygenerování stránky z transformovaného kódu a datové instance. Díky runtime weavingu jednotlivých aspektů je možné přizpůsobit výsledný vzhled stránky aktuálnímu uživateli podle jeho potřeb. Navíc dochází k výrazné redukci duplikovaného kódu, jelikož komponenty a šablony používané ve fázi transformace, mohou být deklarovány genericky (viz ukázka 3.7), tudíž je lze opětovně použít. Pořadí jednotlivých polí v těchto šablonách je určeno anotací `@UiOrder` v datovém modelu. Separace aspektů také usnadňuje údržbu aplikace a redukuje čas strávený implementací UI.

AspectFaces nabízí podporu pro aspektově orientovaný návrh informačních systému se zaměřením na eliminaci duplicity v uživatelském rozhraní. Umožňuje efektivním způsobem zachytit jednotlivé aspekty v systému, s jejichž využitím při runtime aspect wavingu a reflexi aktuálního execution kontextu docílí automatického generování formulářů. Bohužel však neposkytuje dostatečnou podporu byznys pravidel, jelikož umožňuje pouze jednoduchou validaci, která navíc probíhá pouze na serveru.

3.5.2 Nutforms

Knihovna *Nutforms* se zaměřuje na tvorbu distribuovaných uživatelských rozhraní, přičemž je kladen důraz na znovupoužitelnost jednotlivých aspektů a redukci duplicitního kódu. Využívá ADDA přístupu pro dekompozici systému na izolované aspekty a jejich sdružení v jednom místě aplikace (*single focal*

point) za účelem snadné údržby a minimalizaci úprav při změnách. Cílem je oddělit cross-cutting funkcionalitu v UI do těchto aspektů, spravovat je v rámci znalostní báze a transformovat je do jednotlivých částí aplikace bez nutnosti duplicitní manuální deklarace.

V rámci této knihovny je identifikováno pět aspektů - struktura modelu, layout stránky, widgety pro jednotlivé atributy modelu, lokalizace, a byznys pravidla. Ke generování výsledného UI slouží *rich model*, který je automaticky sestaven z jednotlivých aspektů a aktuálního execution kontextu. Tvorba modelu je inspirována technikou READ (*Rich Entity Aspect/Audit Design*) [8], jež se zaměřuje na automatické vytvoření meta-modelu, založené na inspekci kódu spolu s využitím návrhu aplikace v rámci AOP. Díky této inspekci je možné vytvořit plně automaticky novou instanci modelu, která dokáže sdružit informace z jednotlivých aspektů, které jsou spravovány odděleně. Navíc tento proces probíhá za běhu aplikace, tedy je reflektován aktuální execution kontext, což umožňuje zachovat konzistentní stav mezi datovým modelem a *rich modelem* a na změny bezprostředně reagovat v UI [8].

Důležitou částí knihovny je také životní cyklus UI formulářů. Ten je založen na principu událostí (*event*), na které lze v aplikaci reagovat. V rámci knihovny jsou rozlišeny čtyři kroky životního cyklu formuláře:

- **Inicializace** – V této fázi dochází identifikaci aktuálního execution kontextu a načtení příslušných dat ze serveru.
- **Vykreslení formuláře** – Podle hodnot proměnných z execution kontextu je vykreslen samotný formulář, přičemž jsou reflektovány příslušné aspekty - struktura modelu, layout, widgety a lokalizace stránky.
- **Změna hodnoty** – Tento krok slouží identifikaci změn hodnot v jednotlivých formulářových polích. *Nutforms* využívá *two-way data binding* [5], díky čemuž je zajištěn konzistentní stav modelu a zobrazených dat. Tato technika slouží k propagaci změn - pokud dojde ke změně modelu, jsou aktuální data ihned reflektována na stránce a při změně hodnot ve formuláři je taktéž aktualizován model.
- **Odeslání formuláře** – Poslední fáze životního cyklu slouží k přenosu hodnot z formuláře od klienta na server.

Struktura modelu je tvořena třídami reprezentující jednotlivé aspekty a k jejich kompozici je využit návrhový vzor *facade* [34]. Mimo ten využívá také model vzor *observer* [34], který umožňuje reagovat na jednotlivé kroky životního cyklu formuláře. Bližšímu popisu jednotlivých částí se budu věnovat v kapitole 4.

Jednou z výhod *Nutforms* je nezávislost na UI frameworcích. Základní logika aplikace je napsána v JavaScriptu a výsledné UI komponenty jsou standardními HTML tagy s vlastními atributy. Pro vygenerování obsahu stačí na příslušné stránce pouze vložit skript, který definuje element, do nějž bude vložen vygenerovaný kód a zbytek potřebných informací bude

[5] https://docs.angularjs.org/guide/databinding

získán z execution kontextu. K načtení datového modelu ze serveru je v současné chvíli využíváno REST API [19] a načítání jednotlivých aspektů je realizováno pomocí modulu *node-fetch* [6], nicméně samotná inspekce není součástí knihovny - ta pouze udává formát těchto dat. Způsob získávání dat je navíc pouze záležitostí prototypu aplikace, tedy v budoucnu může dojít k jeho změnám.

3.6 Architektura Nutforms

Knihovna *Nutforms* je rozdělena na dvě části - *server-side* a *client-side*. Na straně serveru jsou definovány *servlety* [9], které slouží jako *singal focal point* pro sdružení jednotlivých aspektů do znalostní báze. Tyto servlety jsou zodpovědné za zpracování požadavků a načtení požadovaných aspektů. Jejich získávání je možné prostřednictvím REST API, které zajišťuje transport serializovaných aspektů ze serveru na stranu klienta.

Klientská strana obsahuje aspect weavery pro jednotlivé aspekty, jež slouží k vygenerování advice. Kombinací výstupů jednotlivých weaverů dochází k tvorbě výsledného UI kódu, který bude na stránce použit. K načtení aspektů ze serveru je využito výše zmiňované REST API a aktuální execution kontext. Ten obsahuje potřebné informace pro vytvoření požadavku na server, jako např. aktuální jazyk v user kontextu pro načtení lokalizačního aspektu obsahující popisky v příslušném jazyce, nebo `id` upravované entity, které umožní načtení dat uložených v databázi. Z načtených aspektů je vytvořen *rich model*, který sdružuje aktuální stav formuláře a jednotlivé aspekty, a pomocí *two-way data binding* udržuje konzistentní stav s UI.

Aplikace využívající knihovnu *Nutforms* je zodpovědná za vytvoření datového modelu, widgetů pro atributy objektů a mapovací soubor, specifikující za jakých podmínek použít který widget, a sadu layoutů určující rozložení jednotlivých komponent na stránce. Pro výslednou integraci je třeba do stránky pouze vložit skript, definující element, do nějž bude vložen vygenerovaný kód, odkud převezme nad procesem kontrolu knihovna.

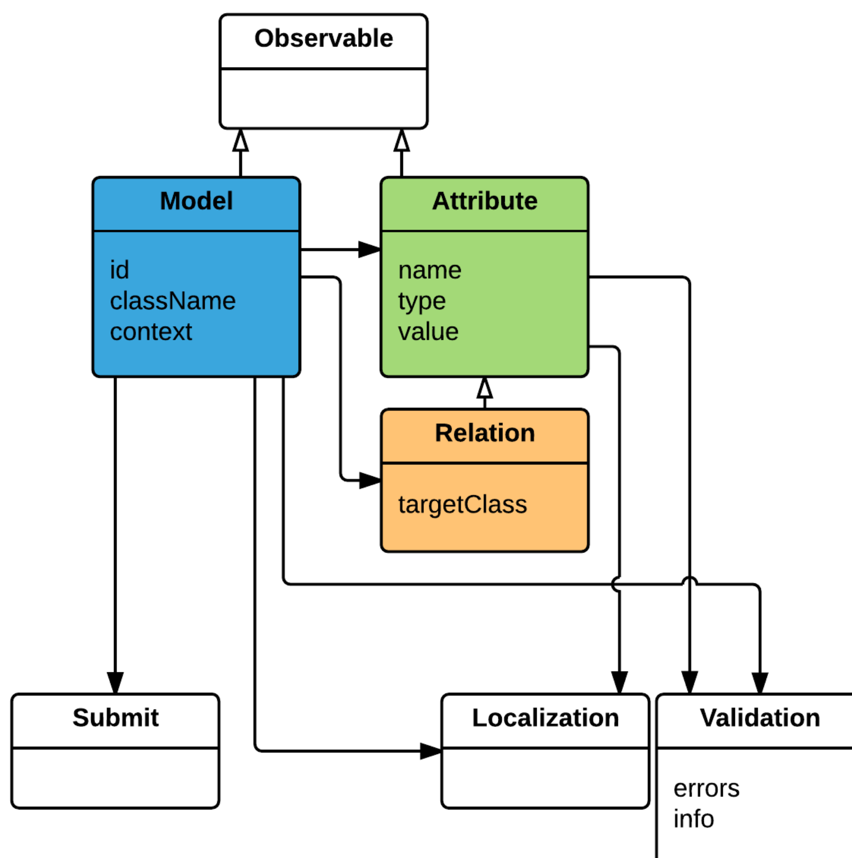
3.6.1 Rich Model

Hlavní datovou strukturou na straně klienta pro obsluhu stránky je *rich model*, který je automaticky sestaven z jednotlivých aspektů a udržuje informace o aktuálních hodnotách atributů. K jeho vytvoření dochází před generováním stránky z dat načtených ze serveru, při odeslání formuláře je aktuální model serializován a odeslán na server. *Rich model* je tvořen **strukturou** datového modelu, z níž zachycuje jméno třídy a `id` aktuálně upravované entity, spolu se seznamem jednotlivých atributů a vazeb na další entity (*relation*). S každým atributem a vazbou je ze serveru načten název, datový typ a aktuální hodnota.

Kromě struktury jsou v *rich modelu* zachyceny také jednotlivé aspekty. Prvním z nich je aspekt **lokalizace**, jenž obsahuje veškeré atributy prvků

[6] https://www.npmjs.com/package/node-fetch

stránky v jazyce podle preferencí uživatele z aktuálního user kontextu. Ty jsou načteny ze serveru při vytváření formuláře a určují např. texty tlačítek, či překlady názvů atributů modelu. Další prvek modelu je **layout** stránky, určující rozložení prvků, jenž je načten ze souboru na serveru podle názvu aktuálního business kontextu. Stejným způsobem jsou načteny i **widgety** pro jednotlivé atributy entity, které určují vzhled a typ elementů ve formuláři. K navázání widgetů na atributy slouží mapovací soubor, který je taktéž načten ze serveru. Poslední součástí modelu je třída **submit**, jež je zodpovědná za odeslání formuláře na server, serializaci modelu a podle odpovědi serveru vyvolání příslušných událostí, na které je možné reagovat. Odeslání je posledním krokem životního cyklu formuláře, který je popsán v kapitole 3.5.2. Vizualizace tříd modelu podstatných pro integraci byznys pravidel je zobrazena na obr. 3.1, kde je uveden i validační aspekt.



Obrázek 3.1: RichModel knihovny Nutforms

Velice důležité je využití návrhového vzoru *observer* [34], který umožňuje reagovat na nastalé událost v modelu. Jednotlivé třídy popisující atributy, vazby i samotný model dědí od společného předka `Observable`, který definuje metodu `listen()`, jež slouží k přidání posluchače dané události k příslušnému

atributu, nebo modelu. Při libovolné akci dojde k notifikaci všech posluchačů daného objektu - ti mohou mít podobu funkce, jež bude zavolána s příslušnými atributy, nebo objektu, na němž dojde k zavolání metody `update()` s týmiž atributy. Tento způsob implementace umožňuje pohodlnou práci s životním cyklem formulářů a snadné přidávání nové funkcionality k jednotlivým událostem.

3.7 Shrnutí

Aspektově orientovaný přístup k návrhu informačních systémů umožňuje řešit problémy OOP týkající se cross-cutting funkcionality, mezi něž patří jejich horizontální a vertikální duplikace či nedostatečná znovupoužitelnost. Díky tomu je výrazně redukován čas strávený vývojem i údržbou aplikace a snižuje se riziko chyb, jelikož změny v jednotlivých aspektech jsou automaticky interpretovány do všech míst aplikace. Stávající přístupy k řešení problémů spojených s duplikací kódu, jako např. *Hibernate Validator*, nebo *RichFaces*, sice umožňují znovupoužití byznys pravidel ve více doménách, nicméně žádné z nich nedokáže transformovat byznys pravidla do všech částí vrstevnatých aplikací ze společné znalostní báze. Přístup ADDA doporučuje k deskripci těchto aspektů použít doménově specifické jazyky, které umožňují inspekci, díky čemuž lze aspekty pohodlně transformovat do požadovaných míst.

Jako aktuálně používaný DSL pro deklaraci byznys pravidel v ADDA je využívána knihovna Drools, která sice neumožňuje efektivní sdružování pravidel do business kontextů a opětovné použití konkrétního pravidla ve více kontextech, nicméně poskytuje možnost inspekce a tedy i následnou transformaci do více domén. Tu však značně ztěžuje robustnost a komplexita tohoto řešení, jelikož při aspect weavingu je třeba zohlednit veškeré atributy pravidel.

Knihovna *Nutforms* implementuje ADDA koncept v uživatelském rozhraní. K reprezentaci jednotlivých aspektů na straně klienta využívá *rich model*, který slouží jako centrální bod pro jejich sdružení a reprezentaci aktuálního execution kontextu. Tento model umožňuje pohodlný způsob reflexe cross-cutting funkcionality bez nutnosti opětovné deklarace, jelikož je vytvářen automaticky a dynamicky za běhu aplikace. Jednotlivé aspekty jsou v *Nutforms* spravovány izolovaně, díky čemuž je výrazně usnadněna jejich správa.

Kapitola 4

Návrh

Kapitola 2.3 se věnuje problémům spojeným s reprezentací byznys pravidel ve webových aplikacích, které pramení zejména v jejich duplikaci. Ta je zapříčiněna cross-cutting povahou těchto pravidel, jelikož v zájmu bezpečnosti a zachování konzistentního stavu dat je nutné omezit byznys procesy operující nad těmito daty sadou pravidel určujících vstupní a výstupní podmínky v příslušných vrstvách. Jednotlivá byznys pravidla jsou navíc kromě duplicitní deklarace často propletena s výkonným kódem, což zvyšuje náročnost provádění úprav v těchto pravidlech, vytváření nových, nebo odstraňování stávajících. Ideálním stavem je striktní izolace veškerých byznys pravidel od výkonového kódu a jejich sdružení v centrální znalostní bázi, která by umožnila separátní správu pravidel. Ta by byla sdružena podle jednotlivých business kontextů a interpretována na příslušná místa, přičemž změny v této bázi by byly automaticky reflektovány na všech místech, kde je dané pravidlo použito.

Stávající možnosti řešení však nenabízejí globálně odstranit duplicitu, ale většinou se omezují na jednotlivé vrstvy. Podle konceptu ADDA jsou vhodným nástrojem pro deklaraci transformovatelných pravidel doménově specifické jazyky umožňující inspekci jejich struktury. Díky tomu je možné sdružit veškerá byznys pravidla na centrálním místě jakožto oddělené aspekty, popsat je vlastním jazykem a pro implementaci do požadovaného jazyka využít transformaci aspect weaverem.

4.1 Validace pomocí AOP

Aspektově orientované programování nabízí alternativní přístup k návrhu aplikací, vedoucí k vyšší modularitě, znovupoužitelnosti komponent a redukci duplicity. Toho je dosaženo dekompozicí systému na jednotlivé aspekty a jejich následné spojení pomocí aspect weaverů. Byznys pravidla jsou z hlediska AOP cross-cutting funkcionalitou, tedy je vhodné je separovat do aspektů. Jednotlivé pojmy AOP pak mohou být formalizovány z hlediska byznys pravidel:

1. **Aspekt** může být chápán jako validace podmínek, jež jsou určeny byznys pravidly. Díky jejich oddělení od výkonného kódu je možné pravidla

spravovat odděleně a sdružovat je do business kontextů. Tento přístup přidává další úroveň oddělení logiky byznys operací, tedy jejich omezení již nejsou součástí kódu obsluhující samotnou operaci, ale místo toho určuje operace business kontext, jenž definuje pravidla, která mají být použita. K popisu těchto aspektů je vhodné využít v souladu s ADDA principem doménově specifické jazyky, například Drools DSL. *Post-condition* jednotlivých pravidel však nejsou součástí této deklarace, jelikož mají pro stejná pravidla rozdílné hodnoty v různých vrstvách, jak je popsáno v kapitole 3.4.

2. **Join point** je místo v aplikaci, do kterého je připojena funkcionalita určená aspektem. V případě validace v uživatelském rozhraní jsou join pointy byznys pravidel jednotlivé fáze životního cyklu formuláře, mezi něž může patřit jeho vykreslení pro navázání na jednotlivá formulářová pole, nebo změna hodnoty a odeslání formuláře pro samotné vyhodnocování byznys pravidel.

3. **Advice** popisuje funkcionalitu, která bude do příslušných join pointů přidána. V případě validace odpovídá advice ověření podmínky pravidla, jež je extrahováno z business kontextu dané operace.

4. **Pointcut** reprezentuje množinu join pointů, na které bude aplikována konkrétní podmínka, tedy advice. V uživatelském rozhraní lze rozlišovat pointcuty podle typu pravidla. Například běžná validační pravidla, jako kontrola délky řetězce, nebo povinnost atributu, budou vyhodnocována při změně hodnoty formulářového pole a při odeslání formuláře. Pravidla týkající se dvou různých polí obsahující operátor `OR` pak pouze při odeslání formuláře. Naopak bezpečnostní pravidla omezující úpravy vybraných polí musí být aplikována již při vykreslování formuláře pro přidání příslušných atributů k jednotlivým polím.

5. **Aspect weaver** slouží k transformaci pravidla definovaného v DSL do vyhodnotitelné advice. Weaving v UI probíhá za běhu aplikace, což umožňuje reflexi aktuálního execution kontextu. Samotný proces probíhá v několika krocích. Nejprve je načten aktuální business kontext, z nějž jsou extrahovány *preconditions* jednotlivých pravidel. Ty jsou následně transformovány na advice, které mohou mít například pro validační pravidla podobu funkcí. Advice jsou nakonec přiřazeny k join pointům, které udává pointcut, jenž je určen typem pravidla. Obohacení formuláře o byznys pravidla probíhá tedy zcela automaticky, je třeba pouze specifikovat byznys kontext, který určuje množinu pravidel, která mají být aplikována.

Využití aspektově orientovaného přístupu k zachycení cross-cutting byznys pravidel v aplikaci umožňuje výrazně zjednodušit jejich údržbu a pomocí automatické transformace aspektů odstranit duplicitní kód týkající se těchto pravidel. Počáteční režie je však výrazně vyšší, než při standardním objektově orientovaném vývoji.

4.2 Integrace validace do Nutforms

Jednotlivé aspekty v knihovně *Nutforms* jsou v souladu s AOP odděleny od výkonného kódu a jejich deklarace je soustředěna na centrálním místě, což umožňuje snadnou správu. Při integraci byznys pravidel je vhodné, aby byly zachovány stejné vlastnosti - deklarace v *single focal point* na serveru, transformace do UI prostřednictvím aspect weaverů bez manuální duplikace kódu a možnost případného opětovného použití pravidla i v ostatních vrstvách. Obohacení knihovny o podporu byznys pravidel v uživatelském rozhraní bude vyžadovat vytvoření znalostní báze, aparátu pro její inspekci a transformaci aspektů na stranu klienta. Client-side pak vyžaduje implementaci aspect weaveru pro transformaci aspektů do spustitelného kódu, rozšíření *rich modelu* o aktuální stav validace a úpravu životního cyklu formuláře s navěšením advice na příslušné události.

4.2.1 Deklarace pravidel

V kapitole 3.3 jsou uvedeny různé přístupy k reprezentaci aspektů ve znalostní bázi a jejich nedostatky. Koncept ADDA doporučuje použít doménově specifické jazyky, jelikož jejich inspekce umožňuje pohodlnou transformaci do různých vrstev systému. Pro reprezentaci byznys pravidel je využívána knihovna JBoss Drools, která specifikuje DSL pro jejich deklaraci a poskytuje možnost inspekce těchto aspektů. Výhody a nevýhody této knihovny jsou blíže popsány v kapitole 3.4, nicméně v současné chvíli je dostatečně vhodným nástrojem pro tvorbu byznys pravidel.

Při deklaraci pravidel je nutné uvést název business kontextu, do kterého daná pravidla patří. Drools bohužel nenabízí dostatečně flexibilní nástroje pro sdružování, protože žádný z přístupů nedokáže efektivně řešit opětovné použití jednoho pravidla ve více kontextech (viz kapitola 3.4), nicméně rozdělení pomocí balíků (*package*) je dostačující pro načítání pravidel podle kontextu. Jméno balíku Drools pravidla tedy odpovídá názvu business kontextu prováděné operace, díky čemuž lze snadno získat seznam pravidel podle názvu kontextu. Rozdělení do balíků však nesouvisí s jednotlivými `.drl` soubory, tedy lze vytvořit dva různé soubory se stejným balíkem, čehož může být využito pro další úroveň separace podle typu, například jeden soubor pro validační pravidla a druhý pro bezpečnostní omezení. Příklad deklarace pravidla včetně business kontextu je demonstrována na v ukázce 4.1;

Pojmenování business kontextů je vytvářeno podle konvence `FQN.akce`, kde `FQN` (Fully Qualified Name) je úplné jméno objektu, na který jsou daná pravidla navázána, včetně cesty v adresářové struktuře projektu. Příklad takového jména je `cz.cvut.fel.nutforms.example.model.Employee`. Proměnná `akce` odpovídá názvu prováděné operace, např. `new` pro vytvoření nové entity, nebo `edit` pro úpravu stávající. Název business kontextu pro vytvoření nového zaměstnance podle této konvence pak bude mít podobu `cz.cvut.fel.nutforms.example.model.Employee.new`. Důvodem pro využití `FQN` je eliminace konfliktů, pokud by v projektu byly dvě entity se

shodným jménem v různých balících. Navíc je `FQN` jedním z atributů, který je na straně klienta transformován do *rich modelu*, díky čemuž lze snadno sestavovat názvy business kontextů.

Zdrojový kód 4.1: Deklarace pravidla v knihovně Nutforms

```
package cz.cvut.fel.nutforms.example.model.Bug.new;
import cz.cvut.fel.nutforms.example.model.Bug;
dialect  "mvel"

rule "[Bug] Is log filled in"
    when
        Bug(log != null && log.length > 0)
    then
end
```

4.2.2 Inspekce pravidel

Knihovna Drools umožňuje inspekci pravidel deklarovaných pomocí Drools DSL a jejich transformaci do POJO (*Plain Old Java Object*). Vzhledem ke komplexitě jazyka je však vhodné tento objekt konvertovat do vlastního modelu, jenž umožní snadnější interpretaci pravidla v UI i dalších vrstvách. V tomto modelu je vhodné například unifikovat jednotlivé typy pravidel (*pattern*, *eval*, *group* [2]) do vyhodnotitelné podmínky v podobě řetězce a zjednodušit rozhraní, jelikož některé nástroje jazyka nebudou využívány. Hlavní výhodou tvorby modelu je však možnost změny technologie pro popis pravidel. Pokud by místo Drools DSL byla pravidla popsána jiným způsobem, stačí pouze vytvořit parser, jenž převede informace do modelu se stejným rozhraním.

Informace, které by model měl zachytit, jsou zejména název balíku pro výběr business kontextu, název pravidla pro jeho identifikaci, jenž může být využita např. při načítání lokalizované zpětné vazby, a samotná podmínka. Dále je také možné načíst proměnné pravidla, globální proměnné a importované funkce, tyto atributy jsou již však specifické pro Drools.

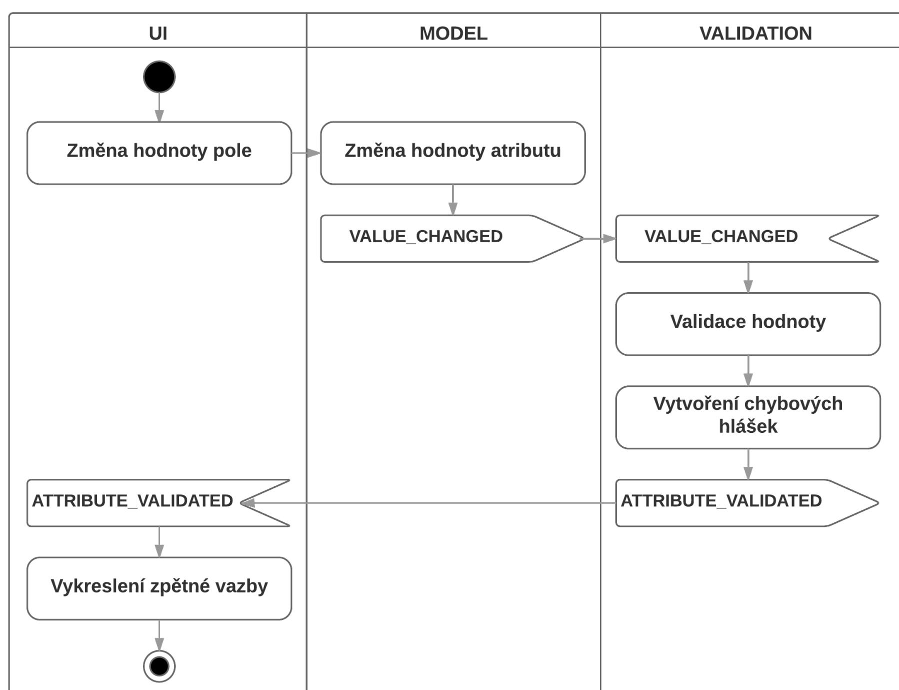
K transportu pravidel ze serveru ke klientovi je využit, stejně jako pro ostatní aspekty, servlet. Ten přijme požadavek obsahující název business kontextu, a odešle seznam odpovídajících pravidel. Jelikož jsou pravidla načtena ve formátu vlastního modelu, tedy POJO, mohou být serializována a na straně klienta mohou být načtena jako JSON (*JavaScript Object Notation* [13]).

4.2.3 Životní cyklus formuláře

Po načtení aspektů byznys pravidel na straně klienta je třeba přidat požadovanou funkcionalitu do formuláře, k čemuž slouží weaving. Vstupními argumenty aspect weaveru pro byznys pravidla jsou vytvořený model, objekt s načtenými pravidly ze serveru, a aktuální execution kontext. Vhodnou reprezentací byznys pravidel v UI, jež může být automaticky generována, je

sada funkcí, která bude reagovat na změny hodnot příslušných formulářových polí a podle výsledku zobrazí zpětnou vazbu, případně zamezí odeslání formuláře, pokud data nebudou validní. To umožňuje rozhraní `Observable`, jež implementují třídy modelu a jeho atributy - jednotlivé validační funkce reprezentují posluchače daných událostí. Tento způsob implementace navíc umožňuje integraci pravidel vázajících se na celý model, například podmínky obsahující operátor `OR`.

Na obr. 3.1 je navržena podoba validačního aspektu v UI modelu, jenž bude navázán na model a všechny atributy a vazby upravované entity. Tato třída je zodpovědná za správu zpětné vazby jednotlivých pravidel. Pokud dojde k porušení podmínky pravidla, bude příslušná hláška uložena do pole `errors` a následně vykreslena k formulářovému poli s nevalidní hodnotou. Pole `info` slouží ke správě zpětné vazby informativního charakteru, například počet zbývajících znaků. Pro zobrazování hlášek v korektním jazyce lze využít lokalizační aspekt a zprávy načítat ze serveru podle aktuálního user kontextu.



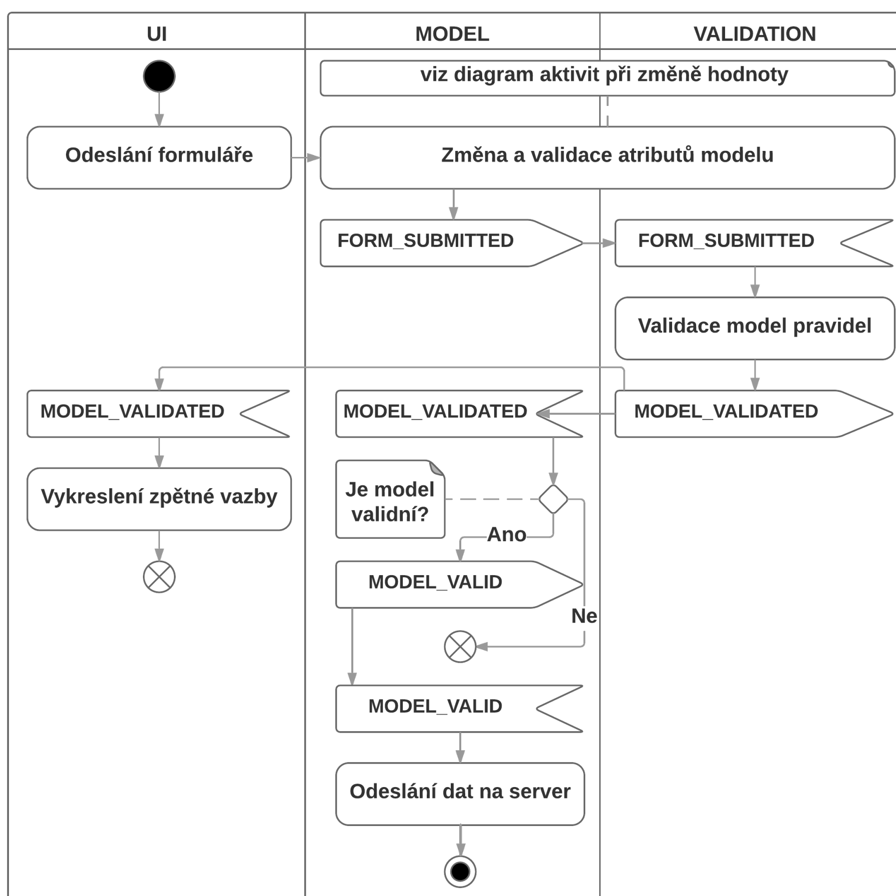
Obrázek 4.1: Diagram aktivit při změně hodnoty

Obrázek 4.1 popisuje návaznost aktivit a vyvolané události při změně hodnoty formulářového pole. V prvním sloupci jsou uvedeny akce prováděné v uživatelském rozhraní, jež jsou přímo viditelné pro uživatele. Druhý sloupec představuje změny v hodnotách atributů *rich modelu*. Poslední sloupec popisuje akce prováděné ve třídě `Validation`, která je součástí modelu a je zodpovědná za správu zpětné vazby atributů a samotného modelu.

Změna hodnoty v UI je nejprve reflektována v modelu pro zachování

konzistentního stavu dat. Následně je vyvolána událost změny hodnoty, což spustí validaci pravidel vázajících se na dané pole. Pokud nejsou podmínky některých pravidel splněny, jsou ze serveru načteny příslušné chybové hlášky a uloženy do modelu, odkud jsou následně zobrazeny uživateli.

Aktivity při odeslání formuláře jsou popsány na obr. 4.2. Prvním krokem je nastavení aktuálních hodnot z UI do modelu, což vyvolá validaci všech atributů viz obr. 4.1. Poté dojde k události samotného odeslání formuláře, na což reaguje validační aspekt, jenž aplikuje všechna pravidla vázající se na model. Následně model zkontroluje, zda jsou jeho hodnoty validní, a pokud ano, dojde k odeslání dat na server. V tomto místě by bylo vhodné například kontrolovat odpověď ze serveru a podle ní adekvátně reagovat, nicméně pro validaci již tento krok není podstatný, proto není v diagramu uveden.



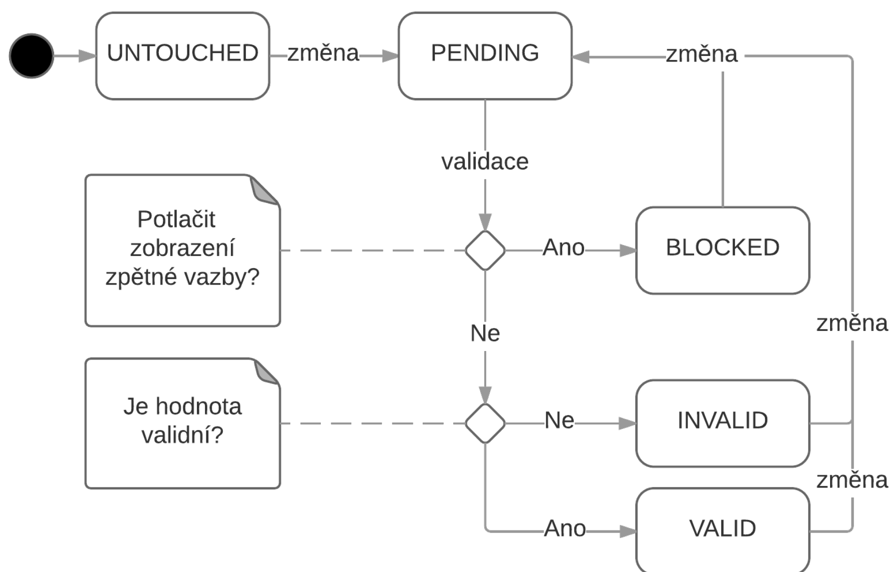
Obrázek 4.2: Diagram aktivit při odeslání formuláře

4.2.4 Stav validace

Vhodným prostředkem pro zjištění validity formuláře je přidání aktuálního stavu k jednotlivým komponentám modelu. Kontrola, zda jsou ve třídě `Validation` vytvořeny chybové hlášky totiž není dostatečná, jelikož mají

pouze informativní charakter pro uživatele. Navíc, odeslání dotazu na server a zpracování odpovědi trvá určitý čas, což znamená, že při kontrole validity modelu nemusí být data ještě k dispozici, tudíž bude model nesprávně považován za validní. Uvažme například situaci, kdy má formulář jedno pole s byznys pravidlem kontrolující neprázdnost atributu. Pokud uživatel nebude pole modifikovat a rovnou odešle formulář, dojde ke kontrole podmínky, jenž bude na straně klienta vyhodnocena jako nesplněná, ale k samotnému přidání zpětné vazby dojde až po načtení ze serveru. Jelikož je však model již kompletně zvalidován (událost `MODEL_VALIDATED`), dojde ke kontrole, zda je v modelu načtena chybová zpětná vazba. Klient však ještě nedostal lokalizovanou hlášku od serveru, tudíž je nevalidní model odeslán na server.

Dalším důvodem pro zavedení stavů atributů je situace, kdy je jedno pravidlo vázáno na více různých formulářových polí. Tuto situaci lze demonstrovat na příkladu, kdy formulář obsahuje dvě pole, jméno uživatele a příjmení. K příslušnému business kontextu se vztahuje pravidlo, jež kontroluje, zda jsou obě tato pole neprázdná. Pokud uživatel načte stránku a začne psát jméno do prvního pole, byla by zobrazena chybová hláška o nesplnění podmínky, přestože uživatel ještě nezačal upravovat příjmení. Ideální by tedy bylo, aby byla zpětná vazba pro tento typ pravidel zobrazována až tehdy, co uživatel upraví obě pole.



Obrázek 4.3: Diagram změn stavů formulářových polí

Pro řešení výše zmíněných důvodů je tedy vhodné definovat stavy jednotlivých polí určující stav validace, případně zda s nimi uživatel interagoval či nikoliv. Formulářové pole může mít následující stavy:

1. **Untouched** – Výchozí stav všech polí. Slouží k rozlišení prvků, se kterými uživatel ještě nemanipuloval.

2. **Pending** – Tento stav je nastaven poli bezprostředně po změně hodnoty a indikuje čekání na vyhodnocení validačního pravidla. Jakmile je známý výsledek, dojde ke změně stavu.

3. **Valid** – Slouží k označení pole, jehož hodnota je validní v souladu s pravidly, jež jsou na něj navázána.

4. **Invalid** – Označuje pole, jehož hodnota porušuje podmínku některého z pravidel.

5. **Blocked** – Tento stav slouží k označení polí ve výše zmíněném příkladu se jménem a příjmením, a vyjadřuje potlačení zobrazení zpětné vazby pro příslušné pole. Je nastaven takovému poli, ve kterém došlo ke změně hodnoty, a na nějž je navázáno pravidlo týkající se více polí, z nichž některé má aktuálně stav *untouched*.

Tento způsob označení stavu validace jednotlivých atributů je inspirován knihovnou *AngularJS* [1], ve které jsou podle stavu validace přiřazeny CSS třídy příslušným elementům. Změny stavů jsou ilustrovány na obr. 4.3. Při kontrole validity formuláře musí být všechny atributy modelu ve stavu *valid*, aby došlo k jeho odeslání na server.

4.3 Shrnutí

Cross-cutting povaha byznys pravidel zapříčiňuje množství problémů, jejichž řešení umožňuje AOP. Návrh integrace byznys pravidel do knihovny *Nutforms* se opírá o koncept ADDA, z nějž využívá způsob deskripce pravidel, k čemuž je využita knihovna Drools. Ta umožňuje separaci pravidel od výkonného kódu a jejich sdružení do znalostní báze, jež usnadňuje tvorbu pravidel i jejich údržbu. Neméně důležitá je možnost opětovného použití pravidel v různých vrstvách aplikace, čehož je docíleno transformací do vlastního modelu a jeho interpretace na příslušných místech.

Validační pravidla jsou v UI reprezentována funkcemi, které jsou navázány na jednotlivé atributy *rich modelu*, nebo na samotný model. Jejich aplikování na aktuální hodnoty je rozlišeno podle typu pravidla, které určuje konkrétní join point, do nějž bude přidána příslušná advice. Po vyhodnocení pravidla dojde ke změně stavu formulářového pole a do modelu je uložena lokalizovaná zpětná vazba, odkud je zobrazena uživateli. Životní cyklus formuláře z hlediska validace je založen na událostech, jež umožňují pohodlně oddělit prováděné akce do jednotlivých kroků.

[1] https://angularjs.org/

Kapitola 5
Implementace

Knihovna *Nutforms* představuje alternativní způsob návrhu uživatelského rozhraní prostřednictvím nástrojů aspektově orientovaného programování. Integrace byznys pravidel je navržena takovým způsobem, aby byla separována od výkonného kódu a sdružena do znalostní báze, jež slouží jako *single focal point* pro aspekty. Izolovaná pravidla by také měla být reprezentována v takové formě, která případně umožní jejich transformaci do různých vrstev aplikace, což současná řešení buď neumožňují vůbec, nebo pouze v omezeném rozsahu. Kompozice byznys pravidel a ostatních aspektů v UI je realizována *rich modelem*, jenž sdružuje oddělené aspekty a poskytuje snadný způsob pro automatické generování uživatelského rozhraní. Validační pravidla ve formulářích ovlivňují jejich životní cyklus, který je založen na systému událostí.

Implementace knihovny je rozdělena do dvou částí - klientské a serverové. Server-side funkcionalita je definována v projektu *nutforms-rules-server* a obsahuje parser, sloužící k transformaci pravidel z Drools DSL do vlastního modelu, a servlet, sloužící k jejich odeslání ke klientovi. Klientskou část aplikace reprezentuje projekt *nutforms-rules-web-client*, který zajišťuje propojení s knihovnou *Nutforms* a její obohacení o byznys pravidla. Klientská část také obsahuje aspect weaver, sloužící k transformaci aspektů a interpretaci pravidel v UI.

5.1 Použité technologie

Implementace řešení vychází z návrhu, který byl představen v kapitole 4. Některé z vybraných technologií způsobují omezení systému, nebo znemožňují ideální reprezentaci navržených konceptů, nicméně v současné chvíli slouží jako ověření konceptu. Navíc je systém navržen modulárním způsobem, který umožňuje poměrně snadné změny týkající se konkrétní implementace či výměny technologie.

5.1.1 Java

Implementace knihovny je realizována na platformě Java [23, 15], která nabízí vhodné nástroje pro navržený koncept. Java je často používána pro

vývoj enterprise aplikací s vrstevnatou architekturou, proto byla zvolena jako jazyk pro implementaci knihovny.

5.1.2 JavaScript

Pro realizaci klientské části knihovny je použit jazyk JavaScript, jenž je spolu s HTML a CSS standardním nástrojem pro tvorbu uživatelského rozhraní. V *Nutforms* je použita verze specifikace ECMAScript 6 (ES6) [16], přičemž kód je navíc zpracován kompilátorem *babel* [1], jenž před nasazením aplikace přeloží ES6 struktury do odpovídajících ekvivalentů v ES5, díky čemuž je umožněna kompatibilita s vyšším množstvím prohlížečů. JavaScript byl zvolen jako flexibilní nástroj, jenž umožní reprezentaci validačních pravidel na straně klienta v podobě funkcí, a zejména k implementaci runtime aspect weavingu pro korektní reflexi aktuálního execution kontextu. Čistá verze jazyka však bohužel neumožňuje typovou kontrolu, kterou sice některé JavaScript knihovny dodávají (například *TypeScript* [2]), nicméně záměrem *Nutforms* je minimální závislost na knihovnách třetích stran.

5.1.3 Drools

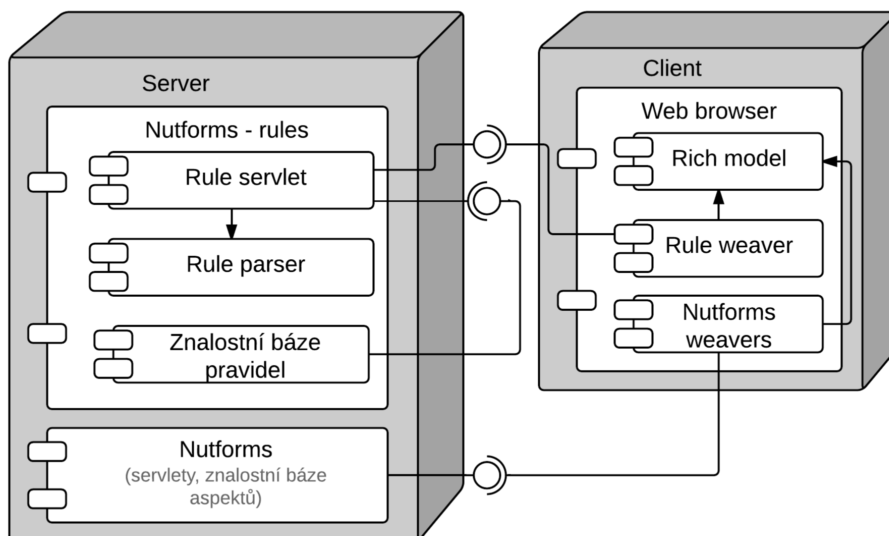
Knihovna JBoss Drools [2] slouží k zachycení byznys pravidel a jejich deskripci v doménově specifickém jazyce Drools DSL. Jednotlivá pravidla jsou spravována v separátních souborech, tedy dochází k efektivnímu oddělení pravidel od výkonného kódu. Navíc tento způsob zápisu tvoří znalostní bázi, ze které lze interpretovat pravidla na různá místa aplikace. Aktuálně používaná verze v knihovně *Nutforms* je Drools 6.

Použití doménově specifických jazyků pro popis aspektů je podle konceptu ADDA vhodným prostředkem, jelikož nabízí syntaxi tvořenou na míru specificky pro danou problematiku. Navíc lze takto vytvořené modely parsovat a distribuovat do více vrstev systému. Komplexita knihovny Drools však způsobuje jistá implementační omezení diskutované v kapitole 3.4, nejvhodnějším způsobem deklarace byznys pravidel by bylo použití vlastního DSL vytvořenému konkrétně pro účely knihovny *Nutforms*. Modularita systému však umožňuje snadnou změnu použitého jazyka pro popis aspektů pravidel za předpokladu zachování stejného rozhraní.

5.2 Server side

Na straně serveru je definován model, reprezentující jednotlivá pravidla, a parser, jenž slouží k inspekci pravidel deklarovaných v DSL a jejich interpretaci do tohoto modelu. Součástí je také servlet, který slouží jako centrální bod pro získávání pravidel a jejich transport na stranu klienta. Schéma komponent knihovny a komunikace mezi nimi je zobrazena na obr. 5.1

[1] https://babeljs.io/
[2] http://www.typescriptlang.org/



Obrázek 5.1: Diagram komponent knihovny a jejich komunikace

5.2.1 Deklarace pravidel

Samotná deklarace konkrétních pravidel není součástí knihovny *Nutforms*, ale je uvedena na straně serveru v projektu, jenž knihovnu využívá. Tato pravidla a způsob deklarace jsou však jejím základním prvkem, proto je jejich popis součástí tohoto textu.

K popisu byznys pravidel slouží jazyk Drools DSL, analýza alternativních možností a důvod výběru jsou uvedeny v kapitole 4. Jednotlivé `.drl` soubory jsou umístěny na serveru v adresáři `resources/rules`, spolu s ostatními aspekty a standardními webovými prostředky, jako konfigurační soubory, CSS, JS, nebo obrázky. Rozdělení pravidel do souborů a jejich sdružování do složek může být realizováno jakýmkoliv způsobem, jelikož business kontext je určen jménem balíku v souboru, nicméně doporučeným způsobem je vytvořit složky pro jednotlivé entity a pravidla sdružovat podle business kontextu z důvodu přehlednosti.

Samotným obsahem souborů je výčet jednotlivých pravidel. Na začátku každého pravidla je nutné deklarovat název business kontextu, ke kterému se pravidla vztahují, k čemuž je využit název balíku (*package*) souboru. Jmenná konvence pro názvy business kontextů je `FQN.akce`, kde `FQN` je plné jméno upravované entity a `akce` je název prováděné operace (např. *edit, new, list*).

Jednotlivá pravidla jsou deklarována standardním způsobem dle konvence Drools. Jejich název je využit jako identifikátor pro případné načtení lokalizované zpětné vazby v požadovaném jazyce, tedy jména pravidel by měla být unikátní v rámci business kontextu. Pro deklaraci podmínky je v souboru uvedena pouze *precondition*, tedy RHS je prázdná, jelikož konsekvence byznys pravidel je určena operací a může se lišit v různých vrstvách.

5.2.2 Rule parser

Důležitou vlastností aspektů při volbě implementace v projektech založených na AOP je jejich přenositelnost a schopnost transformace do různých vrstev. Knihovna Drools umožňuje deklaraci pravidel v rámci znalostních bází (*KieBase*) [2], kterých může být v aplikaci uvedeno více. Tyto báze obsahují kromě pravidel také deklarace funkcí, které jsou spolu s pravidly sdruženy podle výše zmíněných balíků. Pro jejich získání je třeba vytvořit instanci *KieSession*, jejímž prostřednictvím je možné například také nastavovat hodnoty globálních proměnných.

Třída `Inspector` v knihovně *Nutforms* slouží k transformaci deklarovaných pravidel do vlastního modelu (reprezentovaného třídou `Rule`). Její veřejné rozhraní specifikuje tři metody - inspekci celé *KieBase*, jež vrátí mapu všech business kontextů a odpovídající množiny pravidel, inspekci balíku, nebo přímo konkrétního pravidla. Jako vstupní argument pro poslední jmenovanou metodu je objekt, který byl vytvořen knihovnou Drools inspekcí pravidel deklarovaných v DSL, návratovou hodnotou je pak vytvořená instance třídy `Rule`.

Samotná transformace pravidla probíhá zejména pomocí reflexe [21]. Kromě jména a balíku jsou do objektu také uloženy globální proměnné a deklarace funkcí. Jejich použití však není v současné chvíli podporováno na straně klienta, jelikož jsou specifickým znakem pro Drools a umožnění jejich použití není nutné pro dokázání konceptu, nicméně jejich podpora je zahrnuta do budoucí práce. Důležitější je však unifikace různé reprezentace podmínek, kterou knihovna Drools rozlišuje podle typu - např. *pattern*, nebo *eval*. To je pro vyhodnocování v různých vrstvách zbytečně složitý způsob, jelikož podmínka může být reprezentována jako stromová struktura výrazů, nebo jako textový řetězec, jenž je uveden v `.drl` souboru. Pro implementaci byl zvolen textový řetězec, jelikož ho není třeba serializovat a může být rovnou odeslán na stranu klienta. Navíc může být na jeho základě vytvořen strom v cílové doméně.

5.2.3 Rule servlet

Poslední fází po inspekci pravidel je jejich transport ke klientovi. Ten zajišťuje Java servlet [9], jenž je namapován na adresu `/rules/*`, kde hvězdička odpovídá názvu business kontextu. Při konstrukci servletu jsou načtena jména kontextů z výchozí *KieSession*, což je session, jenž má v souboru `kmodule.xml` nastaven atribut `default` na `true`. Servlet přijímá `GET` požadavky a kontroluje, zda existují pravidla pro žádaný business kontext a pokud ano, provede inspekci příslušného souboru a vrátí množinu pravidel. K serializaci POJO objektů a transformaci do JSON formátu čitelného pro klienta je použita knihovna *Google Gson* [3].

[3] https://github.com/google/gson

5.3 Client side

Klientská část aplikace je zodpovědná za interpretaci přijatých pravidel a obohacení formuláře o funkcionalitu s nimi spojenou. Validace umožňuje okamžitou zpětnou vazbu pro uživatele a ovlivňuje životní cyklus formuláře, případně zamezuje jeho odeslání v nevalidním stavu. Integrace byznys pravidel do knihovny *Nutforms* je založena na systému událostí, na které systém reaguje a přidává požadovanou funkcionalitu. Také definuje vlastní události, jako například ověření validity formuláře při jeho odeslání, nebo stav validace jednotlivých částí modelu.

5.3.1 Aspect weaver

Prvním krokem je rozšíření struktury *rich modelu* a vytvoření validačních funkcí, k čemuž dochází po vybudování modelu knihovnou *Nutforms*, což vyvolává událost `MODEL_BUILT`. V této fázi je k modelu a všem jeho atributům přidána třída `Validation`, která slouží k ukládání zpětné vazby. Dále jsou definovány události při změně hodnoty atributu, čímž jsou rozlišeny join pointy, do kterých je později přidána příslušná advice. Po modifikaci modelu pro účely validace jsou ze serveru načtena pravidla pro aktuální business kontext, jež jsou rozdělena podle typu na validační a bezpečnostní. Validační pravidla jsou transformována do advice v několika krocích:

1. Nejprve jsou z pravidel extrahovány názvy atributů, ke kterým se dané pravidlo váže. To umožňuje později přiřadit vytvořené funkce k událostem příslušných formulářových polí.

2. Dalším krokem je transformace podmínky pravidla v podobě řetězce do funkce. K jejímu vyhodnocení je použita funkce `eval()`, nicméně podmínku je třeba upravit, jelikož syntaxe Drools DSL není přímo vyhodnotitelná v JavaScriptu. Úpravy je docíleno aplikováním regulárních výrazů na řetězec, jež přepisují například logické operátory, nebo způsob zápisu regulárních výrazů v podmínce.

   Součástí funkce je již také konsekvence byznys pravidla. V aplikační vrstvě má pro validační pravidla podobu vizuální zpětné vazby, která pro uživatele slouží jako indikátor chyb. Zpětná vazba je načtena v jazyce podle aktuálního user kontextu a při porušení podmínky pravidla je zobrazena u příslušných polí identifikovaných v předchozím kroku.

3. Nakonec jsou vytvořené funkce přidány k modelu a atributům jako posluchači nastalých událostí, tedy v AOP terminologii jsou navázány vytvořené advice do příslušných join pointů.

Další fází je aplikace bezpečnostních pravidel. V současné době knihovna podporuje jako tento typ taková pravidla, jež omezují množinu atributů, které může uživatel upravovat. Jako indikátor těchto pravidel slouží označení `[Security]` v jejich názvu, jelikož Drools neumožňuje efektivní zápis pravidel

tohoto typu, nicméně při použití jiného doménově specifického jazyka by bylo možné typy pravidel rozlišovat automaticky. Z takto identifikovaných pravidel jsou následně načteny názvy polí, které uživatel nemůže upravovat, a příslušným atributům je nastaven příznak `readOnly`. Ten je ve fázi generování formuláře reflektován knihovnou *Nutforms*, která pro takto označené atributy načte widgety určené pouze pro čtení.

5.3.2 Vyhodnocení pravidel

Aplikování *post-condition* validačních pravidel v prezentační vrstvě je realizováno jako zobrazení lokalizované zpětné vazby. Ta je načtena při vyhodnocení pravidla v jednotlivých join pointech - při změně hodnoty pole a při odeslání formuláře, k čemuž slouží události `ATTRIBUTE_CHANGED` a `FORM_SUBMITTED`. Její načítání je asynchronní, jelikož má pouze informativní charakter pro uživatele, tedy není určena k posuzování validity hodnoty. Změna zpětné vazby vyvolává události `ATTRIBUTE_VALIDATED`, případně `MODEL_VALIDATED`, jež způsobuje překreslení elementů na stránce, v nichž je zpětná vazba uložena.

Pro určení validity slouží stav atributů a modelu, k jehož změně dochází bezprostředně po vyhodnocení pravidla. Počáteční stav všech polí je *untouched*, který indikuje, že s polem ještě uživatel neinteragoval. Při změně je před vyhodnocením nastaven stav na *pending*, vyjadřující čekání na výsledek validace. Vyhodnocení pravidla způsobí změnu do jednoho ze tří stavů - *valid*, pokud je hodnota validní, *invalid*, pokud je podmínka vyhodnocena jako nesplněná, nebo *blocked*, jenž indikuje potlačení zobrazení zpětné vazby. Tento stav je využíván například pro pravidla, která se vážou k více polím, ale zpětná vazba při porušení podmínky má být zobrazena až poté, co uživatel interagoval se všemi z nich.

5.3.3 Odeslání formuláře

Poslední fází životního cyklu formuláře je jeho odeslání. To vyvolává událost `FORM_SUBMITTED`, na kterou reaguje validační část aplikace. Nejprve dojde k validaci všech atributů v důsledku jejich explicitního uložení do modelu. To je důležité pro situace, kdy uživatel odešle formulář, ve kterém jsou všechny atributy ve stavu *untouched*, jelikož na nich ještě nedošlo ke změně hodnoty a tedy neproběhla validace, tím pádem by mohlo dojít k odeslání nevalidního formuláře. Následně jsou spuštěna pravidla, jež jsou navázána na model. Ta vyvolávají stejný postup, jako při změně atributu - načtení lokalizované zpětné vazby, její vykreslení a nastavení modelu do příslušného stavu. Pokud není žádné pravidlo vztahující se k formuláři porušeno, dojde k vyvolání události `FORM_VALID`, na níž může uživatel knihovny reagovat například odesláním formuláře na server.

Kapitola 6

Testování

V kapitolách 4 a 5 je představen návrh struktury a implementace knihovny, jež identifikuje byznys pravidla v informačních systémech jako cross-cutting problémy a podle aspektově orientovaného paradigmatu a konceptu ADDA je separuje jako aspekt, který je následně transformován do uživatelského rozhraní. Cílem této kapitoly je ověřit platnost navrženého řešení pomocí jednotkových a integračních testů a demonstrovat využití knihovny na ukázkovém projektu reprezentující reálné prostředí.

6.1 Testy implementace

Návrh testů je orientován zejména na ověření funkčnosti klíčových prvků aplikace. Obě části knihovny (*nutforms-rules-server* a *nutforms-rules-web-client*) jsou pokryty jednotkovými a integračními testy, které ověřují korektní chování jednotlivých částí.

6.1.1 Použité technologie

JUnit

Pro implementaci testů na straně serveru je využita knihovna *JUnit 4* [1], jež je standardním nástrojem pro testování na platformě Java. Knihovna nabízí rozhraní pro vytváření jednotkových testů, spolu se sadou funkcí sloužící k ověření platnosti predikátů, jako například návrat nenulové hodnoty, nebo kontrola vyhození výjimky.

Mocha

Mocha [2] je JavaScript framework pro implementaci testů, spouštěný na platformě *Node.js* [3], nebo přímo v prohlížeči. Syntaxe knihovny je založena na principu *Behavior-driven development* (BDD) [33], jenž prosazuje použití

[1] http://junit.org/
[2] https://mochajs.org/
[3] https://nodejs.org/

přirozeného jazyka k popisu testovacích scénářů, což usnadňuje jejich pochopení pro všechny členy projektu. *Mocha* definuje funkce pro popis rozhraní testovaných tříd, díky kterým je definice testů i jejich zpětná vazba dostatečně čitelná nejen pro programátory.

Chai

Chai [4] slouží jako knihovna funkcí pro kontrolu platnosti predikátů. K těmto účelům definuje struktury jako například `should`, `expect`, nebo `assert`, jež lze využít ve většině JavaScript testovacích frameworků, včetně výše zmíněné knihovny *Mocha*. Řetězením funkcí lze dosáhnout snadno čitelných testů v souladu s konceptem BDD, jejichž struktura se téměř podobá větě v angličtině viz ukázka 6.1.

Zdrojový kód 6.1: Ověření predikátu v knihovně Chai

```
chai.should();
var expect = chai.expect;

user.should.have.property('name');
expect(user.name).to.be.a('string').with.length(5);
```

6.1.2 Typy testů

Jednotkové testování serverové části knihovny má za úkol zejména ověřit funkčnost parseru Drools pravidel (třída `Inspector`). K těmto potřebám je využita knihovna *JUnit* a sada testovacích `.drl` souborů s definovanými pravidly. Testovací scénáře pokrývají různé typy Drools pravidel, jako například *eval*, nebo *pattern*, které jsou parserem převedeny na univerzální model, který serverová část knihovny definuje. Mezi testovacími pravidly jsou i složitější deklarace, ve kterých je využito spojování výrazů logickými operátory. Dále také testovací pravidla obsahují deklarace proměnných, a to nejen lokálních, ale i globálních, jež se vážou k aktuální session, a je kontrolována jejich korektní interpretace. V neposlední řadě je v některých z pravidel definována i následná podmínka, která je však aktuálně v knihovně nevyužita, jelikož její podobu určuje cílová doména. Projekt používá pro správu buildů nástroj *Apache Maven* [5], jenž umožňuje integraci s testovacími frameworky, tedy testy jsou spuštěny při kompilaci knihovny a jejich selhání způsobí i neúspěch samotného buildu.

Pro realizaci jednotkových a integračních testů klientské části je využita kombinace knihoven *Mocha* a *Chai*. To umožňuje tvorbu velice snadno čitelných testů, *Mocha* navíc přináší i uživatelsky přívětivé rozhraní pro jejich spouštění prostřednictvím příkazové řádky. Testy v klientské části jsou primárně zaměřeny na kontrolu funkčnosti validačního aspektu a aspect weaveru, mimo to však také testují ostatní části, jako například callbacky událostí,

[4] http://chaijs.com/

[5] https://maven.apache.org/

třídy s pomocnými metodami, a další. Pro dodržení izolovanosti jednotkových testů jsou využity *mock* objekty.

Testy validačního aspektu pokrývají všechny jeho metody, včetně konstruktoru pro ověření správné inicializace atributů. Mimo to je kontrolována správná funkčnost při přijetí zpětné vazby ze serveru, tedy uložení nových chybových hlášek a smazání starých, pokud byly některé z chyb opraveny. Testy třídy `RuleWeaver` ověřují funkčnost jednotlivých metod, na které je proces aspect weavingu dekomponován, jelikož tvoří klíčový prvek pro interpretaci pravidel na straně klienta. V rámci této třídy jsou testovány i změny stavů jednotlivých částí modelu, viz kapitola 4.2.4. V ostatních testech je kontrolováno zejména vyvolávání správných událostí a korektní volání posluchačů, které jsou reprezentovány *mock* objekty.

Akceptační testování knihovny bylo prováděno manuálně s využitím ukázkové aplikace. V té je definována sada pravidel, podle kterých byla ověřována funkčnost příslušných částí. Akceptační testy byly zaměřeny na kontrolu integrace s knihovnou *Nutforms* a celkový vizuální stav uživatelského rozhraní.

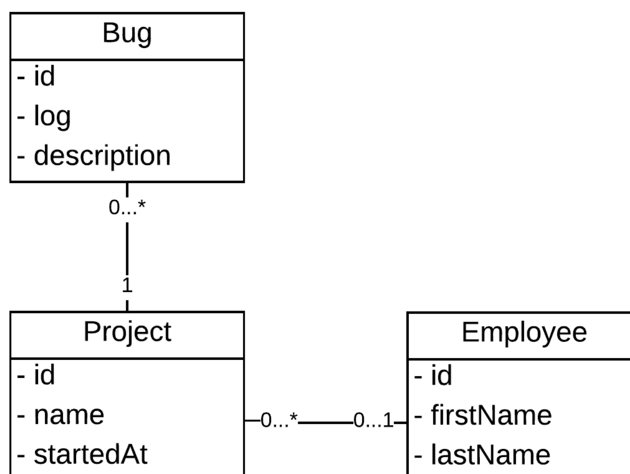
Pro automatické spouštění testů je využíván nástroj *Travis* [6], což je služba pro *continuous integration* (CI), tedy průběžnou integraci změn do projektu. Tento nástroj umožňuje mimo jiné automatické spouštění testů a kontrolu jejich výsledku při přidání změn do repozitáře projektu. Pro serverovou část provádí *Travis* build projektu pomocí nástroje *Apache Maven*, tedy dochází i ke kontrole závislostí, kompilace projektu a spuštění testů. Pro build klientské části *Travis* využívá *npm* a kromě spuštění testů je taktéž kontrolován stav knihoven, na kterých je projekt závislý.

6.2 Ukázková aplikace

K ověření navrženého konceptu v reálné aplikaci slouží projekt *nutforms-example*. Ten využívá knihovnu *Nutforms* a moduly umožňující integraci byznys pravidel. K nasazení aplikace je využit webový server *Apache Tomcat* [7], build projektu zajišťuje technologie *Apache Maven*.

Aplikace slouží jako jednoduchý issue tracking systém s datovým modelem viz obr. 6.1. Do systému je možné přidávat záznamy o nastalých problémech, jež jsou reprezentovány třídou `Bug`, která se váže ke konkrétnímu (právě jednomu) projektu. V rámci projektu může být reportováno více problémů. Projekt může mít také manažera, který je popsán třídou `Employee`, vyjadřující zaměstnance firmy, u kterého je evidováno jeho jméno a příjmení.

[6] https://travis-ci.com/
[7] https://tomcat.apache.org/



Obrázek 6.1: Doménový model ukázkové aplikace

Nad datovým modelem je definována sada CRUD operací - *create*, *read*, *update* a *delete*, které definují business kontexty, jež obsahují jednotlivá pravidla. Operace zároveň také určují možné uživatelské scénáře - vytvoření, úprava, zobrazení, či smazání zaměstnance, nebo problému. Odlišení vazeb scénářů a uživatelů by bylo možné například na základě uživatelských rolí.

Aspekty v ukázkové aplikaci využívané knihovnou *Nutforms* jsou definovány standardně v adresáři `resources`, kde jsou rozděleny podle typu do izolovaných znalostních bází. Organizace byznys pravidel je pro přehlednost členěna v první úrovni adresářové struktury podle názvu entity, ke které se pravidla vážou, a následně do jednotlivých souborů, jež jsou pojmenovány podle názvu business operace (CRUD). Bezpečnostní pravidla jsou odlišena sufixem `*-security`. Jména balíků v jednotlivých souborech jsou pak složena z názvu entity a akce, podle čehož jsou načítány, tedy samotné názvy souborů a jejich adresářová struktura slouží pouze ke zpřehlednění pro vývojáře.

Pravidla pro jednotlivé business kontexty jsou definována takovým způsobem, aby bylo pokryto co největší množství jejich typů. Mezi validační pravidla problémů patří například kontrola minimální a maximální délky popisu, která spojuje jednoduché podmínky logickým operátorem (viz obr. 6.2), nebo kontrola regulárním výrazem, zda popis obsahuje pouze alfanumerické znaky a mezeru. Příkladem validačního pravidla s vazbou na zaměstnance je kontrola neprázdnosti údajů, kdy musí být vyplněné alespoň jméno, nebo příjmení. Toto pravidlo je kontrolováno při odeslání formuláře, na rozdíl od ostatních, kde je zpětná vazba zobrazována okamžitě při úpravě hodnoty pole. Dalším typem pravidel jsou security pravidla, která zajišťují, že pro kontexty *read* a *delete* není možné upravovat hodnoty formulářových polí.

Jazyková data pro zpětnou vazbu byznys pravidel jsou načítána ze znalostní báze lokalizačního aspektu. Jednotlivé soubory určují jazykovou verzi, samotný překladový klíč je pak vytvořen podle konvence `rule.FQN.operace.pravidlo`.

Zdrojový kód 6.2: Deklarace pravidla v ukázkové aplikaci

```
package cz.cvut.fel.nutforms.example.model.Bug.new;
import cz.cvut.fel.nutforms.example.model.Bug;
dialect  "mvel"

rule "[Bug] Is description properly long"
    when
        Bug(description != null && description.length > 14
        && description.length < 31)
    then
end
```

`FQN` je plné jméno entity včetně balíku, `operace` je název prováděné byznys operace, a `pravidlo` je jméno pravidla, pro které je načítána zpětná vazba. Chybové hlášky byznys pravidel v ukázkové aplikaci jsou plně lokalizovány do češtiny a angličtiny.

V příloze C jsou ukázky snímků obrazovek pro různé kontexty. Obrázek C.1 ukazuje vykreslení zpětné vazby pro všechna nevalidní pole formuláře. Na snímku C.2 je demonstrována zpětná vazba pravidla pro nevalidní model, které je kontrolováno při odeslání formuláře. Poslední obrázek C.3 ukazuje needitovatelná pole pro kontext *read*, která jsou určena bezpečnostním byznys pravidlem.

6.2.1 Integrace knihovny

Integrace knihovny *Nutforms* do aplikace je docíleno načtením jednotlivých modulů do projektu. Pro serverovou část je třeba načíst zkompilované *Maven* artefakty do souboru `pom.xml`, tedy projekty *nutforms-server* a *nutforms-rules-server*. Klientská část knihovny je publikována jako *npm* [8] modul (*nutforms-rules-web-client*), tedy stačí na něj pouze vytvořit závislost například pomocí knihovny *webpack* [9] a následně načíst příslušný skript do stránky.

Pro vygenerování formuláře z definovaných aspektů stačí pouze použít funkci `Nutforms.generateForm()` a dodat jí příslušné argumenty, mezi něž patří mj. název entity, jméno business kontextu, nebo požadovaný jazyk. Pokud jsou v projektu použity moduly *nutforms-rules* a pro daný kontext jsou definována byznys pravidla, dojde k automatickému obohacení formuláře o jejich funkcionalitu. Pro použití modulu integrující pravidla je však nutná závislost na *Nutforms*, jelikož představuje jádro knihovny.

[8] https://www.npmjs.com/
[9] https://webpack.github.io/

Kapitola 7

Závěr

Návrh současných informačních systémů je často založen na třívrstvé architektuře, která se zaměřuje na oddělení prezentační vrstvy, aplikační logiky a datové vrstvy. Efektivní zachycení byznys pravidel v tomto typu aplikací je však právě kvůli separaci vrstev náročným úkolem, jelikož pro zachování konzistence je třeba vyhodnocovat stejná pravidla v různých úrovních, k čemuž jsou často použity rozdílné technologie. Důsledkem toho je rozsáhlá duplicita kódu, která navíc výrazně ztěžuje údržbu těchto pravidel a zvyšuje náchylnost k chybám.

Cílem této práce bylo analyzovat problematiku reprezentace byznys pravidel a navrhnout knihovnu umožňující jejich efektivní použití v uživatelském rozhraní. Analýza domény ukázala, že duplicita a decentralizovanost pravidel jsou zásadní problémy při jejich správě a při integraci změn, jelikož je třeba modifikovat všechna místa deklarace pravidla. Časová náročnost tohoto procesu je neúměrná náročnosti požadavku (například zvýšení minimálního věku uživatelů z 15 na 18 let), navíc obnáší manuální rutinní práci, která má velmi vysoké riziko tvorby chyb. Souvisejícím problémem je nemožnost opětovného použití deklarovaných pravidel, jež přispívá k duplicitě ve vertikálním i horizontálním směru.

V rešerši stávajících řešení byly představeny knihovny a frameworky řešící zmíněnou problematiku, nicméně se ukázalo, že dostupné technologie sice umožňují částečnou znovupoužitelnost deklarovaných pravidel, bohužel však ne v plném rozsahu všech vrstev aplikace a často se soustředí pouze na vybraný typ byznys pravidel. Jako vhodný nástroj k návrhu informačních systémů se zaměřením na cross-cutting problémy, mezi něž patří i byznys pravidla, bylo vybráno aspektově orientované paradigma, které slouží k dekompozici systému na jednotlivé aspekty, jež jsou izolovány v centrální znalostní bázi a následně automaticky transformovány do příslušných míst.

V rámci této práce byla navržena knihovna pro flexibilní tvorbu pravidel a jejich využití v uživatelském rozhraní s důrazem na jejich znovupoužitelnost v ostatních vrstvách aplikace. K tomu je využit koncept ADDA, jenž se zaměřuje na identifikaci aspektů v systému a usiluje o řešení nedostatků objektově orientovaného programování z hlediska cross-cutting funkcionality. Funkčnost implementované knihovny byla ověřena sadou jednotkových a integračních testů a demonstrována na ukázkové aplikaci.

7.1 Přínos práce a omezení

Navržený koncept umožňuje řešit problém duplikace kódu v důsledku linearizace multidimenzionálního prostoru, kvůli níž dochází ke ztrátě ortogonality os vyjadřující jednotlivé aspekty. Byznys pravidla, jakožto jeden z aspektů, jsou deklarována na jednom místě, které tvoří znalostní bázi, jež umožňuje reflexi provedených změn do všech míst aplikace pravidla, čímž dochází k významnému usnadnění integrace změn. K jejich popisu je využit v souladu s přísupem ADDA doménově specifický jazyk Drools DSL, jenž slouží k zachycení podmínek a sdružení pravidel do business kontextů.

Dalším přínosem práce je efektivní separace byznys pravidel od kódu uživatelského rozhraní. Díky transformaci aspektů a automatickému generování funkcí aspect weaverem je docíleno úplné absence deklarací pravidel v uživatelském rozhraní, což snižuje komplexitu kódu bez ztráty požadované funkcionality. Tento přístup umožňuje vývojářům soustředit se odděleně na vzhled UI a kontrolu validačních a dalších pravidel, čímž lze například snadněji rozdělit práci mezi vícero doménových expertů.

Způsob deklarace pravidel navíc umožňuje jejich znovupoužitelnost v dalších vrstvách aplikace, případně i přes více platforem, díky čemuž lze výrazně redukovat množství duplicitního kódu. Unifikace použitého jazyka pro deskripci aspektů umožňuje centrální správu, což eliminuje riziko tvorby chyb z důvodu opomenutí jednoho z míst aplikace pravidla. Tento přístup také usnadňuje testování systému, jelikož testy stačí aplikovat pouze na aspect weaver a kód zajišťující integraci advice, tedy výsledný vygenerovaný kód již nemusí být testován.

Z použitých technologií k implementaci návrhu však pramení jistá omezení systému. Zejména knihovna Drools představuje z důvodu její komplexity problematickou součást, jelikož například neumožňuje efektivní sdružování pravidel do business kontextů. Použití stejného typu pravidla ve více kontextech způsobuje horizontální duplikaci, která sice není tak závažná, jako vertikální, nicméně zabraňuje dosažení ideálního řešení problematiky. Eliminace tohoto problému lze docílit změnou doménově specifického jazyka pro popis aspektů, což díky modularitě systému není náročný proces.

Dalším omezením je množina typů deklarovatelných pravidel. V současné chvíli systém umožňuje pouze tvorbu validačních a bezpečnostních pravidel, která slouží k zamezení úpravy vybraných atributů uživatelem. V rámci budoucího rozvoje je však cílem tuto množinu rozšířit.

7.2 Budoucí rozvoj

Implementace navrženého konceptu a ověření jeho základní myšlenky na ukázkové aplikaci dokazuje jeho použitelnost a možnost uplatnění v současných systémech, které trpí problémy spojené s neefektivním zachycením cross-cutting byznys pravidel. Knihovna je cílena spíše na větší enterprise systémy s velkým množstvím pravidel, jelikož pro ně budou přínosy práce

nejmarkantnější a redukce duplicity přinese nejvíce benefitů. K využití v reálných aplikacích by však bylo vhodné knihovnu rozšířit a minimalizovat její stávající omezení.

Jedním z důležitých kroků je zjednodušení použitého doménově specifického jazyka. Drools nabízí komplexní funkcionalitu, ze které knihovna používá pouze malou část, přičemž k popisu byznys pravidel by stačil mnohem jednodušší DSL. Ideálním řešením by byl vlastní jazyk navržený přímo pro potřeby knihovny, spolu s kompilátorem, který by umožňoval transformaci popsaných pravidel do modelu, jenž by mohl být interpretován do příslušných domén. Změna jazyka by přinesla jednodušší způsob deklarace pravidel a přehlednější organizaci znalostní báze.

Další změna se týká sdružování pravidel. V současném stavu nelze kvůli omezením použitého jazyka efektivně sdružovat pravidla do business kontextů bez jejich duplicitní deklarace, pokud je jedno pravidlo součástí více kontextů. Při výměně DSL by byl tento bod jedním ze základních požadavků, jelikož zamezuje dosažení maximální efektivity a nulové duplicity při integraci byznys pravidel.

Budoucí úpravy knihovny by se také mohly věnovat rozšíření sady podporovaných typů byznys pravidel, jelikož současně lze deklarovat pouze validační a bezpečnostní pravidla, což však dokazuje schopnost knihovny využít deklarace ke zcela odlišným účelům. V budoucnu by mohla knihovna umožňovat například reflexi uživatelských rolí a skrývání částí formuláře na jejich základě, případně při zachování jazyka Drools DSL by byla vhodná integrace globálních proměnných a funkcí na straně klienta.

7.3 Shrnutí

Cílem práce bylo analyzovat doménu byznys pravidel a jejich reprezentaci v informačních systémech a navrhnout řešení, jež by umožňovalo jejich využití v uživatelském rozhraní. Na základě aspektově orientovaného přístupu se podařilo implementovat knihovnu, která izoluje byznys pravidla od výkonného kódu a soustředí je ve znalostní bázi, odkud jsou automaticky transformována do prezentační vrstvy. Navržené řešení zjednodušuje správu těchto pravidel, redukuje manuální duplicitu kódu a umožňuje opětovné použití deklarovaných byznys pravidel v dalších vrstvách aplikace. Všechny stanovené cíle práce se podařilo splnit.

Literatura

[1] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[2] Michal Bali. *Drools JBoss Rules 5.0 Developers Guide*. Packt Publishing, 2009. ISBN: 1847195644.

[3] Emmanuel Bernard. Bean Validation 1.1. JSR 349, 2013.

[4] Emmanuel Bernard and Steve Peterson. Jsr 303: Bean validation. *Bean Validation Expert Group*, 2009.

[5] Karel Cemus and Tomas Cerny. Aspect-driven design of information systems. In *SOFSEM 2014: Theory and Practice of Computer Science*, pages 174–186. Springer, 2014.

[6] Karel Cemus, Tomas Cerny, and Michael J Donahoo. Automated business rules transformation into a persistence layer. *Procedia Computer Science*, 62:312–318, 2015.

[7] Tomas Cerny, Karel Cemus, Michael J Donahoo, and Eunjee Song. Aspect-driven, data-reflective and context-aware user interfaces design. *ACM SIGAPP Applied Computing Review*, 13(4):53–66, 2013.

[8] Tomas Cerny, Michael J Donahoo, and Eunjee Song. Towards effective adaptive user interfaces design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 373–380. ACM, 2013.

[9] Shing Wai Chan and Rajiv Mordani. Jsr-340 java servlet 3.1 specification. *Oracle America, Inc*, page 5, 2013.

[10] Codehaus. MVEL Language Guide. `https://en.wikisource.org/w/index.php?title=MVEL_Language_Guide&oldid=6153886`, 2016. [cit. 2016-05-05].

[11] Jim Conallen. Modeling web application architectures with uml. *Communications of the ACM*, 42(10):63–70, 1999.

[12] World Wide Web Consortium et al. Html5 specification. *Technical Specification, Jun*, 24:2010, 2010.

[13] Douglas Crockford. The application/json media type for javascript object notation (JSON). RFC 4627, 2006.

[14] Chris J. Date and Hugh Darwen. *A Guide to SQL Standard (4th Edition)*. Addison-Wesley Professional, 1996. ISBN: 0201964260.

[15] Linda DeMichiel and William Shannon. JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification, 2011.

[16] ECMA Ecma. Ecmascript 2015 language specification. *ECMA (European Association for Standardizing Information and Communication Systems)*, 2015.

[17] Hardy Ferentschik and Gunnar Morling. Hibernate validator JSR 349 reference implementation 5.1. 3. final, 2014.

[18] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–HTTP/1.1, 1999.

[19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[20] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[21] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications, 2004. ISBN: 1932394184.

[22] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN: 0321127420.

[23] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition (Java Series)*. Addison-Wesley Professional, 2014. ISBN: 013390069X.

[24] Martin Johns. Code-injection vulnerabilities in web applications—exemplified at cross-site scripting. *it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(5):256–260, 2011.

[25] Richard Kennard and Robert Steele. Application of software mining to automatic user interface generation. In *International Conference on Software Methods and Tools*. IOS Press, 2008.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-oriented programming*, pages 220–242. Springer, 1997.

[27] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2004. ISBN: 0131489062.

[28] Brian Leathem, Lukas Fryc, and Sean Rogers. Red Hat JBoss Web Framework Kit 2 RichFaces Component Reference Guide, 2013.

[29] Peter Mularien. *Spring Security 3*. Packt Publishing, 2010. ISBN: 1847199747.

[30] Ronald G. Ross. What's wrong with if-then syntax for expressing business rules-one size doesn't fit all. *Business Rules Journal*, 8(7), 2007.

[31] Bran Selic. The pragmatics of model-driven development. *IEEE software*, 20(5):19, 2003.

[32] Leon Shklar and Richard Rosen. Web application architecture. *JohnWilley & Sons, Ltd*, 2003.

[33] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Objects, models, components, patterns*, pages 269–287. Springer, 2012.

[34] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[35] Karel Čemus. Automatická integrace byznys pravidel v adaptabilních uživatelských rozhraních. Master's thesis, České vysoké učení technické v Praze, 2013.

Seznam použitých zkratek

ADDA Aspect-Driven Design Approach

AOP Aspect Oriented Programming

API Application Programming Interface

BDD Behavior-Driven Development

CI Continuous Integration

CRUD Create, Read, Update, Delete

CSS Cascading Style Sheets

DSL Domain Specific Language

DTO Data Transfer Object

ES ECMAScript

FQN Fully Qualified Name

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSF JavaServer^{TM} Faces

JSON JavaScript Object Notation

JSR Java^{TM} Specification Request

LHS Left Hand Side

MDD Model-Driven Development

MVC Model-View-Controller

MVEL MVFLEX Expression Language

OOP	Object Oriented Programming
ORM	Object-Relational Mapping
POJO	Plain Old Java Object
READ	Rich Entity Aspect/Audit Design
REST	Representational State Transfer
RHS	Right Hand Side
SQL	Structured Query Language
UI	User Interface
UX	User Experience

Příloha B

Instalační příručka

B.1 Nutforms

Knihovna *Nutforms* je rozdělena do několika separátních modulů, včetně podpory byznys pravidel:

- **nutforms-server** – Serverová část jádra knihovny definující servlety pro načítání metadat, lokalizace, layoutů a widgetů.
- **nutforms-web-client** – Klientská část jádra knihovny, která umožňuje automatické generování formulářů.
- **nutforms-rules-server** – Serverová část modulu pro integraci byznys pravidel. Obsahuje parser pro transformaci Drools pravidel do objektů a servlet pro jejich odeslání na stranu klienta.
- **nutforms-rules-web-client** – Klientská část modulu pro integraci byznys pravidel. Poskytuje napojení na životní cyklus *Nutforms*, definuje aspect weaver a třídy pro obsluhu validace.

Kompilace serverových částí knihovny probíhá pomocí nástroje *Apache Maven*, jenž slouží ke správě závislostí a buildů. Ten lze spustit příkazem `mvn package` v kořeni adresářové struktury projektu, který vytvoří `.jar` soubor v adresáři `target`. Ke kompilaci klientské části je využit nástroj *webpack*, který je spouštěn stejnojmenným příkazem taktéž v kořeni projektu. Zkompilovaný skript `nutforms-rules-web-client.js` bude vytvořen v adresáři `dist`.

Testy serverové části jsou automaticky spouštěny při kompilaci projektu, nicméně je lze spustit i samostatně příkazem `mvn test`. Klientská část aplikace využívá pro spuštění testů příkaz `npm test` v kořeni projektu, který je aliasem pro zavolání skriptu *mocha* s příslušnými parametry.

Využití knihovny v projektu je docíleno v několika krocích. Předpokladem je projekt na platformě Java a použití nástroje *Apache Maven*.

1. Nejprve je třeba lokálně nainstalovat moduly *nutforms-server* a *nutforms-rules-server* příkazem `mvn install` v jejich kořenech.

2. Dále je třeba přidat do projektu závislosti na knihovně, tedy vložit do souboru `pom.xml` následující blok:

```
<dependency>
    <groupId>cz.cvut.fel.nutforms</groupId>
    <!--meta/layout/widget/localization/rules-->
    <artifactId>rules</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

   Pro využití celého rozsahu knihovny je třeba deklarovat pět závislostí na jednotlivé moduly, jež se liší hodnotou `artifactId` - `meta`, `layout`, `widget`, `localization` a `rules`.

3. Posledním krokem je připojení klientských modulů. Ty jsou publikovány jako *npm* moduly, tedy lze na ně snadno přidat závislosti do konfigurace projektu v souboru `package.json` a následně zkompilovat příkazem `webpack`:

```
"dependencies": {
  "nutforms-rules-web-client": "*",
  "nutforms-web-client": "*"
}
```

   Alternativně lze přímo zkopírovat zkompilované skripty v adresáři `dist`. Ty je třeba následně (i při deklaraci závislosti v `package.json`) načíst do stránky.

B.2 Ukázková aplikace

Ukázková aplikace je realizována projektem *nutforms-example*. Obsahuje malé množství entit, business kontextů a pravidel pro demonstraci navrženého konceptu.

Databáze je v projektu využita *in-memory*, tedy není třeba ji manuálně vytvářet. V projektu je vytvořena konfigurace připojení a při spuštění aplikace jsou vygenerována testovací data.

Spuštění aplikace vyžaduje prostředí pro běh Java aplikací, nástroj *Apache Maven* pro kompilaci a libovolný servletový kontejner, například *Apache Tomcat*. Build projektu je spuštěn příkazem `mvn package` v jeho kořeni, následně je třeba nasadit projekt na webový server. Toho je docíleno zkopírováním vygenerovaného `.war` souboru z adresáře `target` do složky `webapps` v instalačním adresáři Tomcatu.

Příloha C

Snímky obrazovky

Create Bug

ID

Unique id, will be generated automatically, do not fill.

Description

Illegal-value!

Description must be between 15 and 30 characters long

Description can only contain letters, numbers and spaces

Description of the Bug. Please be as concrete as possible, so we can replicate fast and don't have to ask you.

Log

Log can not be empty

Stack trace of exception or just the error message.

Project

1

Choose the Project for which the Bug is reported.

Create

Obrázek C.1: Zpětná vazba pro nevalidní hodnoty při vytvoření problému

Edit Employee

Fill in the name

ID

1

First name

Last name

Edit

Obrázek C.2: Zpětná vazba pro nevalidní model při editaci zaměstnance

Read Employee

ID

1

First name

Dominic

Last name

Strother

Read

Obrázek C.3: Needitovatelná pole pro kontext *read*

Příloha D

Obsah přiloženého CD

Zdrojové kódy knihoven jsou také dostupné online v repozitářích *jSquirrel/nutforms-rules-server* [1] a *jSquirrel/nutforms-rules-web-client* [2].

```
|-- nutforms-example                ukázková aplikace
| |-- dist                          zkompilované zdrojové kódy
| |-- src                           zdrojové kódy
| |-- README.md                     návod k instalaci
|
|-- nutforms-rules-server           serverová část knihovny
| |-- dist                          zkompilované zdrojové kódy
| |-- docs                          dokumentace v HTML
| |-- src                           zdrojové kódy
| |-- README.md                     návod k instalaci a popis knihovny
|
|-- nutforms-rules-web-client       klientská část knihovny
| |-- dist                          zkompilované zdrojové kódy
| |-- docs                          dokumentace v HTML
| |-- src                           zdrojové kódy
| |-- test                          zdrojové kódy testů
| |-- README.md                     návod k instalaci a popis knihovny
|
|-- text                            text bakalářské práce
| |-- src                           zdrojové kódy pro LaTeX
| |-- bachelor.pdf                  soubor s textem bakalářské práce
|
|--README.txt                       tento obsah CD
```

[1] https://github.com/jSquirrel/nutforms-rules-server
[2] https://github.com/jSquirrel/nutforms-rules-web-client