České vysoké učení technické v Praze

Fakulta elektrotechnická

Katedra počítačů

Bakalářská práce 2014/2015

Efektivita technik pro implementaci byznys pravidel v informačních systémech

Petr Tománek

Vedoucí: ***Karel Čemus, Ing.***

Červen 2014/2015

Poděkování

Děkuji Ing. Petru Aubrechtovi za rady a nasměrování, které mi pomohly více pochopit téma a zejména děkuji Ing. Karlu Čemusovi za věcnost, trpělivost a profesionální přístup při vedení této bakalářské práce. Jsem velmi rád, že právě on byl vedoucím tohoto projektu.

Prohlášení

Tímto prohlašuji, že jsem zadanou bakalářskou práci vypracoval samostatně pod vedením Ing. Karla Čemuse a uvedl v seznamu literatury veškerou použitou literatiru a další zdroje. Souhlasím se zveřejněním této práce.

V Praze dne ....

Obsah

1 Úvod 2

2 Analýza 3
2.1 Současná řešení . . . 3
2.2 Řešení v prezentační vrstvě . . . 4
2.3 Požadavky . . . 5
2.4 Databáze . . . 6
2.5 Vrstvy . . . 6

3 Rešerše 8
3.1 Minulost a současný stav . . . 8
3.2 Nedostatky současného vývoje . . . 8
3.3 Řešení nedostatků . . . 9

4 Řešení pravidel v service 11
4.1 Základní pojmy . . . 11
4.2 Neoptimalizované programování . . . 12
4.3 Objektově Orientované Programování - zapouzdření . . . 13
4.4 Objektově Orientované Programování - vzory . . . 14
4.5 Meta-programování . . . 15
4.6 Aspektově Orientované Programování . . . 16

5 Návrh případové studie 17
5.1 Základní pojmy . . . 17
5.2 Návrh . . . 17
5.3 Vrstvy . . . 17

6 Implementace případové studie 19
6.1 Datová vrstva . . . 19
6.2 Aplikační vrstva - Neoptimalizované řešení . . . 19
6.3 Aplikační vrstva - Objektové programování - zapouzdření . . . 20
6.4 Aplikační vrstva - Objektové programování - Chain of Responsibility . . . 21
6.5 Aplikační vrstva - Meta-programování . . . 22
6.6 Aplikační vrstva - Aspektově Orientované Programování . . . 24
6.7 Prezentační vrstva . . . 25

7 Měření 27
7.1 Měření řádků kódu . . . 27
7.2 Měření výskytu stejného pravidla . . . 27
7.3 Měření cyklomatické složitosti business metod . . . 28

8 Závěr **30**

9 Literatura **31**

Abstrakt

Tato práce řeší problémy byznys logiky webových aplikací, jako jsou duplicity, (ne)čitelnost a složitost vývoje a údržby. V řešení byla vytvořena případová studie zpracovaná několika řešeními pro byznys vrstvu třívrtsvé architektury. Provedeným výzkumem byly zjištěny rozdíly, výhody a nevýhody pro jednotlivá řešení. Výsledky této práce popisují jak tvorbu webových aplikací zjednodušit a zefektivnit.

Klíčová slova

Aspektově Orientované Programování, **Meta-programování**, **Objektově Orientované Programování**, Webové aplikace, Efektivita

Abstract

This work addresses the issues of business logic of web applications, such as duplicates, (not)readability and complexity of development and maintenance. In the solution was developed a case study created by several solutions for the business layer of the three-tier architecture. Carried out research has found differences, advantages and disadvantages for particular solutions. The results of this study illustrates how to create simplified and streamlined web applications.

Keywords

Aspect Oriented Programming, **Meta-programming**, **Object Oriented Programming**, Web applications, Efficiency

1 Úvod

Webové aplikace zažívají v posledních letech boom a proto je na místě diskuze o kvalitách jednotlivých programovacích technik vzhledem k jejich využití v této problematice.

První webové stránky vznikly v roce 1993, ale až o několik let později se staly firemní vizitkou, kterou "musí mít" každá firma. V dnešní době (2015) jsou všední webové stránky pro firmy, osobní stránky, nebo blogy, případně i pro projekty, ale také tzv. webové aplikace, kde uživatel má určitá práva pro přidávání, editaci a odebírání různých obsahů.

Existuje množství programovacích jazyků, typů programování a také druhů provedení webových aplikací. Druhy webových aplikací mohou být statické, či dynamické. Statické, také jinak nazývané stálé, se zjednodušeně nazývají webové stránky, protože zde není interakce se systémem, která by měnila jakkoli obsah, pouze přesměrování na jiné stránky. Tyto webové aplikace se od doby, kdy jsou umístěny na internet, bez práce programátorů, nemění. Naproti tomu dynamické webové aplikace se průběžně mění i po nahrání, bez nutnosti přeprogramovávat aplikace. Tyto změny jsou prováděny administrátory, či dokonce běžnými uživateli, při interakci se serverem. Další možností jak můžeme dělit webové aplikace je podle toho, jak zpracovávají interakce s uživateli. Jsou buď na klientské straně, či na straně serverové.

Tato bakalářská práce se zabývá dynamickou webovou aplikací na serverové straně především pro složitost aplikací, náročnost a také nákladnost na vývoj a jejich další údržbu. Navíc díky velkému množství technik, které lze použít na toto zpracování, je v dnešní době velmi důležité využít jejich výhod. Aby se daly tyto výhody využít je potřeba nejdříve jednotlivé techniky pochopit, porovnat a poté vyhodnotit.

Jak již bylo řečeno, webové aplikace jsou nesmírně složité, velmi rozsáhlé, náročné a tím i nákladné jak na vývoj tak i na údržbu. Avšak asi nejdůležitější je spolehlivost, respektive chybovost. Chyby především v byznys logice, tedy v pravidlech, která musí být splněna jsou nesmírně problematické na zpracování i údržbu. Tato pravidla se nachází napříč celými aplikacemi, od ověřování již u klienta až po ověřování na straně serveru. Tato pravidla je dle technik, které se používají v dnešní době, takřka nemožné rozložit do souvislých celků a tak i jednoduše zpracovávat a upravovat. Takový kód je pak komplikovaný, hůře čitelný a drahý na údržbu. Pokud vezmeme v úvahu, že vysoký počet webových aplikací je například pro banky, či jiné velké společnosti a také fakt, že každá chyba může znamenat nevyčíslitelný problém, pak je potřeba se na této problematice více věnovat.

Touto prací bych chtěl porovnat jednotlivé techniky vývoje a vyhodnotit jaká z nich se dá využít v jakém případě. Tím bych si přál zvýšit spolehlivost, zjednodušit a zrychlit a díky tomu i zlevnit jak vývoj tak i údržbu.

2 Analýza

V dnešní době je kladen nedostatečný důraz na kvalitu při vývoji webových aplikací. Často jsou aplikace vyvíjeny narychlo, aby co nejrychleji fungovaly a poté tápou při údržbě a postrádají jak čitelnost, tak jednoduchost provedení pro úpravy. V takových systémem pak je jednoduché něco přehlédnout a tak se nevyhnout chybám. Takové úpravy, které musí být bezchybné, jsou pak velmi složité, drahé a navíc ještě velmi často více znepřehledňují již tak nepřehledný kód.

2.1 Současná řešení

Při pohledu na to, jak se vyvíjí webové aplikace dnes a jak byly dříve vyvíjeny, tak nebude vidět příliš velký rozdíl. Za několik posledních let, přibližně od roku 2000, až do dnes, rok 2015, bude zřejmé, že situace ve zpracování webových aplikací dosti stagnuje. Stále se velmi často webové aplikace vyvíjí pomocí tzv. Neoptimaliovaného programování, či trochu kvalitněji zpracovávají pomocí Objektově Orientovaného Programování. Výhodami jsou jednoznačně jednoduchost a to, že jsou dynamicky měnitelná pravidla.

Listing 2.1: Řešení pomocí Neoptimalizovaného programování

```
public void method() {
    if(..) {..}
    if(..) {..}
    //code
}
```

Takovéto (2.1) řešení je pak zjednodušené řešení pro stejné řešení (2.2), jen rozepsané pro větší přehlednost. Tato řešení pak mohou obsahovat další na sebe navazující podmínky, které asymptoticky znepřehlední kód jejich počtu.

Listing 2.2: Řešení pomocí Neoptimalizovaného programování - rozepsané

```
public void method() {
    if(..) {..}
    else{
        if(..) {..}
    }
    //code
}
```

V případě využití Objektově Orientovaného Programování je to takřka stejné, s vyjímkou toho, že je aplikace trochu přehlednější, má oddělená pravidla a o třídu navíc. Proto se zde,v této kapitole, tímto řešením nebudu zabývat.

Řešením této problematiky je taktéž specifikace JSR 316 [1]. Tato specifikace popisuje Java EE 6. Novinkou přidanou v JSR 316 je například ManagedBean (neplést si s JSF ManagedBean), která zařizuje Context or Dependency Injection (CDI), v čj. doslovně vkládání kontextů a závislostí, pro Plain Old Java Object (POJO), v čj. jednoduchý starý Java objekt. Avšak pokud již je CDI využito, pak je zbytečné označovat objekty pro CDI, protože CDI se o to už samo postará.

Nově se zde vyskytuje JavaBeans Validation. Tato specifikace zařizuje základní validaci prvků s využitím vkládání kontextů a závislostí (CDI). Jedná se o anotace, které se dají přidat k modelu, kde by se však nacházet neměly, protože se jedná o byznys logiku. Další možností je přidat validaci do webové stránky, tedy do prezentační vrstvy. Ani to však není část byznys logiky a proto i zde to není správné.

Listing 2.3: Řešení pomocí JSR 316 - Beans Validation

```
public class Airplane {
    @NotNull
    private String name;

    @NotNull @Size(min=1, max=16)
    private String ownedByCountry;
}
```

Dalším řešením 2.4 je využití specifikace JSR 303 [2], která praktikuje validaci pomocí Meta-programování. To využívá anotování atributů v modelu. To je velmi nešťastné řešení, protože se byznys pravidla nachází v datové vrstvě. Navíc řešení této problematiky pomocí JSR 303 vyžaduje vytvoření velkého množství kódu, který je velmi náchylný na chyby a využívá velké množství anotací.

Listing 2.4: Řešení pomocí JSR 303 - Meta-programování

```
public class Airplane {
    @NotNull
    private String name;

    @NotNull @IsCountry
    private String ownedByCountry;
}
```

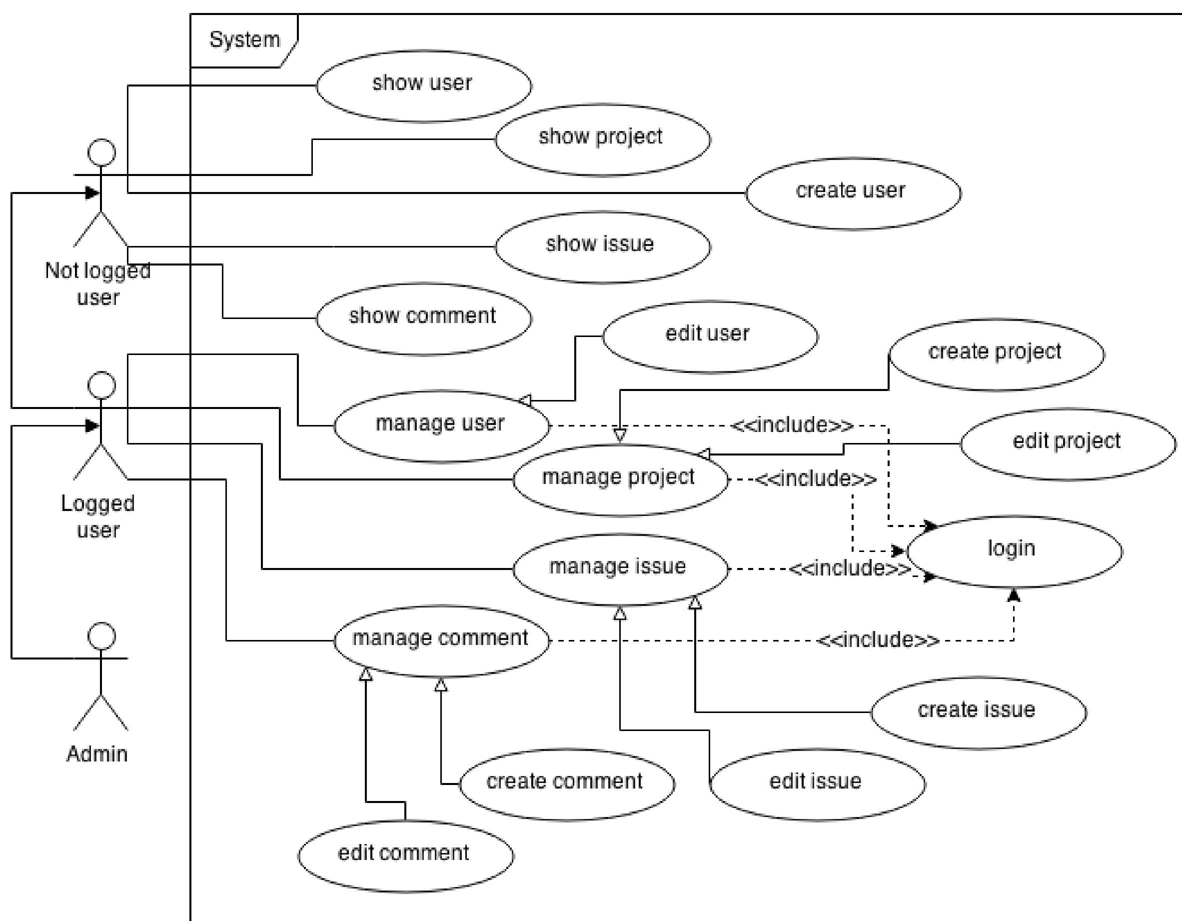
Navíc obě specifikace JSR (jak 316, tak 303) mají ještě jedno společné úskalí. Pokud by se měnily podmínky za běhu aplikace, například vzhledem k 2.4 by byla možnost pro administrátory v situaci, že by neexistovala země, ji takto rovnou přidat. Tyto specifikace fungují výborně, pokud jsou pravidla statická, avšak v jiném případě, tedy že aplikace obsahuje pravidla dynamická, si s nimi tyto specifikace neporadí.

2.2 Řešení v prezentační vrstvě

Aplikace trpí na byznys pravidla nejen v datové vrstvě, ale i v prezentační vrstvě. Toto téma samo o sobě je příliš obsáhlé a nad rámec této práce, proto se o něm zmíním pouze rámcově. Jedná se o problém se zobrazením a kontrolou dat přímo pro uživatele. Uživatel potřebuje vědět jak správně vyplnit formulář, aby měl šanci jej takto vyplnit. Proto je potřeba uživateli dát vidět jak omezení, tak i porušení pravidel, aby mohl na tyto informace reagovat.

2.3 Požadavky

Analýza požadavků zjišťuje jak má systém fungovat dle požadavků klientů, či vlastníků. Požadavky byly dobře stanoveny dle pravidel FURPS (viz. [3] a [4]) společností Hewlett-Packard. FURPS je zkratka funkční požadavky (z aj. functionality), požadavky na uživatelskou použitelnost (z aj. usability), požadavky na spolehlivost (z aj. reliability), požadavky na výkonost (z aj. performance) a požadavky na udržovatelnost (z aj. supportability) (2.1).



Obrázek 2.1: Požadavky popsané pomocí use-case diagram

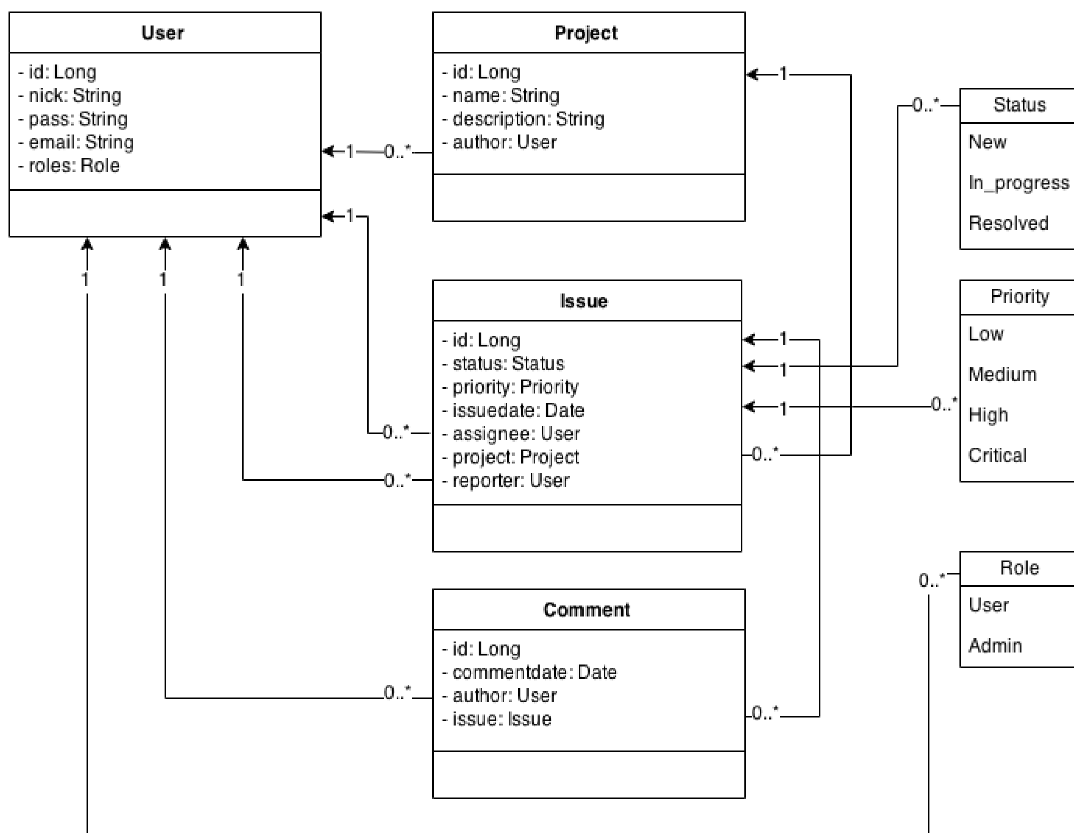
Pro správné navržení aplikace a její zpracování je v tomto případě nejdůležitější analýza funkčních požadavků. Funkční požadavky se dají zpracovat mnoha způsoby, ale asi nejpřehlednější pro diskuzi mezi analytikem a klientem je použití UML diagramu případů užití (Use-case diagramu).

Pokud však nastane chyba při analýze, pak se to promítne do celého systému. Čím později se chyba odhalí, tím náročnější je ji řešit a tím i dražší. Z tohoto důvodu se věnuje velká porce času právě analýze.

2.4 Databáze

Velmi důležité pro další návrh systému je navrhnout databázi. Návrh databáze lze zobrazit jako model, což je v UML diagramech vyobrazeno pomocí Class diagramu, který je známý jako diagram tříd.

V UML diagramu tříd (2.2) jsou zobrazeny jako hlavní prvky třídy modelu. Ve třídách se nachází jejich atributy, které jsou stanoveny jako public, což znamená, že jsou veřejné, značené pomocí znaménka +. Dále jsou zde protected, viditelné podtřídám, tedy třídám, které třídu implementují nebo dědí, značené jsou jako značka #. V poslední řadě se zde nachází private, privátní, či skryté před jinými třídami, značené pomocí znaménka -.



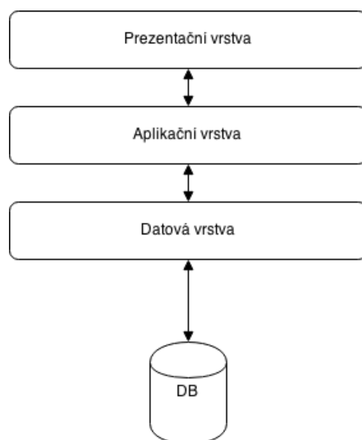
Obrázek 2.2: Třídy popsané pomocí class diagram

Z diagramu je zřejmé, že zde budou 4 třídy, které se budou ukládat do databáze - User, Project, Issue a Comment. Dále se zde nachází množství atributů, které všechny budou skryty a přístup k nim bude pomocí setterů a getterů. Ty jsou jedním z hlavních principů Objektově Orientovaného Programování - mutátory (mutator), známé jako settery, metody, které mění nebo nastavují hodnotu atributu, a accessory (accessor), známé jako gettery, které vrací hodnotu atributu.

2.5 Vrstvy

Ve webových aplikacích se využívají dvě různé architektury [5], které jsou si velmi podobné. MVC, neboli Model-View-Controller a 3-tier architecture, který je rozdělen na

Presentation-Business-Data. V MVC je dosti propojená view vrstva s modelem, jednodušeji lze popsat jako trojúhelníková architektura, naopak 3-tier architecture (2.3) má oddělení jednotlivých vrstev s návazností pouze a jen na ty vedlejší.



Obrázek 2.3: Vrstvy popsané pomocí 3-vrstvá architektura

3 Rešerše

K tomuto tématu se nachází množství textů počínaje knihami, přes odborné články až po různé diplomové práce, ale třeba i statě z odborných konferencí. Zde shrnuji několik důležitých publikací, které se touto tématikou zabývají. Dále zde popisuji a vysvětlím základní pojmy k různým technikám vývoje.

3.1 Minulost a současný stav

V počátcích vývoje webových aplikací se využívalo procedurálního programování. Již dle názvu je jasné, že funguje na principu vytváření procedur, které se skládají z několika pravidel vedoucích k jednomu výsledku. To je velmi podobné dnešnímu neoptimalizovanému vývoji. Tato možnost je využívána programátory-začátečníky pro svou jednoduchost a slušnou čitelnost.

Vylepšenou verzí neoptimalizovaného vývoje je vývoj pomocí objektově orientovaného programování. V knihách [6], [7] se diskutuje vývoj dle objektově orientovaného programování, jak základního, tak i pokročilého pomocí návrhových vzorů.

Základem objektově orientovaného programování je dědičnost, abstrakčnost, mutátory a zapouzdření. Mutátory jsou metody, které vrací, či mění hodnoty nějakých vnitřních proměnných objektů (v aj. settery a gettery). Dědičnost nám umožňuje vytvořit třídu předka, kterou může jiná třída potomka dědit a tím i využít již naprogramovaných metod, či využít proměnných. Abstrakčnost nám navíc přidává možnost vytvořit třídu, která jen naznačuje jak budou vypadat její následníci, kteří ji implementují. Pomocí dědičnosti a abstrakčnosti lze vyřešit spoustu problémů, které by byly těžko řešitelné, například pojem zapouzdření sjednocuje související prvky do jedné společné části. Tím lze minimalizovat duplicity a i zjednodušit případné změny, které ovlivňují celou aplikaci, ale nacházejí se, díky zapouzdření, pouze na jednom místě.

V současné době asi nejčastější variantou vývoje webových aplikací je objektově orientované programování s použitím návrhových vzorů. Dle [7] existují tři skupiny, na které lze programování rozdělit. Creational patterns, neboli vzory k vytváření objektů a tříd. Structural patterns, v překladu vzory struktur, které se soustředí na uspořádání tříd v systému. Poslední jsou Behavioral patterns, které řeší chování systému jak vzhledem ke třídám, tak k objektům. Každý z těchto vzorů řeší nějaký problém, který se tím zjednoduší a zpřehlední.

Méně častým, přesto však stále velmi hojně využívaným řešením aplikační vrstvy je vývoj pomocí Meta-programování. V předešlých technikách se třídy využívaly jako objekty a díky tomu se i vyvolávaly jejich metody velmi jednoduše. Tento styl programování však přebírá metody jako data a dále s nimi podle toho pracuje. Využívá se při tom reflexe, což je schopnost prohlédnout a ovlivnit data za běhu programu.

3.2 Nedostatky současného vývoje

Přes množství výhod, které tyto techniky řešení vývoje webových aplikací zobrazují, se zde nachází i různé nedostatky. Jedním z nich je problém s prolínáním určité vlastnosti naskrz vrstvami. Tento problém patří do skupiny Cross-cutting concerns. Tím, že tyto

techniky nedokáží provést dostatečnou dekompozici, dochází v kódu k duplikacím, které se vyskytují napříč aplikací, nebo ke značnému znepřehlednění kódu se kterým se později velmi špatně pracuje.

Pro názornost zde ukáži problém, který patří do Cross-cutting concerns. Příkladem může být logování (zapisování použití) veškerých metod do konzole, či do souboru.

Při využití Neoptimalizovaného stylu bude v každé metodě jak získání objektu Logger, tak i výpis metody. Avšak toto řešení vytváří velké množství duplicit, protože tento kód se nachází ve všech metodách, které bude potřeba logovat.

Listing 3.1: Řešení pomocí Neoptimalizovaného programování

```
public void method() {
    Logger logger = Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);
    logger.info("Method");
    //code
}
```

V případě, že se využije Objektově Orientovaného Programování se bude opakovat v každé metodě volání statické třídy s metodou loguj. To nám odlehčí kód od duplicit, ale stále v každé z metod bude vždy potřeba tento řádek.

Listing 3.2: Řešení pomocí Objektově Orientované Programování

```
public void method() {
    LoggingService.logInfo("Method");
    //code
}
```

Při využití Meta-programování by pak mohla každá metoda mít nad sebou anotaci, která označí, že se tato třída má logovat. Tato metoda nás oprostí od kódu v metodě a na místo toho přesune kód nad metodu. Ani toto řešení však není dostatečné, ačkoliv velmi zjednodušuje čitelnost.

Listing 3.3: Řešení pomocí Meta-programování

```
@logInfo
public void method() {
    //code
}
```

Dalším velmi dobrým příkladem je řešení chyb, které je opět prolnuté napříč celou aplikací.

3.3 Řešení nedostatků

Dle [8], [9] a [10] tento problém řeší Aspektově Orientované Programování. Nové pojmy pro tuto variantu jsou Aspect, Join point a Pointcut. Join point je místo, kde se střetávají nebo prolínají jedno, či více pravidel ve více třídách, či vrstvách, které spolu nesouvisí. Pointcut je sada několika Join pointů. Tato místa, Join pointy, jsou řešeny Aspekty, které sjednocují pravidla do jednoho celku. Tím se zařizuje oddělení a při tom lepší čitelnost kódu.

V mé práci chci zjistit, zda je vhodné využít této teoretické výhody v podobě Aspektově Orientovaného Programování, či je lepší a sofistikovanější řešení použít jiné metody. Toho dosáhnu použitím objektivního i subjektivního porovnání hodnot.

4 Řešení pravidel v service

Ve světě programování se nalézá množství technik, přesněji přístupů k řešení problémů, jak problém vyřešit a naprogramovat. Touto problematikou se zabývá velké množství knih, odborných článků a pojednání.

V této kapitole je shrnuto několik hlavních principů, výhod a nevýhod, pro jednotlivé techniky. Tato řešení se dají jednoduše rozšířit či kombinovat a tím dosáhnout ještě lepších výsledků jak z hlediska duplicit kódu, tak čitelnosti, která je velmi důležitá pro udržitelnost aplikace při změnách.

4.1 Základní pojmy

Vzhledem k pravidlům a jejich řešení je potřeba určit několik základních pojmů, které lze použít pro porovnávání kvality. Tyto pojmy jsou zmíněny v [6], [7].

Prvním z nich je zapouzdření. V běžném kódu se nachází množství duplicitních prvků, které spolu souvisí, ale jsou velmi často využity vícekrát, mnohdy jsou i napříč systémem v jednotlivých metodách. Toto je klasická situace jíž můžeme vyřešit pomocí zapouzdření. Zapouzdření je součástí skrytí implementace a podstatně zjednodušuje čitelnost kódu tím, že se souvislé prvky zapouzdřují, tedy nějak předávají do společné metody, ze které se poté vyvolávají a zpracovávají. Tím se snižují duplicity v kódu, protože stejný kód není neustále potřeba opakovat. Navíc se i zjednodušuje změna, protože na místo toho, aby se tento kód musel měnit napříč celým systémem, je možnost pouze pozměnit jednu metodu, která tento prvek zpracovává, a tím i ovlivnit všechny její použití.

Veliké důležitosti poté nabývá soudržnost (v aj. Cohesion). Definovaná dle [6] je tak, že žádná samostatná část aplikace nemá mít na starosti několik věcí najednou. Tyto části by měly dělat pouze jeden úkol, úlohu za kterou by byly zodpovědné. Čím lépe se toto pravidlo dodržuje, tím vyšší je soudržnost. Pro zjednodušení se to dá vysvětlit tak, že soudržnost se ptá, zda dělá jeden prvek jednu akci, pro kterou je stvořen. Cílem je mít maximální soudržnost, což znamená, že každý prvek se stará pouze o jednu činnost.

Se soudržností se velmi často využívá i provázanost (v aj. Coupling). Provázanost je definována jako množství využití jiných entit. Správně by vše měla zvládnout sama jedna entita a jen minimimálně se obracet na jiné entity. Čím lépe se toto pravidlo dodržuje, tím je provázanost nižší. Proto tuto vlastnost, provázanost, na rozdíl od soudržnosti, je potřeba minimalizovat. Je samozřejmé, že to nelze dodržet vždy, ale ideální je se tomu co nejvíce přiblížit a minimalizovat vazby mezi entitami.

Nedodržování soudržnosti a provázanosti vede k proplétání a zamotávání pravidel a tím i ke snížení jednoduchosti a tak i čitelnosti kódu.

Pro ukázku byl vybrán kód z metody add ve třídě Comment. Zobrazený kód bude mít kontroly pravidel, dle řešení, ke kterému bude patřit. Cíle je, aby byla ideální soudržnost a také provázanost.

4.2 Neoptimalizované programování

Neoptimalizované programování, také známé pod názvem plain, je dosti podobné programování procedurálnímu. Každá metoda je v podstatě procedura, uvnitř se nachází několik kroků, přes které se dojde k výslednému řešení. Svou jednoduchost nabývá tím, že stačí pouze určení požadavků přes analýzu.

Požadavky se rozdělí do pravidel "buď-nebo"(v aj. "if-else"), které se velmi podobají krokům v již zmíněném procedurálním programování. Takto se vytvoří celý systém.

Toto řešení implementace pravidel má hlavní výhody v tom, že vývoj je velmi snadný a kód jednoduše pochopitelný. Avšak čím více podmínek, tím hůře je kód čitelný, má více duplicitních větví (if-else) a tím je i upravitelný. Nevýhodou je pak to, že se v kódu prolínají business pravidla a oficiální funkce metod. Navíc pokud se jakkoli mění pravidla, tak z důvodu velkého množství větví a tím i duplicit se musí zasahovat na více místech ve třídě, případně i ve více třídách.

Listing 4.1: Ukázka neoptimalizovaného kódu

```
public void add(Comment comment, PrincipalBeanInterface pb)
                    throws ValidationException {
     if (pb.isLogged()) {..}
     if (comment.getIssue() == null) {..}
     if (pb.getId() != comment.getIssue().getAssignee().getId() ||
      pb.getId() != comment.getIssue().getReporter().getId()) {..}
     if (comment.getAuthor() == null) {..}
     if (comment.getName() == null ||
      comment.getName().length() < 4 ||
      comment.getName().length() > 40) {..}
     if (comment.getComment() == null ||
      comment.getComment().length() < 10 ||
      comment.getComment().length() > 200) {..}

     // Pozadavky zkontrolovany, ulozeni komentare
}
```

Výhody

- Jednoduché na tvorbu
- Čitelnost kódu
- Žádné třídy ani vyžadované knihovny navíc

Nevýhody

- Duplicity kódu
- Nízká soudržnost a vysoká provázanost

4.3 Objektově Orientované Programování - zapouzdření

Objektově orientované programování, jak název napovídá, pracuje s objekty, které zpracovává, mění a ukládá. Jedná se o velmi jednoduchý a při tom čistší řešení, oproti neoptimalizovanému programování. Stále využívá podobného řešení jako neoptimalizované programování, avšak odděluje pravidla do speciálních tříd ze kterých je poté vyvolává a takto zpracovává. Velmi se tím zjednodušuje a zpřehledňuje kód, především tím, že se úlohy přesouvají do speciální třídy a jejích metod, které řeší právě tu věc o kterou se mají starat.

Zapouzdření, jak jsem již nastínil, je sjednocení určitých akcí, které spolu souvisí a vytváří stejný výsledek, do jedné metody. Tato metoda se nachází ve specializované třídě na řešení právě jednoho typu problému, příkladem mohou být Business validace, které nepatří do správy objektů, ale jsou potřeba udělat před tím, než se objekt uloží. Tímto se oddělí od sebe úkony, které k sobě nepatří a tak se zajistí vyšší soudržnost. V tomto případě se snižuje i provázanost, protože se delegují pravidla jinam, a tak třída na správu objektů neví jak objekty fungují a i přes to ví, že jsou tato pravidla dodržena.

Dá se ještě využít přetěžování metod, které lze vysvětlit tak, že existuje více metod s jedním názvem, ale různými parametry. Tím lze docílit toho, že kód bude ještě více intuitivní na pochopení, sníží se počet duplicit a tak se i mnohem zjednoduší možnost úprav.

Listing 4.2: Ukázka kódu v OOP zapouzdření

```
public void add(Comment comment, PrincipalBeanInterface pb)
                    throws ValidationException {
    Validator.isLoggedIn(pb);
    BusinessValidator.validate(comment);
    Validator.isOneOfAllowedUsers(comment.getIssue().getAssignee(),
          comment.getIssue().getReporter(), pb);

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Výhody

- Jednoduché na tvorbu
- Méně duplicit v kódu
- Čitelnost kódu
- Vyšší soudržnost a nižší provázanost (oproti neoptimalizovanému programování)

Nevýhody

- Třídy navíc (oproti neoptimalizovanému programování)
- Stále metody zpracovávají spoustu různých činností

4.4 Objektově Orientované Programování - vzory

Při popisování programovacích technik jsem se zmínil, že je jich obrovský počet. Asi největší část zabírají vzory, dle kterých se programuje v objektově orientovaném programování. Krátce shrnu, do jakých kategorií se tyto vzory dle [7] dělí. Existují tři různé kategorie, kde každá zařizuje jinou část pro chod systému. Creational, alias jak vytvářet objekty. Structural jak uspořádat, strukturovat, třídy v aplikaci. Behavioral, které mění chování systému. Programovací vzory, které se v těchto kategoriích nachází se mohou dosti podobat, i naskrz kategoriemi, příkladem je Chain of Responsibility a Decorator.

Vybrány byly dva vzory, Chain of Responsibility a Decorator, které si jsou velmi podobné. Ty byly porovnány a vybrán ten vhodnější pro tuto práci.

Chain of Responsibility, alias řetězení odpovědností, patří do Behavioral patterns a funguje na principu předávání odpovědností. Jakmile objekt dorazí do třídy, která jej dokáže zpracovat, tak se zpracuje a pak se vrací jejich reakce - zde je reakcí chyba validace, či nic.

Decorator, v češtině jak je zřejmé dekorátor, patří do Structural patterns a podobně jako Chain of Responsibility i on rozdává odpovědnosti. Decorator, kterému jsou nastaveny použité dekorace (třídy, které jej dekorují) předává odpovědnost všem těmto třídám.

Chain of Responsibility je vhodnější z důvodu, že pokud nastane chyba, například objekt neprojde nějakým pravidlem, pak lze proces kontrolování zastavit a vyhodit chybu. Proto byl také vybrán jako řešení pomocí návrhových vzorů.

Listing 4.3: Ukázka kódu v OOP pomocí vzoru

```
public void add(Comment comment, PrincipalBeanInterface pb)
                        throws ValidationException {
    BusinessValidator.validate(comment, pb);

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Výhody

- Minimální duplicita kódu
- Čitelnost kódu
- Vyšší soudržnost a nižší provázanost (oproti neoptimalizovanému programování)
- Vysoká soudržnost a nízká provázanost

Nevýhody

- Náročnější na tvorbu (oproti řešení použitím zapouzdření)
- Třídy navíc (oproti řešení použitím zapouzdření)

4.5 Meta-programování

Ještě čistějším řešením je Meta-programování. Meta-programování je možné si představit jako metodu, která zpracovává jiné metody jako data pomocí reflexe. Pro použití to není tak intuitivní a jednoduché jako Objektově Orientované Programování, či neoptimalizované programování, avšak učící křivka tento neduh dosti kompenzuje.

Meta-programování pracuje na principu Proxy a Decorator (z návrhových vzorů), to znamená, že třída, či metoda, je zabalena a při jejím zavolání se nejdříve zavolá její obal, který dále určuje průběh.

V této case study je zpracováno Meta-programování pomocí anotací, které popisují třídu, resp. metodu, a je vyvoláno pomocí interceptoru. To nám zařizuje lepší oddělení kódu, protože se již pravidla nenachází v metodě, ale nad ní. Na druhou stranu využití interceptoru nutí k tomu, aby se využívala reflexe, která více znepřehledňuje kód, může vyvolat chyby navíc a zpomaluje průběh zpracování kódu. Kompilace může být v Runtime, či Deploy-time.

Listing 4.4: Ukázka kódu v Meta-programování

```
@Interceptors(SecurityIntercept.class)
@LoggedIn
@NotNull(object = "getIssue", input = "issue")
@OneOfAllowedUsers(id1Method = "getIssue().getAssignee",
            id2Method = "getIssue().getReporter")
@NotNull(object = "getAuthor", input = "author")
@Length(min = 4, max = 40, param = "getName", input = "Name")
@Length(min = 10,max = 200,param = "getComment",input = "Comment")
public void add(Comment comment, PrincipalBeanInterface pb)
                    throws ValidationException {

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Výhody

- Minimální duplicita kódu
- Velmi vysoká soudržnost a nízká provázanost

Nevýhody

- Čitelnost kódu
- Náročnější na tvorbu (oproti řešení použitím vzorů)
- Třídy navíc (oproti řešení použitím vzorů)
- Potřeba ošetřovat více vyjímek po vyvolávání metod

4.6 Aspektově Orientované Programování

Vývoj je velmi náročný, je potřeba přidat množství tříd navíc, využít množství knihoven navíc a pochopit principy, kterými se tato technika řídí. Podobně jako v Meta-programování je učící křivka velmi strmá a tak lze po naučení využívat veškerých výhod této techniky.

Již zmíněné Cross-cutting concerns, můžeme je nazvat průřezové problémy, jsou akce, které jsou propojené naskrz aplikací, ale nejsou hlavní funkcí aplikace. To pro nás znamená, že nemohou být jednoduše odděleny od hlavní funkce programu.

Místa, která jsou v prostředku Cross-cutting concerns, se nazývají Pointcuty. XCC (Crosscutting Concerns) se shromažďují do aspektů a zde jsou dále zpracovávány.

Máme několik možností jak mohou být zprovozněny. Stejně jako v Meta-programování mohou být Runtime, Deploy-time, či dokonce v Load-time.

Pro pochopení zde popíši Deploy-time. Aby se aspekty mohly zprovoznit je potřeba je speciálně zkompilovat pomocí weaveru, který je zpátky roznese do míst, kam patří.

Ve výsledku, po kompilaci, vypadají dosti podobně neoptimalizovanému programování, avšak přináší výhody výborné čitelnosti kódu (nekompilovaný kód je velmi čitelný) a díky takřka úplnému oddělení pravidel i výhodnější možnosti vývoje aplikace pro více vývojářů.

Listing 4.5: Ukázka kódu v AOP

```
@Validate
@StandardValidation
@RequiredRules( "You must be logged in" )
public void add(Comment comment, PrincipalBeanInterface pb)
                    throws ValidationException {

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Výhody

- Minimální duplicita kódu
- Čitelnost kódu
- Maximální soudržnost a minimální provázanost
- Kompletní oddělení pravidel od zbytku aplikace

Nevýhody

- Náročnější na tvorbu (oproti řešení použitím Meta-programování)
- Spousta tříd a knihoven navíc (oproti řešení použitím Meta-programování)
- Potřeba weaverování (kompilace navíc)

5 Návrh případové studie

Celý návrh je velmi důležitý, protože to je základ celého systému. Návrh systému je dosti komplexní problém a je potřeba mu věnovat velký důraz. Není snadné navrhnout kvalitní systém, protože se zde nachází spousta úskalí. Prvním je analýza požadavků.

Pro tuto bakalářskou práci bylo potřeba vytvořit komplexní případovou studii, ve které se dá vypracovat porovnání dle jednotlivých objektivních metrik a mého subjektivního názoru. Za případovou studii byl zvolen IssueTracking, jenž se skládá z několika částí, které jsou specifické pro webové aplikace. Jedná se o aplikaci, kde je interakce uživatele se systémem, jsou zde různá pravidla, která se musí dodržet a v neposlední řadě je zde množství rolí pro uživatele.

5.1 Základní pojmy

Mezi pojmy, které je potřeba znát při návrhu systému je podle [6] signatura a kontrakt.

Signatura popisuje interface třídy. Interface je třída, která není implementována a jediné co obsahuje jsou public, či protected metody, ve kterých je zobrazeno co mají získávat a co mají vracet. Tyto metody v rozhraní a jejich vlastnosti jsou viditelné z venku, z jiné třídy. V běžných implementacích jsou vidět i public atributy třídy. Pokud je to dokonce třída ve stejném balíčku, pak jsou vidět i atributy a metody protected. Signaturu dokáže překladač zjistit a tak zahlásí chybu syntaxe.

Naopak kontrakt je souhrn zásad, které veřejné nejsou. Často se jedná o omezení jak už velikosti, rozsahu, či dalších různých omezení přípustných hodnot. Velmi jednoduchý příklad je omezení data narození - je jasné, že věk nemůže být záporný, to znamená, že nemůže být stanoveno datum v budoucnosti. Překladač takovéto chyby nedokáže rozpoznat a proto se zobrazují programátorům jako komentáře, aby si jich byli vědomi a mohli s tím počítat.

5.2 Návrh

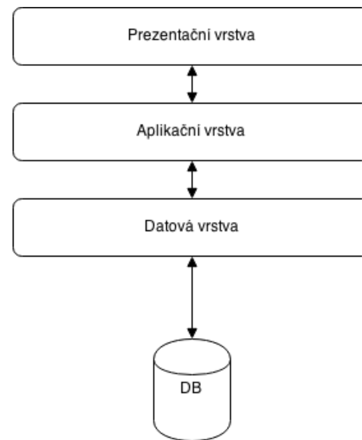
Aplikace je nastavena jako serverové řešení webové aplikace. Dle funkčních požadavků lze začít rozhodovat o návrhu aplikace a jeho provedení. Požadavek je stanovení, co přesně musí systém, v tomto případě aplikace, umět. Návrh, jako dodatek k požadavku, zajišťuje to, jak aplikaci implementovat.

Je potřeba převést všechny požadavky z analýzy požadavků do návrhu tak, že si představíme, jak jednotlivé požadavky implementovat.

5.3 Vrstvy

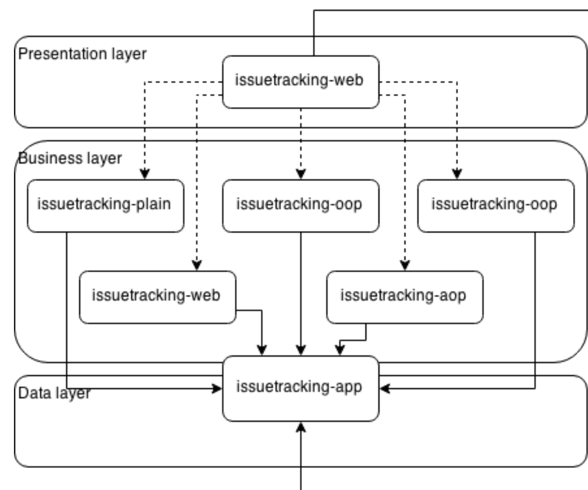
Jak jsem již zmínil ve vývoji webových aplikací se využívají dvě různé architektury [5], které jsou si velmi podobné. MVC, neboli Model-View-Controller pracující na bázi trojúhelníkové architektuře, kdy View může dostat informace přímo od Modelu. Dále pak 3-tier architecture, známý jako třívrstvá architektura, který je rozdělen na Presentation-Business-Data, nebo přeložený jako prezentační-byznys, či aplikační-datová vrstva, který

je rozdělen na patra. Části View a Presentation layer, Controller a Business layer, a Model a Data layer jsou si velmi podobné. Největší rozdíl mezi těmito architekturami je, že třívrstvá architektura má lepší oddělení, protože všechny vrstvy jsou odděleny a komunikují pouze k sousedům. MVC může trpět na to, že Controller nemusí vždy vidět, co Model posílá View vrstvě.



Obrázek 5.1: Vrstvy popsané pomocí 3-vrstvá architektura

Pro lepší zobrazení a oddělení kódu je ideální řešení přes moduly (5.2) , které zpracovávají jednotlivé vrstvy. Tím je umožněno to, že v business vrstvě je jednoduchá možnost výměny některého řešení za jiné a navíc je vývoj jednotlivých modulů nezávislý na ostatních. Proto se dá vyvíjet aplikační vrstva nezávisle na prezentační, neboli webové vrstvě a taktéž na datové vrstvě.



Obrázek 5.2: Vrstvy rozděleny do modulů

6 Implementace případové studie

Po stanovení požadavků, jejich analýze a zpracování do návrhu se přechází na implementaci. Implementace je kompletní zpracování systému do použitelné podoby pomocí programovacích jazyků, využití databázových systémů a poté kompilace.

Díky tří-vrstvé architektuře a rozdělení na moduly je možné implementaci rozdělit na části a tak i aplikaci zpracovávat.

6.1 Datová vrstva

Datová vrstva je zpracována do aplikačního modulu issuetracking-app. Tento modul je děděn všemi moduly, protože obsahuje jak model aplikace, to znamená jednotlivé třídy, které se nachází v databázi, zobrazené jako objekty. Dále tento modul obsahuje pro každou tuto třídu Data Access Object, což jsou třídy, které zprostředkovávají jak přesun objektů, předělaných na data, do databáze, tak výběr dat z databáze, předělaných na objekty.

Důležitou částí datového modulu jsou rozhraní (v aj. interface), která říkají, jak mají třídy, které je implementují, vypadat. Takto jsou zde stanoveny jak třídy z aplikační (business) vrstvy, tak i z prezentační vrstvy. Pro aplikační vrstvu je vytvořeno vždy jedno rozhraní, popisující aplikační logiku, na jednu třídu modelu. Dále je zde generické, což se dá přeložit jako obecně použitelné, nebo také možno nazvat obecné, řešení pro všechny třídy aplikační vrstvy.

Pro prezentační vrstvu je zde rozhraní pro PrincipalBean, která zpracovává data o přihlášeném uživateli. Poslední, avšak velmi důležitá třída, která se nachází vedle aplikačních rozhraní je ValidationException, což je chybové hlášení, kterou aplikace vrátí v případě, že nějaké z pravidel neprojde.

6.2 Aplikační vrstva - Neoptimalizované řešení

Aplikační vrstva v neoptimalizovaném řešení je tvořena pouze čtyřmi třídami, pro každou service třídu z modelu jedna. Pravidla jsou zde sepsána pomocí metody "buď-nebo" (if-else) a jsou vrstvena na sebe. Ačkoliv v kódu není žádné nebo (else) z podmínky, tak se zde vyskytují. Vyzkoušej, zda požadavek neprojde, pokud neprojde, pak vyhoď odpovídající chybu, pokud projde, tak přejdi na další. Nebo (else) je v tomto případě zobrazeno jako pokud projde, protože se takto přechází na další pravidlo a pokud již žádné není, tak zpracování kódu. Tato pravidla se nachází v metodách, které jakkoli zpracovávají a ukládají data vložená uživatelem.

Listing 6.1: Ukázka neoptimalizovaného kódu - přidání komentáře

```
public void add(Comment comment, PrincipalBeanInterface pb)
                throws ValidationException {
    if (pb.isLogged()) {..}
    if (comment.getIssue() == null) {..}
    if (pb.getId() != comment.getIssue().getAssignee().getId() ||
     pb.getId() != comment.getIssue().getReporter().getId()) {..}
```

```
    if (comment.getAuthor() == null) {..}
    if (comment.getName() == null ||
     comment.getName().length() < 4 ||
     comment.getName().length() > 40) {..}
    if (comment.getComment() == null ||
     comment.getComment().length() < 10 ||
     comment.getComment().length() > 200) {..}
    // Pozadavky zkontrolovany, ulozeni komentare
}
```

6.3 Aplikační vrstva - Objektové programování - zapouzdření

Aplikační vrstva v optimalizovaném řešení pomocí Objektově Orientovaného Programování a použití základů ve formě zapouzdření vytváří třídu navíc. Toto řešení je velmi podobné neoptimalizovanému, pouze s tím rozdílem, že místo nutnosti opakovat kód, který se opakoval stačí využít třídy Validátor, ve které metody provádějí požadovanou kontrolu pravidel.

Listing 6.2: Ukázka kódu v OOP zapouzdření - přidání komentáře

```
public void add(Comment comment, PrincipalBeanInterface pb)
                 throws ValidationException {
    Validator.isLoggedIn(pb);
    BusinessValidator.validate(comment);
    Validator.isOneOfAllowedUsers(comment.getIssue().getAssignee(),
          comment.getIssue().getReporter(), pb);

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Kontrola je pak prováděna tak, že se ještě v metodě volá několik kontrol. První kontroluje, zda je uživatel přihlašen, případně nepřihlášen, další je business (aplikační) validátor, který kontroluje správnost objektu, v tomto případě komentáře. Poslední je kontrola, zde se jedná o jednoho z uživatelů, kteří mohou v tomto případě přidávat komentář.

Listing 6.3: Ukázka kódu v OOP zapouzdření - přidání komentáře

```
public class BusinessValidator {
    ..

    public static void validate(Comment comment)
                 throws ValidationException {
        Validator.isNotNull(comment.getIssue(), "issue");
        Validator.isNotNull(comment.getAuthor(), "author");
        Validator.isBetween(comment.getName(), 4, 40, "Name");
        Validator.isBetween(comment.getComment(), 10, 200, "Comment");
    }
}
```

Tím se jednoduše oddělí opakované části a tak i zjednoduší opravy. Opravy jsou zjednodušeny především pro objekty, které patří do modelu a mají svou vlastní validační metodu v business (aplikačním) validátoru. Kontrola, zda je uživatel přihlášen, nebo zda je uživatel oprávněn přidávat komentář, je stále v metodě, aby zůstalo ověřování odděleno od kontroly objektu, které se může lišit.

Podobně jako v [11] je i zde problém jak rozdělit pravidla, protože i zde se nachází odlišná chyba pro každý typ porovnání pravidla. Pokud necháme pouze podmínky, pak se nachází v každé práci velký "buď-nebo" blok. V citované práci je řešení případové studie pomocí návrhového vzoru visitor, tedy využití vyvolání chyby pro návštěvu metody, za pomoci reflexe, aby se chyba správně ošetřila. Toto řešení je centralizované a velmi se podobá řešení z Meta-programování, které využívá na místo návrhového vzoru visitor návrhový vzor proxy, tedy bránu před vstupem do třídy, respektive metody. Řešení, které je dosti podobné tomuto řešení, tedy Objektovému programování s vyžitím zapouzdření, dává všechny aplikační pravidla do jednoho místa. V takovém případě však stále protíná jedno pravidlo skrz jednotlivé metody, což znamená, že je potřeba další dekompozice. V tomto mém řešení, pomocí Objektovému programování s vyžitím zapouzdření, je tento problém vyřešen ponecháním problémových pravidel v metodě.

6.4 Aplikační vrstva - Objektové programování - Chain of Responsibility

Aplikační vrstva v optimalizovaném řešení pomocí Objektově Orientovaného Programování s použitím návrhových vzorů, přesněji návrhový vzor Chain of Responsibility. Toto řešení již odděluje veškerá pravidla do speciální třídy, která se stará o business (aplikační) pravidla. Takto poté není potřeba vůbec ovlivňovat nic navíc a stačí tedy, že je v základní třídě obecná validace.

Listing 6.4: Ukázka kódu v OOP pomocí vzoru - přidání komentáře

```
public void add(Comment comment, PrincipalBeanInterface pb)
                throws ValidationException {
    BusinessValidator.validate(comment, pb);

    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Kontrola je pak prováděna tak, že se zjistí o jaký objekt se jedná a dle toho se validuje.

Listing 6.5: Ukázka kódu v OOP pomocí vzoru - zavolaný aplikační validátor

```
public class BusinessValidator {
    public static void validate(Object obj,
          PrincipalBeanInterface pb) throws ValidationException {
        ..
        if (Comment.class.isInstance(obj)){
            validateComment((Comment) obj, pb);
        }
    }
```

```
    private static void validateComment(Comment comment,
            PrincipalBeanInterface pb) throws ValidationException {
        ObjectValidator.validate(comment);
        Validator.isLoggedIn(pb);
        Validator.isOneOfAllowedUsers
            (comment.getIssue().getAssignee(),
            comment.getIssue().getReporter(), pb);
    }

    ..

}
```

Takto se otevře nová validační metoda, která sama zkontroluje, zda fungují veškerá pravidla, například kontrola přihlášení uživatele, nebo zda je text v rozsahu od 10 do 400 znaků. Rozdíl mezi tímto vzorem a zapouzdřením je, že zapouzdření nemá plošnou kontrolu, ale pouze kontrolu modelového objektu. Se vzorem lze ovlivnit i další části jednoduše a především plošně, například pro přihlášení.

6.5 Aplikační vrstva - Meta-programování

Aplikační vrstva pomocí Meta-programování zajišťuje ještě kvalitnější oddělení. Toto řešení je odděleno od metody (viz. 6.6) tak, že je vloženo před metodu pomocí anotací. Každé pravidlo je nastaveno jako jedna anotace a poté se dle těchto anotací kontroluje, zda jsou pravidla splněna.

Anotace (viz. 6.7) jsou velmi podobné rozhraní, dokonce jsou i anotované jako rozhraní (@interface). Rozdílem je to, že se zde nastavují základní hodnoty atributů pomocí metody default, za kterou se poté vloží základní hodnota. Nastavují se k nim ještě anotace Target a Retention. Target říká, k jaké části kódu se dá anotace připojit, Retention říká, kdy se anotace zpracovává. Ještě se zde může vyskytovat anotace Repeatable. Tato anotace určuje jakou třídu použít v případě, že nad jednou částí kódu se jedna anotace opakuje víckrát.

Listing 6.6: Ukázka kódu v Meta-programování - přidání komentáře

```
@Interceptors(SecurityIntercept.class)
@LoggedIn
@NotNull(object = "getIssue", input = "issue")
@OneOfAllowedUsers(id1Method = "getIssue().getAssignee",
            id2Method = "getIssue().getReporter")
@NotNull(object = "getAuthor", input = "author")
@Length(min = 4, max = 40, param1 = "getName", input = "Name")
@Length(min = 10, max = 200, param1 = "getComment", input = "Comment")
public void add(Comment comment, PrincipalBeanInterface pb)
                throws ValidationException {
    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Listing 6.7: Ukázka kódu v Meta-programování - anotace NotNull

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(value = NotNulls.class)
public @interface NotNull {

    String object() default "";

    String input() default "";
}
```

Aby se mohla kontrola pravidel provést je potřeba ještě vytvořit třídu (viz. 6.8), která bude předskakovat před kód, který je označen anotací Interceptors. Již podle názvu anotace se jedná o Interceptory, v překladu odchytávače, protože zachytávají volání tříd, či metod, případně atributů, atd., které díky anotaci Around Invoke se vyvolají okolo kódu, který je anotován anotací Interceptors. I zde je potřeba dbát zvýšené pozornosti, protože přehlednost není dominantou tohoto kódu.

Listing 6.8: Ukázka kódu v Meta-programování - odchytávání kódu pomocí Interceptoru

```
public class SecurityIntercept {
    @AroundInvoke
    private Object doSecurityCheck(InvocationContext context)
                throws ValidationException {
        Method m = context.getMethod();
        for (Annotation ann : m.getAnnotations()) {
            parseAnnotation(ann, context);
        }
        return context.proceed();
    }

    private void parseAnnotation(Annotation ann,
            InvocationContext context) throws ValidationException {
        // vyber validatoru dle anotace
    }
}
```

Validátory porovnávají dle anotací, zda jsou požadavky splněny. Pro splnění tohoto pravidla je však občas potřeba vyvolat metodu, která vrací požadovaný objekt, potřebný k porovnání. Toho se využívá z Reflexe.

Takový kód je zjevně dosti nečitelný a složitý na pochopení. Další nevýhodou je to, že je potřeba odchytit několik druhů chyb, protože není rozumné zachytit veškeré chyby které mohou nastat. Takto by se mohlo stát, že v kódu bude například NullPointerException a ten jen tak nebude souviset s tím, že se právě vyvolává nějaká metoda.

Dalším úskalím je, že je nutnost věřit anotacím, které ne vždy jsou správně definovány. Pokud nastane případ, kdy je chyba v anotaci, tak se jen velmi složitě odhalí, že tato chybá práve pochází z anotace a není způsobena, jiným ať už špatným návrhem, či třeba jen překlepem.

Listing 6.9: Ukázka kódu v Meta-programování - validátor pravidla NotNull

```
public class NotNullValidator {
    public static void validate(InvocationContext context,
            NotNull ann) throws ValidationException {
        Object[] params = context.getParameters();
        Object obj = params[0];
        if (ann.object() == null && ann.object().isEmpty()) {
            try {
                Method m = obj.getClass().getMethod(ann.object());
                obj = m.invoke(obj);
            } catch (Exception ex) {
                ..
            }
        }
        if (obj == null) {
            throw new ValidationException(..);
        }
    }
}
```

6.6 Aplikační vrstva - Aspektově Orientované Programování

Pro Aspektově Orientované Programování dle [8], [9], [10] a [12] existuje množství přístupů k řešení, od metod, které dokáží předskakovat metody bez označování anotacemi, protože o to se postará kompilátor aspektů. Další možností je například vytváření pravidel, která popisují jak jednat v určitých případech při zavolání metod, které jsou označeny anotacemi. Takto pracují například Drools [13], které fungují podobně jako meta-programování na předskakování před anotované metody. Výhodou oproti meta-programování je to, že zde není potřeba reflexe, ale pouze pre-kompilace, která roznese správná pravidla do míst, kde jsou potřeba a po kompilaci pak kód není příliš odlišný například od neoptimalizovaného přístupu.

Listing 6.10: Ukázka kódu v AOP - přidání komentáře

```
@Validate
@StandardValidation
@RequiredRules( "You must be logged in" )
public void add(Comment comment, PrincipalBeanInterface pb)
                throws ValidationException {
    // Pozadavky zkontrolovany, ulozeni komentare
}
```

Toto řešení 6.10 je odděleno od metody, podobně jako meta-programování, pomocí anotací. Každé pravidlo je nastaveno jako jedna anotace a poté se dle těchto anotací kontroluje, zda jsou pravidla splněna. Anotace Validate říká, že tato metoda má být validována, anotace Standard Validation říká, že se mají použít validace určené pro určenou modelovou třídu, zde se tedy jedná o comment. Required Rules anotace, také anotace

požadovaných pravidel, říká, že se má navíc validovat dle pravidla s tímto názvem. Anotace Required Rules může obsahovat i větší počet těchto anotací a to pokud vytvoříme pole v anotaci požadovaných pravidel s názvy pravidel k porovnání. V tomto frameworku dále existují pravidla, která nesmí platit, ale v této práci nejsou využity.

Pro využití kontextu v pravidlech je potřeba vytvořit takzvané Hooks, což jsou v překladu háky, pro odchycení kontextu. Kontext běžně není dosažitelný v pravidlech a takto jej lze zpřístupnit.

Listing 6.11: Ukázka kódu v AOP - vkládání kontextu pomocí Hook

```
@Hook( name = "principalHook" )
public class PrincipalHook implements ParameterHook {
    @Inject
    private PrincipalBeanInterface context;

    @Override
    public Map<String, Object> get() {
        return Collections.singletonMap("pb", (Object) context);
    }
}
```

Pravidla využívají DSL [14], což je Domain Specific Language, známý jako Doménově Specifický Jazyk. Dialect je mvel expression language, doslovně výrazový jazyk mvel. Ten je velmi inspirovaný jazykem Java a jeho syntaxemi. Pravidlo, v aj. rule, má v uvozovkách název, poté na místo "když-potom", tedy v Javě if-{} je zde when-then. Navíc se zde nevyužívá kulatých závorek pro pravidla, ale odřádkování. Pravidlo je ukončeno slovem konec, tedy end. Takto se může opakovat několik pravidel za sebou pro jednu modelovou třídu.

Listing 6.12: Ukázka kódu v AOP - vytvoření pravidel pomocí Drools

```
package org.issuetracking.model;

global java.util.Set<Object> principal;

dialect "mvel"

rule "You have to select issue"
    when
        Comment( issue != null )
    then
end
```

6.7 Prezentační vrstva

Prezentační vrstva, tvořená modulem issuetracking-web se skládá z několika jazyků, čímž využívá jejich výhod. Využívá pro zobrazení stránek xhtml, aby jednoduše zobrazilo stránky a při tom se dalo využít i pokročilejších nástrojů, od Java Server Pages (JSP),

přes Java Server Faces (JSF) až po stylování pomocí Cascading Style Sheets (CSS). Dále využívá Java Platform, Enterprise Edition (Java EE) a Java Servlet.

Prezentační vrstva využívá závislostí na modul issutracking-app, což je datový modul, ve kterém se nachází jak třídy modelu, tak Database Access Object (DAO), alias objekt pro databázový přístup, tak i rozhraní jak pro aplikační vrstvu, tak i pro prezentační vrstvu. Navíc využívá i závislosti na aplikační vrstvě, zde však může být vložen jakýkoli aplikační modul, díky tomu, že aplikační vrstva je pevně stanovena právě pomocí rozhraní z datové vrstvy a proto může být nastavena prezentační vrstva tak, aby dokázala pracovat s jakoukoli třídou, která tato rozhraní respektuje.

Implementace Java EE, JavaBean, což jsou třídy, které využívají zmíněného CDI, tedy kontextového a závislostního vkládání, je vidět díky anotaci Names ve webových stránkách. Díky těmto třídám a converterům, které zařizují zobrazení objektů jako text na stránce a vrácení objektu zpět z textu ze stránky do aplikace, je možné zobrazit a efektivně vytvářet, či editovat, jednotlivé modelové objekty. Každá Bean, která se stará o modelovou třídu navíc musí ošetřovat chybové hlášení, které je vytvořeno v datové vrstvě. Speciální Bean třídou je PrincipalBean, která využívá Faces Context pro získání informací o přihlášeném uživateli a uložení uživatelských informací.

7 Měření

Měření kódu je pro tuto bakalářskou práci esenciální, ale běžné projekty tuto část nevyžadují, dokonce ve většině projektů by se jednalo o věc navíc. Měřit kvality kódu je dosti složité, protože takřka nikdy nelze přesně určit jaký kód je lepší (až na výjimky). Dokonce je velmi složité dostatečně kvalitně určit, zda kód je vhodný pro řešení určitého aplikačního problému. Většina měření vypovídá o určitých kvalitách kódu, ale vzhledem k tomu, že aplikace má množství parametrů, dle kterých jde určovat kvalitu kódu, zde často není jednoznačný vítěz.

7.1 Měření řádků kódu

Měření počtu veškerých řádků kódu, známých pod zkratkou SLOC, alias Source Lines Of Code, je měření, dle kterého se předpovídá náročnost pro programování kódu a jeho udržitelnost. Jak již název napovídá, tak se pro každou metodu vypočítá počet veškerých řádek kódu. Počet řádek není pro programátory hlavní atribut, i přes to je to důležitý atribut, který objektivně ukazuje rozdíly mezi kódy.

Čím více řádek kódu, tím je pravděpodobnější pracnost vývoje a složitější údržba. Navíc čím více řádek na chybu, tím je jednodušší udělat chybu. Ještě k tomu čím obsáhlejší je kód, tedy čím větší je počet řádek kódu, tím složitější je pochopení a přehlednost. Avšak ne vždy je to podmínkou, protože by bylo možné snížit počet řádek kódu například tím, že podmínky (if) by byly jednořádkové na místo rozdělení na podmínku, řešení a poté pouze závorka.

Měření počtu znovupoužitelných řádků kódu sděluje kolik řádků kódu je možno použít znovu. Lze to definovat přesněji jako kód, ze kterého můžeme dále vycházet při vytváření další implementace, či při vytváření dalších aplikačních pravidel.

	řádky kódu		
	všechny	znovupoužitelné	procent
Neoptimalizované programování	435	0	0
Programování objektové - zapouzdření	377	67	17.8
Programování objektové - vzor	433	191	44.1
Meta-programování	889	476	53.5
Aspektově Orientované Programování	618	294	47.6

Znovupoužitelný kód je ten, se kterým můžeme dále pracovat. Tento fakt většinou dosti napomáhá práci s kódem a tím se zlepšuje přehlednost, snižuje náročnost vývoje i údržby. Navíc se většinou při použití znovupoužitelného kódu markantně snižuje náchylnost na chyby.

7.2 Měření výskytu stejného pravidla

Měření výskytu stejného pravidla, dle jeho sémantické definice, lze chápat jako počet implementací pravidel a jejich maximální a průměrný výskyt.

Tato tabulka ukazuje na to, kolikrát se definice pravidla objevuje v kódu (ne to, kolikrát je volána). Vzhledem k pracnosti vývoje a údržby je toto měření velmi směrodatné. Zde

je totiž vidět kolik práce je potřeba udělat navíc, aby se zpracovalo jedno pravidlo. Pokud by bylo potřeba přidat již existující pravidlo, tak ve všech implementacích, kromě té neoptimalizované, stačí přidat jediný řádek.V neoptimalizovaném řešení je potřeba přidat celé pravidlo.

	opakování	
	maximálně	průměrně
Neoptimalizované programování	37	14.2
Programování objektové - zapouzdření	1	1
Programování objektové - vzor	1	1
Meta-programování	1	1
Aspektově Orientované Programování	1	1

V případě, že by bylo třeba vytvořit nové pravidlo, pak by bylo potřeba ve všech vytvořit toto pravidlo tak, že dle všech implementací ve všech metodách, které by toto pravidlo ovlivňovalo, by byl opět řádek odkazující na řešení, buď pomocí anotace, či volání statické metody z třídy validátoru. Pokud by se jednalo o standardní validaci, pak v Aspektově Orientovaném Programování by nebylo dále potřeba přidávat nic, případně v neoptimalizovaném programování přímo řešení. Nově vytvořená řešení pravidla by se vyskytovala v některém z validátorů jako jedna z metod, či v drl souboru a pro Meta-programování by to byla jak nová anotace, tak i nový validátor.

7.3 Měření cyklomatické složitosti business metod

Měření cyklomatické složitosti, také známé pod podmínkovou složitostí, vyjadřuje jakou komplexitu má business metoda, neboli kolik cest skrz graf zdrojového kódu existuje. Složitost metody se dá počítat jako graf toku programem, pro tento případ tedy se vzorcem $M = E - N + 2P$. V tomto vzorci jsou stanoveny proměnné jako celá čísla. Proměnnou E získáme nasčítáním všech hran grafu průchodu, proměnnou N získáme nasčítáním uzlů grafu průchodu a proměnnou P je počet oboustranných hran. Jiným vzorcem, který počítá cyklomatickou složitost je $M = D - S + 2$, kde proměnná D je počet bodů, kde se rozhoduje do jakého uzlu v grafu jít a proměnná S je počet koncových uzlů. Vzhledem k tomu, že metody této case study mají více koncových uzlů, tak je zvolen druhý vzorec.

	cyklomatická složitost	
	maximálně	průměrně
Neoptimalizované programování	7	4.17
Programování objektové - zapouzdření	7	4.17
Programování objektové - vzor	11	8.42
Meta-programování	20	13.34
Aspektově Orientované Programování	7	4.17

Podmínková složitost ukazuje na to, že pracnost vývoje, ale i údržby Neoptimalizovaného programování, Objektového programování použitím zapouzdření a Aspektově Orientované Programování není tak obrovská ve srovnání s Objektovým progra-

mováním s využitím vzoru nebo dokonce s Meta-programováním, které jednoznačně vychází nejhůře. S tolika podmínkami se zvyšuje i chybovost a přehlednost.

Meta-programování nejvíce ztrácí v případě vyvolání anotace, protože zde je možnost až 8 různých výsledků pro každou anotaci a tak se vrství podmínka na podmínku. V Objektovém programování s využítím vzoru zřetězení závislotí byla chyba v implementaci přes generickou třídu na místo přetížení metody. Tímto problémem se pro každou metodu vytvoří o 4 podmínky navíc.

8 Závěr

Ačkoliv vývoj webových aplikací nadále pokračuje a vyvíjí se nové a nové technologie, tak ne každá nová technologie přináší něco nového, krok kupředu, který by zjednodušil, ulehčil, zpřehlednil a zvýšil spolehlivost při práci s webovými systémy.

Cílem této bakalářské práce bylo práci, ať již vývoj, tak údržbu, zjednodušit, zlevnit a zvýšit spolehlivost. Pokud by bylo možné oddělit nesouvisející části kódu od sebe, tím zvýšit soudržnost a snížit provázanost.

Porovnáním bylo zjištěno, že nejlépe, vzhledem k řešení aplikační vrstvy, vychází Aspektově Orientované programování. Z objektivních porovnání bylo zjištěno, že ačkoliv má, hned po Meta-programování (889), nejvíce řádků kódu (618), což není tak příznivé, ale ani nijak alarmující, tak má i velmi vysoké procento (48%), dokonce opět po Meta-programování nejvyšší (54%), kódu znovupoužitelné. Vychází dobře i dle měření výskytu stejného pravidla (1), protože sémanticky je určeno ve všech řešeních, kromě neoptimalizovaného, pouze jednou. Navíc si udržuje cyklomatickou složitost jako programování Neoptimalizované a Objektově Orientované Programování pomocí zapouzdžení (průměrně 4.17).

Velmi dobře vyšlo aktuálně velmi používáné Objektově Orientované Programování s využitím vzoru Chain of Responsibility. Jednalo se o druhý nejnižší počet řádek kódu (433), třetí nejvyšší procento znovupoužitelného kódu (44%) a jak již bylo zjištěno výskyt pravidla byl i zde kvalitní (1). Nevýhodou byla cyklomatická složitost (8.42), kde jsem se dopustil omylu v návrhu a na místo přetížení metody jsem vytvořil generickou třídu. I tak by však toto řešení mělo podmínky navíc, ale nevyčnívalo by negativně tolik nad ostatními.

Subjektivním porovnáním bylo zjištěno, že vývoj pomocí Aspektově Orientovaného Programování je velmi náročný a je potřeba udělat množství kroků pro zpracování tohoto řešení, ale poté při údržbě má naprosto nejlepší výsledky, protože změnu lze provést odděleně a řešení je velmi čisté a tím i čitelné. Proto se tímto přístupem dá dojít k nejjednodušímu řešení, nejvyšší spolehlivosti a tak i zlevnit údržbu aplikací.

Je potřebné stále pokračovat ve výzkumu nových technologií a tím i ulehčovat možnost práce s nimi. Některá řešení, která jsou implementována, mají velmi kvalitní základ, ale tím to bohužel končí. Příkladem může být validace přinesená v JSR 316 ([1]). Naopak některá řešení sráží ne příliš dobrý popis jak s nimi pracovat a tak se i zvyšuje penzum práce, kterou musí programátor udělat. Příkladem může být Aspektově Orientované Programování. Ačkoliv je to výborné a jednoduché řešení, tak pochopit princip a naučit se s ním správně pracovat není intuitivní.

Pokud mám zhodnotit aktuální stav aplikace, pak je potřeba porovnat požadavky (2.1) s implementovanou funkcionalitou. Požadavky které byly zadány byly zprovozněny pro všechna řešení a každé z řešení pokytuje funkční variantu. Webová aplikace, která se skládá z třívrstvé aplikace má zprovozněny všechny vrstvy a tak lze libovolně zkoušet funkcionalitu aplikace. Jako další vylepšení by bylo možné závest vnitřní testování aplikace pomocí Unit testů.

Díky této práci je možné se soustředit na chyby v aktuálních návrzích řešení a zpracovat je lépe, protože pokud "není známá chyba, nelze ji opravit".

9 Literatura

[1] R. Chinnici and B. Shannon, "Jsr 316: Java platform, enterprise edition (java ee) specification, v6," 2009.

[2] E. Bernard and S. Peterson, "Jsr 303: Bean validation," *Bean Validation Expert Group, March*, 2009.

[3] P. Eeles, "Capturing architectural requirements," *IBM Rational developer works*, 2005.

[4] C. Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3/e*. Pearson Education India, 2005.

[5] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] R. Pecinovskỳ, *Návrhové vzory*, vol. 528. Computer press, 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[8] T. Cerny, M. J. Donahoo, and E. Song, "Towards effective adaptive user interfaces design," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pp. 373–380, ACM, 2013.

[9] K. Cemus and T. Cerny, "Aspect-driven design of information systems," in *SOFSEM 2014: Theory and Practice of Computer Science*, pp. 174–186, Springer, 2014.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97—Object-oriented programming*, pp. 220–242, Springer, 1997.

[11] T. Cerny and M. J. Donahoo, "How to reduce costs of business logic maintenance," in *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, vol. 1, pp. 77–82, IEEE, 2011.

[12] R. Laddad, *Aspectj in action: enterprise AOP with spring applications*. Manning Publications Co., 2009.

[13] P. Browne, *JBoss Drools Business Rules*. Packt Publishing Ltd, 2009.

[14] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.