

Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

**Centrální správa a automatická integrace byznys pravidel v
architektuře orientované na služby**

Bc. Filip Klimeš

Vedoucí práce: Ing. Karel Čemus

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

10. dubna 2018

Poděkování

Zde můžete napsat své poděkování, pokud chcete a máte komu děkovat.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 20. 5. 2018

.....

Abstract

Translation of Czech abstract into English.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její obsah. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.

Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Obsah

1	Úvod	1
2	Analýza	3
2.1	Byznys pravidla	3
2.2	Architektura orientovaná na služby	3
2.3	Problémy	3
2.4	Identifikace požadavků na implementaci frameworku	3
2.5	Shrnutí	3
3	Rešerše	5
3.1	Architektura orientovaná na služby	5
3.2	Modelem řízená architektura	5
3.3	Aspektově orientované programování	5
3.4	Aspect-driven Design Approach	6
3.5	Stávající řešení reprezentace byznys pravidel	6
3.5.1	Drools DSL	6
4	Návrh	7
4.1	Formalizace architektury orientované na služby	7
4.1.1	Join-points	7
4.1.2	Advices	7
4.1.3	Pointcuts	7
4.1.4	Weaving	7
4.2	Architektura frameworku	7
4.3	Metamodel	7
4.4	Expression	7
4.5	Registr byznys kontextů	7
4.6	Byznys kontext weaver	7
4.7	Centrální správa byznys kontextů	7

5	Implementace prototypů knihoven	9
5.1	Výběr použitých platforem	9
5.2	Sdílení byznys kontextů mezi službami	10
5.2.1	Síťová komunikace	10
5.2.1.1	Protocol Buffers	10
5.2.1.2	gRPC	10
5.3	Knihovna pro platformu Java	12
5.3.1	Použité technologie	12
5.3.1.1	Apache Maven	12
5.3.1.2	AspectJ	12
5.4	Knihovna pro platformu Python	13
5.5	Knihovna pro platformu Node.js	13
5.6	Doménově specifický jazyk pro popis byznys kontextů	14
5.7	Systém pro centrální správu byznys pravidel	15
5.7.1	Použité technologie	16
5.7.2	Detekce a prevence potenciálních problémů	16
5.8	Shrnutí	17
6	Verifikace a validace	19
6.1	Testování prototypů knihoven	19
6.1.1	Platforma Java	19
6.1.2	Platforma Python	20
6.1.3	Platforma Node.js	20
6.2	Případová studie: e-commerce systém	20
6.2.1	Model systému	20
6.2.2	Use-cases	20
6.2.3	Byznys kontexty	20
6.2.4	Service discovery	20
6.2.5	Order service	20
6.2.6	Product service	20
6.2.7	User service	20
6.2.8	Nasazení systému pro centrální správu byznys kontextů	20
6.3	Shrnutí	20
7	Závěr	21
7.1	Analýza dopadu použití frameworku	21
7.2	Budoucí rozšiřitelnost frameworku	21
7.3	Možností uplatnění navrženého frameworku	21
7.4	Další možnosti uplatnění AOP v SOA	21

7.5 Shrnutí	21
A TODO Screenshots	25
B Seznam použitých zkratk	27
C Obsah přiloženého CD	29

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Kapitola 2

Analýza

2.1 Byznys pravidla

2.2 Architektura orientovaná na služby

2.3 Problémy

2.4 Identifikace požadavků na implementaci frameworku

2.5 Shrnutí

[T1]fontenc [utf8]inputenc

Kapitola 3

Rešerše

3.1 Architektura orientovaná na služby

3.2 Modelem řízená architektura

3.3 Aspektově orientované programování

Programování je komplexní disciplína s teoreticky neomezeným počtem možností, jakým programátor může řešit zadaný problém. Ačkoliv každá úloha má své specifické požadavky, za relativně krátkou historii programování se stihlo ustálit několik ideologií, tzv. programovacích paradigmat, které programátorovi poskytují sadu abstrakcí a základních principů [9]. Díky znalosti paradigmatu může programátor nejen zlepšit svou produktivitu, ale zároveň může snáze pochopit myšlenky jiného programátora a tím zlepšit kvalitu týmové spolupráce.

Jedním z nejpopulárnějších paradigmat používaných k vývoji moderních enterprise systémů je nepochybně objektově orientované programování (OOP). To vnímá daný problém jako množinu objektu, které spolu intereagují. Program člení na malé funkční celky odpovídající struktuře reálného světa [7]. Je vhodné zmínit, že objekty se rozumí jak konkrétní koncepty, například auto nebo člověk, tak i abstraktní koncepty, namátkou bankovní transakce nebo objednávka v obchodě. Objekty se pak promítají do kódu programu i do reprezentace struktur v paměti počítače. Tento přístup je velmi snadný pro pochopení, vede k lepšímu návrhu a organizaci programu a snižuje tak náklady na jeho vývoj a údržbu.

Ačkoliv je OOP velmi silným a všestranným nástrojem, existují problémy, které nelze jeho pomocí efektivně řešit. Jedním takovým problémem jsou obecné požadavky na systém, které musejí být konzistentně dodržovány na více místech, které spolu zdánlivě nesouvisí. Příkladem může být logování systémových akcí, optimalizace správy paměti nebo uniformní zpracování výjimek [4]. Takové požadavky nazýváme *cross-cutting concerns*. V rámci OOP

je programátor nucen v objektech manuálně opakovat kód, který zodpovídá za jejich realizaci. Duplikace kódu vede k větší náchylnosti na lidskou chybu a k vyšším nárokům na vývoj a údržbu daného softwarového systému [2].

Aspektově orientované programování (AOP) přináší řešení na výše zmiňované problémy. Extrahuje obecné požadavky, tzv. *aspekty* do jednoho místa a pomocí procesu zvaného *weaving* je poté automaticky distribuuje do systému. Weaving může proběhnout staticky při kompilaci programu nebo dynamicky při jeho běhu. V obou případech ale programátorovi ulehčuje práci, protože k definici i změně aspektu dochází centrálně a tím je eliminována potřeba manuální duplikace kódu. Je nutno poznamenat, že AOP není paradigmatickým poskytováním kompletní framework pro návrh programu. V ideálním případě je tedy k návrhu systému využita kombinace AOP s jiným paradigmatickým. Pro účely této práce se zaměříme na kombinaci AOP a OOP.

Aspekt

Join-point

Pointcut

Advice

Weaving

3.4 Aspect-driven Design Approach

Aspect-driven Design Approach (ADDA)

Vzhledem k požadavkům na implementaci našeho frameworku stanoveným v předchozí kapitole 2 se AOP a na něm stavějící ADDA jeví jako vhodný přístup, který nám pomůže dosáhnout cíle.

3.5 Stávající řešení reprezentace byznys pravidel

3.5.1 Drools DSL

```
[T1]fontenc [utf8]inputenc
```


Kapitola 4

Návrh

4.1 Formalizace architektury orientované na služby

4.1.1 Join-points

4.1.2 Advices

4.1.3 Pointcuts

4.1.4 Weaving

4.2 Architektura frameworku

4.3 Metamodel

4.4 Expression

4.5 Registr byznys kontextů

4.6 Byznys kontext weaver

4.7 Centrální správa byznys kontextů

[T1]fontenc [utf8]inputenc

Kapitola 5

Implementace prototypů knihoven

¹ Součástí zadání této práce je implementace prototypů knihoven pro framework navržený v kapitole 4 pro tři rozdílné platformy, z nichž jedna musí být *Java*. V této kapitole si popíšeme, jaké platformy jsme vybraly, a jakým způsobem byly prototypy knihoven implementovány. Součástí kapitoly je i stručná rešerše technologií, které byly použity pro dosažení vytyčených cílů.

² Jelikož vycházejí implementace knihoven pro všechny platformy ze stejného návrhu, popíšeme si kompletní implementaci pro jazyk *Java* a ostatní implementace shrneme komparativní metodou.

³ Pro splnění cílů bylo potřeba vyřešit také několik technických otázek, jako je přenos byznys kontextů mezi jednotlivými službami, výběr formátu pro zápis byznys kontextu, podpora aspektově orientovaného programování v daném programovacím jazyce a využití principu *runtime weavingu* a integrace knihoven do služeb, které je budou využívat.

5.1 Výběr použitých platforem

⁴ Mimo jazyk *Java*, který byl určen zadáním, byla pro implementaci vybrána platforma jazyka *Python* a platforma *Node.js*, který slouží jako běhové prostředí pro jazyk *JavaScript*. Výběr byl proveden na základě aktuálních trendů ve světě softwarového inženýrství. Projekt *GitHut*⁵ z roku 2014, který shrnuje statistiky repozitářů populární služby pro hosting a sdílení kódu *GitHub*⁶, určil jazyky *JavaScript*, *Java* a *Python* jako tři nejaktivnější. Služba

¹[Intended Delivery: Uvedení kapitoly a nastínění obsahu]

²[Intended Delivery: Nástin formátu kapitoly]

³[Intended Delivery: Technické implementační problémy]

⁴[Intended Delivery: Jaké jsme vybrali další platformy a proč]

⁵<http://github.info/>

⁶<https://github.com/>

GitHub následně sama zveřejnila statistiky za rok 2017 v rámci projektu Octoverse⁷ a dospěla ke stejnému závěru, ačkoliv Python se umístil na druhé pozici na úkor jazyka Java. Podle průzkumu oblíbeného programátorského webového portálu Stack Overflow⁸ se umístily tyto jazyky v první čtveřici nejpopulárnějších jazyků pro obecné použití.

5.2 Sdílení byznys kontextů mezi službami

⁹ Pro sdílení byznys kontextů mezi jednotlivými službami je potřeba je přenášet po síti ve formátu, který bude nezávislý na platformách jednotlivých služeb.

5.2.1 Síťová komunikace

Abychom mohli přenášet byznysové kontexty a jejich pravidla po síti, musíme zvolit protokol a jednotný formát, ve kterém spolu budou jednotlivé služby komunikovat. Tento formát tedy musí být nezávislý na platformě a ideálně by měl být co nejefektivnější v rychlosti přenosu.

5.2.1.1 Protocol Buffers

¹⁰ Pro přenos byznysových kontextů byl zvolen open-source formát *Protocol Buffers*¹¹ vyvinutý společností Google¹². Umožňuje explicitně definovat a vynucovat schéma dat, která jsou přenášena po síti, bez vazby na konkrétní programovací jazyk. Zároveň je díky binární reprezentaci v přenosu velmi efektivní, oproti formátům jako je JSON nebo XML [10]. Zdrojový kód 5.1 znázorňuje zápis schématu zasílaných zpráv ve formátu Protobuffer.

5.2.1.2 gRPC

¹³ Pro komunikaci byznys kontextů nám nestačí pouze přenosový formát, je potřeba také popsat schéma samotné komunikace. K tomu byl zvolen open-source framework gRPC¹⁴, který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat komunikaci pomocí protokolu *RPC* [6].

[TODO]

⁷<https://octoverse.github.com/>

⁸<https://insights.stackoverflow.com/survey/2017#technology>

⁹[Intended Delivery: Jak na přenášení business contextů]

¹⁰[Intended Delivery: Proč jsme použili Protobuf]

¹¹<https://developers.google.com/protocol-buffers/>

¹²<https://www.google.com/>

¹³[Intended Delivery: Proč jsme použili gRPC]

¹⁴<https://grpc.io/>

Listing 5.1: Příklad definice schématu zpráv v jazyce Protobuffer

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Listing 5.2: Příklad definice služeb v gRPC

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

- Popis schématu pro komunikaci
- Diagram komunikace

]

5.3 Knihovna pro platformu Java

[TODO

- Popis business context registry
- Popis expression AST
- Popis tříd kolem business kontextu
- Popis XML parseru a generátoru
- Popis server a klient tříd pro obsluhu GRPC
- Popis weaveru
- Popis anotací pro AOP

]

5.3.1 Použité technologie

5.3.1.1 Apache Maven

¹⁵ Pro správu závislostí a automatickou kompilaci a sestavování knihovny napsané v jazyce java byl zvolen projekt *Maven*. Tento nástroj umožňuje vývojáři komfortně a centrálně spravovat závislosti jeho projektu včetně detailního popisu jejich verze. Dále také umožňuje definovat jakým způsobem bude projekt kompilován.

5.3.1.2 AspectJ

¹⁶ Knihovna AspectJ přináší pro jazyk Java sadu nástrojů, díky kterým lze snadno implementovat koncepty aspektově orientovaného programování.

[TODO

- Ukázka kódu knihovny

]

¹⁵[Intended Delivery: Správa závislostí a buildu projektu]

¹⁶[Intended Delivery: Proč AspectJ a co to umí]

5.4 Knihovna pro platformu Python

[TODO

- Srovnání řešení s knihovnou Java
- Problémy pythonu a jak byly vyřešeny
- Ukázka kódu knihovny
- Použité technologie
- Ukázka AOP v pythonu pomocí vestavěných dekorátorů
- Knihovna pro GRPC
- Popis weaveru

]

5.5 Knihovna pro platformu Node.js

[TODO

- Srovnání řešení s knihovnou Java
- Problémy JS a jak byly vyřešeny
- Ukázka kódu knihovny
- Ukázka AOP v JS pomocí vlastního dekorátoru
- Knihovna pro GRPC
- Popis weaveru

]

¹⁷ Podobně jako byl použit nástroj Maven pro knihovnu v jazyce Java byl využit balíčkovací nástroj *NPM*, který je předinstalován v běhovém prostředí *Node.js*. Dále byl využit nástroj *Yarn*¹⁸

¹⁷[Intended Delivery: Použité technologie pro vývoj knihovny]

¹⁸<http://yarnpkg.com>

5.6 Doménově specifický jazyk pro popis byznys kontextů

¹⁹ Ačkoliv není specifikace a vytvoření doménově specifického jazyka (DSL) hlavním úkolem této práce, pro ověření konceptu bylo nutné definovat alespoň jeho zjednodušenou verzi a implementovat část knihovny, která bude umět jazyk zpracovat a sestavit z něj byznysový kontext v paměti programu.

²⁰ Pro popis kontextů byl jako kompromis mezi jednoduchostí implementace a přívětivostí pro koncového uživatele zvolen univerzální formát Extensible Markup Language (XML) [1]. Tento jazyk umožňuje serializaci libovolných dat, přímočarý a formální zápis jejich struktury a také jejich snadné aplikační zpracování. Zároveň poskytuje relativně dobrou čitelnost pro člověka, ačkoliv speciálně vytvořené DSL by bylo jistě čitelnější.

²¹ Dokumenty XML se skládají z tzv. *entit*, které obsahují buď parsovaná nebo neparsovaná data. Parsovaná data se skládají z jednoduchých znaků reprezentujících jednoduchý text a nebo speciálních značek, neboli *markup*, které slouží k popisu struktury dat. Naopak neparsovaná data mohou obsahovat libovolné znaky, které nenesou žádnou informaci o struktuře dat.

²² Vzhledem k tomu, že XML je volně rozšiřitelný jazyk a neklade meze v možnostech struktury dat, bylo potřeba jasně definovat a dokumentovat očekávanou strukturu dokumentu popisujícího byznys kontext. Pro jazyk XML existuje vícero možností jak schéma definovat [5], od jednoduchého formátu *DTD* až po komplexní formáty jako je *Schematron*, či *XML Schema Definition* (XSD), který byl nakonec zvolen. Díky formálně definovanému schématu můžeme popis byznys kontextu automaticky validovat a vyvarovat se tak případných chyb.

²³ Ve zdrojovém kódu 5.3 můžeme vidět příklad zápisu jednoduchého byznys kontextu s jednou precondition. Samotný zápis byznys kontextu je obsažen v kořenovém elementu `<businessContext>` a jeho název je popsán atributy `prefix` a `name`. Rozšířené kontexty jsou vyčteny v entitě `<includedContexts>`. Preconditions jsou definovány uvnitř entity `<preconditions>` a podobně jsou definovány `<postconditions>`. Obsažená data odpovídají navrženému metamodelu byznysového kontextu z kapitoly 4. Pro zápis podmínek jednotlivých preconditions a post-conditions byl zvolen opis Expression AST. Toto rozhodnutí vychází z předpokladu, že lze vzhledem k povaze prototypu relaxovat podmínku na čitelnost zápisu pravidel ve prospěch jednoduššího zpracování.

¹⁹[Intended Delivery: Popsat proč a jak jsme tvořili DSL]

²⁰[Intended Delivery: Důvody pro výběr XML]

²¹[Intended Delivery: Popis jak XML funguje]

²²[Intended Delivery: Popis jaký formát jsme zvolili pro formální zápis schématu XML dokumentu]

²³[Intended Delivery: Popis formátu]

Listing 5.3: Příklad zápisu byznys kontextu v jazyce XML

```
<?xml version="1.0" encoding="UTF-8"?>
<businessContext prefix="user" name="createEmployee">
  <includedContexts/>
  <preconditions>
    <precondition name="Cannot_use_hidden_product">
      <condition>
        <logicalEquals>
          <left>
            <variableReference
              objectName="product"
              propertyName="hidden"
              type="bool"/>
          </left>
          <right>
            <constant type="bool" value="false"/>
          </right>
        </logicalEquals>
      </condition>
    </precondition>
  </preconditions>
  <postConditions/>
</businessContext>
```

²⁴ Podařilo se nám navrhnout přijatelný formát zápisu byznys kontextu a implementovat části knihoven, které umějí formát číst a zároveň vytvářet. Tím jsme dosáhli možnosti zapisovat kontexty bez ohledu na platformu služby, která je bude využívat. Zároveň tomuto formátu mohou snáze porozumět doménoví experti a mohou se tak zapojit do vývojového procesu.

5.7 Systém pro centrální správu byznys pravidel

[TODO]

- Jak funguje systém

²⁴[\[Intended Delivery: Shrnutí DSL\]](#)

- Přehled, detail a úprava pravidla
- `BusinessContextEditor`
- Uložení pravidla

]

5.7.1 Použité technologie

[TODO

- Uživatelské rozhraní v HTML + CSS
- Jak jsme použili Spring Boot a jeho MVC k nastavení základní webové aplikace
- Dependency Injection Container
- Využití knihovny pro platformu Java

]

5.7.2 Detekce a prevence potenciálních problémů

²⁵ Při úpravě nebo vytváření nového byznysového kontextu je potřeba detekovat případné chyby, abychom změnou neuvedli systém do nekonzistentního stavu. Kromě syntaktických chyb, které jsou detekovány automaticky pomocí definovaného schematu, je potřeba věnovat pozornost také sémantickým chybám. Závažné chyby, které mohou být způsobeny rozšiřováním kontextů, jsou

- a) Závislosti na neexistujících kontextech
- b) Cyklus v grafu závislostí kontextů

²⁶ Kontexty a jejich vzájemné závislosti lze vnímat jako orientovaný graf, kde uzel grafu reprezentuje kontext a orientovaná hrana reprezentuje závislost mezi kontexty. Směr závislosti můžeme pro naše účely zvolit libovolně.

²⁷ Detekce závislosti na neexistujících kontextech je relativně jednoduchým úkolem. Nejprve nastavíme seznam existujících kontextů a následně procházíme jednotlivé hrany grafu

²⁵[Intended Delivery: Problémy způsobené rozšiřováním kontextů]

²⁶[Intended Delivery: Chápání kontextů jako grafu]

²⁷[Intended Delivery: Detekce závislostí na neexistujících kontextech]

kontextů a ověřujeme, zda existují oba kontexty náležící dané hraně. Při zvolení vhodných datových struktur lze dosáhnout lineární složitosti v závislosti na počtu hran grafu.

²⁸ Pokud by závislosti v orientovaném grafu vytvořily cyklus, docházelo by při inicializaci služeb obsahující daná pravidla k zacyklení. Tomu můžeme předejít detekcí cyklů v grafu. Pro tuto detekci byl zvolen Tarjanův algoritmus [8] pro detekci souvislých komponent, který disponuje velmi dobrou lineární složitostí, závislou na součtu počtu hran a počtu uzlů grafu.

²⁹ V případě, že zápis nového či praveného kontextu obsahuje syntaktické chyby a nebo způsobuje některou z detekovaných chyb v závislostech, administrace nedovolí uživateli změnu provést a vypíše informativní chybovou hlášku.

5.8 Shrnutí

³⁰ Na základě navrženého frameworku jsme implementovali prototypy knihoven pro platformy jazyka Java, jazyka Python a ekosystému Node.js. Knihovny umožňují centrální správu a automatickou distribuci byznysových kontextů, včetně vyhodnocování jejich pravidel, za použití aspektově orientovaného přístupu. Dále jsme specifikovali DSL, kterým lze popsat byznys kontext nezávisle na platformě.

³¹ Veškerý kód je hostován v centrálním repozitáři ve službě GitHub³² a je zpřístupněn pod open-source licencí MIT³³. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

³⁴ Prototypy knihoven lze využít k implementaci služeb, potažmo k sestavení funkčního systému, jak si ukážeme v následující kapitole. [T1]fontenc [utf8]inputenc

²⁸[Intended Delivery: Detekce cyklů v grafu závislostí]

²⁹[Intended Delivery: Reakce na chyby]

³⁰[Intended Delivery: Dosáhli jsme vytyčených cílů implementace]

³¹[Intended Delivery: Hostování na GitHubu + licence]

³² <https://github.com/klimesf/diploma-thesis>

³³ <http://www.linfo.org/mitlicense.html>

³⁴[Intended Delivery: Validaci a verifikaci si ještě ukážeme]

Kapitola 6

Verifikace a validace

V této kapitole

6.1 Testování prototypů knihoven

Prototypy knihoven, jejichž implementaci jsme popsali v kapitole 5, byly také důkladně otestovány pomocí sady jednotkových a integračních testů.

V rámci konceptu *continuous integration* [3] byl kód po celou dobu vývoje zasílán do centrálního repozitáře a s pomocí nástroje Travis CI¹ bylo automaticky spouštěno jeho sestavení a otestování. Systém zároveň okamžitě informoval vývojáře o jejich výsledcích. To umožnilo v krátkém časovém horizontu identifikovat konkrétní změny v kódu, které do programu vnesly chybu. Tím byla snížena pravděpodobnost regrese a dlouhodobě se zvýšila celková kvalita kódu.

6.1.1 Platforma Java

Prototyp knihovny pro platformu byl testován pomocí nástroje JUnit², který poskytuje všechny potřebné funkce.

¹<https://travis-ci.org/>

²<https://junit.org/junit4/>

6.1.2 Platforma Python

6.1.3 Platforma Node.js

6.2 Případová studie: e-commerce systém

6.2.1 Model systému

6.2.2 Use-cases

6.2.3 Byznys kontexty

6.2.4 Service discovery

6.2.5 Order service

6.2.6 Product service

6.2.7 User service

6.2.8 Nasazení systému pro centrální správu byznys kontextů

6.3 Shrnutí

Kapitola 7

Závěr

- 7.1 Analýza dopadu použití frameworku
- 7.2 Budoucí rozšiřitelnost frameworku
- 7.3 Možností uplatnění navrženého frameworku
- 7.4 Další možnosti uplatnění AOP v SOA
- 7.5 Shrnutí

Literatura

- [1] BRAY, T. et al. Extensible markup language (XML). *World Wide Web Journal*. 1997, 2, 4, s. 27–66.
- [2] FOWLER, M. – BECK, K. *Refactoring: improving the design of existing code*. Boston, Massachusetts, USA : Addison-Wesley Professional, 1999.
- [3] FOWLER, M. – FOEMMEL, M. Continuous integration. *Thought-Works* <http://www.thoughtworks.com/ContinuousIntegration.pdf>. 2006, 122, s. 14.
- [4] KICZALES, G. et al. Aspect-oriented programming. In *European conference on object-oriented programming*, s. 220–242. Springer, 1997.
- [5] LEE, D. – CHU, W. W. Comparative analysis of six XML schema languages. *Sigmod Record*. 2000, 29, 3, s. 76–87.
- [6] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1981. AAI8204168.
- [7] RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*. 1982, 17, 9, s. 51–57.
- [8] TARJAN, R. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, s. 114–121, Oct 1971. doi: 10.1109/SWAT.1971.10.
- [9] VAN ROY, P. et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*. 2009, 104.
- [10] VARDA, K. Protocol buffers: Google’s data interchange format. *Google Open Source Blog*, Available at least as early as Jul. 2008, 72.

Příloha A

TODO Screenshots

[T1]fontenc [utf8]inputenc

Příloha B

Seznam použitých zkratk

AST Abstract syntax tree

DSL Domain-Specific Language

OOP Objektově orientované programování

RPC Remote Procedure Call

XML Extensible Markup Language

XSD XML Schema Definition

Příloha C

Obsah přiloženého CD

-- nutforms-example/	Ukázkový systém využívající knihovnu
-- dist/	Zkompilované zdrojové soubory pro distribuci
-- docs/	Dokumentace
-- src/	Zdrojový kód aplikace
-- nutforms-ios-client/	Klientská část knihovny pro platformu iOS
-- client/	Zdrojové soubory knihovny
-- clientTests/	Zdrojové soubory testů knihovny
-- dist/	Zkompilované zdrojové soubory pro distribuci
-- docs/	Dokumentace
-- nutforms-server/	Serverová část knihovny
-- dist/	Zkompilované zdrojové soubory pro distribuci
-- docs/	Dokumentace
-- layout/	Layout servlet
-- localization/	Localization servlet
-- meta/	Metadata servlet
-- widget/	Widget servlet
-- nutforms-web-client/	Klientská část knihovny pro webové aplikace
-- dist/	Zkompilované zdrojové soubory pro distribuci
-- docs/	Dokumentace
-- src/	Zdrojové soubory knihovny
-- test/	Zdrojové soubory testů knihovny
-- text/	Text bakalářské práce