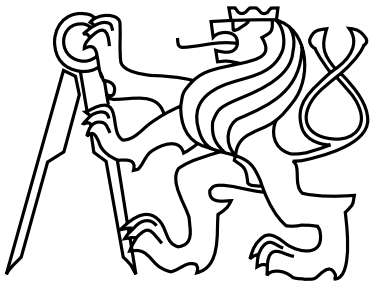
Na tomto místě bude oficiální zadání vaší práce

- 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

12. 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

Seznam obrázků

Seznam zdrojových kódů

Kapitola 1

Úvod

[T1]fontenc [utf8]inputenc graphicx

Kapitola 2

Analýza

2.1 Byznysový kontext

Precondition

Post-condition

2.2 Architektura orientovaná na služby

[1] V posledních dekádách můžeme sledovat trend nárůstu komplexity moderních informačních systémů, který je způsoben stále náročnějšími požadavky na jejich funkcionalitu, výkon a spolehlivost. To nutí vývojáře těchto systémů přizpůsobovat architekturu systému tak, aby uměla splnit všechny očekávané funkční i nefunkční požadavky, zejména pak škálovatelnost systému a jeho schopnost zvládat vysoký objem dat a uživatelů. *Architektura orientovaná na služby* (SOA) je důsledkem této evoluce. Na rozdíl od dřívě běžné *monolitické architektury* SOA podle známého pravidla „rozděl a panuj“ dělí systém na samostatné celky, zvané *služby*, které jsou zodpovědné za dílčí část požadované funkcionality.

[2] To přináší výhody v podobě ...

2.2.1 Historický vývoj SOA

[3] Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture* (CORBA) [20], která vzikala v osmdesátých a devadesátých letech dvacátého století. Ta umožňuje komunikaci mezi aplikacemi implementovanými

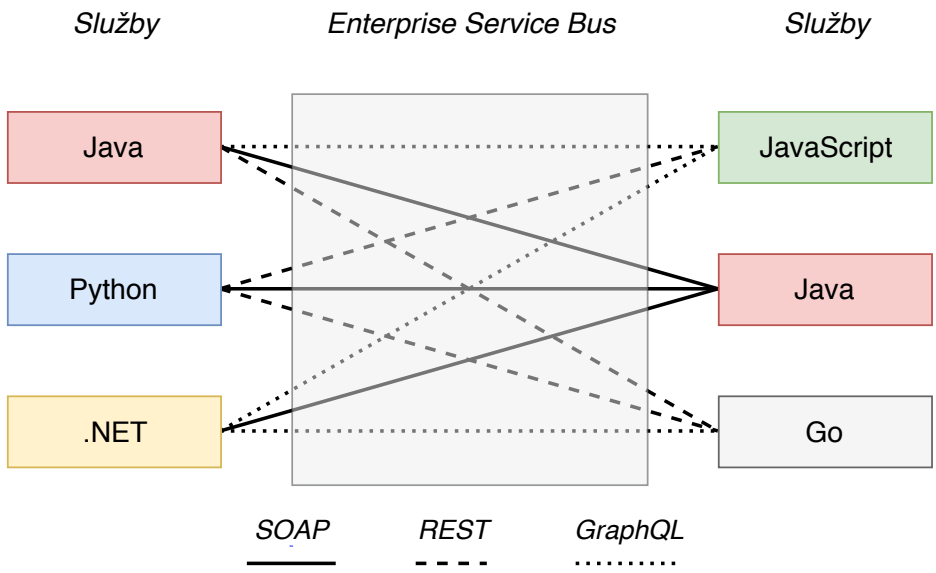
[1] [Intended Delivery: Úvod do SOA, proč je potřeba]
[2] [Intended Delivery: Výhody SOA]
[3] [Intended Delivery: CORBA a její historie]

v různých technologiích a běžícími na vlastních strojích s rozdílnými operačními systémy. Základním stavebním kamenem této architektury je *Object Request Broker* (ORB), který emuluje objekty, na kterých může klient volat jejich metody. Při zavolání metody na objektu, který se fyzicky nachází v aplikaci na vzdáleném stroji, zprostředkovává ORB veškerou komunikaci a svému uživateli poskytuje kompletní rozhraní, které vzdálený objekt emuluje. Uživatel tedy de facto nerozezná, kdy volá metodu na objektu, který je lokálně dostupný, a kdy volá metodu, kterou obsouží vzdálená služba. To je ale zároveň hlavní nevýhodou této architektury, protože komunikace se vzdáleným objektem s sebou nese celou řadu problémů, například mnohem vyšší latenci při komunikaci nebo výjimečné stavy, které je potřeba ošetřit. Ve chvíli, kdy klient není schopen rozeznat mezi metodou volanou lokálně či vzdáleně, se těžko přizpůsobuje těmto okolnostem, což vnáší do kódu zbytečnou komplexitu a zhoršuje jeho kvalitu kvůli obtížnější optimalizaci.

[4] Nedostatky architektury CORBA vedly k volbě jednoduššího formátu pro popis komunikace služeb, spolehlivějšího a méně komplikovaného kanálu pro komunikaci a celkovou redukci objemu komunikovaných dat. Preferovanou cestou komunikace se na přelomu tisíciletí stal protokol HTTP, zatímco preferovaným formátem pro serializaci přenášených dat se stal jazyk XML. Postupně se upustilo od volání metod na vzdálených objektech a přijal se koncept explicitního posílání zpráv mezi službami. Pro popis schématu zpráv vznikl formát *Simple Object Access Protocol* (SOAP) [2], který v kombinaci s *Web Service Description Language* (WSDL) [6] umožňuje kompletní definici rozhraní pro komunikaci služeb systému. V průběhu dalších let vznikla také architektura *Representational State Transfer* (REST) [7], která značně zjednodušila popis webových služeb čistě pomocí protokolu HTTP. Nejnovějším formátem pro popis služby, čerpající z nedostatku REST, je *GraphQL* se kterým v roce 2015 přišla společnost Google.

[5] Ačkoliv zmíněné modely usnaďnují komunikaci služeb a zvyšují jejich spolehlivost, integrace služeb může být obtížnější kvůli různým komunikačním protokolům a formátům. Již v devadesátých letech byl představen koncept *Enterprise Service Bus* (ESB) [5], který měl za úkol propojit heterogenní služby a zajistit jejich komunikační kanály. Tím na sebe přebíral zodpovědnost za překlad jednotlivých zpráv a centralizoval veškerou komunikaci v systému. Na obrázku 2.1 ESB se zároveň staví do role experta na lokalizaci jednotlivých služeb. Službě pro komunikaci s okolním světem tak stačí znát adresu ESB, kterému zašle zprávu, a ten ji sám doručí na místo určení. Tento model ale znamená, že ESB je velmi komplexní komponentou, jejíž výpadek znamená zastavení funkce celého systému a stává se úzkým hrdlem. Tento problém může být částečně vyřešen tzv. *federovaným designem*, kdy je systém rozdělen na byznysově příbuzné části, z nichž každá má svůj ESB.

[4] [Intended Delivery: Web services, SOAP, WSDL]
[5] [Intended Delivery: Enterprise service bus]



Obrázek 2.1: Komunikace služeb skrz Enterprise Service Bus

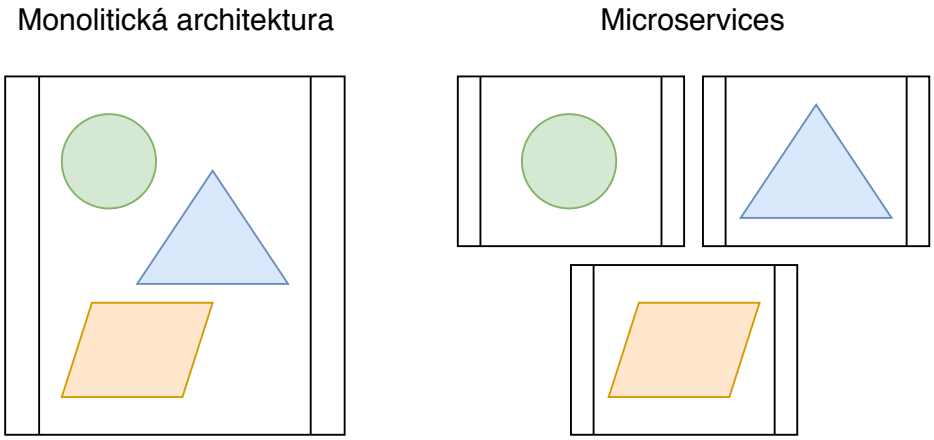
[6] Historicky byl termín SOA vykládán různými způsoby a vývojáři si pod nám představovali několik rozdílných, nekompatibilních konceptů [9]. Zejména pak absence kvalitních definicí toho, co vlastně služba je, vedla k vzájemnému nedorozumění, zmatení a postupném zanevření nad touto architekturou.

2.2.2 Microservices

[7] Novým trendem posledních let jsou takzvané *Microservices*. Přináší několik zajmavých konceptů, které specializují a konkretizují principy SOA. Microservices se tedy dají chápat jako podmnožina SOA. Základní myšlenkou je vývoj informačního systému jako množiny malých oddělených služeb, které jsou spouštěny v samostatných procesech a komunikují spolu pomocí jednoduchých protokolů [15].

[8] Důležitou myšlenkou microservices je organizace služeb kolem byznysových schopností systému. Namísto horizontálního dělení monolitu podle jeho vrstev[9] diktuje rozdělit monolit vertikálně podle jeho byznysových schopností. Na obrázku 2.2 je toto rozdělení demonstrováno. Příkladem může být dělení e-commerce systému na jednu službu obsahující byznysovou logiku pro registraci a správu uživatelů, druhou službu obsahující byznysovou logiku pro práci

[6] [Intended Delivery: Ambiguita termínu SOA v historii]
[7] [Intended Delivery: Microservices a budoucnost SOA]
[8] [Intended Delivery: Stavba služeb kolem byznysových schopností]
[9] Zde předpokládáme klasickou třívrstvou architekturu [8], která rozděluje systém do vrstev s odlišnou zodpovědností, přičemž sousední vrstvy spolu komunikují pomocí jasně definovaných společných rozhraní. Těmito vrstvami jsou: *datová vrstva*, *aplikační vrstva* a *prezentační vrstva*.



Obrázek 2.2: Porovnání struktury monolitické architektury a microservices

s produkty a třetí službu obsahující byznysovou logiku pro práci s objednávkami.

[10] Koncept microservices přemýšlí o službě jako o samostatné komponentě, kterou lze individuálně vyměnit či vylepšit, bez nutnosti zásahu do ostatních služeb [15]. Monolitická architektura vyžaduje i při malé změně jedné části celý systém znovu zkompilovat, sestavit a nasadit. Malé služby sloužící ideálně jedinému byznysovému účelu lze naopak při změně byznysových požadavků snadno vyměnit samostatně bez zásahu do zbytku systému. Tím se usnaďnuje cyklus nasazení a spuštění nové verze služby.

[11] Microservices také přinášejí koncept „smart endpoint, dumb pipes“, který opouští koncept ESB ve prospěch přesunutí veškeré byznys logiky na stranu služeb.

[12]

[18] [4] [21]

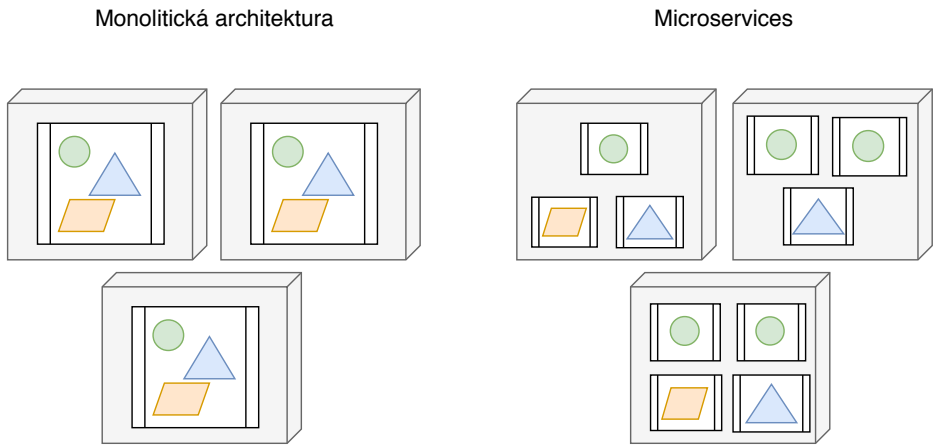
2.2.3 Service discovery

[13]

[14]

Služba je ucelenou systémovou komponentou, kterou lze nasadit a spustit jako samostatný proces, a komunikuje s ostatními službami pomocí zpráv.

[10] [Intended Delivery: Myšlenka nahraditelnosti komponenty]
[11] [Intended Delivery: Myšlenka smart endpoints, dumb pipes]
[12] [Intended Delivery: Škálovatelnost]
[13] [Intended Delivery: Service discovery]
[14] [Intended Delivery: Definice služby]



Obrázek 2.3: Porovnání nasazení monolitické architektury a microservices

2.3 Problémy

[15] Jelikož jedním z cílů microservices je co nejvíce izolovat jednotlivé služby, mají tendenci duplikovat znalosti, které zasahují do více služeb najednou [4]. Zároveň tím ztrácíme

2.4 Identifikace požadavků na implementaci frameworku

- Definice byznys kontextů pomocí doménově specifického jazyka srozumitelného pro doménové experty
- Zápis preconditions a post-conditions pravidla jednotlivých byznys kontextů
- Automatická distribuce kontextů, vyhodnocování jejich preconditions a aplikace post-conditions
- Možnost jednoho kontextu rozšiřovat jiné kontexty
- Možnost centrálně spravovat byznysové kontexty, včetně úpravy stávajících a vytváření nových kontextů

2.5 Shrnutí

V této kapitole jsme nastínili problematiku vysoké komplexity moderních informačních systémů a z toho vyplívající požadavky na jejich architekturu. Analyzovali jsme koncept byznysových pravidel a byznysových kontextů. Dále jsme zkomali architekturu orientovanou

[15] [Intended Delivery: Problémy microservices]

na služby, její výhody a nevýhody, její moderní evoluci v podobě Microservices a identifikovali jsme problémy v souvislosti s byznysovými pravidly, která zasahují do jednotlivých služeb. Nakonec jsme vyjmenovali požadavky, které by měl splňovat framework, který bude výstupem této práce. [T1]fontenc [utf8]inputenc

Kapitola 3

Rešerše

3.1 Modelem řízená architektura

3.2 Architektura klient-server

[1]

3.3 Aspektově orientované programování

[1] 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 [23]. Díky znalosti paradigmatu může programátor nejen zlepšit svou produktivitu, ale zároveň může snáze pochopit myšlenky jiného programátora a tím zlepšit kvalitu týmové spolupráce.

[2] 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 [19]. 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.

[1] [Intended Delivery: Co je paradigma]
[2] [Intended Delivery: OOP a jeho popis]

[3] 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 [13]. Takové požadavky nazýváme *cross-cutting concerns*. V rámci OOP je programátor nucen v ojektech 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 [10].

[4] 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 paradigmatem poskytujícím kompletní framework pro návrh programu. V ideálním případě je tedy k návrhu systému využita kombinace AOP s jiným paradigmatem. 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] [Intended Delivery: Nedostatky OOP]
[4] [Intended Delivery: AOP jako odpověď na nedostatky OOP]

3.5 Stávající řešení reprezentace business pravidel

3.5.1 Drools DSL

[5]

3.5.2 JetBrains MPS

[6]

3.6 Shrnutí

V této kapitole jsme provedli rešerši *architektury orientované na služby*, jejích výhod, nevýhod a známých nedostatků. Dále jsme prozkoumali, jaký způsobem funguje síťová *architektura klient-server*, jaké jsou výhody a nevýhody *modelem řízeného vývoje* pro náš případ a shrnuli jsme paradigma *aspektově orientovaného programování* a z něch vycházející přístup k návrhu softwarových systémů *ADDA*. Nakonec jsme provedli rešerši stávajících řešení reprezentace byznys pravidel včetně komplexního frameworku *Drools* a zhodnotili jsme jeho vhodnost k řešení našeho problému. [T1]fontenc [utf8]inputenc

[5] [Intended Delivery: Drools se nám nehodí, protože je jen pro platformu Java]
[6] [Intended Delivery: MPS je super, ale nevyhovuje nám kvůli dynamickým změnám]

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 byznys kontextu

4.4 Expression

4.5 Registr byznys kontextů

4.6 Byznys kontext weaver

4.7 Centrální správa byznys kontextů

4.7.1 Uložení rozšířeného pravidla

[1] [T1]fontenc [utf8]inputenc

[1] [Intended Delivery: Diskutovat chaining vs. direct update]

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 plaformy 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://githut.info/
[6] https://github.com/

GitHub následně sama zveřejnila statistiky za rok 2017 v rámci projektu Octoverse[7] a dospěla ke stejnému závěru, ačkoliv Python se umístil na druhé pozici na úkor jazyka Java. Podle průzkumu oblíbeného programátorského webového portálu Stack Overflow[8] se umístily tyto jazyky v první čtveřici nejpopulárnějších jazyků pro obecné použití.

5.2 Sdílení byznys kontextů mezi službami

[9] Abychom mohli sdílet byznysové kontexty a jejich pravidla mezi jednotlivými službami, musíme mezi nimi vybudovat síťové komunikační kanály. Je tedy nutné zvolit protokol a jednotný formát, ve kterém spolu budou služby komunikovat. Tento formát musí být nezávislý na platformě a ideálně by měl být co nejefektivnější v rychlosti přenosu.

[10] Pro síťovou komunikaci se nabízí využít architekturu *klient-server*, kterou jsme detailněji popsali v sekci 3.2. Při sdílení kontextů lze chápat *klienta* jako službu, která pro svou funkci vyžaduje získání kontextu definovaného v jiné službě. Jako *server* lze naopak chápat službu, která poskytne své kontexty jiné službě, která na nich závisí. Jinými slovy, klient si vyžádá potřebné kontexty od serveru a ten mu je v odpovědi zašle. Může se také stát, že některá služba bude zároveň serverem jedné služby, a zároveň klientem druhé služby.

5.2.1 Protocol Buffers

[11] Pro přenos byznysových kontextů byl zvolen open-source formát *Protocol Buffers*[12] vyvinutý společností Google[13]. Umožňuje explicitně definovat a vynucovat schéma dat, která jsou přenášena po síti, bez vazby na konkrétní programovací jazyk. Zároveň poskytuje obslužné knihovny pro naše vybrané platformy. Navíc je díky binární reprezentaci dat v přenosu velmi efektivní, oproti formátům jako je JSON nebo XML [24]. Zdrojový kód 5.1 znázorňuje část zápisu schématu zasílaných zpráv obsahující byznys kontexty ve formátu Protobuffer.

Schéma zpráv pro výměnu kontextů opisuje strukturu metamodelu navrženého v sekci 4.3.

ExpressionMessage obsahuje jméno, atributy a argumenty `Expression`

ExpressionPropertyMessage je enumerace obsahující typy atributu `Expression`

PreconditionMessage obsahuje název a podmínku precondition pravidla

[7] https://octoverse.github.com/
[8] https://insights.stackoverflow.com/survey/2017#technology
[9] [Intended Delivery: Formát pro přenos pravidel po síti a jeho výhody]
[10] [Intended Delivery: Architektura klient-server pro komunikaci kontextů mezi službami]
[11] [Intended Delivery: Proč jsme použili Protobuf]
[12] https://developers.google.com/protocol-buffers/
[13] https://www.google.com/

Zdrojový kód 5.1: Část definice schématu zpráv byznys kontextů v jazyce Protobuffer

```
message PreconditionMessage {
    required string name = 1;
    required ExpressionMessage condition = 2;
}

message PostConditionMessage {
    required string name = 1;
    required PostConditionTypeMessage type = 2;
    required string referenceName = 3;
    required ExpressionMessage condition = 4;
}

message BusinessContextMessage {
    required string prefix = 1;
    required string name = 2;
    repeated string includedContexts = 3;
    repeated PreconditionMessage preconditions = 4;
    repeated PostConditionMessage postConditions = 5;
}
```

PostConditionMessage obsahuje název, typ, název odkazovaného pole a podmínku post-condition pravidla

PostConditionTypeMessage je enumerace obsahující typy post-condition pravidla

BusinessContextMessage obsahuje identifikátor, seznam rožšířených kontextů, seznam preconditions a post-conditions byznys kontextu

BusinessContextsMessage obaluje více byznys kontextů

5.2.2 gRPC

[14] Pro realizaci architektury klient-server byl zvolen open-source framework gRPC[15], který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC* [17]. Zdrojový kód 5.2 znázorňuje zápis serveru, který umožňuje svému klientovi volat metody `FetchContexts()`, `FetchAllContexts()` a `UpdateOrSaveContext()`.

[14] [Intended Delivery: Proč jsme použili gRPC]
[15] https://grpc.io/

Zdrojový kód 5.2: Definice služby pro komunikaci byznys kontextů pro gRPC

```
service BusinessContextServer {
    rpc FetchContexts (BusinessContextRequestMessage)
        returns (BusinessContextsResponseMessage) {}
    rpc FetchAllContexts (Empty)
        returns (BusinessContextsResponseMessage) {}
    rpc UpdateOrSaveContext (BusinessContextUpdateRequestMessage)
        returns (Empty) {}
}
```

FetchContexts() je metoda, která umožňuje klientovi získat kontexty, jejichž identifikátory zašle jako argument typu `BusinessContextRequestMessage`. V odpovědi pak obdrží dotazované kontexty a nebo chybovou hlášku, pokud kontexty s danými identifikátory nemá server k dispozici.

FetchAllContexts() dovoluje klientovi získat všechny dostupné kontexty serveru. Tato metoda je využívána pro administraci kontextů, kdy je potřeba získat všechny kontexty všech služeb, aby nad nimi mohly probíhat úpravy a analýzy.

UpdateOrSaveContext() slouží pro uložení nového či editovaného pravidla, které je zasláno v serializované podobě jako jediný argument typu `BusinessContextUpdateRequestMessage`.

5.3 Doménově specifický jazyk pro popis byznys kontextů

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

[17] Pro popis kontextů byl jako kompromis mezi jednoduchostí implementace a přívětivostí pro koncového uživatele zvolen univerzální formát Extensible Markup Language (XML) [3]. 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.

[18] Dokumenty XML se skládají z tzv. *entit*, které obsahují buď parsovaná nebo neparsovaná data. Parsovaná data se skládají z jednoduchých znaků reprezentujících jednoduchý

[16] [Intended Delivery: Popsat proč a jak jsme tvořili DSL]
[17] [Intended Delivery: Důvody pro výběr XML]
[18] [Intended Delivery: Popis jak XML funguje]

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.

[19] 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 [14], 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.

[20] 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.

[21] 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.

5.4 Knihovna pro platformu Java

[TODO

- Popis business context registry
- Popis expression AST
- Popis tříd kolem business kontextu
- Popis XML parseru a generátoru

[19] [Intended Delivery: Popis jaký formát jsme zvolili pro formální zápis schématu XML dokumentu]
[20] [Intended Delivery: Popis formátu]
[21] [Intended Delivery: Shrnutí DSL]

Zdrojový kód 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>
```

Zdrojový kód 5.4: Označení operačního kontextu a jeho parametrů pomocí anotací Java knihovny

```
public class OrderService {

    @BusinessOperation("order.create")
    public Order create(
        @BusinessOperationParameter("user")
        User user,
        @BusinessOperationParameter("email")
        String email,
        @BusinessOperationParameter("shippingAddress")
        Address shippingAddress,
        @BusinessOperationParameter("billingAddress")
        Address billingAddress
    ) { /* ... */ }

}
```

- Popis server a klient tříd pro obsluhu GRPC
- Popis weaveru
- Popis anotací pro AOP
- Návrhové vzory - builder pro kontexty a pravidla
- Návrhové vzory - visitor pro převod expression do xml
- Návrhové vzory - interpreter pro interpretaci pravidel

]

5.4.1 Popis implementace

BusinessContextRegistry

5.4.2 Použité technologie

Apache Maven [22] 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.

[22] [Intended Delivery: Správa závislostí a buildu projektu]

AspectJ [23] Knihovna AspectJ přináší pro jazyk Java sadu nástrojů, díky kterým lze snadno implementovat koncepty aspektově orientovaného programování, zejména pak snadný zápis pointcuts a kompletní engine pro weaving aspektů.

[TODO

- Ukázka kódu knihovny

]

JDOM 2 [24] Knihovna JDOM 2[25] poskytuje kompletní sadu nástroju pro čtení a zápis XML dokumentů. Implementuje specifikaci *Document Object Model* (DOM) [25], pomocí které lze programaticky sestavovat a číst XML dokumenty. Tuto knihovnu jsme využili pro serializaci a deserializaci DSL byznys kontextů popsaných v sekci 5.3.

5.5 Knihovna pro platformu Python

Knihovna pro platformu jazyka Python využívá jeho verzi 3.6. Pomocí nástroje *pip*[26] lze knihovnu nainstalovat a využívat jako python modul. Implementace odpovídá navržené specifikaci.

[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

]

[23] [Intended Delivery: Proč AspectJ a co to umí]
[24] [Intended Delivery: Proč jdom2 a co to umí]
[25] http://www.jdom.org/
[26] https://pip.pypa.io/en/stable/

Zdrojový kód 5.5: Příklad použití dekorátorů pro weaving v jazyce Python

```
def business_operation(name, weaver):
    def wrapper(func):
        def func_wrapper(*args, **kwargs):
            operation_context = OperationContext(name)
            weaver.evaluate_preconditions(operation_context)
            output = func(*args, **kwargs)
            operation_context.set_output(output)
            weaver.apply_post_conditions(operation_context)
            return operation_context.get_output()

        return func_wrapper

    return wrapper


weaver = BusinessContextWeaver()


class ProductRepository:

    @business_operation("product.listAll", weaver)
    def get_all(self) -> List[Product]:
        pass

    @business_operation("product.detail", weaver)
    def get(self, id: int) -> Optional[Product]:
        pass
```

5.5.1 Srovnání s knihovnou pro platformu Java

Weaving Největším rozdílem oproti knihovně pro jazyk Java je implementace weavingu byznys kontextů. Jazyk Python totiž díky své dynamické povaze a vestavěnému systému dekorátorů umožňuje aplikovat principy aspektově orientovaného programování bez potřeby dodatečných knihoven či technologií. Zdrojový kód 5.5 znázorňuje definici a použití dekorátoru `business_operation`. Jak můžeme vidět, je potřeba dekorátoru předat samotný weaver, narozdíl od implementace v Javě, kdy se o předání weaveru postará dependency injection container.

5.5.2 Použité technologie

5.6 Knihovna pro platformu Node.js

Knihovna pro platformu *Node.js* byla implementována v jazyce JavaScript, konkrétně jeho verzi ECMAScript 6.0[27]. Implementace odpovídá specifikaci návrhu, umožňuje instalaci pomocí balíčkovacího nástroje a snadnou integraci do kódu výsledné služby.

5.6.1 Srovnání s knihovnou pro platformu Java

Weaving Podobně jako v knihovně pro jazyk Python, i v knihovně pro Node.js byl oproti knihovně pro jazyk Java největší rozdíl v implementaci weavingu. Platforma Node.js totiž nedisponuje žádnou kvalitní knihovnou, která by ulehčila využití konceptů aspektově orientovaného programování. Jazyk JavaScript je ale velmi flexibilní a lze tedy pro dosažení požadované funkcionality využít podobně jako pro jazyk Python princip dekorátoru jako funkce. Ačkoliv zápis dekorátoru není příliš elegantní a kvůli použití konceptu *Promise* [12] poněkud složitější, podařilo se weaving implementovat spolehlivě. Ukázku můžeme vidět ve zdrojovém kódu 5.6. Funkce `register()` obsahuje logiku pro registraci uživatele, která může obsahovat například uložení entity do databáze a odeslání registračního e-mailu. Při exportování funkce z Node.js modulu využijeme `wrapCall()`, která má za úkol dekorovat předanou funkci `func`, před jejím zavolání vyhodnotit preconditions a po zavolání aplikovat post-conditions. Díky tomu bude každý kód, který využije modul definující funkci pro registraci uživatele, pracovat s dekorovanou funkcí.

Využití gRPC Narozdíl od implementací knihovny v jazycích Java a Python umí knihovna obsluhující gRPC fungovat i bez předgenerovaného kódu. To poněkud usnadnilo práci při serializaci byznys kontextů do přenosového formátu i při deserializaci a ukládání kontextů do paměti. Úspora kódu je ale na úkor typové kontroly a tak může být kód náchylnější na lidskou chybu.

5.6.2 Použité technologie

[28] Podobně jako byl použit nástroj Maven pro knihovnu v jazyce Java byl využit balíčkovací nástroj *NPM*, který je předinstalován v běhovém prostředí *Node.js*. Tento nástroj ale nedisponuje příliš silnou podporou pro správu automatických sestavení knihovny a v

[27] http://www.ecma-international.org/ecma-262/6.0/

[28] [Intended Delivery: Použité technologie pro vývoj knihovny]

Zdrojový kód 5.6: Příklad dekorace funkce v JavaScriptu pro aplikaci weavingu

```
weaver = new BusinessContextWeaver(registry)

function register(name, email) {
  return new Promise((resolve, reject) => {
    // ...
  })
}

function wrapCall(context, func) {
  return new Promise((resolve, reject) => {
      try {
        weaver.evaluatePreconditions(context)
        resolve()
      } catch (error) {
        reject(error.getMessage())
      }
    })
    .then(_ => func())
    .then(result => {
      context.setOutput(result)
      weaver.applyPostConditions(context)
      return new Promise(
          (resolve, reject) => resolve(context.getOutput())
      )
    })
}

exports.register = (name, email) => {
  const context = new BusinessOperationContext('user.register')
  context.setInputParameter('name', name)
  context.setInputParameter('email', email)
  return wrapCall(context, () => register(name, email))
}
```

základním nastavení není ani příliš efektivní pro správu závislostí. Proto bylo nutné využít dodatečné knihovny, jmenovitě *Yarn*[29], *Babel*[30] a *Rimraf*[31].

5.7 Systém pro centrální správu byznys pravidel

[TODO

- Jak funguje systém
- Přehled, detail a úprava pravidla
- `BusinessContextEditor`
- Uložení pravidla

]

5.7.1 Popis implementace

BusinessContextEditor

5.7.2 Detekce a prevence potenciálních problémů

[32] 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

[33] 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.

[29] https://yarnpkg.com/en/
[30] https://babeljs.io/
[31] https://github.com/isaacs/rimraf
[32] [Intended Delivery: Problémy způsobené rozšiřováním kontextů]
[33] [Intended Delivery: Chápání kontextů jako grafu]

[34] Detekce závislosti na neexistujících kontextech je relativně jednoduchým úkolem. Nejprve setavíme seznam existujících kontextů a následně procházíme jednotlivé hrany grafu 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.

[35] 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 [22] pro detekci souvislých komponent, který disponuje velmi dobrou lineární složitostí, závislou na součtu počtu hran a počtu uzlů grafu.

[36] 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.

5.7.3 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

]

5.8 Shrnutí

[37] 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.

[34] [Intended Delivery: Detekce závislostí na neexistujících kontextech]
[35] [Intended Delivery: Detekce cyklů v grafu závislostí]
[36] [Intended Delivery: Reakce na chyby]
[37] [Intended Delivery: Dosáhli jsme vytyčených cílů implementace]

[38] Veškerý kód je hostován v centrálním repozitáři ve službě GitHub[39] a je zpřístupněn pod open-source licencí MIT[40]. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

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

[38] [Intended Delivery: Hostování na GitHubu + licence]
[39] https://github.com/klimesf/diploma-thesis
[40] http://www.linfo.org/mitlicense.html
[41] [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 *continous integration* [11] byl kód po celou dobu vývoje zasílán do centrálního repozitáře a s pomocí nástroje Travis CI[1] 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 JUnit[2], který poskytuje všechny potřebné funkce.

[1] https://travis-ci.org/
[2] https://junit.org/junit4/

Zdrojový kód 6.1: Ukázka využití frameworku Flask pro účely product service

```
from flask import Flask, jsonify

app = Flask(__name__)
product_repository = ProductRepository()

@app.route("/")
def list_all_products():
    result = []
    for product in product_repository.get_all():
        result.append({
            'id': product.id,
            'sellPrice': product.sellPrice,
            'name': product.name,
            'description': product.description
        })
    return jsonify(result)
```

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

Použité technologie Pro vytvoření REST API služby byl využit populární light-weight framework *Flask*. Ve zdrojovém kódu 6.1 můžeme vidět použití tohoto frameworku pro obsluhu požadavku na výpis všech produktů.

6.2.7 User service

6.2.8 Nasazení systému pro centrální správu byznys kontextů

6.2.9 Orchestrace služeb

[3] [16]

6.3 Shrnutí

[T1]fontenc [utf8]inputenc

[3] [Intended Delivery: Spouštění pomocí Docker]

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

[TODO

- Extrakce dokumentace
- Extrakce byznysového modelu
- Konfigurace prostředí

]

7.5 Shrnutí

[TODO

- Dosáhli jsme cílů práce
- Stručné shrnutí co všechno a jak jsme udělali

]

Literatura

[1] BERSON, A. *Client-server architecture.* New York, New York, USA : McGraw-Hill, 1992.

[2] BOX, D. et al. Simple object access protocol (SOAP) 1.1, 2000.

[3] BRAY, T. et al. Extensible markup language (XML). *World Wide Web Journal.* 1997, 2, 4, s. 27–66.

[4] CERNY, T. – DONAHOO, M. J. – PECHANEC, J. Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, s. 228–235. ACM, 2017.

[5] CHAPPELL, D. *Enterprise service bus.* Newton, Massachusetts, USA : O'Reilly Media, Inc., 2004.

[6] CHRISTENSEN, E. et al. Web services description language (WSDL) 1.1, 2001.

[7] FIELDING, R. T. REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California.* 2000.

[8] FOWLER, M. *Patterns of enterprise application architecture.* Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] FOWLER, M. ServiceOrientedAmbiguity. *Martin Fowler–Bliki.* 2005, 1.

[10] FOWLER, M. – BECK, K. *Refactoring: improving the design of existing code.* Boston, Massachusetts, USA : Addison-Wesley Professional, 1999.

[11] FOWLER, M. – FOEMMEL, M. Continuous integration. *Thought-Works http://www.thoughtworks.com/ContinuousIntegration.pdf.* 2006, 122, s. 14.

[12] KAMBONA, K. – BOIX, E. G. – DE MEUTER, W. An evaluation of reactive programming and promises for structuring collaborative web applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, s. 3. ACM, 2013.

[13] KICZALES, G. et al. Aspect-oriented programming. In *European conference on object-oriented programming*, s. 220–242. Springer, 1997.

[14] LEE, D. – CHU, W. W. Comparative analysis of six XML schema languages. *Sigmod Record*. 2000, 29, 3, s. 76–87.

[15] LEWIS, J. – FOWLER, M. Microservices: a definition of this new architectural term. *MartinFowler. com*. 2014, 25.

[16] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*. 2014, 2014, 239, s. 2.

[17] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1981. AAI8204168.

[18] PERREY, R. – LYCETT, M. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, s. 116–119. IEEE, 2003.

[19] RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*. 1982, 17, 9, s. 51–57.

[20] SIEGEL, J. – FRANTZ, D. *CORBA 3 fundamentals and programming*. 2. New York, NY, USA : John Wiley & Sons, 2000.

[21] SPROTT, D. – WILKES, L. Understanding service-oriented architecture. *The Architecture Journal*. 2004, 1, 1, s. 10–17.

[22] 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.

[23] VAN ROY, P. et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*. 2009, 104.

[24] VARDA, K. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul*. 2008, 72.

[25] WOOD, L. et al. Document Object Model (DOM) level 3 core specification, 2004.

Příloha A

TODO Screenshots

[T1]fontenc [utf8]inputenc

Příloha B

Seznam použitých zkratek

ADDA Aspect-Driven Design Approach

API Application Programming Interface

AST Abstract Syntax Tree

CORBA Common Object Request Broker Architecture

DOM Document Object Model

DSL Domain-Specific Language

ESB Enterprise Service Bus

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

MDA Model-Driver Architecture

ORB Object Request Broker

OOP Objektově Oriented Programming (Objektově orientované programování)

REST Representational State Transfer

RPC Remote Procedure Call

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

WSDL Web Service Description Language

XML Extensible Markup Language

XSD XML Schema Definition

[T1]fontenc [utf8]inputenc

Příloha C

Obsah přiloženého CD

```
|-- nutfroms-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
|
|-- nutfroms-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
```