

# 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

<b>1</b>	<b>Úvod</b>	<b>3</b>
1.1	Jak číst tuto práci . . . . .	3
<b>2</b>	<b>Analýza</b>	<b>5</b>
2.1	Byznysový kontext . . . . .	5
2.2	Architektura orientovaná na služby . . . . .	5
2.2.1	Common Object Request Broker Architecture . . . . .	6
2.2.2	Web Services . . . . .	6
2.2.3	Enterprise Service Bus . . . . .	6
2.2.4	Microservices . . . . .	7
2.3	Nedostatky současného přístupu . . . . .	9
2.4	Identifikace požadavků na implementaci frameworku . . . . .	10
2.5	Shrnutí . . . . .	10
<b>3</b>	<b>Rešerše</b>	<b>11</b>
3.1	Modelem řízená architektura . . . . .	11
3.2	Architektura klient-server . . . . .	11
3.3	Aspektově orientované programování . . . . .	12
3.4	Aspect-driven Design Approach . . . . .	13
3.5	Stávající řešení reprezentace business pravidel . . . . .	14
3.5.1	Drools DSL . . . . .	14
3.5.2	JetBrains MPS . . . . .	14
3.6	Shrnutí . . . . .	15
<b>4</b>	<b>Návrh</b>	<b>17</b>
4.1	Formalizace architektury orientované na služby . . . . .	17
4.1.1	Join-points . . . . .	17
4.1.2	Advices . . . . .	17
4.1.3	Pointcuts . . . . .	17
4.1.4	Weaving . . . . .	17

4.2	Architektura frameworku . . . . .	17
4.3	Metamodel byznys kontextu . . . . .	17
4.4	Expression . . . . .	17
4.5	Registr byznys kontextů . . . . .	17
4.6	Byznys kontext weaver . . . . .	17
4.7	Centrální správa byznys kontextů . . . . .	17
4.7.1	Uložení rozšířeného pravidla . . . . .	17
4.8	Service discovery . . . . .	18
<b>5</b>	<b>Implementace prototypů knihoven</b>	<b>19</b>
5.1	Výběr použitých platform . . . . .	19
5.2	Sdílení byznys kontextů mezi službami . . . . .	20
5.2.1	Protocol Buffers . . . . .	20
5.2.2	gRPC . . . . .	21
5.3	Doménově specifický jazyk pro popis byznys kontextů . . . . .	22
5.4	Knihovna pro platformu Java . . . . .	23
5.4.1	Popis implementace . . . . .	25
5.4.2	Použité technologie . . . . .	25
5.5	Knihovna pro platformu Python . . . . .	26
5.5.1	Srovnání s knihovnou pro platformu Java . . . . .	27
5.5.2	Použité technologie . . . . .	28
5.6	Knihovna pro platformu Node.js . . . . .	28
5.6.1	Srovnání s knihovnou pro platformu Java . . . . .	28
5.6.2	Použité technologie . . . . .	28
5.7	Systém pro centrální správu byznys pravidel . . . . .	30
5.7.1	Popis implementace . . . . .	30
5.7.2	Detekce a prevence potenciálních problémů . . . . .	30
5.7.3	Použité technologie . . . . .	31
5.8	Shrnutí . . . . .	31
<b>6</b>	<b>Verifikace a validace</b>	<b>33</b>
6.1	Testování prototypů knihoven . . . . .	33
6.1.1	Platforma Java . . . . .	33
6.1.2	Platforma Python . . . . .	34
6.1.3	Platforma Node.js . . . . .	34
6.2	Případová studie: e-commerce systém . . . . .	34
6.2.1	Model systému . . . . .	34
6.2.2	Use-cases . . . . .	34
6.2.3	Byznys kontexty . . . . .	34

6.2.4	Service discovery . . . . .	34
6.2.5	Order service . . . . .	34
6.2.6	Product service . . . . .	34
6.2.7	User service . . . . .	35
6.2.8	Nasazení systému pro centrální správu byznys kontextů . . . . .	35
6.2.9	Orchestrace služeb . . . . .	35
6.3	Shrnutí . . . . .	35
<b>7</b>	<b>Závěr</b>	<b>37</b>
7.1	Analýza dopadu použití frameworku . . . . .	37
7.2	Budoucí rozšiřitelnost frameworku . . . . .	37
7.3	Možností uplatnění navrženého frameworku . . . . .	37
7.4	Další možnosti uplatnění AOP v SOA . . . . .	37
7.5	Shrnutí . . . . .	37
<b>A</b>	<b>TODO Screenshots</b>	<b>43</b>
<b>B</b>	<b>Seznam použitých zkratk</b>	<b>45</b>
<b>C</b>	<b>Obsah přiloženého CD</b>	<b>47</b>



# Seznam obrázků

2.1	Komunikace služeb skrz Enterprise Service Bus . . . . .	7
2.2	Porovnání struktury monolitické architektury a microservices . . . . .	8
2.3	Porovnání nasazení monolitické architektury a microservices . . . . .	9





# Seznam zdrojových kódů

5.1	Část definice schématu zpráv byznys kontextů v jazyce Protobuffer . . . . .	21
5.2	Definice služby pro komunikaci byznys kontextů pro gRPC . . . . .	22
5.3	Příklad zápisu byznys kontextu v jazyce XML . . . . .	24
5.4	Označení operačního kontextu a jeho parametrů pomocí anotací Java knihovny . . . . .	25
5.5	Příklad použití dekorátorů pro weaving v jazyce Python . . . . .	27
5.6	Příklad dekorace funkce v JavaScriptu pro aplikaci weavingu . . . . .	29
6.1	Ukázka využití frameworku Flask pro účely product service . . . . .	34



[T1]fontenc [utf8]inputenc



# Kapitola 1

## Úvod

[TODO

- popis tématu a jeho důležitost
- motivace práce
- co bude cílem a požadovaným výstupem této práce
- popis struktury DP

]

### 1.1 Jak číst tuto práci

<sup>1</sup> Kapitola 2 se venuje detailní analýze problematiky byznysových pravidel a architektury orientované na služby, včetně jejího historického vývoje až po nejnovější trendy a v závěru identifikuje požadavky kladené na implementaci knihovny pro centrální správu a automatickou distribuci byznysových pravidel v této architektuře. Kapitola 3 se zabývá řešením stávajících přístupů k vyvoji informacních systémů a speciálně se zaměřuje na koncepty aspektově orientovaného programování a moderního aspektu řízeného přístupu k návrhu systémů. Dále se kapitola věnuje průzkumu existujících nástrojů pro správu byznysových pravidel. Kapitola 4 formalizuje prostředí architektury orientované na služby do terminologie aspektově orientovaného programování a na základě této formalizace navrhuje koncept frameworku, který realizuje centrální správu a automatickou distribuci byznysových pravidel. V kapitole 5 je detailně probrána implementace knihoven pro navržený framework pro platformy

---

<sup>1</sup>[Intended Delivery: Popis struktury DP a obsah kapitol]

jazyků Java a Python a ekosystému Node.js. Následující kapitola 6 popisuje, jakým způsobem byly tyto knihovny otestovány a jak byla prokázána jejich funkčnost. Zároveň je zde popsána validace a vyhodnocení konceptu frameworku jeho nasazením při vývoji jednoduchého ukázkového e-commerce systému. V poslední kapitole 7 je shrnuto, jakých cílů bylo v práci dosaženo a jakým dalším směrem se může výzkum v této oblasti ubírat. [T1]fontenc [utf8]inputenc graphicx

# Kapitola 2

## Analýza

### 2.1 Byznysový kontext

Precondition

Post-condition

### 2.2 Architektura orientovaná na služby

<sup>1</sup> V posledních dekáдах 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 dříve běžné a dnes stále používané *monolitické architektury* SOA podle známého pravidla „rozděl a panuj“ dělí systém na samostatné celky, zvané *služby*, které jsou zodpovědné za dílčí část požadované funkcionality.

<sup>2</sup> Historicky byl termín SOA vykládán různými způsoby a vývojáři si pod ním představovali několik rozdílných, nekompatibilních konceptů [13]. Zejména pak absence kvalitních definic toho, co vlastně služba je, vedla k vzájemnému nedorozumění, zmatení a v poslední době i ke snahám o opuštění tohoto konceptu. Abychom lépe prozuměli tomu, co vlastně SOA je, popíšeme si její historický vývoj a shrneme výhody a nevýhody jednotlivých přístupů.

---

<sup>1</sup>[Intended Delivery: Úvod do SOA, proč je potřeba]

<sup>2</sup>[Intended Delivery: Proč tu vlastně píšu o nějaké historii]

### 2.2.1 Common Object Request Broker Architecture

Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture* (CORBA) [33], která vznikala v osmdesátých a devadesátých letech dvacátého století. Ta umožňuje komunikaci mezi aplikacemi implementovanými 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.

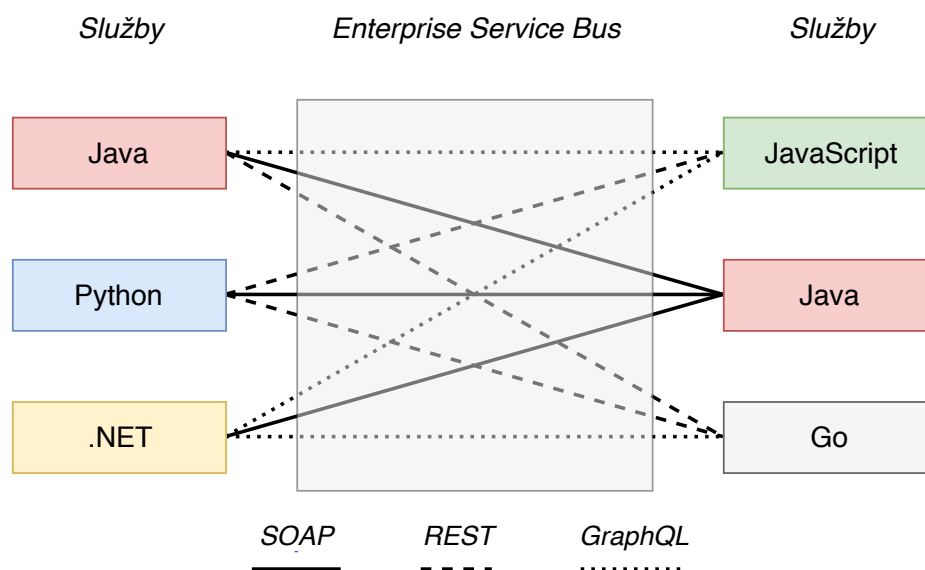
### 2.2.2 Web Services

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) [4], který v kombinaci s *Web Service Description Language* (WSDL) [8] umožňuje kompletní definici rozhraní pro komunikaci služeb systému. V průběhu dalších let vznikla také architektura *Representational State Transfer* (REST) [11], 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.

### 2.2.3 Enterprise Service Bus

Ačkoliv zmíněné modely usnadňují 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) [7], který měl za úkol propojit heterogenní služby a zajistit jejich komunikační kanály. Tím na sebe přebíral





Obrázek 2.1: Komunikace služeb skrz Enterprise Service Bus

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.

#### 2.2.4 Microservices

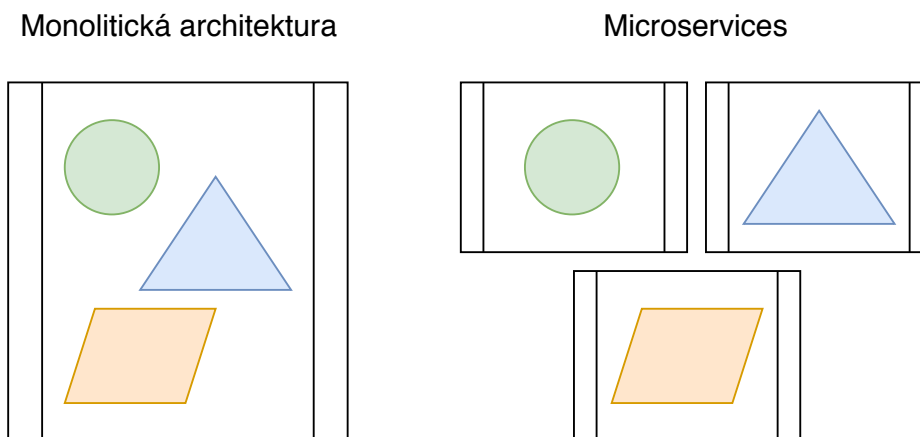
<sup>3</sup> Novým trendem posledních let jsou takzvané *Microservices*. Přináší několik zajímavých konceptů, které specializují a konkretizují principy SOA. Microservices se tedy dají chápat jako podmnožina SOA. Základní myšlenkou je vývoj informačního systému jako množiny malých oddělených služeb, které jsou spouštěny v samostatných procesech a komunikují spolu pomocí jednoduchých protokolů [23].

<sup>4</sup> Důležitou myšlenkou microservices je organizace služeb kolem byznysových schopností systému. Namísto horizontálního dělení monolitu podle jeho vrstev<sup>5</sup> diktuje rozdělit monolit

<sup>3</sup>[Intended Delivery: Microservices a budoucnost SOA]

<sup>4</sup>[Intended Delivery: Stavba služeb kolem byznysových schopností]

<sup>5</sup> Zde předpokládáme klasickou třívrstvou architekturu [12], rozdělující systém na *datovou vrstvu*, *aplikační vrstvu* a *prezentační vrstvu*. Tyto vrstvy mají oddělené zodpovědnosti a komunikují spolu pomocí jasně definovaných společných rozhraní.



Obrázek 2.2: Porovnání struktury monolitické architektury a microservices

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

<sup>6</sup> Koncept microservices přemýšlí o službě jako o samostatné komponentě, kterou lze individuálně vyměnit či vylepšit, bez nutnosti zásahu do ostatních služeb [23]. 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 usnadňuje cyklus nasazení a spuštění nové verze služby.

<sup>7</sup> Microservices také přináší koncept „smart endpoint, dumb pipes“, který opouští koncept ESB ve prospěch přesunutí veškeré byznys logiky na stranu služeb.

<sup>8</sup>

[28] [6] [34]

<sup>9</sup>

<sup>10</sup> [27]

<sup>11</sup>

<sup>12</sup>

<sup>6</sup>[Intended Delivery: Myšlenka nahraditelnosti komponenty]

<sup>7</sup>[Intended Delivery: Myšlenka smart endpoints, dumb pipes]

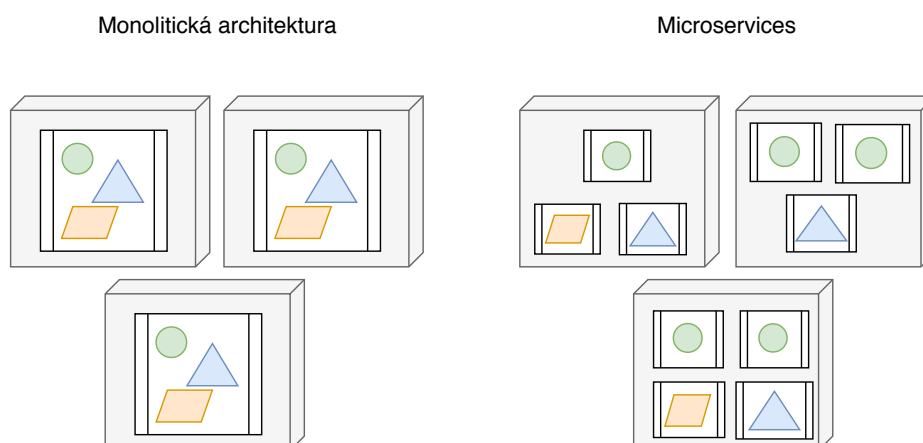
<sup>8</sup>[Intended Delivery: Škálovatelnost]

<sup>9</sup>[Intended Delivery: Distribuce ]

<sup>10</sup>[Intended Delivery: Four tier architecture]

<sup>11</sup>[Intended Delivery: Service discovery]

<sup>12</sup>[Intended Delivery: Definice služby]



Obrázek 2.3: Porovnání nasazení monolitické architektury a microservices

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

## 2.3 Nedostatky současného přístupu

<sup>13</sup> Jelikož jedním z cílů SOA, potažmo microservices, je izolovat jednotlivé služby, mají tyto architektury tendenci duplikovat části kódu zajišťující funkcionalitu, která vyžaduje konzistentní zpracování ve více službách [6], tzv. *cross-cutting concerns*. Příkladem mohou být právě byznysová pravidla, která je potřeba zohlednit v rámci různých byznysových kontextů realizovaných ve více službách. Mezi další příklady se řadí logování, monitoring či sběr dat o telemetrii procesů.

<sup>14</sup> Abychom si modli lépe představit diskutovaný problém, znázorněme si ho na konkrétním příkladu. Představme si e-commerce systém skládající se z několika služeb naprogramovaných v různých technologiích, organizovaných kolem jeho byznysových funkcí. Jedna služba obsluhuje byznysové operace vázající se na uživatele systému, jejich registraci a administraci. Druhá služba realizuje operace s produkty, jejich vytváření, úpravu, správu skladových zásob a informace o dostupnosti. Třetí služba je zodpovědná za vytváření a správu objednávek, informování uživatelů o změnách jejich stavů a vytváření statistik a reportů pro management. Poslední, čtvrtá služba, poskytuje uživatelské rozhraní pro uživatele systému a umožňuje jeho komfortní obsluhu.

<sup>15</sup> Nyní si představme, že každá byznysová operace prováděná v těchto službách má své

<sup>13</sup>[\[Intended Delivery: Problémy SOA s cross-cutting concerns\]](#)

<sup>14</sup>[\[Intended Delivery: Nastínění konkrétního příkladu\]](#)

<sup>15</sup>[\[Intended Delivery: Konkrétní problémy zpracování cross-cutting concerns na příkladu\]](#)

preconditions, které musejí být splněny aby mohla být operace vykonána, a také své post-conditions, které musejí být aplikovány po skončení operace.

<sup>16</sup> Pozorný čtenář může namítnout, že výše zmiňované problémy by bylo možné vyřešit vyčleněním dané funkcionality do samostatné služby a vystavit její rozhraní pro ostatní služby, v souladu s nosnou myšlenkou microservices. Je pravda, že microservices v názvu nese slovo „micro“ a evokuje tak, že služby by měly být co nejmenší. Pokud ale služby ponesou příliš málo odpovědnosti, přináší to s sebou několik problémů, které je nutné zvážit. Musíme mít na paměti, že nasazení a provoz každé služby s sebou přináší náklady navíc a zvyšuje časové nároky na jejich vývojáře a administrátory. Komunikace služeb po síti je navíc podstatně pomalejší a náchylnější na chybu než komunikace jednotlivých komponent v rámci jednoho procesu. S rostoucím počtem *cross-cutting concerns* by tak i rychle rostl počet služeb v systému a celkové náklady na jeho vývoj a údržbu.

## 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 complexity moderních informačních systémů a z toho vyplívající požadavky na jejich architekturu. Analyzovali jsme koncept byznysových pravidel a byznysových kontextů. Dále jsme zkomali architekturu orientovanou 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

---

<sup>16</sup>[\[Intended Delivery: Microservices neříká nic o tom, jak velké je mikro\]](#)

# Kapitola 3

## Rešerše

### 3.1 Modelem řízená architektura

[TODO

- Co to je MDA
- Výhody
- Nevýhody
- Shrnutí a proč se nám nehodí

]

### 3.2 Architektura klient-server

[TODO

- Co to je
- Výhody
- Nevýhody
- Shrnutí a proč se nám hodí
- Citovat [\[3\]](#)

]

### 3.3 Aspektově orientované programování

<sup>1</sup> 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 [37]. Díky znalosti paradigmatu může programátor nejen zlepšit svou produktivitu, ale zároveň může snáze pochopit myšlenky jiného programátora a tím zlepšit kvalitu týmové spolupráce.

<sup>2</sup> Jedním z nejpoblárnějších 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 [32]. Je vhodné zmínit, že objekty se rozumí jak konkrétní koncepty, například auto nebo člověk, tak i abstraktní koncepty, například 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.

<sup>3</sup> 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 [21]. 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 [14].

<sup>4</sup> 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.

---

<sup>1</sup>[Intended Delivery: Co je paradigma]

<sup>2</sup>[Intended Delivery: OOP a jeho popis]

<sup>3</sup>[Intended Delivery: Nedostatky OOP]

<sup>4</sup>[Intended Delivery: AOP jako odpověď na nedostatky OOP]

[TODO

- diagram cross cutting concerns
- roztáhnout do více odstavců
- diagram weaveru
- section BPEL

]

**Aspekt**

**Join-point**

**Pointcut**

**Advice**

**Weaving**

## 3.4 Aspect-driven Design Approach

[TODO

- co to je
- jak nám to pomůže
- využití jeho konceptů
- shrnutí

]

Aspect-driven Design Approach (ADDA)

<sup>5</sup> 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.

---

<sup>5</sup>[Intended Delivery: Vhodnost AOP pro náš úkol]

## 3.5 Stávající řešení reprezentace business pravidel

### 3.5.1 Drools DSL

[TODO

- co to je
- jak to funguje
- výhody
- nevýhody
- shrnutí a proč se nám nehodí

]

<sup>6</sup> ...

### 3.5.2 JetBrains MPS

[TODO

- co to je
- jak to funguje
- výhody
- nevýhody
- shrnutí a proč se nám nehodí

]

<sup>7</sup> ...

---

<sup>6</sup>[Intended Delivery: Drools se nám nehodí, protože je jen pro platformu Java]

<sup>7</sup>[Intended Delivery: MPS je super, ale nevyhovuje nám kvůli dynamickým změnám]



## 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



# 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

<sup>1</sup>

---

<sup>1</sup>[\[Intended Delivery: Diskutovat chaining vs. direct update\]](#)

## 4.8 Service discovery

<sup>2</sup> [T1]fontenc [utf8]inputenc

---

<sup>2</sup>[Intended Delivery: Popsat nezávislost na service discovery]

## Kapitola 5

# Implementace prototypů knihoven

<sup>1</sup> 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.

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

<sup>3</sup> 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 platform

<sup>4</sup> 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 GitHub [30] z roku 2014, který shrnuje statistiky repozitářů populární služby pro hosting a sdílení kódu GitHub<sup>5</sup>, určil jazyky *JavaScript*, *Java* a *Python* jako tři nejaktivnější. Služba GitHub následně sama zveřejnila statistiky za rok 2017 v rámci projektu Octoverse [16]

---

<sup>1</sup>[Intended Delivery: Uvedení kapitoly a nastínění obsahu]

<sup>2</sup>[Intended Delivery: Nástin formátu kapitoly]

<sup>3</sup>[Intended Delivery: Technické implementační problémy]

<sup>4</sup>[Intended Delivery: Jaké jsme vybrali další platformy a proč]

<sup>5</sup><https://github.com/>

a dospěla ke stejnému závěru, ačkoliv Python se umístil na druhé pozici na úkor jazyka Java. Podle průzkumu oblíbeného programátorského webového portálu Stack Overflow [35] se umístily tyto jazyky v první čtveřici nejpopulárnějších jazyků pro obecné použití.

## 5.2 Sdílení byznys kontextů mezi službami

<sup>6</sup> 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.

<sup>7</sup> 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

<sup>8</sup> Pro přenos byznysových kontextů byl zvolen open-source formát *Protocol Buffers* [31][38] vyvinutý společností Google<sup>9</sup>. 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]. Oproti protokolům *Apache Thrift* [1] a *Apache Avro* [39], které poskytují velmi srovnatelnou funkcionalitu, mají Protocol Buffers kvalitnější a lépe srozumitelnou dokumentaci.

Zdrojový kód 5.1 znázorňuje část zápisu schématu zasílaných zpráv obsahující byznys kontexty ve formátu Protobuf. 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

---

<sup>6</sup>[Intended Delivery: Formát pro přenos pravidel po síti a jeho výhody]

<sup>7</sup>[Intended Delivery: Architektura klient-server pro komunikaci kontextů mezi službami]

<sup>8</sup>[Intended Delivery: Proč jsme použili Protobuf]

<sup>9</sup><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 rozšířených kontextů, seznam preconditions a post-conditions byznys kontextu

**BusinessContextsMessage** obaluje více byznys kontextů

### 5.2.2 gRPC

<sup>10</sup> Pro realizaci architektury klient-server byl zvolen open-source framework gRPC [17], který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC* [26]. Zdrojový kód 5.2 znázorňuje zápis serveru, který umožňuje svému klientovi volat metody `FetchContexts()`, `FetchAllContexts()` a `UpdateOrSaveContext()`.

---

<sup>10</sup>[\[Intended Delivery: Proč jsme použili gRPC\]](#)

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

<sup>11</sup> 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.

<sup>12</sup> Pro popis kontextů byl jako kompromis mezi jednoduchostí implementace a přívětivostí pro koncového uživatele zvolen univerzální formát Extensible Markup Language (XML) [5]. 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.

<sup>13</sup> Dokumenty XML se skládají z tzv. *entit*, které obsahují buď parsovaná nebo neparsovaná data. Parsovaná data se skládají z jednoduchých znaků reprezentujících jednoduchý

---

<sup>11</sup>[Intended Delivery: Popsat proč a jak jsme tvořili DSL]

<sup>12</sup>[Intended Delivery: Důvody pro výběr XML]

<sup>13</sup>[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.

<sup>14</sup> 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 [22], 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.

<sup>15</sup> 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.

<sup>16</sup> 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

---

<sup>14</sup>[Intended Delivery: Popis jaký formát jsme zvolili pro formální zápis schématu XML dokumentu]

<sup>15</sup>[Intended Delivery: Popis formátu]

<sup>16</sup>[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** <sup>17</sup> 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.

---

<sup>17</sup>[\[Intended Delivery: Správa závislostí a buildu projektu\]](#)

**AspectJ** <sup>18</sup> 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** <sup>19</sup> Knihovna JDOM 2 [19] poskytuje kompletní sadu nástrojů pro čtení a zápis XML dokumentů. Implementuje specifikaci *Document Object Model* (DOM) [41], 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* [29] 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

]

---

<sup>18</sup>[Intended Delivery: Proč AspectJ a co to umí]

<sup>19</sup>[Intended Delivery: Proč jdom2 a co to umí]

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 [9]. 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* [20] 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 zavoláním 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

<sup>20</sup> 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 základním nastavení není ani příliš efektivní pro správu závislostí. Proto bylo nutné využít dodatečné knihovny, jmenovitě *Yarn* [10], *Babel* [2] a *Rimraf* [18].

---

<sup>20</sup>[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))
}
```

## 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

<sup>21</sup> 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 schématu, 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

<sup>22</sup> 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.

<sup>23</sup> Detekce závislosti na neexistujících kontextech je relativně jednoduchým úkolem. Nejprve nastavíme 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.

<sup>24</sup> 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.

---

<sup>21</sup>[Intended Delivery: Problémy způsobené rozšiřováním kontextů]

<sup>22</sup>[Intended Delivery: Chápání kontextů jako grafu]

<sup>23</sup>[Intended Delivery: Detekce závislostí na neexistujících kontextech]

<sup>24</sup>[Intended Delivery: Detekce cyklů v grafu závislostí]



Pro tuto detekci byl zvolen Tarjanův algoritmus [36] pro detekci souvislých komponent, který disponuje velmi dobrou lineární složitostí, závislou na součtu počtu hran a počtu uzlů grafu.

<sup>25</sup> 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

<sup>26</sup> 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.

<sup>27</sup> Veškerý kód je hostován v centrálním repozitáři ve službě GitHub<sup>28</sup> a je zpřístupněn pod open-source licencí MIT [40]. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

<sup>29</sup> Prototypy knihoven lze využít k implementaci služeb, potažmo k sestavení funkčního systému, jak si ukážeme v následující kapitole. [T1]fontenc [utf8]inputenc

---

<sup>25</sup>[Intended Delivery: Reakce na chyby]

<sup>26</sup>[Intended Delivery: Dosáhli jsme vytyčených cílů implementace]

<sup>27</sup>[Intended Delivery: Hostování na GitHubu + licence]

<sup>28</sup> <https://github.com/klimesf/diploma-thesis>

<sup>29</sup>[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* [15] byl kód po celou dobu vývoje zasílán do centrálního repozitáře a s pomocí nástroje Travis CI<sup>1</sup> 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<sup>2</sup>, který poskytuje všechny potřebné funkce.

---

<sup>1</sup><https://travis-ci.org/>

<sup>2</sup><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

<sup>3</sup> [25]

### 6.3 Shrnutí

[T1]fontenc [utf8]inputenc

---

<sup>3</sup>[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] *Apache Thrift - Home* [online]. Dostupné z: <<https://thrift.apache.org/>>.
- [2] *Babel · The compiler for writing next generation JavaScript* [online]. Dostupné z: <<https://babeljs.io/>>.
- [3] BERSON, A. *Client-server architecture*. New York, New York, USA : McGraw-Hill, 1992.
- [4] BOX, D. et al. Simple object access protocol (SOAP) 1.1, 2000.
- [5] BRAY, T. et al. Extensible markup language (XML). *World Wide Web Journal*. 1997, 2, 4, s. 27–66.
- [6] 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.
- [7] CHAPPELL, D. *Enterprise service bus*. Newton, Massachusetts, USA : O'Reilly Media, Inc., 2004.
- [8] CHRISTENSEN, E. et al. Web services description language (WSDL) 1.1, 2001.
- [9] *ECMAScript® 2015 Language Specification - Ecma-262 6th Edition* [online]. Dostupné z: <<http://www.ecma-international.org/ecma-262/6.0/>>.
- [10] *Fast, reliable, and secure dependency management*. [online]. Dostupné z: <<https://yarnpkg.com/en/>>.
- [11] FIELDING, R. T. REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*. 2000.
- [12] FOWLER, M. *Patterns of enterprise application architecture*. Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.
- [13] FOWLER, M. ServiceOrientedAmbiguity. *Martin Fowler-Bliki*. 2005, 1.

- [14] FOWLER, M. – BECK, K. *Refactoring: improving the design of existing code*. Boston, Massachusetts, USA : Addison-Wesley Professional, 1999.
- [15] FOWLER, M. – FOEMMEL, M. Continuous integration. *Thought-Works* <http://www.thoughtworks.com/ContinuousIntegration.pdf>. 2006, 122, s. 14.
- [16] *GitHub Octoverse 2017* [online]. 2017. Dostupné z: <<https://octoverse.github.com/>>.
- [17] *gRPC open-source universal RPC framework* [online]. Dostupné z: <<https://grpc.io/>>.
- [18] *isaacs/rimraf* [online]. Dostupné z: <<https://github.com/isaacs/rimraf>>.
- [19] *JDOM* [online]. Dostupné z: <<http://www.jdom.org/>>.
- [20] 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.
- [21] KICZALES, G. et al. Aspect-oriented programming. In *European conference on object-oriented programming*, s. 220–242. Springer, 1997.
- [22] LEE, D. – CHU, W. W. Comparative analysis of six XML schema languages. *Sigmod Record*. 2000, 29, 3, s. 76–87.
- [23] LEWIS, J. – FOWLER, M. Microservices: a definition of this new architectural term. *MartinFowler. com*. 2014, 25.
- [24] MAEDA, K. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, s. 177–182. IEEE, 2012.
- [25] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*. 2014, 2014, 239, s. 2.
- [26] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1981. AAI8204168.
- [27] NOMMENSEN, P. *It's Time to Move to a Four-Tier Application Architecture* [online]. Dostupné z: <<https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/>>.

- [28] PERREY, R. – LYCETT, M. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, s. 116–119. IEEE, 2003.
- [29] *pip - pip 9.0.3 documentation* [online]. Dostupné z: <<https://pip.pypa.io/en/stable/>>.
- [30] *Programming Languages and GitHub* [online]. 2014. Dostupné z: <<http://github.info/>>.
- [31] *Protocol Buffers / Google Developers* [online]. Dostupné z: <<https://developers.google.com/protocol-buffers/>>.
- [32] RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*. 1982, 17, 9, s. 51–57.
- [33] SIEGEL, J. – FRANTZ, D. *CORBA 3 fundamentals and programming*. 2. New York, NY, USA : John Wiley & Sons, 2000.
- [34] SPROTT, D. – WILKES, L. Understanding service-oriented architecture. *The Architecture Journal*. 2004, 1, 1, s. 10–17.
- [35] *Stack Overflow Developer Survey 2017* [online]. 2017. Dostupné z: <<https://insights.stackoverflow.com/survey/2017#technology>>.
- [36] 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.
- [37] VAN ROY, P. et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*. 2009, 104.
- [38] VARDA, K. Protocol buffers: Google’s data interchange format. *Google Open Source Blog, Available at least as early as Jul*. 2008, 72.
- [39] *Welcome to Apache Avro!* [online]. Dostupné z: <<https://avro.apache.org/>>.
- [40] *What is the MIT license? – definition by The Linux Information Project (LINFO)* [online]. Dostupné z: <<http://www.linfo.org/mitlicense.html>>.
- [41] 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 zkratk

<b>ADDA</b>	Aspect-Driven Design Approach
<b>API</b>	Application Programming Interface
<b>AST</b>	Abstract Syntax Tree
<b>CORBA</b>	Common Object Request Broker Architecture
<b>DOM</b>	Document Object Model
<b>DSL</b>	Domain-Specific Language
<b>ESB</b>	Enterprise Service Bus
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IDL</b>	Interface Description Language
<b>JSON</b>	JavaScript Object Notation
<b>MDA</b>	Model-Driven Architecture
<b>ORB</b>	Object Request Broker
<b>OOP</b>	Objektově Oriented Programming (Objektově orientované programování)
<b>REST</b>	Representational State Transfer
<b>RPC</b>	Remote Procedure Call
<b>SOA</b>	Service Oriented Architecture
<b>SOAP</b>	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

-- 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