

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: Klimeš Jméno: Filip Osobní číslo: 406183
Fakulta/ústav: Fakulta elektrotechnická
Zadávající katedra/ústav: Katedra počítačů
Studijní program: Otevřená informatika
Studijní obor: Softwarové inženýrství

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

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

Název diplomové práce anglicky:

Central management and automatic integration of business rules in service oriented architecture

Pokyny pro vypracování:

- 1) Proveďte rešerší možnosti využití, vyjádření a znova použití byznys pravidel v architektuře orientované na služby a identifikujte potenciální problémy.
- 2) Prozkoumejte možnosti reprezentace byznys pravidel ve stávajících frameworkch, identifikujte jejich nedostatky a zhodnotěte jejich schopnost deduplikace a recyklace byznys pravidel.
- 3) Navrhněte efektivní způsob správy, distribuce a znovupoužití byznys pravidel v architektuře orientované na služby za použití aspektově orientovaného programování.
- 4) Funkčnost navrženého řešení demonstrejte implementací prototypu knihovny alespoň ve třech programovacích jazycích, přičemž jeden z nich musí být Java.
- 5) S využitím navržené knihovny implementujte ukázkovou e-commerce aplikaci tvořenou alespoň třemi službami napsanými v různých programovacích jazycích. Na této aplikaci ukažte a otestujte, že knihovna umožňuje
- 6) centrálně spravovat byznys pravidla v aplikaci
- 7) sdílet byznys pravidla mezi službami
- 8) integrovat sdílená byznys pravidla do jednotlivých služeb.
- 9) Změřte a analyzujte počet duplikací byznys pravidel v ukázkové aplikaci a diskutujte vliv na jejich údržbu oproti konvenčnímu přístupu.
- 10) Analyzujte další oblasti architektury orientované na služby, kde lze aplikovat aspektově orientované programování.

Seznam doporučené literatury:

- [1] CEMUS, Karel; CERNY, Tomas. Aspect-driven design of information systems. International Conference on Current Trends in Theory and Practice of Informatics. Springer, Cham, 2014. p. 174-186.
- [2] CEMUS, Karel, et al. Distributed Multi-Platform Context-Aware User Interface for Information Systems. In: IT Convergence and Security (ICITCS), 2016 6th International Conference on. IEEE, 2016. p. 1-4.
- [3] KENNARD, Richard; EDMONDS, Ernest; LEANEY, John. Separation anxiety: stresses of developing a modern day separable user interface. In: Human System Interactions, 2009. HSI'09. 2nd Conference on. IEEE, 2009. p. 228-235.
- [4] KICZALES, Gregor, et al. Aspect-oriented programming. ECOOP'97?Objectoriented programming, 1997, 220-242.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Karel Čemus, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **08.02.2018**

Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: **30.09.2019**

Ing. Karel Čemus
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

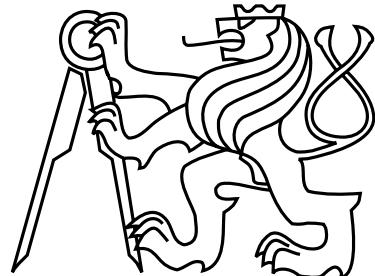
III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

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

23. května 2018

Poděkování

Chtěl bych poděkovat Ing. Karlovi Čemusovi za jeho trpělivost, podporu a cenné rady nejen při vedení této práce, ale po celou dobu mého studia. Děkuji své rodině, přítelkyni a přátelům za zázemí a podporu, kterou mi po dobu studia poskytovali a bez které bych ho nemohl dokončit. Děkuji také svým kolegům ve škole i v zaměstnání, kteří mě motivovali k dosažení vynikajících výsledků v průběhu magisterského studia.

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 23. 5. 2018

.....

Abstract

Service-oriented architecture decomposes information systems into small standalone services lowering their complexity through loose coupling and component reuse. Business rules often cross-cut multiple services and must be addressed consistently. Current approaches reach their limits when it comes to the business rules. They tend to require manual information restatement and code duplication, which raises development and maintenance costs and efforts.

This thesis analyses the problem of business rules in service-oriented architecture and proposes a novel approach that uses aspect-oriented programming to enable central administration and automatic integration of business rules in such systems. Moreover, the thesis provides a proof of concept showing reduction of manual duplication of business rules within an example system utilizing the proposed approach.

Keywords: Service-oriented architecture, enterprise information systems, business rules, aspect-oriented programming, separation of concerns

Abstrakt

Architektura orientovaná na služby člení funkcionalitu komplexních informačních systémů do samostatných služeb a díky tomu usnadňuje oddělení zodpovědností a zvyšuje znovupoužitelnost jednotlivých komponent. Byznysová pravidla ale často zasahují do více služeb najednou a vyžadují konzistentní vykonávání. To při použití současných přístupů vede k manuální duplikaci pravidel, která zvyšuje náklady na vývoj a údržbu systému.

Tato práce analyzuje problematiku byznysových pravidel v architektuře orientované na služby a navrhuje alternativní způsob, jakým lze s využitím aspektově orientovaného programování usnadnit práci vývojářů a administrátorů pomocí centrální správy a automatické integrace těchto pravidel. Součástí práce je implementace navrženého přístupu na ukázkovém příkladu, který demonstruje snížení manuální duplikace byznysových pravidel.

Klíčová slova: Architektura orientovaná na služby, podnikové informační systémy, byznysová pravidla, aspektově orientované programování

Obsah

1	Úvod	1
1.1	Struktura práce	2
2	Analýza problematiky	3
2.1	Byznysová pravidla	3
2.2	Architektura orientovaná na služby	6
2.3	Nedostatky současného přístupu	9
2.4	Identifikace požadavků na implementaci frameworku	10
2.5	Shrnutí	10
3	Rešerše existujících řešení	11
3.1	Modelem řízený vývoj	11
3.2	Generativní programování	12
3.3	Metaprogramování	12
3.4	Aspektově orientované programování	13
3.5	Stávající řešení správy a reprezentace business pravidel	17
3.6	Síťové architektury	19
3.7	Shrnutí	20
4	Návrh frameworku	21
4.1	Vize frameworku	21
4.2	Formalizace architektury orientované na služby	21
4.3	Popis byznysových kontextů	25
4.4	Metamodel byznysového kontextu	27
4.5	Organizace byznysových kontextů	28
4.6	Distribuce sdílených byznysových kontextů mezi službami	29
4.7	Inicializace byznysových kontextů	30
4.8	Centrální správa byznysových kontextů	31
4.9	Architektura frameworku	32
4.10	Shrnutí	34

5 Implementace prototypu frameworku	35
5.1 Výběr použitých platforem	35
5.2 Doménově specifický jazyk pro popis byznys kontextů	35
5.3 Distribuce byznysových kontextů mezi službami	40
5.4 Knihovna pro platformu Java	43
5.5 Knihovna pro platformu Python	45
5.6 Knihovna pro platformu Node.js	47
5.7 Systém pro centrální správu byznys pravidel	49
5.8 Shrnutí	50
6 Testování a validace frameworku	51
6.1 Techniky testování	51
6.2 Testování prototypů knihoven	52
6.3 Ukázkový příklad: e-commerce systém	56
6.4 Srovnání s konvenčním přístupem	65
6.5 Shrnutí	66
7 Závěr	67
7.1 Dosažené výsledky	67
7.2 Přínos a možnosti použití frameworku	68
7.3 Budoucí rozšiřitelnost frameworku	68
7.4 Další možnosti uplatnění AOP v SOA	69
A Seznam použitých zkratek	77
B Přehledové obrázky a snímky	79
C Obsah přiloženého CD	83

Seznam obrázků

2.1	Komunikace služeb pomocí Enterprise Service Bus	7
2.2	Porovnání orchestrace a choreografie služeb	8
2.3	Příklad funkcionality zasahující do více služeb	10
3.1	Průřezové problémy v informačních systémech	13
3.2	Proces weavingu aspektů	15
3.3	Architektura systému využívajícího ADDA	17
4.1	Diagram životního cyklu služby a identifikovaných join-pointů	22
4.2	Proces weavingu byznysových pravidel	24
4.3	Znázornění abstraktního byznysového kontextu	26
4.4	Metamodel byznysového kontextu	27
4.5	Postavení registrů byznysových kontextů vůči jednotlivým službám	29
4.6	Proces inicializace byznysových kontextů	30
4.7	Proces centrální správy byznysových kontextů	31
4.8	Architektura navrženého frameworku	33
5.1	Použití vzoru Interpreter pro vyhodnocování logických výrazů	37
5.2	Syntaktický strom jednoduchého validačního pravidla	39
5.3	Využití vzoru Visitor pro převod logických výrazů do DSL	40
5.4	Moduly prototypu knihovny pro jazyk Java a jejich závislosti	43
6.1	Doménový model ukázkového e-commerce systému	58
6.2	Komponenty ukázkového e-commerce systému	61
B.1	Detail byznysového kontextu v centrální administraci	79
B.2	Formulář pro vytvoření nebo úpravu byznysového kontextu v centrální administraci	80
B.3	Propagace byznysového pravidla při přidávání produktu do košíku v ukázkovém systému	80
B.4	Diagram hierarchie byznysových kontextů ukázkového systému	81

Seznam tabulek

5.1	Přehled výrazů pro zápis byznysového pravidla	38
6.1	Testovací scénáře prototypů knihoven	53
6.2	Případy použití ukázkového e-commerce systému	57
6.3	Byznysová pravidla ukázkového e-commerce systému	59
6.4	Byznysové kontexty ukázkového e-commerce systému	60
6.5	Využití byznysových pravidel ve službách ukázkového systému	65

Seznam zdrojových kódů

3.1	Příklad průřezových problémů zohledněných při vytváření objednávky	14
4.1	Ukázka zápisu validačních pravidel pomocí anotací v jazyku Java	22
4.2	Příklad provázání byznysové operace a byznysového kontextu pomocí anotací v jazyce Java	23
5.1	Příklad zápisu byznys kontextu v jazyce XML	36
5.2	Část definice schématu zpráv byznys kontextů ve formátu Protocol Buffers . .	41
5.3	Definice služby pro komunikaci byznys kontextů pro gRPC	42
5.4	Označení operačního kontextu a jeho parametrů pomocí anotací Java knihovny	44
5.5	Příklad použití dekorátorů pro weaving v jazyce Python	46
5.6	Příklad dekorace funkce v JavaScriptu pro aplikaci weavingu	47
6.1	Jednotkový test knihovny pro jazyk Java s využitím nástroje JUnit 4	54
6.2	Jednotkový test knihovny pro jazyk Python s využitím nástroje Unittest . .	55
6.3	Jednotkový test knihovny pro platformu Node.js s využitím nástroje Mocha a Chai	56
6.4	Ukázka využití frameworku Spring Boot pro účely Order service	61
6.5	Ukázka využití frameworku Flask pro účely Product service	62
6.6	Ukázka využití frameworku Express.js pro účely User service	63
6.7	Ukázka zápisu aplikace s více kontejnery pro Docker Compose	64

Kapitola 1

Úvod

Informační systémy se ve 21. století staly neodmyslitelnou součástí našich každodenních životů. Do styku s nimi přicházíme jak při výkonu našich povolání, tak ve volném čase. Usnadňují řadu našich činností, od vzdělání a vědy, kde umožňují snadný přístup ke studijním materiálům, přes zdravotnictví, kde pomáhají zvyšovat efektivitu a úroveň péče o pacienty, až po sociální síť, kde umožňují lidem globálně komunikovat a sdílet své myšlenky, pocity a zážitky. Očekávání na kvalitu a množství funkcí informačních systémů se zvyšuje každým rokem. Jedním z úkolů výzkumu v oblasti softwarového inženýrství je zjednodušení a zefektivnění vývoje informačních systémů. Díky tomu budou tyto systémy moci splňovat stále rostoucí množství požadavků.

Náročnost vývoje některých informačních systémů překračuje možnosti jednotlivců, ale i celých týmů či skupin. Tyto systémy často využívají větší počet různorodých technologií kvůli širokému spektru funkcionality, kterou nabízejí. Jedním z přístupů, který tyto problémy řeší, je použití architektury orientované na služby. Ta se zaměřuje na sestavení systému z menších, vzájemně nezávislých celků, tzv. služeb. Každá služba pak zastřešuje pouze část funkcionality systému. Tím je umožněno využívat teoreticky neomezené množství technologií a rozdělit práci na systému mezi více nezávislých vývojářských týmu.

Tato architektura bohužel nepřináší odpověď na všechny problémy, které je potřeba v informačních systémech řešit. Jak je popsáno v následujících kapitolách, jedním z těchto problémů jsou tzv. byznysová pravidla. Ta slouží k zajištění správné funkcionality systému a konzistenci uložených dat. Některá tato pravidla zasahují do celého systému, tedy i do více služeb. To při použití konvenčního přístupu přináší nutnost manuální duplikace zdrojového kódu a tím jsou zvýšeny náklady na vývoj systému a riziko lidské chyby.

Popsáním byznysových pravidel na jednom místě a jejich následnou automatickou distribucí a integrací do služeb systému by mohly být sníženy náklady na vývoj a údržbu informačních systémů. Cílem této práce je prozkoumat myšlenku inovativního přístupu k

centrální správě a automatické integraci byznysových pravidel v systémech využívajících architekturu orientovanou na služby a navrhnout framework, který by umožnil uplatnit tento koncept v praxi. Dílčí cíle práce jsou:

1. Analyzovat problematiku byznysových pravidel v architektuře orientované na služby a provést rešerší stávajících řešení
2. Navrhnut framework, který umožní centrální správu a automatickou distribuci byznysových pravidel v architektuře orientované na služby
3. Implementovat a otestovat prototypy knihoven navrženého frameworku pro tři rozdílné platformy a vyhodnotit dopad jejich použití na ukázkovém příkladu

1.1 Struktura práce

Kapitola 2 se věnuje 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 framework pro centrální správu a automatickou integraci byznysových pravidel v této architektuře. Kapitola 3 se zabývá rešerší stávajících přístupů k vývoji informačních systémů a zaměřuje se zejména na koncepty aspektově orientovaného programování a moderního aspekty řízeného přístupu k návrhu systémů. Dále se kapitola věnuje průzkumu existujících nástrojů pro správu byznysových pravidel a existujícím síťovým architekturám, které slouží pro distribuci byznysových pravidel mezi službami. 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 integraci byznysových pravidel. V kapitole 5 je detailně probrána implementace knihoven pro navržený framework na platformách jazyků Java a Python a frameworku Node.js. Následující kapitola 6 popisuje, jakým způsobem byly tyto knihovny otestovány. 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 dosáhнуто a jakým dalším směrem se může výzkum v této oblasti ubírat.

Kapitola 2

Analýza problematiky

Tato kapitola analyzuje problematiku byznysových pravidel v informačních systémech a detailně popisuje architekturu orientovanou na služby, včetně jejího historického vývoje a moderních trendů. Na základě toho kapitola identifikuje nedostatky současných přístupů při sdílení byznysových pravidel. V závěru kapitoly jsou specifikovány požadavky, které by měl splňovat framework, jež bude výstupem této diplomové práce.

2.1 Byznysová pravidla

Podnikové informační systémy ([EIS](#) z anglického *Enterprise Information System*) mají za úkol ulehčit, automatizovat či poskytovat podporu pro byznysové procesy organizace, která je využívá [\[21\]](#). Byznysové procesy jsou souborem dílčích úkolů a aktivit, které naplňují cíle přinášející organizaci byznysovou hodnotu [\[66\]](#). Jednotlivé kroky byznysového procesu jsou nazývány byznysové operace. Byznysové procesy a operace dohromady tvoří tzv. byznysovou logiku, která podléhá byznysovým pravidlům. Ta zajišťují konzistenci dat systému a zabraňují nepovoleným operacím [\[12\]](#).

Definice. Byznysové pravidlo je logická rozhodovací podmínka, která definuje či omezuje byznysové chování informačního systému, je specifické pro konkrétní byznysovou doménu, a je atomické, tj. nelze dále rozdělit na dílčí pravidla [\[12, 45\]](#).

[EIS](#) obsahuje mnoho byznysových pravidel, typicky stovky či tisíce [\[45\]](#). Samotné pravidlo však bývá relativně krátké a lze shrnout do jedné věty. Díky tomu je pochopitelné pro všechny zainteresované strany, které se podílejí na návrhu a vývoji systému. Byznysová pravidla jsou podle jedné z klasifikací dělena do tří skupin [\[10\]](#):

Bezkontextová pravidla jsou validační pravidla, která musejí být obecně platná v každé byznysové operaci, jinak by mohlo dojít k porušení integrity dat systému. Příkladem může být pravidlo „*Adresa uživatele je platnou e-mailovou adresou*“.

Kontextová pravidla jsou pravidla, která musejí být zohledněna v daném kontextu byznysové operace, například „*Při přidání produktu do košíku nesmí součet položek v košíku přesahovat částku milion korun*“

Průřezová pravidla jsou parametrizována stavem systému nebo uživatelského účtu a mají dopad na velkou část byznysových operací, například pravidlo „*V systému nesmí probíhat žádné změny po dobu účetní uzávěrky*“.

Pravidla lze také rozdělit do dvou skupin podle jejich vztahu k byznysové operaci, a těmi jsou *preconditions* a *post-conditions* [12, 45].

2.1.1 Precondition

Aby mohla být byznysová operace vykonána, musejí být splněny předem definované podmínky, neboli předpoklady, které nazýváme *preconditions*. Pokud alespoň jedna z podmínek není splněna, byznysová operace nemůže proběhnout [38]. Například při registraci uživatele musí být splněna podmínka, že uživatel vyplnil svojí emailovou adresu, a zároveň dosud v systému neexistuje žádný uživatel se stejnou emailovou adresou.

2.1.2 Post-condition

Na byznysovou operaci mohou být kladený požadavky, které musejí být splněny po jejím úspěšném vykonání [12]. Příkladem může být anonymizace uživatelů při vytváření statistického reportu e-commerce společnosti, který nesmí obsahovat citlivé údaje zákazníků. Dalším případem může být filtrování výstupu byznysové operace, například při výpisu objednávek pro zákazníka musí všechny vypsané objednávky patřit danému zákazníkovi.

2.1.3 Byznysový kontext

Jak již bylo uvedeno, vykonávání byznysové operace je podmíněno byznysovými pravidly, která se k ní vztahují. Před spuštěním operace musejí být splněny všechny preconditions, jinak není možné operaci vykonat. Po jejím dokončení musejí být splněny všechny post-conditions. Aby EIS mohl tyto podmínky ověřit dynamicky za běhu systému, využívá tzv. *execuční kontext* (z anglického *execution context*), který se skládá z několika dílčích kontextů [13]:

Aplikační kontext drží stav globálních proměnných systému, jako například stav databáze, nastavení produkčního režimu, nebo příznak o tom, zda právě probíhá obchodní uzávěrka.

Uživatelský kontext obsahuje informace o aktuálně přihlášeném uživateli.

Kontext požadavku (z anglického *Request context*) se váže zejména na webové služby a obsahuje informace o aktuálním požadavku, jako je IP adresa či geolokace uživatele.

Byznysový kontext obsahuje informace o probíhající byznysové operaci včetně byznysových pravidel.

Byznysový kontext je tedy důležitým prvkem při výkonu byznysové operace, který umožňuje vyhodnocování byznysových pravidel. Všechny proměnné, které jsou dostupné v exekučním kontextu, jsou dostupné při vyhodnocování pravidel. Díky tomu je možné definovat široké spektrum logických podmínek, které se mohou přizpůsobit aktuálnímu stavu systému.

Definice. Byznysový kontext je množina preconditions a post-conditions, která se váže na konkrétní byznysovou operaci [12].

2.1.4 Reprezentace byznysového pravidla

Existuje několik možností, jak v rámci **EIS** zachytit a reprezentovat byznysová pravidla [12]. Nejběžnější metodou zachycení byznysových pravidel je zápis v obecném programovacím jazyce. Ačkoliv většina současných obecných programovacích jazyků poskytuje reflexi, nepřináší dostatečné možnosti inspekce a extrakce pravidel pro jejich další využití. Tento způsob však umožňuje snadno definovat i velmi komplikovaná pravidla.

Pokročilejší metodou je pro zachycení pravidel pomocí meta-instrukcí. Těmi mohou být například anotace, případně speciální jazyk, tzv. *Expression Language* [47]. Tento způsob poskytuje dostatečnou možnost inspekce, ale není typově bezpečný a kvůli tomu je náchylnější na chybu. Anotace je navíc potřeba vázat na existující komponenty systému, což může být pro některá byznysová pravidla nevhodné [12]. Příkladem je průřezové pravidlo, které zamyká systém pro úpravy po dobu účetní uzávěrky. Takové pravidlo se nevztahuje k žádné konkrétní komponentě systému a musí tak být zachyceno v každé komponentě zvlášť.

Nejpokročilejší metodou zachycení pravidla je využití **DSL**, které jsou snadno srozumitelné a poskytují snadnou inspekci a typovou bezpečnost. Mezi jejich nevýhody patří vysoká počáteční investice v podobě návrhu jazyka, nutnost jeho komplikace nebo interpretace a také proškolení personálu, který s ním bude pracovat. Kvůli tomu může být vhodné využít existující řešení [12], jako například formální jazyk *Object Constraint Language* [65]. Ten umožňuje snadné automatické zpracování a transformaci, ale i formální matematické operace nad jeho výrazy. Jeho nevýhodou je pracnost zápisu. Využít lze i některé z průmyslových řešení, jako je framework Drools¹, který je popsaný v sekci 3.5.2.

¹<<https://www.drools.org/>>

2.2 Architektura orientovaná na služby

Architektura orientovaná na služby (SOA) je odpověď na stále se zvyšující nároky na informační systémy a jejich rostoucí velikost a komplexitu. Na rozdíl od *monolitické architektury*, dělí SOA systém na samostatné nezávislé celky, zvané *služby*, které poskytují dílčí části požadované funkcionality. Historicky byl termín SOA vykládán několika způsoby a představoval několik rozdílných, nekompatibilních konceptů [25]. Absence kvalitních definic služby a obecně SOA vedla v poslední době i ke snahám o opuštění tohoto konceptu [15]. Pro lepší porozumění se tato kapitola věnuje stručnému historickému přehledu SOA a shrnuje vlastnosti a z nich vyplývající výhody a nevýhody jednotlivých přístupů.

Definice. Služba je znovupoužitelný, soudržný, spravovatelný, nasaditelný a nezávislý proces komunikující s ostatními službami pomocí zpráv [20, 48].

2.2.1 Common Object Request Broker Architecture

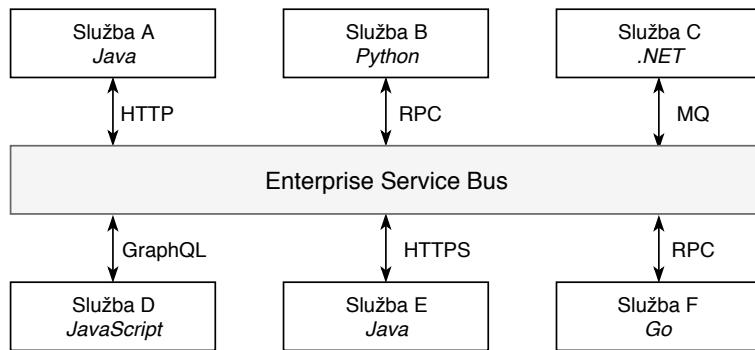
Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture (CORBA)* [56]. Ta umožňuje vzájemnou komunikaci aplikací implementovaných v různých technologiích. Její základní komponentou je *Object Request Broker (ORB)*, který simuluje vzdálené objekty, na kterých může klient volat jejich metody. Při zavolání metody na objektu, který se fyzicky nachází na vzdáleném stroji, zprostředkovává ORB veškerou komunikaci a poskytuje kompletní rozhraní volaného objektu. Stejný způsob komunikace s lokálním jako se vzdáleným objektem je však problematický kvůli nižší robustnosti a vyšší latenci síťového přenosu.

2.2.2 Web Services

Nedostatky architektury CORBA vedly k vývoji jednoduššího a kvalitnějšího formátu pro popis komunikace služeb. Volání metod na vzdálených objektech bylo nahrazeno explizitním posíláním zpráv mezi službami pomocí protokolu *HTTP*. Pro popis schématu zpráv vznikl formát *Simple Object Access Protocol* [6], který v kombinaci s *Web Service Description Language* [18] umožňuje kompletní definici rozhraní pro komunikaci mezi službami.

2.2.3 Message Queue

Dalším z konceptů, který v rámci SOA vznikl, je tzv. *Message Queue (MQ)*. Jeho základní myšlenkou je asynchronní komunikace služeb pomocí zpráv nezávislých na platformě. Komunikaci zprostředkovává fronta, která přijímá a rozesílá zprávy mezi službami. To přináší vyšší škálovatelnost a menší provázanost mezi službami. Všechny služby ale musí používat jednotný formát zpráv.



Obrázek 2.1: Komunikace služeb pomocí Enterprise Service Bus

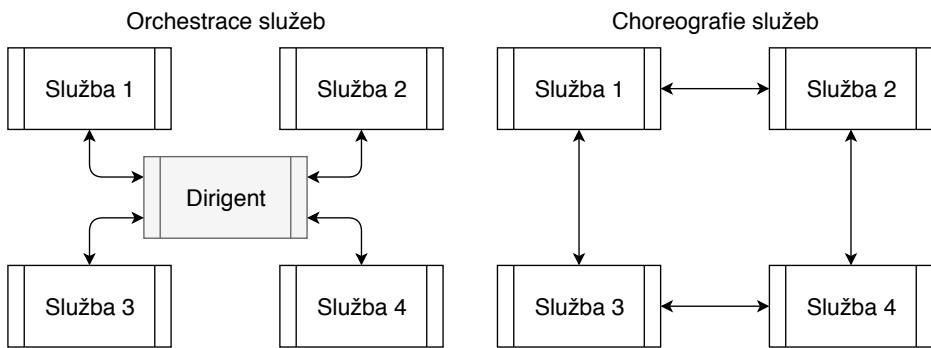
2.2.4 Enterprise Service Bus

Ačkoliv zmíněné modely usnadňují komunikaci služeb a zvyšují jejich spolehlivost, integrace služeb může být obtížná, pokud služby používají navzájem různé komunikační protokoly a formáty. Tento problém řeší *Enterprise Service Bus* (ESB) [17], znázorněný na obrázku 2.1, který má za úkol propojit heterogenní služby využívající různé technologie a sestavit mezi nimi komunikační kanály. Tím na sebe ESB přebírá zodpovědnost za překlad jednotlivých zpráv a centralizuje veškerou komunikaci v systému. Kvůli tomu musí poskytovat routování síťových zpráv a překlad mezi jednotlivými formáty. ESB navíc umožňuje zabezpečení komunikace a může sloužit i pro orchestraci služeb, jak je popsáno v sekci 2.2.6.

2.2.5 Microservices

Microservices je moderní architekturou, která podobně jako SOA přináší řešení problémů pramenících z vysoké komplexity současných EIS. Tato architektura se dá chápat jako podmnožina SOA [15, 52], ačkoliv existují i názory, že jde o odlišné architektury. V poslední době postupně nahrazuje standardní SOA [39, 67] a vzhledem k její vzrůstající adopci je nutno ji v rámci této práce zohlednit.

Její základní myšlenkou je vývoj informačního systému jako množiny malých oddělených služeb, které jsou spouštěny v samostatných procesech a komunikují spolu pomocí jednoduchých protokolů nezávislých na platformě [39]. Microservices preferuje decentralizaci a samostatnost služeb a zaměřuje se na jejich organizaci kolem byznysových schopností systému, namísto horizontálního dělení systému podle jeho vrstev. Hlavní výhodou tohoto přístupu je flexibilita nasazení a škálování, která je vhodná pro stále populárnější nasazení v cloudu [16, 36, 67].



Obrázek 2.2: Porovnání orchestrace a choreografie služeb

2.2.6 Orchestrace a choreografie služeb

Základní podmínkou pro funkci systému využívajícího **SOA** je komunikace a spolupráce jednotlivých služeb. K tomu slouží principy *orchestrace služeb* a *choreografie služeb*. Porovnání obou přístupů je graficky znázorněno na obrázku 2.2.

Orchestrace Orchestrace služeb má za úkol zajistit, že komunikace mezi službami proběhne úspěšně a ve správném časovém sledu [61], za použití centrální komponenty – tzv. *dirigenta*. Typicky je dirigent implementován jako součást **ESB**. Orchestrace implikuje centrální řízení a koordinaci služeb při vykonávání byznysových procesů.

Choreografie Opakem orchestrace je tzv. *choreografie služeb* a znamená vykonávání byznysových operací autonomně a asynchronně, bez centrální autority. Choreografie popisuje zejména pravidla a podobu komunikace jednotlivých služeb. K tomu zpravidla využívá **MQ** a zasílání zpráv o událostech v systému. Každá služba při vykonávání byznysových operací plní svoji roli bez centrálního řízení [15]. Tento přístup je využíván zejména v rámci Microservices [20], protože umožňuje nižší provázání služeb a rovnoměrnější rozložení zodpovědností v systémů.

2.2.7 Shrnutí

Z předchozího textu vyplývá, že přístupy k realizaci **SOA** vychází ze společné myšlenky členění systémů do dílčích izolovaných služeb poskytujících byznysovou funkcionalitu. Přístupy se liší zejména v řešení komunikace služeb a v centralizaci jejich správy. Historické přístupy využívají komplexní komunikační technologie a umožňují centrální správu a orchestraci systému, zatímco moderní přístup Microservices se zaměřuje zejména na nízké provázání služeb a využití jednoduchých komunikačních kanálů. To přináší výzvu při sdílení byznysových pravidel, která je rozebrána v následující sekci.

Definice. SOA je soubor návrhových principů, který organizuje komponenty software kolem byznysové funkcionality a spojuje je pomocí rozhraní a komunikačních protokolů. Každá komponenta je soběstačná a izolovaná, okolnímu světu poskytuje pouze své komunikační rozhraní [39, 48].

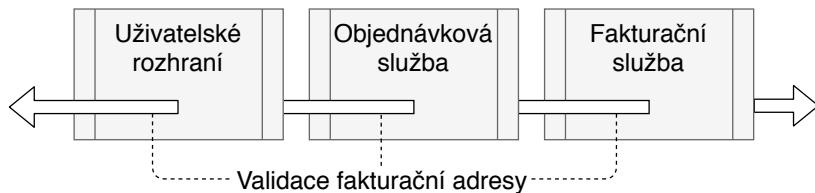
2.3 Nedostatky současného přístupu

Některá složitější byznysová funkctionalita vyžaduje spolupráci více služeb najednou [48]. K tomuto účelu slouží tzv. kompozitní služby, které využívají funkctionalitu ostatních služeb. Při kompozici služeb je nutné zohlednit jejich byznysová pravidla, aby bylo zabráněno nekonzistentním stavům v systému a zbytečným spouštěním byznysových operací, jejichž preconditions nejsou splněny [15]. To je však v přímém rozporu s požadavkem na nízkou provázanost služeb, které by neměly vzájemně znát svoji interní strukturu. Tato skutečnost vede k nutnosti duplikace byznysových pravidel v kompozitních službách [14].

Definice. Kompozitní služba získává a kombinuje informace a funkce z ostatních služeb [48].

Pro lepší představu je problém znázorněn na obrázku 2.3. Uvažme e-commerce systém skládající se z několika služeb naprogramovaných v různých technologiích, a procesy vytváření faktury a vytváření objednávky. Při dokončení objednávky je potřeba vytvořit fakturu, objednávková služba k tomu využívá fakturační službu. Podmínkou pro vytvoření faktury je platná fakturační adresa, která je zadána již při vytváření objednávky. Nevalidní adresa by však při vytváření faktury mohla způsobit řadu problémů vyžadujících manuální řešení. Proto musí být tato adresa validována již při vytváření objednávky. V ideálním případě by navíc měl zákazník být upozorněn na nevalidní fakturační adresu co nejdříve, ještě před odesláním objednávkového formuláře, přímo v uživatelském rozhraní [13]. To bývá často implementováno samostatnou kompozitní službou, která agreguje funkctionalitu celého systému.

Na příkladu lze pozorovat, že jedno byznysové pravidlo se promítá do tří služeb, z nichž každá má zodpovědnost za jiné byznysové operace. Stejná logika, která realizuje validaci fakturační adresy, musí být implementována v každé ze zmínovaných služeb, navíc v různých technologiích. Pokud by vzešel změnový požadavek na validaci fakturační adresy, změnu by bylo nutno provést konzistentně na třech různých místech, všechny tři služby znova sestavit a nasadit ve správném pořadí tak, aby nedošlo k nekonzistentní validaci adresy při provádění jednotlivých byznysových operací. Změny byznysových pravidel se dělí častěji, než změny kódu a struktury samotných služeb v SOA [53]. Pokud je potřeba s každou změnou byznysového pravidla sestavit a nasadit jednu či více služeb, dramaticky se zvyšuje náročnost na údržbu takového systému.



Obrázek 2.3: Příklad funkcionality zasahující do více služeb

2.4 Identifikace požadavků na implementaci frameworku

Pro usnadnění vývoje a údržby systému stavějícího na **SOA**, který obsahuje kompozitní služby, je nutné umožnit sdílení byznysových pravidel. Ta by měla být zachycena mimo samotnou implementaci služby, ideálně ve formátu, který bude nezávislý na konkrétní platformě, bude poskytovat možnost automatické inspekce a bude srozumitelný doménovým expertům. Framework nesmí vynucovat manuální duplikaci byznysových pravidel. Díky tomu framework sníží náklady na vývoj a údržbu systému a riziko lidské chyby. Úprava pravidel navíc nesmí vyžadovat změnu kódu služby a její opětovné nasazování. Administrátoři systému by měli mít možnost byznysová pravidla spravovat centrálně a bez přerušení provozu systému, aby mohli co nejrychleji a flexibilně reagovat na změnové požadavky. Vzhledem k různému chápání **SOA** a postupné adopci Microservices framework nesmí klást nároky na způsob organizace služeb, tj. měl by umožňovat orchestraci i choreografii.

Framework, který bude výstupem této práce, musí splňovat následující vlastnosti:

- Možnost definovat byznysová pravidla pomocí platformově nezávislého **DSL** srozumitelného pro doménové experty
- Možnost centrálně spravovat byznysová pravidla, včetně úpravy stávajících a vytváření nových dynamicky za běhu systému
- Automatická distribuce a integrace byznysových pravidel včetně vyhodnocování pre-conditions a aplikace post-conditions
- Možnost využívat framework na více plafomrých
- Nezávislost na organizaci služeb

2.5 Shrnutí

V této kapitole byla provedena analýza byznysových pravidel a architektury orientované na služby. Na základě toho byly popsány nedostatky **SOA** při kompozici služeb a sdílení byznysových pravidel. Nakonec byly identifikovány požadavky, které by měl splňovat framework, který bude výstupem této práce.

Kapitola 3

Rešerše existujících řešení

Tato kapitola se věnuje rešerši existujících řešení a výzkumu relevantnímu k tématu této práce. Provedená rešerše je rozdělena do následujících částí:

1. Architektury a paradigmata umožňující snížení nákladů na vývoj software
2. Stávající řešení správy a reprezentace byznysových pravidel
3. Síťové architektury umožňující sdílení a distribuci byznysových pravidel mezi službami

Díky provedené rešerši bude možno dosáhnout kvalitního a efektivního návrhu frameworku pro centrální správu a automatickou distribuci byznysových pravidel.

3.1 Modelem řízený vývoj

Modelem řízený vývoj ([MDD](#) z anglického *Model-Driven Development*) se zaměřuje na návrh [EIS](#) s využitím modelů a jejich následnou transformaci do spustitelného kódu pomocí nástrojů pro jeho automatickou generaci [54, 58]. Hlavní výhodou [MDD](#) je vysoká úroveň abstrakce, která snižuje potřebu manuální duplikace informací napříč systémem. Další výhodou je možnost sdílení modelů mezi více platformami.

[MDD](#) využívá více druhů modelů a rozděluje je podle jejich úrovně abstrakce. Při vývoji se pak postupuje od nejabstraktnějšího modelu postupným přidáváním detailů. V první fázi vývoje je využit Computation Independent Model ([CIM](#)), který reprezentuje řešení nezávislé na použitých výpočetních metodách a algoritmech. Z [CIM](#) je následně model převeden do Platform Independent Model ([PIM](#)), který popisuje koncepci systému bez ohledu na implementační detaily a typicky využívá jazyk [UML](#). [PIM](#) je následně převeden do Platform Specific Model ([PSM](#)), tedy do modelu využívajícího specifických aspektů platformy, pro

kterou má být systém postaven. **PIM** může být převeden do více **PSM**, pokud má být výsledný systém využíván pro více platform. Nakonec je **PSM** transformován do spustitelného kódu [35]. Převod mezi jednotlivými modely bývá automatizován, ale zejména poslední krok často vyžaduje manuální zásah.

Hlavní nevýhodou **MDD**, která zabraňuje jeho využití pro účel této práce, je jednosměrný dopředný proces, kterým je výsledný kód generován. Pokud dojde ke změně v některém z modelů, je potřeba znova vygenerovat všechny navazující modely a zdrojový kód. Manuálně vytvořený nebo upravený kód je potřeba doplnit znova. Navíc je tento přístup odvozen od **OOP**, které není schopné se efektivně vypořádat s průřezovými problémy [10, 33], jak bude naznačeno v sekci 3.4.

3.2 Generativní programování

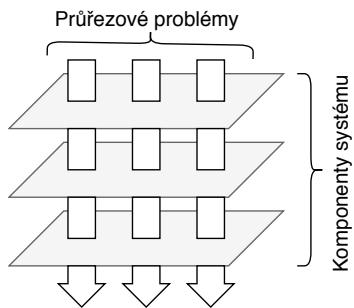
Generativní programování (**GP**) je programovací paradigma, které je podobně jako tato práce motivováno snížením nákladů na vývoj software díky zvýšení znovupoužitelnosti jeho komponent. Na rozdíl od **MDD** se navíc přímo věnuje oddělení zodpovědností [13, 19].

GP se zaměřuje na maximalizaci automatizace vývoje systému skrz generování a syntézu vysoce přizpůsobitelných komponent. Vývojář popíše komponentu v abstraktním jazyce přizpůsobeném doméně řešeného problému a generátor se postará o její automatické vytvoření [19]. Díky tomu je možné oddělit popis jednotlivých vlastností systému a dosáhnout tak jejich vysoké znovupoužitelnosti.

GP by tak mohlo být využito pro abstrakci bynysových pravidel a jejich automatickému začleňování do služeb v systému stavějícím na **SOA**. Statické generování pro každý z možných stavů systému je však neúnosné. Tento přístup proto neumožňuje zohledňovat exekuční, potažmo byznysový kontext aplikace.

3.3 Metaprogramování

Metaprogramování je alternativním paradigmatem, který vnímá kód programu zároveň jako data, se kterými program může pracovat. To mu umožňuje čist, vytvářet či upravovat jiné programy včetně sama sebe. Tyto činnosti lze provádět staticky, ale i za běhu daného programu [19, 55]. Schopnost jazyka manipulovat s vlastním kódem se nazývá *reflexe* [57]. Ta je součástí mnoha moderních programovacích jazyků a je využívána moderními frameworky a knihovnami [23, 62]. Tento přístup přináší vysokou úroveň abstrakce a zvýšenou efektivitu vývojářů, kteří jsou schopni automaticky provádět inspekci, generovat a upravovat programy [55].



Obrázek 3.1: Průřezové problémy v informačních systémech

Jak již bylo zmiňováno v sekci 2.1.4, zachycení byznysových pravidel v obecném programovacím jazyce neposkytuje dostatečné možnosti pro jejich extrakci a sdílení. Metaprogramování ale umožňuje získat informace o struktuře programu a identifikovat místa, ve kterých mají být byznysová pravidla aplikována. Využitím tohoto přístupu ve spojení s aspektově orientovaným programováním, které bude popsáno v následující sekci, je možné realizovat automatickou integraci sdílených byznysových pravidel.

3.4 Aspektově orientované programování

Při implementaci informačních systémů se osvědčilo členit systém do komponent, které zapouzdřují funkcionality a umožňují její snadné znovupoužití. Na tomto konceptu staví velmi rozšířené objektově-orientované programování (OOP) [51], které v současné době dominuje návrh a vývoj EIS. Tento přístup ale neumožňuje se efektivně vypořádat s některými požadavky. Těmi jsou tzv. *průřezové problémy* (z anglického *cross-cutting concerns*), které typicky ovlivňují více komponent systému, kde vyžadují konzistentní zpracování. Kvůli izolaci komponent je programátor nucen manuálně opakovat kód, který zodpovídá za realizaci průřezové funkcionality. Duplikace kódu vede k větší náchylnosti na lidskou chybu a k vyšším nárokům na vývoj a údržbu daného softwarového systému [27]. Obrázek 3.1 znázorňuje vzájemné postavení průřezových problémů a komponent informačního systému.

Definice. Průřezový problém je vlastnost systému, která ovlivňuje více jeho komponent zásahem do jejich funkcionality [34].

Příkladem průřezového problému může být logování systémových akcí, optimalizace správy paměti nebo jednotné zpracování výjimek [34], ale i aplikace byznysových pravidel [10]. Ve zdrojovém kódu 3.1 je znázorněno, jak průřezové problémy zasahují do kódu metody imaginární třídy implementované v jazyce Java, která slouží pro vytváření objednávek v e-commerce systému popsaném v sekci 2.3. Logování akcí je zohledněno v celém systému stejně

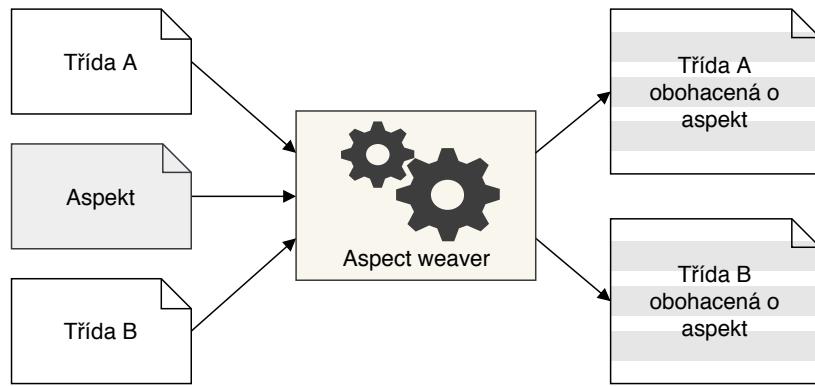
Zdrojový kód 3.1: Příklad průřezových problémů zohledněných při vytváření objednávky

```
1 void createOrder(User user, Collection<Product> products,
2                   Address shipping, Address billing) {
3     logger.info("Creating order"); // Logging aspect
4     transaction.begin(); // Transaction aspect
5     try {
6         validator.validateAddress(shipping); // Shipping business rule
7         validator.validateAddress(billing); // Billing business rule
8         Order order = new Order(user, product, shipping, billing);
9         database.save(order);
10        transaction.commit(); // Transaction aspect
11        logger.info("Order created successfully"); // Logging aspect
12    } catch (Exception e) {
13        transaction.rollback(); // Transaction aspect
14        logger.error("Could not create order"); // Logging aspect
15    }
16 }
```

jako při vytváření objednávky, liší se pouze text logovacích hlášek. Aspekt transakce je stejným způsobem zachycen v každé třídě, která implementuje byznysovou operaci pracující s databází. Byznysové pravidlo validující doručovací a fakturační adresu musí být aplikováno i v metodách implementujících úpravu objednávky, vystavení faktury a logistiku objednávek. Konvenční přístup vyžaduje manuální duplikaci logiky těchto aspektů.

Aspektově orientované programování ([AOP](#)) přináší řešení výše zmiňovaných problémů. Využívá k tomu princip oddělení zodpovědností (z anglického *separation of concerns*) – extra-huji logiku zachycující průřezové problémy do jednoho bodu, tzv. (*single focal point*). Pomocí procesu zvaného *weaving* je poté tato logika automaticky integrována. Weaving může proběhnout staticky při komplikaci programu nebo dynamicky při jeho běhu. V obou případech ale programátorovi ulehčuje práci, protože k definici i změně aspektu dochází na jednom místě, a tím je eliminována potřeba manuální duplikace a synchronizace kódu. V uvedeném případě by tedy byly aspekty logování, transakcí a byznysových pravidel zachyceny mimo metodu `createOrder`, a pomocí weavingu by byly do této metody automaticky integrovány.

Základním pojmem v rámci [AOP](#) je *aspekt*, který zapouzdřuje průřezovou funkcionalitu a zároveň adresuje místa, kde má být funkctionalita aplikována. Aspekt vždy obsahuje alespoň jeden *advice* a jeden *pointcut*.



Obrázek 3.2: Proces weavingu aspektů

Místo v kódu, na které může být aplikována funkcionálnita aspektu, se nazývá *join-point*. Typů join-pointů je více a závisí na použitém paradigmatu, na který je AOP aplikováno, a také na programovacím jazyce. V případě kombinace s OOP a klasickým víceúčelovým jazykem, jako je například Java, mohou jako join-pointy sloužit konstruktory tříd, volání metod, zápis a čtení z atributu objektu, inicializace třídy nebo objektu a mnoho dalších [37].

Pointcut vybírá podmnožinu join-pointů, na které je aplikován konkrétní aspekt. Tato podmnožina může být určena staticky, a být tak známá při komplikaci programu, nebo dynamicky za běhu programu, což přináší výpočetní složitost navíc výměnou za vyšší flexibilitu.

Funkcionálnita, kterou aspekt přidává v konkrétním join-pointu, se nazývá *advice*. Existuje více typů advice, podle toho, kam je daná funkcionálnita přidána. Například při volání metody může být funkcionálnita přidána před, za, nebo místo metody.

Proces, kterým jsou advice začleňovány podle pointcutu do jednotlivých join-pointů se nazývá *weaving*. Ten může probíhat již při komplikaci nebo dynamicky za běhu programu, tzv. *run-time weaving*. Proces weavingu je ilustrován na obrázku 3.2. Komponenta zodpovědná za weaving se nazývá *aspect weaver*. Pro weaving je často využito metaprogramování, která bylo popsáno v předchozí sekci 3.3.

3.4.1 Aspect-driven Design Approach

Alternativním způsobem návrhu informačních systémů, který staví na principech AOP, je Aspect-Driven Design Approach¹ (ADDA) [10]. Tento přístup se zaměřuje na identifikování aspektů v informačních systémech, jejich separaci do *single focal point* a využití weavingu pro jejich automatickou distribuci. K popisu aspektu přístup doporučuje využití doménově specifického jazyka (DSL), který je navržen na míru danému průřezovému problému.

¹Autoři historicky používali termín *Aspect-Oriented Design Approach* (AODA), který byl později změněn. Oba tyto názvy jsou vzájemně zaměnitelné.

3.4.2 Možnosti aplikace

Autoři **ADDA** aplikovali tento koncept v několika oblastech **EIS**. Mezi tyto oblasti patří automatické začleňování byznysových pravidel do datové vrstvy informačních systémů [12], automatické generování uživatelských rozhraní citlivých na kontext uživatele [13], validaci vstupů formulářů v uživatelském rozhraní vůči byznysovým pravidlům [9, 13] a automatické extrakci dokumentace [11].

Jednou z možných aplikací přístupu **ADDA** je automatické začleňování byznysových pravidel do datové vrstvy **EIS**². Byznysová pravidla jsou nejprve popsána pomocí **DSL** a následně jsou extrahována do jednoho bodu, ze kterého jsou automaticky distribuována. Pomocí specializovaného weaveru jsou pravidla překládána do podmínek jazyka **JPQL**, potažmo **SQL**, který je využíván k získávání dat z databázových systémů. To vede ke snížení manuální duplikace byznysových pravidel.

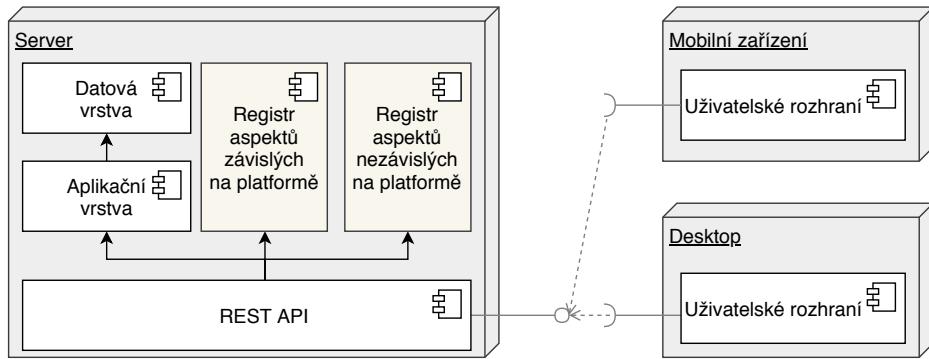
Definice. **DSL** je programovací jazyk určený k popisu specifické vlastnosti či funkce systému v rámci dané domény [26].

Další oblastí, kde je možné tento přístup aplikovat, jsou uživatelská rozhraní, která tvoří až 48 % kódu informačních systémů a zabírají až 50 % jejich vývojového času [33]. Do **UI** se přitom typicky promítá mnoho aspektů, které jsou v systému obsaženy. Například byznysová pravidla jsou promítána do **UI** při validaci vstupních dat formulářů na straně klienta [13]. Autoři přístupu **ADDA** přicházejí s řešením v podobě využití několika **DSL** pro popis jednotlivých aspektů a run-time weavingu, který aspekty při běhu aplikace dynamicky začlení do **UI** s ohledem na aktuální kontext uživatele, například na jeho geolokační polohu či velikost displeje, na kterém je rozhraní zobrazováno. Díky tomu je dosaženo významné redukce kódu [9] potřebného pro popis adaptibilního uživatelského rozhraní.

3.4.3 Architektura systému využívajícího **ADDA**

Přístup **ADDA** přináší speciální úpravu standardní třívrstvé architektury. Pro zachycení aspektů v *single-focal point* tento přístup využívá speciálních registrů, ze kterých jsou aspekty automaticky distribuovány a začleňovány pomocí aspect-weaverů. Architektura distribuovaného systému využívajícího **ADDA** pro implementaci aspektově orientovaného uživatelského rozhraní je znázorněna na obrázku 3.3. Zatímco datová a aplikační vrstva zůstává zachována, prezentacní vrstva je distribuována na koncová zařízení, které poskytuje různé platformy. Pomocí **REST API** jsou pak aspekty ze *single-focal point* distribuovány ze serveru do koncových zařízení. Registry jsou navíc rozloženy podle toho, zda obsahují platformově závislé nebo nezávislé aspekty [13]. Mezi platformově nezávislé aspekty patří například struktura

²Zde je předpokládána standardní třívrstvá architektura informačních systémů [24].



Obrázek 3.3: Architektura systému využívajícího ADDA

datového modelu, která se promítá do formulářů uživatelského rozhraní. Mezi platformově závislé aspekty patří například jednotlivé widgety, ze kterých je uživatelské rozhraní složeno, a které se liší podle cílové platformy.

3.4.4 Výhody a nevýhody

Využití přístupu **ADDA** poskytuje nástroje, které umožní splnění požadavků, které jsou identifikovány v sekci 2.4. **ADDA** poskytuje vývojářům způsob, jakým mohou výrazně snížit náklady na vývoj a údržbu systému díky snížení duplikace byznysové logiky, které je dosaženo extrakcí aspektů do *single focal point* a jejich automatickou distribucí do příslušných komponent systému.

Tento přístup však nese vysokou počáteční investici v podobě vývoje specializovaných **DSL** a dynamických aspect weaverů. Ačkoliv autoři tohoto přístupu implementovali prototypy knihoven poskytující zmiňovanou funkcionality, pro nasazení do reálného systému nejsou tyto knihovny připraveny. Pro popis byznysových pravidel využívá **ADDA** nástroj **Drools**, který však není pro tento úkol optimální, jak bude popsáno v sekci 3.5.2.

3.5 Stávající řešení správy a reprezentace business pravidel

Tato sekce se zaměřuje na současné možnosti správy a zachycení byznysových pravidel ve specializovaných jazycích a vhodnost jejich použití pro účel této práce. Ačkoliv existuje relativně velké množství knihoven umožňujících automatickou distribuci byznysových pravidel a poskytujících **DSL** pro jejich popis, žádný z nich nepodporuje dostatečně velké množství platforem. Příkladem může být projekt **FlexRule**³ pro platformy .NET a JavaScript nebo **BRMS JRules** [8] od společnosti **IBM** pro platformu **Java EE**. Tato sekce se proto zaměřuje

³<<http://www.flexrule.com/archives/business-rule-language/>>

zejména na framework **BPEL**, který se věnuje byznysovým pravidlům v prostředí **SOA**, a na framework Drools, který využívají autoři přístupu **ADDA**. Pozornost je věnována také nástroji JetBrains MPS, který umožňuje vytvářet vlastní **DSL** a transformovat ho do dalších jazyků.

3.5.1 Business Process Execution Language

Technologie Business Process Execution Language (**BPEL**) využívá speciálního **DSL** postaveného na jazyku **XML** k popisu byznysových procesů realizovaných webovými službami [1]. Umožňuje *top-down* realizaci **SOA** skrz kompozici, orchestraci a koordinaci služeb [31]. Přístup **BPEL** využívá meta-služby, které se starají o uložení a transformaci byznysových pravidel a také o zachycení byznysových operací a aplikaci těchto pravidel [53].

BPEL přináší možnost využít byznysová pravidla spravovaná doménovými experty v procesně orientovaném prostředí **SOA**. Díky tomu výrazně zvyšuje kvalitu a snižuje náročnost vývoje. K tomu ale vynucuje využití orchestrace, od které nejnovější výzkum v oblasti **SOA** a zejména Microservices ustupuje na úkor decentralizace a choreografie služeb [2, 16]. Tento fakt je zároveň v rozporu s požadavky definovanými v sekci 2.4.

3.5.2 Drools

Framework Drools⁴ je open-source projekt realizující *business rule management system* (**BRMS**), tedy nástroj pro vývoj a správu byznysových pravidel. Framework umožňuje vývoj tzv. *produkčních systémů* tvořených sadou *produkčních pravidel*. Produkční pravidlo se skládá z levé strany (**LHS** z anglického *left-hand side*), a z pravé strany (**RHS** z anglického *right-hand side*). **LHS** popisuje situaci, při které má být pravidlo aplikováno. **RHS** popisuje akci, která má být vykonána.

Součástí frameworku Drools je speciální doménově specifický jazyk vyvinutý přímo pro modelování produkčních pravidel. Tento jazyk umožňuje popsat **LHS** i **RHS** daného pravidla včetně zápisu logických výrazů, využití lokálních i globálních proměnných s plnou typovou kontrolou a podporu regulárních výrazů. Navíc je možno importovat i pomocné funkce, které lze využít v podmínkách pravidla.

Ačkoliv je jazyk Drools **DSL** vymodelovaný přímo pro zápis pravidel doménovými experty, produkční pravidla se liší od byznysových pravidel zavedených v sekci 2.1. Využít tak lze pouze **LHS**. Zároveň jazyk Drools **DSL** postrádá nástroje pro kvalitní popis byznysového kontextu držícího byznysová pravidla, zejména pak rozšířování jiných kontextů a popis typu jednotlivých pravidel [11]. Ze strany frameworku Drools navíc nejsou podporovány jiné platformy než Java a .NET, což nevyhovuje požadavkům na platformovou nezávislost.

⁴<<https://www.drools.org/>>

3.5.3 JetBrains MPS

Moderním nástrojem pro tvorbu doménově specifických jazyků je *JetBrains MPS* (Meta Programming System)⁵. Staví na konceptu *language-oriented programming* (*LOP*) [64] zaměřujícího se na vývoj specifického abstraktního jazyka a jeho použití pro implementaci programu. Pro překlad ze specifického jazyka do spustitelného kódu je použit automatický překladač. Příkladem jazyka, který využívá koncept *LOP*, je *LATEX*, který byl využit pro sazbu této diplomové práce. Ten totiž pomocí maker jazyka *TEX* sestavuje abstraktnější jazyk, který umožňuje autorovi soustředit se na strukturu textu, aniž by se musel příliš detailně zaobírat samotnou sazbou.

MPS umožnuje uživateli nadefinovat gramatiku speciálního *DSL* a následně poskytuje editor pro tento jazyk včetně automatického validátoru. MPS také umožňuje transformování nadefinovaného jazyka do obecných programovacích jazyků, zejména pak do jazyka Java. Díky tomu lze nejen vytvářet libovolné *DSL*, ale také rozšiřovat existující jazyky.

Výhodou tohoto přístupu je vysoká úroveň abstrakce a možnost zapojit do vývoje doménové experty. *DSL* zvyšuje expresivitu kódu a díky tomu se zmenšuje jeho objem. Nižší objem kódu vede ke snížení nákladů na jeho údržbu a vývoj [40, 59]. Významnou výhodou MPS, potažmo *LOP*, je nezávislost na cílové platformě. Nástroj MPS umožňuje snadné znovaupoužití pravidel a jejich transformaci do neomezeného počtu jazyků pro využití na mnoha platformách. Podobně jako u *MDD* je však problém v dopředném generování – editor MPS totiž neumožňuje načítat víceúčelový jazyk zpět do *DSL*.

3.6 Síťové architektury

Závěrem se tato kapitola věnuje přehledu síťových architektur, které mohou být využity pro distribuci byznysových pravidel v systému stavějícímu na *SOA*.

3.6.1 Architektura klient-server

Model klient-server popisuje vztah mezi komponentami systému, klienty a serverem. Klient zašle požadavek serveru a ten mu vrátí odpověď [4]. Tento model může být použit obecně i v rámci jednoho počítače, nejčastěji je však využíván v síťové komunikaci mezi více počítači.

Tento přístup má několik zásadních výhod. Díky svojí obecnosti je nezávislý na jakékoli platformě. Zároveň tato architektura přesouvá byznysovou logiku a ukládání dat na server a umožňuje snadnější kontrolu nad systémem a jeho centrální administrací. S tím je spojena i snazší škálovatelnost systému. Model klient-server přináší díky centralizaci i lepší zabezpečení, kdy server může snadno definovat a vynucovat přístupová pravidla.

⁵<<https://www.jetbrains.com/mps/>>

3.6.2 Remote procedure call

Architektura **RPC** staví na modelu klient-server a umožňuje jednomu procesu (klientovi) zavolat proceduru na druhém, vzdáleném procesu (serveru). **RPC** zapouzdřuje síťovou komunikaci a v programu samotném je vzdálená procedura volána stejným způsobem jako lokální procedury [46]. Základním prvkem architektury na klientovi i na serveru je tzv. *stub*. Tato komponenta umožňuje volat, resp. obsloužit, vzdálenou proceduru lokálně a zapozdřuje veškerou síťovou komunikaci a serializaci či deserializaci argumentů, resp. návratových hodnot.

3.6.3 Representational state transfer

Alternativou k **RPC** je architektura Representational state transfer (**REST**). Ta je využívána pro webové služby a staví na protokolu **HTTP**. **REST** klade na systém několik architektonických omezení, díky kterým může systém dosáhnout lepšího výkonu, vyšší škálovatelnosti, jednoduchému používání a lepší odolnosti vůči chybám [22]. Principy architektury **REST** zahrnují využití architektury klient-server, bezestavovost a kešování požadavků, vrstvení systému, zdrojový kód na vyžádání a jednotné rozhraní. **REST** modeluje systém jako množinu zdrojů (z anglického *resources*), nad kterými jsou prováděny operace pomocí **HTTP** požadavků.

Nevýhodou architektury **REST** je náročná implementace transakcí, které zahrnují více zdrojů najednou. Protokol **HTTP** nepodporuje uzavření více požadavků do jedné atomické transakce. To může být problém v **SOA** zejména pokud je vyžadována kooperace více služeb najednou při vykonávání byznysové operace. Existují však koncepty, které využívají model Try-Cancel/Confirm [49], umožňující zajistit atomické transakce nad **REST** architekturou.

3.7 Shrnutí

V této kapitole byla provedena rešerše architektur, paradigmat a frameworků, které by mohly být vhodné pro řešení sdílení byznysových pravidel v **SOA**, a byly shrnuty jejich výhody a nevýhody. Velká část kapitoly byla věnována inovativnímu přístupu k návrhu softwarových systémů **ADDA**, ze kterého bude vycházet návrh frameworku, jež bude výstupem této práce. Kapitola dále shrnula rešerši stávajících řešení správy a reprezentace byznys pravidel a zhodnotila jejich vhodnost pro použití v této práci. Nakonec se kapitola zabývala existujícími síťovými architekturami, které by mohly být využity pro distribuci byznysových pravidel v rámci **SOA**.

Kapitola 4

Návrh frameworku

V této kapitole je diskutován návrh frameworku pro centrální správu a automatickou integraci business pravidel v prostředí architektury orientované na služby. Tento návrh staví na znalostech získaných v předchozí kapitole 3, zejména na paradigmatu [AOP](#) a přístupu [ADDA](#), a vyhovuje požadavkům identifikovaným v sekci [2.4](#)

4.1 Vize frameworku

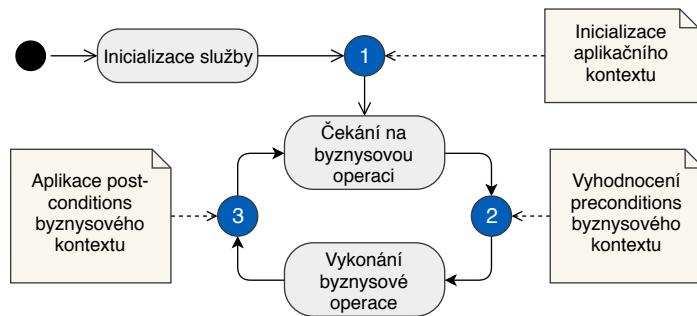
Navrhovaný framework vychází z přístupu [ADDA](#) představeného v sekci [3.4.1](#) a jeho cílem je umožnit zápis byznysových pravidel v *single-focal point*, ze kterého budou pravidla automaticky integrována do jednotlivých služeb systému. Pro zápis pravidel bude framework využívat speciální [DSL](#), které umožní jejich snadný zápis a inspekci. Díky tomu bude možné redukovat manuální duplikaci pravidel, která jsou sdílena mezi jednotlivými službami a tím snížit náklady na vývoj a údržbu systému a zmírnit riziko lidské chyby, které je spojeno s manuální synchronizací duplikované logiky.

4.2 Formalizace architektury orientované na služby

Využítí přístupu [ADDA](#) vyžaduje formalizaci problematiky byznysových pravidel v [SOA](#) do termínů [AOP](#). Za aspekt bude framework považovat byznysové pravidlo. Dále je nutno identifikovat *join-points*, určit podobu *advices*, popsat způsob jakým budou zachyceny *point-cuts* a nakonec navrhnout proces *weavingu* pravidel.

4.2.1 Join-points

Identifikace join-points vychází ze životního cyklu služby, který je znázorněn na obrázku [4.1](#). První fází v životním cyklu služby je její inicializace, konkrétně načtení exekučního



Obrázek 4.1: Diagram životního cyklu služby a identifikovaných join-pointů

kontextu. V tomto bodě je potřeba získat veškerá pravidla, která bude služba potřebovat ke své funkci. Po inicializaci vstupuje služba do fáze, ve které může přijímat požadavky na vykonání byznysových operací. Při přijmutí požadavku je nejprve nutno určit byznysový kontext a poté vyhodnotit veškeré *preconditions*. Pokud jsou všechny předpoklady pro spuštění operace splněny, může být vykonána. Po dokončení operace je nutno aplikovat relevantní *post-conditions*.

Identifikované join-points tedy jsou:

- ① Inicializace instance služby
- ② Volání byznysové operace
- ③ Dokončení byznysové operace

4.2.2 Pointcuts

Zdrojový kód 4.1: Ukázka zápisu validačních pravidel pomocí anotací v jazyku Java

```

1  class BillingAddress {
2
3      @NotBlank(message = "country is compulsory")
4      private String country;
5
6      /* ... */
7
8  }

```

V join-pointech ② a ③ musejí být aplikována všechna byznysová pravidla vztahující se k prováděné byznysové operaci. Pro zápis pointcutu byznysového pravidla pro tyto join-pointy se lze inspirovat standardem [JSR 303 \[3\]](#), který umožňuje validovat data byznysových objektů vstupujících do byznysových operací pomocí anotací atributů těchto objektů. Příklad

validačních anotací je znázorněn ve zdrojovém kódu 4.1, kde je pomocí anotace `@NotBlank` zajištěno, že fakturační adresa bude mít vyplněno pole `country` – v kontextu navrhovaného frameworku se jedná o paralelu preconditions. Podobným způsobem by každá byznysová operace mohla pomocí meta-instrukcí specifikovat, která byznysová pravidla bude využívat. Toto řešení však neposkytuje možnost dynamicky při běhu programu změnit sadu byznysových pravidel, ani sdílení pravidel mezi jednotlivými operacemi. Tento problém lze řešit využitím konceptu byznysového kontextu, který zapouzdřuje byznysová pravidla, a byznysová operace se na něj může explicitně odkázat. Obsah byznysového kontextu přitom může být dynamicky změněn za běhu programu. Příklad provázání byznysové operace a byznysového kontextu v jazyce Java je znázorněn ve zdrojovém kódu 4.2. Implementace byznysové operace pomocí anotace `@BusinessContext` definuje, který byznysový kontext má být při jejím vykonávání využit. Kontext je definován separátně pomocí `DSL` a lze ho měnit bez zásahu do implementace operace.

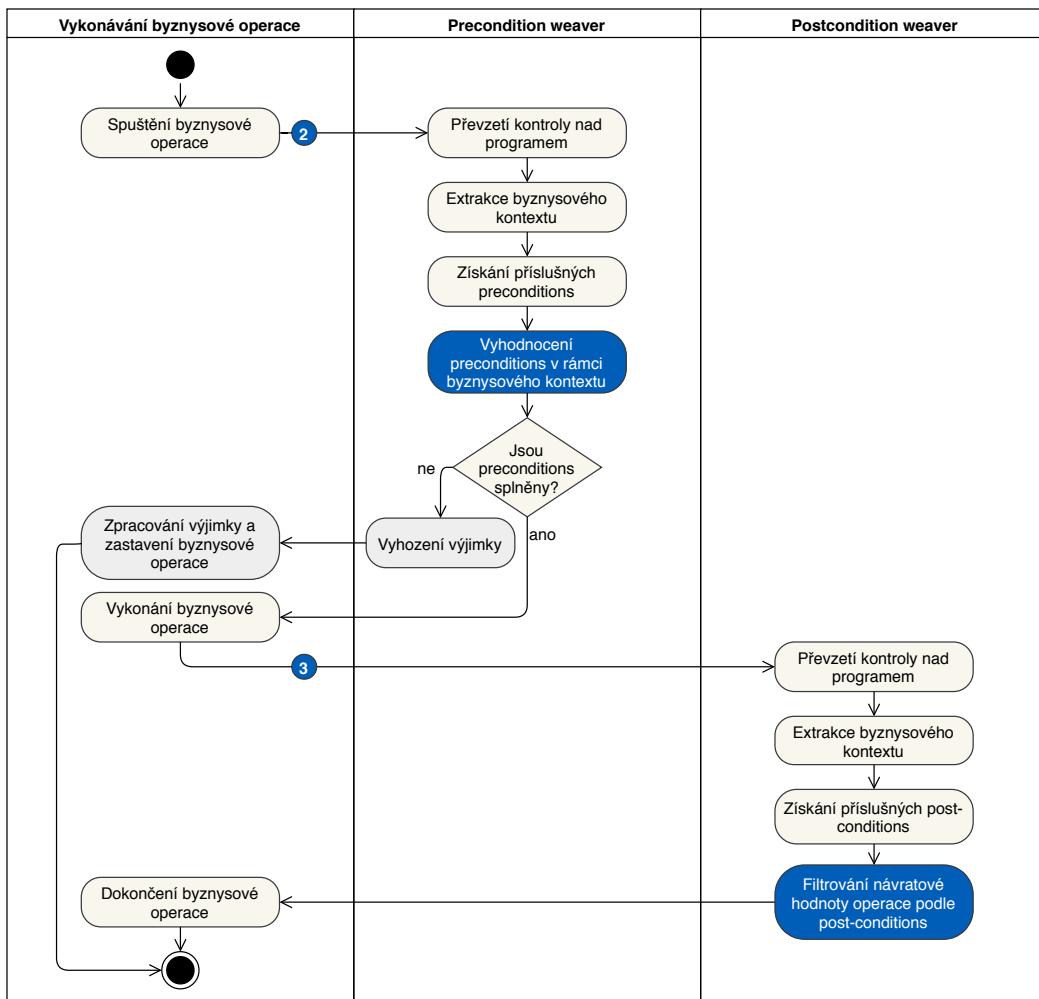
Zdrojový kód 4.2: Příklad provázání byznysové operace a byznysového kontextu pomocí anotací v jazyce Java

```

1 public class OrderService {
2
3     @BusinessContext("order.create")
4     public void createOrder() { /* ... */ }
5
6 }
```

Sdílení pravidel mezi byznysovými kontexty, potažmo byznysovými operacemi a mezi jednotlivými službami, lze realizovat pomocí dědičnosti kontextů. Každý kontext, který by potřeboval validovat fakturační adresu, by tak mohl dědit od kontextu vytváření faktury. Uvažme příklad uvedený v sekci 2.3. Vytváření objednávky sdílí byznysová pravidla operace vytváření faktury. Byznysové operace se mohou odkazovat na své byznysové kontexty, které mají být při jejich vykonávání použity. Pokud by kontext vytváření objednávky dědil od kontextu vytváření faktury, mohl by sdílet jeho pravidla, včetně validace fakturační adresy. Pokud by bylo potřeba upravit, přidat či sdílet další pravidla, nebylo by nutné zasahovat do kódu služeb.

V join-pointu ① načte služba všechna byznysová pravidla, která bude potřebovat ke své činnosti. V případě kompozitní služby je nutno načíst i sdílená pravidla definovaná v jiných službách. Kompozitní služba tedy musí zjistit, která pravidla je potřeba získat, a následně si je vyžádat od ostatních služeb. Pointcut byznysových pravidel v tomto join-pointu vychází z dědičnosti byznysových kontextů, která určuje, jaká pravidla budou mezi službami přenesena.



Obrázek 4.2: Proces weavingu byznysových pravidel

V příkladu ze sekce 2.3 je pointcut pravidla validujícího fakturační adresu určen dědičností kontextu vytváření objednávky od kontextu vytváření faktury.

4.2.3 Advices

V případě join-pointu ① je advice samotná reprezentace byznysového kontextu přenáše-ného mezi službami. Naopak v join-pointech ② a ③ je přidanou funkcionalitou vyhodnocování preconditions nad aplikačním kontextem, resp. aplikování post-conditions na návratovou hodnotu operace.

4.2.4 Weaving

Proces weavingu je zachycen na obrázku 4.2. Weaving v případě join-pointu ① bude provádět komponenta frameworku, která analyzuje lokálně dostupná pravidla služby, vyhod-

notí, která pravidla je potřeba stáhnout, a vyžádá tato pravidla od příslušných služeb. V případě join-pointů ② a ③ je k weavingu potřeba využít speciální aspect weaver. Ten zachytí volání byznysové operace a získá informace o aktuálním stavu aplikačního kontextu. Následně zjistí, který byznysový kontext má být aplikován, shromáždí všechny preconditions a každou z nich vyhodnotí. Pokud některá precondition není splněna, byznysová operace je zastavena a je vyhozena výjimka, kterou služba zpracuje. V opačném případě je kontrola vrácena zpět službě, která vykoná byznysovou operaci. Po dokončení operace aspect weaver zachytí výstup byznysové operace a aplikuje post-conditions daného byznysového kontextu.

4.3 Popis byznysových kontextů

Přístup **ADDA** doporučuje popsat byznysová pravidla pomocí vlastního, na míru šitého, doménově specifického jazyka [12]. Pro účely frameworku bude popsán pomocí **DSL** celý byznysový kontext. Jak bylo popsáno v sekci 3.5, vlastnosti nástrojů Drools a JetBrains MPS nejsou pro použití v této práci optimální. Pro účely frameworku budou specifikovány vlastnosti, kterými by **DSL** mělo disponovat, a jeho konkrétní podoba bude přenechána na implementaci frameworku.

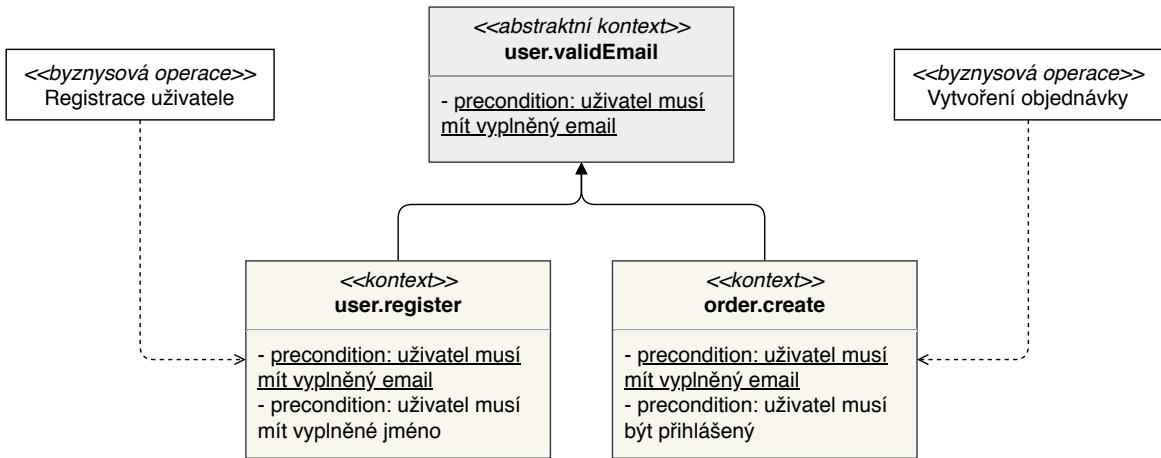
4.3.1 Dědičnost byznysových kontextů

V předchozím textu byl představen koncept dědičnosti byznysových kontextů. Každý kontext díky němu může dědit od libovolného množství jiných kontextů, a sdílet tak jejich byznysová pravidla. Byznysové operace pak mohou samy určit, který byznysový kontext se k nim váže. **DSL** pro popis pravidel musí umožňovat dědičnost zachytit. Tento koncept však přináší několik problémů, které jsou rozebrány v následujících odstavcích.

Může nastat situace, kdy je potřeba sdílet pouze některá byznysová pravidla daného kontextu. Při mapování kontextů jedna ku jedné s operacemi by to ale nebylo možné. Řešením je využití tzv. *abstraktních kontextů*, které přímo nevyužívá žádná byznysová operace. Příklad znázorněný na obrázku 4.3 popisuje situaci, kdy je nežádoucí, aby kontext `user.register` zdědil pravidlo vyžadující přihlášení uživatele.

Kvůli dědičnosti může vzniknout v grafu závislostí kontextů cyklus, který by způsobil zacyklení procesu inicializace v ①. Řešením je validátor vestavěný do nástroje pro správu byznysových kontextů, který bude případné cykly detekovat a reportovat uživateli.

Vícenásobná dědičnost může přinést problém, kdy jeden kontext zdědí více stejných pravidel z různých zdrojů, tzv. *diamond problem* [7]. Tomu lze předejít tak, že každé pravidlo bude mít unikátní identifikátor v rámci celého systému a při dědění budou zohledněna pouze unikátní pravidla. Zajištění unikátního identifikátoru lze zajistit díky nástroji pro centrální



Obrázek 4.3: Znázornění abstraktního byznysového kontextu

administraci byznysových pravidel, který má přehled o všech pravidlech. Díky tomu může administrátora upozornit na případnou duplikaci.

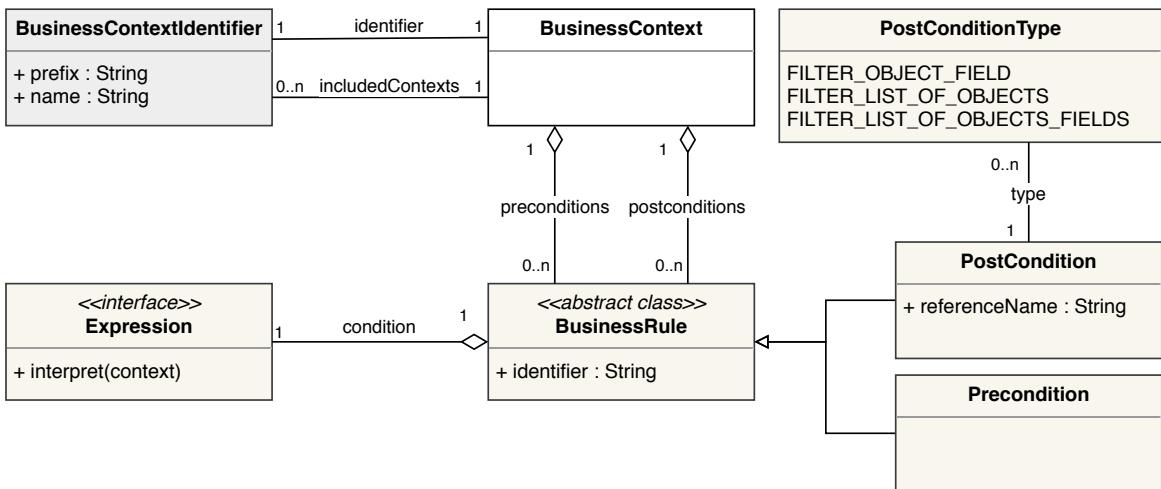
4.3.2 Logické podmínky byznysových pravidel

Sekce 2.1 uvádí, že pravidla obsahují logické podmínky. V případě preconditions je to ověření podmínky, která musí být platná před spuštěním byznysové operace v ②. V případě post-condition může filtrování návratové hodnoty podléhat splnění určité podmínky, která musí být vyhodnocena v ③. Zápis logických podmínek bude závislý na konkrétní implementaci **DSL**. Nicméně návrh frameworku bude předpokládat, že po načtení z **DSL** budou jednotlivé výrazy podmínek v paměti uloženy jako strom samostatných objektů podle návrhové vzoru Composite [24]. Díky tomu bude framework moci využít k jejich vyhodnocování návrhový vzor Interpreter [24]. **DSL** tedy musí umožňovat takový zápis logických podmínek, který půjde převést do této podoby.

4.3.3 Filtrování návratových hodnot byznysové operace

Při aplikování post-conditions je filtrována návratová hodnota byznysové operace. Tou může být proměnná obsahující číslo, text, objekt, či jejich kolekce. Filtrování jednoduchých hodnot nemá pro byznysová pravidla reálný přínos. V případě objektu lze filtrovat jeho atributy, například skrýt e-mailovou adresu uživatele. V případě kolekce lze filtrovat jejich prvky, například skrýt objednávky, které uživateli nepatří. Pokud se v kolekci nachází objekty, lze požadovat, aby byly zakryty atributy jednotlivých objektů, například filtrování e-mailových adres v kolekci více uživatelů. Identifikovanými typy post-conditions jsou:

- `FILTER_OBJECT_FIELD` filtruje atribut objektu, který je výstupem operace.



Obrázek 4.4: Metamodel byznysového kontextu

- FILTER_LIST_OF_OBJECTS filtruje objekty v kolekci, která je výstupem operace.
- FILTER_LIST_OF_OBJECTS_FIELDS filtruje atributy objektů v kolekci, která je výstupem operace.

4.3.4 Shrnutí

Na základě textu této kapitoly lze vyvodit specifikaci, kterou musí implementace [DSL](#) pro zápis byznysových kontextů splňovat. Požadované vlastnosti jsou:

- Označení byznysových kontextů unikátními identifikátory, které se skládají z prefixu a jména
- Podpora dědičnosti byznysových kontextů včetně abstraktních konceptů
- Zápis preconditions a jejich označení unikátním identifikátorem
- Zápis post-conditions, jejich označení unikátním identifikátorem a zápis jejich typu
- Zápis logických výrazů byznysových pravidel a možnost jejich transformace do návrhového vzoru Composite

4.4 Metamodel byznysového kontextu

Z předchozího textu vyplývá podoba metamodelu byznysových kontextů, pomocí které budou reprezentovány v paměti počítače. Kromě samotných logických výrazů musí pravidlo nést informace o tom, zda se jedná o precondition nebo post-condition, a také jeho

identifikátor. Post-condition navíc potřebuje uložit informaci o jejím typu a názvu. Pravidla jsou uskupována do byznysových kontextů, z nichž každý má svůj unikátní identifikátor skládající se z prefixu a samotného jména a seznam kontextů, od kterých dědí. Diagram tříd navrženého kontextu je znázorněn na obrázku 4.4. Abstraktní třída `BusinessRule` reprezentuje byznysové pravidlo a je rozšířena třídami `Precondition` a `PostCondition`. Typ postcondition je reprezentován enumerací `PostConditionType`. Logické podmínky byznysových pravidel budou rozšiřovat rozhraní `Expression`, které podporuje návrhový vzor Interpreter. Byznysová pravidla jsou uskupena do byznysových kontextů, které jsou reprezentovány třídou `BusinessContext`. Identifikátory kontextů jsou reprezentovány třídou `BusinessContextIdentifier`.

4.5 Organizace byznysových kontextů

Každá služba bude mít lokálně uložen popis byznysových kontextů, které se sémanticky vztahují k její doméně. Pro snazší přidělení byznysových kontextů ke službám bude v identifikátoru kontextu sloužit tzv. *prefix*. Kontexty se stejným prefixem pak budou spravovány výhradně jednou službou. Například kontexty služby spravující objednávky budou označeny prefixem `order`, zatímco kontexty služby zajišťující fakturaci budou označeny prefixem `billing`.

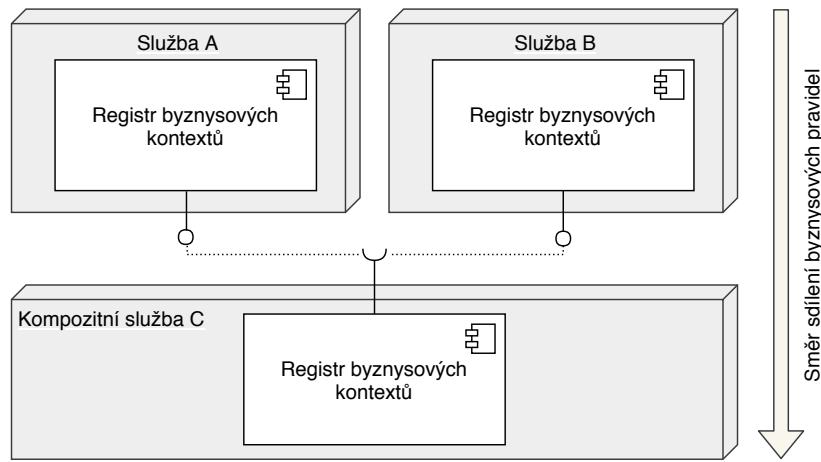
4.5.1 Registr byznysových kontextů

Cílem frameworku je popsat byznysové kontexty v jednom místě, tzv. *single-focal point*, ze kterého budou automaticky distribuovány. Pro tento účel bude každá služba disponovat registrem byznysových pravidel (třída `BusinessContextRegistry`), který bude mít za úkol kontexty načítat z [DSL](#) do metamodelu, stahovat lokálně nedostupné kontexty z ostatních služeb a načtené kontexty uchovávat pro použití při weavingu. Při inicializaci kontextů spolu budou registry jednotlivých služeb komunikovat a vyměňovat si sdílené kontexty.

Postavení byznysových registrů vůči jednotlivým službám je ilustrováno na obrázku 4.5. Registry kompozitních služeb budou využívat rozhraní registrů ostatních služeb, jejichž byznysová pravidla sdílejí. Tento koncept vychází z architektury [ADDA](#) systémů, která byla popsána v sekci 3.4.3.

4.5.2 Uložení kontextů

Byznysové kontexty popsané pomocí [DSL](#) mohou být v příslušné službě uloženy v souborech na disku či v databázi. Navrhovaný framework by na způsobu uložení neměl být závislý



Obrázek 4.5: Postavení registrů byznysových kontextů vůči jednotlivým službám

a o potřebné kroky pro načtení či případně uložení kontextu se postará konkrétní implementace. Pro tento účel je tedy vhodné, aby registr pracoval s nekonkrétními rozhraními, na jejichž implementaci nebude nijak záviset.

4.6 Distribuce sdílených byznysových kontextů mezi službami

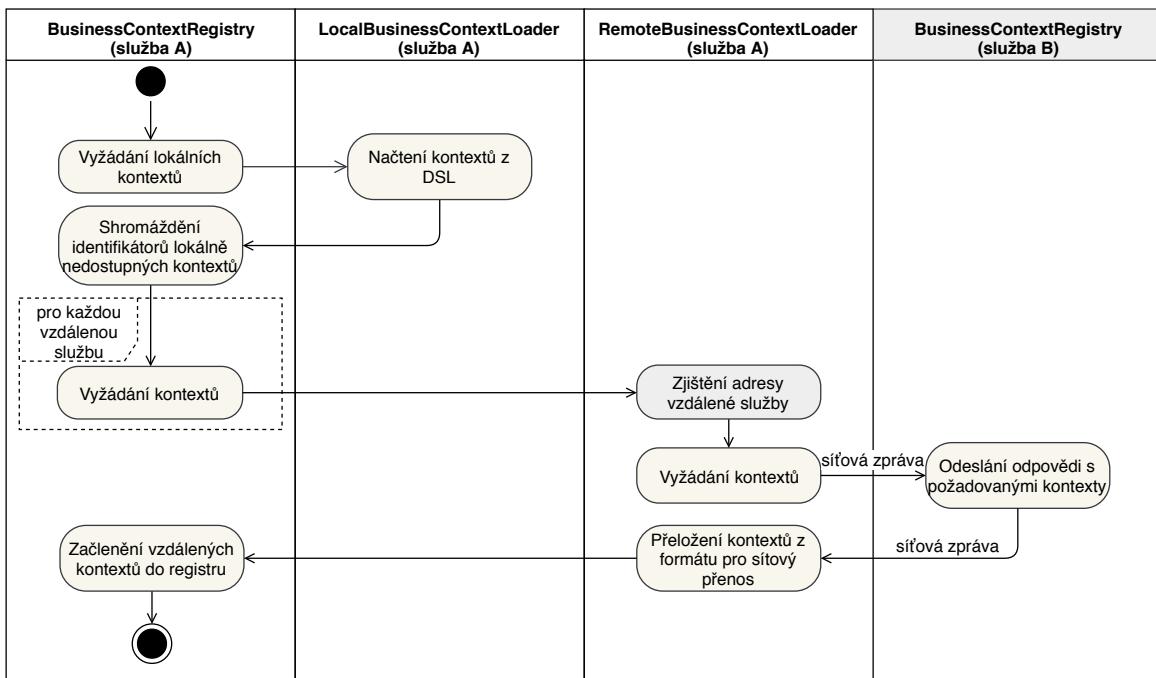
Navržený framework má za úkol automaticky integrovat byznysová pravidla, potažmo byznysové kontexty, do jednotlivých služeb systému. V případě, že je byznysový kontext sdílen mezi více službami, je nutné ho distribuovat ze služby, ve které je definován. K tomu framework využije síťovou komunikaci.

4.6.1 Formát pro síťový přenos

Pro přenos kontextů po síti lze využít přímo [DSL](#), ve kterém jsou pravidla uložena. Takové řešení by ale mohlo být neefektivní kvůli velikosti přenášených dat. Implementace proto může zvolit specializovaný síťový formát. Framework umožňuje využít oba tyto způsoby.

4.6.2 Architektura síťové komunikace

Architektura pro síťový přenos musí stavět na modelu klient-server. Framework předpokládá, že kompozitní služba bude zastávat roli klienta, a služba, která sdílí svá pravidla, bude zastávat roli serveru. Konkrétní podoba je stejně jako přenosový formát přenechána na implementaci. Lze využít přístup [RPC](#), který umožní volání procedur vracejících požadované byznysové kontexty. Stejně tak může implementace využít architekturu [REST](#), kdy budou byznysové kontexty vnímány jako jednotlivé resources a budou dostupné na unikátní [URI](#).



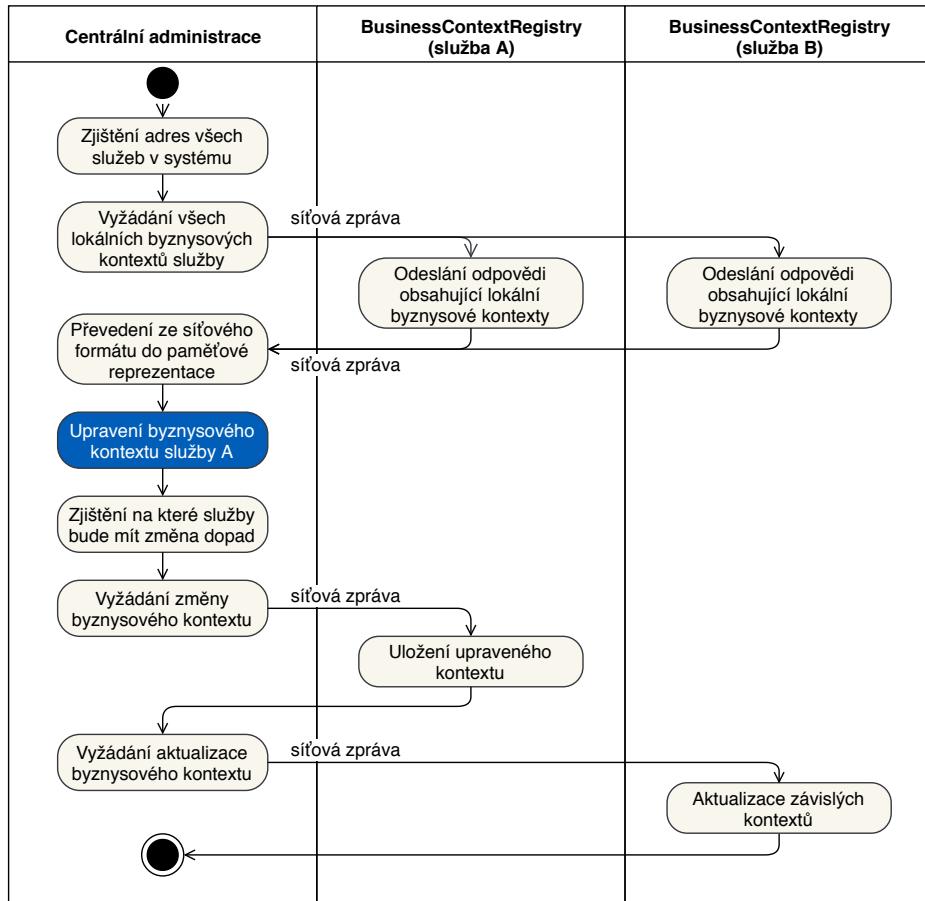
Obrázek 4.6: Proces inicializace byznysových kontextů

4.6.3 Service discovery

Aby framework mohl distribuovat byznysové kontexty mezi službami, musí služba vyžadující kontext znát adresu služby, od které ho vyžaduje. Adresy služeb mohou podléhat různým konfiguracím, které se mohou lišit systém od systému. Framework proto nesmí být závislý na způsobu, jakým se adresování služeb provádí. Nejlepším řešením je přenechat na uživateli frameworku, aby sám získal a předal adresy služeb ve chvíli, kdy je framework potřebuje – tedy ve chvíli, kdy je potřeba načíst lokálně nedostupné kontexty.

4.7 Inicializace byznysových kontextů

Jak bylo uvedeno v předchozím textu, v join-pointu ① je potřeba inicializovat byznysové kontexty, které bude služba ke své funkci potřebovat. Na obrázku 4.6 je znázorněn navržený proces inicializace. Na obrázku 4.6 je znázorněn navržený proces inicializace. Při něm jsou nejprve načteny lokálně dostupné kontexty popsané pomocí [DSL](#). Po převedení kontextů z [DSL](#) do metamodelu je shromážděn seznam kontextů, od kterých ostatní kontexty dědí, a z nich jsou vybrány ty, které nejsou lokálně dostupné. Následně jsou tyto vzdálené kontexty vyžádány od příslušných služeb a po obdržení jsou převedeny ze síťového formátu do metamodelu. Nakonec jsou sdílená pravidla kontextů začleněna do kontextů, které od nich dědí. Celou inicializaci bude zastřešovat komponenta [BusinessContextRegistry](#), která



Obrázek 4.7: Proces centrální správy byznysových kontextů

má znalost o všech subsystémech, které jsou k tomuto procesu potřeba. Tato komponenta implementuje návrhový vzor *Facade* [24].

4.8 Centrální správa byznysových kontextů

Jak bylo popsáno v předchozím textu, byznysové kontexty budou podle prefixu přiděleny službám, které budou spravovat jejich aktuální a jediný stav a poskytovat je jiným službám. Aby bylo možno pravidla centrálně spravovat, musí framework poskytovat proces, kterým je bude moci ze služeb získat a při úpravě je do nich znova uložit.

4.8.1 Uložení sdíleného kontextu

Při ukládání byznysového kontextu je potřeba jeho změnu propagovat také do všech kontextů, které od něj dědí. Při změně sdíleného kontextu budou všechny služby, které ho

využívají, informovány pomocí nástroje pro centrální správu byznysových pravidel. Ten má informaci o všech závislostech v systému a zároveň zná i adresu všech služeb. Nevýhodou tohoto přístupu je zvýšená komunikační zátěž kvůli většímu objemu přenesených informací, stejný kontext je totiž potřeba rozeslat mezi více služeb. Při implementaci je nutno zvážit, zda je tato zátěž vůči absolutnímu objemu přenášených dat v systému významná. Při implementaci by bylo vhodné vybrat vhodný přenosový formát, který minimalizuje dopad veškeré síťové komunikace týkající se distribuce byznysových pravidel.

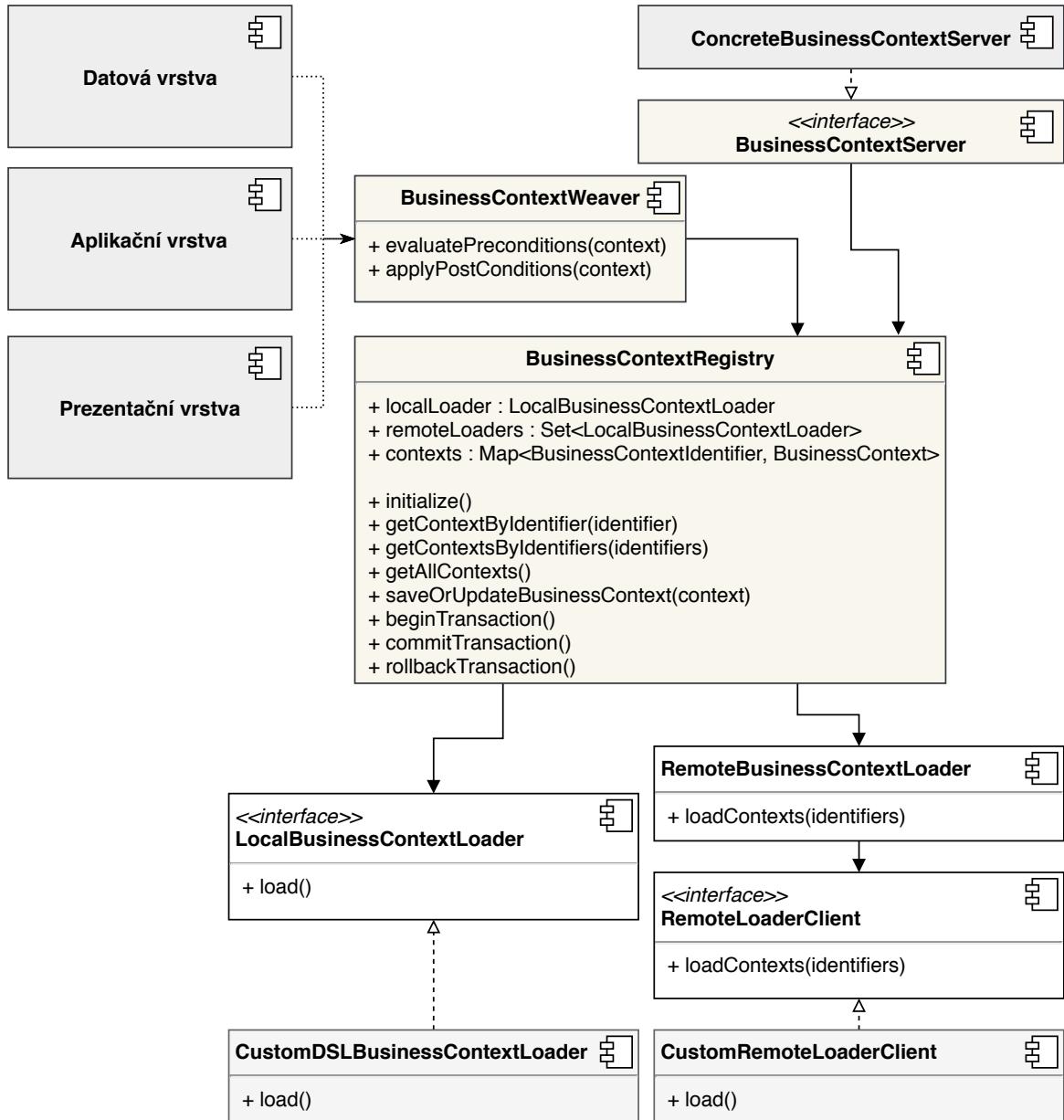
4.8.2 Proces úpravy kontextu

Na obrázku 4.7 je znázorněn proces úpravy kontextu. Nástroj pro centrální administraci nejprve načte všechny byznysové kontexty všech služeb v systému. Administrátor si vybere konkrétní pravidlo, a nástroj mu následně zobrazí formulář pro úpravu pravidla. Pravidlo je pro účely formuláře převedeno z metamodelu do [DSL](#). Po odeslání formuláře je pravidlo převedeno zpět do metamodelu. Nástroj pro administraci poté analyzuje, na které služby bude mít změna pravidla dopad. Následně zašle upravené pravidlo službě, která ho spravuje, a ostatním službám, jejichž kontexty upravené pravidlo rozšířují, rozešle požadavek na jejich aktualizaci. Proces uložení nového kontextu je analogický, až na rozeslání požadavku na aktualizaci závislých kontextů, která není u nového pravidla potřeba.

4.9 Architektura frameworku

V této sekci je popsána obecná architektura navrženého frameworku v rámci služby využívající klasickou třívrstvou architekturu [24], která se skládá z prezentační, aplikační a datové vrstvy. Každá z těchto vrstev může framework využívat pro weaving byznysových pravidel – prezentační vrstva při validování vstupních polí formuláře, aplikační vrstva při aplikaci byznysových pravidel v byznysových operacích a datová vrstva při aplikaci post-conditions pro filtrování dat při jejich získávání z databáze.

Architektura frameworku je zachycena na obrázku 4.8. Základem frameworku je komponenta `BusinessContextRegistry`, tedy registr byznysových kontextů, který je zodpovědný za inicializaci a uchovávání byznysových kontextů. Načítání kontextů lze rozdělit na lokální a vzdálené. Při načítání lokálně dostupných kontextů je potřeba získat [DSL](#) kontextu ze souboru či databáze a převést ho do metamodelu. K tomu bude využito rozhraní `LocalBusinessContextLoader`. Implementace rozhraní může být libovolná a záviset na použitém [DSL](#) či místu uložení pravidel. Naopak při načítání vzdálených kontextů je potřeba vyžádat kontexty od vzdálené služby. Třída `RemoteBusinessContextLoader` požadované kontexty zorganizuje podle prefixu a poté pomocí rozhraní `RemoteLoaderClient`



Obrázek 4.8: Architektura navrženého frameworku

načte pravidla od jednotlivých služeb. Implementace tohoto rozhraní bude záviset na použité technologii a zajistí síťovou komunikaci a převod do a z formátu pro síťový přenos. Aby mohl framework poskytovat lokální byznysové kontexty dané služby ke stažení, musí zařešit i serverovou funkcionalitu. K tomu slouží rozhraní **BusinessContextServer**. To bude využívat **BusinessContextRegistry**, ze kterého načte byznysové kontexty, které si vyžádá **RemoteLoaderClient**. Implementace serveru bude opět závislá na konkrétní technologii. Nakonec bude framework obsahovat sadu aspect weaverů, které umožní weaving byznysových pravidel do jednotlivých vrstev systému. Pro účely této práce bude framework poskytovat weavery pro využití v aplikační vrstvě pro weaving preconditions a post-conditions do byznysových operací.

4.10 Shrnutí

V této kapitole byl popsán návrh frameworku pro centrální správu a automatickou distribuci byznysových pravidel v **SOA** na základě přístupu **ADDA**. Nejprve byla formalizována doména byznysových pravidel v **SOA** do názvosloví **AOP**. Dále byla diskutována podobu byznysových pravidel, jejich logických výrazů a jakým způsobem je lze zachytit v metamodelu a v **DSL**. Kapitola dále popisuje organizaci kontextů a procesy, kterými budou distribuovány a spravovány. Nakonec byla shrnuta architektura frameworku.

Kapitola 5

Implementace prototypu frameworku

Jedním z cílů této práce je implementace prototypů knihoven frameworku navrženého v kapitole 4 pro tři rozdílné platformy, z nichž jedna musí být *Java*. Prototypy slouží pro demonstraci toho, že navržený framework lze realizovat. Tato kapitola popisuje výběr platform a konkrétní implementace knihoven pro tyto platformy. Jelikož jednotlivé implementace vycházejí ze stejného návrhu, kompletní implementace je popsána pouze pro platformu Java. Ostatní implementace jsou shrnuty komparativní metodou. Součástí kapitoly je i stručný popis použitých technologií.

5.1 Výběr použitých platem

Mimo jazyk Java, který byl určen zadáním, byla pro implementaci vybrána platforma jazyka *Python* a platforma *Node.js*, která slouží jako běhové prostředí pro jazyk *JavaScript*. Výběr byl proveden na základě aktuálních trendů ve světě softwarového inženýrství [29, 50, 60]. Tyto jazyky se v posledních letech stabilně umísťují na prvních příčkách nejpopulárnějších programovacích jazyků pro obecné použití.

5.2 Doménově specifický jazyk pro popis byznys kontextů

Pro popis byznysových pravidel v *single-focal point* navrhují framework využít speciální **DSL**. Ačkoliv není implementace **DSL** pro popis byznysových kontextů cílem této práce, pro ověření konceptu je nutné nadefinovat alespoň jeho zjednodušenou verzi a implementovat část knihovny, která bude umět jazyk zpracovat a sestavit metamodel popsaného kontextu.

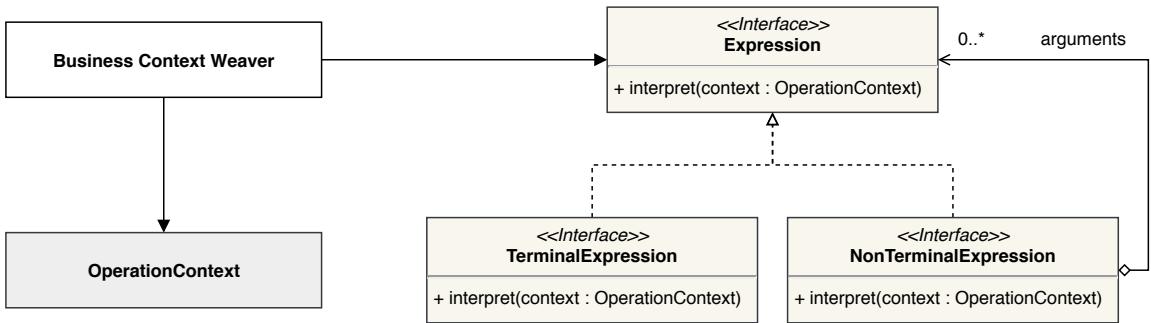
Pro popis kontextů byl zvolen univerzální formát Extensible Markup Language ([XML](#)) doplněný o definici schématu dat pomocí *XML Schema Definition*. Díky formálně definovanému schématu lze popis byznys kontextu automaticky validovat a vyhnout se tak případným

chybám v zápisu. Tento jazyk bude možno v produkční verzi knihovny nahradit komplexnějším.

Ve zdrojovém kódu 5.1 je znázorněn 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`. Identifikátory rozšířených kontextů jsou vypsány v entitě `<includedContexts>`. Preconditions jsou definovány uvnitř entity `<preconditions>` a podobně jsou definovány `<postconditions>`. Obsažená data odpovídají navrženému metamodelu byznysového kontextu v sekci 4.4. Pro zápis podmínek jednotlivých preconditions a post-conditions byl zvolen opis derivačního stromu logicích výrazů, který bude vysvětlen v následujícím textu. Vzhledem k povaze prototypu byla relaxována podmínka na čitelnost zápisu pravidel ve prospěch jednoduššího zpracování.

Zdrojový kód 5.1: Příklad zápisu byznys kontextu v jazyce XML

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <businessContext prefix="user" name="createEmployee">
3   <includedContexts/>
4   <preconditions>
5     <precondition name="Cannot use hidden product">
6       <condition>
7         <logicalEquals>
8           <left>
9             <variableReference
10               objectName="product"
11               propertyName="hidden"
12               type="bool"/>
13           </left>
14           <right>
15             <constant type="bool" value="false"/>
16           </right>
17         </logicalEquals>
18       </condition>
19     </precondition>
20   </preconditions>
21   <postConditions/>
22 </businessContext>
```



Obrázek 5.1: Použití vzoru Interpreter pro vyhodnocování logických výrazů

5.2.1 Gramatika logických podmínek byznysových pravidel

Podmínky byznysových pravidel jsou součástí [DSL](#) pro popis byznysových kontextů. Návrh frameworku vyžaduje, aby logické výrazy byznysových pravidel byly reprezentovány v paměti jako strom objektů, jak je popsáno v sekci [4.3.2](#). V rámci implementace byla specifikována konkrétní podoba logických výrazů, která je společná pro jednotlivé knihovny.

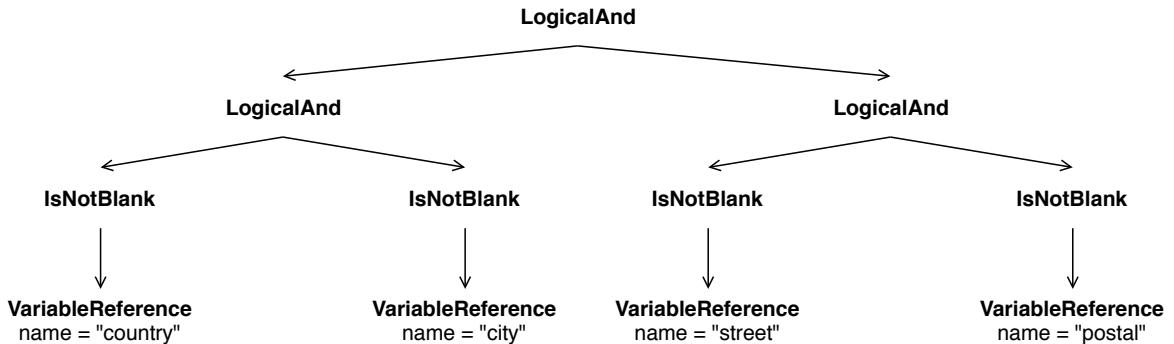
Podmínky byznysových pravidel se skládají z jednotlivých výrazů, které tvoří orientovaný acyklický graf, tzv. *derivační strom*. Výrazy se dělí na *terminály* a *neterminály* [43]. Terminál znamená, že z daného výrazu již nevychází žádná hrana do jiného výrazu. Neterminál je opak terminálu. Použití návrhového vzoru *Interpreter* [24] je demonstrováno na obrázku [5.1](#).

Prototypy knihoven disponují základní sadou výrazů pro zápis byznysových pravidel. Mezi ně patří logické operace `and`, `or`, `equals` a `negate`. Knihovny dále obsahují výraz `VariableReference`, který získá hodnotu proměnné či konstanty z kontextu. Pokud bude v kontextu uložen objekt, je potřeba přistupovat i k jeho veřejným atributům, což zajišťuje výraz `ObjectPropertyReference`. K ověření přítomnosti hodnoty v proměnné slouží výraz `IsNotNull`. Výraz `IsNotBlank` ověří, zda je v proměnné řetězec nenulové délky. Pro vložení konstantní hodnoty přímo do byznysového pravidla slouží terminál `Constant`. Pro zvýšený komfort byly přidány i výrazy realizující základní matematické operace sčítání, odečítání, násobení a dělení. Pro volání uživatelských funkcí definovaných v operačním kontextu slouží speciální výraz `FunctionCall`. V jeho případě je nutno zohlednit skutečnost, že funkce může přijímat libovolný počet argumentů. Protože volaná funkce může potřebovat přistupovat k operačnímu kontextu, musejí být argumenty také interpretovány. Bohužel nelze u uživatelem definovaných funkcí ověřit, že bude při jejich volání odpovídat počet a typ argumentů. Přehled všech výrazů, které knihovny podporují, je v tabulce [5.1](#).

Pro snazší implementaci na více platformách a prevenci sémantických chyb v pravidlech výrazy obsahují i explicitní definici svého návratového typu. Výraz byznysového pravidla může nabývat logických hodnot, může vracet číslo, textový řetězec a také objekt. Je potřeba počítat také s tím, že výraz nevrací žádnou hodnotu.

Název	Argumenty	Atributy	Návratový typ	Typ výrazu
Constant	-	Hodnota a typ konstanty	?	Terminal
FunctionCall	Libovolný počet argumentů	Návratový typ funkce	?	Terminal
IsNotNull	Jeden argument libovolného typu	-	BOOL	Neterminál
IsNotBlank	Jeden argument typu STRING	-	BOOL	Neterminál
LogicalAnd	2 argumenty typu BOOL	-	BOOL	Neterminál
LogicalEquals	2 argumenty libovolného typu	-	BOOL	Neterminál
LogicalNegate	1 argument typu BOOL	-	BOOL	Neterminál
LogicalOr	2 argumenty typu BOOL	-	BOOL	Neterminál
NumericAdd	2 argumenty typu NUMBER	-	NUMBER	Neterminál
NumericSubtract	2 argumenty typu NUMBER	-	NUMBER	Neterminál
NumericMultiply	2 argumenty typu NUMBER	-	NUMBER	Neterminál
NumericDivide	2 argumenty typu NUMBER	-	NUMBER	Neterminál
ObjectReference	-	Název objektu a název a typ proměnné	?	Terminal
VariableReference	-	Název a typ proměnné	?	Terminal

Tabulka 5.1: Přehled výrazů pro zápis byznysového pravidla



Obrázek 5.2: Syntaktický strom jednoduchého validačního pravidla

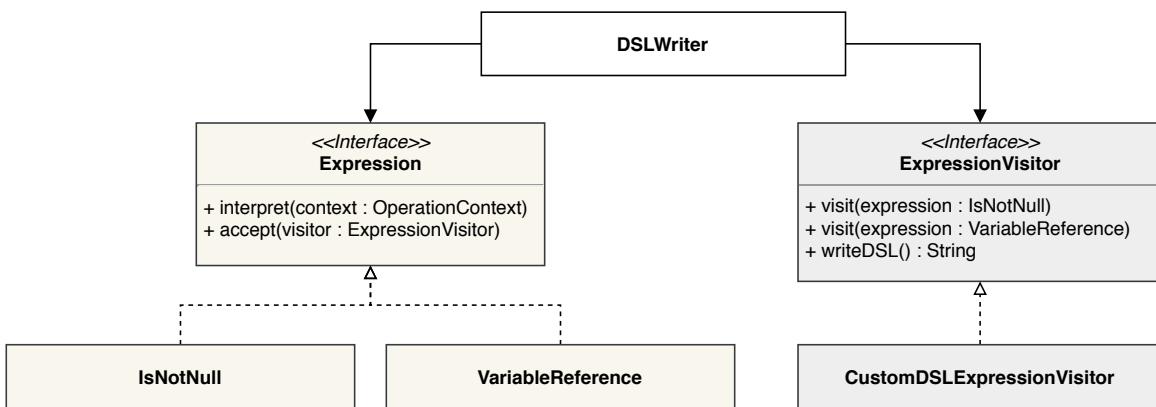
- **BOOL** je logický typ, který nabývá hodnoty **true** a **false**.
- **NUMBER** je reálné číslo zapsáno ve tvaru s desetinnou tečkou a neomezeným počtem číslic.
- **OBJECT** je objekt libovolného typu.
- **STRING** je textový řetězec.
- **VOID** je pseudotyp značící, že výraz nemá návratovou hodnotu.

Kromě argumentů neterminálů je v některých případech potřeba k výrazu uložit i dodatečné informace – *atributy*. Jedním z atributů je typ návratové hodnoty výrazu, pokud není přímo implikována. V případě výrazu **Constant** je potřeba uložit hodnotu a typ konstanty. Reference na proměnnou musí obsahovat její název a typ, reference na pole objektu navíc musí obsahovat název odkazovaného pole. Volání funkce musí obsahovat její název a návratový typ.

Na obrázku 5.2 je znázorněn syntaktický strom, který zachycuje jednoduché validační pravidlo validující fakturační adresu. Jedná se o ekvivalent validačních pravidel zachycených ve zdrojovém kódu 4.1 pomocí anotací standardu **JSR 303**. Pravidlo je tvořeno čtyřmi teminály, které se odkazují na proměnné operačního kontextu. Hodnoty proměnných jsou validovány výrazem **IsNotBlank** a jednotlivé validace jsou spojeny pomocí binárních výrazu **LogicalAnd** odpovídajících logické konjunkci.

5.2.2 Převod byznysových kontextů mezi DSL a metamodelem

Pro načtení byznysového kontextu z **DSL** využívají knihovny **XML** parsery implementující rozhraní *Document Object Model*, které je nezávislé na platformě a umožňuje přistupovat k dokumentu jako ke stromu jednotlivých elementů.



Obrázek 5.3: Využití vzoru Visitor pro převod logických výrazů do DSL

Pro uložení kontextu z metamodelu do **DSL** je vzhledem k reprezentaci logických výrazů vhodný návrhový vzor *Visitor* [24]. Ten umožňuje převádět libovolně složité logické výrazy pomocí metody *double-dispatch*. Jeho volbou je zároveň zajištěna rozšiřitelnost knihoven pro komplexnější **DSL** – bude stačit implementovat konkrétní visitor pro zvolený jazyk, aniž by bylo nutno dále zasahovat do implementace knihoven.

Princip použití vzoru *Visitor* je znázorněn na obrázku 5.3. Rozhraní **Expression** je implementováno konkrétními logickými výrazy (pro ilustraci jsou zobrazeny konkrétní výrazy **IsNotNull** a **VariableReference**), které po zavolení metody **accept()** zavolají na visitoroji příslušnou metodu **visit()**. Visitor tak může oddělit logiku, která obstarává převod jednotlivých výrazů do **DSL**.

5.3 Distribuce byznysových kontextů mezi službami

Pro sdílení byznysových kontextů a jejich pravidel mezi jednotlivými službami framework navrhuje využití síťové komunikace. Ta musí probíhat ve formátu nezávislém na platformě, ideálně s vysokou efektivitou přenosu. Návrh frameworku však pro implementaci nevynucuje použití konkrétní technologie.

5.3.1 Protocol Buffers

Pro prototypy knihoven byl pro přenos byznysových kontextů zvolen open-source formát *Protocol Buffers*¹[63] vyvinutý společností Google². Ten umožňuje explicitně definovat a využívat schéma dat, která jsou přenášena po síti, bez vazby na konkrétní programovací jazyk. Zároveň poskytuje obslužné knihovny pro vybrané platformy. Díky binární reprezentaci dat

¹<<https://developers.google.com/protocol-buffers/>>

²<<https://www.google.com/>>

je v přenosu velmi efektivní, oproti formátům jako je [JSON](#) nebo [XML](#) [42]. Na rozdíl od protokolů *Apache Thrift*³ a *Apache Avro*⁴, které poskytují velmi srovnatelnou funkcionality, má zvolený protokol kvalitnější a lépe srozumitelnou dokumentaci.

Zdrojový kód 5.2: Část definice schématu zpráv byznys kontextů ve formátu Protocol Buffers

```

1  message PreconditionMessage {
2      required string name = 1;
3      required ExpressionMessage condition = 2;
4  }
5
6  message PostConditionMessage {
7      required string name = 1;
8      required PostConditionTypeMessage type = 2;
9      required string referenceName = 3;
10     required ExpressionMessage condition = 4;
11 }
12
13 message BusinessContextMessage {
14     required string prefix = 1;
15     required string name = 2;
16     repeated string includedContexts = 3;
17     repeated PreconditionMessage preconditions = 4;
18     repeated PostConditionMessage postConditions = 5;
19 }
```

Zdrojový kód 5.2 znázorňuje zápis schématu síťových zpráv pro distribuci byznys kontexty ve formátu Protobuffer. Schéma zpráv pro výměnu kontextů dodržuje strukturu metamodelu navrženého v sekci 4.4.

ExpressionMessage obsahuje jméno, atributy a argumenty **Expression**.

ExpressionPropertyMessage je enumerace obsahující typy atributu **Expression**.

PreconditionMessage obsahuje název a podmínu precondition pravidla.

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

PostConditionTypeMessage je enumerace obsahující typy post-condition pravidla.

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

³<<https://thrift.apache.org/>>

⁴<<https://avro.apache.org/>>

BusinessContextsMessage obaluje více byznys kontextů.

5.3.2 gRPC

Síťovou komunikaci potřebnou pro distribuci byznysových kontextů realizuje open-source framework gRPC⁵, který staví na technologii Protocol Buffers. Tento framework poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC*. Zdrojový kód 5.3 znázorňuje zápis serveru, který umožňuje klientovi volat procedury `FetchContexts`, `FetchAllContexts` a `UpdateOrSaveContext`.

Zdrojový kód 5.3: Definice služby pro komunikaci byznys kontextů pro gRPC

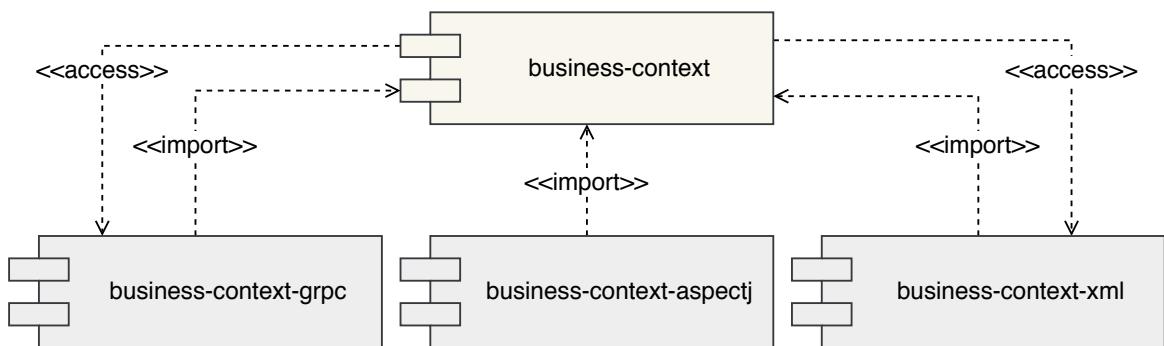
```
1 service BusinessContextServer {  
2     rpc FetchContexts (BusinessContextRequestMessage)  
3         returns (BusinessContextsResponseMessage) {}  
4  
5     rpc FetchAllContexts (Empty)  
6         returns (BusinessContextsResponseMessage) {}  
7  
8     rpc SaveOrUpdateContext (BusinessContextUpdateRequestMessage)  
9         returns (Empty) {}  
10 }
```

FetchContexts() je procedura, která umožňuje klientovi získat kontexty, jejichž identifikátory zašle jako argument typu `BusinessContextRequestMessage`. V odpovědi pak obdrží dotazované kontexty anebo 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.

SaveOrUpdateContext() slouží pro uložení nového či editovaného pravidla, které je zasláno v serializované podobě jako argument typu `BusinessContextUpdateRequestMessage`. Tato procedura může být volána pouze pokud byla nejprve spuštěna transakce.

⁵<<https://grpc.io/>>



Obrázek 5.4: Moduly prototypu knihovny pro jazyk Java a jejich závislosti

5.4 Knihovna pro platformu Java

Knihovna pro platformu Java se skládá ze čtyř modulů, které zajišťují její funkcionality:

- **business-context** obsahuje jádro knihovny
- **business-context-aspectj** poskytuje integraci s nástrojem AspectJ
- **business-context-grpc** přináší klienta a server pro distribuci byznysových pravidel pomocí frameworku gRPC
- **business-context-xml** obsahuje třídy pro čtení a zápis byznysových kontextů do XML

Moduly knihovny a jejich vzájemné závislosti jsou znázorněny na obrázku 5.4. Jádro knihovny je nezávislé na konkrétních implementaci síťové komunikace a zvoleném DSL, ale využívá rozhraní, která jsou těmito moduly implementována.

5.4.1 Správa závislostí projektu

Pro správu závislostí a automatickou komplikaci a sestavování modulů knihovny byl zvolen projekt *Maven*⁶. Tento nástroj umožňuje vývojáři komfortně a centrálně spravovat závislosti jeho projektu včetně detailního popisu jejich verze. Dále také umožňuje specifikovat a rozšiřovat komplikaci projektu.

5.4.2 Jádro knihovny

Jádro knihovny obsahuje třídy zajišťující základní funkci knihovny, tedy metamodel a registr byznysových kontextů, weaver byznysových pravidel a logické výrazy pravidel. Registr

⁶<<https://maven.apache.org/>>

kontextů je implementován třídou `BusinessContextRegistry`, která kromě funkcí popsaných v sekci 4.5.1 umožňuje snadnou konfiguraci stahování vzdálených byznysových kontextů a načítání lokálních kontextů z [DSL](#) pomocí návrhové vzoru Builder [24]. Registr není závislý na konkrétních technologiích použitých pro síťovou komunikaci ani na použitém [DSL](#).

Logické výrazy byznysových pravidel jsou implementovány samostatnými třídami, které rozšiřují jednotné rozhraní `Expression`. Toto rozhraní podporuje návrhový vzor Interpreter, který umožňuje vyhodnocení výrazů, jak bylo popsáno v předchozím textu. Zároveň je tím usnadněno rozšíření knihovny o nové výrazy. Toto rozhraní navíc poskytuje metody, které jsou využity při serializaci výrazů.

5.4.3 Weaving

Weaver byznysových pravidel musí být schopen extrahat informace o kontextu probíhající byznysové operace, aby mohl správně zvolit příslušný byznysový kontext. V knihovně pro jazyk Java byla pro tento účel zvolena anotace `BusinessOperation`, která umožňuje uživateli knihovny označit metodu vykonávající byznysovou operaci. Parametry této metody mohou být označeny anotací `BusinessContextParameter`. Weaver takto anotované parametry vloží jako proměnné do operačního kontextu a podmínky byznysových pravidel se na ně mohou odkazovat. Ukázka použití těchto anotací je ve zdrojovém kódě 5.4. Pomocí knihovny AspectJ⁷, která poskytuje sadu nástrojů pro použití principů [AOP](#), je pak weaver pravidel volán ve chvíli, kdy je vykonávána anotovaná byznysová metoda.

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

```
1 public class OrderService {  
2  
3     @BusinessOperation("order.create")  
4     public Order create(  
5         @BusinessOperationParameter("user") User user,  
6         @BusinessOperationParameter("email") String email,  
7         @BusinessOperationParameter("shippingAddress") Address shipping,  
8         @BusinessOperationParameter("billingAddress") Address billing  
9     ) { /* ... */ }  
10 }
```

⁷<<https://www.eclipse.org/aspectj/>>

5.4.4 Serializace a deserializace XML

Pro serializaci a deserializaci [DSL](#) byznysových kontextů byly implementovány třídy `BusinessContextXmlWriter` a `BusinessContextXmlLoader`. Ty využívají knihovnu JDOM 2⁸, která poskytuje kompletní sadu nástrojů pro čtení a zápis [XML](#) dokumentů. Implementuje specifikaci *Document Object Model*, pomocí které lze automaticky sestavovat a číst [XML](#) dokumenty.

5.4.5 Distribuce byznysových pravidel

K distribuci byznysových pravidel mezi službami byly implementovány obsluhující třídy `GrpcBusinessContextServer` a `GrpcBusinessContextClient`. Framework gRPC poskytuje knihovnu pro jazyk Java, která vygeneruje *Stub* zapouzdřující veškerou síťovou komunikaci. Obsluha komunikace tak vyžaduje pouze sepsání obslužného kódu, který převede byznysové kontexty z metamodelu a zpět. Obě třídy navíc obsahují metody pro snadnou konfiguraci síťových adres.

5.5 Knihovna pro platformu Python

Knihovna pro platformu Python využívá verzi jazyka 3.6, která byla v době psaní práce nejnovější stabilní verzí. Knihovna se skládá ze tří částí: `business_context` obsahuje jádro knihovny, `business_context_grpc` umožňuje síťovou komunikaci pomocí frameworku gRPC a `business_context_xml` poskytuje nástroje pro zápis a čtení kontextů z [XML](#). Pomocí nástroje `pip`⁹ lze jednotlivé části knihovny nainstalovat a využívat jako samostatné Python moduly. Implementace odpovídá navržené specifikaci.

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`, který je využit pro anotaci funkce, která implementuje byznysovou operaci. Dekorátoru je potřeba předat samotný weaver, na rozdíl od implementace v Javě, kdy se o předání weaveru postará dependency injection container.

⁸<<http://www.jdom.org/>>

⁹<<https://pypa.io/en/stable/>>

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

```
1 def business_operation(name, weaver):
2     def wrapper(func):
3         def func_wrapper(*args, **kwargs):
4             operation_context = OperationContext(name)
5             weaver.evaluate_preconditions(operation_context)
6             output = func(*args, **kwargs)
7             operation_context.set_output(output)
8             weaver.apply_post_conditions(operation_context)
9             return operation_context.get_output()
10            return func_wrapper
11        return wrapper
12
13
14 weaver = BusinessContextWeaver()
15
16
17 class ProductRepository:
18
19     @business_operation("product.listAll", weaver)
20     def get_all(self) -> List[Product]:
21         pass
```

Jména tříd a metod Jazyk Python využívá jiné jmenné konvence a jinak člení kód do jednotlivých souborů. Aby byly tyto konvence zachovány, třídy knihovny a názvy jejich metod se liší od těch v jazyce Java.

5.5.2 Použité technologie

Jazyk Python poskytuje ve své standardní knihovně téměř všechny nástroje, které byly pro implementaci prototypu knihovny potřeba. Jedinou výjimkou byl modul pro obsluhu frameworku gRPC. Ze standardní knihovny byly využity moduly `typing` pro statické typování, `re` pro regulární výrazy, `enum` pro implementaci výčtových hodnot, `copy` pro kopírování objektů, `fs` pro práci se soubory, `thrading` pro práci s vlákny a zámky, `itertools` pro práci s iterátory, `inspect` pro využití reflexe a `xml` pro práci s XML dokumenty.

5.6 Knihovna pro platformu Node.js

Knihovna pro platformu *Node.js* byla implementována v jazyce JavaScript, konkrétně ve verzi *ECMAScript 6.0*. Implementace odpovídá specifikaci návrhu, umožňuje instalaci pomocí balíčkovacího nástroje a snadnou integraci do kódu výsledné služby. Stejně jako knihovna pro jazyk Python se skládá ze tří modulů: **business-context** obsahuje jádro knihovny, **business-context-grpc** poskytuje server a klienta pro distribuci byznysových kontextů po síti pomocí frameworku gRPC a **business-context-xml** umožňuje serializaci a deserializaci z XML.

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 však podobně jako jazyk Python umožňuje využít princip dekorátoru funkce. Ukázka takového dekorátoru je ve zdrojovém kódu 5.6. Funkce `register()` obsahuje logiku pro registraci uživatele, která může obsahovat například uložení entity do databáze a odeslání registračního emailu. Při exportování funkce z Node.js modulu je využita funkce `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 Na rozdíl od implementací knihovny v jazycích Java a Python umí knihovna obsluhující gRPC fungovat i bez předgenerovaného kódu. To 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.

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

```

1 const weaver = new BusinessContextWeaver(registry)
2
3 function register(name, email) {
4   return new Promise((resolve, reject) => {
5     // ...
6   })
7 }
8

```

```

9  function wrapCall(context, func) {
10    return new Promise((resolve, reject) => {
11      try {
12        weaver.evaluatePreconditions(context)
13        resolve()
14      } catch (error) {
15        reject(error.getMessage())
16      }
17    })
18    .then(_ => func())
19    .then(result => {
20      context.setOutput(result)
21      weaver.applyPostConditions(context)
22      return new Promise(
23        (resolve, reject) => resolve(context.getOutput())
24      )
25    })
26  }
27
28  exports.register = (name, email) => {
29    const context = new BusinessOperationContext('user.register')
30    context.setInputParameter('name', name)
31    context.setInputParameter('email', email)
32    return wrapCall(context, () => register(name, email))
33  }

```

5.6.2 Použité technologie

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*¹¹ a *Rimraf*¹². Platforma Node.js neposkytuje nástroje pro práci s XML, proto byla využita knihovna *xmldom*¹³.

¹⁰<<https://yarnpkg.com/en/>>

¹¹<<https://babeljs.io/>>

¹²<<https://github.com/isaacs/rimraf>>

¹³<<https://www.npmjs.com/package/xmldom>>

5.7 Systém pro centrální správu byznys pravidel

Cílem této práce je i centrální správa byznysových pravidel. V sekci 4.8 framework navrhuje využít speciální samostatný nástroj, který umožní administrátorům upravovat či vytvářet byznysové kontexty z jednoho místa za běhu systému. Návrh frameworku však neomezuje konkrétní implementaci nástroje, ani využité technologie.

5.7.1 Popis implementace

Systém pro centrální správu byl implementován jako webová služba v jazyce Java s využitím frameworku Spring Boot¹⁴. Ten umožňuje využití návrhového vzoru Model-View-Controller, který je vhodný pro aplikace s uživatelským rozhraním.

Model systému obsahuje třídu `BusinessContextEditor`, která implementuje návrhový vzor Facade [24]. Umožňuje načtení byznysových kontextů ze vzdálených služeb a jejich následnou úpravu a validaci. K tomu využívá prototyp knihovny pro jazyk Java, díky kterému může kontexty přenášet po síti, načítat a zapisovat ze do DSL a využívat metamodel kontextů.

Pro komfortní obsluhu centrální administrace bylo naprogramováno uživatelské rozhraní pomocí technologií Hypertext Markup Language (HTML) a Cascading Style Sheets (CSS). V uživatelském rozhraní lze zobrazit přehled všech byznysových kontextů a pomocí formulářů je upravovat a vytvářet nové. Detail byznysového kontextu v uživatelském rozhraní je zobrazen na snímku B.1 a formulář pro úpravu kontextu na snímku B.2.

5.7.2 Detekce a prevence potenciálních problémů

Systém pro centrální správu byznysových pravidel musí být schopen předejít případným chybám a zabránit tak nekonzistencím v systému, či jeho výpadku. Návrh frameworku v sekci 4.3.1 identifikuje problémy, které mohou nastat při úpravě nebo vytváření nového byznysového kontextu. Kromě syntaktických chyb, které jsou detekovány automaticky pomocí definovaného schématu DSL, je potřeba detektovat následující sémantické chyby, které mohou být způsobeny rozšířováním kontextů:

- a) Neunikátní identifikátory byznysových pravidel
- b) Závislosti na neexistujících kontextech
- c) Cyklus v grafu závislostí kontextů

¹⁴<<https://projects.spring.io/spring-boot/>>

Unikátnost byznysových pravidel lze zajistit postupným ukládáním jejich identifikátorů do vhodné datové struktury a kontrolovat, zda v ní již nejsou obsaženy. Jako vhodná struktura pro tento účel byl zvolen **Set** [32]. Kontexty a jejich vzájemné závislosti lze vnímat jako orientovaný graf, kde uzel grafu reprezentuje kontext a orientovaná hrana reprezentuje závislost mezi kontexty. Směr závislosti lze pro tento účel zvolit libovolně. Pro detekci závislosti na neexistujících kontextech je nejprve sestaven seznam existujících kontextů a následně jsou navštíveny jednotlivé hrany grafu kontextů a je ověřeno, zda existují oba kontexty náležící dané hraně. Díky zvolení vhodných datových struktur bylo dosaženo lineární časové složitosti v závislosti na počtu hran grafu. Pro detekci cyklů v grafu závislosti pravidel popsanou v sekci 4.3.1 byl zvolen modifikovaný algoritmus prohledávání do hloubky (z anglického *depth-first search*). Ten umožňuje detekci zpětných hran a má lineární složitost závislou na součtu počtu hran a počtu uzlů grafu. V případě, že zápis nového či upraveného kontextu obsahuje syntaktické nebo sémantické chyby, administrace nedovolí uživateli změnu provést a vypíše informativní chybovou hlášku.

5.8 Shrnutí

Na základě navrženého frameworku byly implementovány prototypy knihoven pro platformy jazyka Java, jazyka Python a Node.js. Knihovny umožňují centrální správu a automatickou distribuci a integraci byznysových kontextů, včetně vyhodnocování jejich pravidel, za použití aspektově orientovaného přístupu. V rámci této kapitoly byl specifikován **DSL**, kterým lze popsat byznys kontext nezávisle na platformě. Implementované prototypy knihoven lze využít k implementaci služeb a k sestavení funkčního systému, jak je popsáno v následující kapitole.

Veškerý kód implementace je hostován v centrálním repozitáři ve službě GitHub¹⁵ a je zpřístupněn pod open-source licencí **MIT**¹⁶. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

¹⁵<<https://github.com/klimesf/diploma-thesis>>

¹⁶<<http://www.gnu.org/licenses/mitlicense.html>>

Kapitola 6

Testování a validace frameworku

Tato kapitola popisuje, jakým způsobem bylo provedeno testování naprogramovaných knihoven. Kapitola se dále věnuje popisu ukázkového systému, na kterém byla demonstrována správná funkčnost navrženého frameworku a provedena analýza vlivu jeho nasazení na duplikaci byznysových pravidel.

6.1 Techniky testování

Pro testování prototypů knihoven bylo využito několik technik testování, které pomáhají odhalit chyby a zvýšit kvalitu výsledného software.

6.1.1 Jednotkové testy

Jednotkové testy jsou nejnižší úrovní testování, která má za úkol ověřit správnou funkčnost dílčích částí systémů – tzv. *jednotek*. Tato technika se soustředí na odhalení chyb v implementaci dané jednotky. Jednotkové testy umožňují odhalit chyby v dřívějších fázích vývoje [41]. Usnadňují také následnou integraci jednotek, při které lze využít předpokladu, že jednotky fungují správně.

6.1.2 Integrační testy

Integrační testy ověřují správnou spolupráci jednotlivých komponent programu a mají za úkol odhalit chyby v jejich komunikaci a nekonzistence v jejich rozhraních. Existuje více strategií integračního testování. Při integraci lze inkrementálně postupovat od implementovaných jednotek směrem k celku, tzv. *bottom-up*, nebo směrem od celku k jednotkám, tzv. *top-down*, kdy je chování dosud neimplementovaných jednotek simulováno [30]. Integrovat lze i celý systém najednou, tato metoda se nazývá *big-bang*. V tomto případě je ale těžké detektovat původ nalezených chyb.

6.1.3 Systémové testy

Systémové testy mají za úkol ověřit, že výsledný program splňuje specifikaci a funguje jako celek. Tato metoda se věnuje jak funkčním, tak nefunkčním vlastnostem software [30]. Systémové testy patří narozdíl od výše uvedených technik do kategorie *black-box*, tzn. neznají vnitřní strukturu testovaného programu.

6.1.4 Průběžná integrace

Průběžná integrace (z anglického *continuous integration* – CI) [28] slouží k urychlení nalezení defektů software ve fázi vývoje. Nové verze zdrojového kódu jsou do projektu pravidelně integrovány v krátkých časových intervalech a jejich funkčnost je ověřována automatickými testy. K tomu je zpravidla využíván tzv. *integrační server*, na kterém jsou testy spouštěny. Díky CI je snížena pravděpodobnost regrese a dlouhodobě zvýšena celková kvalita kódů.

6.2 Testování prototypů knihoven

Prototypy knihoven, jejichž implementace je popsána v kapitole 5, byly důkladně otestovány pomocí sady jednotkových a integračních testů a tím byla ověřena jejich správná funkcionality. Způsob testování knihoven je popsán zvlášť pro každou platformu. Jako systémový test pak sloužil ukázkový příklad, který je popsán v následující sekci 6.3.

V rámci CI byl kód knihoven po celou dobu vývoje verzován systémem Git¹, zaslán do centrálního repozitáře a s pomocí nástroje Travis CI² bylo automaticky spouštěno jeho sestavení a testování. Tento nástroj zároveň vývojáře okamžitě informoval o výsledcích tohoto procesu.

6.2.1 Testovací scénáře

Každá z knihoven byla testována stejnou sadou testovacích scénářů, skládající se z jednotkových i integračních testů. Scénáře jsou shrnutы v tabulce 6.1. K implementaci a vyhodnocení testů byly pro jednotlivé knihovny použity různé technologie, které jsou popsány v následujícím textu.

¹<<https://git-scm.com/>>

²<<https://travis-ci.org/>>

#	Popis
TC01	Pouze identifikátor byznysového kontextu obsahující alfanumerický prefix a název je validní
TC02	Precondition weaver zkontroluje všechny preconditions vztahující se k aktuálnímu kontextu
TC03	Precondition weaver nevyhodí výjimku, pokud jsou všechny preconditions splněny
TC04	Precondition weaver vyhodí výjimku, pokud alespoň jedna precondition není splněna
TC05	Post-condition weaver aplikuje všechny post-conditions vztahující se k aktuálnímu kontextu
TC06	Post-condition weaver správně filtruje atribut objektu
TC07	Post-condition weaver správně filtruje pole hodnot
TC08	Post-condition weaver správně filtruje atributy objektů v poli
TC09	Logický výraz <code>And</code> odpovídá logické konjunkci
TC10	Logický výraz <code>Equals</code> odpovídá logické ekvivalence
TC11	Logický výraz <code>Negate</code> odpovídá logické negaci
TC12	Logický výraz <code>Or</code> odpovídá logické disjunkci
TC13	Výraz <code>Constant</code> správně doplňuje do pravidla konstantu
TC14	Výraz <code>FunctionCall</code> správně volá externí funkci
TC15	Výraz <code>IsNotNull</code> správně kontroluje, zda v jeho argumentu není prázdný výraz
TC16	Výraz <code>IsNotBlank</code> správně kontroluje, zda v jeho argumentu není prázdný řetězec
TC17	Výraz <code>ObjectPropertyReference</code> správně vkládá do výrazu hodnotu atributu objektu z kontextu
TC18	Výraz <code>VariableReference</code> správně vkládá do výrazu hodnotu proměnné z kontextu
TC19	Byznysové kontexty jsou korektně inicializovány, jejich závislosti staženy a spojeny
TC20	Byznysové kontexty jsou správně a kompletně přenášeny mezi službami
TC21	Při dokončení transakce se správně uloží upravené i nové byznysové kontexty
TC21	Při zrušení transakce se neuloží upravené ani nové byznysové kontexty
TC22	Spouštění byznysových operací je pozastaveno při probíhající transakci
TC23	Byznysové kontexty jsou správně deserializovány z XML
TC24	Byznysové kontexty jsou správně serializovány do XML

Tabulka 6.1: Testovací scénáře prototypu knihoven

6.2.2 Platforma Java

Prototyp knihovny pro platformu jazyka Java byl testován pomocí nástroje JUnit³, který poskytuje veškerou potřebnou funkcionality pro jednotkové i integrační testování. Všechny testy byly spuštěny automaticky při sestavování knihovny pomocí nástroje Maven⁴.

Zdrojový kód 6.1: Jednotkový test knihovny pro jazyk Java s využitím nástroje JUnit 4

```
1 import org.junit.Assert;
2 import org.junit.Test;
3
4 public class BusinessContextWeaverTest {
5
6     @Test
7     public void test() {
8         BusinessContextWeaver evaluator =
9             new BusinessContextWeaver(createRegistry());
10        BusinessOperationContext context =
11            new BusinessOperationContext("user.create");
12
13        context.setOutput(new User(
14            "John Doe",
15            "john.doe@example.com"
16        ));
17        evaluator.applyPostConditions(context);
18        User user = (User) context.getOutput();
19
20        Assert.assertEquals("John Doe", user.getName());
21        Assert.assertNull(user.getEmail());
22    }
23 }
```

Ve zdrojovém kódu 6.1 je znázorněn jednotkový test třídy `BusinessContextWeaver` odpovídající testovacímu scénáři *TC06*, konkrétně filtrování pole `email` objektu `user`. Anotace `@Test` metody `test()` značí, že metoda obsahuje *test case* a framework JUnit zajistí, že bude spuštěna a vyhodnocena. Statické metody třídy `Assert` ověří, zda uživateli zůstalo vyplněno jméno, ale emailová adresa ne.

³<<https://junit.org/junit4/>>

⁴<<https://maven.apache.org/>>

6.2.3 Platforma Python

Prototyp knihovny pro platformu jazyka Python byl testován pomocí nástroje `unittest`⁵, inspirovaného nástrojem JUnit. Ačkoliv jméno obou nástrojů nasvědčuje, že slouží zejména pro jednotkové testy, lze je plně využít i pro integrační testy.

Zdrojový kód 6.2: Jednotkový test knihovny pro jazyk Python s využitím nástroje Unittest

```

1 import unittest
2 from business_context.identifier import Identifier
3
4
5 class IdentifierTest(unittest.TestCase):
6     def test_split(self):
7         identifier = Identifier("auth", "loggedIn")
8         self.assertEqual("auth", identifier.prefix)
9         self.assertEqual("loggedIn", identifier.name)
10
11    def test_single(self):
12        identifier = Identifier("auth.loggedIn")
13        self.assertEqual("auth", identifier.prefix)
14        self.assertEqual("loggedIn", identifier.name)
15
16    def test_str(self):
17        identifier = Identifier("auth.loggedIn")
18        self.assertEqual('auth.loggedIn', identifier.__str__())

```

Ve zdrojovém kódu 6.2 je příklad jednotkového testu třídy `Identifier` se třemi metodami ověřujícími testovací scénář *TC01*. Funkce `test_split()` ověruje, zda konstruktor správně přijímá dva argumenty, kde první z nich je prefix identifikátoru a druhý je jméno identifikátoru. Funkce `test_single()` naopak ověruje, zda konstruktor správně přijímá jeden argument a rozdělí ho na prefix a jméno identifikátoru. Nakonec funkce `test_str()` ověruje správnou funkcionality převedení identifikátoru na textový řetězec.

6.2.4 Platforma Node.js

Tendence ve světě moderního JavaScriptu je vytvářet knihovny s co nejmenším polem působnosti, které jdou kombinovat do většího celku. Prototyp knihovny pro platformu Node.js

⁵<<https://docs.python.org/3/library/unittest.html>>

byl proto testován pomocí kombinace dvou nástrojů. Spouštění testů obstarává knihovna *Mocha*⁶, zatímco ověřování a zápis testů ve stylu *Behaviour Driven Development* poskytuje knihovna *Chai*⁷.

Zdrojový kód 6.3: Jednotkový test knihovny pro platformu Node.js s využitím nástroje Mocha a Chai

```
1 const chai = require('chai')
2 // Other imports
3
4 chai.should();
5
6 describe('IsNotNull', () => {
7     describe('#interpret', () => {
8         it('returns true if the argument is null', () => {
9             const ctx = new BusinessOperationContext('user.create')
10            const expression = new IsNotNull(new Constant(
11                true,
12                ExpressionType.BOOL
13            ))
14            const result = expression.interpret(ctx)
15            result.should.equal(true)
16        })
17    })
18})
```

Zdrojový kód 6.3 znázorňuje použití testovacích knihoven k implementaci testovacího scénáře *TC15* ověřující funkcionality výrazu *IsNotNull*. Konkrétně je výraz zkonstruován s konstantním argumentem typu *boolean* s hodnotou *true* a je ověřeno, že výraz se vyhodnotí jako *true*.

6.3 Ukázkový příklad: e-commerce systém

Pro systémové testování a validaci navrženého frameworku bylo nutné vyzkoušet jeho nasazení při vývoji aplikace. Pro tento účel vznikl v rámci této práce ukázkový příklad na fiktivním e-commerce systému využívajícím architekturu orientovanou na služby. Na tomto

⁶<<https://mochajs.org/>>

⁷<<http://www.chaijs.com/>>

příkladě je demonstrována schopnost frameworku poradit si s průřezovými problémy v rámci [SOA](#) a také jeho schopnost plnit požadavky identifikované v sekci [2.4](#).

Ukázkový systém simuluje reálný e-shop, který se skládá z několika služeb a umožňuje zákazníkovi prohlížet a objednávat zboží online. Zaměstnancům umožňuje spravovat produkty a jejich skladové zásoby a vystavovat faktury k objednávkám. Administrátorům systém umožňuje nastavovat cenovou politiku produktů a vytvářet a mazat uživatele.

6.3.1 Use-cases

Pro ukázkový systém bylo vymodelováno třináct případů užití (z anglického *Use case* ([UC](#)) [5]), jejich přehled je v tabulce [6.2](#).

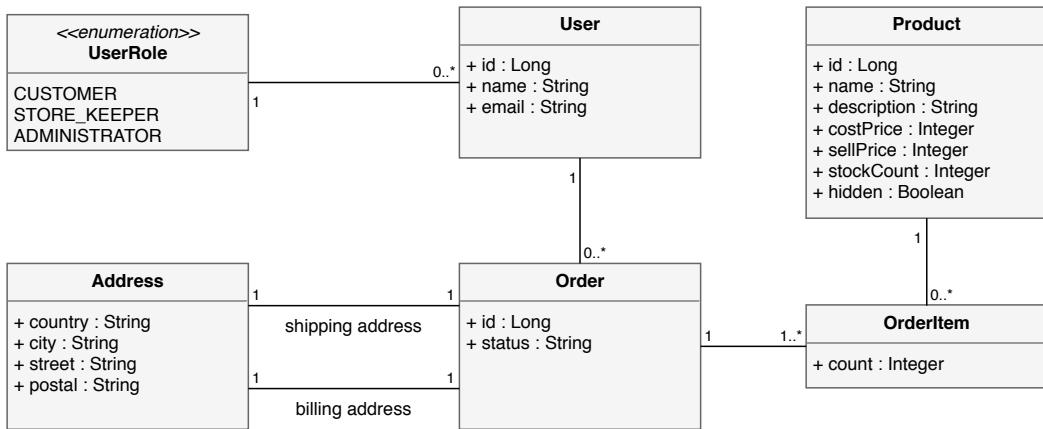
#	Use-case
UC01	Nepřihlášený uživatel si může vytvořit zákaznický účet
UC02	Zákazník může prohlížet produkty
UC03	Zákazník může vkládat produkty do košíku
UC04	Zákazník může vytvořit objednávku
UC05	Skladník může do systému zadávat nové produkty
UC06	Skladník může upravovat u produktů stav skladových zásob
UC07	Skladník si může zobrazovat objednávky
UC08	Skladník může upravovat stav objednávek
UC09	Administrátor si může prohlížet objednávky
UC10	Administrátor může upravovat cenu produktů
UC11	Administrátor může vytvářet uživatele (skladníky)
UC12	Administrátor může mazat uživatele (skladníky i zákazníky)
UC13	Administrátor může vystavit fakturu

Tabulka 6.2: Případy použití ukázkového e-commerce systému

6.3.2 Model systému

Na obrázku [6.1](#) je diagram tříd reprezentujících kompletní doménový model ukázkového systému.

- **UserRole** reprezentuje uživatelskou roli v systému.
- **User** je entita odpovídající uživateli, ať už zákazníkovi, či zaměstnanci.
- **Product** popisuje konkrétní produkt v nabídce společnosti a jeho nákupní a prodejní cenu.



Obrázek 6.1: Doménový model ukázkového e-commerce systému

- **Order** odpovídá objednávce, má vazbu na dodací a fakturační adresu a také na položky objednávky.
- **OrderItem** reprezentuje položku objednávky a uchovává údaje o počtu objednaných kusů produktu.
- **Address** je entita popisující dodací či fakturační adresu.

Tento model je využíván v každé ze služeb. Nicméně, jednotlivé služby využívají pouze jeho podmnožinu, kterou potřebují ke svojí práci.

6.3.3 Byznysová pravidla a kontexty

V tabulce 6.3 je výčet všech byznysových pravidel, která byla vymodelována pro ukázkovou aplikaci. V tabulce kromě identifikátoru a popisu byznysového pravidla vidíme, na které užitné případy se pravidlo aplikuje, a jaký je typ pravidla (*pre* pro precondition, *post* pro post-condition).

Dále jsou v tabulce 6.4 vypsány všechny byznysové kontexty v ukázkové aplikaci. Některé z nich jsou konkrétní a jsou namapovány na jeden nebo více UC, jiné jsou abstraktní a ostatní kontexty od nich dědí a sdílejí jejich byznysová pravidla. Prefixy byly vybrány na základě byznysové domény, ke které se kontext vztahuje, stejně jako jsou podle domén děleny i jednotlivé služby systému.

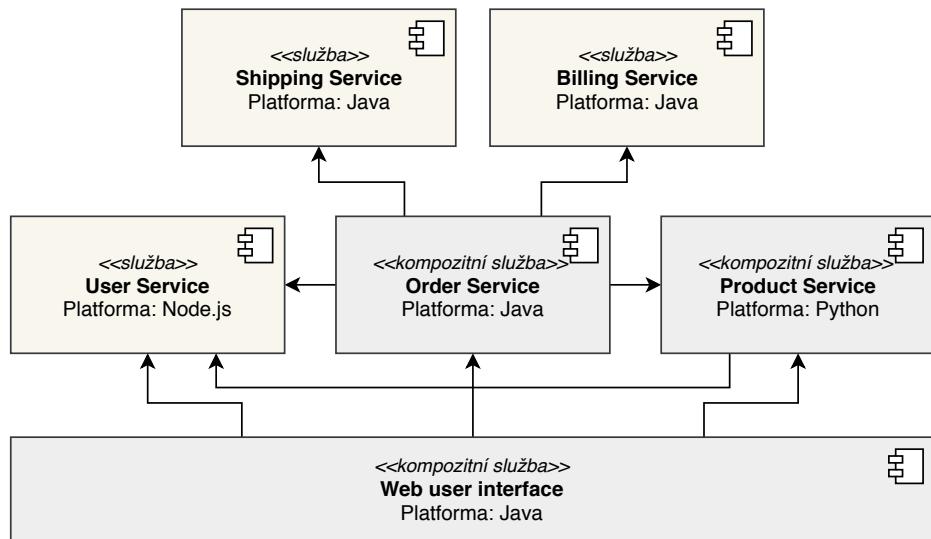
Na obrázku B.4 je vizualizována hierarchie byznysových kontextů v ukázkovém systému, jejich vazba na UC a také byznysová pravidla, která se v kontextech aplikují.

#	Use-cases	Pravidlo	Typ
BR01	UC01	Uživatel nesmí být přihlášený	pre
BR02	UC02, UC03	Uživatel nesmí zobrazovat ani manipulovat s produkty, které nejsou aktivní	post
BR03	UC02 až UC04	Uživatel nesmí u produktu vidět nákupní cenu, pouze výslednou cenu	post
BR04	UC04	Zákazník musí řádně vyplnit doručovací adresu (č.p., ulice, město, PSČ, stát)	pre
BR05	UC04, UC13	Zákazník musí řádně vyplnit fakturační adresu (č.p., ulice, město, PSČ, stát)	pre
BR06	UC01, UC04, UC11	Uživatel musí mít vyplněnou emailovou adresu	pre
BR07	UC03	Nákupní košík může obsahovat maximálně 10 položek	pre
BR08	UC04	Položky objednávky musí mít počet kusů menší, než je aktuální stav skladových zásob produktu	pre
BR09	UC04	Stát musí být v seznamu zemí, do kterých firma doručuje	pre
BR10	UC04	Zákazník musí být přihlášen	pre
BR11	UC05 až UC08	Skladník musí být do systému přihlášen a mít roli "Skladník"	pre
BR12	UC02	Skladník u produktu nesmí vidět nákupní cenu, pouze výslednou cenu	post
BR13	UC05	Produkt musí mít jméno s délkou >5	pre
BR14	UC06	Stav zásob produktů musí být číslo větší nebo rovno 0	pre
BR15	UC08	Stav objednávky musí být pouze "přijato", "expedováno" a "doručeno"	pre
BR16	UC09 až UC13	Administrátor musí být do systému přihlášen a mít roli "Administrátor"	pre
BR17	UC10	Výsledná cena produktu musí být větší než jeho nákupní cena	pre
BR18	UC11	Skladník musí mít jméno delší než 2 znaky	pre
BR19	UC12	Smazený uživatel nesmí být administrátor	pre

Tabulka 6.3: Byznysová pravidla ukázkového e-commerce systému

Identifikátor	Use-cases	Byznysová pravidla
auth.adminLoggedIn	-	BR16
auth.employeeLoggedIn	-	BR11
auth.userLoggedIn	-	BR10
billing.correctAddress	-	BR05
billing.createInvoice	UC13	BR05
order.addToShippingCart	UC03	BR02, BR07, BR08, BR10
order.changeStatus	UC08	BR04, BR05, BR06, BR08, BR09, BR11, BR15
order.create	UC04	BR03, BR04, BR05, BR06, BR07, BR08, BR09, BR10, BR15
order.listAll	UC07, UC09	BR11
order.valid	-	BR04, BR05, BR06, BR09, BR15
product.changePrice	UC10	BR16, BR17
product.changeStock	UC06	BR08, BR11, BR14
product.create	UC05	BR10, BR11, BR13, BR14
product.detail	UC02	BR02, BR03, BR10, BR12
product.hidden	-	BR02
product.listAll	UC02	BR02, BR03, BR12
shipping.correctAddress	-	BR04, BR09
user.createCustomer	UC01	BR01, BR06
user.createEmployee	UC11	BR06, BR16, BR18
user.delete	UC12	BR16, BR19
user.validEmail	-	BR06

Tabulka 6.4: Byznysové kontexty ukázkového e-commerce systému



Obrázek 6.2: Komponenty ukázkového e-commerce systému

6.3.4 Služby

Na obrázku 6.2 jsou zobrazeny komponenty systému a jejich vzájemné závislosti. Pro ověření schopnosti podporovat více platform byly pro implementaci systému využity jazyky Java, Python a JavaScript v kombinaci s běhovým prostředím Node.js. Komunikace služeb probíhá pomocí REST API využívající formát JSON. Specifikace jednotlivých rozhraní služeb není pro tuto kapitolu podstatná, a proto se jí text dále nevěnuje. Pro demonstrativní účely byly síťové adresy nastaveny přímo v kódu jednotlivých služeb. Nicméně, navržený framework nevynucuje tento přístup, a tudíž by bylo možné komplexnější způsob *service discovery* do systému integrovat.

Billing service Služba *Billing service* má na starosti funkcionality týkající se fakturace objednávek a byla implementována v jazyce Java s použitím frameworku Spring Boot⁸.

Order service Kompozitní služba *Order service* sloužící pro vytváření a správu objednávek byla implementována v jazyce Java a její API bylo sestaveno za použití frameworku Spring Boot, jak je ukázáno ve zdrojovém kódu 6.4, kde je ukázka obsluhy požadavků na výpis zboží v košíku uživatele.

Zdrojový kód 6.4: Ukázka využití frameworku Spring Boot pro účely Order service

```

1  @RestController
2  public class ShoppingCartController {
```

⁸<<https://projects.spring.io/spring-boot/>>

```
3     @GetMapping("/shopping-cart")
4     public ResponseEntity<?> listShoppingCart() {
5         List<ShoppingCartItem> shoppingCartItems = shoppingCartFacade
6             .listShoppingCartItems();
7         return new ResponseEntity<>(
8             new ListShoppingCartItemsResponse(
9                 shoppingCartItems.size(),
10                shoppingCartItems
11            ),
12            HttpStatus.OK
13        );
14    }
15
16 }
```

Product service Služba *Product service* realizuje [UC](#) týkající se prohlízení a adminis-trací nabízených produktů a jejich skladových zásob. Služba byla implementována v jazyce Python. Pro vytvoření [REST API](#) služby byl využit light-weight framework *Flask*⁹. Ve zd-rojovém kódu 6.5 je znázorněno použití tohoto frameworku pro obsluhu požadavku na výpis všech produktů.

Zdrojový kód 6.5: Ukázka využití frameworku Flask pro účely Product service

```
1 from flask import Flask, jsonify
2 from model import ProductRepository
3
4 app = Flask(__name__)
5 product_repository = ProductRepository()
6
7 @app.route("/")
8 def list_all_products():
9     result = []
10    for product in product_repository.get_all():
11        result.append({
12            'id': product.id,
13            'sellPrice': product.sellPrice,
```

⁹<<http://flask.pocoo.org/>>

```

14         'name': product.name,
15         'description': product.description
16     })
17     return jsonify(result)

```

Shipping service Služba *Shipping service* má na starosti funkcionalitu týkající se odesílání objednávek a byla implementována v jazyce Java s použitím frameworku Spring Boot.

User service Služba *User service* realizující funkcionalitu týkající se uživatelských účtů byla implementována v jazyce JavaScript na platformě Node.js s použitím frameworku Express¹⁰. Ve zdrojovém kódu 6.6 je ukázka mapování controllerů na URL /users a různé metody protokolu HTTP.

Zdrojový kód 6.6: Ukázka využití frameworku Express.js pro účely User service

```

1 module.exports = app => {
2     const userController = require('../controllers/userController')
3
4     app.route('/users')
5         .get(userController.listUsers)
6         .post(userController.register)
7         .delete(userController.deleteUser)
8
9     app.route('/users/:userId')
10        .get(userController.getUser)
11 }

```

Webové uživatelské rozhraní Služba, která slouží uživatelům ukázkového systému jako webové uživatelské rozhraní, byla implementována v jazyce Java s použitím frameworku Spring Boot. Na snímku B.3 je vidět UI ukázkového systému, konkrétně informování uživatele o tom, že se nepodařilo přidat produkt do košíku, protože bylo porušeno byznysové pravidlo – košík nesmí obsahovat více než 10 položek.

Centrální správa byznysových pravidel Do ukázkového systému byl nasazen také systém pro centrální správu byznysových kontextů, který je popsáný v sekci 5.7. Systém byl

¹⁰<<https://expressjs.com/>>

napojen na všechny služby systému, kromě webového UI, a bylo úspěšně demonstrováno, že lze za běhu systému dynamicky upravovat byznysové kontexty, resp. jejich byznysová pravidla.

Běhové prostředí služeb Pro jednoduché spuštění celého ukázkového systému byla využita technologie Docker [44], která umožňuje vytvořit virtuální běhové prostředí pro aplikaci pomocí kontejnerizace využívající virtualizaci nad operačním systémem. Uživatel si nadefiniuje tzv. *image*, který se skládá z jednotlivých vrstev. Základní vrstvou je operační systém, dalšími mohou být jednotlivé knihovny instalované do systému.

Spouštění služeb Pro samotné spuštění byla využita funkce *Docker Compose*, která umožňuje definovat a spouštět aplikace s více kontejnery. Ve zdrojovém kódu 6.7 je příklad zápisu Order service. Pro její image je použit filipklices-diploma/example-order-service. V sekci *ports* je deklarováno, že služba má mít z vnějšku přístupný port 5501, na kterém poskytuje své REST API, a port 5551, na kterém poskytuje své gRPC API pro sdílené byznysových kontextů. V sekci *links* je deklarováno, že pro kontejner, ve kterém Order Service poběží, mají být na síti přístupné služby *product*, *user*, *billing* a *shipping*. Vše je popsáno ve formátu YAML, který je vhodný pro konfigurační soubory díky jeho snadné čitelnosti pro člověka a jednoduchému používání.

Zdrojový kód 6.7: Ukázka zápisu aplikace s více kontejnery pro Docker Compose

```
 1 version: '3'
 2 services:
 3   order:
 4     image: filipklices-diploma/example-order-service
 5     ports:
 6       - "5501:5501"
 7       - "5551:5551"
 8     links:
 9       - product
10       - user
11       - billing
12       - shipping
```

#	Použito ve službách	#	Použito ve službách
BR01	user	BR11	auth, order, product
BR02	order, product	BR12	product
BR03	order, product	BR13	product
BR04	order, shipping	BR14	product
BR05	billing, order	BR15	order
BR06	order, user	BR16	(auth), product, user
BR07	order	BR17	product
BR08	order	BR18	user
BR09	order, shipping	BR19	user
BR10	(auth), order, product		

Tabulka 6.5: Využití byznysových pravidel ve službách ukázkového systému

6.4 Srovnání s konvenčním přístupem

Z tabulky 6.4 plyne, že čtrnáct z devatenácti byznysových pravidel je využito ve více byznysových kontextech a devět pravidel je sdíleno mezi více službami. V tabulce 6.5 je přehledně shrnuto, která pravidla jsou využívána ve kterých službách. Při použití konvenčního přístupu by tato pravidla bylo nutné implementovat alespoň jednou pro každou ze služeb, za předpokladu, že by nedocházelo k duplikacím ve službách samotných. Manuální duplikace navíc přináší nutnost synchronizovat podobu pravidla při každém změnovém řízení, což zvyšuje náklady na vývoj a riziko lidské chyby.

Díky použití navrženého frameworku je však možné každé pravidlo na definovat na jednom místě a framework se postará o jeho automatickou integraci do všech částí systému, kde má být aplikováno. To umožňuje byznysová pravidla, resp. kontexty, spravovat pomocí nástroje pro centrální správu, který je součástí navrženého frameworku. Z toho vyplývá snížení nároků na vývoj a snížené riziko lidské chyby.

Jako nevýhodu použití frameworku lze považovat počáteční investici v podobě integrace knihoven do služeb systému, a cena popisu byznysových pravidel v [DSL](#), který se musejí vývojáři systému naučit navíc oproti programovacímu jazyku, ve kterém popisují služby. Tento jazyk však mohou používat i doménoví experti, a tak ulehčit práci vývojářů. Framework také není vhodné nasazovat v systémech, jejichž doména nemá tendenci sdílet byznysová pravidla.

Navržený framework tedy oproti konvenčnímu přístupu nabízí možnost získat dlouhodobě nižší náklady na vývoj za cenu počáteční investice. Díky provedené případové studii bylo ukázáno, že v [SOA](#) lze efektivně řešit sdílení byznysových pravidel navrženým způsobem.

6.5 Shrnutí

Tato kapitola popisuje, jakým způsobem byly testovány prototypy knihoven pro platformy jazyků Java a Python a pro platformu Node.js. Dále kapitola specifikuje ukázkový systém, na kterém byla provedena demonstrace použití frameworku, a popisuje jeho implementaci. Tím bylo otestováno, že navržený framework je funkční, splňuje požadavky identifikované v sekci 2.4 a prokazuje reálné snížení manuální duplikace byznysových pravidel oproti konvenčnímu přístupu k návrhu a implementaci softwarových systému.

Kapitola 7

Závěr

Architektura orientovaná na služby člení funkcionality komplexních informačních systémů do dílčích služeb. Díky tomu lze lépe oddělit zodpovědnost a zvýšit znovupoužitelnost jednotlivých komponent systému. Existuje však funkctionalita, která zasahuje do více služeb najednou, a je potřeba ji všude vykonávat konzistentně. Zástupcem této funkctionality jsou byznysová pravidla, která zajišťují validní vykonávání byznysových procesů a konzistence dat uložených v systému. Při využití stávajících přístupů je potřeba tato pravidla ve službách manuálně duplikovat, což zvyšuje náklady spojené s vývojem a údržbou takových systémů.

Tato práce se věnuje problematice byznysových pravidel v architektuře orientované na služby a navrhuje způsob, jakým lze usnadnit práci vývojářů a administrátorů pomocí centrální správy pravidel a jejich automatické distribuce a integrace. K tomu využívá aspektově orientovaného programování a na něm založeného přístupu [ADDA](#).

7.1 Dosažené výsledky

V rámci této práce byly dosaženy cíle stanovené v úvodu práce a byly splněny všechny požadavky zadání. Nejprve bylo analyzováno využití, vyjádření a znovupoužití byznysových pravidel v [SOA](#) a byly identifikovány potenciální problémy a z nich vyplývající požadavky na framework navržený v této práci. Dále byla provedena rešerše vhodných architektur, paradigm a nástrojů, které by mohly být použity pro řešení těchto problémů, včetně stávajících frameworků pro reprezentaci a sdílení byznysových pravidel a jejich výhod a nevýhod. Na základě analýzy a rešerše byl navržen framework pro správu a automatickou integraci byznysových pravidel v architektuře orientované na služby. Tento framework využívá koncepty aspektově orientovaného programování a na něm založeného přístupu [ADDA](#). Funkčnost navrženého řešení byla demonstrována implementací a otestováním prototypů knihoven pro

platformy jazyků Java a Python a platformu Node.js¹. Tyto knihovny byly použity pro vývoj jednoduché ukázkové e-commerce aplikace, která je tvořena šesti službami implementovanými ve třech různých programovacích jazycích. Na této aplikaci byla ukázána schopnost frameworku centrálně administrovat byznysová pravidla a automaticky je sdílet a integrovat do jednotlivých služeb aplikace. Zároveň byl změren a analyzován dopad použití frameworku na počet duplikací byznysových pravidel a byl diskutován jeho vliv na údržbu systému. V závěru práce jsou navíc analyzovány další oblasti SOA, kde by bylo možné aplikovat aspektově orientované programování.

7.2 Přínos a možnosti použití frameworku

Framework navržený v této práci přináší způsob, kterým mohou vývojáři systémů stojících na SOA výrazně ušetřit náklady spojené s manuální duplikací byznysových pravidel v jednotlivých službách. Tento framework je nezávislý na platformě a díky tomu může být využíván i v technologicky heterogenních systémech, jejichž služby využívají více programovacích jazyků. Ačkoliv se text této práce odkazuje zejména na třívrstvou architekturu, framework neklade omezení na architekturu jednotlivých služeb.

Pro využití frameworku musejí vývojáři zachytit byznysová pravidla ve speciálním DSL. Díky frameworku ale zvyšují znovupoužitelnost pravidel, ulehčují jejich centrální administraci a realizaci změnových požadavků. Tato počáteční investice se vyplatí od určité velikosti systému, kdy náklady na manuální duplikaci pravidel přesáhnou cenu za návrh a využití DSL.

7.3 Budoucí rozšiřitelnost frameworku

V této práci se podařilo implementovat prototyp frameworku a využít ho k centrální správě byznysových pravidel a jejich automatické integraci do jednotlivých služeb. Potenciál navrženého řešení je však mnohem větší. Framework lze do budoucna rozšířit v dalších směrech a tím dále snížit náklady na vývoj a údržbu informačních systémů.

7.3.1 Univerzální doménově specifický jazyk

Cílem této práce nebylo zkonstruovat vlastní DSL k účelům automatické integrace a centrální správy byznysových pravidel, nicméně v sekci 2.4 byla potřeba takového jazyka identifikována. Kapitola 3 došla k závěru, že momentálně neexistuje vhodné DSL, které by splňovalo všechny požadavky kladené na navržený framework. V rámci implementace prototypu knihoven bylo navrženo a implementováno DSL v jazyce XML, popsáné v sekci 5.2. Tento jazyk je však

¹Uživatelská a instalovační příručka knihoven je součástí obsahu přiloženého CD

omezený a slouží pouze jako nástroj pro demonstraci navrženého řešení. Sestavení komplexního jazyka pro popis byznysových kontextů je proto vhodným rozšířením frameworku. Ten je k tomu navíc plně připraven.

7.3.2 Integrace frameworku s uživatelským rozhraním

V sekci 4.9 je popsána architektura frameworku, která umožňuje využití ve všech třech standardních vrstvách EIS. Autoři přístupu ADDA již vyvinuli způsob, kterým lze integrovat vyhodnocování byznysových pravidel do uživatelského rozhraní. Propojení s navrženým frameworkem by znamenalo implementovat adaptér, který by převáděl reprezentaci byznysového pravidla do podoby, kterou je schopen využívat aspect weaver v UI. Díky tomu bylo umožněno další snížení nákladů na vývoj a údržbu systému využívající framework. Zároveň by došlo ke zvýšení uživatelského komfortu díky dynamické validaci vstupních hodnot formulářů na straně klienta.

7.3.3 Integrace frameworku s datovou vrstvou

Integrace do datové vrstvy EIS je další z možností budoucí rozšiřitelnosti navrženého frameworku. Podobně jako v případě UI, autoři přístupu ADDA navrhují způsob, kterým lze automaticky distribuovat post-conditions do datové vrstvy transformováním jejich podmínek do výrazů v jazyce SQL. Implementací a napojením příslušného aspect weaveru na navržený framework by byla pokryta další oblast, ve které může docházet k manuální duplikaci byznysových pravidel.

7.4 Další možnosti uplatnění AOP v SOA

Byznysová pravidla nejsou jediným průřezovým problémem, se kterým se systémy stávající na SOA musejí vypořádat. Jak bylo popsáno v sekci 3.4.1, autoři přístupu ADDA identifikují strukturu doménového modelu jako průřezový problém zasahující do všech standardních vrstev EIS, zejména pak do UI. V SOA může být struktura doménového modelu navíc sdílena mezi jednotlivými službami, což se promítá zejména do rozhraní, pomocí kterých spolu komunikují. Díky AOP by bylo možné automaticky integrovat tuto strukturu do kódu, který komunikaci obsluhuje.

Dalším průřezovým problémem, kterému se autoři ADDA ve svém výzkumu věnují, je extrakce dokumentace. V rámci SOA by se pak AOP dalo využít k extrakci informací o doménovém modelu, implementovaných use-cases a byznysových pravidlech a následnému automatickému generování dokumentace.

Za průřezový problém lze v **SOA** považovat také společnou konfiguraci parametrů služeb, zejména pak v decentralizovaném systému využívajícím architekturu Microservices. Příkladem takového parametru může být v ukázkovém e-commerce systému výše DPH, která je využita v objednávkové i fakturační službě. Pomocí centrální, automaticky distribuované konfigurace by se dalo zamezit nekonzistencím a zabránit tak poškozením dat v systému, či jiným závažným chybám.

Literatura

- [1] ANDREWS, T. et al. Business process execution language for web services, 2003.
- [2] BAKSHI, K. Microservices-based software architecture and approaches. In *Aerospace Conference, 2017 IEEE*, s. 1–8. IEEE, 2017.
- [3] BERNARD, E. – PETERSON, S. JSR 303: Bean validation. *Bean Validation Expert Group, March*. 2009.
- [4] BERSON, A. *Client-server architecture*. New York, New York, USA : McGraw-Hill, 1992.
- [5] BITTNER, K. *Use case modeling*. Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.
- [6] BOX, D. et al. Simple object access protocol (SOAP) 1.1, 2000.
- [7] BOYEN, N. – LUCAS, C. – STEYAERT, P. Generalized mixin-based inheritance to support multiple inheritance. Technical report, Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.
- [8] BOYER, M. J. e. r. o. m. – MILI, H. IBM websphere ilog jrules. In *Agile business rule development*. Cham, Switzerland: Springer, 2011. s. 215–242.
- [9] CEMUS, K. Context-aware input validation in information systems. In *POSTER 2016-20th International Student Conference on Electrical Engineering*, 2016.
- [10] CEMUS, K. – CERNY, T. Aspect-driven design of information systems. In *International Conference on Current Trends in Theory and Practice of Informatics*, s. 174–186. Springer, 2014.
- [11] CEMUS, K. – CERNY, T. Automated extraction of business documentation in enterprise information systems. *ACM SIGAPP Applied Computing Review*. 2017, 16, 4, s. 5–13.

- [12] CEMUS, K. – CERNY, T. – DONAHOO, M. J. Automated business rules transformation into a persistence layer. *Procedia Computer Science*. 2015, 62, s. 312–318.
- [13] CEMUS, K. et al. Distributed Multi-Platform Context-Aware User Interface for Information Systems. In *IT Convergence and Security (ICITCS), 2016 6th International Conference on*, s. 1–4. IEEE, 2016.
- [14] CERNY, T. – DONAHOO, M. J. Survey on concern separation in service integration. In *International Conference on Current Trends in Theory and Practice of Informatics*, s. 518–531. Springer, 2016.
- [15] 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.
- [16] CERNY, T. – DONAHOO, M. J. – TRNKA, M. Contextual understanding of micro-service architecture: current and future directions. *ACM SIGAPP Applied Computing Review*. 2018, 17, 4, s. 29–45.
- [17] CHAPPELL, D. *Enterprise service bus*. Newton, Massachusetts, USA : O'Reilly Media, Inc., 2004.
- [18] CHRISTENSEN, E. et al. Web services description language (WSDL) 1.1, 2001.
- [19] CZARNECKI, K. et al. Generative programming and active libraries. In *Generic Programming*. Cham, Switzerland: Springer, 2000. s. 25–39.
- [20] DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017. s. 195–216.
- [21] DUMAS, M. – AALST, W. M. – TER HOFSTEDE, A. H. *Process-aware information systems: bridging people and software through process technology*. Hoboken, New Jersey, USA : John Wiley & Sons, 2005.
- [22] FIELDING, R. T. REST: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*. 2000.
- [23] FORMAN, I. R. – FORMAN, N. – IBM, J. V. Java reflection in action. 2004.
- [24] FOWLER, M. *Patterns of enterprise application architecture*. Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.
- [25] FOWLER, M. ServiceOrientedAmbiguity. *Martin Fowler–Bliki*. 2005, 1.

- [26] FOWLER, M. *Domain-specific languages*. London, England, UK : Pearson Education, 2010.
- [27] FOWLER, M. – BECK, K. *Refactoring: improving the design of existing code*. Boston, Massachusetts, USA : Addison-Wesley Professional, 1999.
- [28] FOWLER, M. – FOEMMEL, M. Continuous integration. *Thought-Works* <http://www.thoughtworks.com/ContinuousIntegration.pdf>. 2006, 122, s. 14.
- [29] GitHub Octoverse 2017 [online]. 2017. Dostupné z: <<https://octoverse.github.com/>>.
- [30] GRAHAM, D. – VAN VEENENDAAL, E. – EVANS, I. *Foundations of software testing: ISTQB certification*. Andover, United Kingdom : Cengage Learning EMEA, 2008.
- [31] A Hands-on Introduction to BPEL [online]. Dostupné z: <<http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html>>.
- [32] HOPCROFT, J. E. – ULLMAN, J. D. *Data structures and algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing, 1983.
- [33] KENNARD, R. – EDMONDS, E. – LEANEY, J. Separation anxiety: stresses of developing a modern day separable user interface. In *Human System Interactions, 2009. HSI'09. 2nd Conference on*, s. 228–235. IEEE, 2009.
- [34] KICZALES, G. et al. Aspect-oriented programming. In *European conference on object-oriented programming*, s. 220–242. Springer, 1997.
- [35] KLEPPE, A. G. et al. The model driven architecture: practice and promise, 2003.
- [36] KRATZKE, N. – QUINT, P.-C. Understanding cloud-native applications after 10 years of cloud computing-A systematic mapping study. *Journal of Systems and Software*. 2017, 126, s. 1–16.
- [37] LADDAD, R. *AspectJ in action: practical aspect-oriented programming*. New Delhi, India : Dreamtech Press, 2003.
- [38] LARMAN, C. – APPLYING, U. Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2001.
- [39] LEWIS, J. – FOWLER, M. Microservices: a definition of this new architectural term. *MartinFowler. com*. 2014, 25.
- [40] LITTMAN, D. C. et al. Mental models and software maintenance. *Journal of Systems and Software*. 1987, 7, 4, s. 341–355.

- [41] LUO, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA.* 2001, 15232, 1-19, s. 19.
- [42] 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.
- [43] MELICHAR, B. *Jazyky a překlady*. Praha, Česká republika : Vydavatelství ČVUT, 2003.
- [44] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*. 2014, 2014, 239, s. 2.
- [45] MORGAN, T. *Business rules and information systems: aligning IT with business goals*. Boston, Massachusetts, USA : Addison-Wesley Professional, 2002.
- [46] NELSON, B. J. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1981. AAI8204168.
- [47] NEMURAITA, L. – CEPONIENE, L. – VEDRICKAS, G. Representation of business rules in UML&OCL models for developing information systems. In *IFIP Working Conference on The Practice of Enterprise Modeling*, s. 182–196. Springer, 2008.
- [48] PAPAZOGLOU, M. P. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, s. 3–12. IEEE, 2003.
- [49] PARDON, G. – PAUTASSO, C. Towards distributed atomic transactions over RESTful services. In *REST: From Research to Practice*. Cham, Switzerland: Springer, 2011. s. 507–524.
- [50] *Programming Languages and GitHub* [online]. 2014. Dostupné z: <<http://github.info/>>.
- [51] RENTSCH, T. Object oriented programming. *ACM Sigplan Notices*. 1982, 17, 9, s. 51–57.
- [52] RICHARDS, M. Microservices vs. service-oriented architecture. 2015.
- [53] ROSENBERG, F. – DUSTDAR, S. Business rules integration in BPEL-a service-oriented approach. In *E-Commerce Technology, 2005. CEC 2005. Seventh IEEE International Conference on*, s. 476–479. IEEE, 2005.

- [54] SELIC, B. The pragmatics of model-driven development. *IEEE software*. 2003, 20, 5, s. 19–25.
- [55] SHEARD, T. Accomplishments and research challenges in meta-programming. In *International Workshop on Semantics, Applications, and Implementation of Program Generation*, s. 2–44, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [56] SIEGEL, J. – FRANTZ, D. *CORBA 3 fundamentals and programming*. 2. New York, NY, USA : John Wiley & Sons, 2000.
- [57] SOBEL, J. M. – FRIEDMAN, D. P. An introduction to reflection-oriented programming. In *Proceedings of reflection*, 96, 1996.
- [58] SOLEY, R. et al. Model driven architecture. *OMG white paper*. 2000, 308, 308, s. 5.
- [59] SOLOWAY, E. – EHRLICH, K. Empirical studies of programming knowledge. In *Readings in artificial intelligence and software engineering*. New York, USA: Elsevier, 1986. s. 507–521.
- [60] *Stack Overflow Developer Survey 2017* [online]. 2017. Dostupné z: <<https://insights.stackoverflow.com/survey/2017#technology>>.
- [61] *The Role of Service Orchestration Within SOA* [online]. Dostupné z: <<https://www.nomagic.com/news/insights/the-role-of-service-orchestration-within-soa>>.
- [62] VANDEVOORDE, D. – JOSUTTIS, N. M. *C++ Templates*. Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.
- [63] VARDA, K. Protocol buffers: Google’s data interchange format. *Google Open Source Blog, Available at least as early as Jul.* 2008, 72.
- [64] WARD, M. P. Language-oriented programming. *Software-Concepts and Tools*. 1994, 15, 4, s. 147–161.
- [65] WARMER, J. B. – KLEPPE, A. G. The object constraint language: Precise modeling with uml (addison-wesley object technology series). 1998.
- [66] WESKE, M. Business process management architectures. In *Business Process Management*. Cham, Switzerland: Springer, 2012. s. 333–371.
- [67] XIAO, Z. – WIJEGUNARATNE, I. – QIANG, X. Reflections on SOA and Microservices. In *Enterprise Systems (ES), 2016 4th International Conference on*, s. 60–67. IEEE, 2016.

Příloha A

Seznam použitých zkratek

ADDA	Aspect-Driven Design Approach. 15–18 , 21 , 25 , 28 , 34 , 67 , 69
AOP	Aspect Oriented Programming. 14 , 15 , 21 , 34 , 44 , 69
API	Application Programming Interface. 16 , 61 , 62 , 64
BPEL	Business Process Execution Language. 18
BRMS	Business Rules Management System. 17 , 18
CI	Continuous Integration. 52
CIM	Computation Independent Model. 11
CORBA	Common Object Request Broker Architecture. 6
CSS	Cascading Style Sheets. 49
DSL	Domain-Specific Language. 5 , 10 , 15–19 , 21 , 23 , 25–30 , 32 , 34 , 35 , 37 , 39 , 40 , 43–45 , 49 , 50 , 65 , 68
EIS	Enterprise Information System. 3–5 , 7 , 11 , 13 , 16 , 69
ESB	Enterprise Service Bus. 7 , 8
GP	Generative Programming. 12
HTTP	Hypertext Transfer Protocol. 6 , 20 , 63
Java EE	Java Platform, Enterprise Edition. 17
JPQL	Java Persistence Query Language. 16
JSON	JavaScript Object Notation. 41 , 61
JSR	Java Specification Request. 22 , 39
LHS	Left-hand side. 18
LOP	Language-Oriented Programming. 19
MDD	Model-Driven Architecture. 11 , 12 , 19
MIT	Massachusetts Institute of Technology. 50
MQ	Message Queue. 6 , 8
OOP	Object Oriented Programming. 12 , 13 , 15

ORB	Object Request Broker. 6
PIM	Platform Independent Model. 11 , 12
PSM	Platform Specific Model. 11 , 12
REST	Representational State Transfer. 16 , 20 , 29 , 61 , 62 , 64
RHS	Right-hand side. 18
RPC	Remote Procedure Call. 20 , 29 , 42
SOA	Service Oriented Architecture. 6–10 , 12 , 18–21 , 34 , 57 , 65 , 67–70
SQL	Structured English Query Language. 16 , 69
UC	Use Case. 57 , 58 , 62
UI	User Interface. 16 , 64 , 69
UML	Unified Modeling Language. 11
URI	Uniform Resource Identifier. 29
URL	Uniform Resource Locator. 63
XML	Extensible Markup Language. xvii , 18 , 35 , 36 , 39 , 41 , 43 , 45 , 53 , 68
YAML	YAML Ain't Markup Language. 64

Příloha B

Přehledové obrázky a snímky

Business Context Administration

Business context: auth.userLoggedIn

Included contexts No included contexts

Preconditions

- User must be signed in:
\$user is not null

Postconditions No post-conditions

[Edit](#)

Obrázek B.1: Detail byznysového kontextu v centrální administraci

PŘÍLOHA B. PŘEHLEDOVÉ OBRÁZKY A SNÍMKY

Business Context Administration

Business context: shipping.correctAddress

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 - <businessContext prefix="shipping" name="correctAddress">
3   <includedContexts />
4   <preconditions>
5     <precondition name="Shipping address must contain a country, city, street and postal code">
6       <condition>
7         <logicalAnd>
8           <left>
9             <logicalAnd>
10            <left>
11              <isNotNull>
12                <argument>
13                  <objectPropertyReference propertyName="country" objectName="shippingAddress" type="string"/>
14                </argument>
15              </isNotNull>
16            </left>
17            <right>
18              <isNotNull>
19                <argument>
20                  <objectPropertyReference propertyName="city" objectName="shippingAddress" type="string"/>
21                </argument>
22              </isNotNull>
23            </right>
24          </logicalAnd>
25        </left>
26        <right>
27          <logicalAnd>
28            <left>
29              <isNotNull>
30                <argument>
31                  <objectPropertyReference propertyName="street" objectName="shippingAddress" type="string"/>
32                </argument>
33              </isNotNull>
34            </left>
35            <right>
36              <isNotNull>
37                <argument>
```

Save changes

Obrázek B.2: Formulář pro vytvoření nebo úpravu byznysového kontextu v centrální administraci

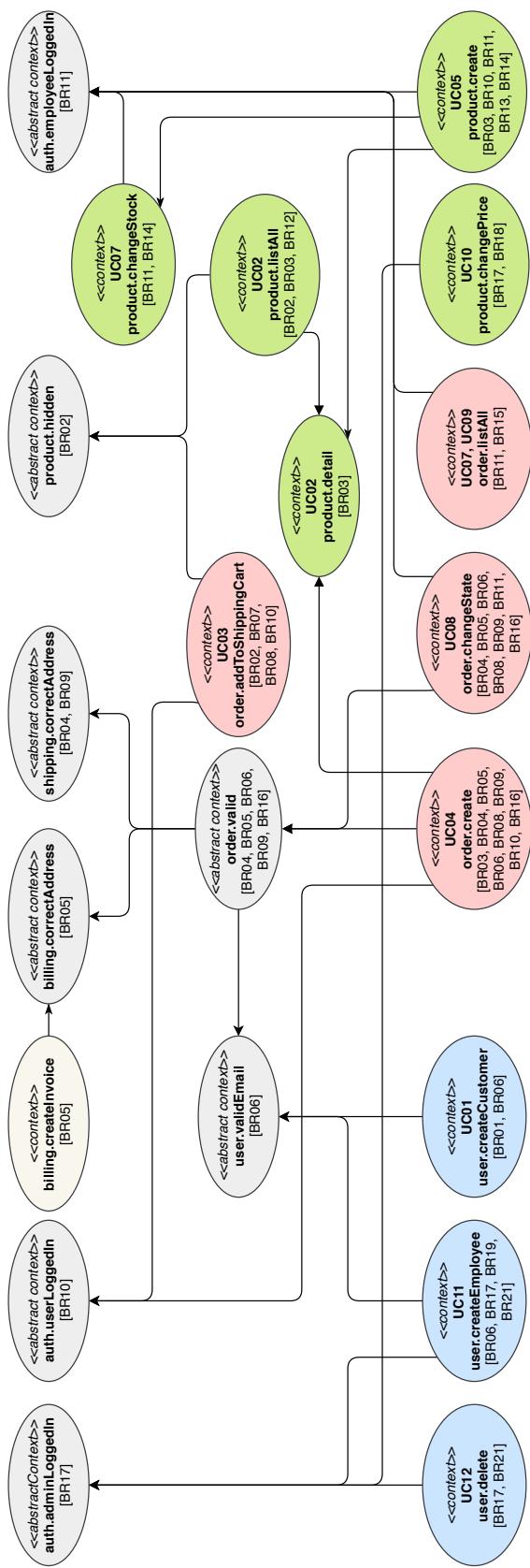
Example User: John Doe | Shopping cart has 10 items

Example e-commerce system
Buy everything you need!

Could not add product to cart: Shopping cart must contain less than 10 items

Rolex	Ferrari 458 Italia	Hamburger
50,000.00 € Beautiful watch for wealthy engineers Add to cart	150,000.00 € Beautiful car for wealthy engineers Add to cart	6.00 € A meal for the working class Add to cart
Wardrobe	A chair	
300.00 € Wooden wardrobe for your clothes Add to cart	399.00 € Prime Italian leather, it can bend over backwards Add to cart	

Obrázek B.3: Propagace byznysového pravidla při přidávání produktu do košíku v ukázkovém systému



Obrázek B.4: Diagram hierarchie byznysových kontextů ukázkového systému

Příloha C

Obsah přiloženého CD

-- central-administration/	Systém centrální administrace byz. kontextů
-- example/	Ukázkový e-commerce systém
-- billing/	Fakturační služba
-- order/	Objednávková služba
-- product/	Produktová služba
-- shipping/	Doručovací služba
-- ui/	Uživatelské rozhraní
-- user/	Uživatelská služba
-- java/	Prototyp knihovny pro jazyk Java
-- business-context/	Jádro knihovny
-- business-context-aspectj/	Adaptér pro framework AspectJ
-- business-context-grpc/	Knihovna pro síťovou komunikaci využívající gRPC
-- business-context-xml/	Knihovna pro načítání a zápis byz. kontextů do XML
-- nodejs/	Prototyp knihovny pro platformu Node.js
-- business-context/	Jádro knihovny
-- business-context-grpc/	Knihovna pro síťovou komunikaci využívající gRPC
-- business-context-xml/	Knihovna pro načítání a zápis byz. kontextů do XML
-- proto/	Schéma síťové komunikace ve formátu Proto Buffers
-- python/	Prototyp knihovny pro jazyk Python
-- business_context/	Jádro knihovny
-- business_context_grpc/	Knihovna pro síťovou komunikaci využívající gRPC
-- business_context_xml/	Knihovna pro načítání a zápis byz. kontextů do XML
-- xml/	Schéma XML pro zápis byz. kontextů
-- text/	Zdrojový kód bakalářské práce v jazyce LaTeX