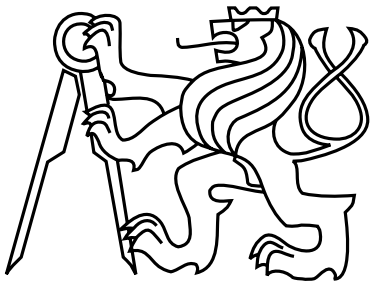
Na tomto místě bude oficiální zadání vaší práce

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

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

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

Bc. Filip Klimeš

Vedoucí práce: Ing. Karel Čemus

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

9. 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, kteří mě motivovali k dosahování 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 20. 5. 2018 .....

Abstract

Translation of Czech abstract into English.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její obsah. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

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 aspektů našich činností. Jejich využití sahá do mnoha sektorů, od vzdělání a vědy, kde mimo jiné významně usnadňují přístup ke studijním materiálům, přes zdravotnictví, kde pomáhají zvyšovat efektivitu a úroveň péče o pacienty [28], až po sociální sítě, kde umožňují lidem globálně komunikovat a sdílet své myšlenky, pocity a zážitky. Jedním z úkolů výzkumu v oblasti softwarového inženýrství je zjednodušení a zefektivnění procesu 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.

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 tento přístup uplatnit v praxi.

Tento koncept by měl díky využití aspektově orientovaného programování usnadnit práci vývojářů a doménových expertů. Díky tomu by mohl přinést snížení nákladů na vývoj a údržbu informačních systémů a tím zvýšit jejich kvalitu a snížit náklady na jejich vývoj.

Kapitola 2 se věnuje detailní analýze problematiky bynysových 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 prístupu k vývoji informacních systému a speciálně se zaměřuje 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 budou sloužit 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 pro platformy jazyků Java a Python a frameworku Node.js. Následující kapitola 6 popisuje, jakým způsobem byly tyto knihovny otestovány a jak byla prokázána jejich funkčnost. Zároveň je zde popsána validace a vyhodnocení konceptu frameworku jeho nasazením při vývoji jednoduchého ukázkového e-commerce systému. V poslední kapitole 7 je shrnuto, jakých cílů bylo v práci dosáhnuto a jakým dalším směrem se může výzkum v této oblasti ubírat.

Kapitola 2

Analýza

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 popisuje nedostatky současných přístupů při sdílení byznysových pravidel. V závěru kapitoly jsou identifikovány 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 [26]. K tomuto účelu uchovávají a spravují data a měly by zaručit, že nedojde k jejich poškození či narušení jejich integrity. Byznysové operace proto podléhají byznysovým pravidlům, která zajišťují konzistenci dat informačního systému a zabraňují nepovoleným operacím [17].

Definice. Byznysové pravidlo je výraz, který definuje či omezuje některý z byznysových aspektů. Jeho úkolem je ověřovat byznysovou strukturu nebo ovlivňovat byznysové chování [52].

EIS obsahují mnoho byznysových pravidel, typicky stovky či tisíce [52]. 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 dělena do tří skupin [15]:

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* [17].

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 [44]. 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 kladeny požadavky, které musejí být splněny po jejím úspěšném vykonání [17]. Příkladem může být anonymizace uživatelů při vytváření statistického reportu e-commerce společnosti – vygenerovaný report nesmí obsahovat citlivé údaje. 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

Informační systém zpravidla implementuje více byznysových procesů, které se vážou na jeden či více uživatelských scénářů [44]. Uživatelský scénář se pak dělí na jednotlivé kroky, například zaslání potvrzovacího e-mailu k objednávce, či uložení objednávky do databáze. Tyto kroky nazýváme *byznysové operace* – tedy operace, které mají byznysovou hodnotu.

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. *exekuční kontext* (z anglického *execution context*), který se skládá z několika dílčích kontextů [18]:

Aplikační kontext drží stav globálních proměnných systému, jako například 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 IP adresa uživatele či jeho geolokace.

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

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

Definice. Byznysový kontext je množina preconditions a post-conditions s byznysovou hodnotou, která se váže na konkrétní byzynsovou operaci [17]

2.1.4 Reprezentace byznysového pravidla

Existuje několik možností, jak v rámci EIS zachytit a reprezentovat byznysová pravidla [17]. Těmi jsou:

Ⓐ Zápis v obecném programovacím jazyce
Ⓑ Zápis pomocí meta-instrukcí
Ⓒ Zápis pomocí doménově specifických jazyků

Ⓐ je nejběžnější metodou, umožňující použití stejného jazyka pro popis byznysových pravidel jako pro popis ostatních částí systému. Bohužel, tato metoda nepřináší možnosti inspekce a extrakce pravidel pro jejich další využití. Ⓑ je pokročilejší metodou, která může využívat například anotací, nebo tzv. *Expression Language* (EL) [54]. Tato metoda poskytuje dobrou možnost inspekce, ale zpravidla není typově bezpečná. Navíc je potřeba meta-instrukce vázat na existující prvky systému, což může být pro některé případy použití omezující [17].

Ⓒ je nejpokročilejší metodou. DSL jsou snadno srozumitelné nejen pro programátory, ale i pro doménové experty. Navíc mohou být typově bezpečné. Mezi jejich nevýhody ale patří vysoká počáteční investice v podobě návrhu jazyka, nutnosti jeho kompilace nebo interpretace a také proškolení vývojářů, kteří s ním budou pracovat. Kvůli tomu může být vhodné využít existující řešení [17]. Tím může být například Object Constraint Language (OCL) [74], který je často využívaný ve výzkumu, nebo některé průmyslové řešení, jako je framework Drools[1], který je popsaný v sekci 3.8.1.

[1] https://www.drools.org/

2.2 Architektura orientovaná na služby

Architektura orientovaná na služby (SOA) je odpovědí na stále se zvyšující nároky na informační systémy a jejich rostoucí velikost. Na rozdíl od *monolitické architektury*, dělí SOA systém na samostatné nezávislé celky, zvané *služby*, které jsou poskytují dílčí části požadované funkcionality systému. Historicky byl termín SOA vykládán několika způsoby a představoval několik rozdílných, nekompatibilních konceptů [33]. Absence kvalitních definic služby a obecně SOA vedla k v poslední době i ke snahám o opuštění tohoto konceptu [20]. Pro lepší porozumění se tato kapitola věnuje stručnému historickému přehledu SOA a shrnuje výhody a nevýhody jednotlivých přístupů.

Definice. Služba je znovupoužitelný, soudržný, spravovatelný, nasaditelný a nezávislý proces komunikující pomocí zpráv [55][25].

2.2.1 Common Object Request Broker Architecture

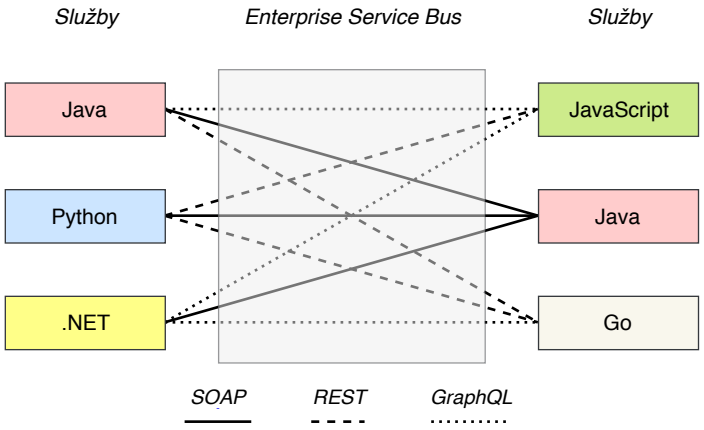
Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture* (CORBA) [62]. Ta umožňuje vzájemnou komunikaci aplikací implementovaných v různých technologiích. Její základní komponentou je je *Object Request Broker* (ORB), který emuluje objekty, 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. Komunikace se vzdáleným objektem s sebou však nese celou řadu problémů, například vyšší latenci při komunikaci nebo výjimečné stavy, které je potřeba ošetřit, či obížnou optimalizaci kódu využívající ORB.

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 explicitním posíláním zpráv mezi službami pomocí protokolu HTTP. Pro popis schématu zpráv vznikl formát *Simple Object Access Protocol* (SOAP) [10], který v kombinaci s *Web Service Description Language* (WSDL) [23] 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). Základní myšlenkou MQ, znázorněnou na obrázku **??**, je asynchronní komunikace služeb pomocí zpráv nezávislých na platformě. Komunikaci zprostředkovává fronta, která přijímá a rozesílá zprávy



Obrázek 2.1: Komunikace služeb skrz Enterprise Service Bus

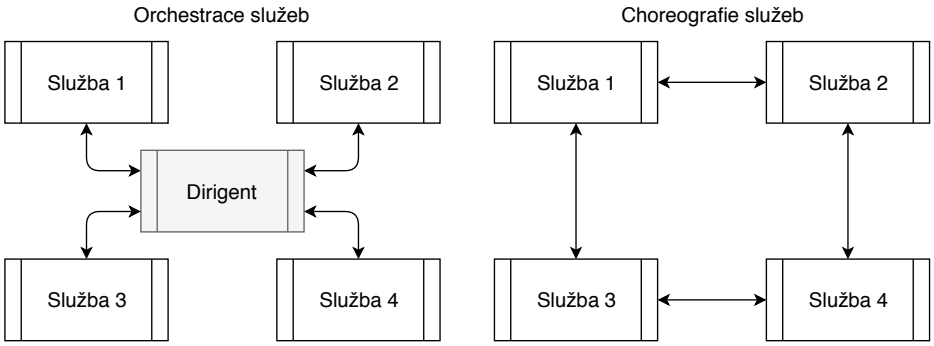
mezi službami. To přináší vyšší škálovatelnost a menší provázanost mezi službami. Všechny služby ale musí používat jednotný formát zpráv.

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) [22], znázorněný na obrázku 2.1, který má za úkol propojit heterogenní služby a sestavit mezi nimi komunikační kanály. Tím na sebe ESB přebírá zodpovědnost za překlad jednotlivých zpráv a centralizuje veškerou komunikaci v systému.

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, ačkoliv existují i názory, že jde o odlišné architektury [59][20]. Vzhledem k její vzrůstající adopci v posledních letech je nutno ji v rámci této práce zohlednit [77]. 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 [46]. 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 flexibilní nasazování a škálování, které je vhodné pro stále populárnější nasazení v cloudu [43][21][77].



Obrázek 2.2: Porovnání orchestrace a choreografie služeb [20]

2.2.6 Orchestrace a choreografie služeb

Základní podmínkou pro funkci systému stavějícímu na SOA je komunikace a spolupráce jednotlivých služeb. K tomu slouží principy *orchestrace služeb* a *choreografie služeb*.

Orchestrace Orchestrace služeb má za úkol zajistit, že komunikace mezi službami proběhne úspěšně a ve správném časovém sledu [69], za použití centrální komponenty – tzv. *dirigenta*. Typicky je jako dirigent využíván ESB.

Choreografie Přímým opakem orchestrace je tzv. *choreografie služeb* a znamená vykonávání byznysových operací autonomně a asynchronně, bez centrální autority. Tento přístup je preferován zejména v rámci microservices [25], protože orchestrace vede k vyššímu provázání služeb a nerovnoměrnému rozložení zodpovědností v systémů. Porovnání obou přístupů je graficky znázorněno na obrázku 2.2.

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 systému, zatímco moderní přístupy od těchto vlastností upouštění. 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 rozhraní [55][46].

2.3 Nedostatky současného přístupu

Některá složitější byznysová funkcionalita vyžaduje kompozici více služeb najednou [55]. Kompozitní služby by měly zohlednit byznysová pravidla služeb, které ke své funkci využívají, aby zabránily nekonzistentním stavům v systému a zbytečným spouštěním byznysových operací, jejichž preconditions nejsou splněny [20]. To je však s přímým rozporem 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 [19].

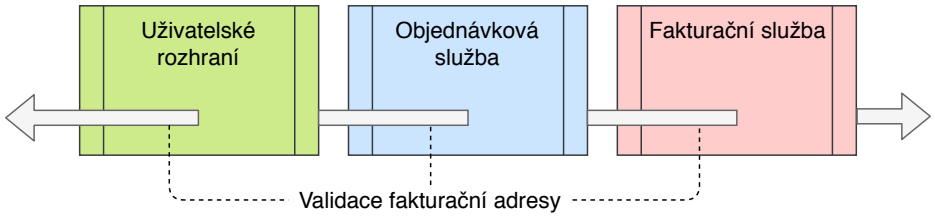
Definice. Kompozitní služba získává a kombinuje informace a funkce existujících poskytovatelné služeb [55].

Pro lepší představu tohoto problému uvažme e-commerce systém skládající se z několika služeb naprogramovaných v různých technologiích, a procesy vytváření faktury a vytváření objednávky, každý z nich implementovaný jinou službou. Systém navíc obsahuje službu poskytující webové uživatelské rozhraní. Při vytváření faktury za objednávku musí být nejprve zvalidována fakturační adresa. Protože by mohla nastat situace, kdy by v případě nevalidní adresy museli zaměstnanci společnosti kontaktovat zákazníka – pokud vůbec takovou možnost mají – musí být 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 [18]. Pro ilustraci je problém znázorněn na obrázku 2.3,

Na příkladu lze pozorovat, že stejná funkcionalita se promítá do tří služeb, z nichž každá má zodpovědnost za jiné byznysové operace. Stejný kód, který realizuje validaci fakturační adresy, musí být implementován v každé ze zmiňovaných 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 znovu sestavit a nasadit ve správném pořadí tak, aby nedošlo k nekonzistení validaci adresy při provádění jednotlivých byznysových operací. Změny byznysových pravidel se dějí častěji, než změny kódu a struktury samotných služeb v SOA [60]. Pokud je potřeba s každou změnou byznysového pravidla sestavit a nasadit jednu či více služeb, dramaticky se zvyšuje náročnost na údržbu takového systému.

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



Obrázek 2.3: Příklad zásahu jedné funkcionality do více služeb

expertům. Úprava pravidel by navíc neměla vyžadovat změnu kódu služby a její opětovné nasazování. Administrátoři systému by měly mít možnost byznysová pravidla spravovat centrálně a bez přerušení provozu systému, aby mohli co nejrychleji a flexibilně reagovat na změnové požadavky.

Framework, který bude výstupem této práce, by tedy měl 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 byzynsových pravidel včetně vyhodnocování preconditions a aplikace post-conditions
- Možnost využívat framework na více plaformách

2.5 Shrnutí

V této kapitole byla provedena analýza byznysových pravidel a byznysových kontextů a architektury orientované na služby. Dále 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

Tato kapitola se věnuje rešerši existujících řešení a výzkumu relevantnímu k tématu této práce. Díky tomu bude umožněno dosáhnout kvalitního a efektivního návrhu frameworku pro centrální správu a automatickou distribuci byznysových pravidel.

3.1 Modelem řízená architektura

Modelem řízená architektura (MDA z anglického *Model-Driven Architecture*) se zaměřuje na návrh EIS s využitím modelů a jejich následnou transformaci do spustitelného kódu pomocí generativních nástrojů [64]. Hlavní výhodou MDA je vysoká úroveň abstrakce, která zbavuje vývojáře nutnosti manuálně duplikovat informace. Další výhodou je nezávislost na platformě a zvýšení kvality kódu díky jeho automatickému generování.

MDA v první fázi vývoje využívá 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, typicky k popisu 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 na jeden či více PSM. Nakonec je PSM transformován do spustitelného kódu [42].

Hlavní nevýhodou MDA, která zabraňuje jejímu využití pro účel této práce, je jednosměrný dopředný proces, kterým je výsledný kód generován. Pokud dojde ke změně požadavků, která se promítne do modelu, je potřeba přegenerovat kód celého systému. Kód, který bylo nutno doplnit ručně, může snadno zastarat a je tak potřeba ho manuálně projít a opravit. Další nevýhodou tohoto přístupu je jeho závislost na OOP, které samotné není schopné se efektivně vypořádat s průřezovými problémy [40][15], jak bude demonstrováno v sekci 3.6.

3.2 Generativní programování

Generativní programování (GP) je dalším příkladem paradigmatu, který využívá vyšší úroveň abstrakce a díky tomu zvyšuje znovupoužitelnost kódu. 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 [24]. Díky tomu je možné oddělit popis jednotlivých vlastností systému a dosáhnout tak jejich vysoké znovupoužitelnosti.

GP by mohlo být využito pro abstrakci bynysových pravidel a jejich automatickému začleňování do kódu služeb v systému stavějícím na SOA. Statické generování komponent však nesplňuje požadavek na dynamickou správu byznysových pravidel za běhu systému.

3.3 Metaprogramování

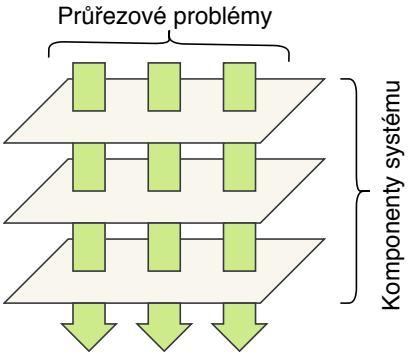
Metaprogramování je alternativním paradigmatem, které vnímá kód programu zároveň jako data, se kterými program může pracovat. To programu umožňuje číst, vytvářet či upravovat jiné programy včetně sama sebe. Tyto činnosti lze provádět staticky, ale i za běhu daného programu [61][24]. Program, který manipuluje s kódem, se nazývá *metaprogram*. Programovací jazyk, který umožňuje programování, se nazývá *metajazyk* (z anglického *metalanguage*) [72]. Schopnost programovacího jazyka být sám sobě metajazykem se nazývá *reflexe* [63].

Tento přístup přináší vysokou úroveň abstrakce a zvýšenou efektivitu vývojářů, kteří jsou schopni automaticky provádět inspekci, generovat a upravovat programy. Reflexe je součástí moderních programovacích jazyků a je využívána mnoha frameworky a knihovnami [70] [31]. Jedním z vhodných využití metaprogramování je implementace DSL [61].

3.4 Business Process Execution Language

Technologie Business Process Execution Language (BPEL) využívá speciálního DSL postaveného na jazyku XML k popisu byznysových procesů realizovaných webovými službami [2]. Umožňuje *top-down* realizaci SOA skrz kompozici, orchestraci a koordinaci služeb [1]. Přístup BPEL využívá meta-služby, které se starají o uložení a transformaci byznysových pravidel a také o zachycení byznysových operací a aplikaci těchto pravidel [60].

BPEL přináší možnost využít byznysová pravidla spravovaná doménovými experty v procesně orientovaném prostředí SOA. Díky tomu výrazně zvyšuje kvalitu a snižuje náročnost vývoje. Nejnovější výzkum v oblasti SOA a zejména Microservices však od orchestrace ustupuje na úkor decentralizace a choreografie služeb [3][21].



Obrázek 3.1: Průřezové problémy v informačních systémech

3.5 Objektově orientované programování

Jedním z nejpoužívanějších paradigmat používaných k vývoji moderních EIS je objektově orientované programování (OOP). To používá koncept tzv. objektů, které zapouzdřují data a funkcionalitu do malých funkčních celků odpovídající struktuře reálného světa [58]. Objekty se rozumí jak konkrétní koncepty, například auto nebo člověk, tak i abstraktní koncepty, jako je bankovní transakce nebo objednávka v obchodě. Podoba objektů se pak promítá do kódu programu i do reprezentace struktur v paměti počítače. Tento přístup je velmi snadný pro pochopení, vede k lepšímu návrhu a organizaci programu a snižuje tak náklady na jeho vývoj a údržbu.

Vlastnosti OOP, jako je zapouzdření, dědičnost a polymorfismus, přináší vysokou znovupoužitelnost kódu, nižší riziko lidské chyby, zjednodušení návrhu systému a nižší náklady na vývoj a údržbu software.

3.6 Aspektově orientované programování

3.6.1 Motivace

Ačkoliv je OOP velmi silným nástrojem, existují problémy, které nelze v jeho rámci efektivně řešit. Příkladem takového problému jsou obecné požadavky na systém, které musejí být konzistentně dodržovány na více místech systému, které spolu zdánlivě nesouvisí, tzv. *průřezové problémy* (z anglického *cross-cutting concerns*). V rámci OOP je programátor nucen v ojektech manuálně opakovat kód, který zodpovídá za jejich realizaci. Duplikace kódu vede k větší náchylnosti na lidskou chybu a k vyšším nárokům na vývoj a údržbu daného softwarového systému [35]. Obrázek 3.1 znázorňuje vzájemné postavení průřezových problémů a komponent informačního systému.

Zdrojový kód 3.1: Příklad průřezových problémů zohledněných při vytváření objednávky

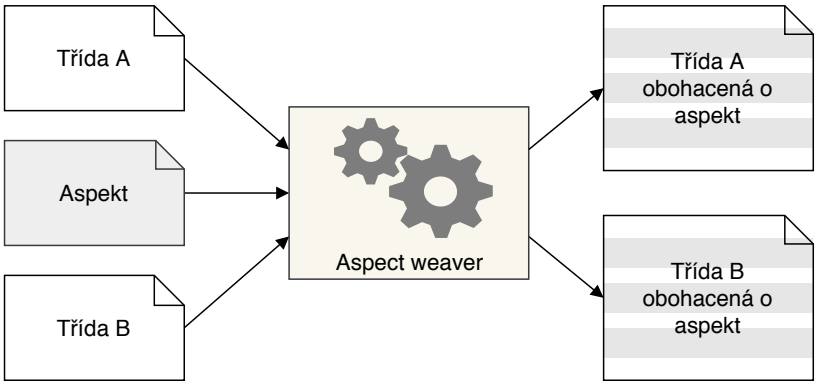
```
void createOrder(User user, Collection<Product> products, Address shipping,
                 Address billing) {
    logger.info("Creating order"); // Logging aspect
    transaction.begin(); // Transaction aspect
    try {
        validator.validateAddress(shipping); // Shipping business rules aspect
        validator.validateAddress(billing); // Billing business rules aspect
        Order order = new Order(user, product, shipping, billing);
        database.save(order);
        transaction.commit(); // Transaction aspect
        logger.info("Order created successfully"); // Logging aspect
    } catch (Exception e) {
        transaction.rollback(); // Transaction aspect
        logger.error("Could not create order"); // Logging aspect
    }
}
```

Definice. Průřezový problém je vlastnost systému, která ovlivňuje více jeho komponent zásahem do jejich funkcionality [41].

Příkladem průřezového problému může být logování systémových akcí, optimalizace správy paměti nebo jednotné zpracování výjimek [41], ale i aplikace byznysových pravidel [15]. Ve zdrojovém kódu 3.1 je znázorněno, jak průřezové problémy zasahují do kódu imaginární třídy implementované v jazyce Java, která slouží pro vytváření objednávek v e-commerce systému popsaném v sekci 2.3. Aspekt logování je zohledněn na třech místech, stejně jako aspekt transakcí. Navíc jsou zde zohledněna i byznysová pravidla pro validaci doručovací a fakturační adresy objednávky.

3.6.2 Vlastnosti

Aspektově orientované programování (AOP) přináší řešení výše zmiňovaných problémů. Využívá k tomu *separation of concerns* – extrahuje kód zachycující průřezové problémy, tzv. *aspekty*, do jednoho bodu, tzv. (*single focal point*). Pomocí procesu zvaného *weaving* je poté tento kód automaticky distribuován. Weaving může proběhnout staticky při kompilaci programu nebo dynamicky při jeho běhu. V obou případech ale programátorovi ulehčuje práci, protože k definici i změně aspektu dochází centrálně, a tím je eliminována potřeba manuální duplikace kódu. AOP není paradigmatem poskytujícím kompletní framework pro



Obrázek 3.2: Proces weavingu aspektů

návrh programu. V ideálním případě je tedy k návrhu systému využita kombinace AOP s jiným paradigmatem.

3.6.3 Názvosloví

Základním pojmem v rámci AOP je *aspekt*, který zapozdřuje průřezovou funkcionalitu a zároveň adresuje místa, kde má být funkcionalita aplikována. Aspekt vždy obsahuje alespoň jeden *advice* a jeden *pointcut*.

Místo v kódu, na které může být aplikována funkcionalita 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.

Množina join-pointů, na které je jeden konkrétní aspekt aplikován, se nazývá *pointcut*. Tato množina může být určena staticky, a být tak známá při kompilaci programu, nebo dynamicky za běhu programu, což přináší výpočetní složitost navíc výměnou za vyšší flexibilitu.

Funkcionalita, kterou aspekt přidává v jeho pointcutu, se nazývá *advice*. Existuje více typů advice, podle toho, kam je daná funkcionalita přidána. Například při volání metody může být funkcionalita přidána před, za, nebo kolem metody.

Proces, kterým jsou advice začleňovány podle pointcutu do jednotlivých join-pointů se nazývá *weaving*. Ten může probíhat již při kompilaci 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*.

3.7 Aspect-driven Design Approach

3.7.1 Vlastnosti

Alternativním způsobem návrhu informačních systémů, který staví na principech AOP, je Aspect-driven Design Approach[1] (ADDA) [15], představený v roce 2014. Tento přístup se zaměřuje na formalizaci jednotlivých komponent informačních systémů identifikování aspektů v informačních systémech a jejich separaci do *single focal point*. Následně přístup využívá weaving pro automatickou distribuci aspektů do systému. K popisu aspektu doporučuje využití doménově specifického jazyka, který bude navržen na míru danému průřezovému problému.

3.7.2 Možnosti aplikace

Autoři ADDA aplikovali tento koncept v několika oblastech EIS. Mezi tyto oblasti patří automatické začleňování byznysových pravidel do datové vrstvy informačních systémů [17], automatické generování uživatelských rozhraní citlivých na kontext uživatele [18], validaci vstupů formulářů v uživatelském rozhraní vůči byznysovým pravidlům [14][18] a automatické extrakci dokumentace [16].

Jednou z možných aplikací přístupu ADDA je automatické začleňování byznysových pravidel do datové vrstvy EIS[2]. Byznysová pravidla jsou nejprve vhodně 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.

Uživatelská rozhraní tvoří až 48 % kódu informačních systému a zabírají až 50 % jejich vývojového času [40]. Do UI se přitom typicky promítá mnoho aspektů, které jsou již v systému obsaženy. Například byznysová pravidla jsou promítána do UI při validaci vstupních dat formulářů na straně klienta [18]. 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 [14] potřebného pro popis adaptibilního uživatelského rozhraní.

Další oblastí informačních systému, do které se promítají jeho aspekty, je dokumentace [16]. Autoři ADDA využívají data mining pro získání metainformací o byznysových

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

[2] Předpokládáme standardní třívrstvou architekturu informačních systémů [32]

operacích, datovém modelu systému a o byznysových pravidlech. Díky tomu mohou automaticky vygenerovat seznam byznysových operací, potažmo implementovaných use-cases, strukturu doménového modelu a formální popis byznysových pravidel, který může sloužit pro verifikaci jejich správnosti.

3.7.3 Výhody a nevýhody

ADDA poskytuje vývojářům způsob jakým výrazně snížit náklady na vývoj a údržbu systému díky deduplikaci, která je dosažena 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 umožňující požadovanou funkcionalitu, pro nasazení do reálného systému nejsou tyto knihovny připraveny.

Přístup ADDA splňuje požadavky identifikované v sekci 2.4, zejména využití speciálních DSL pro popis aspektů a jejich automatickou distribuci za běhu systému. Pro popis byznysových pravidel využívá ADDA nástroj *Drools*, který je popsán v následující sekci.

3.8 Stávající řešení reprezentace business pravidel

Tato kapitola se zaměřuje i na současné možnosti zachycení byznysových pravidel ve specializovaných jazycích a vhodnost jejich použití pro účel frameworku, který bude výstupem této práce. Ačkoliv existuje relativně velké množství knihoven poskytujících DSL pro popis byznysových pravidel a umožňující automatickou distribuci byznys pravidel, žádný z nich nepodporuje dostatečně velké množství platforem, ve kterých by mohl být jazyk použit. Příkladem může být projekt *business-ruless* pro jazyk Python[3], projekt FlexRule[4] pro platformy .NET a JavaScript nebo BRMS JRules [12] od společnosti IBM pro platformu Java EE. Tato sekce se proto zaměřuje zejména na framework Drools, který používají autoři přístupu ADDA, a také na moderní nástroj JetBrains MPS, který umožňuje vytvářet vlastní DSL a transformovat ho do dalších jazyků.

Definice. DSL je programovací jazyk určený k popisu specifické vlastnosti či funkce systému v rámci dané domény [34].

3.8.1 Drools DSL

Framework Drools[5] je open-source projekt realizující *business rule management engine* (BRMS), tedy nástroj pro vývoj a správu byznysových pravidel. Framework umožňuje vývoj

[3] https://pypi.org/project/business-rules/
[4] http://www.flexrule.com/archives/business-rule-language/
[5] https://www.drools.org/

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. Pro určení produkčních pravidel, která mají být aplikována, je využit algoritmus RETE [30].

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 rozšiřování jiných kontextů a popis typu jednotlivých pravidel [16]. Ze strany frameworku Drools navíc nejsou podporovány jiné platformy než Java a .NET, což nevyhovuje požadavkům na platformovou nezávislost.

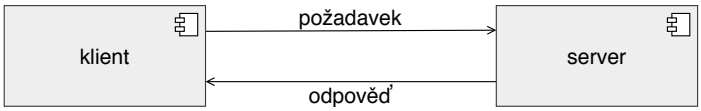
3.8.2 JetBrains MPS

Moderním nástrojem pro tvorbu doménově specifických jazyků je *JetBrains MPS* (Meta Programming System)[6]. Staví na konceptu *language-oriented programming* (LOP) [73] 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 hlavně na strukturu textu, aniž by se musel příliš detailně zaobírat samotnou sazbou.

MPS umožňuje 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.

Výhoda 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 [47][66]. Významnou výhodou MPS, potažmo LOP, je nezávislost na cílové platformě. Nástroj MPS by umožnil snadné znovupoužití pravidel a jejich transformaci do neomezeného počtu jazyků pro využití na

[6] https://www.jetbrains.com/mps/



Obrázek 3.3: Architektura klient-server

mnoha platformách. Podobně jako u MDA je však problém v dopředném generování – editor MPS totiž neumožňuje načíst víceúčelový jazyk zpět do DSL.

3.9 Síťové architektury

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

3.9.1 Architektura klient-server

Model klient-server popisuje vztah mezi komponentami systému, klienty a serverem. Klient zašle požadavek serveru a ten mu vrátí odpověď [7]. Schéma komunikace je znázorněno na obrázku 3.3. 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 kterým se stal široce využívaným. Díky svojí velmi obecné myšlence je nezávislý na jakékoliv platformě. Zároveň tato architektura přesouvá byznysovou logiku a ukládání dat na server a díky tomu umožňuje snadnější kontrolu nad systémem a jeho centrální administraci. S tím je spojena i snažší škálovatelnost systému. V neposlední řadě přináší model klient-server díky centralizaci i lepší zabezpečení, kdy server může jasně definovat a vynucovat přístupová pravidla.

Hlavní nevýhodu této architektury je vytvoření jednoho centrálního bodu, jehož výpadek ochromí funkci celého systému (v angličtině *single point of failure*) – tímto bodem je server. Pokud na serveru nastane chyba či výpadek, žádný z klientů není schopen využívat jeho služeb.

3.9.2 Architektura Peer-to-peer

Opakem modelu klient-server je síťová architektura zvaná *Peer-to-peer (P2P)*. Jednotlivé počítače v síti spolu komunikují přímo, bez centrální autority. Všechny počítače v síti jsou si vzájemně rovnocenné [37]. Hlavním cílem P2P sítí je distribuce dat nebo výpočetních operací.

Vysoká datová propustnost a robustnost P2P by mohla být vhodná pro sdílení byznysových pravidel. Absence centrální správy by však mohla způsobit nekonzistenci při úpravě či

přidání byznysového pravidla. Změna pravidla by se musela šířit postupně napříč systémem, přičemž některé uzly by stále využívaly starou verzi pravidla.

3.9.3 Representational state transfer

Representational state transfer (REST) je architektura webových služeb, která staví na protokolu HTTP, a klade na systém několik architektonických omezení, díky kterým může systém dosáhnout lepšího výkonu, vyšší škálovatelnosti, jednoduchému používání a lepší odolnosti vůči chybám [29]. Principy architektury REST zahrnují využití architektury klient-server, bezestavovost a kešování požadavků, vrsvení 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 [56], umožňující zajistit atomické transakce nad REST architekturou.

3.9.4 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 [53]. 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.

3.10 Shrnutí

V této kapitole byla provedena rešerše architektur, paradigmat a frameworků, které by mohly být vhodné pro řešení sdílení byznysových pravidel v SOA, a shrnula jejich výhody a nevýhody. Nejvíce se kapitola věnovala inovativnímu přístupu k návrhu softwarových systémů *ADDA*, ze kterého bude vycházet návrh frameworku, který bude výstupem této práce. Kapitola také shrnula rešerši stávajících řešení reprezentace byznys pravidel a zhodnotila jejich vhodnost pro použití v této práci. Nakonec kapitola studovala existující síťové architektury, které by mohly být využity pro distribuci byznysových pravidel v rámci SOA.

Kapitola 4

Návrh

V této kapitole je diskutován návrh frameworku pro centrální správu a automatickou integraci business pravidel vyhovující požadavkům identifikovaným v sekci 2.4. Tento návrh staví na znalostech získaných v předchozí kapitole 3, zejména na paradigmatu AOP a přístupu ADDA.

4.1 Formalizace architektury orientované na služby

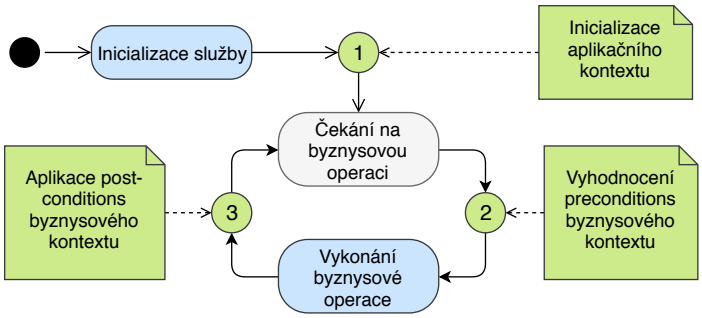
Pro formalizaci problému byznysových pravidel v SOA do termínů AOP je nutno identifikovat *join-points*, určit podobu *advices*, popsat způsob jakým budou zachyceny *pointcuts* a nakonec navrhnout proces *weavingu* pravidel.

4.1.1 Join-points

Identifikace join-points vychází ze životního cyklu služby, který je znázorněn na obrázku 4.1. První fází v životě instance služby je její inicializace, konkrétně načtení aplikačního 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



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

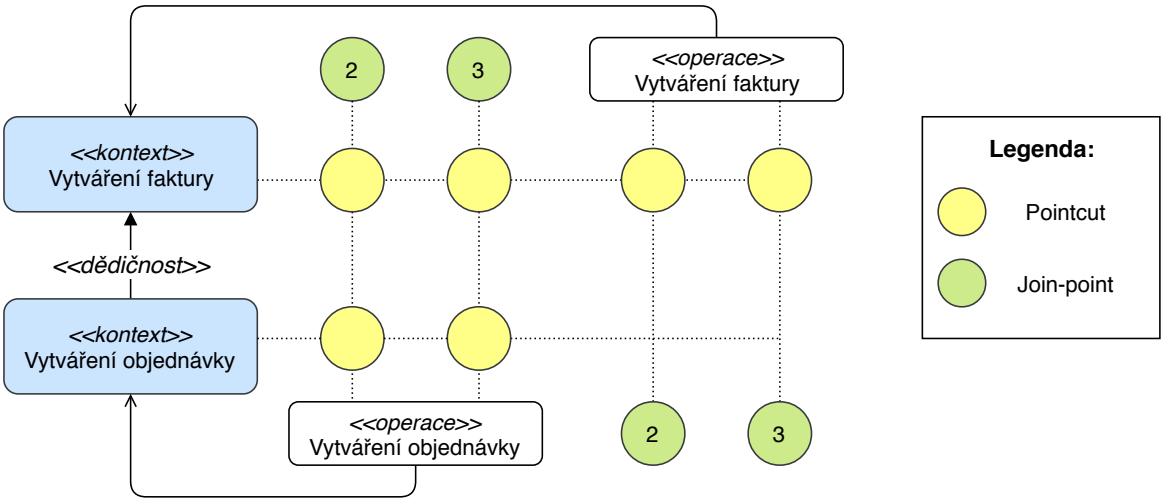
4.1.2 Pointcuts

V join-pointu ① by služba měla načíst všechna byznysová pravidla, která bude potřebovat ke své činnosti, a nejsou pro ni lokálně dostupná. Služba tedy musí zjistit, která pravidla je potřeba získat, a následně si je vyžádat od ostatních služeb. V join-pointech ② a ③ musejí být aplikována byznysová pravidla každého kontextu vztahujícího se k dané operaci.

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      @NotBlank(message = "city is compulsory")
7      private String city;
8
9      @NotBlank(message = "street is compulsory")
10     private String street;
11
12     @NotBlank(message = "postalCode is compulsory")
13     private String postalCode;
14
15     /* ... */
16
17 }
```

Pro zápis selektoru poincutu byznysového pravidla se lze inspirovat standardem JSR 303 [5], 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 `@NotNull` zajištěno, že fakturační adresa



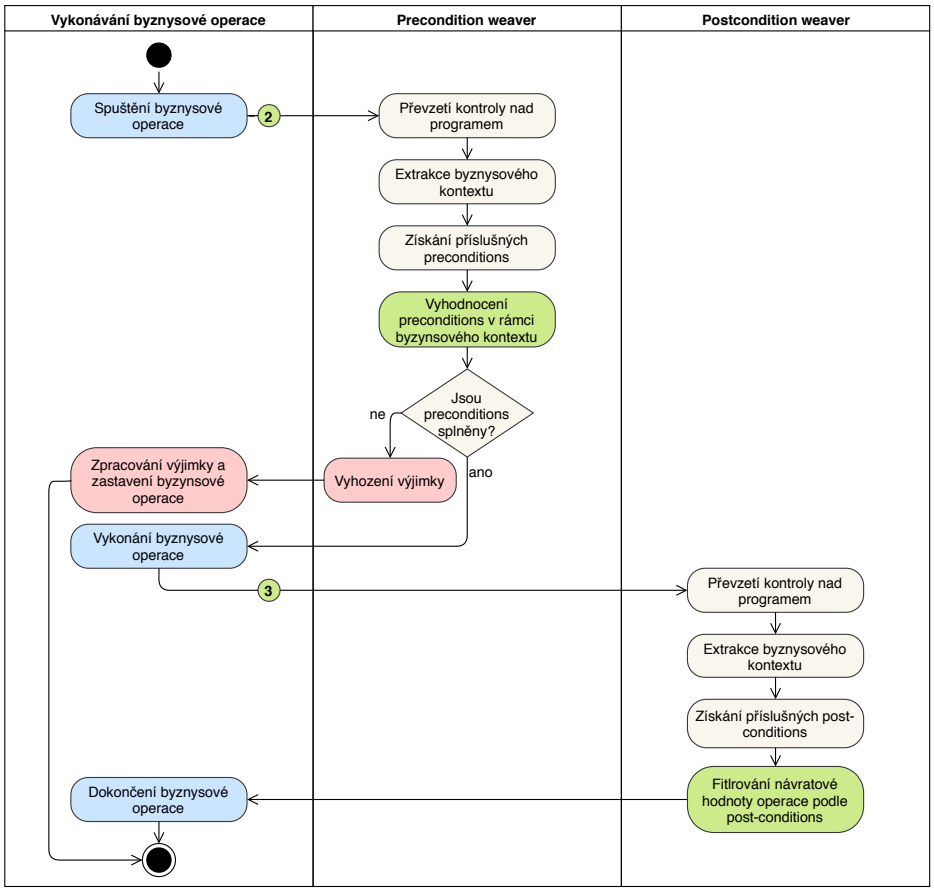
Obrázek 4.2: Diagram znázorňující dědičnost kontextů ve vztahu k join-pointům a pointcuts

bude mít vyplněna všechna pole (v kontextu našeho frameworku se jedná o paralelu preconditions). Podobným způsobem by každá byznysová operace mohla pomocí metainstrukcí specifikovat, která byznysová pravidla bude využívat. Toto řešení však neposkytuje možnost dynamicky při běhu programu změnit sadu byzynsových pravidel. Tento problém lze řešit zavedením konceptu byznysového kontextu, který zapouzdřuje byznysová pravidla, a byznysová operace se na něj může explicitně odkázat. Obsah byznysového kontextu by přitom mohl být dynamicky změněn za běhu programu.

Sdílení pravidel mezi byznysovými kontexty, potažmo byznysovými operacemi a mezi jednotlivými službami, by lze realizovat pomocí dědičnosti kontextů. Každý kontext, který by potřeboval validovat fakturační adresu, by tak mohl pouze dědit od kontextu vytváření faktury. Na obrázku 4.2 je dědičnost kontextů znázorněna. Kontext vytváření objednávky dědí od kontextu vytváření faktury a znovupoužívá jeho byznysová pravidla. Byznysové operace se odkazují na byznysové kontexty, které mají být při jejich vykonávání použity. Před spuštěním a po dokončení operace vytváření objednávky jsou aplikována pravidla obou kontextů, zatímco při vytváření faktury jsou zohledněna pouze pravidla jednoho kontextu.

4.1.3 Advices

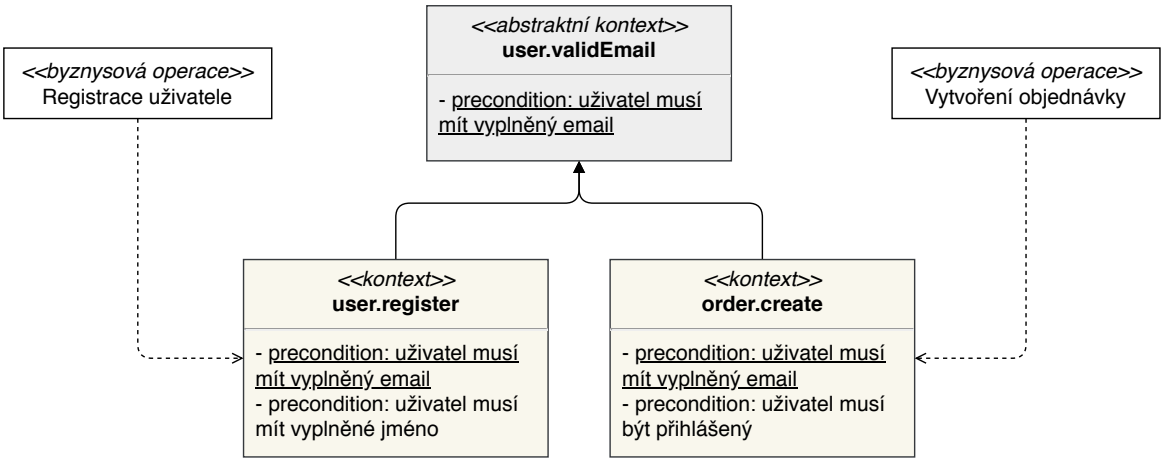
V případě join-pointu ① je advice samotná reprezentace byznysového kontextu přenášeného mezi službami. Naopak v join-pointech ② a ③ je přidanou funkcionalitou vyhodnocování preconditions nad aplikačním kontextem, resp. aplikování post-conditions na návratovou hodnotu operace.



Obrázek 4.3: Diagram aktivit weaverů byznysových pravidel

4.1.4 Weaving

Weaving v případě join-pointu ① bude provádět komponenta frameworku, která analyzuje lokálně dostupná pravidla služby, vyhodnotí, 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, shromaždí 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. Proces weavingu je zachycen na obrázku 4.3.



Obrázek 4.4: Diagram konceptu abstraktního byznysového kontextu

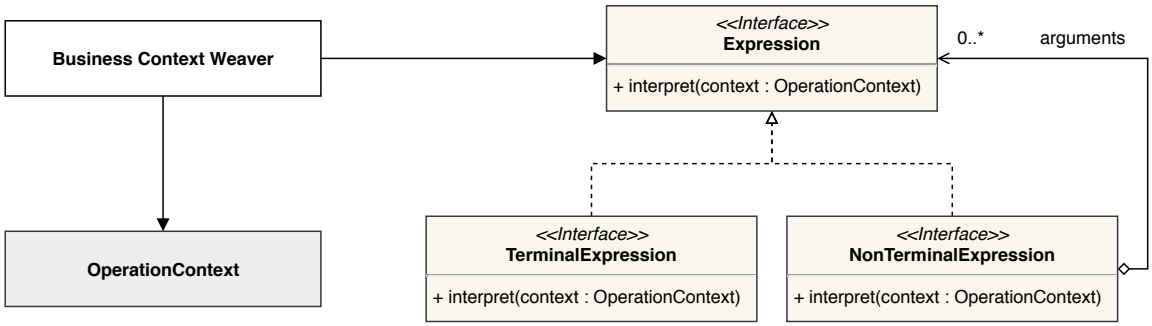
4.2 Dědičnost byznysových kontextů

V předchozím textu byl představen koncept dědičnosti byznysových kontextů. Každý kontext díky němu může rozšiřovat libovolné množství jiných kontextů, a sdílet jejich byznysová pravidla. Byznysové operace pak mohou samy určit, který byznysový kontext se k ním váže. Tento kontext 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 4.4 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. Tuto situaci nelze z hlediska frameworku vyřešit, ale dá se jí předejít. K prevenci by mohl sloužit validátor vestavěný do nástroje pro správu byznysových kontextů.

Vícenásobná dědičnosti může přinést problém, kdy jeden kontext zdědí více stejných pravidel z různých zdrojů, tzv. *diamond problem* [11]. 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 administraci byznysových pravidel.



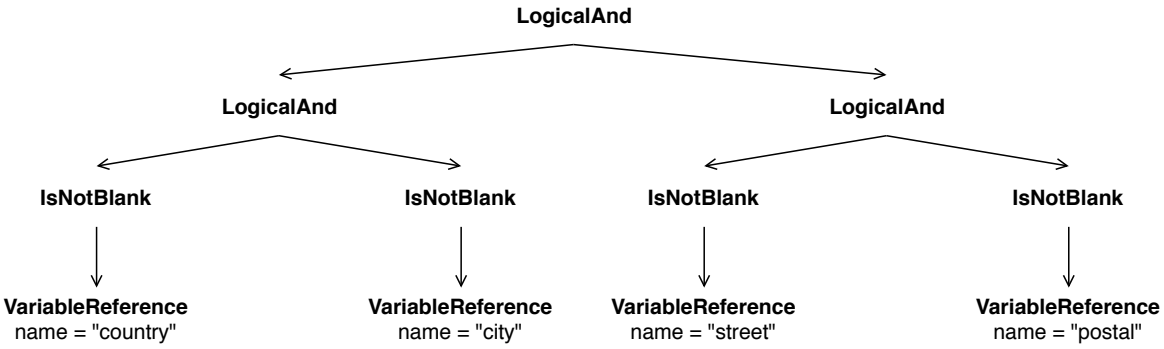
Obrázek 4.5: Diagram tříd popisující použití vzoru Intepreter pro vyhodnocování logických výrazů

4.3 Logické výrazy 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 hodnotry podléhat splnění určité podmínky, která musí být vyhodnocena v ③.

Podmínky byznysových pravidel se skládají z jednotlivých výrazů, které tvoří orientovaný acyklický graf (DAG), tzv. *derivační strom*. Výrazy se dělí na *terminály* a *neterminály* [50]. Terminál znamená, že z daného výrazu již nevychází žádná hrana do jiného výrazu. Neterminál je opak terminálu. Pro reprezentaci stromu bude využit návrhový vzor *Composite* [32]. K vyhodnocování podmínek popsaných v byznysovém pravidle je vhodný návrhový vzor *Interpreter* [32], jehož použití je demonstrováno na obrázku 4.5.

Framework bude disponovat základní sadou výrazů pro zápis byznysových pravidel. Mezi ně budou patřit logické operace `and`, `or`, `equals` a `negate`. Dále 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 bude zajišťovat výraz `ObjectPropertyReference`. K ověření přítomnosti hodnoty v proměnné bude sloužit výraz `IsNotNull`. Výraz `IsNotBlank` ověří, zda je v proměnné řetězec nenulové délky. Pro vložení konstantní hodnoty přímo do byznysového pravidla bude sloužit terminál `Constant`. Pro zvýšený komfort budou přidány i výrazi realizující základní matematické operace sčítání, odečítání, násobení a dělení. Pro volání uživatelských funkcí definovaných v operačním kontextu bude sloužit 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 bude framework podporovat, je v tabulce B.1,



Obrázek 4.6: Syntaktický strom jednoduchého validačního pravidla

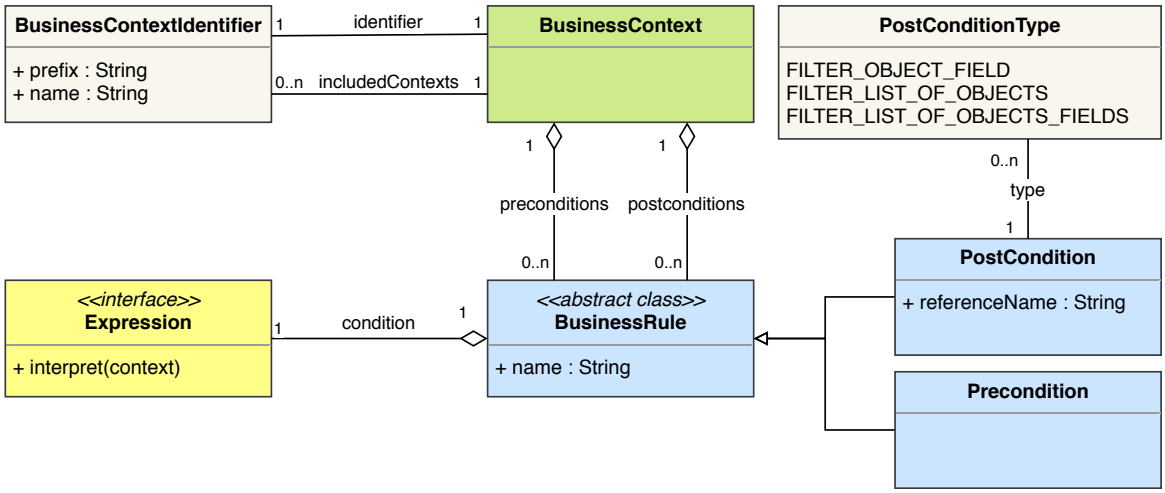
[1] Pro snažší implementaci na více platformách a prevenci sémantických chyb v pravidlech budou výrazy obsahovat i explicitní definici svého návratového typu. Výraz byznysového pravidla může nabývat logických hodnot, může vracet číslo, textový řetězec a také objekt. Je potřeba počítat také s tím, že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

- `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.

[2] Kromě argumentů neterminálů je v některých případech potřeba k výrazu uložit i dodatečné informace – *atributy*. Jedním z atributů je typ návratové hotnoty 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.

[3] Na obrázku 4.6 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

[1] [Intended Delivery: Typované výrazy]
[2] [Intended Delivery: Atributy pravidel]
[3] [Intended Delivery: Příklad AST pravidla]



Obrázek 4.7: Diagram tříd metamodelu byznysového kontextu

validovány výrazem `IsNotBlank` a jednotlivé validace jsou spojeny pomocí binárních výrazu `LogicalAnd` odpovídajících logické konjunkci.

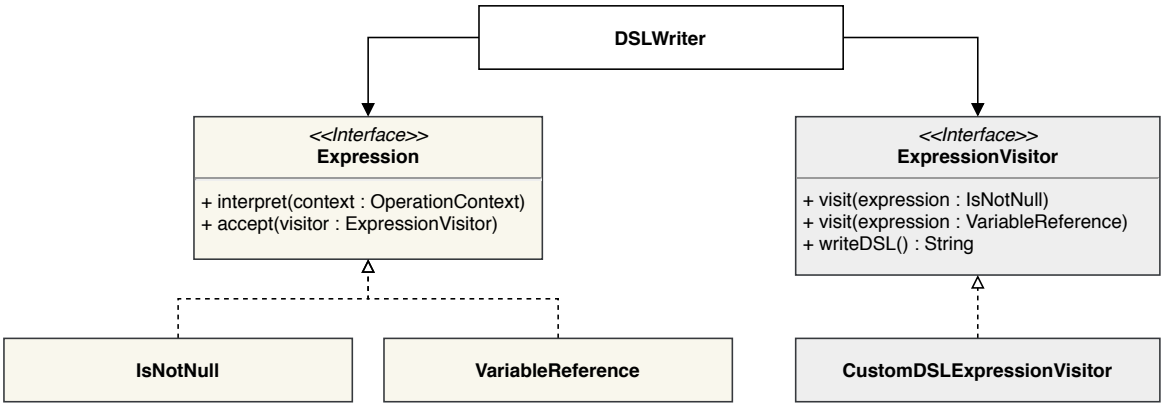
4.4 Filtrování návratových hodnot byznysové operace

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. Itentifikovanými typy post-conditions tedy jsou:

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

4.5 Metamodel byznysového kontextu

Z předchozího textu vyplývá podoba metamodelu byznysových pravidel, resp. byznysových kontextů. Kromě samotných logických výrazů musí pravidlo nést informace o tom, zda



Obrázek 4.8: Diagram tříd popisující využití vzoru Visitor pro zápis logických výrazů v DSL

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 rozšířených kontextů. Diagram tříd navrženého kontextu je znázorněn na obrázku 4.7.

4.6 Popis byznysových kontextů pomocí DSL

Přístup ADDA doporučuje popsat byznysová pravidla pomocí vlastního, na míru šitého, doménově specifického jazyka [17]. Pro účely frameworku bude popsán pomocí DSL celý byznysový kontext. Jak bylo popsáno v sekci 3.8, vlastnosti nástrojů Drools a JetBrains MPS, nejsou optimální pro dosažení vytyčených cílů. Pro účely frameworku je tedy nutné specifikovat vlastnosti, které by DSL mělo nést. Konkrétní podoba DSL bude přenechána na implementaci frameworku.

Pro uložení kontextu z metamodelu do DSL, aby ho mohl vývojář či administrátor systému upravovat, je vzhledem k reprezentaci logických výrazů vhodný návrhový vzor *Visitor* [32]. Ten umožní převádět libovolně složité logické výrazy pomocí metody *double-dispatch*. Jeho volbou je zároveň zajištěna rozšiřitelnost frameworku pro libovolné DSL – bude stačit implementovat konkrétní visitor pro zvolený jazyk, aniž by bylo nutno zasahovat přímo do implementace frameworku. Princip použití vzoru Visitor je znázorněn na obrázku 4.8.

4.7 Organizace byznysových kontextů

Každá služba bude mít lokálně uložen popis byznysových kontextů, které se sémanticky vztahují k její doméně. Pro snažší přidělení byznysových kontextů ke službám bude v iden-

tifikátoru 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`. Může nastat i situace, kdy jedna služba bude spravovat více prefixů.

4.7.1 Registr byznysových kontextů

Cílem frameworku je soustředit byznysové kontexty na jedno místo, ze kterého budou automaticky distribuovány. Pro tento účel bude využit registr byzynsových pravidel (`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. Každá služba pak bude disponovat svým registrem. Při inicializaci kontextů spolu budou registry komunikovat a vyměňovat si sdílené kontexty.

4.7.2 Uložení kontextů

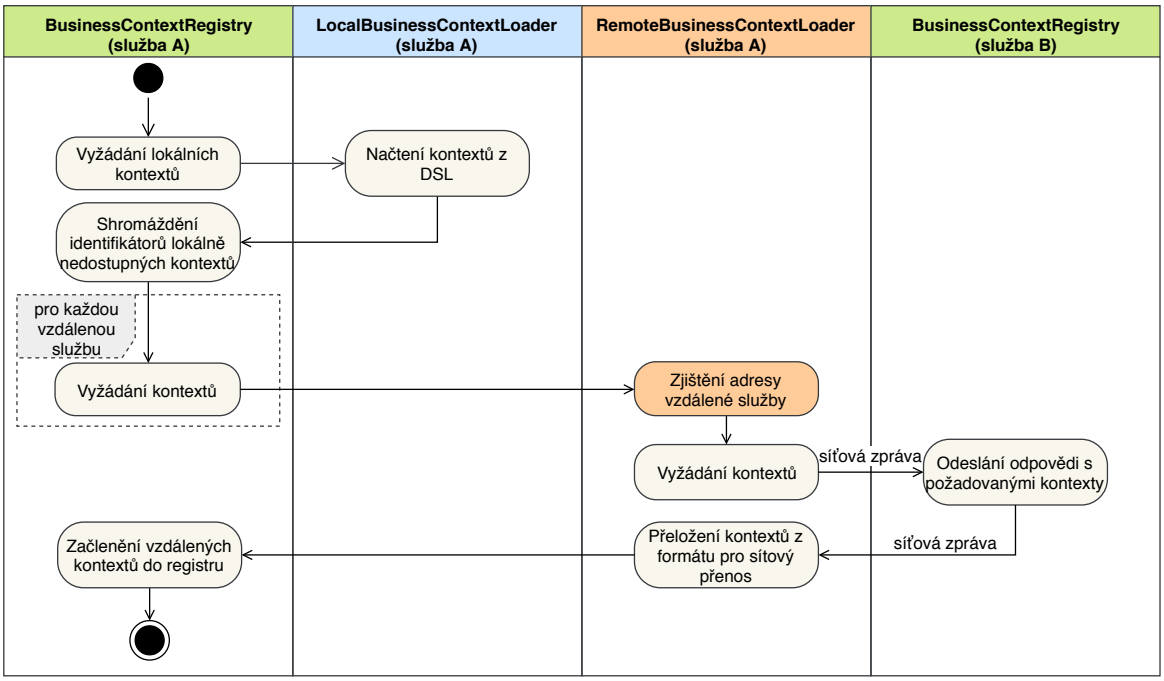
Byznysové kontexty popsané pomocí DSL mohou být v příslušné službě uloženy v souborech na disku či v databázi. Navrhovaný framework by na způsobu uložení neměl být závislý a o potřebné kroky pro načtení či případně uložení kontextu se postará konkrétní implementace. Pro tento účel je tedy vhodné, aby registr pracoval s nekonkrétními rozhraními, na jejichž implementaci nebude nijak záviset.

4.8 Inicializace byznysových kontextů

Při inicializiaci byzynsových kontextů jsou nejprve načteny lokálně dostupné kontexty popsané pomocí DSL. Po převedení kontextů z DSL do metamodelu je shromážděn seznam rozšířených kontextů 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 rozšířených kontextů začleněna do kontextů, které od nich dědí. Celou inicializaci bude zastřešovat komponenta `BusinessContextRegistry`, která má znalost o všech subsystémech, které jsou k tomuto procesu potřeba. Tato komponenta implementuje návrhový vzor *Facade* [32]. Na obrázku 4.9 je znázorněn navržený proces inicializace.

4.9 Centrální správa byznysových kontextů

Vzhledem k nutnosti centralizovat správu byznysových kontextů se architektura P2P představená v sekci 3.9.2 nehodí. Při úpravě kotextů by totiž v systému mohly existovat

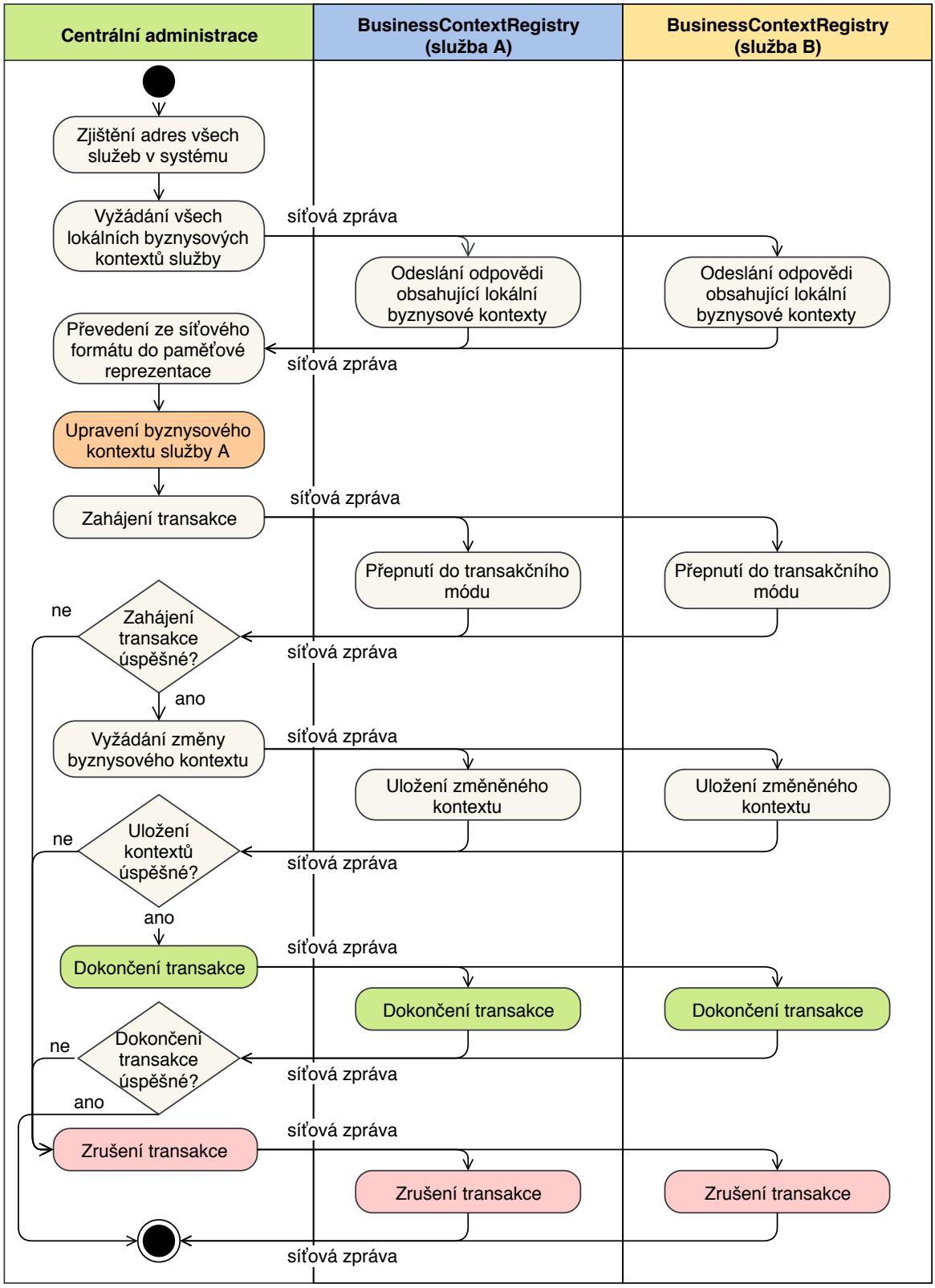


Obrázek 4.9: Diagram procesu inicializace byznysových kontextů

najednou staré i nové verze byznysových pravidel, což je pro správnou funkci systému nepřijatelné. Framework tedy využije architektury klient-server s více servery. Byznysové kontexty budou podle prefixu přideleny službám, které budou spravovat jejich aktuální a jediný stav a poskytovat je jiným službám.

4.9.1 Uložení rozšířeného pravidla

Při ukládání byznysového kontextu je potřeba změnu propagovat do všech ostatních kontextů, které od něj dědí. Při změně rozšířeného kontextu budou všechny služby, které ho využívají, informovány pomocí nástroje pro centrální správu byzynsových 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. Bylo by vhodné vybrat vhodný přenosový formát, který minimalizuje dopad veškeré síťové komunikace týkající se distribuce byznysových pravidel.



Obrázek 4.10: Diagram procesu centrální správy byznysových kontextů

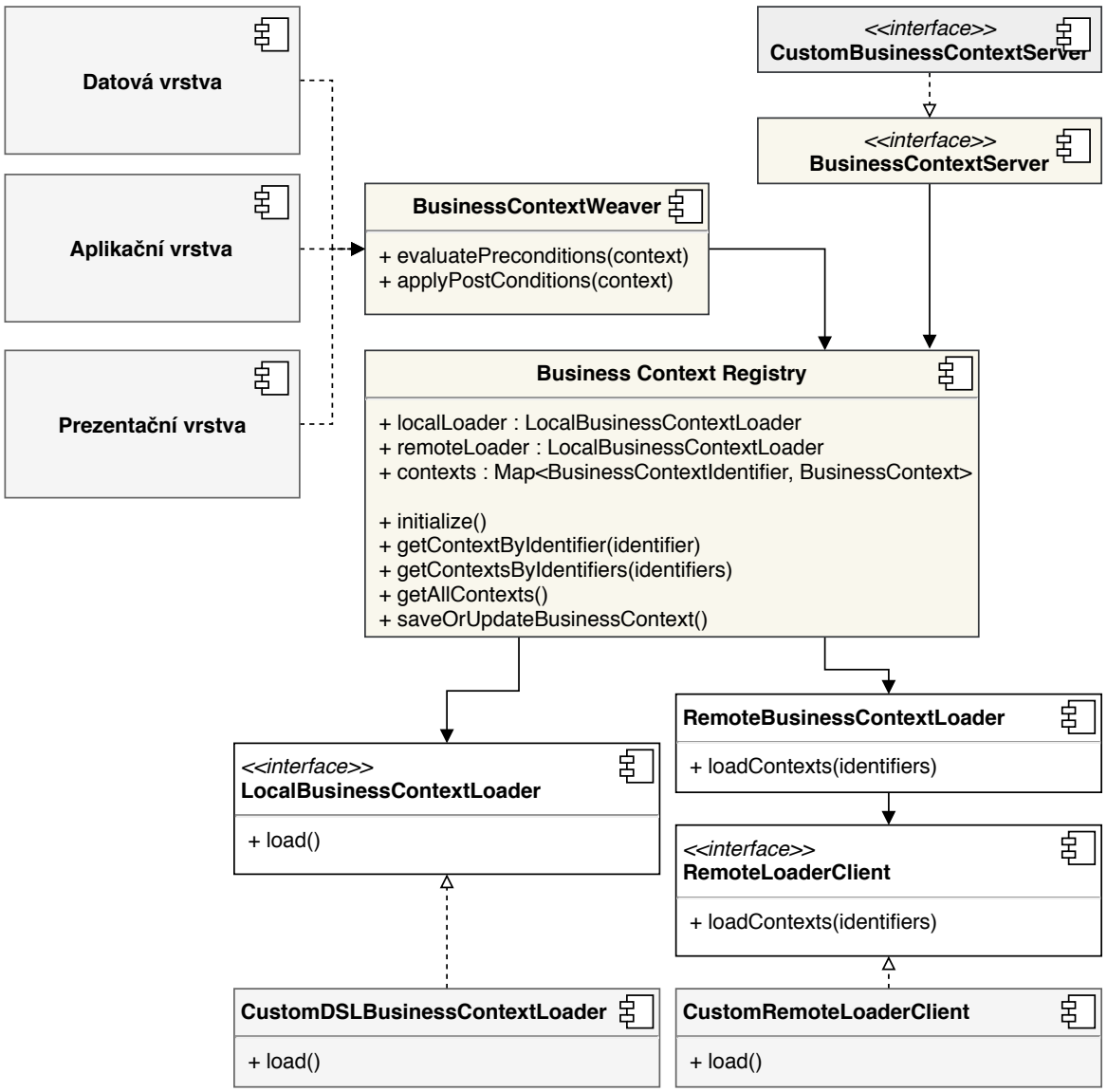
4.9.2 Proces úpravy kontextu

Proces úpravy byznysového kontextu pomocí nástroje pro centrální administraci nejprve načte všechny byznysové kontexty všech služeb v systému. Následně zobrazí administrátorovi formulář pro úpravu pravidla. Pravidlo je pro účely formuláře převedeno z metamodelu do DSL. Po odeslání formuláře bude pravidlo převedeno zpět do metamodelu. Nástroj pro administraci poté analyzuje, na které služby bude mít změna pravidla dopad. Následně je s těmito službami zahájena transakce, při které v nich nesmí probíhat žádná byznysová operace. Když všechny ovlivněné služby zahájí transakci, je možno jim rozeslat novou podobu pravidla. Pokud vše proběhne v pořádku, je možno transakci dokončit a služby otevřít byznysovým transakcím. Pokud naopak některý z kroků transakce selže, je nutno informovat všechny zúčastněné služby o zrušení transakce a změnu inkriminovaného pravidla zrušit. Na obrázku 4.10 je celý proces znázorněn. Proces pro uložení nového kontextu je analogický.

4.10 Architektura frameworku

V této sekci je popsána obecná architektura navrženého frameworku v rámci služby využívající klasickou třívrstvou architekturu [32], která se skládá z prezentační, aplikační a datové vrstvy. Každá z těchto vrstev může framework využívat – prezentační vrstva při validování vstupních polí formuláře, aplikační vrstva při aplikaci byzynsových pravidel v byznysových operacích a datová vrstva při aplikaci post-conditions pro filtrování dat při jejich získávání z databáze.

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. O to se postará třída `RemoteBusinessContextLoader`, která požadované kontexty zorganizuje podle prefixu a poté pomocí rozhraní `RemoteLoaderClient` načte pravidla od jednotlivých služeb. Implementace rozhraní `RemoteLoaderClient` 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 zastřešit 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.



Obrázek 4.11: Diagram tříd zachycující architekturu navrženého frameworku

Pro účely této práce bude framework poskytovat weavery pro využití v aplikační vrstvě pro weaving preconditions a post-conditions do byznysových operací. Architektura je zachycena na obrázku 4.11.

4.10.1 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.

4.11 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 prototypů knihoven

[1] Součástí zadání této práce je implementace prototypů knihoven pro framework navržený v kapitole 4 pro tři rozdílné platformy, z nichž jedna musí být *Java*. Tato kapitola popisuje výběr plaforem a konkrétní implementace knihoven pro tyto platformy. Jelikož jednotlivé implementace vycházejí ze stejného návrhu, kompletní implementace je opsána pouze pro platformu Java a ostatní jsou shrnuty komparativní metodou. Součástí kapitoly je i stručný popis použitých technologií pro lepší kontext.

5.1 Výběr použitých platforem

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 [57][38][67]. Tyto jazyky se v posledních letech stabilně umísťují na prvních příčkách nejpopulárnějších programovacích jazyků pro obecné použití.

5.2 Sdílení byznys kontextů mezi službami

Pro sdílení byznysových kontextů a jejich pravidel mezi jednotlivými službami bude využita síťová komunikace ve formátu nezávislém na platformě, s vysokou efektivitou přenosu.

[1] [Intended Delivery: Uvedení kapitoly a nastínění obsahu]

5.2.1 Protocol Buffers

Pro přenos byznysových kontextů byl zvolen open-source formát *Protocol Buffers*[2][71] vyvinutý společností Google[3]. Umožňuje explicitně definovat a vynucovat schéma dat, která jsou přenášena po síti, bez vazby na konkrétní programovací jazyk. Zároveň poskytuje obslužné knihovny pro vybrané platformy. Díky binární reprezentaci dat v přenosu velmi efektivní, oproti formátům jako je JSON nebo XML [49]. Oproti protokolům *Apache Thrift*[4] a *Apache Avro*[5], které poskytují velmi srovnatelnou funkcionalitu, mají Protocol Buffers kvalitnější a lépe srozumitelnou dokumentaci.

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

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

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

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

Zdrojový kód 5.1 znázorňuje zápis schématu síťových zpráv pro distribucu byznys kontexty ve formátu Protobuffer. Schéma zpráv pro výměnu kontextů dodržuje strukturu metamodelu navrženého v sekci 4.5.

ExpressionMessage obsahuje jméno, atributy a argumenty `Expression`

ExpressionPropertyMessage je enumerace obsahující typy atributu `Expression`

[2] https://developers.google.com/protocol-buffers/
[3] https://www.google.com/
[4] https://thrift.apache.org/
[5] https://avro.apache.org/

PreconditionMessage obsahuje název a podmínku precondition pravidla

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

PostConditionTypeMessage je enumerace obsahující typy post-condition pravidla

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

BusinessContextsMessage obaluje více byznys kontextů

5.2.2 gRPC

Pro realizaci architektury klient-server byl zvolen open-source framework gRPC[6], který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC* [53]. Zdrojový kód 5.2 znázorňuje zápis serveru, který umožňuje volat metody `FetchContexts`, `FetchAllContexts` a `UpdateOrSaveContext`.

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

```
service BusinessContextServer {
    rpc FetchContexts (BusinessContextRequestMessage)
        returns (BusinessContextsResponseMessage) {}

    rpc FetchAllContexts (Empty)
        returns (BusinessContextsResponseMessage) {}

    rpc UpdateOrSaveContext (BusinessContextUpdateRequestMessage)
        returns (Empty) {}
}
```

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

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

[6] https://grpc.io/

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

5.3 Doménově specifický jazyk pro popis byznys kontextů

Ačkoliv není specifikace a implementace DSL pro popis byznysových kontextů úkolem 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. Tento jazyk však bude možno v produkční verzi knihovny nahradit komplexnejším.

Pro popis kontextů byl jako kompromis mezi jednoduchostí implementace a přívětivostí pro koncového uživatele zvolen univerzální formát Extensible Markup Language (XML) [13] doplněný o definici schematu dat pomocí *XML Schema Definition* (XSD) [45]. Díky formálně definovanému schématu lze popis byznys kontextu automaticky validovat a vyvarovat se tak případných chyb.

Ve zdrojovém kódu 5.3 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 4.5. Pro zápis podmínek jednotlivých preconditions a post-conditions byl zvolen opis derivačního stromu. Toto rozhodnutí vychází z předpokladu, že lze vzhledem k povaze prototypu relaxovat podmínku na čitelnost zápisu pravidel ve prospěch jednoduššího zpracování.

Zdrojový kód 5.3: Příklad zápisu byznys kontextu v jazyce XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<businessContext prefix="user" name="createEmployee">
  <includedContexts/>
  <preconditions>
    <precondition name="Cannot use hidden product">
      <condition>
        <logicalEquals>
          <left>
            <variableReference
                objectName="product"
                propertyName="hidden"
                type="bool"/>
          </left>
          <right>
```

```
15              <constant type="bool" value="false"/>
16            </right>
17          </logicalEquals>
18        </condition>
19      </precondition>
20    </preconditions>
21    <postConditions/>
22  </businessContext>
```

5.4 Knihovna pro platformu Java

[TODO

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

]

5.4.1 Popis implementace

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  }
```

5.4.2 Použité technologie

Apache Maven Pro správu závislostí a automatickou kompilaci a sestavování knihovny napsané v jazyce java byl zvolen projekt *Maven*[7]. Tento nástroj umožňuje vývojáři komfortně a centrálně spravovat závislosti jeho projektu včetně detailního popisu jejich verze. Dále také umožňuje definovat jakým způsobem bude projekt kompilován.

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

[TODO

- Ukázka kódu knihovny

]

JDOM 2 Knihovna JDOM 2[8] poskytuje kompletní sadu nástroju pro čtení a zápis XML dokumentů. Implementuje specifikaci *Document Object Model* (DOM) [76], pomocí které lze automaticky sestavovat a číst XML dokumenty.

5.5 Knihovna pro platformu Python

Knihovna pro platformu jazyka Python využívá jeho verzi 3.6. Pomocí nástroje *pip*[9] lze knihovnu nainstalovat a využívat jako python modul. Implementace odpovídá navržené specifikaci.

[TODO

- Srovnání řešení s knihovnou Java

[7] https://maven.apache.org/
[8] http://www.jdom.org/
[9] https://pip.pypa.io/en/stable/

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

]

5.5.1 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`. Dekorátoru je potřeba předat samotný weaver, narozdíl od implementace v Javě, kdy se o předání weaveru postará dependency injection container.

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

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

        return func_wrapper

    return wrapper


weaver = BusinessContextWeaver()


class ProductRepository:
```

```
20
21      @business_operation("product.listAll", weaver)
22      def get_all(self) -> List[Product]:
23          pass
24
25      @business_operation("product.detail", weaver)
26      def get(self, id: int) -> Optional[Product]:
27          pass
```

5.5.2 Použité technologie

5.6 Knihovna pro platformu Node.js

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

5.6.1 Srovnání s knihovnou pro platformu Java

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

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

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

```
const weaver = new BusinessContextWeaver(registry)

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

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

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

5.6.2 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].

5.7 Systém pro centrální správu byznys pravidel

Součástí této práce je i implementace nástroje, který umožní centrální správu byznysových pravidel

5.7.1 Popis implementace

Systém pro centrální správu ...

Pro komfortní obsluhu centrální administrace bylo naprogramováno uživatelské rozhraní pomocí technologií Hypertext Markup Language [6] (HTML) a Cascading Style Sheets [9] (CSS). Detail byznysového kontextu v uživatelském rozhraní je zobrazen na snímku A.2 a formulář pro úpravu kontextu na snímku A.1.

[TODO

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

]

5.7.2 Detekce a prevence potenciálních problémů

Sekce 4.2 identifikuje problémy, které mohou nastat při úpravě nebo vytváření nového byznysového kontextu. Kromě syntaktických chyb, které jsou detekovány automaticky pomocí definovaného schematu, je potřeba detekovat následující sémantické chyby, které mohou být způsobeny rozšiřováním kontextů.

a) Neunikátní identifikátory byznysových pravidel

b) Závislosti na neexistujících kontextech

c) Cyklus v grafu závislostí kontextů

[10] https://yarnpkg.com/en/
[11] https://babeljs.io/
[12] https://github.com/isaacs/rimraf

Unikátnost byznysových pravidel lze zajistit postupným ukládáním jejich identifikátorů do vhodné datové struktury a kontrolovat, zda již nejsou ve struktuře obsaženy. Vhodnou strukturou je například `Set` [39]. 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. Při zvolení vhodných datových struktur lze dosáhnout lineární složitosti v závislosti na počtu hran grafu. Pro detekci cyklů v grafu závislosti pravidel popsanou v sekci 4.2 byl zvolen Tarjanův algoritmus [68]. Ten umožňuje detekci souvislých komponent 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 protoypy knihoven lze využít k implementaci služeb a k sestavení funkčního systému, jak je popisáno v následující kapitole.

Veškerý kód implementace je hostován v centrálním repozitáři ve službě GitHub[13] a je zpřístupněn pod open-source licencí MIT [75]. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

[13] https://github.com/klimesf/diploma-thesis

Kapitola 6

Verifikace a validace

Tato kapitola popisuje, jaký způsobem byla provedena verifikace naprogramovaných knihoven pomocí jednotkových a integračních testů. 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 Testování prototypů knihoven

Prototypy knihoven, jejichž implementaci je popsána v kapitole 5, byly důkladně otestovány pomocí sady jednotkových a integračních testů [48] a tím byla verifikována jejich správná funkcionalita. Způsob testování knihoven je popsán zvlášť pro každou platformu.

V rámci konceptu *continous integration* (CI) [36] byl kód po celou dobu vývoje verzován systémem Git[1], zasílán do centrálního repozitáře a s pomocí nástroje Travis CI[2] bylo automaticky spouštěno jeho sestavení a otestování. Systém zároveň okamžitě informoval vývojáře o výsledcích. To umožnilo v krátkém časovém horizontu identifikovat konkrétní změny v kódu, které do programu vnesly chybu. Tím byla snížena pravděpodobnost regrese a dlouhodobě se zvýšila celková kvalita kódu.

6.1.1 Platforma Java

Prototyp knihovny pro platformu jazyka Java byl testován pomocí nástroje JUnit[3], který poskytuje veškerou potřebnou funkcionalitu pro jednotkové i integrační testování. Všechny testy byly spouštěny automaticky při sestavování knihovny pomocí nástroje Maven[4].

[1] https://git-scm.com/
[2] https://travis-ci.org/
[3] https://junit.org/junit4/
[4] https://maven.apache.org/

Zdrojový kód 6.1: Příklad jednotkového testu knihovny pro jazyk Java s využitím nástroje JUnit 4

```java
import org.junit.Assert;
import org.junit.Test;

public class BusinessContextWeaverTest {

    /* ... */

    @Test
    public void test() {
        BusinessContextWeaver evaluator =
            new BusinessContextWeaver(createRegistry());
        BusinessOperationContext context =
            new BusinessOperationContext("user.create");

        context.setOutput(new User(
            "John Doe",
            "john.doe@example.com"
        ));

        evaluator.applyPostConditions(context);

        User user = (User) context.getOutput();
        Assert.assertEquals("John Doe", user.getName());
        Assert.assertNull(user.getEmail());
    }
}
```

Ve zdrojovém kódu 6.1 je znázorněn jednotkový test třídy `BusinessContextWeaver` ověřující, že byly správně aplikovány post-conditions daného byznys kontextu, konkrétně zakrytí 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.

6.1.2 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

[5] https://docs.python.org/3/library/unittest.html

pro jednotkové testy, lze je plně využít i pro integrační testy.

Zdrojový kód 6.2: Příklad jednotkového testu 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 jeho správnou funkcionalitu. Funkce `test_split()` ověřuje, zda konstruktor správně přijímá dva argumenty, kde první z nich je prefix identifikátoru a druhý je jméno identifikátoru. Funkce `test_single()` naopak ověřuje, zda kontruktor správně přijímá jeden argument a rozdělí ho na prefix a jméno identifikátoru. Nakonec funkce `test_str()` ověřuje správnou funkcionalitu převedení identifikátoru na textový řetězec.

6.1.3 Platforma Node.js

Jelikož tendence ve světě moderního JavaScriptu je vytvářet knihovny s co nejmenším polem působnosti, které jdou kombinovat do většího celku, byl prototyp knihovny pro platformu Node.js testován pomocí kombinace několika nástrojů. Spouštění testů obstarává knihovna *Mocha*[6], zatímco o ověřování a zápis testů ve stylu *Behaviour Driven Development* (BDD) [65] se stará knihovna *Chai*[7].

[6] https://mochajs.org/
[7] http://www.chaijs.com/

Zdrojový kód 6.3: Příklad jednotkového testu knihovny pro platformu Node.js s využitím nástroje Mocha a Chai

```
const chai = require('chai');

// Imports ...

chai.should();

describe('IsNotNull', () => {
    describe('#interpret', () => {
        it('evaluates if the argument is null', () => {
            const ctx = new BusinessOperationContext('user.create')
            let expression = new IsNotNull(new Constant(
                true,
                ExpressionType.BOOL
            ))
            let result = expression.interpret(ctx)
            result.should.equal(true)

            expression = new IsNotNull(new Constant(
                null,
                ExpressionType.VOID
            ))
            result = expression.interpret(ctx)
            result.should.equal(false)
        })
    })

    // Other tests ...
})
```

Zdrojový kód 6.3 znázorňuje použití knihoven k ověření správné funkcionality výrazu `IsNotNull`. Konkrétně je nejprve zkonstruován s konstantním argumentem typu `boolean` s hodnotou `true` a je ověřeno, že výraz se vyhodnotí jako `true`. Následně je zkonstruován výraz, kterému je předán argument `null` a je ověřeno, že výraz se vyhodnotí jako `false`.

6.2 Případová studie: e-commerce systém

Abychom mohli navržený a implementovaný framework pro centrální správu a automatickou integraci byznys pravidel verifikovat v praxi a validovat jeho myšlenku, bylo nutné vy-

zkoušet jeho nasazení při vývoji aplikace. Pro tento účel vznikla v rámci této práce případová studie na fiktivním ukázkovém e-commerce systému využívající architekturu orientovanou na služby. Na tomto příkladě demonstrujeme schopnost frameworku poradit si s průřezovými problémy v rámci SOA a také jeho schopnost plnit požadavky identifikované v sekci 2.4.

6.2.1 Use-cases

Pro ukázkový systém bylo vymodelováno třináct případů užití (z anglického *Use case* (UC) [8]), jejich přehled je v tabulce 6.1.

#	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 si může prohlížet produkty
UC06	Skladník může do systému zadávat nové produkty
UC07	Skladník může upravovat u produktů stav skladových zásob
UC08	Skladník si může zobrazovat objednávky
UC09	Skladník může upravovat stav objednávek
UC10	Administrátor si může prohlížet objednávky
UC11	Administrátor může upravovat cenu produktů
UC12	Administrátor může vytvářet uživatele (skladníky)
UC13	Administrátor může mazat uživatele (skladníky i zákazníky)

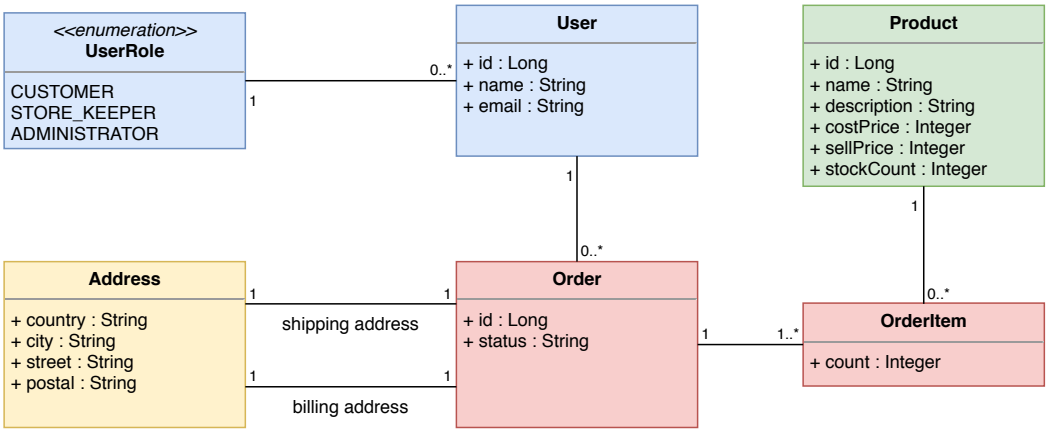
Tabulka 6.1: Přehled use-cases ukázkového e-commerce systému

6.2.2 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.
- `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.



Obrázek 6.1: Diagram tříd modelu ukázkového e-commerce systému

Tento model je využíván v každé ze služeb. Nicméně, ne každá služba využije všechny jeho entity, ale pouze jejich podmnožinu, kterou potřebuje ke svojí práci.

6.2.3 Byznysová pravidla a kontexty

V tabulce 6.2 je výčet všech dvaceti 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 6.3 vypsány všechny byznysové kontexty v ukázkové aplikaci. Některé z nich jsou konkrétní a jsou namapovány na jeden nebo více UC, jiné jsou abstraktní a slouží ostatní kontexty je rozšiřují. Prefixy byly vybrány na základě byznysové domény, ke které se kontext vzahuje, stejně jako jsou podle domén děleny i jednotlivé služby systému.

Na obrázku A.3 je vizualizována hirearchie byznysových kontextů v ukázkovém systému, jejich vazba na UC a také byznysová pravidla, která se v kontextech aplikují.

6.2.4 Služby

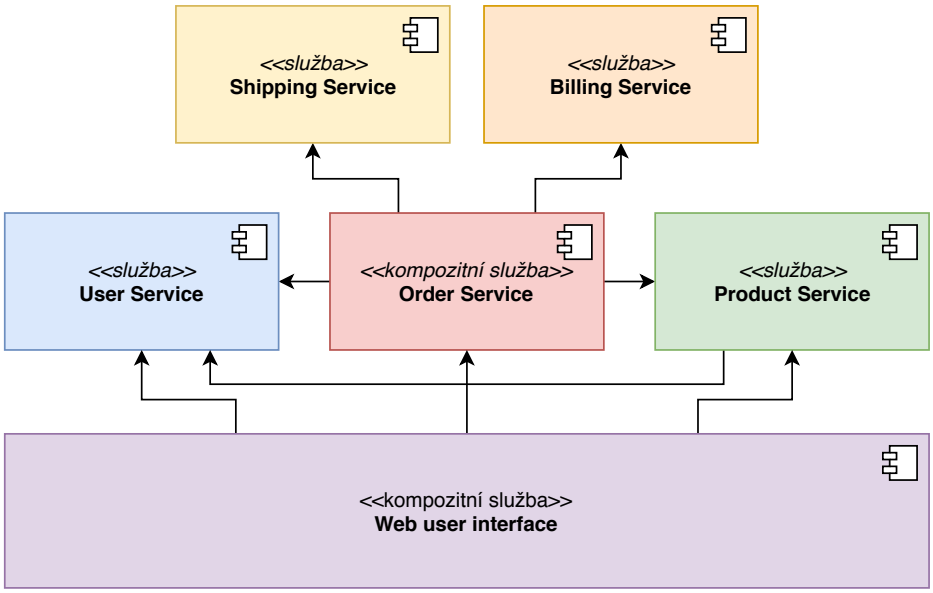
Na obrázku 6.2 jsou zobrazeny komponenty systému a jejich vzájemné závislosti. Pro ověření schopnosti podporovat více platforem byly pro implementaci systému využity jazyky

#	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	Uživatel musí řádně vyplnit doručovací adresu (č.p., ulice, město, PSČ, stát)	pre
BR05	UC04	Uživatel musí řádně vyplnit fakturační adresu (č.p., ulice, město, PSČ, stát)	pre
BR06	UC01, UC04	Zákazník musí mít vyplněnou emailovou adresu	pre
BR07	UC04	Položky objednávky musí mít počet kusů větší než 0	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 UC09	Skladník musí být do systému přihlášen a mít roli "Skladník"	pre
BR12	UC05	Skladník u produktu nesmí vidět nákupní cenu, pouze výslednou cenu	post
BR13	UC06	Produkt musí mít jméno s délkou >5	pre
BR14	UC07	Stav zásob produktů musí být číslo větší nebo rovno 0	pre
BR15	UC08	Skladník nesmí vidět celkový součet cen objednávek	post
BR16	UC09	Stav objednávky musí být pouze "přijato", "expedováno"a "doručeno"	pre
BR17	UC10 až UC13	Administrátor musí být do systému přihlášen a mít roli "Administrátor"	pre
BR18	UC11	Výsledná cena produktu musí být větší než jeho nákupní cena	pre
BR19	UC12	Skladník musí mít jméno delší než 2 znaky	pre
BR20	UC12	Skladník musí mít emailovou adresu v platném formátu	pre

Tabulka 6.2: Přehled byznysových pravidel ukázkového e-commerce systému

Identifikátor	Use-cases	Byznysová pravidla
auth.adminLoggedIn	-	BR17
auth.employeeLoggedIn	-	BR11
auth.userLoggedIn	-	BR10
billing.correctAddress	-	BR05
order.addToBasket	UC03	BR02, BR08, BR10
order.changeState	UC09	BR04, BR05, BR06, BR08, BR09, BR11, BR16
order.create	UC04	BR03, BR04, BR05, BR06, BR07, BR08, BR09, BR10, BR16
order.listAll	UC08, UC10	BR11, BR15
order.valid	-	BR04, BR05, BR06, BR09, BR16
product.buyingPrice	-	BR03
product.changePrice	UC11	BR17, BR18
product.changeStock	UC07	BR08, BR11, BR14
product.create	UC06	BR10, BR11, BR13
product.hidden	-	BR02
product.listAll	UC02, UC05	BR02, BR03, BR12
product.stock	-	BR08
shipping.correctAddress	-	BR04, BR09
user.createCustomer	UC01	BR01, BR06
user.createEmployee	UC12	BR06, BR17, BR19, BR20
user.delete	UC13	BR17, BR21
user.validEmail	-	BR06

Tabulka 6.3: Přehled byznysových kontextů ukázkového e-commerce systému



Obrázek 6.2: Diagram komponent ukázkového e-commerce systému

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 nebudeme dále věnovat. Pro demonstrativní účely byly síťové adresy nastaveny přímo v kódu jednotlivých služeb. Nicméně, navržený framework nevynucuje tento přístup, a tudíž by složitější způsob *service discovery* nebylo problém do systému integrovat.

Billing service Služba *Billing service* má na starosti funkcionalitu týkající se fakturace objednávek a byla implementována v jazyce Java s použitím frameworku Spring Boot[8].

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 {
3
4     /* ... */
5
```

[8] https://projects.spring.io/spring-boot/

```
6       @GetMapping("/shopping-cart")
7       public ResponseEntity<?> listShoppingCart() {
8           List<ShoppingCartItem> shoppingCartItems = shoppingCartFacade
9               .listShoppingCartItems();
10          return new ResponseEntity<>(
11              new ListShoppingCartItemsResponse(
12                  shoppingCartItems.size(),
13                  shoppingCartItems
14              ),
15              HttpStatus.OK
16          );
17      }
18
19  }
```

Product service Služba *Product service* realizuje UC týkající se prohlížení a administrací nabízených produktů a jejich skladových zásob. Služba byla implementována v jazyce Python. Pro vytvoření REST API služby byl využit populární light-weight framework *Flask*[9]. Ve zdrojovém 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
3  app = Flask(__name__)
4  product_repository = ProductRepository()
5
6  @app.route("/")
7  def list_all_products():
8      result = []
9      for product in product_repository.get_all():
10         result.append({
11             'id': product.id,
12             'sellPrice': product.sellPrice,
13             'name': product.name,
14             'description': product.description
15         })
16     return jsonify(result)
```

[9] http://flask.pocoo.org/

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 Express[10]. Ve zdrojovém kódu 6.6 je ukázka mapování controllerů na jednotlivé metody URL `/users`.

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

```
module.exports = app => {
    const userController = require('../controllers/userController')

    app.route('/users')
        .get(userController.listUsers)
        .post(userController.register)

    app.route('/users/:userId')
        .get(userController.getUser)
}
```

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 A.4 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 popsaný v sekci 5.7. Systém byl napojen na všechny služby systému, kromě webového UI, a bylo úspěšně prokázáno, 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 [51], která umožňuje vytvořit virtuální běhové prostředí pro aplikaci pomocí kontejnerizace využívající virtualizaci nad operačním systémem. Uživatel si nadefinuje tzv. *image*, který se skládá z jednotlivých vrstev. Základní vrstvou je operační systém,

[10] https://expressjs.com/

dalšími mohou být jednotlivé knihovny instalované do systému. Příklad definice image pomocí technologie Docker je znázorněn ve zdrojovém kódu 6.7. Konkrétně se jedná o definici image, který rozšiřuje oficiální image `library/node:9.11.1`[11] stavějící nad operačním systémem *Linux*[12], a přidává vrstvy s prototypem knihovny pro platformu Node.js.

Zdrojový kód 6.7: Ukázka zápisu Docker image obsahující knihovnu pro platformu Node.js

```
FROM library/node:9.11.1

WORKDIR /usr/src/framework
COPY ./nodejs/business-context ./business-context
COPY ./nodejs/business-context-grpc ./business-context-grpc
COPY ./proto ./proto

RUN cd ./business-context \
    && yarn install \
    && yarn link \
    && npm run-script build \
    && cd ../business-context-grpc \
    && yarn install \
    && yarn link business-context-framework \
    && yarn link \
    && npm run-script build
```

Spouštění služeb Pro samotné spuštění byla využita funkce *Compose*, která umožňuje definovat a spouštět více-kontejnerové aplikace. Ve zdrojovém kódu 6.8 můžeme vidět zápis Order service. Pro její image je použit `filipklimes-diploma/example-order-service`. V sekci `ports` deklarujeme, že služba má mít z vnějšku přístupný port `5501`, na kterém poskytuje své REST API, a port `5551`, nak terém poskytuje své gRPC API pro sdílené byzynsových kontextů. Order service je závislá na Product, Billing, Shipping a User service, což explicitně specifikujeme v sekci `depends_on`, aby Docker Compose mohl spustit služby ve správném pořadí. Nakonec pomocí `links` deklarujeme, 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 [4], který je dnes běžně využíván pro konfigurační soubory, kvůli jeho snadné čitelnosti pro člověka a jednoduchému používání.

Zdrojový kód 6.8: Ukázka zápisu více-kontejnerové aplikace pro Docker Compose

```
version: '3'
```

[11] https://hub.docker.com/_/node/
[12] https://www.linuxfoundation.org/projects/linux/

```
services:
  order:
    image: filipklimes-diploma/example-order-service
    ports:
      - "5501:5501"
      - "5551:5551"
    depends_on:
      - product
      - billing
      - shipping
      - user
    links:
      - product
      - user
      - billing
      - shipping
```

6.3 Srovnání s konvenčním přístupem

[13] Z tabulky 6.3 plyne, že 60 % byznysových pravidel v ukázkovém systému je využíváno ve více kontextech, a polovina je využívána napříč více službami. V tabulce 6.4 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.

[14] Díky použití navrženého frameworku je však možné každé pravidlo nadefinovat centrálně a framework se postará o jeho automatickou integraci do všech částí systému, kde má být aplikováno. Díky tomu je možno 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.

[15] Jako nevýhodu použití frameworku lze považovat počáteční investici v podobě integrace knihoven do služeb systému. Zvážit musíme i cenu popisu byznysových pravidel v DSL, který se musejí vývojáři systému naučit navíc oproti programovacímu jazyku, ve kterém popisují služby. Dále je při návrhu systému potřeba identifikovat byznysové kontexty, jejich

[13] [Intended Delivery: Ukázka na konkrétním příkladě]
[14] [Intended Delivery: Výhody frameworku]
[15] [Intended Delivery: Nevýhody použití]

#	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**	order
BR07	order	**BR17**	(auth), product, user
BR08	order, product	**BR18**	product
BR09	order, shipping	**BR19**	user
BR10	(auth), order, product	**BR20**	user

Tabulka 6.4: Přehled využití byznysových pravidel ve službách ukázkového systému

hirearchii a vzájemnou vazbu s byznysovńi pravidly, aby bylo možno framework efektivně využívat. To může vyžadovat více času, než klasický návrh.

[16] Navržený framework tedy oproti konvenčnímu přístupu nabízí možnost získat dlouhodobě nižší náklady na vývoj za cenu počáteční investice. Architekt softwarového systému musí případné nasazení frameworku zvážit z několika úhlů pohledu a posoudit, zda bude životnost systému dostatečně dlouhá a systém dostatečně velký. Dalším podstatným bodem ke zvážení je reálná míra znovupoužití byznysových pravidel. Mohou existovat domény, ve kterých bude nasazení frameworku jistě mnohem vhodnější, než v jiných. Díky provedené případové studii bylo prokázáno, že v SOA lze efektivně řešit sdílení byznysových pravidel navrženým způsobem.

6.4 Shrnutí

Tato kapitola popisuje, jakým způsobem byly testovány prototypy knihoven pro platformy jazyků Java a Python a pro platformu Node.js. Tím byla verifikována jejich správná fukcionalita. Dále kapitola specifikuje ukázkový systém, na kterém byla provedena případová studie použití frameworku, a popisuje jeho implementaci. Díky studii bylo zvalidováno, že navržený framework je funkční a splňuje požadavky identifikované v sekci 2.4. Nakonec je na ukázkovém systému změřen počet duplikací pravidel a je srovnáno použití frameworku a konvenčního přístupu k návrhu a implementaci softwarových systému.

[16] [Intended Delivery: Závěr]

Kapitola 7

Závěr

Architektura orientovaná na služby usnadňuje vývoj komplexních informačních systému díky jejich členění do menších, samostatných celků. Díky tomu lze snáze oddělit zodpovědnost a zvýšit znovupoužitelnost komponent systému. Existuje však funkcionalita, která zasahuje do více služeb najednou a je potřeba ji všude vykonávat konzistentně. Zástupcem této funkcionality jsou byznysová pravidla, která zajišťují validní vykonávání byznysových procesů a konzistenci 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 Přínos a možnosti použití frameworku

Framework navržený v této práci přináší způsob, kterým mohou vývojáři systémů stavějících na SOA výrazně ušetřit náklady spojené s manuální duplikací byznysových pravidel v jednotlivých službách. Framework je nezávislý na platformě a díky tomu může být využíván i v heterogenních systémech, jejichž služby využívají rozdílné technologie. Ačkoliv se tato práce odkazuje zejména na třívrstvou architekturu, framework neklade nároky na architekturu jednotlivých služeb.

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

7.2 Budoucí rozšiřitelnost frameworku

Navržený framework podporuje budoucí rozšíření o další moduly a funkce, které dále zvýší jeho schopnost znovupoužití byzynsových pravidel v SOA. Díky svobodné licenci MIT může framework rožšířit a volně používat kdokoliv.

7.2.1 Univerzální doménově specifický jazyk

Zadáním této práce nebylo zkonstruovat vlastní DSL k účelům automatické integrace a centrální správy byznys 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, popsané v sekci 5.3. Tento jazyk je však 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.

7.2.2 Integrace frameworku s uživatelským rozhraním

V sekci 4.10 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 by bylo umožněno další snížení nákladů na vývoj a údržbu systému využívající framework. Zároveň by došlo ke zvýšení uživatelského komfortu díky real-time validaci vstupních hodnot formulářů.

7.2.3 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.

7.3 Další možnosti uplatnění AOP v SOA

Byznysová pravidla nejsou jediným průřezovým problémem, se kterým se systémy stavějící na SOA musejí vypořádat. 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. Pomocí 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.

Posledním průřezovým problémem v SOA je společná konfigurace parametrů služeb, zejména pak v decentralizovaném systému využívajícím architekturu Microservices. Příkladem takového parametru může být v ukázkovém e-commerce systému výše DPH, která se využívá 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 problémům.

7.4 Shrnutí

V rámci této práce byly dosaženy cíle stanovené v úvodu práce a byly splněny všechny požadavky zadání. Nejprve byla provedena analýza využití, vyjádření a znovupoužití byznysových pravidel v SOA a byly identifikovány potenciální problémy a požadavky na framework navržený v této práci. Dále byla provedena rešerše vhodných architektur, paradigmat a nástrojů, které by mohly být použity pro řešení těchto problémů, včetně stávajících frameworků pro reprezentaci 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.js. Tyto knihovny byly použity pro vývoj jednoduché ukázkové e-commerce aplikace, která je tvořena šesti službami naprogramovanými ve třech různých jazycích. Na této aplikaci byla ukázána schopnost frameworku centrálně administrovat byznysová pravidla, sdílet a automaticky je integrovat do služeb aplikace. Zároveň byl změřen a analyzován dopad použití frameworku na počet duplikací byznysových pravidel a byl diskutován jeho vliv na údržbu systému. V závěru práce byly navíc analyzovány další oblasti SOA, kde by bylo možné aplikovat aspektově orientované programování.

Literatura

[1] *A Hands-on Introduction to BPEL* [online]. Dostupné z: <http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html>.

[2] ANDREWS, T. et al. Business process execution language for web services, 2003.

[3] BAKSHI, K. Microservices-based software architecture and approaches. In *Aerospace Conference, 2017 IEEE*, s. 1–8. IEEE, 2017.

[4] BEN-KIKI, O. – EVANS, C. – INGERSON, B. Yaml ain't markup language (yaml™) version 1.1. *yaml. org, Tech. Rep.* 2005, s. 23.

[5] BERNARD, E. – PETERSON, S. JSR 303: Bean validation. *Bean Validation Expert Group, March.* 2009.

[6] BERNERS-LEE, T. – CONNOLLY, D. Hypertext markup language-2.0. Technical report, 1995.

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

[8] BITTNER, K. *Use case modeling.* Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] BOS, B. et al. Cascading style sheets, level 2 CSS2 specification. *Available via the World Wide Web at http://www. w3. org/TR/1998/REC-CSS2-19980512.* 1998, s. 1472–1473.

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

[11] 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.

[12] 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.

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

[14] CEMUS, K. Context-aware input validation in information systems. In *POSTER 2016-20th International Student Conference on Electrical Engineering*, 2016.

[15] 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.

[16] CEMUS, K. – CERNY, T. Automated extraction of business documentation in enterprise information systems. *ACM SIGAPP Applied Computing Review*. 2017, 16, 4, s. 5–13.

[17] CEMUS, K. – CERNY, T. – DONAHOO, M. J. Automated business rules transformation into a persistence layer. *Procedia Computer Science*. 2015, 62, s. 312–318.

[18] 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.

[19] 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.

[20] 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.

[21] CERNY, T. – DONAHOO, M. J. – TRNKA, M. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*. 2018, 17, 4, s. 29–45.

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

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

[24] CZARNECKI, K. et al. Generative programming and active libraries. In *Generic Programming*. Cham, Switzerland: Springer, 2000. s. 25–39.

[25] DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*. Cham, Switzerland: Springer, 2017. s. 195–216.

[26] 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.

[27] *ECMAScript® 2015 Language Specification - Ecma-262 6th Edition* [online]. Dostupné z: <http://www.ecma-international.org/ecma-262/6.0/>.

[28] FICHMAN, R. G. – KOHLI, R. – KRISHNAN, R. Editorial overview—the role of information systems in healthcare: current research and future trends. *Information Systems Research*. 2011, 22, 3, s. 419–428.

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

[30] FORGY, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*. New York, USA: Elsevier, 1988. s. 547–559.

[31] FORMAN, I. R. – FORMAN, N. – IBM, J. V. Java reflection in action. 2004.

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

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

[34] FOWLER, M. *Domain-specific languages*. London, England, UK : Pearson Education, 2010.

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

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

[37] FOX, G. Peer-to-peer networks. *Computing in Science & Engineering*. 2001, 3, 3, s. 75–77.

[38] *GitHub Octoverse 2017* [online]. 2017. Dostupné z: <https://octoverse.github.com/>.

[39] HOPCROFT, J. E. – ULLMAN, J. D. *Data structures and algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing, 1983.

[40] 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.

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

[42] KLEPPE, A. G. et al. The model driven architecture: practice and promise, 2003.

[43] 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.

[44] LARMAN, C. – APPLYING, U. Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2001.

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

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

[47] LITTMAN, D. C. et al. Mental models and software maintenance. *Journal of Systems and Software*. 1987, 7, 4, s. 341–355.

[48] LUO, L. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*. 2001, 15232, 1-19, s. 19.

[49] 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.

[50] MELICHAR, B. v. i. *Jazyky a p ř eklady*. Praha, Česká republika : Vydavatelstv í Č VUT, 2003.

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

[52] MORGAN, T. *Business rules and information systems: aligning IT with business goals*. Boston, Massachusetts, USA : Addison-Wesley Professional, 2002.

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

[54] NEMURAITE, 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.

[55] 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.

[56] PARDON, G. – PAUTASSO, C. Towards distributed atomic transactions over RESTful services. In *REST: From Research to Practice*. Cham, Switzerland: Springer, 2011. s. 507–524.

[57] *Programming Languages and GitHub* [online]. 2014. Dostupné z: <http://githut.info/>.

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

[59] RICHARDS, M. Microservices vs. service-oriented architecture. 2015.

[60] 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.

[61] 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.

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

[63] SOBEL, J. M. – FRIEDMAN, D. P. An introduction to reflection-oriented programming. In *Proceedings of reflection*, 96, 1996.

[64] SOLEY, R. et al. Model driven architecture. *OMG white paper*. 2000, 308, 308, s. 5.

[65] SOLIS, C. – WANG, X. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, s. 383–387. IEEE, 2011.

[66] 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.

[67] *Stack Overflow Developer Survey 2017* [online]. 2017. Dostupné z: <https://insights.stackoverflow.com/survey/2017#technology>.

[68] TARJAN, R. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, s. 114–121, Oct 1971. doi: 10.1109/SWAT.1971.10.

[69] *The Role of Service Orchestration Within SOA* [online]. Dostupné z: <https://www.nomagic.com/news/insights/the-role-of-service-orchestration-within-soa>.

[70] VANDEVOORDE, D. – JOSUTTIS, N. M. *C++ Templates*. Boston, Massachusetts, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.

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

[72] VISSER, E. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*, s. 299–315. Springer, 2002.

[73] WARD, M. P. Language-oriented programming. *Software-Concepts and Tools*. 1994, 15, 4, s. 147–161.

[74] WARMER, J. B. – KLEPPE, A. G. The object constraint language: Precise modeling with uml (addison-wesley object technology series). 1998.

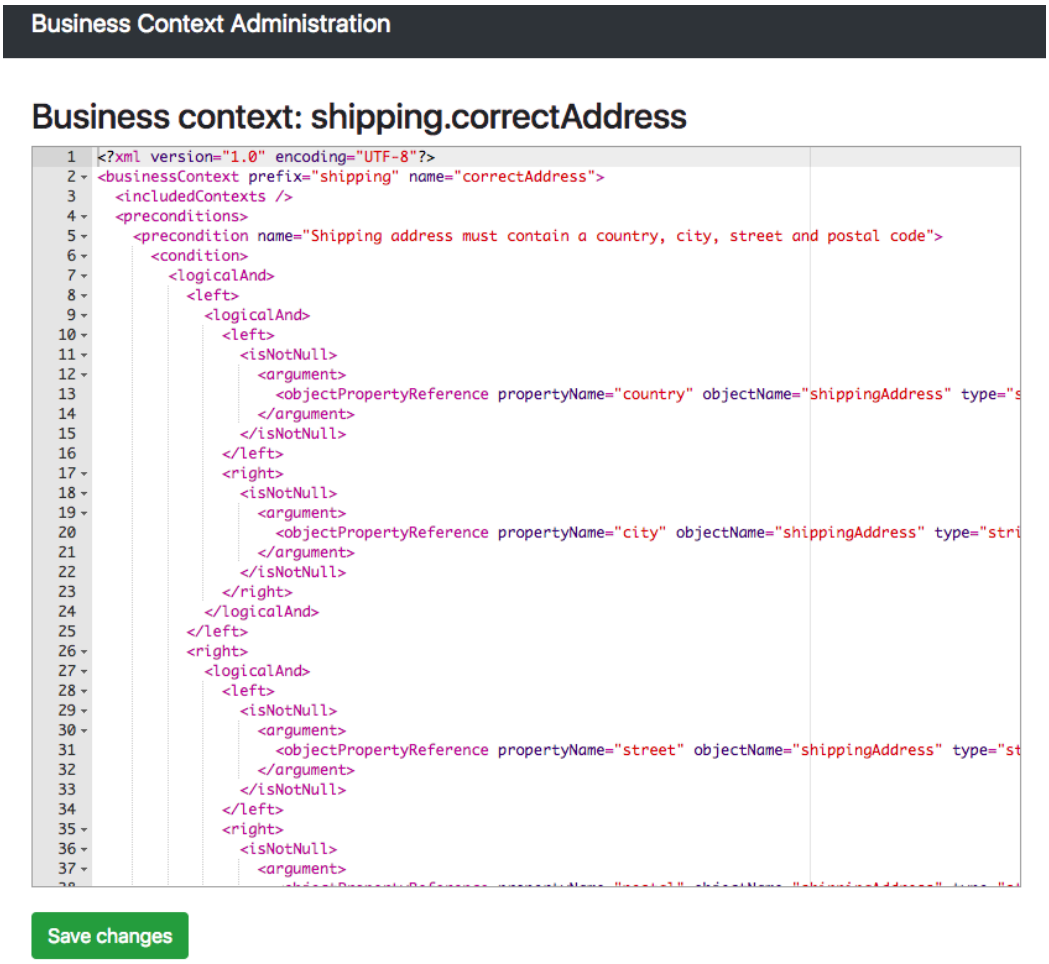
[75] *What is the MIT license? – definition by The Linux Information Project (LINFO)* [online]. Dostupné z: <http://www.linfo.org/mitlicense.html>.

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

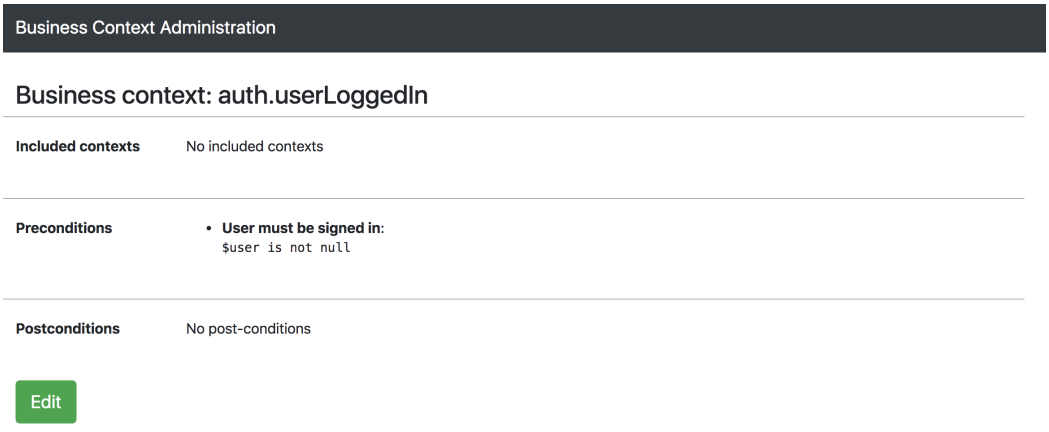
[77] XIAO, Z. – WIJEGUNARATNE, I. – QIANG, X. Reflections on SOA and Microservices. In *Enterprise Systems (ES), 2016 4th International Conference on*, s. 60–67. IEEE, 2016.

Příloha A

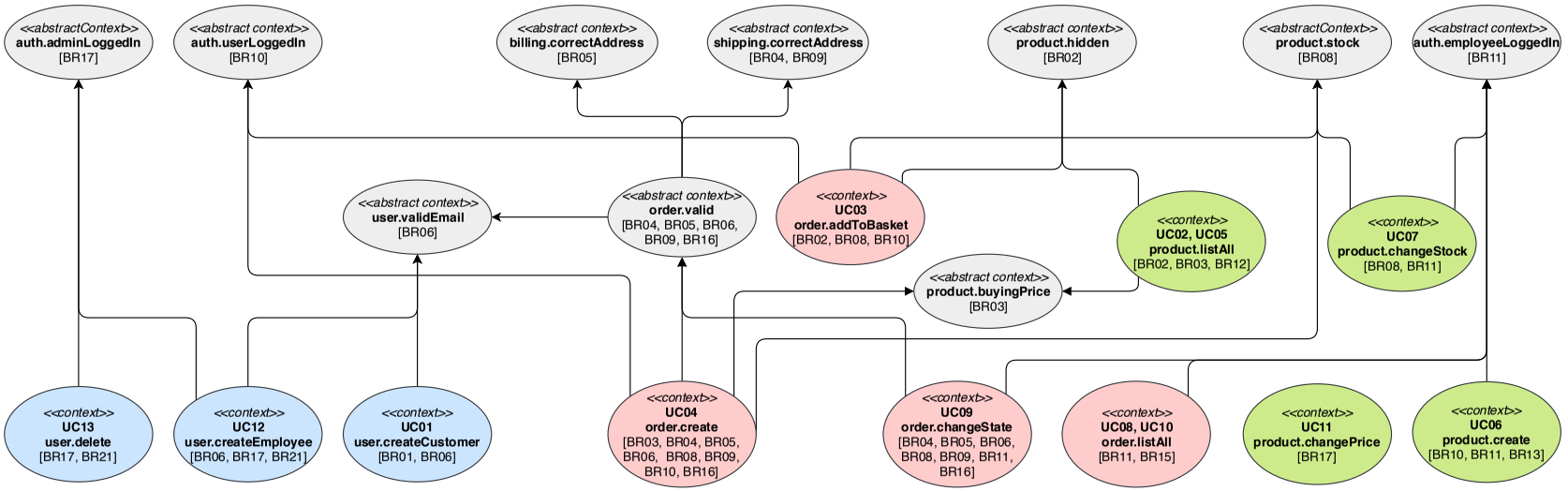
Přehledové obrázky a snímky



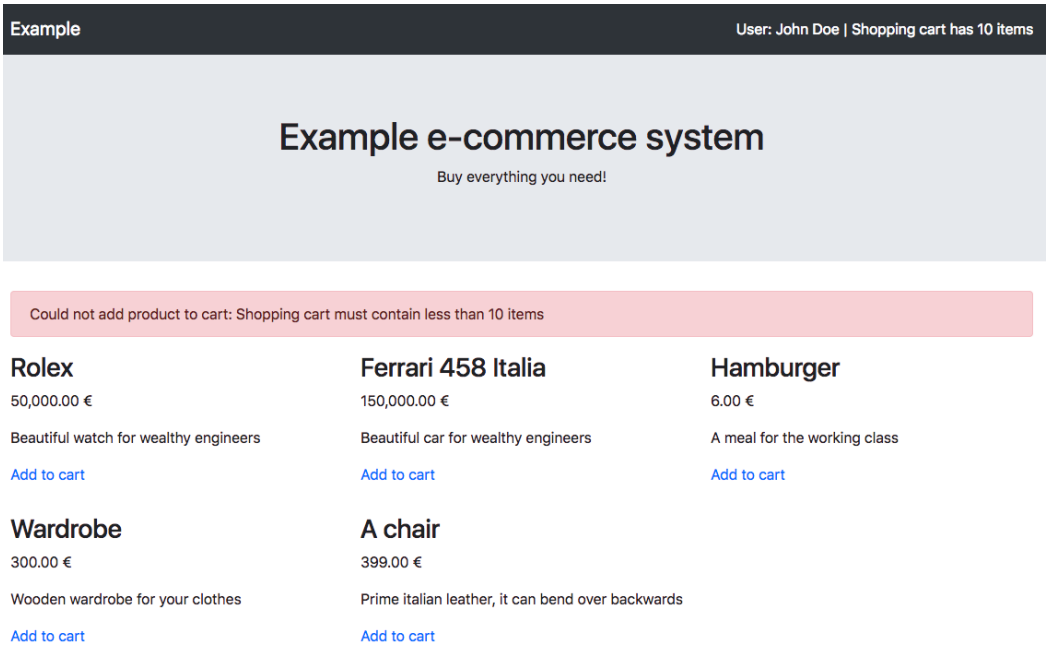
Obrázek A.1: Formulář pro vytvoření nebo úpravu byznysového kontextu v centrální administraci



Obrázek A.2: Detail byznysového kontextu v centrální administraci



Obrázek A.3: Diagram hirearchie byznysových kontextů ukázkového systému



Obrázek A.4: Propagace byznysového pravidla při přidávání produktu do košíku v ukázkovém systému

Příloha B

Přehledové tabulky

Název	Argumenty	Atributy	Návratový typ	Typ výrazu
Constant	-	Hodnota a typ konstanty	`?`	Terminál
FunctionCall	Libovolný počet argumentů	Návratový typ funkce	`?`	Terminál
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	`?`	Terminál
VariableReference	-	Název a typ proměnné	`?`	Terminál

Tabulka B.1: Přehled výrazů pro zápis byznysového pravidla

Příloha C

Uživatelská příručka

[TODO

- Popsat docker
- Popsat instalaci pomocí maven
- Oblšhnout vlastně to co je v readme

]

Příloha D

Seznam použitých zkratek

ADDA Aspect-Driven Design Approach. 16, 17, 21, 29, 35, 63–65
AOP Aspect Oriented Programming. xii, 14–16, 21, 35, 65
API Application Programming Interface. 57, 58, 60
BDD Behaviour Driven Development. 51
BPEL Business Process Execution Language. 12
BRMS Business Rules Management System. 17
CI Continuous Integration. 49
CIM Computation Independent Model. 11
CORBA Common Object Request Broker Architecture. 6
CSS Cascading Style Sheets. 46
DAG Directed Acyclic Graph. 26
DOM Document Object Model. 42
DSL Domain-Specific Language. xi, xiii, 5, 10, 12, 16–19, 29, 30, 33, 35, 40, 47, 61, 63, 64
EIS Enterprise Information System. 3–5, 7, 11, 13, 16, 64, 65
EL Expression Language. 5
ESB Enterprise Service Bus. 7, 8
GP Generative Programming. 12
HTTP Hypertext Transfer Protocol. 6, 20
IBM International Business Machine. 17
Java EE Java Platform, Enterprise Edition. 17
JPQL Java Persistence Query Language. 16
JSON JavaScript Object Notation. 38, 57
JSR Java Specification Request. 22, 27
LHS Left-hand side. 18
LOP Language-Oriented Programming. 18

MDA Model-Driver Architecture. 11, 19
MIT Massachusetts Institute of Technology. 47, 64
MQ Message Queue. 6
OCL Object Constraint Language. 5
OOP Object Oriented Programming. 11, 13, 15
ORB Object Request Broker. 6
P2P Peer-to-peer. 19, 30
PIM Platform Independent Model. 11
PSM Platform Specific Model. 11
REST Representational State Transfer. 20, 57, 58, 60
RHS Right-hand side. 18
RPC Remote Procedure Call. 20, 39
SOA Service Oriented Architecture. xii, 6–10, 12, 19–21, 35, 53, 62–65
SOAP Simple Object Access Protocol. 6
SQL Structured English Query Language. 16, 64
UC Use Case. 53, 54, 58
UI User Interface. 16, 59, 64, 65
UML Unified Modeling Language. 11
URL Uniform Resource Locator. 59
WSDL Web Service Description Language. 6
XML Extensible Markup Language. xvii, 12, 38, 40, 42, 64
XSD XML Schema Definition. 40
YAML YAML Ain't Markup Language. 60

Příloha E

Obsah přiloženého CD

```
|-- nutfroms-example/       Ukázkov\'y systém využ\'{\i}vaj\'{\i}c\'{\i} knihovnu
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- src/                 Zdrojov\'y kód aplikace
|
|-- nutforms-ios-client/    Klientská část knihovny pro platformu iOS
|  |-- client/              Zdrojové soubory knihovny
|  |-- clientTests/         Zdrojové soubory testů knihovny
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|
|-- nutfroms-server/        Serverová část knihovny
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- layout/              Layout servlet
|  |-- localization/        Localization servlet
|  |-- meta/                Metadata servlet
|  |-- widget/              Widget servlet
|
|-- nutforms-web-client/    Klientská část knihovny pro webové aplikace
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- src/                 Zdrojové soubory knihovny
|  |-- test/                Zdrojové soubory testů knihovny
|
|-- text/                   Text bakalářské práce
```