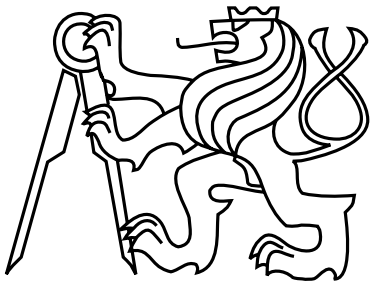
Na tomto místě bude oficiální zadání vaší práce

- 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

29. dubna 2018

Poděkování

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

Prohlášení

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

V Praze dne 20. 5. 2018 .....

Abstract

Translation of Czech abstract into English.

Abstrakt

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

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Kapitola 1

Úvod

[1]

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 mnoho aspektů našich činností. Jejich využití sahá do mnoha sektorů, od vzdělání a vědy, kde významně usnadnili přístup ke studijním materiálům, přes zdravotnictví, kde pomáhají zvyšovat efektivitu a úroveň léčby pacientů [28], až po sociální sítě, kde umožňují lidem globálně komunikovat a sdílet své myšlenky, pocity a zážitky. Jako budoucí profesionálové v oblasti informačních technologií máme za úkol nejen rozvíjet a vylepšovat funkcionalitu informačních systému, ale také zkoumat možnosti, kterými bychom mohli tento proces zjednodušit a zefektivnit. Jen díky tomu budeme moci dostát stále rostoucímu množství požadavků na informační systémy.

[2]

[3]

[4] Motivací této práce je prozkoumat myšlenku inovativního přístupu k centrální správě a automatické distribuci byznysových pravidel, která by usnadnila práci vývojářů a doménových expertů. Díky tomu by tento přístup mohl přinést snížení nákladů na vývoj a údržbu systémů využívajících architekturu orientovanou na služby.

[5] 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 implementaci knihovny pro centrální správu a automatickou distribuci byznysových pravidel v této architektuře. Kapitola 3 se zabývá rešerší stávají-

[1] [Intended Delivery: Informační systémy a jejich důležitost]
[2] [Intended Delivery: SOA]
[3] [Intended Delivery: Byznysová pravidla]
[4] [Intended Delivery: Motivace a cíle]
[5] [Intended Delivery: Popis struktury DP a obsah kapitol]

cích 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. Kapitola 4 formalizuje prostředí architektury orientované na služby do terminologie aspektově orientovaného programování a na základě této formalizace navrhuje koncept frameworku, který realizuje centrální správu a automatickou distribuci byznysových pravidel. V kapitole 5 je detailně probrána implementace knihoven pro navržený framework pro platformy 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 moderního trendu v podobě microservices. Na základě toho kapitola popisuje nedostatky současných přístupů při řešení průřezových problémů v těchto architekturách, s důrazem na byznysová pravidla. V závěru kapitoly jsou identifikovány požadavky, které by měl splňovat framework, jež bude výstupem této diplomové práce.

2.1 Byznysová pravidla

Informační systémy (IS) mají za úkol ulehčit, automatizovat či poskytovat podporu pro byznysové procesy společností, které je využívají. Tyto procesy jsou tedy stěžejním prvkem IS. Systém má také za úkol uchovávat a spravovat data společnosti a měl by zaručit, že nedojde k jejich poškození či narušení jejich integrity. Byznysové procesy, potažmo byznysové operace, proto musejí podléhat jasně definovaným byznysovým pravidlům, která zajišťují konzistenci dat informačního systému a také zabraňují nepovoleným operacím [17].

Byznysová pravidla dělíme do tří skupin [15]:

Bezkontextová pravidla jsou validační pravidla, která musejí být obecně platná v každé 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. Uvažme pravidlo „*V systému nesmí probíhat žádné změny po dobu účetní uzávěrky*“.

Dále také rozlišujeme dva typy byznysových pravidel, 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.

Pro lepší ilustraci uveďme příklad: aby mohla být provedena registrace uživatele s danou emailovou adresu, 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. Příkladem může být anonymizace uživatelů při vytváření statistického reportu e-commerce společnosti – po vygenerování reportu post-condition zajistí, že z něj budou smazány veškeré citlivé údaje. Dalším případem může být filtrování výstupu byznysové operace. Například při výpisu objednávek pro zákazníka se chceme ujistit, že všechny vypsané objednávky patří danému zákazníkovi.

2.1.3 Reprezentace byznysového pravidla

Existuje několik možností, jak zachytit a reprezentovat byznysová pravidla [17]. Nejběžnější a nejpoužívanější metodou je jejich zachycení v programovacím jazyce. Tato metoda je snadná, protože programátor může použít stejný jazyk pro popis pravidel stejně jako pro popis celého systému. Bohužel, tato metoda nám nedává příliš možností jak provést inspekci a extrakci pravidel. Další, pokročilejší metodou, je zápis pravidel pomocí meta-instrukcí, například anotací, nebo tzv.*Expression Language* (EL). Tato metoda poskytuje dobrou možnost inspekce, ale zpravidla není typově bezpečná a může snáze způsobovat chyby v programu. Poslední, nejpokročilejší metodou, je zápis pomocí doménově specifických jazyků. Ty jsou snadno srozumitelné nejen pro programátory, ale i pro doménové experty. Nevyžadují inspekci a mohou být typově bezpečné. Mezi jejich nevýhody ale patří vysoká počáteční investice v podobě návrhu takového jazyka a nutnost jeho kompilace nebo interpretace.

2.1.4 Byznysový kontext

Informační systém zpravidla implementuje více byznysových procesů, které se vážou na jeden či více uživatelských scénářů. 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. Ke každé byznysové operaci přísluší množina byznysových pravidel, konkrétně preconditions a post-conditions.

Při běhu informačního systému je v paměti držen tzv. *exekuční kontext* (z anglického *execution context*), který se skládá z několika dílčích kontextů [18]. Prvním je *aplikační kontext* (z anglického *application context*), ve kterém je uložen stav globálncích proměnných systému, jako např. nastavení produkčního režimu, nebo příznak o tom, zda právě probíhá obchodní uzávěrka. Dalším je *uživatelský kontext*, který obsahuje informace o aktuálně přihlášeném uživateli. *Kontext požadavku* (z anglického *Request context*) obsahuje informace o aktuálním požadavku, jako IP adresa uživatele či jeho geolokace, a vztahuje se zejména k webovým službám. Posledním je *byznysový kontext*. Ten chápeme jako množinu preconditions a post-conditions s byznysovou hodnotou, která se váže na konkrétní byzynsovou operaci [17]. Abychom mohli efektivně definovat co nejširší škálu byzynsových pravidel, musejí při jejich vyhodnocování být dostupné proměnné exekučního kontextu,

2.2 Architektura orientovaná na služby

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

[2] Historicky byl termín SOA vykládán různými způsoby a vývojáři si pod ním představovali několik rozdílných, nekompatibilních konceptů [33]. Zejména pak absence kvalitních definic toho, co vlastně služba je, vedla k vzájemnému nedorozumění, zmatení a v poslední době i ke snahám o opuštění tohoto konceptu [19]. Abychom lépe prozuměli tomu, co vlastně

[1] [Intended Delivery: Úvod do SOA, proč je potřeba]
[2] [Intended Delivery: Proč tu vlastně píšu o nějaké historii]

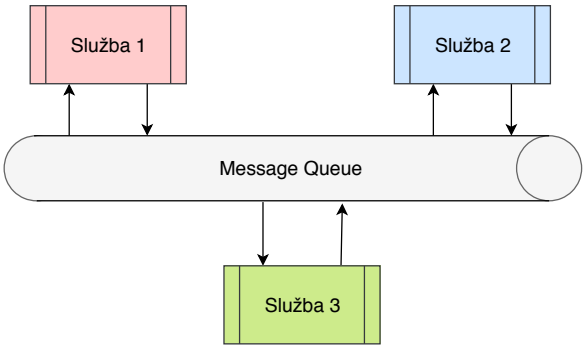
SOA je, popíšeme si její historický vývoj a shrneme výhody a nevýhody jednotlivých přístupů.

2.2.1 Common Object Request Broker Architecture

Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture* (CORBA) [69], která vzikala v osmdesátých a devadesátých letech dvacátého století. Ta umožňuje komunikaci mezi aplikacemi implementovanými v různých technologiích a běžícími na vlastních strojích s rozdílnými operačními systémy. Základním stavebním kamenem této architektury je *Object Request Broker* (ORB), který emuluje objekty, na kterých může klient volat jejich metody. Při zavolání metody na objektu, který se fyzicky nachází v aplikaci na vzdáleném stroji, zprostředkovává ORB veškerou komunikaci a svému uživateli poskytuje jeho kompletní rozhraní. Uživatel tedy de facto nerozezná, kdy volá metodu na objektu, který je lokálně dostupný, a kdy volá metodu, kterou obsouží vzdálená služba. To je ale zároveň hlavní nevýhodou této architektury, protože komunikace se vzdáleným objektem s sebou nese celou řadu problémů, například mnohem vyšší latenci při komunikaci nebo výjimečné stavy, které je potřeba ošetřit. Ve chvíli, kdy klient není schopen rozeznat mezi metodou volanou lokálně či vzdáleně, se těžko přizpůsobuje těmto okolnostem, což vnáší do kódu zbytečnou komplexitu a zhoršuje jeho kvalitu kvůli obtížnější optimalizaci.

2.2.2 Web Services

Nedostatky architektury CORBA vedly k volbě jednoduššího formátu pro popis komunikace služeb, spolehlivějšího a méně komplikovaného kanálu pro komunikaci a celkové redukci objemu komunikovaných dat. Preferovanou cestou komunikace se na přelomu tisíciletí stal protokol HTTP, zatímco preferovaným formátem pro serializaci přenášených dat se stal jazyk XML. Postupně se upustilo od volání metod na vzdálených objektech a přijal se koncept explicitního posílání zpráv mezi službami. Pro popis schématu zpráv vznikl formát *Simple Object Access Protocol* (SOAP) [11], který v kombinaci s *Web Service Description Language* (WSDL) [22] umožňuje kompletní definici rozhraní pro komunikaci mezi službami. V průběhu dalších let vznikla také velmi populární architektura *Representational State Transfer* (REST) [30], která pro popis webových služeb využívá čistě protokol HTTP a jeho slovesa. To službám přináší společný slovník a umožňuje snažší dokumentaci a rychlejší orientaci vývojářů, kteří takovou službu implementují či konzumují. Kvůli těžkopádnosti XML se pro služby implementující REST architekturu stal preferovaným formátem přenosu *JavaScript Object Notation* (JSON). Nejnovějším formátem pro popis služeb, čerpající z nedostatků architektury REST, je *GraphQL*, se kterým v roce 2015 přišla společnost Google.



Obrázek 2.1: Komunikace služeb pomocí Message Queue

2.2.3 Message Queue

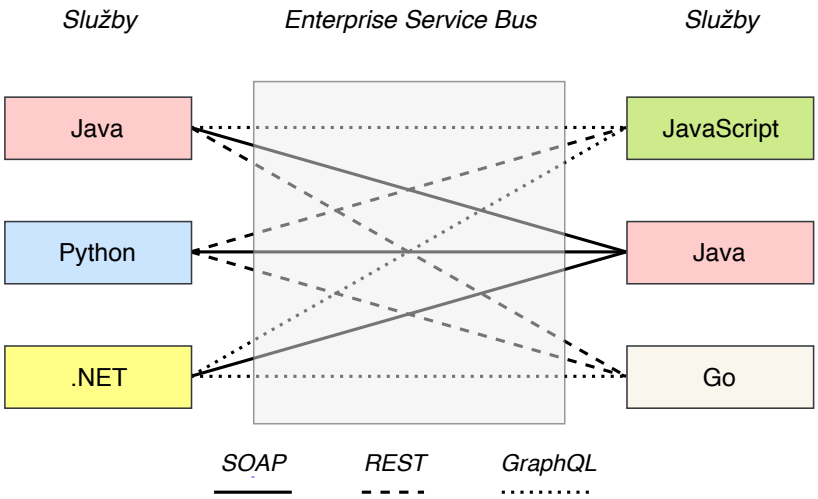
Dalším z konceptů, který v rámci SOA vznikl, je tzv. *Message Queue* (MQ). Základní myšlenkou MQ, znázorněnou na obrázku 2.1, je asynchronní komunikace služeb pomocí zpráv nezávislých na platformě. Komunikaci zprostředkovává fronta, která přijímá a rozesílá zprávy mezi službami. To přináší vyšší škálovatelnost a menší provázanost mezi službami. Všechny služby ale musí používat jednotný formát zpráv.

MQ přináší dva způsoby, kterými mohou služby komunikovat. Prvním je *Request/Reply*, připomínající konverzaci dvou lidí. Jedna služba zašle zprávu obsahující identifikátor konverzace. Druhá služba na obdrženou zprávu zašle odpověď a pomocí identifikátoru označí, ke které otázce odpověď patří. Druhým způsobem je *publish-subscribe*, kdy existuje více front s různými tématy (*topics*) a služby mohou do těchto front přispívat relevantními zprávami nebo je konzumovat jako odběratelé.

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. Již v devadesátých letech minulého století byl představen koncept *Enterprise Service Bus* (ESB) [21], znázorněný na obrázku 2.2, který má za úkol propojit heterogenní služby a zajistit mezi nimi komunikační kanály. Tím na sebe ESB přebírá zodpovědnost za překlad jednotlivých zpráv a centralizuje veškerou komunikaci v systému.

ESB se zároveň staví do role experta na lokalizaci jednotlivých služeb. Službě tak pro komunikaci s okolním světem stačí znát adresu ESB, kterému zašle zprávu, a ten ji sám doručí na místo určení. Tento model ale znamená, že ESB je velmi komplexní komponentou. Výpadek ESB navíc v způsobí zastavení funkce celého systému a ESB se tak stává tzv. *single point of failure*, což v praxi snižuje škálovatelnost systému. V případě vlastního nízkého výkonu se ESB může snadno stát úzkým hrdlem. Tyto problémy mohou být částečně vyřešeny



Obrázek 2.2: Komunikace služeb skrz Enterprise Service Bus

tzv. *federovaným designem*, kdy je systém rozdělen na byznysově příbuzné části, z nichž každá má svůj ESB.

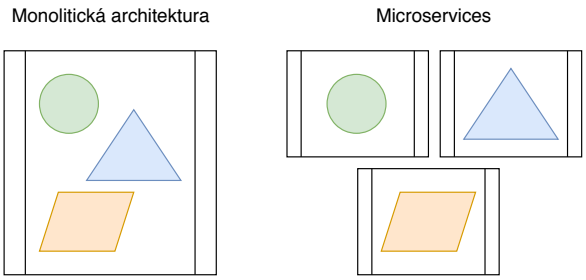
2.2.5 Microservices

[3] Novým trendem posledních let je architektura zvaná *Microservices*. Přináší několik zajmavých konceptů, které specializují a konkretizují principy SOA. Microservices se tedy dají chápat jako podmnožina SOA, ačkoliv existují i názory, že jde o odlišné architektury [67]. Základní myšlenkou je vývoj informačního systému jako množiny malých oddělených služeb, které jsou spouštěny v samostatných procesech a komunikují spolu pomocí jednoduchých protokolů [48].

[4] Důležitou myšlenkou microservices je organizace služeb kolem byznysových schopností systému. Namísto horizontálního dělení systému podle jeho vrstev[5] navrhuje rozdělit systém vertikálně podle jeho byznysových schopností. Na obrázku 2.3 je toto rozdělení demonstrováno. Příkladem může být dělení e-commerce systému na jednu službu obsahující byznysovou logiku pro registraci a správu uživatelů, druhou službu obsahující byznysovou logiku pro práci s produkty a třetí službu obsahující byznysovou logiku pro práci s objednávkami.

[6] Koncept microservices přemýšlí o službě jako o samostatné komponentě, kterou lze individuálně vyměnit či vylepšit, bez nutnosti zásahu do ostatních služeb [48]. Monolitická

[3] [Intended Delivery: Microservices a budoucnost SOA]
[4] [Intended Delivery: Stavba služeb kolem byznysových schopností]
[5] Zde předpokládáme klasickou třívrstvou architekturu [32], rozdělující systém na *datovou vrstvu*, *aplikační vrstvu* a *prezentační vrstvu*. Tyto vrstvy mají oddělené zodpovědnosti a komunikují spolu pomocí jasně definovaných společných rozhraní.
[6] [Intended Delivery: Myšlenka nahraditelnosti komponenty]



Obrázek 2.3: Porovnání struktury monolitické architektury a microservices [48]

architektura vyžaduje i při malé změně jedné části celý systém znovu zkompilovat, sestavit a nasadit. Malé služby sloužící ideálně jedinému byznysovému účelu lze naopak při změně byznysových požadavků snadno nahradit samostatně bez zásahu do zbytku systému. Tím se usnadňuje cyklus nasazení a spuštění nové verze služby.

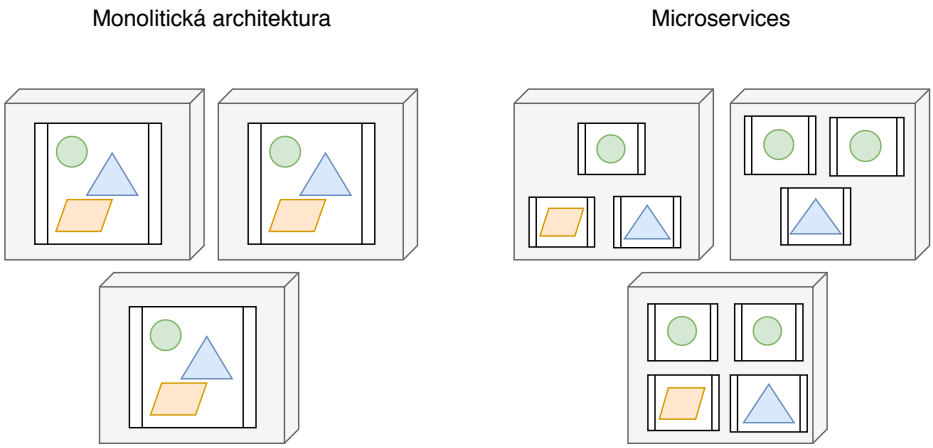
[7] Microservices také přinášejí koncept „smart endpoint, dumb pipes“, který opouští koncept ESB ve prospěch přesunutí veškeré byznys logiky na stranu služeb. Tím se zvyšuje zapouzdřenost služeb a snižuje se jejich vzájemné provázání. Nutno podotknout, že microservices často využívají ke své funkci Message Queues.

Škálovatelnost Další nespornou výhodou microservices je vysoká škálovatelnost systému. Pokud je na některou ze služeb kladen vyšší nárok na výkon než na ostatní, mají vývojáři možnost konkrétní službu horizontálně škálovat aniž by museli škálovat kompletně celý systém, na rozdíl od monolitické architektury. Srovnání přístupů je znázorněno na obrázku 2.4. Díky této vlastnosti je možné snížit nároky na systémové zdroje při zachování stejného výkonu.

Využití rozlišných technologií Monolitické aplikace jsou často implementovány v jednom programovacím jazyce a využívají omezenou množinu technologií. Ne pro každý úkol je ale vhodný jeden programovací jazyk a s rostoucí velikostí informačního systému často roste i rozmanitost jeho funkcionality. Rozdělením systému na více služeb, které komunikují protokolem nezávislým na platformě, je vývojářům umožněno využít širší spektrum technologií a implementovat požadovanou funkcionalitu efektivněji.

Decentralizace úložiště Dalším z principů, které microservices přináší, je oddělení a decentralizace databázového úložiště. Každá služba, či cluster instancí jedné služby, zapisují a čtou data ze své oddělené databáze. Pokud potřebujou načíst data jiné služby, musejí k tomu využít její API. Tím se ještě výrazněji odděluje zodpovědnost služeb. Typickou

[7] [Intended Delivery: Myšlenka smart endpoints, dumb pipes]



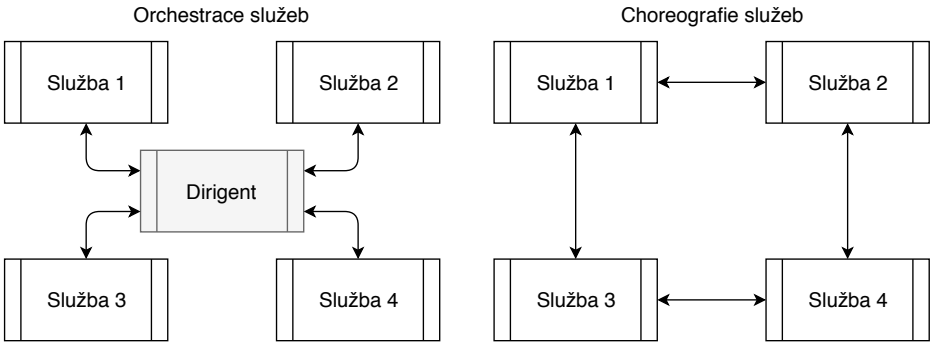
Obrázek 2.4: Porovnání nasazení monolitické architektury a microservices [48]

praktikou v rámci SOA je naopak sdílení jedné databáze mezi více službami, což je často dáno komerčním modelem externího dodatavele databáze. Jediná databáze má navíc obrovskou výhodu v transakčním zpracování, které je centralizované. V případě microservices je nutno transakce řešit distribuovaně, což je velmi náročný úkol a společnosti často volí koncept tzv. *eventual consistency*, kdy je preferována občasná nekonzistence v datech, která je následně manuálně opravena. Tento přístup je opodstatněn tím, že občasná manuální oprava může být často levnější než investice do kvalitního řešení distribuovaných transakcí – zejména pokud by jeho řešení znamenalo zpoždění vývoje produktu a způsobilo by ztrátu obchodní příležitosti [48].

2.2.6 Orchestrace a choreografie služeb

Jak již bylo zmíněno, aby informační systém skládající se ze služeb mohl vykonávat své funkce, musejí spolu služby komunikovat. Aby tato komunikace opravdu vedla ke správné funkci systému, musí podléhat jasně danému řádu.

Orchestrace služeb Pro vykonání byznysové operace je v rámci SOA často potřeba součinnost více služeb najednou. *Orchestrace služeb* má za úkol zajistit, že komunikace mezi službami proběhne úspěšně a ve správném časovém sledu [77], pomocí centrální komponenty – tzv. *dirigenta*. Abychom si mohli tento koncept lépe představit, uvažme následující příklad. Uživatel pomocí UI vytvoří a odešle objednávku. V tuto chvíli je spuštěn byznysový proces, který musí zajistit, že objednávka bude založena v databázi, budou o ní informováni skladníci, bude zažádáno o vytvoření faktury a nakonec bude odeslán potvrzovací e-mail zákazníkovi. Po úspěšném dokončení operace je navíc potřeba uživateli zobrazit v UI informaci, že vše proběhlo v pořádku. V případě orchestrace služba poskytující UI požádá dirigenta o



Obrázek 2.5: Porovnání orchestrace a choreografie služeb [19]

vytvoření objednávky a ten se již postará o komunikaci tohoto požadavku všem službám zapojeným do procesu. Typicky je jako dirigent využíván ESB, který je pro tuto roli vhodný, protože má informace o lokaci jednotlivých služeb a zprostředkovává mezi nimi komunikační kanály.

Choreografie služeb Přímým opakem orchestrace je tzv. *choreografie služeb* a znamená vykonávání byznysových operací autonomně a asynchronně, bez centrální autority. V případě microservices je preferován tento přístup [23], protože orchestrace vede k vyššímu provázání služeb a nerovnoměrnému rozložení zodpovědností v systémů. Porovnání obou přístupů je pro lepší pochopení graficky znázorněno na obrázku 2.5 [60].

2.3 Nedostatky současného přístupu

[8] Jak jsme zjistili v předchozích odstavcích, SOA se zaměřuje zejména na dělení systému na služby a detailně rozebírá formu jejich vzájemné komunikace. Neodpovídá ale na několik závažných otázek, se kterými se v praxi musejí architekti informačních systémů vypořádat, aby architektura byla schopná uspokojivě plnit požadavky, které jsou na ní kladené.

[9] Jelikož jedním z cílů SOA, potažmo microservices, je co nejvíce izolovat jednotlivé služby, mají tyto architektury tendenci duplikovat části kódu zajišťující funkcionalitu, která vyžaduje konzistentní zpracování ve více službách [19], tzv. *průřezových problémů* (z anglického *cross-cutting concerns*). Příkladem mohou být právě byznysová pravidla [15], která je potřeba zohlednit v rámci různých byznysových kontextů realizovaných ve více službách. Mezi další příklady se řadí logování, monitoring či sběr dat o telemetrii procesů.

[8] [Intended Delivery: Navázání na předchozí sekci]
[9] [Intended Delivery: Problémy SOA a průřezových problémů]

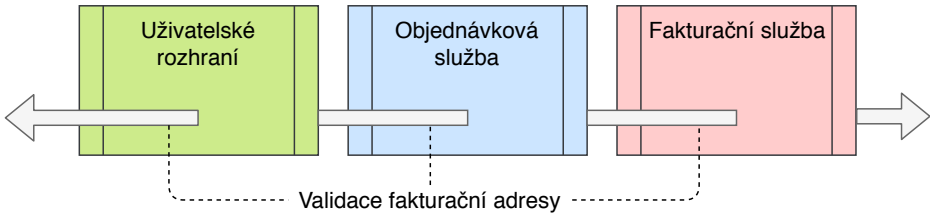
[10] Abychom si mohli lépe představit diskutovaný problém, znázorněmě si ho na konkrétním příkladu. Uvažme e-commerce systém skládající se z několika služeb naprogramovaných v různých technologiích, organizovaných kolem jeho byznysových funkcí. Jedna služba obsluhuje byznysové operace vázající se na uživatele systému, jejich registraci a administraci. Druhá služba realizuje operace s produkty, jejich vytváření, úpravu, správu skladových zásob a informace o dostupnosti. Třetí služba je zodpovědná za vytváření a správu objednávek, informování uživatelů o změnách jejich stavů a vytváření statistik a reportů pro management. Čtvrtá služba má na starosti účetnictví, tedy vystavování a přijímání faktur a komunikaci s bankovními službami o potvrzení přijatých plateb. Poslední, pátá služba, poskytuje uživatelské a umožňuje komfortní obsluhu systému.

[11] Jak již víme, každá byznysová operace má své preconditions, které musejí být splněny, aby mohla být vykonána. Operace má také post-conditions, které musejí být aplikovány po skončení operace. Například při vytváření faktury za objednávku musí být zvalidována fakturační adresa, bez níž nemůže být faktura vystavena. Pokud chceme ušetřit práci účetníkům, kteří by v případě nevalidní adresy musely kontaktovat zákazníka – pokud vůbec takovou možnost mají – musíme tento fakt zohlednit již při vytváření objednávky. Proces vytváření objednávky ale realizuje jiná služba, než vystavování faktur. V ideálním případě bychom chtěli zákazníka upozornit na nevalidní fakturační adresu dynamicky ještě před odesláním objednávkového formuláře přímo v uživatelském rozhraní [18]. Pro lepší představu je problém znázorněn na obrázku 2.6,

[12] Z tohoto příkladu je jasně vidět, že stejná funkcionalita se promítá do tří služeb, z nichž každá má zodpovědnost za jiné byznysové operace. To znamená, že stejný kód, který realizuje validaci fakturační adresy, musí být implementován v každé ze služeb – v našem případě navíc ve třech různých programovacích jazycích. Ve chvíli, kdy vzejde požadavek na změnu validace fakturační adresy – řekněme, že chceme zobecnit validaci PSČ a umožníme přijímat i jeho tvar s mezerou – musíme stejnou změnu provést konzistentně na třech různých místech, všechny tři služby znovu sestavit a nasadit ve správném pořadí tak, aby nedošlo ke stavu, kdy jedna služba přijme nový tvar PSČ, ale navazující služba ho není schopna zpracovat.

[13] Pozorný čtenář může namítnout, že problém validace fakturačních adres by bylo možné vyřešit vyčleněním této funkcionality do samostatné služby a vystavit její rozhraní pro ostatní služby, v souladu s nosnou myšlenkou microservices. Je pravda, že microservices v názvu nese slovo „micro“ a evokuje tak, že služby by měly být co nejmenší a nést co nejméně zodpovědnosti. Může ale nastat stav, kdy je služba příliš malá? Pokud služby ponesou příliš

[10] [Intended Delivery: Nastínění konkrétního příkladu]
[11] [Intended Delivery: Konkrétní problémy zpracování průřezových problémů na příkladu]
[12] [Intended Delivery: Náročná údržba a reakce na změnu požadavku]
[13] [Intended Delivery: Microservices neříká nic o tom, jak velké je mikro]



Obrázek 2.6: Příklad zásahu jedné funkcionality do více služeb

málo odpovědnosti, přináší to s sebou několik problémů, které je nutné zvážit. Musíme mít na paměti, že nasazení a provoz každé služby s sebou přináší náklady navíc a zvyšuje časové nároky na jejich vývojáře a administrátory. Komunikace služeb po síti je navíc podstatně pomalejší a náchylnější na chybu, než komunikace jednotlivých komponent v rámci jednoho procesu. S rostoucím počtem *průřezových problémů* by tak i rychle rostl počet služeb v systému a celkové náklady na jeho vývoj a údržbu.

[14] Na příkladu můžeme vidět, že existuje typ problémů, které v rámci architektury orientované na služby při využití současného přístupu nejsme schopni uspokojivě vyřešit na jednom místě a vedou k duplikaci znalostí na více místech systému. Taková duplikace může vést k zvýšenému riziku lidské chyby vývojáře a tím k nekonzistentnímu chování systému. Navíc zvyšuje cenu na vývoj a údržbu systému.

2.4 Identifikace požadavků na implementaci frameworku

Z příkladu popsaného výše můžeme identifikovat požadavky, které by měly být zohledněny při návrhu a implementaci frameworku, který bude sloužit pro centrální administraci a automatickou distribuci byznysových pravidel v architektuře orientované na služby.

Framework, resp. jeho knihovny, by měly umožňovat:

- Definice byznys kontextů pomocí platformně nezávislého doménově specifického jazyka srozumitelného pro doménové experty
- Zápis preconditions a post-conditions pravidla jednotlivých byznys kontextů
- Možnost jednoho kontextu rozšiřovat jiné kontexty
- Možnost centrálně spravovat byznysové kontexty, včetně úpravy stávajících a vytváření nových kontextů, to vše dynamicky za běhu systému
- Automatickou distribuci kontextů, vyhodnocování jejich preconditions a aplikaci post-conditions

[14] [Intended Delivery: Shrnutí problémů]

- Možnost využívat framework na více plaformách

2.5 Shrnutí

V této kapitole jsme nastínili problematiku vysoké komplexity moderních informačních systémů a z toho vyplývající požadavky na jejich architekturu. Analyzovali jsme koncept byznysových pravidel a byznysových kontextů. Dále jsme prozkoumali architekturu orientovanou na služby, její výhody a nevýhody, její moderní evoluci v podobě microservices a identifikovali jsme nedostatky současných přístupů v řešení průřezových problémů, které zasahují do více služeb najednou. Nakonec jsme vyjmenovali požadavky, které by měl splňovat framework, jež bude výstupem této práce.

Kapitola 3

Rešerše

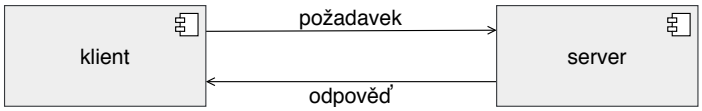
V této kapitole se budeme věnovat rešerši existujících řešení a výzkumu relevantního k tématu této práce. Prozkoumáme modelem řízenou architekturu, její výhody, nevýhody a vhodnost použití. Věnovat se budeme také síťovým architekturám umožňujícím komunikaci služeb v rámci architektury SOA. Dále se zaměříme na aspektově orientované programování a na něm založený inovativní přístup k návrhu iformačních systémů ADDA. Nakonec prozkoumáme existující nástroje a specializované jazyky pro zápis byznysových pravidel.

3.1 Modelem řízená architektura

Modelem řízená architektura (MDA z anglického *Model-Driven Architecture*) se zaměřuje na návrh IS pomocí modelů a jejich následnou transformaci do spustitelného kódu pomocí generativních nástrojů [70]. Autorem specifikací MDA je konsorcium Object Modeling Group (OMG), které se zaměřuje na standardizaci modelovacích standardů pro software a stojí za modelovacím jazykem UML, který je de facto globálním standardem pro vizualizaci statických i dynamických aspektů softwarových systémů.

MDA využívá Platform Independent Model (PIM), který popisuje systém pomocí platformově nezávislého DSL, typicky pomocí UML. Ten je následně převeden do Platform Specific Model (PSM), tedy modelu využívajícího specifických aspektů platformy, pro kterou má být systém postaven. Tím může být obecný programovací jazyk či jiné DSL. Nakonec je PSM transformován do spustitelného kódu.

Hlavní výhodou MDA je vysoká úroveň abstrakce a z toho vyplívající deduplikace kódu. Pokud je byznysové pravidlo zachyceno v nejvyšší úrovni modelu, může pak snadno být distribuováno do systému na všechna místa, kde má být aplikováno. Další výhodou je usnadnění tvorby a zvýšení kvality kódu díky jeho automatickému generování.



Obrázek 3.1: Architektura klient-server

Hlavní nevýhodou MDA, která zabraňuje jejímu využití pro náš účel, je jednosměrný dopředný proces, kterým je výsledný kód z modelu generován. Pokud dojde ke změně požadavků, je potřeba přegenerovat celou aplikaci od PIM až k finálnímu kódu. Kód, který bylo nutno doplnit ručně, může snadno zastarat a je potřeba ho manuálně projít a opravit. Změna byznysových pravidel v našem případě by navíc při využití MDA vyžadovala znovu-nasazení celého systému, což je v přímém rozporu s požadavkem na možnost dynamicky upravovat či přidávat byznysová pravidla za běhu systému.

Další nevýhodou tohoto přístupu je jeho závislost na OOP, které samotné není schopné se efektivně vypořádat s průřezovými problémy [15], jak si popíšeme v sekci 3.3. Ačkoliv je MDA známá již od roku 2000, generativní nástroje pro její podporu jsou stále nevyspělé a pro některé platformy úplně chybí.

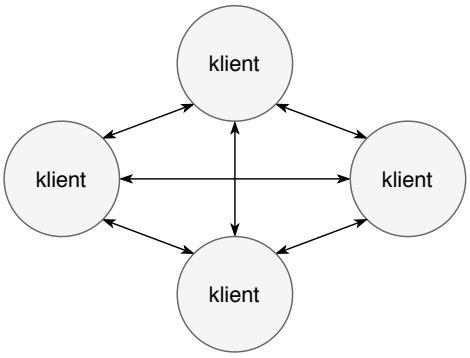
3.2 Síťové architektury

Jak jsme již naznačili v sekci 2.2, architektura orientovaná na služby staví na počítačových sítích, díky kterým spolu mohou jednotlivé služby efektivně komunikovat. Proto je vhodné provést rešerši základních síťových architektur, abychom mohli co nejlépe navrhnout framework pro centrální správu a automatickou distribuci byznysových pravidel.

3.2.1 Architektura klient-server

Model klient-server popisuje vztah mezi komponentami systému, klienty a serverem. Klient zašle požadavek serveru a ten mu vrátí odpověď [8]. Tento model může být použit obecně i v rámci jednoho počítače, nejčastěji je však využíván v síťové komunikaci mezi více počítači. V tomto případě klient sestaví síťové spojení k serveru a po získání odpovědi od serveru spojení zase uzavře. Schéma komunikace je znázorněno na obrázku 3.1.

Tato architektura je jednou ze základních stavebních kamenů internetových protokolů. Využívá ji zejména protokol TCP [63], který je hlavním komunikačním protokolem v síti Internet. Jako příklad si můžeme představit prohlížení internetových stránek. Uživatel zadá URL adresu stránky, kterou chce navštívit, a internetový prohlížeč, potažmo uživatelův osobní počítač, v roli klienta odešle požadavek na server nacházející se na dané adrese. Server požadavek přijme, zpracuje, a odešle odpověď obsahující tělo webové stránky. Klient stránku přijme a zobrazí pro koncového uživatele.



Obrázek 3.2: Architektura peer-to-peer

Tento přístup má několik zásadních výhod, díky kterým se stal široce využívaným. Díky svojí velmi obecné myšlence je nezávislý na jakékoliv platformě a jako klient i server mohou sloužit jak vysoce výkonné počítače, tak i osobní počítače nebo chytré telefony, z nichž každý může využívat odlišné operační systémy – stačí aby klient i server uměl komunikovat stejným protokolem. 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.

Pro sdílení byznysových pravidel se tato architektura jeví jako vhodná. Klient, který potřebuje byznysové pravidlo ke svém funkci, by zažádal server o dané pravidlo a po jeho získání by se postaral o jeho spuštění. Tím by bylo dosaženo automatické distribuce a integrace byznysových pravidel. Vzhledem k tomu, že server je v této architetkuře centrální autoritou, usnadnila by se tak centrální správa byznysových pravidel.

3.2.2 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. [36] Na obrázku 3.2 je tato architektura znázorněna. Hlavním cílem P2P sítí je distribuce dat nebo výpočetních operací.

Jednotliví klienti mezi sebou zpravidla vytvářejí virtuální síť, tzv. *overlay*, která je postavená nad fyzickou sítí, přes kterou jsou reálně fyzicky zasílány zprávy mezi klienty. Typicky je

tato virtuální síť podmnožinou existující fyzické sítě. Výhodou tohoto přístupu je, že klienti jsou abstrahováni od fyzického uspořádání počítačů a mohou spolu komunikovat napřímo, i když mezi nimi mohou reálně být v síti zapojeny jiné počítače.

Nespornou výhodou architektury P2P je, že s rostoucím počtem klientů roste i kapacita a výkon sítě, narozdíl od modelu klient-server, kdy se klienti musí dělit o výkon serveru. Navíc v takové síti neexistuje *single point of failure* a tak se zvyšuje její robustnost.

Mezi silné nevýhody této architektury patří zvýšená bezpečnostní rizika způsobená tím, že klienti jsou otevřeni komunikaci s jakýmkoliv jiným, potenciálně nebezpečným, klientem. Potenciální útočník tak může velmi snadno využít zranitelností na dálku. Další hrozbou je to, že některý z klientů může *otrávit* síť (z anglického *network poisoning*) podvrženými daty, která jsou pak nekontrolovatelně šířena mezi všechny klienty. Z toho vyplývající nevýhodou může být absence jakékoliv centrální správy sdílených dat.

Ačkoliv se může P2P jevit jako vhodný přístup pro sdílení byznysových pravidel díky svojí orientaci na sdílení dat, vysoké datové propustnosti a robustnosti, právě absence centrální správy by mohla uvrhnout systém do nekonzistentního stavu při úpravě či přidání byznysového pravidla. Tato architektura totiž implikuje, že by byznysové pravidlo bylo replikováno a distribuováno mezi více uzlů sítě. Změna pravidla by se tak musela šířit postupně napříč systémem, přičemž některé uzly by stále využívaly starou verzi pravidla, a nad samotným šířením by neměl administrátor systému kontrolu.

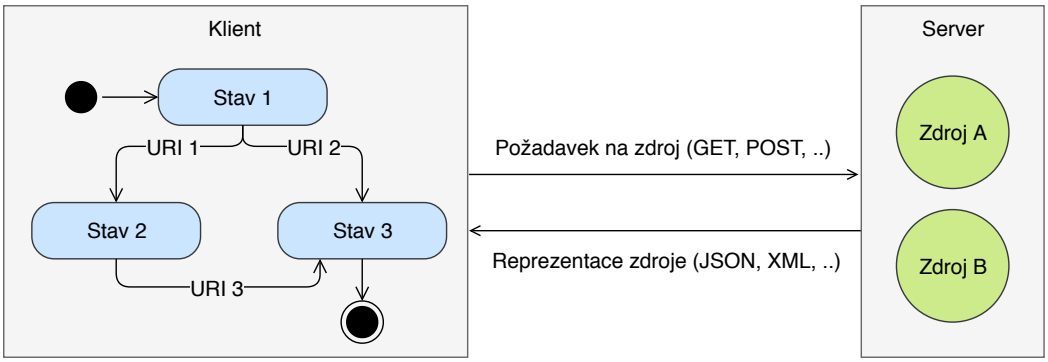
3.2.3 Representational state transfer

Representational state transfer [30] (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 díky jednotnému rozhraní a lepší odolnosti vůči chybám. V roce 2000 ho ve své dizertační práci představil Roy Fielding, který je zároveň hlavním autorem specifikace HTTP 1.1 [29].

REST chápe data systému jako množinu zdrojů (z anglického *resources*), nad kterými jsou prováděny operace pomocí HTTP požadavků. K odlišení operací nad jedním zdrojem jsou využívána slovesa protokolu HTTP, zejména pak `GET` pro čtení, `POST` pro vytváření, `PUT` pro úpravu a `DELETE` pro mazání. Tím jsou zastřešeny všechny CRUD operace.

Principy architektury REST jsou:

Klient-server Systém by měl využívat model klient-server. Díky tomu může jasně oddělit zodpovědnost za uživatelské rozhraní na klienta a zodpovědnost za ukládání dat na server. To zvyšuje škálovatelnost systému díky nižším nárokům na server.



Obrázek 3.3: Znázornění architektury REST

Bezstavovost Každý požadavek na server by měl obsahovat všechny informace potřebné k jeho vykonání. Kromě těla HTTP požadavku se k tomuto účelu často využívají i hlavičky požadavku, například hlavička `Authorization` pro autentizaci uživatele kvůli přístupu k zabezpečeným zdrojům. Tím se zjednodušuje komplexita serveru, který plně přesouvá zodpovědnost za uchování stavu uživatelského rozhraní na klienta, jak je znázorněno na obrázku 3.3. Nutno poznamenat, že stav dat v systému je stále uchováván na serveru.

Kešování Odpovědi serveru musí obsahovat explicitní informaci o tom, zda lze odpověď uložit do cache. Díky tomu je možné znovupoužívat data, která již server jednou vrátil, a jejich životnost má dlouhodobější charakter. Tím se zvyšuje výkon celého systému.

Vrstvený systém Klient by neměl mít možnost rozeznat, zda komunikuje přímo se serverem, nebo s prostředníkem, např. s proxy serverem, load balancerem nebo cache.

Code-on-demand Volitelným požadavkem na systém je tzv. *code-on-demand*, který umožňuje serveru vracet spustitelný kód jako odpověď. Klient kód poté spustí na své straně. Typickým příkladem jsou klientské JavaScriptové aplikace spouštěné ve webových prohlížečích.

Jednotné rozhraní Zdroje systému musejí mít unikátní identifikátor, např. URI. Zdroje jsou při komunikaci reprezentovány libovolným formátem, který se může lišit od interní reprezentace zdroje v programu, např. JSON či XML. Reprezentace zdroje musí být dostatečná k tomu, aby šlo na zdroji provést úpravu či smazání. Zprávy mezi klientem a serverem musejí obsahovat veškerá potřebná metadata, aby druhá strana mohla zprávu plně zpracovat. K tomu se používají například HTTP hlavičky `Content-type` či `Accept`, ve kterých je popsán typ reprezentace zdroje. Rozhraní by také mělo dodržovat koncept *Hypermedia as the engine ot application state (HATEOAS)*, který vyžaduje, aby server v odpovědích vracel metainformace o struktuře jeho API. Klient je tak

schopen dynamicky navigovat v rozhraní serveru aniž by bylo vyžadováno předem znát přesné adresy zdrojů. Princip HATEOAS se však ve většině reálných API zanedbává či implementuje pouze částečně.

Nevýhody architektury REST spočívají zejména v náročnosti její korektní implementace, která se v praxi často zjednodušuje, a tím jsou degradovány její výhody. Přes rošířenost a popularitu této architektury ji stále mnoho vývojářů nezná do detailu a zanedbává některé její části. Většina programovacích jazyků je založena na volání funkcí či metod, což svádí vývojáře ke špatnému návrhu systému. REST totiž naopak vyžaduje od vývojáře nad systémem přemýšlet jako nad množinou zdrojů. Jinou nespornou nevýhodou 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 [61], umožňující zajistit atomické transakce nad REST architekturou. Jak jsme již zmínili v sekci 2.2.5, moderní systémy se přiklánějí ke konceptu *eventual consistency*, kdy jsou tolerovány drobné nekonzistence, které jsou nakonec dořešeny manuálně, pokud je cena za ně menší než cena za implementaci systému, který by plně podporoval atomické transakce.

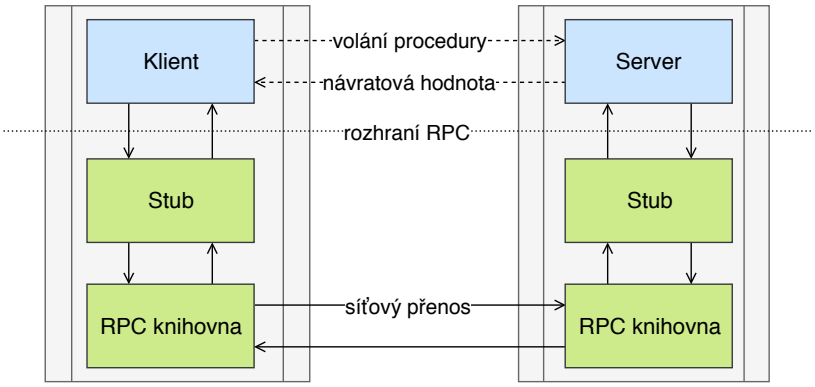
3.2.4 Remote procedure call

Remote procedure call (RPC) je podstatně starší architekturou než REST. Tento termín byl použit již v roce 1981 Brucem Nelsonem [59]. Architektura staví na modelu klient-server a její princip je velmi jednoduchý – umožňuje jednomu procesu, klientovi, zavolat proceduru na druhém, vzdáleném procesu, tedy serveru. Klient zašle serveru zprávu vyžadující zavolání specifické procedury. Server proceduru provede a po jejím dokončení zašle klientovi odpověď s návratovou hodnotou. Klient poté může pokračovat ve své práci.

Zásadním bodem je fakt, že RPC kompletně obstarává komunikaci, a v programu samotném je vzdálená procedura volána stejným způsobem jako lokální procedury. Základním prvkem architektury na klientovi i na serveru je tzv. *stub*, tedy komponenta, která umožňuje volat, resp. obsloužit, vzdálenou proceduru lokálně a zapozdřuje veškerou síťovou komunikaci a serializaci či deserializaci argumentů, resp. návratových hodnot. Schéma komunikace je znázorněno na obrázku 3.4.

Představitelem architektury RPC je například technologie CORBA, kterou jsme již popsali v sekci 2.2.1, nebo technologie Remote Method Invocation (RMI) v jazyce Java. Modernějším pojetím je technologie gRPC [39] od společnosti Google[1], která je v dnešní době hojně využívána úspěšnými technologickými společnostmi.

[1] https://www.google.com/



Obrázek 3.4: Schéma komunikace RPC

Již zmiňovanou nevýhodou abstrakce lokálních a vzdálených volání jsou negativní vlastnosti síťové komunikace, tedy její zvýšená latence a nižší robustnost. Pokud programátor nemá možnost zjistit, zda volá lokální či vzdálenou proceduru, výsledný kód může být těžké optimalizovat a správně ošetřit výjimky, které mohou při jeho běhu nastat.

Na druhou stranu, narozdíl od REST není pro RPC potřeba implementovat komplexní middleware obstarávající síťovou komunikaci, serializaci a zpracování chyb. Middleware je zpravidla dodáván v podobě knihoven dané technologie. Exsitují příklady funkcionality, která může být mnohem přirozeněji realizována pomocí volání procedur, než jako entita reprezentující systémový zdroj.

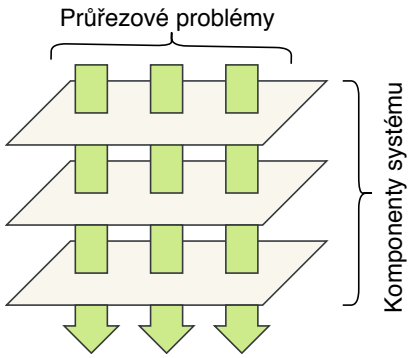
Vhodnost RPC architektury pro distribuci a správu byznysových pravidel je do značné míry ovlivněna tím, jakou konkrétní technologii využijeme. Stejně jako REST ani RPC nedefinuje jakým způsobem by měly být obslouženy transakce. Byli bychom tedy nuceni si transakční systém doimplementovat. To by však mohlo být o něco snažší díky tomu, že by obsluha transakce mohla být volána jako procedura, a tím bychom si ušetřili práci s obslužným kódem.

3.3 Aspektově orientované programování

3.3.1 Motivace

[2] Programování je komplexní disciplína s teoreticky neomezeným počtem možností, jakým programátor může řešit zadaný problém. Ačkoliv každá úloha má své specifické požadavky, za relativně krátkou historii programování se stihlo ustálit několik ideologií, tzv. programovacích paradigmat, které programátorovi poskytují sadu abstrakcí a základních principů [78]. Díky znalosti paradigmatu může programátor nejen zlepšit svou produktivitu,

[2] [Intended Delivery: Co je paradigma]



Obrázek 3.5: Průřezové problémy v informačních systémech

ale zároveň může snáze pochopit myšlenky jiného programátora a tím zlepšit kvalitu týmové spolupráce.

[3] Jedním z nejpopulárnějších paradigmat používaných k vývoji moderních enterprise systémů je nepochybně objektově orientované programování (OOP). To vnímá daný problém jako množinu objektu, které spolu intereagují. Program člení na malé funkční celky odpovídající struktuře reálného světa [66]. Je vhodné zmínit, že objekty se rozumí jak konkrétní koncepty, například auto nebo člověk, tak i abstraktní koncepty, namátkou bankovní transakce nebo objednávka v obchodě. Objekty se pak promítají do kódu programu i do reprezentace struktur v paměti počítače. Tento přístup je velmi snadný pro pochopení, vede k lepšímu návrhu a organizaci programu a snižuje tak náklady na jeho vývoj a údržbu.

[4] Ačkoliv je OOP velmi silným a všestranným nástrojem, existují problémy, které nelze s jeho pomocí efektivně řešit. Jedním takovým problémem jsou obecné požadavky na systém, které musejí být konzistentně dodržovány na více místech systému, které spolu zdánlivě nesouvisí. Takové požadavky nazýváme *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 [34]. Obrázek 3.5 znázorňuje vzájemné postavení průřezových problémů a komponent informačního systému.

[5] Příkladem průřezového problému může být logování systémových akcí, optimalizace správy paměti nebo jednotné zpracování výjimek [46], ale i aplikace byznysových pravidel [15]. Uvážme e-commerce systém a zpracování transakcí. To musí být zohledněno jak při vytváření objednávek, tak při registraci nového uživatele – dvě byznysové akce, které by měly být podle OOP implementovány v naprosto odlišných objektech a striktně odděleny, ale část jejich funkcionality je identická, a tudíž dochází k duplikaci kódu.

[3] [Intended Delivery: OOP a jeho popis]
[4] [Intended Delivery: Nedostatky OOP]
[5] [Intended Delivery: Konkrétní příklad nedostatku OOP]

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
    }
}
```

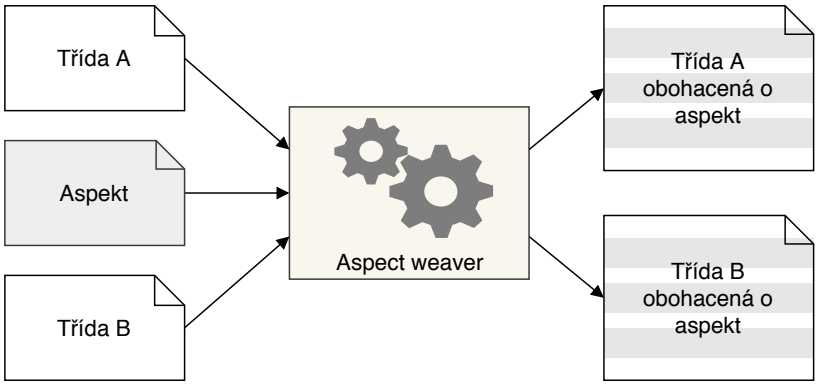
Ve zdrojovém kódu 3.1 můžeme vidět, jak průřezové problémy zasahují do kódu imaginární třídy implementované v jazyce Java, která slouží pro vytváření objednávek v e-commerce systému popsaném v sekci 2.3. Můžeme si všimnout, že aspekt logování je zohledněn na třech místech, stejně jako aspekt transakcí. Navíc jsou zde zohledněna i byznysová pravidla pro validaci doručovací a fakturační adresy objednávky.

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

3.3.2 Názvosloví

Aspekt Jak jsme již naznačili, základním pojmem v rámci AOP je *aspekt*, který zapozdřuje průřezovou funkcionalitu a zároveň adresuje místa, kde má být funkcionalita aplikována.

[6] [Intended Delivery: AOP jako odpověď na nedostatky OOP]



Obrázek 3.6: Proces weavingu aspektů

Aspekt vždy obsahuje alespoň jeden *advice* a jeden *pointcut*.

Join-point 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.

Pointcut Ne každý aspekt je aplikován na každý join-point. 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. Dynamické určení pointcutu ale umožňuje vývojářům postihnout i případy, kdy nelze předem jasně určit místa, kde má být aspekt začleněn. Příkladem může být zpracování transakcí v zanořených byznysových operacích, kdy transakce má být započata pouze při vstupu do vnější operace a dokončena pouze při výstupu z vnější operace.

Advice 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.

Weaving 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 3.6. Komponenta zodpovědná za weaving se nazývá *aspect weaver*.

3.4 Aspect-driven Design Approach

Alternativním způsobem návrhu informačních systémů, který staví na principech AOP, je Aspect-driven Design Approach[7] (ADDA) [15], představený Karlem Čemusem v roce 2014. Tento přístup se zaměřuje na formalizaci jednotlivých komponent informačních systémů identifikování aspektů v informačních systémech a jejich separaci do jednoho bodu, tzv. *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.

3.4.1 Možnosti aplikace

Autoři ADDA aplikovali tento koncept v několika oblastech IS. 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].

Automatické začleňování byznysových pravidel do datové vrstvy Jednou z možných aplikací přístupu ADDA je automatické začleňování byznysových pravidel do datové vrstvy IS[8]. 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.

Automatické generování uživatelského rozhraní Uživatelská rozhraní tvoří až 48 % kódu informačních systému a zabírají až 50 % jejich vývojového času [45]. Do UI se přitom typicky promítá mnoho aspektů, které jsou již v systému obsaženy, a vývojáři je musí manuálně duplikovat. Typickým příkladem je struktura datového modelu, která se promítá zejména do struktury formulářů sloužících pro manipulaci s daty systému. Byznysová pravidla jsou promítána do UI při validaci vstupních dat formulářů na straně klienta [18]. Dalšími příklady může být lokalizace UI do různých jazyků nebo rozložení a stylizace jednolivých ovládacích prvků, která je zpravidla uniformní v celém systému.

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

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

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

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 a tím jsou ušetřeny náklady na vývoj a údržbu informačního systému využívajícího tento přístup.

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

3.4.2 Výhody a nevýhody

Jak můžeme pozorovat z uvedených příkladů, 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.

Je nutno poznamenat, že tento přístup má svoje úskalí v podobě vysoké počáteční investice, kterou vyžaduje vývoj 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.

Pro náš účel se však jeví přístup ADDA jako vysoce vhodný, protože rezonuje s cíly této práce. Přístup navíc 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 prozkoumáme v následující sekci.

3.5 Stávající řešení reprezentace business pravidel

V rámci této kapitoly se zaměřujeme 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 neposkytuje podporu velkého množství programovacích jazyků, resp. platforem, ve kterých by mohl být jazyk použit. Namátkou můžeme zmínit projekt *business-ruless* pro jazyk Python [13], projekt FlexRule pro platformy .NET a JavaScript nebo BRMS JRules [40] od společnosti IBM pro platformu Java EE.

V této sekci se tedy zaměříme zejména na framework Drools, který používají autoři přístupu ADDA, a také na moderní nástroj JetBrains MPS, který umožňuje vytvářet DSL a transformovat je do libovolných víceúčelových jazyků.

3.5.1 Drools DSL

Framework Drools [24] vyvijený společností JBoss[9] je open-source projekt realizující *business rule management engine* (BRMS), tedy nástroj pro vývoj a správu byznysových pravidel. Framework umožňuje realizovat tzv. *produkční systémy*, tedy systémy tvořené sadou *produkčních pravidel* určujících chování programu. Tato pravidla obsahují popis situace a její řešení v případě, že nastane. Tyto systémy tedy poskytují určitou formu umělé inteligence, která simuluje rozhodování experta na danou doménu.

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 správnou funkci systému je nutno při vyhodnocování správně určit, která produkční pravidla mají být aplikována. Pro tento účel využívá framework Drools algoritmus RETE [31], vynalezený Charlesem Forgym v roce 1983, který je přímo navržený pro párování pravidel produkčních systémů. Využívá stromové paměťové struktury, která minimalizuje výpočetní složitost na úkor paměťové složitosti. Framework Drools navíc implementuje některá vylepšení algoritmu optimalizující jeho paměťovou složitost.

Součástí frameworku Drools je speciální doménově specifický jazyk vyvinutý přímo pro modelování produkčních pravidel. Tento jazyk umožňuje popsat LHS i RHS daného pravidla a k tomu využívá několik užitečných konstruktů. Pro popis situací i důsledků využívá dialekt MVEL umožňující komfortní zápis logických výrazů. V rámci Drools DSL lze využívat lokální i globální proměnné s plnou typovou podporu pramenící z jazyka Java a také podporu regulárních výrazů. Navíc je možno importovat i pomocné funkce, které lze využít v podmínkách pravidla.

Ve zdrojovém kódu 3.2 můžeme vidět příklad zápisu byznysového pravidla v jazyce Drools DSL. Kromě názvu pravidla je v hlavičce uvedeno, které dialekty jsou v pravidle využity. Dialekt `mvel` jsme již popsali výše a dialekt `java` nám umožňuje pro RHS využít přímo jazyk Java. Popsané produkční pravidlo vypíše uživatelovo jméno a email, pokud má uživatel email vyplněný.

Ačkoliv je jazyk Drools DSL vymodelovaný přímo pro zápis pravidel doménovými experty, odlišnost produkčních pravidel a byznysových pravidel tak, jak jsme si je zavedli v sekci 2.1, je na první pohled zřejmá. Pro naše účely bychom využili pouze LHS, protože nám

[9] http://www.jboss.org/

Zdrojový kód 3.2: Ukázka zápisu byznysového pravidla v jazyce Drools DSL

```
rule "print user email"

dialect "mvel"
dialect "java"

when
    $u : User( email != null )
then
    System.out.println($u.name + ": " + $u.email);
end
```

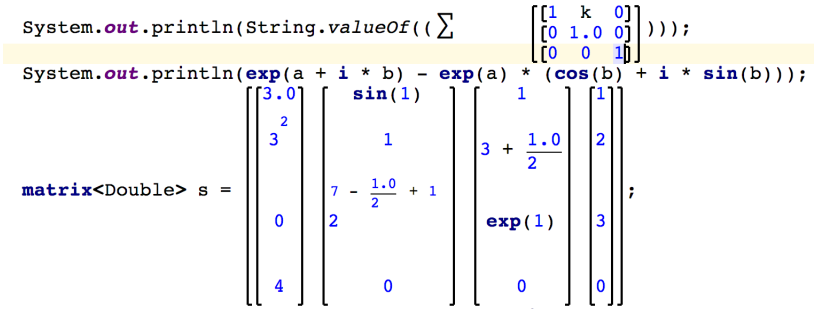
stačí popsat nastalou situaci, ale o následnou akci se postará systém – v případě neúspěšné validace precondition je probíhající byznysová operace zastavena, v případě post-condition je aplikován filtr na všechny proměnné splňující danou podmínku. Zároveň jazyk Drools DSL postrádá nástroje pro kvalitní popis byznysového kontextu držícího byznysová pravidla, zejména pak rozšiřování jiných kontextů a popis typu jednotlivých pravidel [16]. V neposlední řadě nejsou ze strany frameworku Drools podporovány jiné platformy než Java a .NET, což nevyhovuje našim požadavkům na platformovou nezávislost.

3.5.2 JetBrains MPS

Zajímavým nástrojem z dílny společnosti JetBrains[10] je tzv. *MPS – Meta Programming System* [58], který si klade za úkol být univerzálním nástrojem pro tvorbu doménově specifických jazyků. Staví na konceptu *language-oriented programming* (LOP) [80] zaměřujícího se na vývoj velmi specifického jazyka, který je následně použit pro implementaci programu namísto obecného mnohoúčelového jazyka. 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 validace. MPS podporuje také transformování kódu napsaného v nadefinovaném jazyce do jiných, nízkoúrovňovějších jazyků, zejména pak do jazyka Java. Díky tomu lze nejen vytvářet libovolné DSL, ale také rozšiřovat existující jazyky – například

[10] https://www.jetbrains.com/



Obrázek 3.7: Rozšíření jazyka java o zápis matic pomocí JetBrains MPS [58]

přidat do jazyka Java podporu pro jednoduchý vizuální zápis matic, jak můžeme vidět na obrázku 3.7.

Výhoda tohoto přístupu je podobně jako u MDA vysoká úroveň abstrakce, která velmi dobře odděluje problém návrhu systému od implementačních problémů. Navíc lze díky použití DSL zapojit do vývoje doménové experty a snížit tak zátěž na programátory, kteří se mohou více věnovat právě implementačním problémům. DSL typicky zvyšuje expresivitu kódu a díky tomu se zmenšuje jeho objem. Nižší objem kódu vede ke snížení nákladů na jeho údržbu a vývoj [51][72]. Významnou výhodou MPS, potažmo LOP, je vysoká portabilita vyvinutého jazyka. Pro migraci na jinou platformu stačí doprogramovat překladač jazyka, ale samotný jazyk může zůstat zachovaný.

Z výčtu výhod plyne, že bychom mohli MPS využít k popisu byznysových pravidel, resp. byznysových kontextů. Tím bychom získali silný aparát k zachycení a znovupoužití pravidel a jejich transformaci do neomezeného počtu jazyků pro využití na mnoha platformách. Podobně jako u MDA je však problém v dopředném generování – pro případ úpravy pravidla za běhu programu bychom museli nějakým způsobem upravované pravidlo přetransformovat z dané platformy zpět do našeho DSL, upravit ho pomocí editoru MPS, a následně ho opět přeložit do spustitelného jazyka pro danou platformu. To by vyžadovalo potřebu překompilovat a znovu nasadit všechny služby využívající dané pravidlo.

3.6 Shrnutí

V této kapitole jsme provedli rešerši *modelem řízené architektury*, jejích výhod a nevýhod. Prozkoumali jsme síťové architektury, které mohou být využity pro komunikaci služeb v architektuře SOA a zvážili vhodnost jejich použití pro účely této práce. Shrnuli jsme paradigma *aspektově orientovaného programování* a věnovali se inovativnímu přístupu k návrhu softwarových systémů *ADDA*. Nakonec jsme provedli rešerši stávajících řešení reprezentace byznys pravidel včetně komplexního frameworku *Drools* a zhodnotili jsme jeho vhodnost k řešení našeho problému.

Kapitola 4

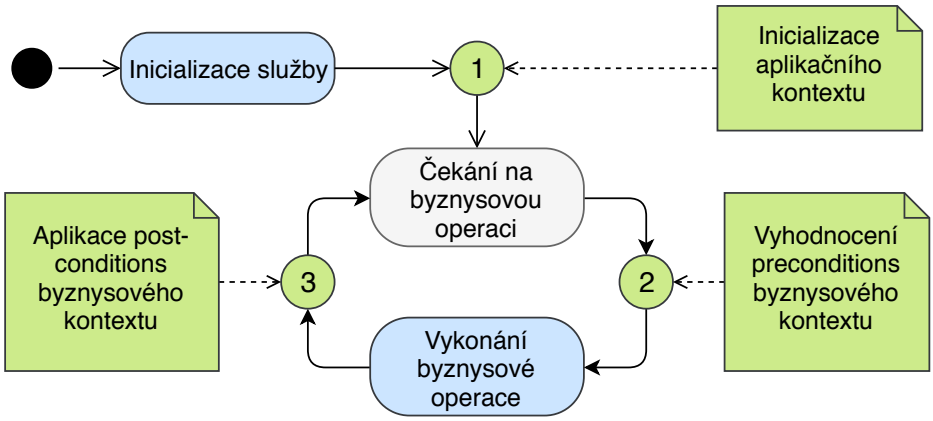
Návrh

V této kapitole budeme diskutovat návrh frameworku pro centrální správu a automatickou distribuci business pravidel vyhovující požadavkům identifikovaným v sekci 2.4. V předchozí kapitole 3 jsme prozkoumali architektury, které bychom mohli při návrhu využít, a shrnuli jsme jejich výhody a nevýhody. Došli jsme k závěru, že alternativní přístup ADDA nám poskytne nejlepší aparát pro dosažení vytyčených cílů. Abychom ho mohli plně využít, je nejprve potřeba formalizovat prostředí SOA v rámci AOP.

V rámci této kapitoly navrhneme vhodný způsob zachycení byznysových pravidel, jejich uložení a organizaci v rámci systému. Bude potřeba vymyslet proces, jakým budou pravidla automaticky distribuována. To bude vyžadovat vytvoření metamodelu, tedy struktury, která bude sloužit k zachycení pravidel v paměti počítače. Zároveň musíme navrhnout, jakým způsobem budou pravidla vyhodnocována při vykonávání byznysových operací. Centrální správa pravidel vyžaduje vytyčení procesu, kterým bude možné administrovat veškerá pravidla v systému s ohledem na zachování jejich konzisteního vykonávání.

4.1 Formalizace architektury orientované na služby

V kapitole 2 jsme již identifikovali, jaké průřezové problémy, resp. aspekty, jsou řešeny v informačních systémech. Dospěli jsme k závěru, že byznysová pravidla jsou významným zástupcem těchto problémů. V sekci 2.3 jsme shrnuli konkrétní problémy byznysových pravidel, které konvenční přístup k návrhu a vývoji IS neumí v rámci SOA efektivně řešit. Pro formalizaci SOA do termínů AOP musíme také identifikovat *join-points*, ve kterých je možné aspekty v podobě byznysových pravidel aplikovat. Dále je potřeba určit podobu *advices*, popsat způsob jakým budou zachyceny *pointcuts* a nakonec navrhnout proces *weavingu* pravidel.



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

4.1.1 Join-points

Při identifikování join-points budeme vycházet 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. Ve chvíli, kdy je inicializace hotova, vstupuje služba do fáze, ve které může přijímat požadavky na vykonání byznysových operací. Pokud služba přijme takový požadavek, je nejprve nutno určit byznysový kontext, a poté vyhodnotit veškeré *preconditions*. Pokud jsou všechny předpoklady pro spuštění operace splněny, může být vykonána. Po dokončení operace je nutno aplikovat relevantní post-conditions.

Identifikované join-points tedy jsou:

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

4.1.2 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.

Nyní je potřeba se zamyslet, jakým způsobem budou selektory join-pointů pro jednotlivá pravidla zapsány. Pokud bychom chtěli u každého byznysového pravidla zapsat, ke kterým byznysovým operacím se vztahuje, museli bychom předem znát seznam veškerých byznysových operací implementovaných v celém systému. Pokud si představíme příklad ze sekce 2.3, musela by služba implementující vystavování faktur předem vědět o všech případech, kde

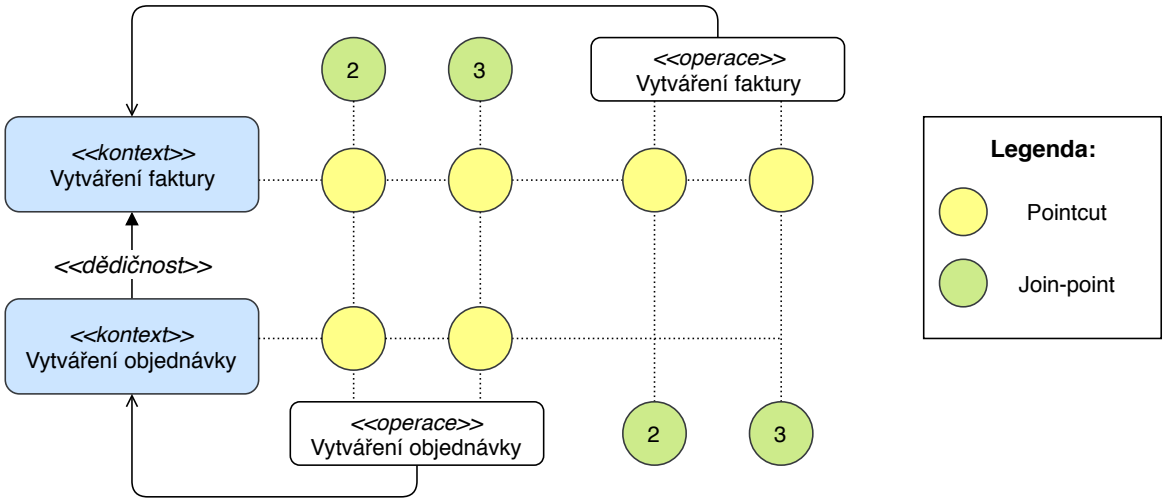
Zdrojový kód 4.1: Ukázka zápisu validačních pravidel pomocí anotací v jazyku Java

```
class BillingAddress {

    @NotBlank(message = "country is compulsory")
    private String country;

    @NotBlank(message = "city is compulsory")
    private String city;

    @NotBlank(message = "street is compulsory")
    private String street;

    @NotBlank(message = "postalCode is compulsory")
    private String postalCode;

    /* ... */

}
```

bude potřeba validovat validační adresu, aby tato místa mohla adresovat. To ovšem není příliš vhodné řešení.

Lepším způsobem by bylo nechat kontrolu na byznysových operacích. Ty by si mohly samy vyžádat byznysová pravidla, která potřebují. Tento koncept je využíván například standardem JSR 303 [6], který umožňuje validovat data byznysových objektů vstupujících do byznysových operací pomocí anotací atributů těchto objektů. Příklad validačních anotací můžeme vidět ve zdrojovém kódu 4.1, kde je pomocí anotace `@NotNull` zajištěno, že fakturační adresa bude mít vyplněná všechna náležitá pole. V našem kontextu se tedy jedná o paralelu preconditions. Místo toho, abychom u validačního pravidla `@NotNull` vypisovali všechna místa v kódu, kde má být použito, využijeme na těchto místech jeho anotaci. V našem případě by podobným způsobem každá byznysová operace, která by využívala pravidla pro validaci fakturační adresy, mohla specifikovat, že toto pravidlo bude využívat.

Toto řešení nám však neposkytuje možnost dynamicky při běhu programu změnit sadu byzynsových pravidel, které mají být aplikovány na byzynsovou operaci. Museli bychom konfiguraci toho, která pravidla budou aplikována na kterou operaci, přesunout do jakési externí dynamické konfigurace. To by ale významně snížilo přehlednost kódu. Vhodným kompromisem by mohl být koncept byznysového kontextu, který zapouzdřuje byznysová pravidla, a byznysová operace se na něj může explicitně odkázat. Byznysový kontext by



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

přitom mohl být dynamicky změněn za běhu programu.

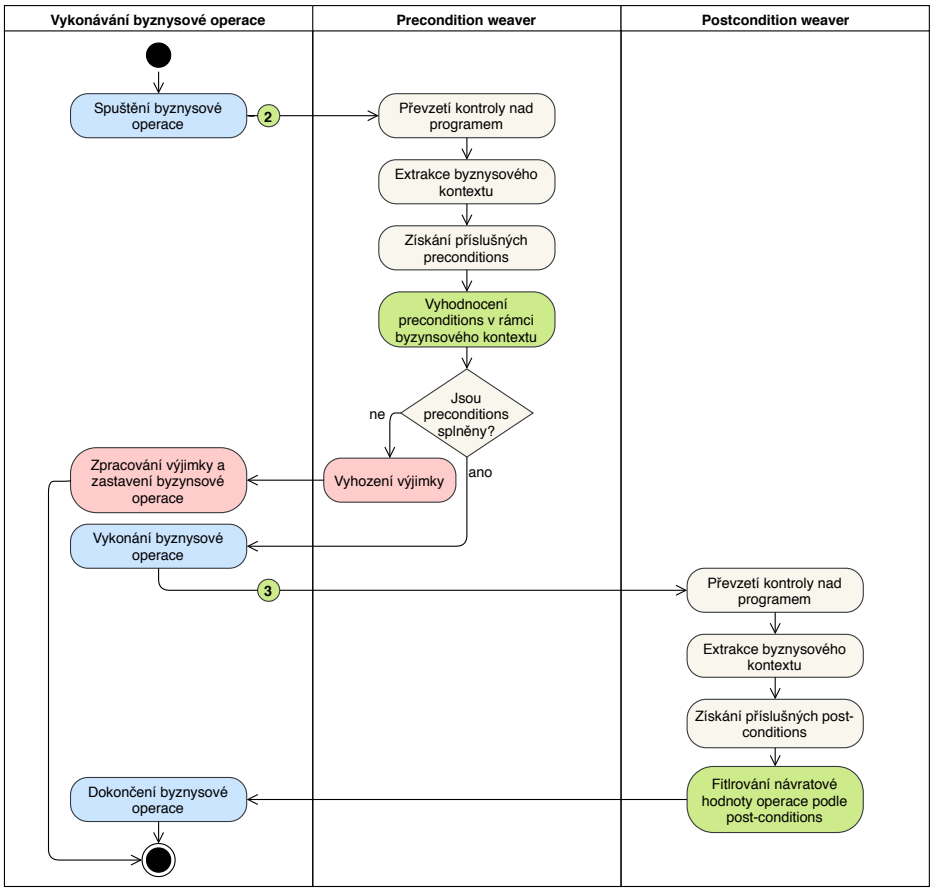
Sdílení pravidel mezi byznysovými kontexty, potažmo byznysovými operacemi a mezi jednotlivými službami, by mohlo být realizováno pomocí dědičnosti kontextů. Každý kontext, který by potřeboval validovat fakturační adresu, by tak mohl pouze dědit od kontextu vytváření faktury. Na obrázku 4.2 můžeme vidět, jak by takový případ mohl vypadat. Kontext vytváření objednávky dědí od kontextu vytváření faktury a sdílí tak jeho byznysová pravidla. Byznysové operace se odkazují na byznysové kontexty, které mají být při jejich vykonávání použity. Jak můžeme vidět, 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 se za advice dá považovat repretenzace 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.

4.1.4 Weaving

Weaving v případě join-pointu ① provádí komponenta frameworku, která analyzuje lokálně dostupná pravidla služby, vyhodnotí, která pravidla je potřeba stáhnout, a vyžádá tato pravidla od příslušných služeb.

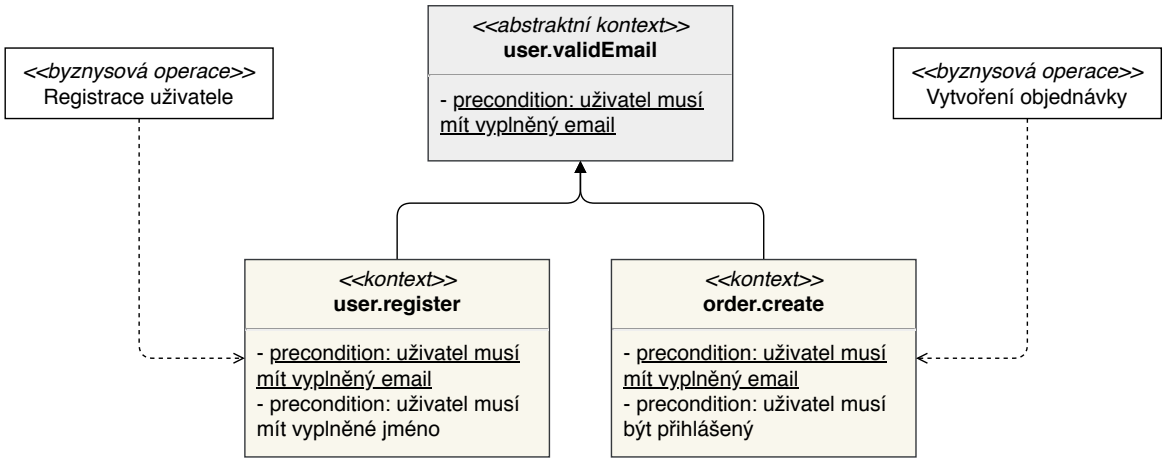


Obrázek 4.3: Diagram aktivit weaverů byznysových pravidel

V případě join-pointů ② a ③ se o weaving postará 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, na kterou musí služba reagovat. V opačném případě je kontrola vrácena zpět službě, která vykoná byznysovou operaci. Po jejím dokončení opět přichází na řadu aspect weaver, který zachytí výstup byznysové operace a aplikuje post-conditions daného byznysového kontextu. Proces weavingu je zachycen na obrázku 4.3.

4.2 Dědičnost byznysových kontextů

V předchozím textu jsme představili kontext dědičnosti byznysových kontextů. Ten funguje tak, že libovolný kontext může rozšiřovat libovolné množství jiných kontextů, a sdílet jejich byznysová pravidla, tedy jejich preconditions a post-conditions. Byznysové operace pak



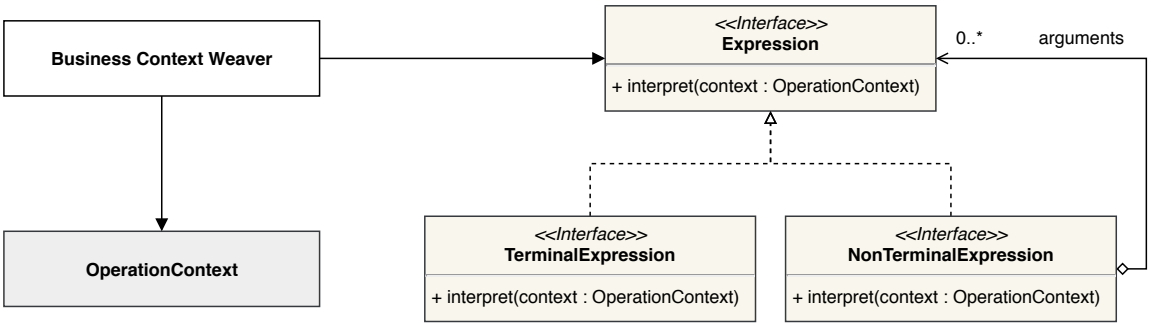
Obrázek 4.4: Diagram konceptu abstraktního byznysového kontextu

mohou samy určit, který byznysový kontext se k ním váže. Budeme-li chtít tento koncept využít, musíme však vyřešit několik otázek, které přináší.

Pokud bychom mapovali byznysové kontexty a byznysové operace jedna ku jedné, mohla by nastat situace, kdy chceme využít pouze nějaká byznysová pravidla jiného kontextu, ale ne všecha. Příkladem může být proces vytváření objednávky a proces registrace uživatele. Při vytváření objednávky chceme zaslat uživateli potvrzující mail, a proto potřebujeme, aby měl vyplněnou e-mailovou adresu. Zároveň chceme, aby byl uživatel při vytváření objednávky přihlášený. Při registraci uživatele bychom mohli rozšířit kontext vytváření objednávky, ovšem nechceme využít pravidlo vyžadující přihlášení uživatele. Tuto situaci lze vyřešit tak, že rozvolníme vztah mezi kontexty a operacemi a umožníme využívat tzv. *abstraktní kontexty* – tedy takové kontexty, které přímo nevyužívá žádná byznysová operace. V našem příkladu bychom mohli tedy pravidlo vyžadující vyplnění emailové adresy vyčlenit do abstraktního kontextu, ze kterého by dědil jak kontext vytváření uživatele, tak kontext vytváření objednávky. Situace je znázorněna na obrázku 4.4.

Dalším problémem je kruhová závislost kontextů, která nastává, když kontext A dědí od kontextu B, a ten opět dědí od kontextu A. Pokud tato situace nastane, nemůžeme ji z hlediska frameworku vyřešit. Je tedy nutné, aby byznysové kontexty systému využívajícího námi navrhovaný framework neobsahovaly cyklus. K zajištění této skutečnosti by mohl sloužit validátor vestavěný do nástroje pro správu byznysových kontextů.

Kvůli vícenásobné dědičnosti může nastat problém, kdy jeden kontext zdědí více stejných pravidel z různých zdrojů. Představme si, že kontext D dědí od kontextů B a C, přičemž kontexty B a C dědí od kontextu A. Kontext D pak zdědí pravidla kontextu A dvakrát. Tomu lze předejít tak, že každé pravidlo bude mít unikátní identifikátor v rámci celého systému. V případě, že kontext získá dvě pravidla se stejným identifikátorem, vybere si jen jedno z



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

nich a druhé může zahodit, protože jsou identická. Zajištění unikátního identifikátoru můžeme vynutit díky nástroji pro centrální administraci byznysových pravidel, která je součástí navrhovaného frameworku.

Nakonec bychom měli zvážit problém, kdy kontext kvůli dědičnosti získá takovou množinu preconditions, která nebude nikdy splnitelná. Příklad je velmi jednoduchý – pokud jedna precondition vyžaduje přihlášení uživatele a druhá precondition naopak požaduje, aby uživatel nebyl přihlášen, nemůže nastat situace, kdy budou obě preconditions uspokojeny. Takovéto stavy by měl primárně hlídat administrátor či architekt systému, nicméně můžeme jeho práci usnadnit díky nástroji pro centrální administraci kontextů.

4.3 Logické výrazy byznysových pravidel

V sekci 2.1 jsme analyzovali byznysová pravidla a jedním z našich zjištění bylo, že pravidla obsahují logické podmínky, které je potřeba vyhodnocovat. V případě preconditions je to ověření podmínky, která musí být platná před spuštěním byznysové operace. V případě postconditions, kdy chceme filtrovat návratovou hodnotu byzynsové operace, nemusíme logickou podmínku potřebovat vždy. Může ale nastat moment, kdy chceme filtrovat hodnotu pouze pokud je splněna nějaká podmínka – například pokud uživatel není administrátorem.

Vyhodnocování logických výrazů byznysových pravidel bude prováděno naším frameworkem při weavingu byznysových kontextů. K tomuto účelu je velmi vhodný návrhový vzor *Interpreter* [32]. Základní myšlenkou tohoto vzoru je interpretace jazyka, kdy každý jeho výraz (z anglického *expression*) je reprezentován samostatným objektem, který přebírá zodpovědnost za správnou interpretaci daného výrazu. Logické výrazy tvoří orientovaný acyklický graf (DAG), tzv. *derivační strom*, a rozdělují se na tzv. *terminály* a *neterminály* [55]. Terminál znamená, že z daného výrazu již nevychází žádná hrana do jiného výrazu. Neterminál je opak terminálu. Na obrázku 4.5 můžeme vidět použití vzoru interpreter pro náš účel. Můžeme si

také všimnout, že strom výrazů opisuje návrhový vzor *Composite* [32].

V rámci vzoru Interpreter si dále můžeme všimnout toho, že vyhodnocované výrazy mají přístup k objektu `OperationContext`. V něm jsou držena veškerá data, která jsou pro interpretaci výrazu potřebná. Jednotlivé výrazy se tedy mohou odkazovat na proměnné či konstanty obsažené v tomto kontextu. Pokud bude uživatel frameworku potřebovat rozšířit funkcionalitu pravidel, mohly by se mu hodit i speciální funkce, které by si mohl do operačního kontextu zadefinovat.

V rámci této kapitoly bychom bychom měli navrhnout i základní sadu výrazů, které budou sloužit pro zápis byznysových pravidel. Kromě základních logických operací jako je `and`, `or`, `equals` a `negate` budeme potřebovat i výraz, který získá hodnotu proměnné či konstanty z kontextu. Pokud bude v proměnné uložen objekt, bude potřeba přistupovat i k jeho veřejným atributům, což bude vyžadovat další speciální výraz. Uživatel taky potřebuje možnost otestovat, zda je v odkazované proměnné hodnota. K tomu může sloužit výrat `IsNotNull`, který ověří proměnnou libovolného typu na přítomnost hodnoty, a výraz `IsNotBlank`, který ověří, zda je v proměnné řetězec nenulové délky. Některé případy použití mohou vyžadovat vložení konstantní hodnoty přímo do byznysového pravidla – k tomuto účel by mohl sloužit speciální terminál `Constant`. Pro zvýšený komfort můžeme do frameworku přidat i podporu základních matematických operací, jako je sčítání, odečítání, násobení a dělení. Pro volání funkcí definovaných v operačním kontextu je zapotřebí další speciální výraz. V jeho případě je nutno dbát na to, že funkce může přijímat libovolný počet argumentů. Protože volaná funkce může potřebovat přistupovat k operačnímu kontextu, musejí být argumenty také interpretovány naším frameworkem. Bohužel nemůžeme u uživatelem definovaných funkcí ověřit, že bude při volání byznysového pravidla odpovídat počet a typ argumentů. Toto si tedy bude muset uživatel frameworku zajistit sám. Přehled všech výrazů, které bude framework podporovat, je v tabulce 4.1,

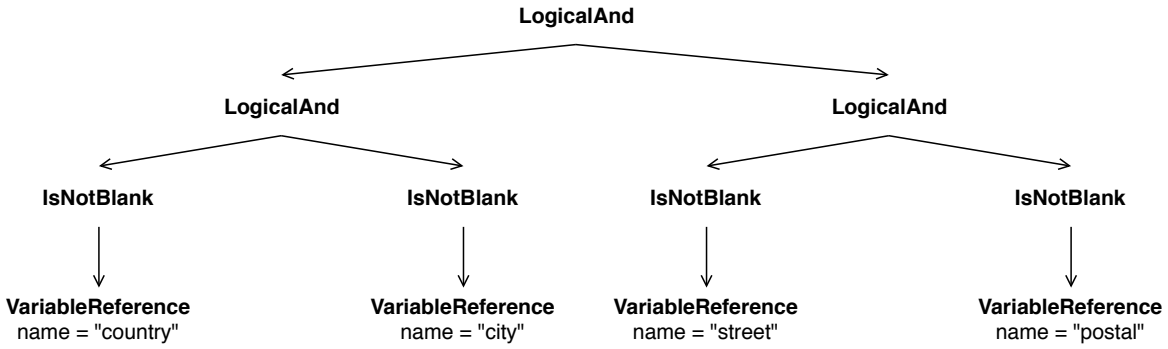
[1] Musíme mít na paměti, že konstanty a proměnné mohou nabývat různých hodnot rozdílného typu. Pokud navrhujeme framework, který má sloužit pro více platforem, existuje možnost, že některý z podporovaných jazyků bude využívat typový systém. Pro ulehčení implementace tedy musíme k logickým výrazum uložit i informace o typu jejich argumentů a jejich návratové hodnoty. Díky tomu bude možné framework rozšířit o typovou kontrolu, což přinese menší náchylnost k sémantickým chybám v zápisu pravidla. Výraz byznysového pravidla může nabývat logických hodnot, může vracet číslo, textový řetězec a také objekt. Musíme také počítat s tím, že výraz nevrací žádnou hodnotu. Je nutno podotknout, že vzhledem k povaze byznysových pravidel musí kořen pravidla vždy vracet logickou hodnotu. Následuje výčet typů, kterých mohou výrazy nabývat.

- `BOOL` je logický typ, který nabývá hodnoty `true` a `false`.

[1] [Intended Delivery: Typované výrazy]

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 4.1: Přehled výrazů pro zápis byznysového pravidla



Obrázek 4.6: Syntaktický strom jednoduchého validačního pravidla

- `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. Nazývejme je *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 můžeme vidět 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 `country`, `city`, `street` a `postal`. Každá z těchto proměnných je argumentem validačního výrazu `IsNotBlank`. Tento výraz vrací hodnotu `true`, pokud jeho argumentem je řetězec, který obsahuje alespoň jeden znak. Jednotlivé validace jsou poté spojeny pomocí binárního výrazu `LogicalAnd`, který vrací hodnotu `true` tehdy a pouze tehdy, když oba jeho argumenty mají hodnotu `true`. Pro přehlednost jsme do diagramu nezanášely návratové typy výrazů.

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

4.4 Filtrování návratových hodnot byznysové operace

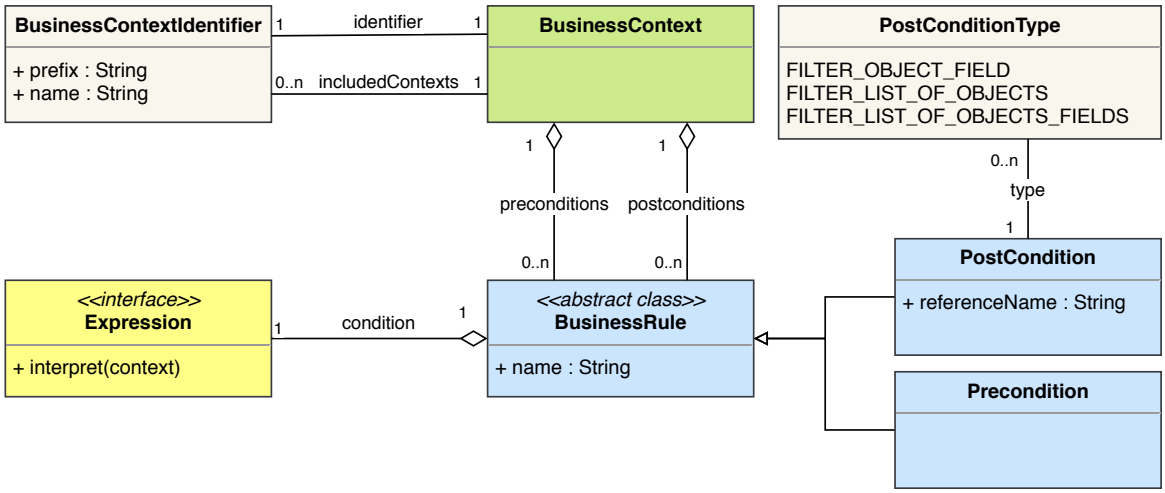
Při aplikování post-conditions je potřeba navrhnout, jakým způsobem bude probíhat filtrování návratové hodnoty byznysové operace. Návratovou hodnotou může být proměnná obsahující jednoduchou hodnotu, jako je číslo nebo text, objekt, či kolekci hodnot nebo objektů. Filtrování jednoduchých hodnot nemá význam, protože by to znamenalo, že byznysová operace nevrátí žádnou hodnotu, ačkoliv s tím kód sytému může počítat. Pokud bude vývojář systému chtít tuto možnost opravdu použít, může jednoduchou návratovou hodnotu zabalit do objektu, který ji bude přenášet. V případě návratové hodnoty v podobě objektu pak můžeme požadovat filtrování některých jeho atributů, například můžeme chtít při zobrazování uživatelského profilu filtrovat e-mailovou adresu uživatele, pokud uživatel není administrátor. V případě kolekce můžeme požadovat, aby byly některé její prvky odstraněny, například nechceme zobrazit objednávky, které uživateli nepatří. Pokud se v kolekci nachází objekty, můžeme také požadovat, aby byly zakryty atributy jednotlivých objektů – analogie na 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.

V tuto chvíli je potřeba zavést konvenci, která bude říkat, kdy bude návratová hodnota vyfiltrována, konkrétně zda to bude při splnění či nesplnění podmínky post-condition. V textu výše jsme uváděli, že post-condition bude aplikována, pokud je podmínka splněna. Držme se tedy tohoto značení.

4.5 Metamodel byznysového kontextu

Když nyní známe způsob, jakým zachytíme podmínky byznysových pravidel, můžeme navrhnout kompletní model byznysových pravidel, potažmo byznysových kontextů. Kromě samotných logických výrazů je potřeba také ukládat informace o tom, zda se jedná o precondition nebo post-condition, a identifikátor pravidla. Post-condition navíc potřebuje uložit informaci o jejím typu a názvu. Jak jsme již v předchozím textu psali, pravidla budou uskupována do byznysových kontextů, z nichž každý má svůj unikátní identifikátor skládající se z prefixu a samotného jména a také drží informaci o tom, od kterých ostatních kontextů dědí. Diagram tříd navrženého kontextu je znázorněn na obrázku 4.7.



Obrázek 4.7: Diagram tříd metamodelu byznysového kontextu

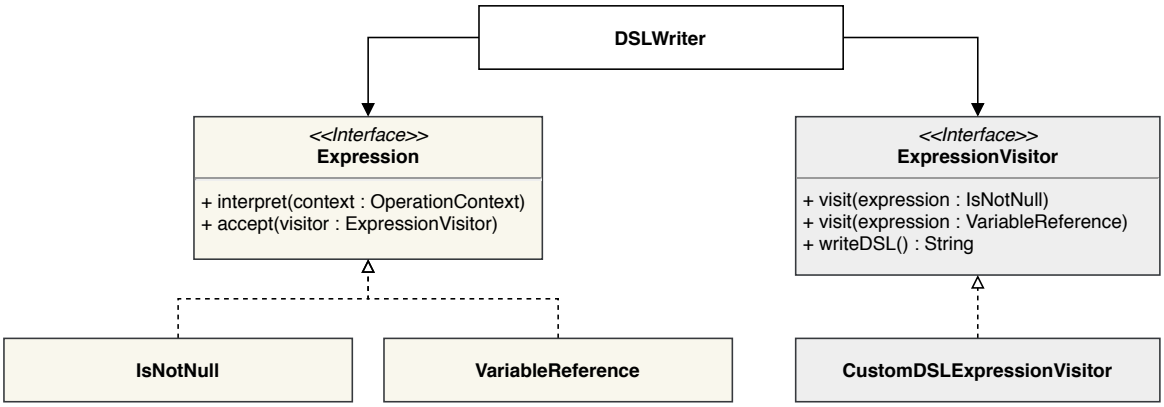
4.6 Popis byznysových pravidel pomocí DSL

Přístup ADDA doporučuje popsat byznysová pravidla pomocí vlastního, na míru šitého, doménově specifického jazyka [17]. V našem případě můžeme jazykem DSL popsat kompletně i celý byznysový kontext.

V sekci 3.5 jsme zjistili, že ačkoliv jsou nástroje Drools a JetBrains MPS velmi silnými aparáty, jejich vlastnosti nejsou plně vhodné pro řešení problému centrální administrace a automatické distribuce byznysových pravidel. Můžeme jít cestou volby co nejvhodnějšího jazyka pro každou platformu, kterou chceme využívat pro naše účely, a implementace adapterů pro tyto jazyky, které transformují dané DSL do metamodelu. Vzhledem k různým vlastnostem těchto jazyků by ale takový přístup byl suboptimální, museli bychom totiž zohlednit „nejmenší společný jmenovatel“ – tedy zápis kontextu by byl limitován sjednocením všech nedostatků těchto jazyků.

Na tomto místě bychom měli specifikovat, jak by DSL popisující byznysový kontext mělo vypadat. DSL popisující byznysový kontext by mělo umožňovat zachytit identifikátor, seznam kontextů, které rozšiřuje, a poté jednotlivá pravidla, tedy preconditions a post-conditions. Každé byznysové pravidlo musí mít svůj vlastní identifikátor a podmínku vyjádřenou logickým výrazem, který jsme detailně rozebrali v předchozím textu. Post-condition musí navíc obsahovat popis typu a kromě typu `FILTER_LIST_OF_OBJECTS` je také potřeba popsat název pole, které má být v objektu filtrováno.

Pro efektivní využití DSL musíme také navrhnout způsob, jakým bude popis byznysového kontextu a jeho pravidel převáděn do metamodelu a vice versa. Pro načtení kontextu z DSL může sloužit návrhový vzor a *parser* [68]. Jejich volba se bude odvijet od zvoleného DSL. Pro uložení kontextu do DSL, aby ho mohl vývojář či administrátor systému upra-



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

vovat, je vzhledem ke zvolené reprezentaci a popisu logických výrazů derivačním stromem ideální volbou návrhový vzor *Visitor* [32]. Ten umožní elegantně převádět libovolně složité logické výrazy pomocí metody *double-dispatch*. Jeho volbou zároveň umožníme 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.

Samotný návrh a implementace kvalitního DSL splňující naše požadavky není předmětem této práce a je bohužel mimo její rozsah. Nicméně, v tuto chvíli můžeme pokračovat v návrhu frameworku, protože známe všechny potřebné detaily popisu byznysových kontextů. Framework je navíc na konkrétním DSL nezávislý. Uživatelé frameworku dokonce mohou sestrojit DSL na míru potřebě svého systému a díky tomu dosáhnou maximálního komfortu při jeho vývoji.

4.7 Organizace byznysových pravidel

Předpokládáme, že každá služba má lokálně uložen popis byznysových kontextů, které se sémanticky vztahují k její doméně. Jak jsme již nastínili v sekci 2.3, služba při výkonu jedné byznysové operace může potřebovat aplikovat byznysová pravidla, která jsou aplikována také při výkonu jiné byznysové operace v jiné službě – tedy patří do byznysového kontextu jiné služby. Abychom mohli lépe určit, kde jsou jednotlivá pravidla, potažmo jednotlivé kontexty definovány, přidáme do identifikátoru kontextu tzv. *prefix*. Jeden prefix pak bude spravován výhradně jednou službou a podle toho poznáme, kde v systému jednotlivé kontexty hledat. Například kontexty služby spravující objednávky budou označeny prefixem `order`, zatímco kontexty služby zajišťující fakturaci budou označeny prefixem `billing`. Může nastat situace, kdy jedna služba bude spravovat více prefixů – to však ničemu nevadí.

4.7.1 Registr byznysových kontextů

Cílem našeho přístupu je soustředit byznysové kontexty na jedno místo, ze kterého budou automaticky distribuovány. Pro tento účel využijeme komponentu, která bude mít za úkol kontexty načítat z DSL do metamodelu, načítat lokálně nedostupné kontexty z ostatních služeb a načtené kontexty uchovávat pro použití při weavingu. Tuto komponentu budeme nazývat registr byzynsových kontextů (`BusinessContextRegistry`). Každá služba pak bude disponovat svým registrem. Při inicializaci kontextů spolu budou registry komunikovat a vyměňovat si sdílené kontexty.

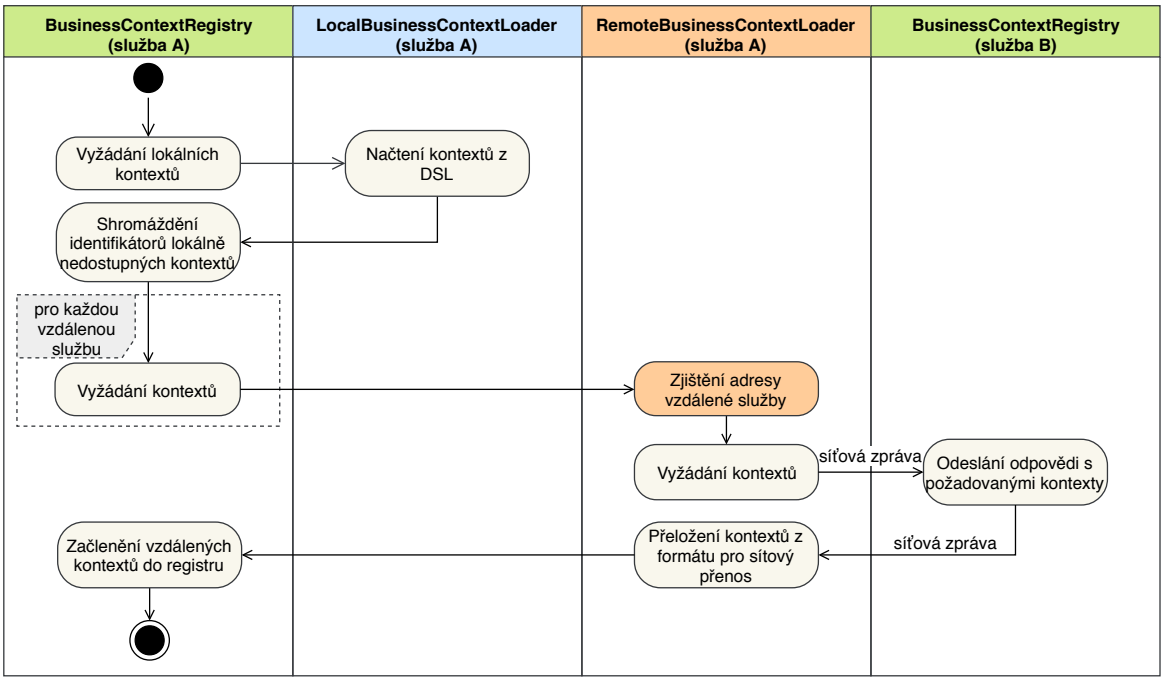
4.7.2 Uložení kontextů

Byznysové kontexty popsané pomocí DSL mohou být v příslušné službě uložena 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 inicializaci služby je potřeba načíst všechny byznysové kontexty, které bude služba pro svůj běh potřebovat. Mohli bychom argumentovat, že načítat příslušný kontext by šlo až ve chvíli, kdy je opravdu zavolána byznysová operace. Kontext bychom pak mohli uložit do paměti, aby se nemusel načítat při dalším spuštění operace. Toto řešení by rozprostřelo latenci při inicializaci služby mezi jednotlivá první spuštění každé byznysové operace. Zároveň by se však zvýšila komplexita frameworku. V rámci principů KISS [76] a YAGNI [4] využijeme první zmíněné řešení. Nicméně, pokud by druhé řešení v budoucnu přinášelo další benefity, můžeme ho do frameworku snadno přidat.

Během inicializiace byzynsových kontextů je potřeba nejprve načíst lokálně dostupné kontexty popsané pomocí DSL. Po převedení z DSL do metamodelu je potřeba shromáždit seznam rozšířených kontextů a zjistit, které rozšířené kontexty nejsou lokálně dostupné. Následně je potřeba vyžádat si vzdálené kontexty od příslušných služeb a převést obdržené kontexty z formátu pro síťový přenos do formátu, ve kterém budou kontexty uloženy v paměti. Nakonec je potřeba, aby do všech kontextů, které využívají dědičnosti, byly začleněny byznysová pravidla rozšířených kontextů. Celou inicializaci může zastřešovat komponenta `BusinessContextRegistry`, která je vhodným kandidátem, protože má znalost o všech subsystémech, které jsou k tomuto procesu potřeba. Jak si můžeme všimnout, tato komponenta implementuje návrhový vzor *Facade* [32]. Na obrázku 4.9 je znázorněn navržený proces inicializace.



Obrázek 4.9: Diagram procesu inicializace byznysových kontextů

4.9 Centrální správa byznys kontextů

Vzhledem k nutnosti centralizovat správu byznysových kontextů se nám architektura P2P představená v subsekci 3.2.2 nehodí. Při úpravě kotextů by totiž v systému mohly existovat staré i nové verze byznysových pravidel, což je pro správnou funkci systém nepřijatelné. Využijeme tedy architektury klient-server s více servery. Byznysové kontexty budou podle svého prefixu rozděleny do skupin a každou ze skupin bude spravovat jedna služba, která bude zároveň držet jejich aktuální a jediný stav.

4.9.1 Uložení rozšířeného pravidla

[4] Při ukládání byznysového kontextu, od kterého dědí jiné kontexty, musíme zajistit, že bude změna korektně propagována. Máme dva způsoby, kterými toho můžeme docílit. Jeden z nich je, že služba, která je původcem upraveného pravidla, sama informuje o změně všechny ostatní služby, které si od ní pravidlo vyžádaly. Tento způsob však vyžaduje implementaci registru, do kterého by si služba poznamenávala které služby při změně kterého pravidla kontaktovat. Navíc není zaručeno, že to vždy půjde, protože služba teoreticky nemusí být v opačném směru dostupná – například kvůli překladu síťových adres NAT.

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

Druhým způsobem je přenechat kontaktování zúčastněných služeb na nástroji pro centrální správu byzynsových pravidel. Ten má totiž přehled o všech závislostech v systému a zároveň zná i adresu všech služeb. Nástroj by tak mohl při ukládání upraveného pravidla zmapovat, které kontexty je potřeba aktualizovat, a následně požádat služby, ke kterým kontexty patří, aby tyto kontexty znovu sestavili. Nevýhodou tohoto přístupu je zvýšená komunikačmní zátěž kvůli většímu objemu přenesených informací. V tuto chvíli lze spekulovat, že tato zátěž je vůči absolutnímu objemu přenášených dat v systému využívajícímu SOA zanedbatelná. Bylo by ale vhodné při implementaci vybrat vhodný přenosový formát, který minimalizuje dopad veškeré síťové komunikace týkající se byznysových pravidel.

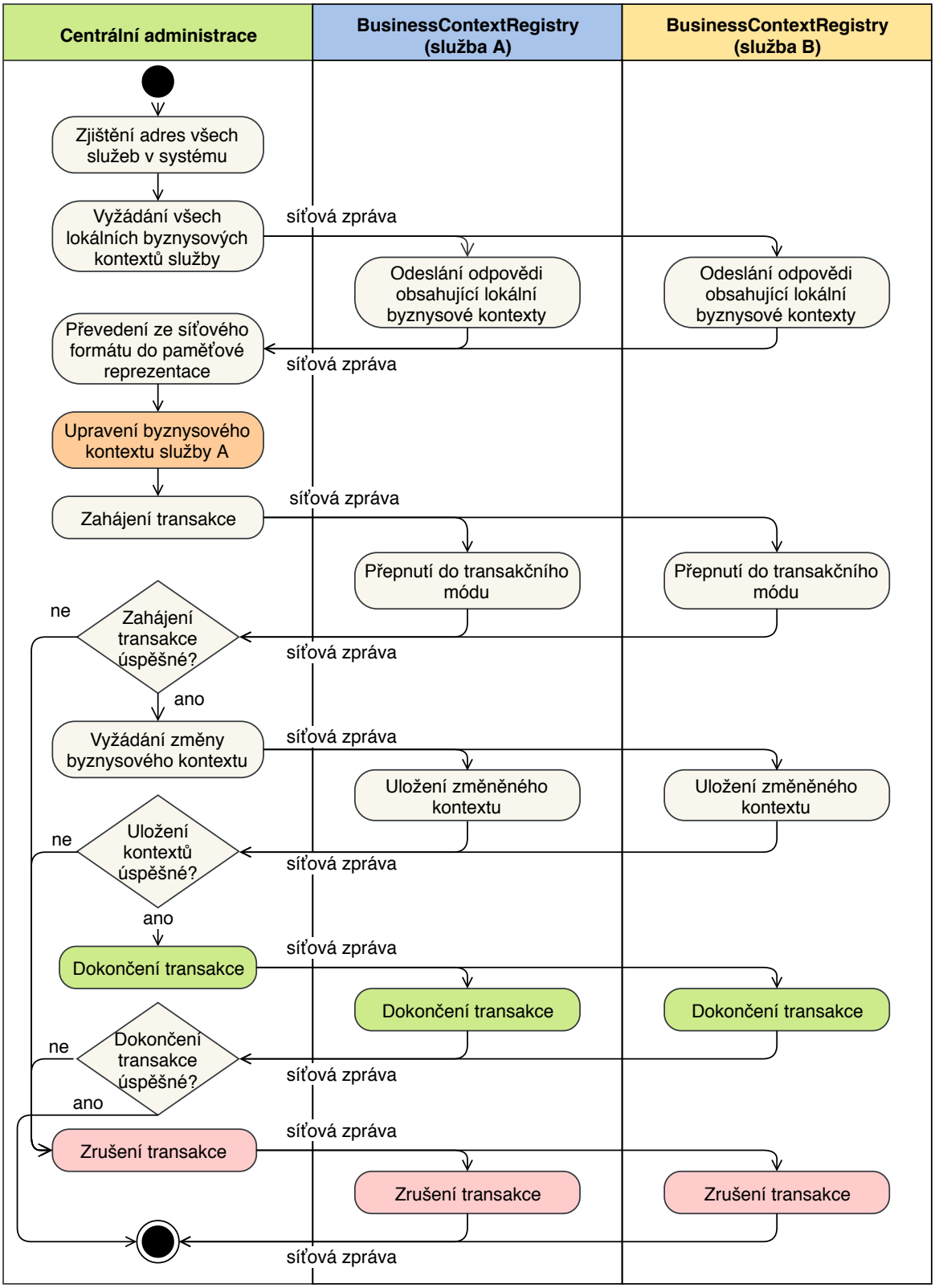
4.9.2 Proces úpravy kontextu

Nyní můžeme navrhnout samotný proces úpravy byznysového kontextu pomocí nástroje pro centrální administraci. Ten pro svou funkci musí načíst všechny byznysové kontexty všech služeb v systému. Následně může zobrazit administrátorovi formulář pro úpravu pravidla. Pravidlo by mělo být převedeno z metamodelu do DSL pro snažší úpravu. Po odeslání formuláře bude pravidlo převedeno zpět do metamodelu. Nástroj pro administraci by měl v tuto chvíli analyzovat 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.

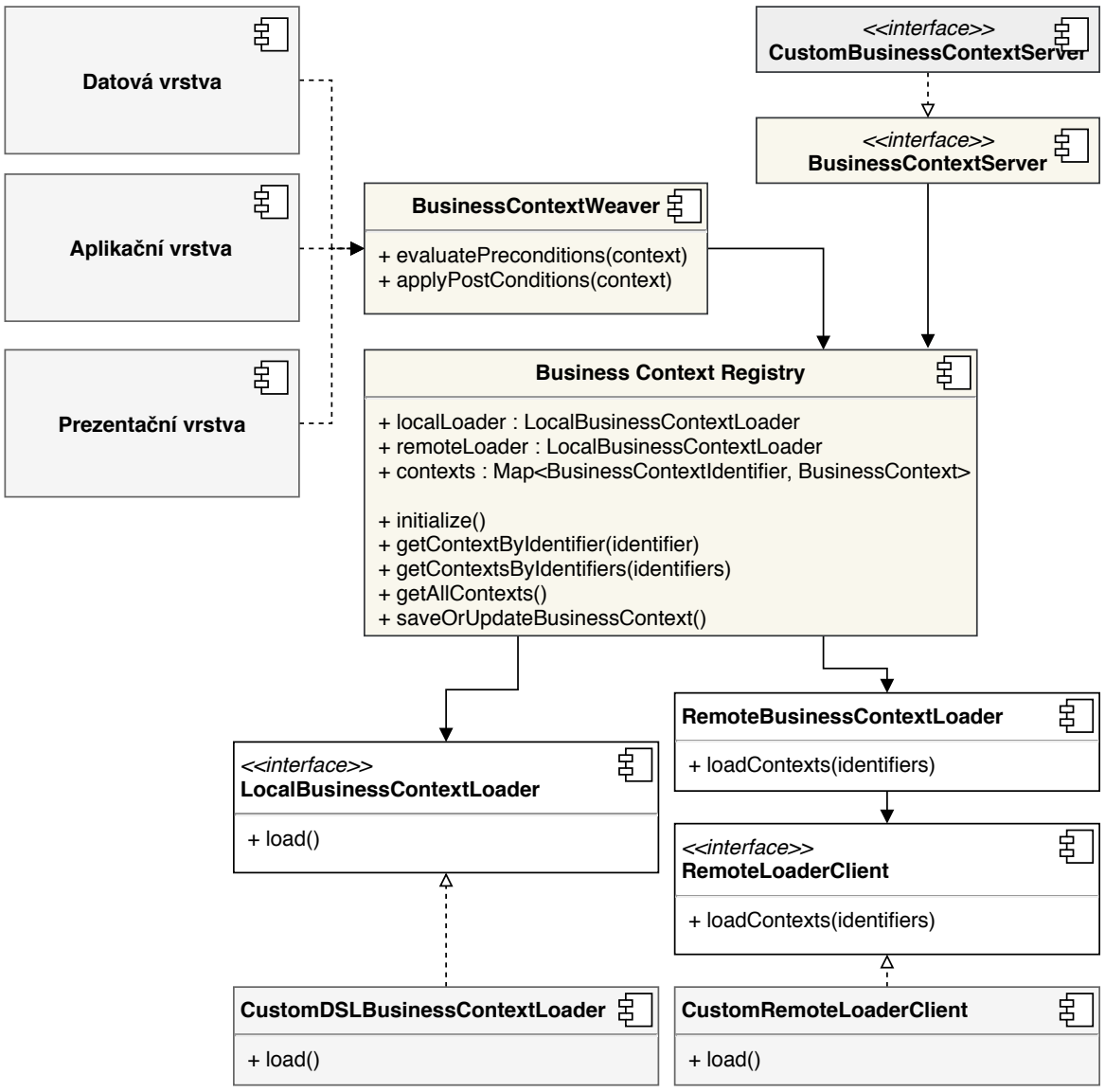
4.10 Architektura frameworku

Nyní, když jsme navrhli jednotlivé části systému a jejich součinnost, můžeme popsat architekturu celého frameworku. V této sekci budeme předpokládat klasickou třívrstvou architekturu [32], která se skládá z prezentační, aplikační a datové vrstvy. Každá z těchto vrstev může využívat náš framework – 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. Pro účely této práce se soustředíme zejména na aplikaci v aplikační vrstvě.

Středobodem našeho frameworku je komponenta `BusinessContextRegistry`, tedy registr byznysových kontextů, který je zodpovědný za inicializaci a uchovávání byznysových kontextů. Načítání 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.



Obrázek 4.10: Diagram procesu centrální správy byznysových kontextů



Obrázek 4.11: Diagram tříd navrženého frameworku

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 zoraginzuje 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. Pro účely této práce poskytneme weavery pro využití v aplikační vrstvě pro weaving preconditions a post-conditions do byznysových operací.

4.10.1 Service discovery

Abychom mohli přenášet 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. Náš framework proto nesmí být závislý na způsobu, jakým se adresování služeb provádí. Nejlepším způsob je přenechat na uživateli, aby sám frameworku 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 jsme navrhli framework pro centrální správu a automatickou distribuci byznysových pravidel v SOA na základě přístupu ADDA. Formalizovali jsme doménu SOA do názvosloví AOP. Diskutovali jsme podobu byznysových pravidel a jejich logických výrazů a jakým způsobem ji zachytit jak v paměti počítače, tak pomocí DSL. Zvážili jsme výhody organizace pravidel do byznysových kontextů a představili koncept dědičnosti. Vymodelovali jsme procesy, kterými budou pravidla distribuována a také proces pro přídání či úpravu byznysového pravidla za běhu systému. Nakonec jsme shrnuli celkovou architekturu 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*. V této kapitole si popíšeme, jaké plaformy jsme vybraly, a jakým způsobem byly prototypy knihoven implementovány. Součástí kapitoly je i stručná rešerše technologií, které byly použity pro dosažení vytyčených cílů.

[2] Jelikož vycházejí implementace knihoven pro všechny platformy ze stejného návrhu představeného v předchozí kapitole 4, popíšeme si kompletní implementaci pro jazyk Java a ostatní implementace shrneme komparativní metodou.

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

5.1 Výběr použitých platforem

[4] Mimo jazyk Java, který byl určen zadáním, byla pro implementaci vybrána platforma jazyka *Python* a platforma *Node.js*, který slouží jako běhové prostředí pro jazyk *JavaScript*. Výběr byl proveden na základě aktuálních trendů ve světě softwarového inženýrství. Projekt GitHut [64] z roku 2014, který shrnuje statistiky repozitářů populární služby pro hosting a sdílení kódu GitHub[5], určil jazyky JavaScript, Java a Python jako tři nejaktivnější. Služba GitHub následně sama zveřejnila statistiky za rok 2017 v rámci projektu Octoverse [38]

[1] [Intended Delivery: Uvedení kapitoly a nastínění obsahu]
[2] [Intended Delivery: Nástin formátu kapitoly]
[3] [Intended Delivery: Technické implementační problémy]
[4] [Intended Delivery: Jaké jsme vybrali další platformy a proč]
[5] https://github.com/

a dospěla ke stejnému závěru, ačkoliv Python se umístil na druhé pozici na úkor jazyka Java. Podle průzkumu oblíbeného programátorského webového portálu Stack Overflow [74] se umístily tyto jazyky v první čtveřici nejpopulárnějších jazyků pro obecné použití.

5.2 Sdílení byznys kontextů mezi službami

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

[7] Pro síťovou komunikaci se nabízí využít architekturu *klient-server*, kterou jsme detailněji popsali v sekci 3.2.1. Při sdílení kontextů lze chápat *klienta* jako službu, která pro svou funkci vyžaduje získání kontextu definovaného v jiné službě. Jako *server* lze naopak chápat službu, která poskytne své kontexty jiné službě, která na nich závisí. Jinými slovy, klient si vyžádá potřebné kontexty od serveru a ten mu je v odpovědi zašle. Může se také stát, že některá služba bude zároveň serverem jedné služby, a zároveň klientem druhé služby.

5.2.1 Protocol Buffers

[8] Pro přenos byznysových kontextů byl zvolen open-source formát *Protocol Buffers* [65][79] vyvinutý společností Google[9]. Umožňuje explicitně definovat a vynucovat schéma dat, která jsou přenášena po síti, bez vazby na konkrétní programovací jazyk. Zároveň poskytuje obslužné knihovny pro naše vybrané platformy. Navíc je díky binární reprezentaci dat v přenosu velmi efektivní, oproti formátům jako je JSON nebo XML [53]. Tím splňujeme i požadavek ze sekce 4.9.1 na minimalizaci dopadu síťového provozu týkajícího se byznysových kontextů na výkon celého systému. Oproti protokolům *Apache Thrift* [2] a *Apache Avro* [82], 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

```
1 message PreconditionMessage {
2     required string name = 1;
3     required ExpressionMessage condition = 2;
4 }
5
```

[6] [Intended Delivery: Formát pro přenos pravidel po síti a jeho výhody]
[7] [Intended Delivery: Architektura klient-server pro komunikaci kontextů mezi službami]
[8] [Intended Delivery: Proč jsme použili Protobuf]
[9] https://www.google.com/

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

Zdrojový kód 5.1 znázorňuje část zápisu schématu zasílaných zpráv obsahující byznys kontexty ve formátu Protobuffer. Schéma zpráv pro výměnu kontextů opisuje strukturu metamodelu navrženého v sekci 4.5.

ExpressionMessage obsahuje jméno, atributy a argumenty `Expression`

ExpressionPropertyMessage je enumerace obsahující typy atributu `Expression`

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

[10] Pro realizaci architektury klient-server byl zvolen open-source framework gRPC [39], který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC* [59]. Zdrojový kód 5.2 znázorňuje zápis serveru, který umožňuje svému klientovi volat metody `FetchContexts()`, `FetchAllContexts()` a `UpdateOrSaveContext()`.

[10] [Intended Delivery: Proč jsme použili gRPC]

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

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

    rpc FetchAllContexts (Empty)
        returns (BusinessContextsResponseMessage) {}

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

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

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

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

5.3 Doménově specifický jazyk pro popis byznys kontextů

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

[12] Pro popis kontextů byl jako kompromis mezi jednoduchostí implementace a přívětivostí pro koncového uživatele zvolen univerzální formát Extensible Markup Language (XML) [12]. Tento jazyk umožňuje serializaci libovolných dat, přímočarý a formální zápis jejich struktury a také jejich snadné aplikační zpracování. Zároveň poskytuje relativně dobrou čitelnost pro člověka, ačkoliv speciálně vytvořené DSL by bylo jistě čitelnější.

[11] [Intended Delivery: Popsat proč a jak jsme tvořili DSL]
[12] [Intended Delivery: Důvody pro výběr XML]

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

[14] Vzhledem k tomu, že XML je volně rozšiřitelný jazyk a neklade meze v možnostech struktury dat, bylo potřeba jasně definovat a dokumentovat očekávanou strukturu dokumentu popisujícího byznys kontext. Pro jazyk XML existuje vícero možností jak schéma definovat [47], od jednoduchého formátu *DTD* až po komplexní formáty jako je *Schematron*, či *XML Schema Definition* (XSD), který byl nakonec zvolen. Díky formálně definovanému schématu můžeme popis byznys kontextu automaticky validovat a vyvarovat se tak případných chyb.

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

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

Zdrojový kód 5.3: Příklad zápisu byznys kontextu v jazyce XML

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <businessContext prefix="user" name="createEmployee">
3   <includedContexts/>
4   <preconditions>
5     <precondition name="Cannot use hidden product">
6       <condition>
7         <logicalEquals>
```

[13] [Intended Delivery: Popis jak XML funguje]
[14] [Intended Delivery: Popis jaký formát jsme zvolili pro formální zápis schématu XML dokumentu]
[15] [Intended Delivery: Popis formátu]
[16] [Intended Delivery: Shrnutí DSL]

```
          <left>
            <variableReference
                objectName="product"
                propertyName="hidden"
                type="bool"/>
          </left>
          <right>
            <constant type="bool" value="false"/>
          </right>
        </logicalEquals>
      </condition>
    </precondition>
  </preconditions>
  <postConditions/>
</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
- Návrhové vzory - builder pro kontexty a pravidla
- Návrhové vzory - visitor pro převod expression do xml
- Návrhové vzory - interpreter pro interpretaci pravidel

]

5.4.1 Popis implementace

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

```
public class OrderService {

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

5.4.2 Použité technologie

Apache Maven [17] Pro správu závislostí a automatickou kompilaci a sestavování knihovny napsané v jazyce java byl zvolen projekt *Maven* [54]. 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 [18] 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

]

[17] [Intended Delivery: Správa závislostí a buildu projektu]
[18] [Intended Delivery: Proč AspectJ a co to umí]

JDOM 2 [19] Knihovna JDOM 2 [42] poskytuje kompletní sadu nástroju pro čtení a zápis XML dokumentů. Implementuje specifikaci *Document Object Model* (DOM) [84], pomocí které lze programaticky sestavovat a číst XML dokumenty. Tuto knihovnu jsme využili pro serializaci a deserializaci DSL byznys kontextů popsaných v sekci 5.3.

5.5 Knihovna pro platformu Python

Knihovna pro platformu jazyka Python využívá jeho verzi 3.6. Pomocí nástroje *pip* [62] lze knihovnu nainstalovat a využívat jako python modul. Implementace odpovídá navržené specifikaci.

[TODO

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

]

5.5.1 Srovnání s knihovnou pro platformu Java

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

[19] [Intended Delivery: Proč jdom2 a co to umí]

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

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

        return func_wrapper

    return wrapper


weaver = BusinessContextWeaver()


class ProductRepository:

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

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

5.5.2 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 [25]. Implementace odpovídá specifikaci návrhu, umožňuje instalaci pomocí balíčkovacího nástroje a snadnou integraci do kódu výsledné služby.

5.6.1 Srovnání s knihovnou pro platformu Java

Weaving Podobně jako v knihovně pro jazyk Python, i v knihovně pro Node.js byl oproti knihovně pro jazyk Java největší rozdíl v implementaci weavingu. Platforma Node.js totiž nedisponuje žádnou kvalitní knihovnou, která by ulehčila využití konceptů aspektově orientovaného programování. Jazyk JavaScript je ale velmi flexibilní a lze tedy pro dosažení požadované funkcionality využít podobně jako pro jazyk Python princip dekorátoru jako funkce. Ačkoliv zápis dekorátoru není příliš elegantní a kvůli použití konceptu *Promise* [44] poněkud složitější, podařilo se weaving implementovat spolehlivě. Ukázku můžeme vidět ve zdrojovém kódu 5.6. Funkce `register()` obsahuje logiku pro registraci uživatele, která může obsahovat například uložení entity do databáze a odeslání registračního e-mailu. Při exportování funkce z Node.js modulu využijeme `wrapCall()`, která má za úkol dekorovat předanou funkci `func`, před jejím zavolání vyhodnotit preconditions a po zavolání aplikovat post-conditions. Díky tomu bude každý kód, který využije modul definující funkci pro registraci uživatele, pracovat s dekorovanou funkcí.

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

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())
      }
    })
```

```
18      .then(_ => func())
19      .then(result => {
20        context.setOutput(result)
21        weaver.applyPostConditions(context)
22        return new Promise(
23            (resolve, reject) => resolve(context.getOutput())
24        )
25      })
26  }
27
28  exports.register = (name, email) => {
29    const context = new BusinessOperationContext('user.register')
30    context.setInputParameter('name', name)
31    context.setInputParameter('email', email)
32    return wrapCall(context, () => register(name, email))
33  }
```

5.6.2 Použité technologie

[20] 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* [27], *Babel* [3] a *Rimraf* [41].

5.7 Systém pro centrální správu byznys pravidel

[TODO

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

]

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

5.7.1 Popis implementace

BusinessContextEditor

5.7.2 Detekce a prevence potenciálních problémů

[21] Jak jsme již naznačili v sekci 4.2, při úpravě nebo vytváření nového byznysového kontextu je potřeba detekovat případné chyby, abychom změnou neuvedli systém do nefunkčního stavu. Kromě syntaktických chyb, které jsou detekovány automaticky pomocí definovaného schematu, je potřeba věnovat pozornost také sémantickým chybám. Závažné chyby, které mohou být způsobeny rozšiřováním kontextů, jsou

a) Neunikátní identifikátory byznysových pravidel

b) Závislosti na neexistujících kontextech

c) Cyklus v grafu závislostí kontextů

d) Vzájemná kontradikce preconditions

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

[23] Detekce závislosti na neexistujících kontextech je relativně jednoduchým úkolem. Nejprve setavíme seznam existujících kontextů a následně procházíme jednotlivé hrany grafu kontextů a ověřujeme, zda existují oba kontexty náležící dané hraně. Při zvolení vhodných datových struktur lze dosáhnout lineární složitosti v závislosti na počtu hran grafu.

[24] Pokud by závislosti v orientovaném grafu vytvořily cyklus, tedy kruhovou závislost kontextů, kteoru jsme představili v sekci 4.2, docházelo by při inicializaci služeb obsahující daná pravidla k zacyklení. Tomu můžeme předejít detekcí cyklů v grafu. Pro tuto detekci byl zvolen Tarjanův algoritmus [75] pro detekci souvislých komponent, který disponuje velmi dobrou lineární složitostí, závislou na součtu počtu hran a počtu uzlů grafu.

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

[21] [Intended Delivery: Problémy způsobené rozšiřováním kontextů]
[22] [Intended Delivery: Chápání kontextů jako grafu]
[23] [Intended Delivery: Detekce závislostí na neexistujících kontextech]
[24] [Intended Delivery: Detekce cyklů v grafu závislostí]
[25] [Intended Delivery: Reakce na chyby]

5.7.3 Použité technologie

Uživatelské rozhraní Pro komfortní obsluhu centrální administrace bylo naprogramováno uživatelské rozhraní pomocí technologií Hypertext Markup Language [7] (HTML) a Cascading Style Sheets [10] (CSS), které jsou již několik desetiletí standarderm pro tvorbu webových uživatelských rozhraní. Detail byznysového kontextu v uživatelském rozhraní můžeme vidět na snímku A.2 a formulář pro úpravu 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

]

5.8 Shrnutí

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

[27] Veškerý kód je hostován v centrálním repozitáři ve službě GitHub[28] a je zpřístupněn pod open-source licencí MIT [83]. Knihovny pro jednotlivé platformy tedy lze libovolně využívat, modifikovat a šířit.

[29] Protoypy knihoven lze využít k implementaci služeb, potažmo k sestavení funkčního systému, jak si ukážeme v následující kapitole.

[26] [Intended Delivery: Dosáhli jsme vytyčených cílů implementace]
[27] [Intended Delivery: Hostování na GitHubu + licence]
[28] https://github.com/klimesf/diploma-thesis
[29] [Intended Delivery: Validaci a verifikaci si ještě ukážeme]

Kapitola 6

Verifikace a validace

V této kapitole si popíšeme, jaký způsobem byla provedena verifikace naprogramovaných knihoven pomocí jednotkových a integračních testů [52] a také jak byly knihovny nasazeny při vývoji ukázkového systému. Tím zároveň zvalidujeme koncept frameworku a shrneme výhody a nevýhody jeho použití.

6.1 Testování prototypů knihoven

Prototypy knihoven, jejichž implementaci jsme popsali v kapitole 5, byly důkladně otestovány pomocí sady jednotkových a integračních testů a tím byla verifikována jejich správná funkcionalita. Způsob testování knihoven si popíšeme zvlášť pro každou platformu.

V rámci konceptu *continous integration* (CI) [35] byl kód po celou dobu vývoje verzován systémem Git [37], zasílán do centrálního repozitáře a s pomocí nástroje Travis CI[1] 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 [43], který poskytuje veškerou potřebnou funkcionalitu pro jednotkové i integrační testování. Všechny testy byly spouštěny automaticky při sestavování knihovny pomocí nástroje Maven [54].

[1] https://travis-ci.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 můžeme vidět jednotkový test třídy `BusinessContextWeaver` ověřující, že byly správně aplikovány post-conditions daného byznys kontextu, konkrétně že bylo správně zakryto 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` [1], inspirovaného nástrojem JUnit. Ačkoliv jméno obou nástrojů nasvědčuje, že slouží zejména

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

Zdrojový kód 6.2: 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* [57], zatímco o ověřování a zápis testů ve stylu *Behaviour Driven Development* (BDD) [71] se stará knihovna *Chai* [20].

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 distribuci byznys pravidel verifikovat v praxi a validovat jeho myšlenku, bylo nutné

vyzkoušet 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) [9]), 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)

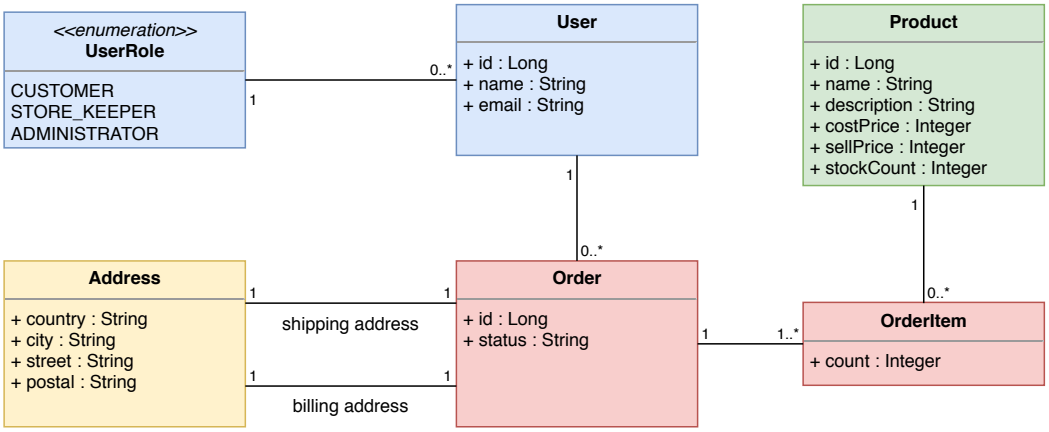
Tabulka 6.1: Přehled use-cases ukázkového e-commerce systému

6.2.2 Model systému

Na obrázku 6.1 můžeme vidět 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

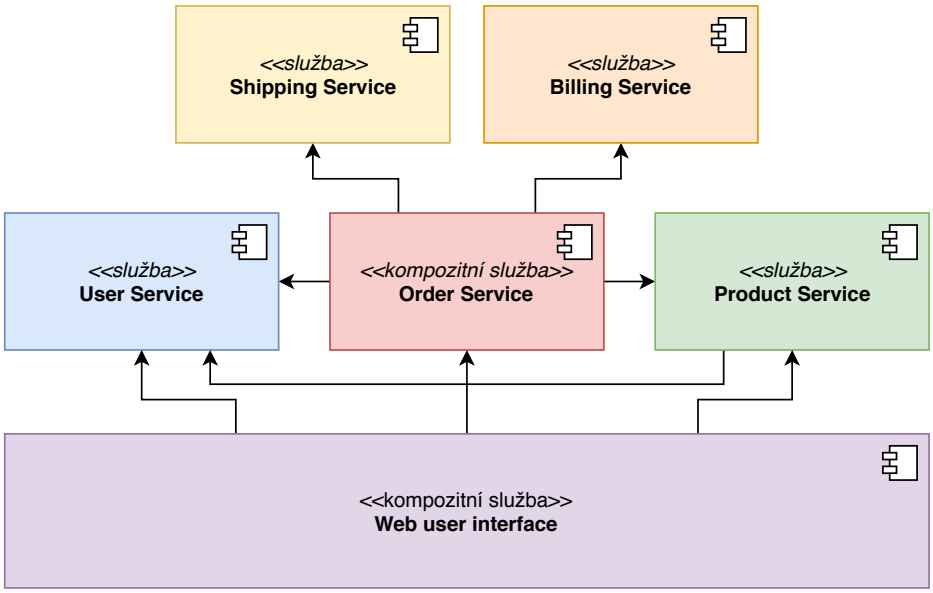
Na obrázku 6.2 jsou zobrazeny komponenty systému a jejich vzájemné závislosti. Pro ověření schopnosti podporovat více platforem byly pro implementaci systému využity jazyky

#	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 [73].

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 [73], jak můžeme vidět 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
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* [81]. Ve zdrojovém kódu 6.5 můžeme vidět 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)
```

Shipping service Služba *Shipping service* má na starosti funkcionalitu týkající se odesílání objednávek a byla implementována v jazyce Java s použitím frameworku Spring Boot [73].

User service Služba *User service* realizující funkcionalitu týkající se uživatelských účtů byla implementována v jazyce JavaScript na platformě Node.js s použitím frameworku Express [26]. 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 [73]. 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 [56], 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, dalšími mohou být jednotlivé knihovny instalované do systému. Příklad definice image pomocí technologie Docker můžeme vidět ve zdrojovém kódu 6.7. Konkrétně se jedná o definici

image, který rozšiřuje oficiální image `library/node:9.11.1` [49] stavějící nad operačním systémem *Linux* [50], a přidává vrstvy s prototypem knihovny našeho frameworku 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 byznysových 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 [5], 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'
services:
  order:
    image: filipklimes-diploma/example-order-service
```

#	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

```
5     ports:
6       - "5501:5501"
7       - "5551:5551"
8     depends_on:
9       - product
10      - billing
11      - shipping
12      - user
13    links:
14      - product
15      - user
16      - billing
17      - shipping
```

6.3 Srovnání s konvenčním přístupem

[2] Z tabulky 6.3 vidíme, 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 bychom museli tato pravidla 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.

[2] [Intended Delivery: Ukázka na konkrétním příkladě]

[3] Díky použití naší knihovny je však možné každé pravidlo nadefinovat centrálně a framework se postará o jeho automatickou distribuci do všech míst, kde je potřeba ho aplikovat. Díky tomu je možno byznysová pravidla, resp. kontexty, spravovat pomocí nástroje pro centrální správu, který je součástí našeho frameworku. Z toho vyplývá snížení nároků na vývoj a snížené riziko lidské chyby.

[4] Jako nevýhodu použití frameworku můžeme 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 hirearchii a vzájemnou vazbu s byznysovńi pravidly, aby bylo možno framework efektivně využívat. To může vyžadovat více času, než klasický návrh.

[5] 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 jsme zjistili, že v SOA lze efektivně řešit otázku byznysových pravidel, potažmo průřezových problémů obecně, námi navrženým způsobem.

6.4 Shrnutí

V této kapitole jsme popsali, jakým způsobem byly testovány prototypy knihoven pro platformy jazyků Java a Python a pro platformu Node.js. Tím jsme verifikovali jejich správnou fukcionalitu. Dále jsme naspecifikovali a popsali implementaci ukázkového systému, na kterém jsme provedli případovou studii použití našeho frameworku. Díky tomu jsme validovali, že navržený framework je funkční a splňuje požadavky identifikované v sekci 2.4. Nakonec jsme na ukázkovém systému diskutovali srovnání použití našeho frameworku a konvenčního přístupu k návrhu a implementaci softwarových systému.

[3] [Intended Delivery: Výhody našeho frameworku]
[4] [Intended Delivery: Nevýhody použití]
[5] [Intended Delivery: Závěr]

Kapitola 7

Závěr

7.1 Analýza dopadu použití frameworku

7.2 Budoucí rozšiřitelnost frameworku

7.2.1 Kvalitní doménově specifický jazyk

Zadáním této práce nebylo zkonstruovat vlastní DSL k účelům automatické distribuce a centrální správy byznys pravidel, nicméně v sekci 2.4 jsme potřebu takového jazyka jasně identifikovali a následně v kapitole 3 jsme došli k závěru, že momentálně neexistuje vhodné DSL, které by splňovalo všechny naše požadavky a mohli bychom ho využít pro naše účely. V rámci implementace prototypu knihoven jsme navrhli a implementovali vlastní DSL v jazyce XML, jak jsme popsali v sekci 5.3. Tento jazyk je však velmi omezený a snaží se vyhovět co nejnižším nárokům na implementaci. Sestavení kvalitního jazyka pro naše účely je tématem nejméně pro bakalářskou práci. Nicméně, námi navržený framework je schopen toto rozšíření pojmout, stačí doimplementovat plug-in, který se bude starat o převod z daného DSL do paměťové reprezentace byznysového kontextu.

Kvalitní jazyk by měl kromě výše zmíněných požadavků pro zachycení pravidla poskytovat co nejpřehlednější zápis, aby ho mohl snadno číst a zapisovat nejen vývojář, ale i doménový expert či administrátor systému. Tím by se ještě zvýšil přínos centrální administrace byznysových pravidel, kterou jsme v rámci této práce implementovali a popsali v sekci 5.7. Můžeme také diskutovat, že by jazyk pro popis byznysových kontextů sloužil pouze jako platforma a samotná pravidla by byla popsána v DSL vytvořeném na míru byznysové doméně, pro kterou by byl implementován systém využívající našeho frameworku.

7.2.2 Integrace frameworku s uživatelským rozhraním

V sekci 4.10 jsme nastínili způsob, jakým lze využívat náš framework. Jedním ze způsobů je integrace do uživatelského rozhraní. Autoři přístupu ADDA již vyvinuli způsob, kterým lze integrovat vyhodnocování byznysových pravidel v uživatelském rozhraní [18]. Propojení s naším frameworkem by znamenalo pouze implementovat adaptér, který by převáděl námi použitou reprezentaci byznysového pravidla do podoby, kterou je schopen využívat aspect weaver v UI. Tím by se rozšířila působnost našeho frameworku a zároveň by se zvýšil uživatelský komfort IS, který framework využívá, díky real-time validaci vstupních hodnot formulářů.

7.2.3 Integrace frameworku s datovou vrstvou

Jak jsme také zmínili v sekci 4.10, integrace do datové vrstvy je také jednou z možností. 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 SQL jazyce **??**. Aplikováním příslušného aspect weaveru by byl zvýšen dosah frameworku a byla by pokryta další oblast, ve které může docházet k manuální duplikaci byznysových pravidel.

7.3 Možností uplatnění navrženého frameworku

7.4 Další možnosti uplatnění AOP v SOA

[TODO

- Extrakce dokumentace
- Extrakce byznysového modelu
- Konfigurace prostředí

]

7.5 Shrnutí

[TODO

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

]

Literatura

[1] *26.4. unittest — Unit testing framework – Python 3.6.5 documentation* [online]. Dostupné z: <https://docs.python.org/3/library/unittest.html>.

[2] *Apache Thrift - Home* [online]. Dostupné z: <https://thrift.apache.org/>.

[3] *Babel · The compiler for writing next generation JavaScript* [online]. Dostupné z: <https://babeljs.io/>.

[4] BECK, K. *Extreme programming explained: embrace change*. Boston, Massachusetts, USA : Addison-Wesley Professional, 2000.

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

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

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

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

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

[10] 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.

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

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

[13] *business-rules – PyPi* [online]. Dostupné z: <https://pypi.org/project/business-rules/>.

[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. – 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.

[20] *Chai* [online]. Dostupné z: <http://www.chaijs.com/>.

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

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

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

[24] *Drools - Drools - Business Rules Management System (Java™, Open Source)* [online]. Dostupné z: <https://www.drools.org/>.

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

[26] *Express – Node.js web application framework* [online]. Dostupné z: <https://expressjs.com/>.

[27] *Fast, reliable, and secure dependency management.* [online]. Dostupné z: <https://yarnpkg.com/en/>.

[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. et al. Hypertext transfer protocol–HTTP/1.1. Technical report, 1999.

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

[31] 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.

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

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

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

[37] *Git* [online]. Dostupné z: <https://git-scm.com/>.

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

[39] *gRPC open-source universal RPC framework* [online]. Dostupné z: <https://grpc.io/>.

[40] *IBM Knowledge Center - JRules engine mode* [online]. Dostupné z: <https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_11.3.0/com.ibm.swg.im.iis.conn.jrules.use.doc/topics/c_jrules_enginemode.html>.

[41] *isaacs/rimraf* [online]. Dostupné z: <https://github.com/isaacs/rimraf>.

[42] *JDOM* [online]. Dostupné z: <http://www.jdom.org/>.

[43] *JUnit - About* [online]. Dostupné z: <https://junit.org/junit4/>.

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

[45] 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.

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

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

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

[49] *library/node - Docker Hub* [online]. Dostupné z: <https://hub.docker.com/_/node/>.

[50] *Linux – The Linux Foundation* [online]. Dostupné z: <https://www.linuxfoundation.org/projects/linux/>.

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

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

[53] 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.

[54] *Maven – Welcome to Apache Maven* [online]. Dostupné z: <https://maven.apache.org/>.

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

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

[57] *Mocha – the fun, simple, flexible JavaScript test framework* [online]. Dostupné z: <https://mochajs.org/>.

[58] *MPS: Domain-Specific Language Creator by JetBrains* [online]. Dostupné z: <https://www.jetbrains.com/mps/>.

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

[60] *Orchestration vs. Choreography* [online]. Dostupné z: <https://stackoverflow.com/questions/4127241/orchestration-vs-choreography>.

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

[62] *pip - pip 9.0.3 documentation* [online]. Dostupné z: <https://pip.pypa.io/en/stable/>.

[63] POSTEL, J. Transmission control protocol. 1981.

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

[65] *Protocol Buffers | Google Developers* [online]. Dostupné z: <https://developers.google.com/protocol-buffers/>.

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

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

[68] SCOTT, M. L. *Programming language pragmatics*. Burlington, Massachusetts, USA : Morgan Kaufmann, 2000.

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

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

[71] 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.

[72] 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.

[73] *Spring Boot* [online]. Dostupné z: <https://projects.spring.io/spring-boot/>.

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

[75] 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.

[76] *The Kiss Principle* [online]. Dostupné z: <http://people.apache.org/~fhanik/kiss.html>.

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

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

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

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

[81] *Welcome | Flask (A Python Microframework)* [online]. Dostupné z: <http://flask.pocoo.org/>.

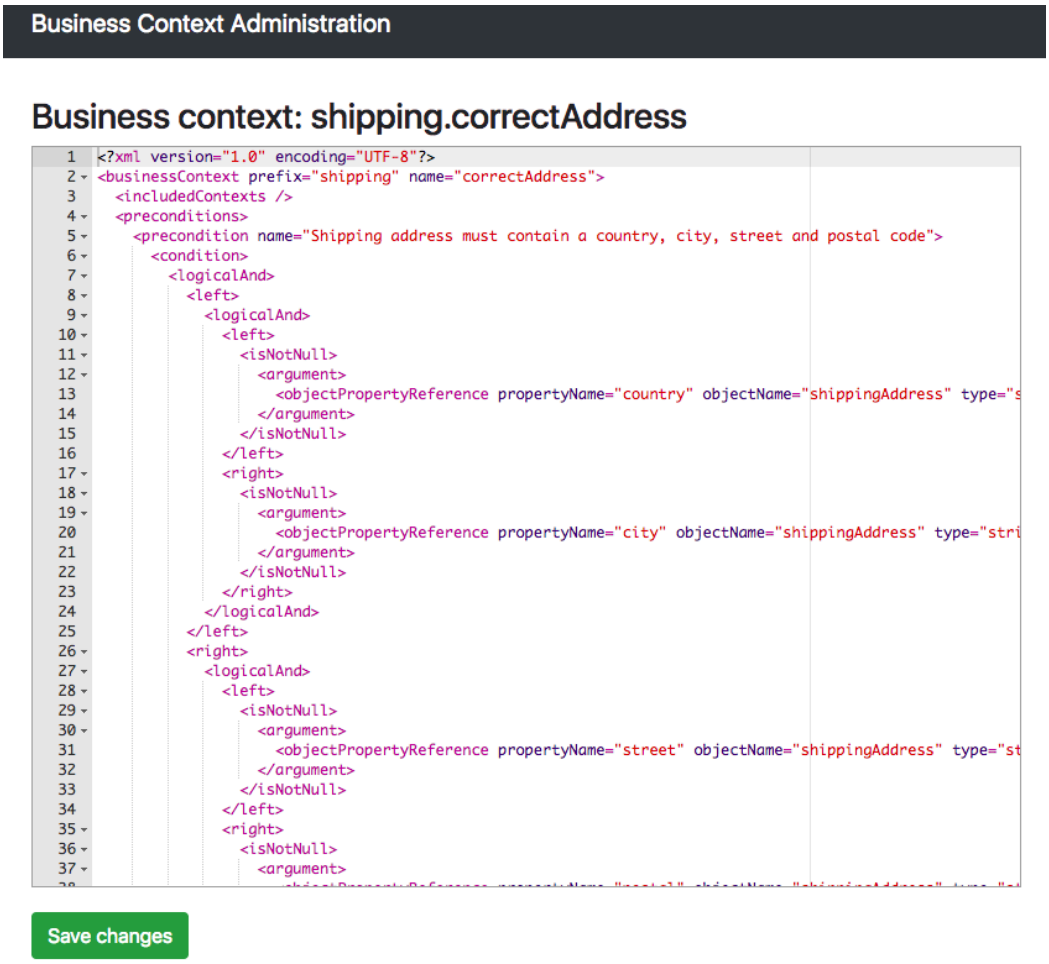
[82] *Welcome to Apache Avro!* [online]. Dostupné z: <https://avro.apache.org/>.

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

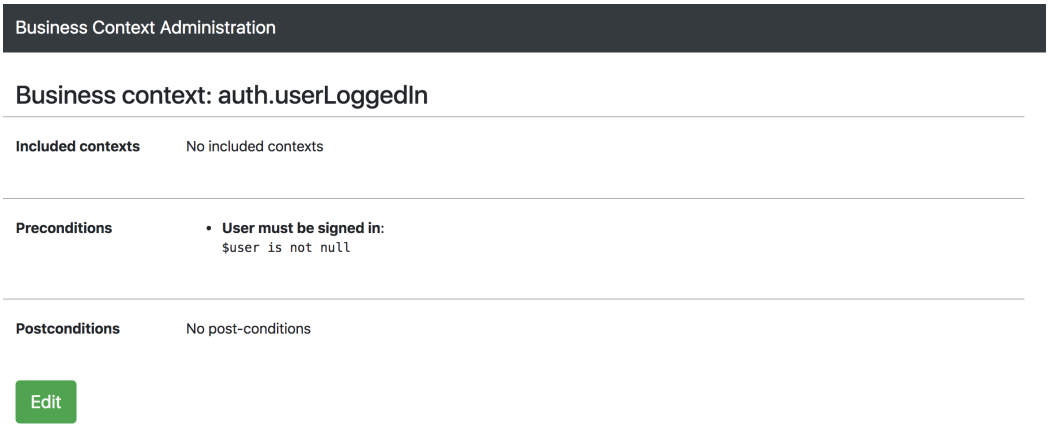
[84] WOOD, L. et al. Document Object Model (DOM) level 3 core specification, 2004.

Příloha A

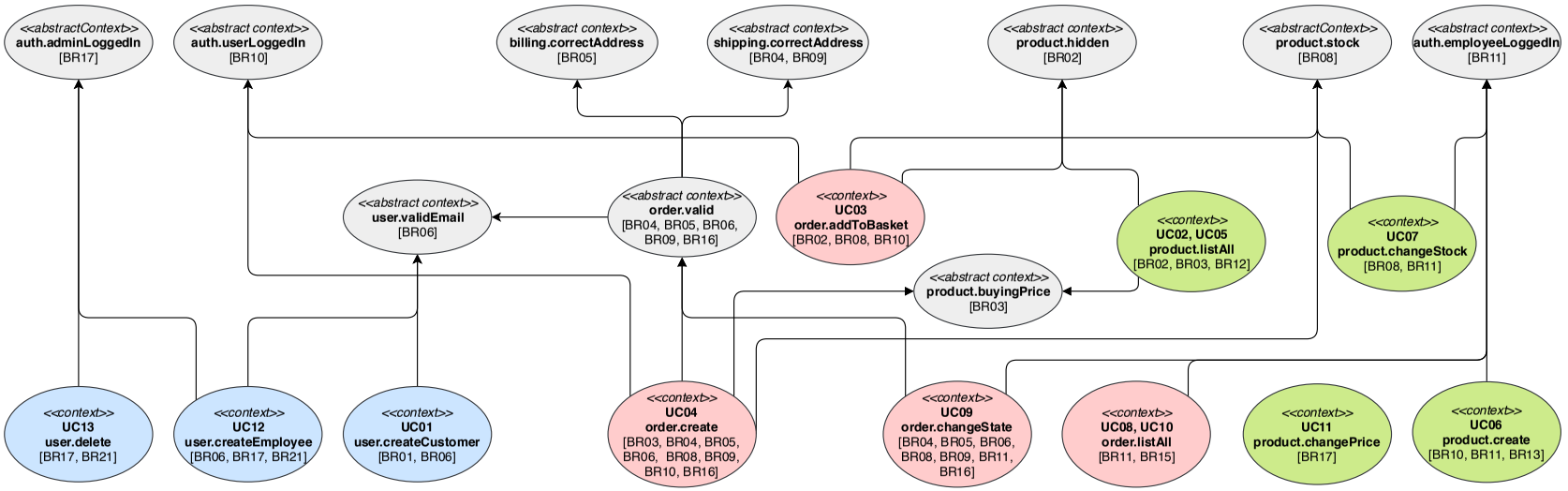
Přehledové obrázky a snímky



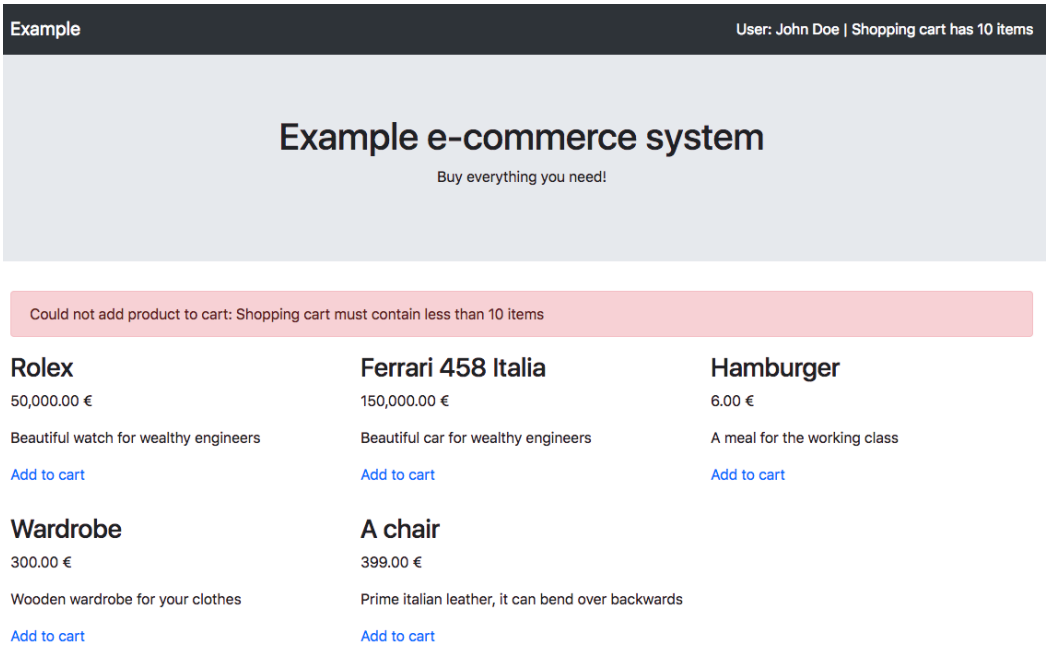
Obrázek A.1: Formulář pro vytvoření nebo úpravu byznysového kontextu v centrální administraci



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

Uživatelská příručka

[TODO

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

]

Příloha C

Seznam použitých zkratek

ADDA	Aspect-Driven Design Approach. 15, 25–27, 31, 42, 49, 80
AOP	Aspect Oriented Programming. xiii, 23–25, 31, 49, 80
API	Application Programming Interface. 9, 19, 20, 73, 74, 76
AST	Abstract Syntax Tree. 55
BDD	Behaviour Driven Development. 67
BRMS	Business Rules Management System. 26, 27
CI	Continuous Integration. 65
CORBA	Common Object Request Broker Architecture. 6, 20
CRUD	Create, Read, Update, Delete. 18
CSS	Cascading Style Sheets. 63
DAG	Directed Acyclic Graph. 37
DOM	Document Object Model. 58
DSL	Domain-Specific Language. xii, xv, 15, 25–29, 42–44, 46, 49, 54, 58, 63, 78, 79
EL	Expression Language. 4
ESB	Enterprise Service Bus. 7–9, 11
HATEOAS	Hypermedia as the engine ot application state. 19, 20
HTTP	Hypertext Transfer Protocol. 6, 18–20
IS	Informační systém. 3, 15, 25, 31, 80

Java EE Java Platform, Enterprise Edition. 26
JPQL Java Persistence Query Language. 25
JSON JavaScript Object Notation. 6, 19, 52, 73
JSR Java Specification Request. 33, 40

KISS Keep it simple, stupid. 44

LHS Left-hand side. 27
LOP Language-Oriented Programming. 28, 29

MDA Model-Driver Architecture. 15, 16, 29
MQ Message Queue. 7

NAT Native Address Translation. 45

OMG Object Modeling Group. 15
OOP Object Oriented Programming. 16, 22–24
ORB Object Request Broker. 6

P2P Peer-to-peer. 17, 18, 45
PIM Platform Independent Model. 15, 16
PSM Platform Specific Model. 15

REST Representational State Transfer. xv, 6, 18–21, 73, 74, 76
RHS Right-hand side. 27
RMI Remote Method Invocation. 20
RPC Remote Procedure Call. xv, 20, 21, 53

SOA Service Oriented Architecture. xiii, 5–8, 10, 11, 15, 20, 29, 31, 46, 49, 69, 78, 80
SOAP Simple Object Access Protocol. 6
SQL Structured English Query Language. 25, 80

TCP Transmission Control Protocol. 16

UC Use Case. 69, 70, 74
UI User Interface. 10, 25, 26, 75, 80
UML Unified Modeling Language. 15
URI Uniform Resource Identifier. 19

URL Uniform Resource Locator. 16, 75

WSDL Web Service Description Language. 6

XML Extensible Markup Language. xix, 6, 19, 52, 54, 55, 58, 79

XSD XML Schema Definition. 55

YAGNI You aren't gonna need it. 44

YAML YAML Ain't Markup Language. 76

Příloha D

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
```