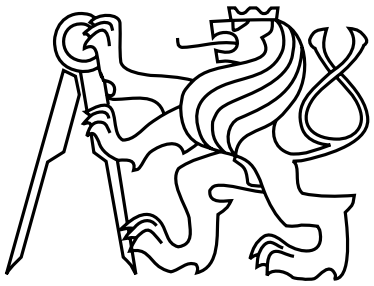
Na tomto místě bude oficiální zadání vaší práce

- 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

13. dubna 2018

Poděkování

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

Prohlášení

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

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

Abstract

Translation of Czech abstract into English.

Abstrakt

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

Obsah

Seznam obrázků

Seznam zdrojových kódů

Kapitola 1

Úvod

[TODO

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

]

1.1 Jak číst tuto práci

[1] Kapitola 2 se venuje detailní analyze 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 zabyvá rešerší stávajících prístupu k vyvoji 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

[1] [Intended Delivery: Popis struktury DP a obsah kapitol]

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. Dále 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 [7].

Byznysová pravidla dělíme do tří skupin [6]:

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* [7].

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 [7]. 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 [8]. Prvním je *aplikační kontext* (z anglického *application context*), ve kterém je uložen stav globálncíh 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 [7]. 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 [17]. 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 [9]. 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]

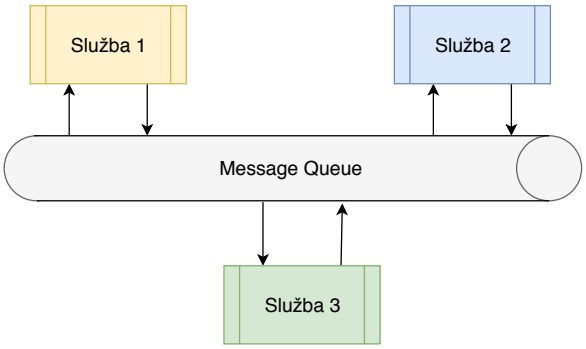
SOA je, popíšeme si její historický vývoj a shrneme výhody a nevýhody jednotlivých přístupů.

2.2.1 Common Object Request Broker Architecture

Prvním historickým předchůdcem architektury orientované na služby byla tzv. *Common Object Request Broker Architecture* (CORBA) [36], 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) [4], který v kombinaci s *Web Service Description Language* (WSDL) [11] umožňuje kompletní definici rozhraní pro komunikaci mezi službami. V průběhu dalších let vznikla také velmi populární architektura *Representational State Transfer* (REST) [15], 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

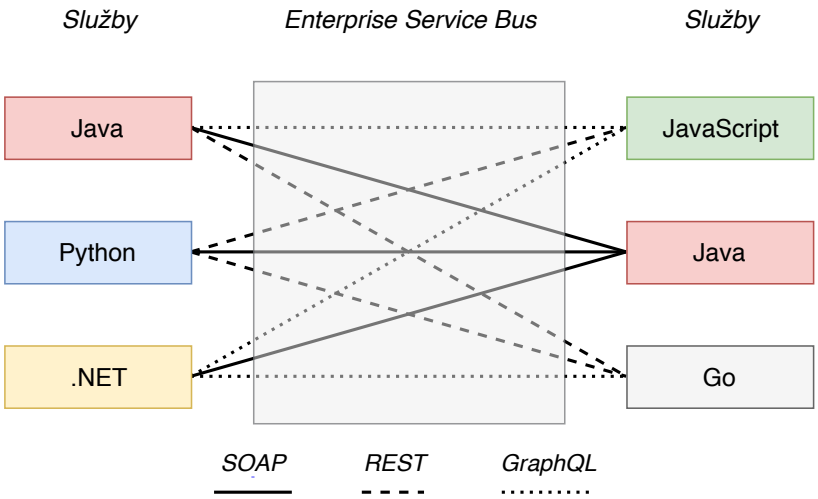
Dalším z konceptů, který v rámci SOA vznikl, je tzv. *Message Queue* (MQ). Základní myšlenkou MQ, znázorněnou na obrázku 2.1, je asynchronní komunikace služeb pomocí zpráv nezávislých na platformě. Komunikaci zprostředkovává fronta, která přijímá a rozesílá zprávy mezi službami. To přináší vyšší škálovatelnost 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) [10], 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.

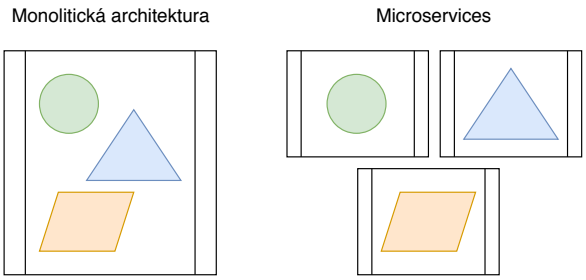
2.2.5 Microservices

[3] Novým trendem posledních let jsou takzvané *Microservices*. Přináší několik zajmavých konceptů, které specializují a konkretizují principy SOA. Microservices se tedy dají chápat jako podmnožina SOA. Základní myšlenkou je vývoj informačního systému jako množiny malých oddělených služeb, které jsou spouštěny v samostatných procesech a komunikují spolu pomocí jednoduchých protokolů [27].

[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 [27]. Monolitická architektura vyžaduje i při malé změně jedné části celý systém znovu zkompilovat, sestavit

[3] [Intended Delivery: Microservices a budoucnost SOA]
[4] [Intended Delivery: Stavba služeb kolem byznysových schopností]
[5] Zde předpokládáme klasickou třívrstvou architekturu [16], 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

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.

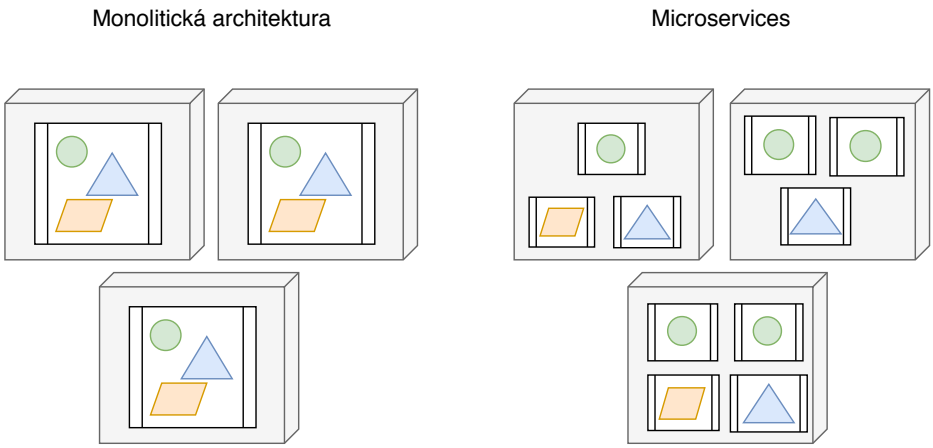
[7] Microservices také přinášejí koncept „smart endpoint, dumb pipes“, který opouští koncept ESB ve prospěch přesunutí veškeré byznys logiky na stranu služeb. Tím se zvyšuje zapouzdřenost služeb a snižuje se jejich vzájemné provázání. Nutno podotknout, že microservices často využívají ke své funkci Message 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 praktikou v rámci SOA je naopak sdílení jedné databáze mezi více službami, což je často dáno

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



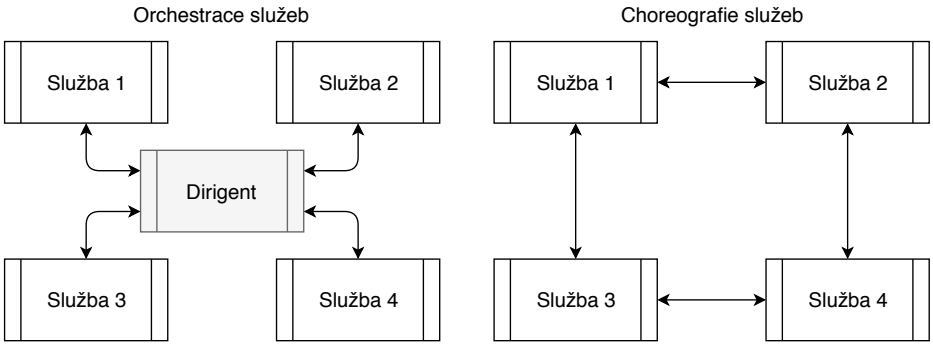
Obrázek 2.4: Porovnání nasazení monolitické architektury a microservices

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 [27].

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 [39], 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 vytvoření objednávky a ten se již postará o komunikaci tohoto požadavku všem službám za-



Obrázek 2.5: Porovnání orchestrace a choreografie služeb

pojeným 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 [12], 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 [31].

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 [9], tzv. *průřezových problémů* (z anglického *cross-cutting concerns*). Příkladem mohou být právě byznysová pravidla [6], 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.

[10] Abychom si mohli lépe představit diskutovaný problém, znázorněmě si ho na konkrétním příkladu. Uvažme e-commerce systém skládající se z několika služeb naprogramovaných

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

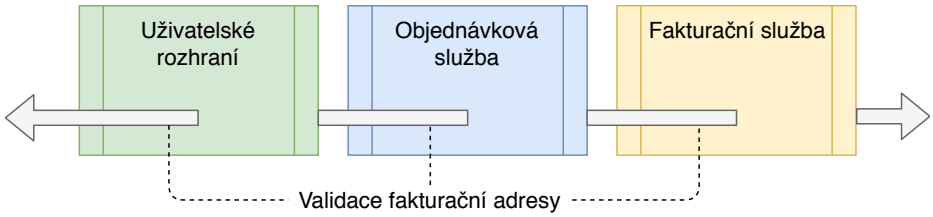
v různých technologiích, organizovaných kolem jeho byznysových funkcí. Jedna služba obsluhuje byznysové operace vázající se na uživatele systému, jejich registraci a administraci. Druhá služba realizuje operace s produkty, jejich vytváření, úpravu, správu skladových zásob a informace o dostupnosti. Třetí 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 [8]. 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 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

[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

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
- Automatickou distribuci kontextů, vyhodnocování jejich preconditions a aplikaci post-conditions
- Možnost využívat framework na více plaformách

[14] [Intended Delivery: Shrnutí problémů]

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

3.1 Modelem řízená architektura

[TODO

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

]

3.2 Architektura klient-server

[TODO

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

]

3.3 Aspektově orientované programování

[1] Programování je komplexní disciplína s teoreticky neomezeným počtem možností, jakým programátor může řešit zadaný problém. Ačkoliv každá úloha má své specifické požadavky, za relativně krátkou historii programování se stihlo ustálit několik ideologií, tzv. programovacích paradigmat, které programátorovi poskytují sadu abstrakcí a základních principů [40]. Díky znalosti paradigmatu může programátor nejen zlepšit svou produktivitu, ale zároveň může snáze pochopit myšlenky jiného programátora a tím zlepšit kvalitu týmové spolupráce.

[2] Jedním z nejpopulárnějších paradigmat používaných k vývoji moderních enterprise systémů je nepochybně objektově orientované programování (OOP). To vnímá daný problém jako množinu objektu, které spolu intereagují. Program člení na malé funkční celky odpovídající struktuře reálného světa [35]. 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.

[3] Ačkoliv je OOP velmi silným a všestraným nástrojem, existují problémy, které nelze jeho pomocí efektivně řešit. Jedním takovým problémem jsou obecné požadavky na systém, které musejí být konzistentně dodržovány na více místech, které spolu zdánlivě nesouvisí. Příkladem může být logování systémových akcí, optimalizace správy paměti nebo uniformní zpracování výjimek [25]. 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 [18].

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

[1] [Intended Delivery: Co je paradigma]
[2] [Intended Delivery: OOP a jeho popis]
[3] [Intended Delivery: Nedostatky OOP]
[4] [Intended Delivery: AOP jako odpověď na nedostatky OOP]

[TODO

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

]

Aspekt

Join-point

Pointcut

Advice

Weaving

3.4 Aspect-driven Design Approach

[TODO

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

]

Aspect-driven Design Approach (ADDA)

[5] Vzhledem k požadavkům na implementaci našeho frameworku stanoveným v předchozí kapitole 2 se AOP a na něm stavějící ADDA jeví jako vhodný přístup, který nám pomůže dosáhnout cíle.

[5] [Intended Delivery: Vhodnost AOP pro náš úkol]

3.5 Stávající řešení reprezentace business pravidel

3.5.1 Drools DSL

[TODO

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

]

[6] ...

3.5.2 JetBrains MPS

[TODO

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

]

[7] ...

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

3.6 Shrnutí

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

Kapitola 4

Návrh

4.1 Formalizace architektury orientované na služby

4.1.1 Join-points

4.1.2 Advices

4.1.3 Pointcuts

4.1.4 Weaving

4.2 Architektura frameworku

4.3 Zachycení byznysového konextu

Přístup ADDA doporučuje popsat byznysová pravidla pomocí vlastního, na míru šitého, doménově specifického jazyka [7]. V našem případě můžeme jazykem DSL popsat kompletně i celý byznysový kontext.

4.4 Metamodel byznys kontextu

4.5 Expression

4.6 Registr byznys kontextů

4.7 Byznys kontext weaver

4.8 Centrální správa byznys kontextů

4.8.1 Uložení rozšířeného pravidla

[1]

4.9 Service discovery

[2] [T1]fontenc [utf8]inputenc

[1] [Intended Delivery: Diskutovat chaining vs. direct update]
[2] [Intended Delivery: Popsat nezávislost na service discovery]

Kapitola 5

Implementace prototypů knihoven

[1] Součástí zadání této práce je implementace prototypů knihoven pro framework navržený v kapitole 4 pro tři rozdílné platformy, z nichž jedna musí být *Java*. V této kapitole si popíšeme, jaké plaformy jsme vybraly, a jakým způsobem byly prototypy knihoven implementovány. Součástí kapitoly je i stručná rešerše technologií, které byly použity pro dosažení vytyčených cílů.

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

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

5.1 Výběr použitých platforem

[4] Mimo jazyk Java, který byl určen zadáním, byla pro implementaci vybrána platforma jazyka *Python* a platforma *Node.js*, který slouží jako běhové prostředí pro jazyk *JavaScript*. Výběr byl proveden na základě aktuálních trendů ve světě softwarového inženýrství. Projekt GitHut [33] 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 [20]

[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 [37] 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. 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* [34][41] 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 [28]. Oproti protokolům *Apache Thrift* [1] a *Apache Avro* [42], které poskytují velmi srovnatelnou funkcionalitu, mají Protocol Buffers kvalitnější a lépe srozumitelnou dokumentaci.

Zdrojový kód 5.1 znázorňuje část zápisu schématu zasílaných zpráv obsahující byznys kontexty ve formátu Protobuffer. Schéma zpráv pro výměnu kontextů opisuje strukturu metamodelu navrženého v sekci 4.4.

ExpressionMessage obsahuje jméno, atributy a argumenty `Expression`

ExpressionPropertyMessage je enumerace obsahující typy atributu `Expression`

PreconditionMessage obsahuje název a podmínku precondition pravidla

[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/

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

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

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

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

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

PostConditionTypeMessage je enumerace obsahující typy post-condition pravidla

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

BusinessContextsMessage obaluje více byznys kontextů

5.2.2 gRPC

[10] Pro realizaci architektury klient-server byl zvolen open-source framework gRPC [21], který staví na technologii Protocol Buffers a poskytuje vývojáři možnost definovat detailní schéma komunikace pomocí protokolu *RPC* [30]. 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) [5]. Tento jazyk umožňuje serializaci libovolných dat, přímočarý a formální zápis jejich struktury a také jejich snadné aplikační zpracování. Zároveň poskytuje relativně dobrou čitelnost pro člověka, ačkoliv speciálně vytvořené DSL by bylo jistě čitelnější.

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

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

text a nebo speciálních značek, neboli *markup*, které slouží k popisu struktury dat. Naopak neparsovaná data mohou obsahovat libovolné znaky, které nenesou žádnou informaci o struktuře dat.

[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 [26], 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.

5.4 Knihovna pro platformu Java

[TODO

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

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

Zdrojový kód 5.3: Příklad zápisu byznys kontextu v jazyce XML

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

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

```
public class OrderService {

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

}
```

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

]

5.4.1 Popis implementace

BusinessContextRegistry

5.4.2 Použité technologie

Apache Maven [17] Pro správu závislostí a automatickou kompilaci a sestavování knihovny napsané v jazyce java byl zvolen projekt *Maven*. Tento nástroj umožňuje vývojáři komfortně a centrálně spravovat závislosti jeho projektu včetně detailního popisu jejich verze. Dále také umožňuje definovat jakým způsobem bude projekt kompilován.

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

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

]

JDOM 2 [19] Knihovna JDOM 2 [23] poskytuje kompletní sadu nástroju pro čtení a zápis XML dokumentů. Implementuje specifikaci *Document Object Model* (DOM) [44], 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* [32] 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

]

[18] [Intended Delivery: Proč AspectJ a co to umí]
[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
```

5.5.1 Srovnání s knihovnou pro platformu Java

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

5.5.2 Použité technologie

5.6 Knihovna pro platformu Node.js

Knihovna pro platformu *Node.js* byla implementována v jazyce JavaScript, konkrétně jeho verzi ECMAScript 6.0 [13]. 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* [24] poněkud složitější, podařilo se weaving implementovat spolehlivě. Ukázku můžeme vidět ve zdrojovém kódu 5.6. Funkce `register()` obsahuje logiku pro registraci uživatele, která může obsahovat například uložení entity do databáze a odeslání registračního e-mailu. Při exportování funkce z Node.js modulu využijeme `wrapCall()`, která má za úkol dekorovat předanou funkci `func`, před jejím zavolání vyhodnotit preconditions a po zavolání aplikovat post-conditions. Díky tomu bude každý kód, který využije modul definující funkci pro registraci uživatele, pracovat s dekorovanou funkcí.

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

5.6.2 Použité technologie

[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* [14], *Babel* [2] a *Rimraf* [22].

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

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

```
weaver = new BusinessContextWeaver(registry)

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

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

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

5.7 Systém pro centrální správu byznys pravidel

[TODO

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

]

5.7.1 Popis implementace

BusinessContextEditor

5.7.2 Detekce a prevence potenciálních problémů

[21] Při úpravě nebo vytváření nového byznysového kontextu je potřeba detekovat případné chyby, abychom změnou neuvedli systém do nekonzistentního stavu. Kromě syntaktických chyb, které jsou detekovány automaticky pomocí definovaného schematu, je potřeba věnovat pozornost také sémantickým chybám. Závažné chyby, které mohou být způsobeny rozšiřováním kontextů, jsou

a) Závislosti na neexistujících kontextech

b) Cyklus v grafu závislostí kontextů

[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, docházelo by při inicializaci služeb obsahující daná pravidla k zacyklení. Tomu můžeme předejít detekcí cyklů v grafu.

[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]

Pro tuto detekci byl zvolen Tarjanův algoritmus [38] 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.

5.7.3 Použité technologie

[TODO

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

]

5.8 Shrnutí

[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 [43]. 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. [T1]fontenc [utf8]inputenc

[25] [Intended Delivery: Reakce na chyby]
[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 ...

6.1 Testování prototypů knihoven

Prototypy knihoven, jejichž implementaci jsme popsali v kapitole 5, byly také důkladně otestovány pomocí sady jednotkových a integračních testů.

V rámci konceptu *continous integration* [19] byl kód po celou dobu vývoje zasílán do centrálního repozitáře a s pomocí nástroje Travis CI[1] bylo automaticky spouštěno jeho sestavení a otestování. Systém zároveň okamžitě informoval vývojáře o jejich výsledcích. To umožnilo v krátkém časovém horizontu identifikovat konkrétní změny v kódu, které do programu vnesly chybu. Tím byla snížena pravděpodobnost regrese a dlouhodobě se zvýšila celková kvalita kódu.

6.1.1 Platforma Java

Prototyp knihovny pro platformu byl testován pomocí nástroje JUnit[2], který poskytuje všechny potřebné funkce.

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

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

```
from flask import Flask, jsonify

app = Flask(__name__)
product_repository = ProductRepository()

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

6.1.2 Platforma Python

6.1.3 Platforma Node.js

6.2 Případová studie: e-commerce systém

6.2.1 Model systému

6.2.2 Use-cases

6.2.3 Byznys kontexty

6.2.4 Service discovery

6.2.5 Order service

6.2.6 Product service

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

6.2.7 User service

6.2.8 Nasazení systému pro centrální správu byznys kontextů

6.2.9 Orchestrace služeb

[3] [29]

6.3 Shrnutí

[T1]fontenc [utf8]inputenc

[3] [Intended Delivery: Spouštění pomocí Docker]

Kapitola 7

Závěr

7.1 Analýza dopadu použití frameworku

7.2 Budoucí rozšiřitelnost frameworku

7.3 Možností uplatnění navrženého frameworku

7.4 Další možnosti uplatnění AOP v SOA

[TODO

- Extrakce dokumentace
- Extrakce byznysového modelu
- Konfigurace prostředí

]

7.5 Shrnutí

[TODO

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

]

Literatura

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

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] 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.

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

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

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

[28] 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.

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

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

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

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

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

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

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

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

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

[38] 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.

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

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

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

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

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

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

Příloha A

TODO Screenshots

[T1]fontenc [utf8]inputenc

Příloha B

Seznam použitých zkratek

ADDA Aspect-Driven Design Approach

API Application Programming Interface

AST Abstract Syntax Tree

CORBA Common Object Request Broker Architecture

DOM Document Object Model

DSL Domain-Specific Language

EIS Enterprise Information System

EL Expression Language

ESB Enterprise Service Bus

HTTP Hypertext Transfer Protocol

IDL Interface Description Language

IS Informační systém

JSON JavaScript Object Notation

MDA Model-Driver Architecture

MQ Message Queue

ORB Object Request Broker

OOP Objektově Oriented Programming (Objektově orientované programování)

REST Representational State Transfer

RPC Remote Procedure Call

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

UI User Interface

WSDL Web Service Description Language

XML Extensible Markup Language

XSD XML Schema Definition

[T1]fontenc [utf8]inputenc

Příloha C

Obsah přiloženého CD

```
|-- nutfroms-example/       Ukázkový systém využívající knihovnu
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- src/                 Zdrojový kód aplikace
|
|-- nutforms-ios-client/    Klientská část knihovny pro platformu iOS
|  |-- client/              Zdrojové soubory knihovny
|  |-- clientTests/         Zdrojové soubory testů knihovny
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|
|-- nutfroms-server/        Serverová část knihovny
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- layout/              Layout servlet
|  |-- localization/        Localization servlet
|  |-- meta/                Metadata servlet
|  |-- widget/              Widget servlet
|
|-- nutforms-web-client/    Klientská část knihovny pro webové aplikace
|  |-- dist/                Zkompilované zdrojové soubory pro distribuci
|  |-- docs/                Dokumentace
|  |-- src/                 Zdrojové soubory knihovny
|  |-- test/                Zdrojové soubory testů knihovny
|
|-- text/                   Text bakalářské práce
```