VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

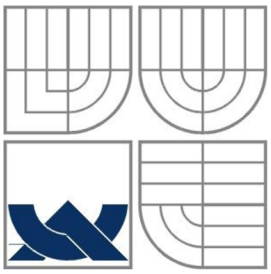
POPIS CHOREOGRAFIE SLUŽEB V ARCHITEKTUŘE SOA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

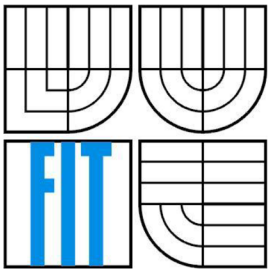
AUTOR PRÁCE
AUTHOR

MARTIN UŘÍDIL

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

POPIS CHOREOGRAFIE SLUŽEB V ARCHITEKTUŘE SOA

CHOREOGRAPHY DESCRIPTION IN SERVICE-ORIENTED ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MARTIN UŘÍDIL

VEDOUCÍ PRÁCE
SUPERVISOR

Mgr. MAREK RYCHLÝ, Ph.D.

BRNO 2011

Abstrakt

Tato práce se zabývá architekturou orientovanou na služby a popisem choreografií služeb. Je zde uveden podrobný přehled dostupných prostředků pro popis choreografií. Zejména se pak práce soustředí na využití formálních popisů při návrhu architektury. Praktická část práce ukazuje celý životní cyklus návrhu SOA a její implementaci v prostředí Microsoft .NET.

Abstract

This work is focused on service-oriented architecture and service choreography description. There is a discussion on various means for choreography description. This work especially emphasizes usage of formal description within architecture design. Practical part of the work brings out full life cycle of SOA design and its implementation in Microsoft .NET environment.

Klíčová slova

SOA, choreografie služeb, π-kalkul, WS-CDL, formální popis choreografie, implementace SOA v .NET

Keywords

SOA, choreography of services, π-calculus, WS-CDL, formal choreography description, implementation of SOA in .NET

Citace

Uřídil Martin: Popis choreografie služeb v architektuře SOA, bakalářská práce, Brno, FIT VUT v Brně, 2011

Popis choreografie služeb v architektuře SOA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Mgr. Marka Rychlého, Ph.D.

Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……

Martin Uřídil

1. 5. 2011

Poděkování

Chtěl bych poděkovat vedoucímu mé práce Mgr. Markovi Rychlému, Ph.D., který mi v průběhu tvorby práce poskytoval velmi užitečné rady a konzultace.

Obsah

1 Úvod

Spolu s rozvojem obchodních společností a velkých organizací v 90. letech rostly také požadavky na jejich softwarové vybavení. Důležitým cílem bylo propojování podniků a automatizace obchodních procesů za účelem jejich urychlení a usnadnění. Ke konci 90. let začalo díky rychlému rozvoji internetu pronikat mezi veřejnost elektronické obchodování. Internet tak přestal být pouhým skladištěm dokumentů a stal se aktivní infrastrukturou, využívanou různými subjekty k obousměrné komunikaci. Internet se tedy stal komunikační dálnicí, kde různé subjekty poskytují různé služby. Pravý význam však termín „služba“ získal až začátkem nového tisíciletí, kdy byl standardizován formát výměny zpráv a společnosti si začaly uvědomovat tento nový pohled na software poskytovaný jako služba. Tyto události stejně jako standardizace výměny zpráv mezi službami a přijetí univerzálního registru služeb UDDI velkou měrou přispěly ke vzniku architektury orientované na služby.

Architektura orientovaná na služby (dále SOA nebo také servisně orientovaná architektura) získala důležité postavení a nabývá v dnešním světě informačních technologií stále většího významu. Toto pojetí informační architektury přispívá ke zvýšení flexibility podniku a zkrácení odezvy na změny ve fyzickém systému. Vzhledem k tomu, že SOA nestaví na jedné technologii, ale pouze na standardech a spolupráci služeb, usnadňuje výměnu informací mezi různými organizacemi, napříč používanými platformami. I to je jedním z důvodů, proč mnoho dnešních společností na tento přístup přechází (nebo již přešlo) a těží z jeho výhod.

Jak již název napovídá, SOA je informační architektura sestávající z jednotlivých služeb, které mezi sebou komunikují pomocí zpráv. Služby s jednoduchou funkcionalitou jsou komponovány do větších celků za účelem plnění složitějších úkolů. Řízení komunikace služeb není složité, jsou-li všechny služby vlastněné jednou organizací. Jak ale řešit situace, kdy spolu komunikují dvě a více organizací, každá prostřednictvím vlastních služeb? Jak delegovat práva řízení? Na toto téma se v práci zaměříme, rozebereme možnosti testování takových kompozic ještě před jejich implementací a v praxi si ukážeme celý vývojový cyklus SOA na demonstračním příkladu.

Cílem této práce je představit servisně orientovanou architekturu z pohledu využití napříč různými organizacemi. Ukázat jaký přístup ke kompozici služeb je v takovém případě nejvhodnější a jak různé popisy kompozice služeb usnadňují vývojový cyklus. Práce okrajově zabrousí do tématiky testování a validace kompozic služeb. Celou teorii převedeme do praxe při návrhu a implementaci ukázkové architektury elektronického obchodu. Ten bude komunikovat se svými dodavateli a zákazníky skrze vystavené servisně orientované rozhraní.

Práce je rozdělená na šest hlavních kapitol. První část práce pojednává spíše o teoretické stránce problematiky, druhá používá teoretické poznatky při návrhu a implementaci reálné architektury. Teoretické kapitoly 2 až 4 čtenáře postupně provedou od obecných definic SOA jako celku až po konkrétní náplň tématu, kterou je popis choreografie služeb. Kapitoly 5 a 6 popisují návrh a implementaci SOA v prostředí v Microsoft .NET.

2 SOA

2.1 Definice SOA

Jak uvádí Thomas Erl ve své knize [1], servisně orientovaná architektura je pouze jedním z prvků, které dohromady tvoří servisně orientovanou výpočetní platformu. Všemi prvky, které ji tvoří, tedy jsou:

- Servisně orientovaná architektura
- Servisní orientace
- Servisně orientovaná logika řešení
- Služby
- Kompozice služeb
- Soupis služeb

Pro zjednodušení pojmů budeme ve většině případů používat termín SOA ve smyslu komplexního řešení zahrnujícího všechny výše zmíněné prvky[1].

Nyní definujeme celou platformu SOA s využitím všech prvků, které by měla zahrnovat:

SOA je výpočetní systém sestávající ze slabě nebo vůbec provázaných komponent, které nazýváme služby. Přičemž slabá vazba mezi službami je klíčovou vlastností SOA a odlišuje ji od jiných komponentových architektur.

Služby jsou fyzicky nezávislé softwarové programy navržené tak, aby splňovaly požadavky servisní orientace a návrhu. To znamená, že každé službě je přiřazena její vlastní jedinečná funkcionalita. Metody každé takové služby jsou pak zveřejněny prostřednictvím rozhraní služby, aby je mohli žadatelé služby volat. Žadatel služby je většinou jiná služba, může jím však být i konzolová aplikace nebo webové rozhraní, jednoduše jakýkoliv prostředek, který respektuje komunikační rozhraní služby.

Komunikace služeb mezi sebou se uskutečňuje pomocí mechanizmu výměny zpráv. Zpráva je základní komunikační jednotkou nesoucí obsah ve formátu XML.

Thomas Erl ve své knize doporučuje delegovat na každou službu pouze jednoduchou a jednoznačnou funkcionalitu. Takové služby pak slouží jako stavební kameny pro komplexní systém. Jsou skládány do větších kompozic, které nazýváme orchestrace nebo choreografie, podle toho jakým způsobem jsou řízeny. Kompozicím a obzvláště choreografiím bude věnována hlavní část této práce.

2.2 Historický vývoj SOA

Vznik a následný vývoj SOA byl nejvíce ovlivněn zavedením a standardizací formátu XML a webových služeb.

---

[1] Pro účely této práce nebudeme v definici SOA rozlišovat SOA (architekturu) a servisní výpočetní platformu (service oriented computing), zájemce o podrobnější definici odkazuji na [1], str. 37.

Dle Thomase Erla [1] představuje XML v SOA základní vrstvu pro reprezentaci dat. Jazyk XML definuje formát zpráv přenášených mezi službami, schémata XSD chrání integritu a platnost dat a transformace XSLT umožňuje mapování rozdílných schémat. Shrnuto a podtrženo, XML je nezbytnou součástí SOA.

Vznik webových služeb byl pro SOA stejně důležitý jako formát XML. Za počátek webových služeb můžeme považovat začátek nového tisíciletí, kdy v roce 2000 konsorcium W3C přijalo návrh na specifikaci SOAP (*Simple Object Access Protocol*) a v roce 2001 přijalo návrh jazyka WSDL (*Web Services Description Language*). SOAP je protokol pro výměnu zpráv pomocí HTTP a stal se standardem pro komunikaci webových služeb. WSDL je jazyk pro popis veřejného rozhraní služeb a rovněž se stal standardem. Třetím, avšak neméně důležitým, pilířem SOA je registr služeb UDDI. Právě ten je centrálním bodem pro přístup k definicím popisů služeb a následně ke komunikaci s nimi. Protokol SOAP, jazyk WSDL a registr služeb UDDI společně tvoří tři hlavní pilíře, které souhrnně nazýváme „první generací webových služeb“. Tato generace se řídí standardem vydaným společností WS-I, který je označován názvem *Basic Profile*[2]. Systém na nich postavený pak dle [2] autor Thomas Erl nazývá *prvotní SOA*.

Následovníkem prvotní SOA je *současná SOA*, která staví na rozšíření webových služeb nazývaném *WS-**. Servisně orientovaný systém této druhé generace poskytuje možnosti, jako jsou transakce, zabezpečení na úrovni zpráv, zásady a mnoho dalších vlastností. Všechny jsou standardizované v příslušných specifikacích WS-*[3]. Například zabezpečení nalezneme ve specifikaci *WS-security*, transakční zpracování ve *WS-AtomicTransaction*, apod.

Z hlediska vývoje architektury představují servisně orientované systémy novou generaci systémů distribuovaných. Ostatně na distribuovaných systémech samotná SOA staví a navíc je doplňuje množstvím nových principů a technologií. Ačkoliv by se tak mohlo zdát, SOA není ten samý přístup jako distribuovaná architektura. Jedním z důležitých rozdílů je, že SOA spoléhá na zprávy typu dokument, na rozdíl od distribuovaných architektur, kde je standardem zpráva typu RPC. Také díky základnímu pravidlu SOA, slabé vazbě mezi službami, je potřeba, aby zprávy byly co nejvíce samostatné. Proto se systém výměny zpráv v SOA stal mnohem propracovanějším a vede k menšímu objemu jednotlivých přenosů oproti původní distribuované architektuře.

2.3 Služby v SOA

Služby jsou základní stavební jednotkou celé SOA platformy. Zjednodušeně si je můžeme představit jako nezávislé programy, na které je možné se vzdáleně připojit a volat jejich metody. To jaké metody, která služba nabízí, je definováno v jejím popisu v jazyce WSDL. Popis s takovým veřejným rozhraním je možné od služby vyžádat, automatizovaně zpracovat a získat tak veškeré informace o vlastnostech služby. Komunikace se službou pak standardně probíhá pomocí protokolu SOAP, je však možné použít i alternativní přístup jako je REST (*Representational State Transfer*) nebo čisté XML[4]. REST je architektura rozhraní navržená pro přístup k datům, která jsou chápána jako zdroje a mají své identifikátory URI.

---

[2] Viz http://www.ws-i.org/profiles/basicprofile-1.1.html.
[3] Některé z nich si v dalším textu představíme.
[4] Přístup SOAP je orientován procedurálně, zatímco REST datově. Každý je proto vhodnější pro jiné účely.

Na jazyk WSDL se blíže podíváme nyní, na protokol SOAP se zaměříme později.

2.3.1 WSDL

WSDL je zkratkou pro název *Web Service Desription Language*, což je jazyk založený na XML a je standardizovaný mezinárodní organizací W3C. Momentálně je dostupná nejnovější verze 2.0, která byla vydána v roce 2007[5]. Dokumenty v tomto jazyce slouží k detailnímu popisu webové služby. To zahrnuje jak informace, na jaké URL služba leží, jaký protokol pro komunikaci s ní je potřeba použít, tak i formát těla předávaných zpráv. Především však definuje kompletní rozhraní služby, což je výčet všech metod, které služba nabízí a typy přenášených objektů.

Základními elementy v jazyce WSDL verze 2.0 dle [3] tedy jsou:

- *Types* – obsahuje definice přenášených typů
- *Operation* – definuje metodu, kterou služba poskytuje a její vstupní a výstupní typy
- *Interface* – obsahuje elementy operation, tedy všechny metody, které služba nabízí
- *Binding* – specifikuje jak interface, tak i způsob komunikace se službou, tedy např. zda se používá protokol SOAP nebo REST
- *Endpoint* – slouží jako přípojný bod ke službě, definuje adresu služby a způsob připojení, tedy odkaz na příslušný element binding
- *Service* – definuje množinu všech endpoint.

Dokument v tomto jazyce je možné od každé webové služby získat a použít ho pro automatické generování šablon kódu žadatele služby. Existuje proto mnoho nástrojů, které jsou většinou již integrovány do vývojových prostředí a generují tzv. *proxy třídy*, které se používají pro volání služby. Volání metod služby je potom naprosto srovnatelné s voláním lokálních metod.

Nyní se podíváme na to, jaké základní typy služeb rozlišujeme z pohledu SOA a jak lze služby vytvářet.

2.3.2 Typy služeb

V servisně orientované architektuře můžeme služby rozdělit podle tří základních faktorů [1]. Je to funkčnost, kterou zapouzdřují, možnosti opětovného použití v jiných kompozicích, a to jak úzce je daná služba svázaná s nějakou entitou na serveru (tj. nějakou třídou reprezentující objekt reálného světa, např. třída člověk). Uvažujeme-li tyto aspekty, rozdělujeme pak služby na:

- *Entitní služby (Entity services)*
- *Úlohově zaměřené služby (Task services)*
- *Služby utilit (Utility services)*

Entitní služby skrze své rozhraní vystavují operace, které se týkají pouze jedné entity. V terminologii REST přístupu bychom řekli zdroje. Většinou se jedná o operace typu „vytvoř“, „čti“, „uprav“ a „vymaž“.

[5] Viz http://www.w3.org/TR/wsdl20-primer/.

Úlohově zaměřené služby většinou zastřešují celou kompozici služeb a jejich opětovné použití je často obtížné nebo nereálné. Takové služby se vyskytují v rodičovském byznys procesu[6], který je zodpovědný za komplexnější činnost. Může to být například objednávková služba a její příkaz „vytvoř objednávku“.

Služby utilit podle autora Thomase Erla [1] představují služby, pro jejichž návrh je nejdůležitější jejich opětovné použití v celém řešení. Mohou to být různé konverzní nástroje, nástroje exportu nebo nástroje pro záznam chyb. Nazýváme je také službami infrastruktury nebo službami aplikace.

2.4 Zprávy v SOA

Komunikace mezi službami je zajištěna pomocí mechanizmu zpráv. Jejich formát může být různý, standardně se však v servisně orientovaných řešeních používá protokol SOAP. O protokolu SOAP pojednává následující podkapitola.

Základní definice vidí SOA jako architekturu sestavenou z volně provázaných služeb. To znamená, že všechny stavy aplikace, které je potřeba sdílet mezi službami je potřeba delegovat na úroveň zpráv. Stejně tak autentizace a autorizace se v SOA odehrává na úrovni zpráv. Zprávy tedy nejsou jen jednoduchou jednotkou komunikace, ale zapouzdřují relativně velké množství aplikační logiky a logiky řízení. SOA tedy používá inteligentní a samostatné zprávy. Protože požadavky na takový formát zpráv nejlépe splňuje protokol SOAP, byl později vybrán jako standardní protokol pro servisně orientované systémy. Nyní si o něm řekneme něco více.

2.4.1 SOAP

Protokol SOAP je založený na XML a pracuje nad aplikační vrstvou protokolu HTTP nebo SMTP, viz dokumentace W3C [4]. Díky volbě těchto aplikačních protokolů odpadají ve většině případů problémy s nastavováním síťových prvků, protože protokol HTTP nebývá nijak omezen. To je obrovskou výhodou oproti dříve používaným protokolům jako DCOM (*Distributed Component Object Model*) nebo CORBA (*Common Object Request Broker Architecture*). Pro ty bylo potřeba povolit komunikaci na příslušných portech.

Protokol SOAP zpočátku sloužil jako náhrada za RPC (*Remote Procedure Call*). Až později se stal využívaný mezi webovými službami a v servisně orientovaných platformách.

Zpráva SOAP se skládá z hlavičky a těla, které jsou zabalené do elementu známého jako obálka (*envelope*). Dle specifikace SOAP může být hlavička vynechána. V souvislosti se SOA je však hlavička téměř vždy použita, především protože obsahuje důležité bloky hlaviček, pomocí kterých lze implementovat řadu rozšíření WS-*[7]. Lze tak implementovat různé funkce od zabezpečení přes směrování a spolehlivost až po informace o souvisejících zprávách.

Tělo zprávy nese samotný obsah, který se skládá ze serializovaných dat ve formátu XML. Data je možné serializovat buď pomocí množiny pravidel definované přímo ve standardu SOAP nebo je

[6] Problematika rodičovského procesu souvisí s kompozicí služeb typu orchestrace, která bude probrána později.

[7] Některá rozšíření jsou popsána v dalších podkapitolách.

možné v hlavičce zprávy použít atribut *env:encodingStyle* a definovat tak vlastní schéma kódování dat.

Jedním ze způsobů, jak sestavenou zprávu SOAP přenést je v těle požadavku HTTP, který navíc obsahuje hlavičku *SOAPAction*. Naopak odpověď na tento požadavek už nemá žádné přídavné informace. Možných způsobů pro přenos těchto zpráv je ale více, například je možné využít protokolu SMTP nebo různých rozšíření protokolu HTTP[8].

Nyní se podíváme na některá rozšíření webových služeb z množiny specifikací WS-*, která značně vylepšují koncept výměny zpráv v SOA. Všechny se nacházejí v hlavičkách zpráv SOAP.

2.4.2 Adresování (WS-Addressing)

Adresování při výměně zpráv v SOA je možné přirovnat k doručovacímu listu u běžného balíku. Poskytuje tyto a další adresovací funkce:

- Říká, odkud zpráva přichází
- Kam má být doručena
- Komu má být doručena
- Jaké jsou alternativy při neúspěšném doručení
- Identifikátory jednotlivých instancí služeb

Toto rozšíření zprávám přidává dva nové koncepty. Přidává do hlaviček identifikaci jednotlivých instancí služeb a také informace, jak se má příjemce zachovat. Díky těmto funkcím není doručení zprávy přímo závislé na protokolu přenosu a zprávy si přenos samy řídí. To vylepšuje samostatnost zpráv, volnou vazbu mezi službami a poskytuje určité pro prostředky pro korelaci při výměně zpráv. O korelaci bude řeč v některé z dalších kapitol.

2.4.3 Zabezpečení na úrovni zpráv (WS-Security)

Zabezpečení souvisí s kvalitou služby a rozhodně je nedílnou součástí SOA. Proto vzniklo rozšíření *WS-Security*, které definuje celou množinu specifikací, které řeší:

- Identifikaci
- Autentizaci
- Autorizaci
- Důvěrnost zpráv
- Integritu zpráv

První tři principy jsou ve světě SOA řešeny pomocí systému známého pod názvem *single sign-on*. To znamená, že žadatel služby je autentizován a autorizován první službou, na kterou vznese požadavek a ta mu přidělí jeho identitu. Při dalších požadavcích žadatele na jiné služby se opětovná autentizace neprovádí, pouze se daná identita ověřuje u služby, která ji vydala. Díky tomuto principu se šetří prostředky a služby mohou zůstat ve volné vazbě.

[8] Tato rozšíření definuje tzv. HTTP extension framework, podrobnosti lze nalézt na http://www.ietf.org/rfc/rfc2774.txt.

Aby byla v SOA zjištěna důvěrnost zprávy (nikdo neoprávněný ji neprohlížel) a integrita zprávy (zpráva nebyla změněna), jsou zprávy podepisovány a šifrovány digitálním podpisem.

2.4.4 Spolehlivá výměna zpráv (WS-ReliableMessaging)

Daní za volně vázané služby v servisně orientovaných systémech jsou problémy s komunikací. Rozšíření *WS-ReliableMessaging* tyto problémy řeší a zvyšuje tak kvalitu služby v SOA. Umožňuje zajistit:

- Informaci o úspěšném doručení zprávy
- Zjistit, že došlo k chybě a odeslat tak zprávu znovu
- Zaručit, že zprávy dorazily v požadovaném pořadí

Všechna pravidla a požadavky odesílatelů na notifikaci jsou uvedeny v hlavičkách zpráv, podobně jako to bylo u předchozích rozšíření.

Rozšíření WS-ReliableMessaging úzce souvisí s rozšířením WS-Addressing. To dokazuje například i skutečnost, že identifikátory zpráv zavedené rozšířením adresování jsou používány i rozšířením spolehlivého doručování. Musela se však změnit jejich specifikace, protože původně bylo vyžadováno, aby každá zpráva měla vlastní identifikátor. To se ale neslučovalo s principem opětovného odeslání nedoručených zpráv, které potřebovaly stejný identifikátor pro všechny znovu odeslané zprávy.

2.4.5 Korelace při výměně zpráv

Korelací při výměně zpráv rozumíme jakékoliv udržování kontextu, tedy stavu komunikace (neboli konverzace). Ten je potřeba udržovat i u jednoduché komunikace typu dotaz – odpověď a jeho význam stoupá u složitějších kompozic služeb. Znovu tedy narážíme na problémy, které s sebou volná vazba v SOA přináší. Není totiž možné uchovávat kontext na úrovni služeb, protože jsou volně vázané. Kontext komunikace je potřeba, stejně jako tomu bylo u předchozích principů, vložit do logiky zpráv.

To, jak kontext vypadá, se může lišit v závislosti na kompozicích a akcích, kterých se účastní. Například ve specifických kompozicích typu orchestrace[9] je potřeba uchovávat celé sady korelace, protože jedna zpráva se může účastnit více instancí orchestrací. Naopak u jednodušších aktivit výměny zpráv zcela postačuje jediný identifikátor zprávy. Často se proto využívá WS-Addressing.

V předchozích kapitolách jsme si blíže představili, jak vypadají a jakou roli plní zprávy v servisně orientovaných řešeních. Ukázali jsme si, že z důvodu volné vazby mezi službami je potřeba většinu informací delegovat na úroveň zpráv. Zprávy se tak v SOA stávají plně autonomní a inteligentní jednotkou komunikace, která zastává funkce, které bychom v běžných architekturách přisuzovali spíše aplikační logice.

Nyní přistoupíme k problematice objevování služeb a získávání popisů služeb v celém systému. Řeč bude také o registru služeb UDDI.

---

[9] O orchestraci bude pojednávat kapitola 3.1.

2.5 Zjistitelnost služeb

Představme si psaní jednoduché aplikace, která nemá se servisní orientací nic společného, ani není rozdělena do různých knihoven. Taková aplikace většinou sestává ze tříd a třídy mají vlastní metody. Vždy se snažíme třídy koncipovat tak, aby nedocházelo k duplikování již napsaného kódu a aby se maximálně využily již existující třídy. Protože tříd v projektu není mnoho a vše vidíme řekněme „pohromadě“, není problém zjistit, zda požadovaná funkcionalita je již implementovaná nebo je potřeba napsat třídu/metodu novou.

Stejně tak se chceme vyvarovat duplikování kódu v servisně orientovaných projektech a služby navrhovat jako maximálně znovupoužitelné. Problém ale nastává ve velkých systémech, které čítají stovky až tisíce služeb. Není v našich silách při psaní nové funkcionality procházet všechny existující služby a zjišťovat, zda něco podobného již existuje. Proto byl v SOA zaveden mechanizmus objevování služeb.

Zjistitelnost a objevování služeb představují techniky, jak v množství již implementovaných služeb s rozličnou funkcionalitou nalézt tu správnou, kterou bychom mohli znovu použít pro nové účely. Tyto techniky podtrhují jednu z předních vlastností SOA, a to znovupoužitelnost služeb.

Aby služby mohly být prohledávány, je potřeba, aby každá z nich poskytovala nějakým způsobem informace o svých vlastnostech, metodách a účelech použití. Většinou jsou tyto metadata kombinací obsahu kontraktu služby (většinou popis WSDL) a popisů, které vytvořil člověk v nějakém registru služeb. Tím získáme ucelenou informaci o tom, jaký je hlavní účel služby, jaké metody služba poskytuje a co která metoda provádí. To, jak kvalitní popis služba poskytuje a tedy jak snadno může být objevena, většinou závisí na systémových návrhářích a vývojářích, kteří popisy vytváří.

2.5.1 Registr UDDI

Potřeba centralizovat popisy na jedno místo dala vznik registru služeb UDDI. Za předchůdce UDDI můžeme považovat statické informace vyvěšené na interních stránkách podniku. Později se začalo využívat adresářů LDAP až byl nakonec vyvinut registr UDDI. Také díky němu se prvotní SOA odlišovala od běžných distribuovaných architektur.

UDDI je zkratkou pro *Universal Description Discovery and Integration*. Registr obsahuje popisy všech služeb v systému a také jejich kontrakty, často v jazyce WSDL. Sám registr je také službou, je s ním proto možné komunikovat pomocí zpráv SOAP.

Běžně je registr používán vývojáři a architekty pro vyhledání požadované funkcionality. Protože však registr poskytuje odpovědi ve formátu SOAP, je možné je zpracovávat v jiné aplikaci a provádět tak operace typu „objevování za běhu“. To znamená, že samy běžící aplikace mohou nad registrem provádět dynamické dotazy na popisy služeb a získávat jejich kontrakty.

2.6 Od obchodních procesů k implementaci služeb v SOA

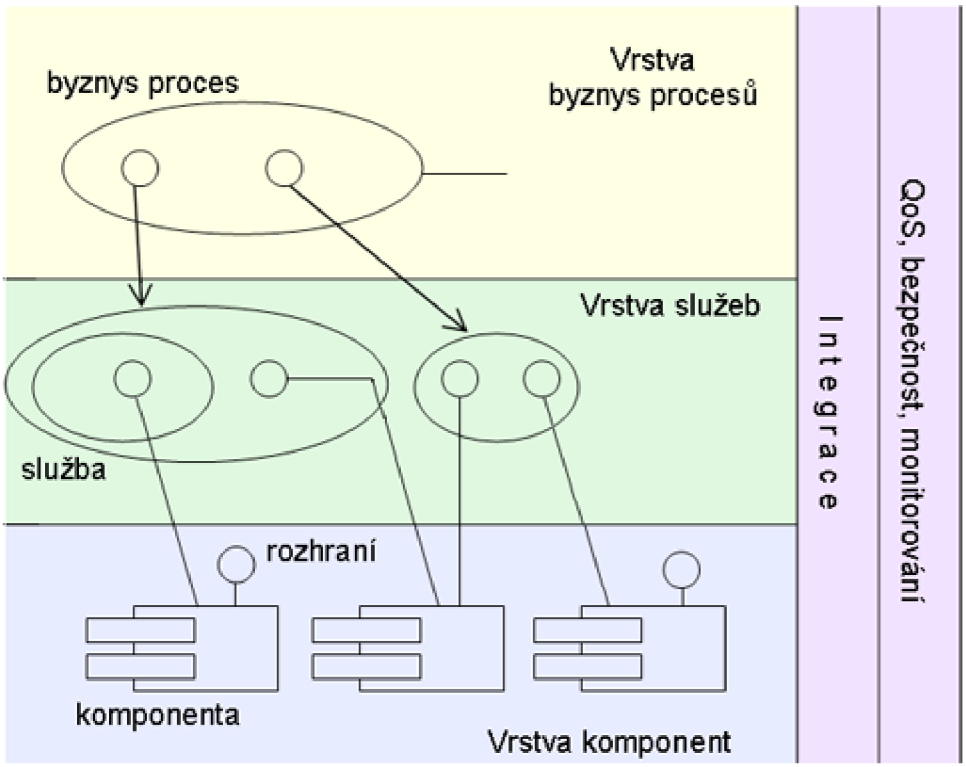
I přesto, že jsme si již představili mnohé výhody SOA a servisní orientace jako takové, nezmínili jsme pravděpodobně ten největší přínos tohoto přístupu. Přístup SOA přímo vyžaduje, aby byl

realizován s ohledem na podnik a procesy v něm probíhající. Často je samotná SOA charakterizována jako implementace obchodních procesů (neboli *byznys procesů*).

Jak uvádí Rychlý a Weiss ve své práci [5], "Byznys proces je posloupnost kroků, která respektuje určitá byznys pravidla a vede k zisku (hmotnému i nehmotnému)."

Proto u správně navržené servisní architektury musí existovat velice úzká vazba mezi tím, jak podnik funguje, jaké má obchodní záměry a tím jak jsou služby navrženy. Návrh SOA musí být přímo řízen obchodními procesy.

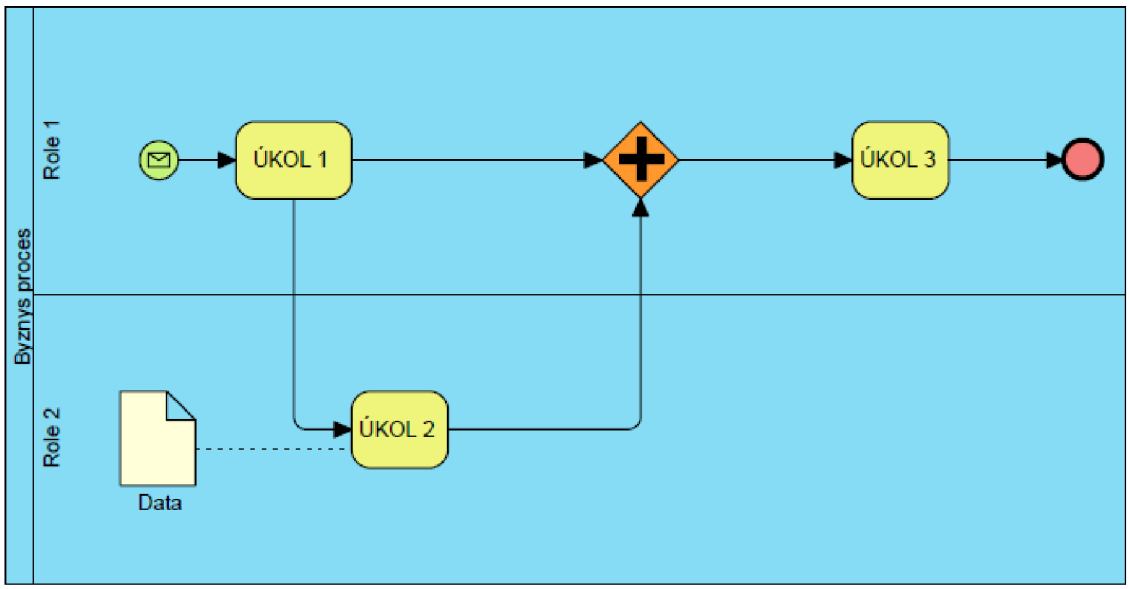
Na následujícím obrázku vidíme hierarchii vrstev, ze kterých se SOA skládá. Na vrcholu hierarchie jsou abstrahované byznys procesy. Pod nimi leží vrstva služeb, navržená na základě vrstvy procesů. A v nejnižší vrstvě je vnitřní logika služeb složená z jednotlivých komponent.



obr. 2.1: Hierarchie vrstev v SOA, převzato z [5]

Proto, abychom dodrželi návaznost na obchodní procesy, měla by posloupnost tvorby SOA tyto vrstvy respektovat a postupovat od nejvyšší vrstvy byznys procesů až po samotnou implementaci služeb. Tím dosáhneme toho, že výsledná architektura bude přímo vycházet z reálného fungování podniku.

Specifikací a analýzou byznys procesů se zabývá odvětví s názvem BPM, což je zkratka pro *Business Process Management*, tedy procesní řízení. BPM zároveň vytváří abstrakci procesů i procesy analyzuje. Pro tvorbu abstraktních popisů procesů BPM používá prostředku BPMN (*Business Process Modeling Notation*), což je grafická reprezentace modelu byznys procesů. Příklad byznys procesu zapsaného v této notaci můžeme vidět na obrázku 2.2 níže.



obr. 2.2: Příklad diagramu zapsaného v notaci BPMN

Na základě abstraktních modelů byznys procesů v BPMN je možné identifikovat tzv. *kandidátní služby*, tedy služby, jejichž souhrnná funkcionalita bude pokrývat celý byznys proces. Takové služby i jejich spolupráce se pak obvykle modelují pomocí jazyka UML. Především se jedná o diagramy tříd a sekvenční a komponentové diagramy[10].

Máme-li popisy rozhraní služeb definovány, je možné vygenerovat příslušné kontrakty služeb v jazyce WSDL. Tyto kontrakty by měly být závazné a neměly by se měnit. Jakákoliv změna kontraktu služby totiž naruší sestavenou kompozici a celý koncept.

Posledním krokem je implementace vnitřní funkcionality služeb na základě jejich kontraktů.

Některé ze zmíněných kroků je možné automatizovat pomocí různých nástrojů. V dnešní době například existují prostředky pro automatické vygenerování kontraktů služeb na základě jejich popisu v BPMN.

[10] Prostředky notace BPMN a jazyka UML pro modelování SOA si představíme v kapitole 4.1.

3 Kompozice služeb v SOA

V předchozích kapitolách jsme si vysvětlili, co znamená servisní orientace, co je to SOA a jaké jsou její výhody. Blíže jsme si představili hlavní prvky SOA, jako jsou služby a zprávy. Nyní přistoupíme k hlavnímu tématu této práce a tím je speciální případ kompozice služeb, jejich choreografie. Abychom choreografiím ale lépe porozuměli, podíváme se nejprve na kompozice služeb obecně.

Kompozicí služeb rozumíme uskupení spolupracujících služeb za účelem plnit komplexní úlohy. Takovou komplexní úlohou může být například celý byznys proces. Pavel Knap ve svém článku [6] uvádí, že v odborné literatuře můžeme nalézt mnoho pojmů pro pojmenování různých vzorů spolupráce mezi službami. Nejvýznamnější z nich však jsou orchestrace a choreografie[11].

3.1 Orchestrace

Orchestrace je speciálním typem kompozice služeb. Řízení podřízených služeb je delegováno na jednu službu nadřízenou, která v sobě nese logiku celé kompozice, potažmo procesu. Logika řízení kompozice sestává z posloupnosti volání metod podřízených služeb. Svůj název získala tato kompozice díky podobnosti s hudebním orchestrem. Ten má také svého dirigenta, který jako jediný řídí všechny ostatní.

Mike Rosen ve svém článku [7], kde srovnává orchestraci oproti choreografii, definuje pojem orchestrace následovně. „Orchestrace definuje posloupnost kroků v rámci procesu včetně všech podmínek a výjimek. Za účelem realizování posloupnosti zavádí orchestrace centrálního arbitra, který orchestraci řídí. Orchestrace je běžný přístup používaný jak v kompozicích služeb, tak v byznys procesech.“

V SOA bývá tímto řídícím prvkem vždy jedna nadřazená služba, která na základě sekvence událostí volá metody služeb, které kompozici tvoří. U jednoduchých kompozicí je možné definovat posloupnost událostí přímo v kódu této nadřazené služby. Jedná-li se ale o složitější kompozice, je potřeba nejprve navrhnout vizuální model kompozice v nějakém nástroji pro popis byznys procesů a ten poté vyexportovat do vykonatelného kódu. Jako vizuální model se často používá notace BPMN. Tu je totiž možné většinou automatizovaně převést do jazyka BPEL (*Business Process Execution Language*), který je již vykonatelný.

Jazyk BPEL je založený na XML a tvoří kód jedné služby, která řídí orchestraci. BPEL využívá popisů služeb ve WSDL a podle definované posloupnosti služby volá. Popis služby v tomto jazyce je možné přímo nasadit na aplikační server[12], který je schopný ho provést.

Přístup orchestrace je relativně jednoduchý a ve spojení s grafickými nástroji pro její návrh i velice oblíbený. Orchestrace je podporována mnoha nástroji velkých i menších výrobců software.

---

[11] V literatuře můžeme také nalézt pojem kompozitní služba. Tím buď souhrnně označujeme soubor všech služeb, které spolu spolupracují nebo jednu službu, která kompozici řídí.

[12] BPEL je podporován například aplikačním serverem JBoss nebo Oracle.

Způsob centrálního řízení orchestrace ale přináší jeden závažný problém. Ten se týká spolupráce dvou organizací, kde každá vlastní svoji informační infrastrukturu. V takovém případě není možné určit jednoho arbitra, který by celou spolupráci napříč hranicemi řídil, protože by vždy musel patřit buď k jedné, nebo ke druhé organizaci. Při použití orchestrace v takovém případě je zapotřebí určitých kompromisů.

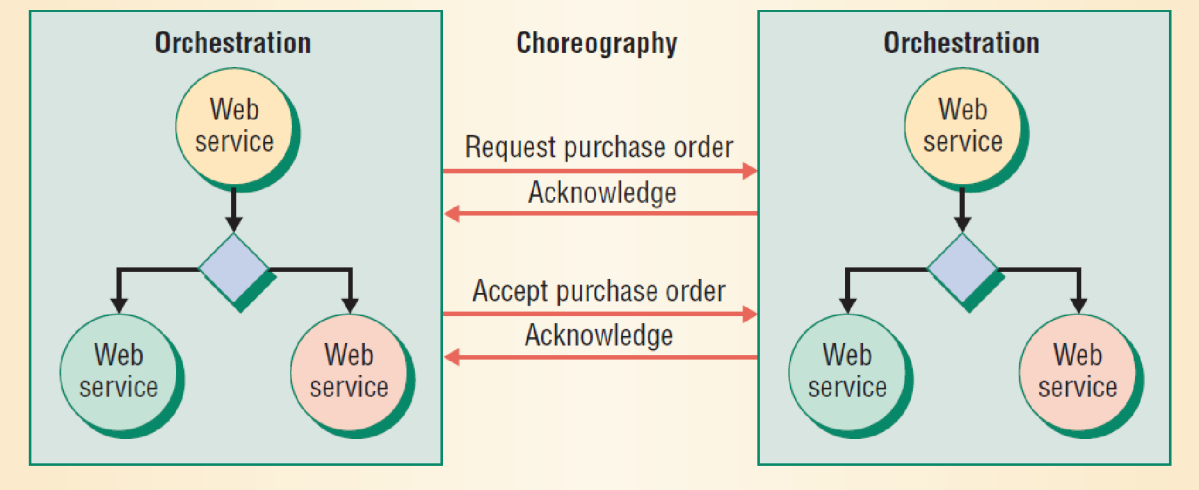
Ideálním řešením spolupráce přes hranice organizací nabízí choreografie. Pojďme se na ni tedy blíže podívat.

3.2 Choreografie

Zatímco orchestrace popisovala sekvenci kroků v rámci procesu, choreografie se soustředí spíše na cíl spolupráce a výměnu zpráv mezi jednotlivými službami. Jak uvádí Thomas Erl v [2], „choreografie většinou předpokládá, že neexistuje žádný vlastník logiky spolupráce“.

Svůj název získala díky podobnosti s choreografií divadelního představení, kde každý herec zná pouze svou roli a tu také vykonává. Stejně tak je to i se službami v choreografii. Každá služba zná svou část úlohy a ví, jak má reagovat na příchozí zprávy. Nezná ale, jak probíhá celý rodičovský proces. Až teprve díky spolupráci všech zúčastněných služeb je možné proces realizovat a dosáhnout cíle. Můžeme zde vidět silnou podobnost v multiagentních systémech, které se chovají naprosto rovnocenně.

Peltz [8] zdůrazňuje že, choreografie se soustředí více na spolupráci a dovoluje každé zúčastněné straně popsat svou část úlohy. Na rozdíl od orchestrace tedy nepopisuje specifický byznys proces, ale posloupnost výměny zpráv mezi více stranami a zdroji. Z těchto důvodů je choreografie často využívána jako spojující most mezi více organizacemi, jejichž vnitřní struktura je definována byznys procesy. Uvnitř organizace tedy vládne orchestrace, komunikace s partnery ale probíhá pomocí choreografií. Takové využití můžeme vidět na následujícím obrázku.



obr. 3.1: Využití choreografie jako pojícího prvku mezi orchestracemi, převzato z [8]

Popis choreografií byl téměř standardizován organizací W3C a v roce 2005 vznikl jazyk WS-CDL (*Web Service Choreography Description Language*)[13]. Jazyk WS-CDL popisuje z globálního pohledu výměnu zpráv mezi službami za účelem dosažení společného cíle.

V dalších kapitolách si povíme o tom, jaké existují prostředky pro popis choreografií. Tyto prostředky rozdělíme na tři kategorie. Bude se jednat o prostředky pro abstraktní popis, standardizované popisy, kam spadá i WS-CDL a popisy choreografií založené na formálním matematickém modelu.

3.3 Srovnání orchestrace oproti choreografii

V předchozích dvou kapitolách jsme si představili dva základní typy kompozice služeb. Mluvili jsme o orchestraci a choreografii. Řekli jsme si, že orchestrace definuje posloupnost operací. Celou posloupnost řídí jedna nadřazená služba, která je řízena popisem v jazyce BPEL. Naopak choreografie zachycuje z globálního pohledu interakce mezi službami. Neexistuje zde žádný nadřazený element, který by vše řídil. Orchestrace popisujeme jazykem WS-CDL.

Orchestrace jsou od svého počátku podporovány v notaci BPMN, která slouží pro konceptuální popis. Naopak choreografie se této podpory dočkaly až ve verzi BPMN 2.0, která vyšla v roce 2009.

Mike Rosen ve svém článku [7] názorně srovnává použití obou přístupů, které si nyní také ukážeme. Uvádí, že orchestraci je vhodné použít v následujících případech:

- Stavíme-li uzavřené řešení, které nemá předpoklad se příliš měnit
- Pokud je vyžadována grafická reprezentace procesu
- Pokud se definice procesu příliš nemění

Naopak choreografie se podle Rosena spíše hodí v těchto podmínkách:

- Je-li potřeba, aby proces znali všichni zúčastnění partneři, typickým případem může být B2B řešení
- Pokud účastníci procesu vyžadují vlastní implementace procesu
- Pokud se procesy dynamicky mění

Vidíme tedy, že každá technika má své uplatnění a nelze říci, že jedna je obecně lepší. V praxi se nejčastěji využívá kombinovaného přístupu.

[13] Jazyk bude podrobněji vysvětlen v kapitole 4.2.1.

4 Popis choreografií

S rostoucím používáním choreografií vznikají i podpůrné prostředky pro jejich popis, a to nejen na půdě standardizačních organizací. O toto téma se zajímá i akademické prostředí, kde vznikají popisy založené na matematickém formálním modelu. Rozličné možnosti popisů choreografií a jejich uplatnění si nyní blíže rozebereme.

Bavíme-li se obecně o prostředcích pro popis kompozic, je potřeba říci, že většina nástrojů primárně podporovala orchestrace a až v jejich dalších verzích byla doplněna i choreografie. Například první verze notace pro popis byznys procesů BPMN (bude popsáno v příští kapitole) byla vydána roku 2004, ale speciální specifikace pro choreografie se jí dostalo až o tři verze později roku 2009.

4.1 Prostředky pro abstraktní popis

V této podkapitole si představíme dva nástroje, které se běžně používají k modelování choreografií, potažmo celé SOA na abstraktní úrovni. Budeme mluvit o množině abstraktních popisů. Ta bude zastoupena především diagramy, jejichž účelem je spíše napomoci společnému vyjádření konceptu, než detailně mapovat funkčnost systému.

Diagramy, které zde představíme, se využívají především v první fázi návrhu, kdy je potřeba sjednotit komunikační prostředek, tak aby mu rozuměly osoby všech profesí v týmu. Diagramy ukázané v této kapitole budou velmi triviální, aby pouze demonstrovali jejich základní vlastnosti. Použití těchto diagramů při návrhu reálných procesů uvidíme v praktické části práce.

4.1.1 BPMN

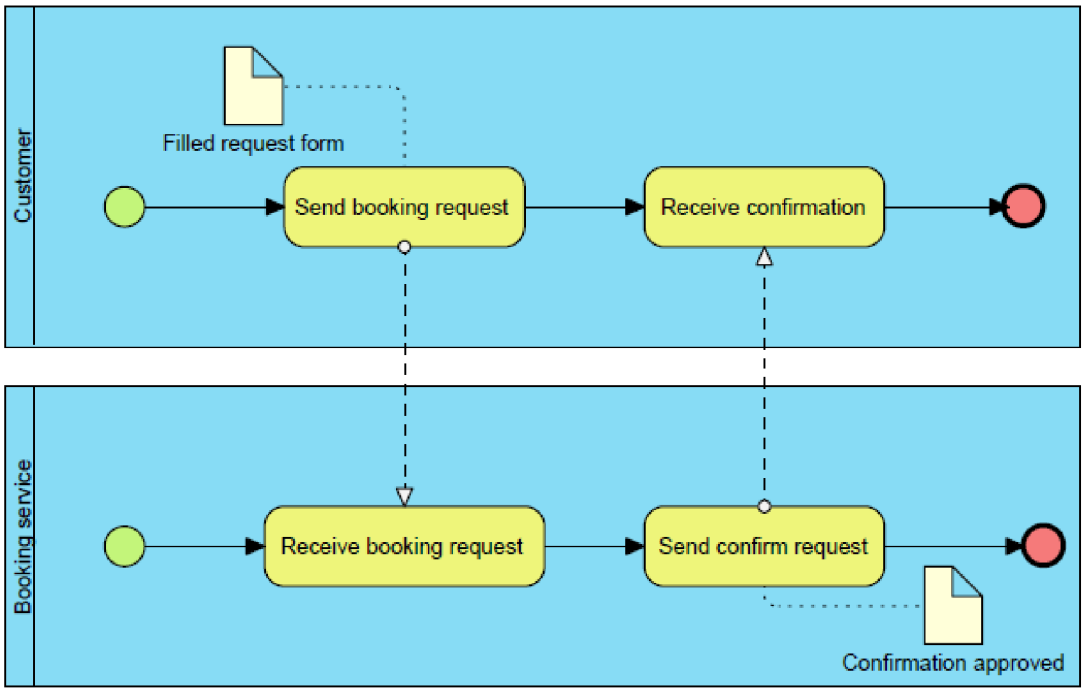
BPMN je zkratkou pro název *Business Process Modeling Notation*. Jsou to vývojové diagramy sloužící pro znázornění byznys procesů. Při vzniku BPMN bylo cílem vytvořit nástroj, který usnadní komunikaci mezi všemi osobami, které se podílí na životním cyklu procesu. Mezi tyto osoby zahrnujeme např. byznys analytiky, vývojáře nebo konzultanty. Vzhledem k tomu, že každá z těchto osob se svými znalostmi pohybuje na jiné úrovni specifikace procesu, je společný vyjadřovací prostředek nezbytný. Jak uvádí Petr Vašíček v [9]: „Díky BPMN se úspěšně podařilo zmenšit komunikační mezeru mezi návrhem a implementací procesu a díky desítkám nástrojů, které jej používají, se stalo de facto standardem pro modelování procesů.“

Tuto notaci není možné přímo převést na vykonatelný jazyk, což je dáno především její volností vyjadřování. Jsou však dostupné některé nástroje, které to umí za cenu některých omezení v BPMN.

Přestože popisovat choreografie v této notaci bylo možné již od její první verze, způsob jakým byly choreografie vyjádřeny, nebyl zrovna ideální. Zvláštní specifikaci pro choreografie zavedla až verze z roku 2009, kde můžeme poprvé nalézt tzv. *Choreography-Model*. Pojďme si nyní tuto novější specifikaci ukázat a porovnat s možnostmi, jaké nabízela verze 1.0 z roku 2004.

Pro názornost budeme modelovat velice jednoduchý proces rezervace letenky. V tomto procesu budou vystupovat pouze dva účastníci, a to rezervační systém a klient. Každý účastník je zastoupen službou.

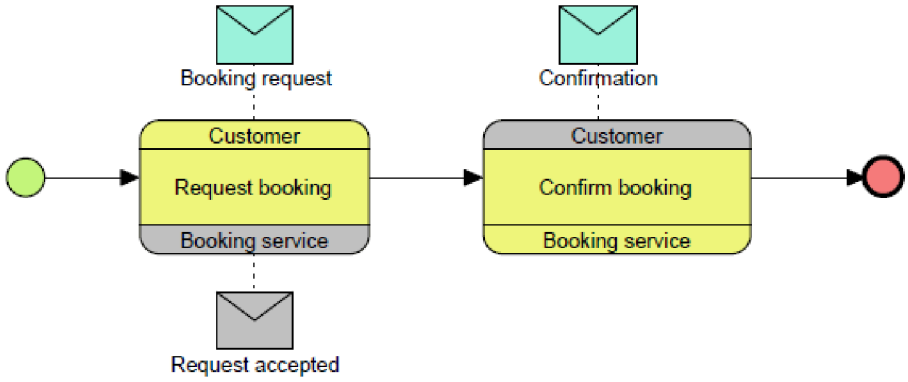
Nejprve si ukážeme, jak by choreografie vypadala ve starší specifikaci BPMN bez využití specifikace Choreography-Model.



obr. 4.1: Zápis choreografie v BPMN bez využití Choreography-Model

Na obrázku 4.1 vidíme dva účastníky choreografie, každého ve vlastním tzv. *pool*, což je modrá oblast, která ohraničuje jeho aktivitu. Mezi účastníky probíhá komunikace pomocí zpráv, která je znázorněna pomocí šipek, které pooly spojují. Specifické akce jsou znázorněné ve žlutých obdélnících i s jejich pojmenováním. Důležitým prvkem v BPMN je *dokument*, což jsou většinou nějaká data, která se vážou k dané akci. V diagramu je znázorněn pomocí listu s přehnutým rohem a v tomto případě vyjadřuje vyplněný formulář rezervace (*Filled request form*) a schválené potvrzení rezervace (*Confirmation approved*).

Nyní si ukážeme stejnou choreografii, akorát vyjádřenou pomocí novější verze notace za využití prostředku Choreography-Model. Uvidíme, že diagram je úspornější a lépe vystihuje povahu komunikace v choreografii.



obr. 4.2: Zápis choreografie v BPMN s použitím Choreography-Model

Choreography-Model zavádí nový prvek s názvem *Choreography-Task*, což je obdélník s alespoň třemi řádky (viz obrázek 4.2). Prostřední řádek označuje název aktivity a krajní řádky nesou název účastníků komunikace. Přičemž účastník, který komunikaci zahajuje, je vyznačen stejnou barvou jako název aktivity. Choreography-Task modeluje jednu výměnu zprávy mezi účastníky. Důležitým prvkem v Choreography-Model diagramu je zpráva. Značí ji obálka a představuje obsah reálné komunikace mezi účastníky spolupráce. BPMN rozlišuje dva druhy zpráv, a to *zahajující* (*initiating message*), podbarvenou zelenou barvou a *odpovídající* (*responding message*), podbarvenou šedou barvou.

Již na první pohled je zřejmé, že Choreography-Model činí zápis mnohem stručnějším a přehlednějším. Ještě významnější výhoda však vynikne spíše u složitějších modelů. Při použití Choreography-Modelu se totiž vyvarujeme duplicit, které při použití starší verze notace nevyhnutelně vznikají.

4.1.2 UML 2

Jazyk UML je v oblasti softwarového inženýrství známým pojmem. Zkratka zastupuje celý název Unified Modeling Language. UML je především množinou specifikací diagramů, které se používají pro návrh software s objektovou orientací. Od verze 2.0, která byla vydána a přijata organizací OMG[14] roku 2005, poskytuje UML vynikající podporu i pro servisně orientovaný návrh. Důležité je, že dostatečně podporuje nejen kompozice typu orchestrace, ale i choreografie.

Pro servisně orientovanou architekturu jsou významné zejména tyto diagramy:

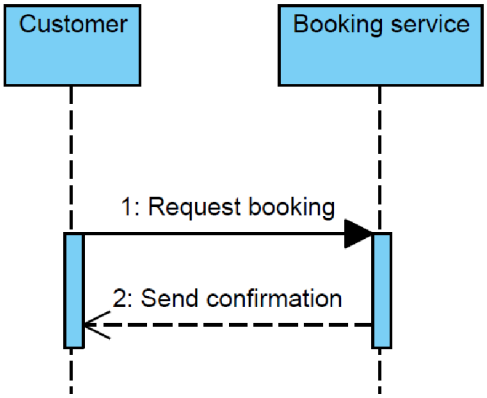
- Diagram tříd
- Komponentový diagram
- Sekvenční diagram
- Diagram aktivit

Diagram tříd se v objektovém světě běžně používá pro definování vnitřní struktury tříd a vztahů mezi nimi. Při servisně orientovaném návrhu ho využijeme především pro definice rozhraní služeb.

Komponentový diagram se využívá pro znázornění služeb a jejich vztahů. Vztahy definují, které metody rozhraní služba vyžaduje a zároveň, které nabízí ostatním službám.

U *sekvenčního diagramu* se zastavíme o chvíli déle. Jelikož tento diagram patří do skupiny tzv. *interakčních diagramů*, velmi dobře se hodí pro popis souběžných procesů, což jsou i choreografie. Na tzv. *lifelines* zobrazuje procesy (v případě SOA jsou to služby) a zprávy, které si mezi sebou posílají. Díky tomu, že má každý proces svou vlastní lifeline, je možné zavést časovou závislost mezi zprávami. Druhým klíčovým prvkem těchto diagramů jsou tzv. *fragment*. To je ohraničená část diagramu, opatřená nějakou podmínkou. Podle toho jakou podmínku uvedeme, je možné znázornit provádění fragmentu jako smyčku nebo větvení typu „if – else“. Jeho vyjadřovací schopnosti a stejně tak forma návrhu jsou si tedy velmi podobné s již zmíněnou specifikací BPMN, Choreography-Model. Jak takový diagram vypadá, můžeme vidět při modelování stejné choreografie jako v předchozí kapitole na obrázku 4.3.

[14] Object Management Group – Organizace, která se stará o vydávání standardů v oblasti návrhu software.



obr. 4.3: Příklad sekvenčního diagramu, popisuje jednoduchou komunikaci mezi zákazníkem a rezervačním systémem

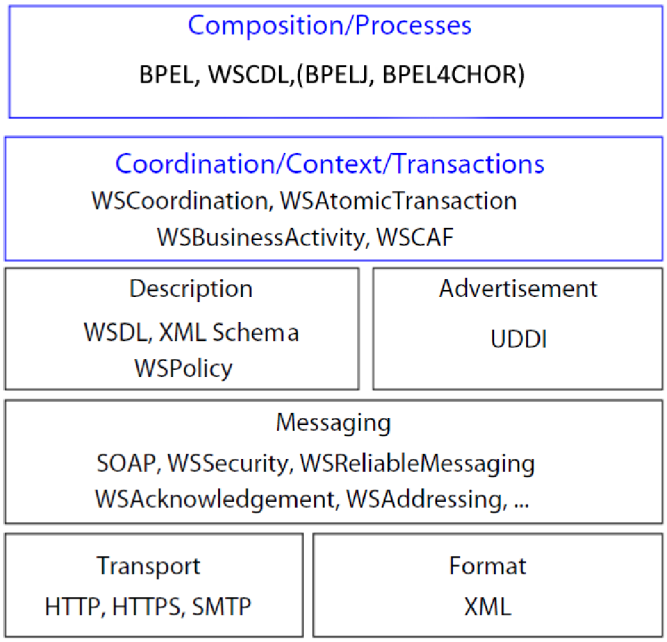
Poslední diagram zmíněný v souvislosti s návrhem SOA je *diagram aktivit*. Jedná se o další typ vývojového diagramu, který se nejčastěji používá pro zachycení workflow nebo jednoho vyvíjejícího se procesu. Protože tento diagram většinou nezachycuje interakce v systému, ale pouze vývoj stavu systému, používá se v souvislosti se SOA k popisu orchestrací.

4.2 Standardy pro popis choreografií

Postupem času se SOA stává stále více rozšířeným paradigmatem pro integraci software uvnitř organizací a mezi nimi. Zatímco nástroje představené v minulé kapitole (BPMN, UML 2) jsou využívány i mimo SOA, nástroje, které si představíme v této kapitole, se SOA přímo souvisí. Nebýt vzrůstající obliby SOA, tyto nástroje by se pravděpodobně ani nedočkaly širokého přijetí. Řeč bude především o tzv. *DSL* (*Domain specific language*)[15].

Abychom si mohli tyto nástroje lépe zařadit a ujasnit si tak, jakou část z celé SOA zastupují, ukážeme si na následujícím obrázku 4.4 množinu všech standardizovaných prostředků, které se v SOA využívají.

[15] Úzce zaměřený jazyk, vytvořený speciálně pro danou doménu problému.



obr. 4.4: Množina technologií a standardů, na které SOA staví, převzato z [10]

Bez množiny standardizovaných specifikací by SOA nemohla fungovat. Standardizace totiž zajišťuje shodnou interpretaci každým, kdo chce SOA implementovat. Jak vidíme na obrázku 4.4, pro kompozice, které stojí na vrcholu všech specifikací, byly vytvořeny doporučení BPEL a WS-CDL. V závorce jsou uvedeny ještě jazyky BPELJ a BPEL4CHOR, které sice nebývají chápány jako obecně přijaté standardy, ale protože oba vychází ze standardu BPEL a podporují tvorbu SOA, uvedl jsem je zde také.

Jazyk BPEL jsme již představili v kapitole 3.1 v souvislosti s orchestrací služeb. WS-CDL plní obdobný účel jako BPEL, soustředí se ale speciálně na choreografie. Tento jazyk bude podrobně vysvětlen hned v následující kapitole. BPELJ je kombinací čistého BPEL a jazyka Java. Jak uvádí článek od společnosti IBM [11], „Tento jazyk umožňuje používat Javu a BPEL dohromady. Díky tomuto spojení může každý jazyk dělat to, co umí nejlépe.“ Jako poslední zbývá BPEL4CHOR, jehož cílem bylo přidat rozšíření pro choreografie k čistému BPEL. Blíže o něm bude pojednávat kapitola 4.2.2.

4.2.1 WS-CDL

WS-CDL (neboli *Web Service Choreography Description Language*) je jazyk založený na XML, podporující WSDL (viz kapitola 2.3.1) a patřící do rodiny standardů, kterou označujeme WS-* (viz kapitola 2.2). Jazyk vznikl na základě potřeby jednoznačného a ověřitelného popisu spolupráce mezi službami. Mnoho prvků v tomto jazyce je proto inspirováno procesní algebrou π-kalkul[16]. Jazyk vytvořila skupina W3C – *Web Services Choreography Working Group*[17]. Roku 2009 byl přijat organizací W3C jako *Candidate Recommendation*[18] a skupina, která se o něj starala, ukončila svou

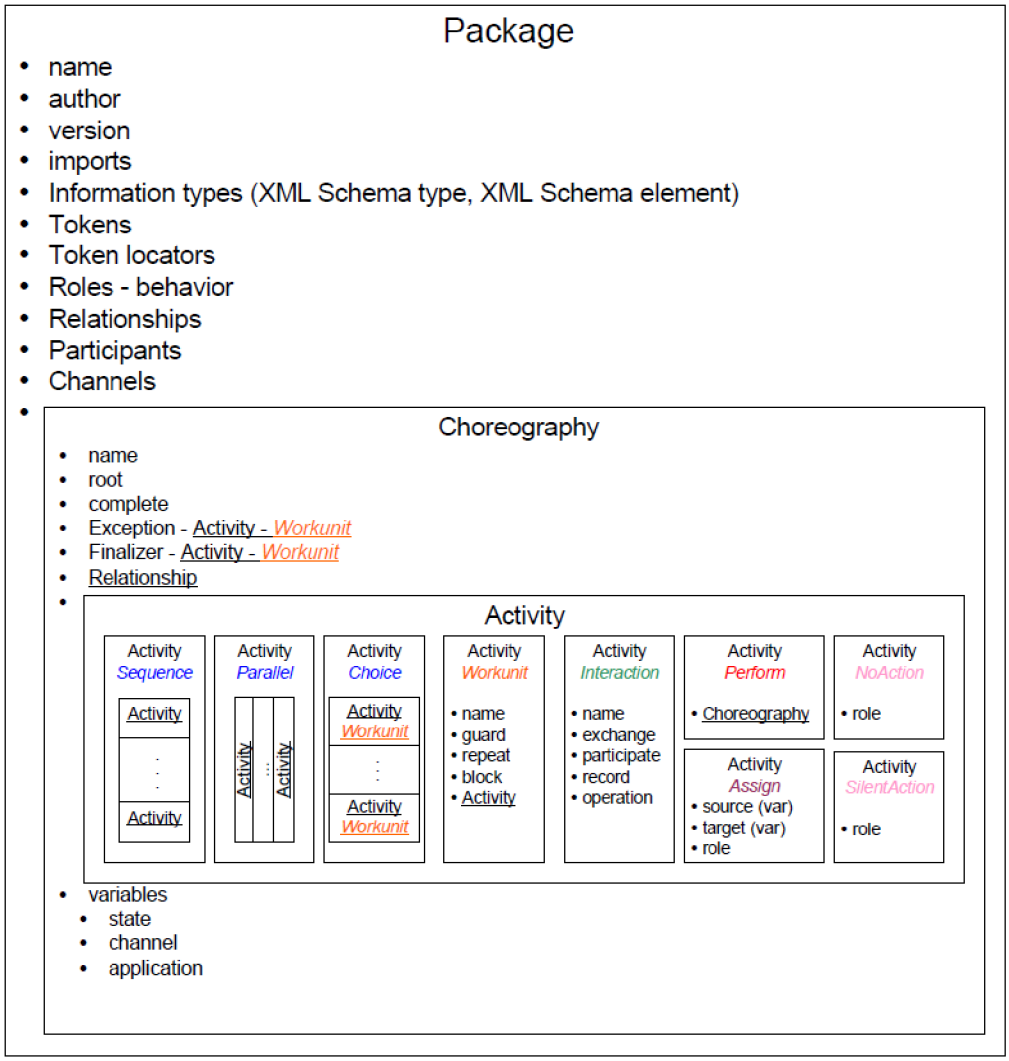
[16] Bude popsáno v kapitole 4.3.2.
[17] Viz http://www.w3.org/2002/ws/chor/.
[18] Toto označení se používá pro fázi, kdy dokument ještě není přijat jako W3C recommendation (což de facto odpovídá technickému standardu), ale je již zveřejněn a čeká se na komentáře přispěvatelů.

činnost. Jazyk dosud nebyl přijat jako W3C doporučení. Přestože mnoho zdrojů vyzdvihuje jeho kvality, díky několika nedokonalostem se jazyk zatím nedočkal podpory v nástrojích společností IBM, Microsoft, BEA nebo Sun, které udávají směr vývoje.

Záměrem tohoto jazyka je globální popis spolupráce služeb, což by mělo otevřít cestu ke generování koster kódu těchto služeb. Zároveň je možné popis použít při testování již implementované choreografie a ověřování, zda vše odpovídá původnímu návrhu. V neposlední řadě je možné na základě popisu detekovat výjimečné stavy v systému, například když je přijata zpráva, která podle choreografie neměla být odeslána.

Blíže tento jazyk definuje Rychlý a Weiss v [5]. „WS-CDL slouží k vytváření předpisů definujících podmínky a omezení vztahujících se na komunikaci pomocí zpráv v rámci spolupráce nějaké skupiny služeb se záměrem dosáhnout určitý cíl.“ Podle těchto autorů je důležité, že předpisy zachycují chování všech služeb z globálního pohledu. Celkové řešení je pak tvořeno dílčími řešeními jednotlivých účastníků, která jsou vytvořená na základě globálního předpisu. Takový přístup nazýváme „shora-dolů“ a jeho výhodou je, že „zatímco způsob řešení v dílčích (lokálních) systémech se může měnit, globální předpisy zůstávají stejné a součinnost všech zúčastněných je zachována“.

Pojďme si nyní ukázat, jak dokumenty v tomto jazyce vypadají a jaké jsou jejich hlavní prvky. Koncepční schéma jazyka WS-CDL můžeme vidět na obrázku 4.5. Dokument sestává z kořenového prvku *package*, který v sobě nese informace o účastnících, kanálech komunikace a jednotlivých choreografiích. Myšlenka WS-CDL je taková, že účastníci jsou propojeni pomocí kanálů a přes ně spolu komunikují pomocí zpráv různých typů. Kanály je možné popsat do nejmenších podrobností od typů zpráv, které přes ně mohou být zasílány, přes definování komunikujících stran až po různá omezení počtu použití či informací, které může kanál přenášet. Pravidla vlastní komunikace potom definují prvky s názvem *choreography*.



obr. 4.5: Schéma struktury jazyka WS-CDL, převzato z [10]

Strukturu dokumentu si představíme na následujícím ukázkovém příkladu[19], tak jak ji definuje organizace W3C [12].

```
<package
      name="NCName"
      author="xsd:string"?
      version="xsd:string"?
      targetNamespace="uri"
      xmlns="http://www.w3.org/2005/10/cdl">

      <informationType/>*
      <variable/>*
      <token/>*
      <tokenLocator/>*
      <roleType/>*
      <relationshipType/>*
      <participantType/>*
      <channelType/>*
```

[19] ? značí nepovinnost oddílu, * značí 0–n výskytů oddílu a + značí 1–n výskytů oddílu.

```
    <choreography/>*
</package>
```

- `name, author, version` – značí jméno, autora a verzi dokumentu
- `targetNamespace` – jmenný prostor použitých datových typů v dokumentu
- `informationType` – definuje typy informací použitých v choreografii
- `variable` – proměnné zachycující informace o účastnících choreografie, mohou být různých typů
  - proměnné obsahující informace
  - proměnné obsahující stav
  - proměnné komunikačního kanálu
  - proměnné výjimek
- `token` – slouží jako alias pro některé části variable
- `tokenLocator` – výraz zapsaný v XPath[20], slouží jako prostředek pro získání informace prostřednictvím tokenu
- `roleType` – definuje výčet rolí, ve kterých se může účastník choreografie nacházet
- `relationshipType` – definuje vztah mezi rolemi
- `participantType` – seskupuje ty části chování (role), které musí účastník implementovat
- `channelType` – popisuje komunikační kanál mezi dvěma účastníky
- `choreography` – popisuje pravidla, která budou použita při komunikaci účastníků

Po tomto stručném přehledu všech prvků jazyka WS-CDL se blíže podíváme na ústřední element choreography. Prvků choreography může být v dokumentu (package) několik, jeden ale musí být vždy kořenový. Ten je zpracováván přednostně.

```
<choreography  name="NCName"
     isolation="true"|"false"?
     root="true"|"false"?>

     <relationship  type="QName" />+

     variableDefinitions?

     Choreography-Notation*

     Activity-Notation

     <exceptionBlock  name="NCName">
            WorkUnit-Notation+
     </exceptionBlock>?

     <finalizerBlock  name="NCName">
            Activity-Notation
     </finalizerBlock>*
</choreography>
```

- `name` – jednoznačné jméno, unikátní pro každou choreografii
- `isolation` – nastavuje dostupnost vnořené choreografie pro ostatní choreografie
- `root` – označuje, zda je tato choreografie kořenová

[20] XPath je standardem společnosti W3C a slouží k adresování částí XML dokumentu. Podrobnosti naleznete na URL: http://www.w3.org/TR/xpath/.

- `relationship` – definuje vztahy (relationshipType) mezi účastníky, které bude choreografie popisovat
- `variableDefinitions` – definuje výčet všech proměnných (variable) použitých v této choreografii
- `Choreography-Notation` – definuje vnořené lokální choreografie
- `Activity-Notation` – specifikuje provedení akce (např. provedení choreografie, přiřazení do proměnné, interakci)
- `exceptionBlock` – definuje podmínky, za jakých mohou být vyvolány výjimky a jak na ně reagovat
  - `WorkUnit-Notation` – popisuje podmínky a omezení provedení nějaké akce (Activity-Notation)
- `finalizerBlock` – pomocí Activity-Notation specifikuje každý finalizerBlock akci k provedení po dokončení choreografie

Na těchto dvou příkladech jsme si ukázali základní strukturu dokumentu popisu choreografie WS-CDL. Tento jazyk je značně složitější a obsahuje mnoho detailů, které je potřeba pro jeho dokonalou znalost pochopit. Všechny tyto detaily však není možné v práci probrat, více informací naleznete v doporučení od společnosti W3C [12].

Paul Bouché v [13] dokazuje, že přestože WS-CDL má mnoho společných prvků s formalizmem π-kalkul, není na tomto formalizmu přímo založený a vystavěný. Společných prvků ale i tak můžeme nalézt relativně mnoho a proto je WS-CDL s π-kalkulem často spojován.

Na závěr této kapitoly bych rád zmínil pár poznatků týkající se praktického uplatnění tohoto jazyka. Díky inspiraci π-kalkulem může jazyk velmi dobře sloužit jako přesný popis choreografií, který je nutný při rozšiřování choreografie a zapojení dalších účastníků. Problém však vidím v jeho nedostatečné podpoře ze strany tvůrců software. Momentálně je dostupný pouze jeden nástroj, který s tímto jazykem pracuje, a to PI4SOA (bude o něm řeč v dalších kapitolách). Jazyk však momentálně nemá vůbec žádnou podporu ze strany Microsoftu, IBM, Sun nebo jiných velkých organizací, tak jako se jí dostalo orchestračnímu jazyku BPEL (viz kapitola 3.1). Proto si myslím, že pokud se tato situace nezmění a velcí tvůrci software nezačlení do svých vývojových nástrojů podporu pro tento jazyk, bude jeho využití spíše omezené.

4.2.2 BPEL4CHOR

Jazyk BPEL4CHOR vznikl jako rozšíření již používaného a standardizovaného jazyka BPEL (viz kapitola 3.1). Motivací pro vznik tohoto rozšíření byla pravděpodobně široká podpora čistého BPEL, jak ze strany organizace OASIS, která ho standardizovala, tak i ze strany velkých výrobců. Bohužel čistý BPEL je využitelný pouze pro specifikaci posloupnosti akcí v rámci jednoho byznys procesu, nelze s ním popsat spolupráci služeb.

Zatímco čistý BPEL je vykonatelný, jeho rozšíření BPEL4CHOR přímo vykonatelné není. Určení tohoto jazyka je tedy velmi podobné jako u WS-CDL. Především je to generování koster kódu služeb, jejich rozhraní a v neposlední řadě také verifikace choreografií oproti návrhu.

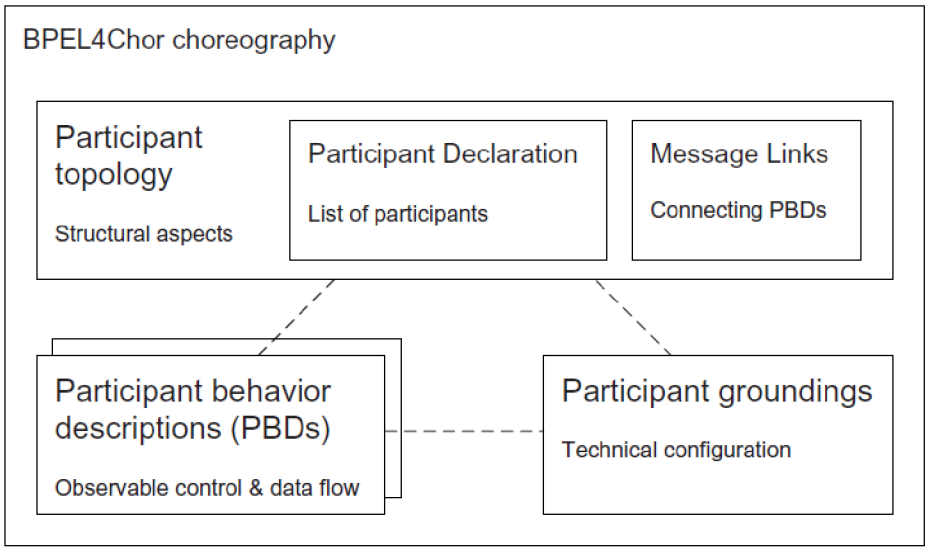
BPEL4CHOR je tedy založený na čistém BPEL, jehož vlastnosti rozšiřuje. Chování každého účastníka v choreografii, kterou popisuje, je popsáno v tzv. *participant behavior description* (PBD), což není nic jiného než pouhý popis v jazyce BPEL. Propojení mezi aktivitami PBD je tvořeno tzv.

message links. Zajímavé je, že tento popis neobsahuje typy portů a operace služeb, které se nacházejí v popisu rozhraní služby WSDL. Má-li ale být choreografie vykonána, každý message link musí být navázán k operaci v popisu WSDL.

Celá choreografie popsaná v tomto jazyce tedy sestává z následujících prvků:

- *Participant topology* – seznam všech účastníků choreografie a všech propojení (message links) mezi jejich aktivitami
- *Participant behavior descriptions* – popisy chování účastníků
- *Participant groundings* – vazba mezi abstraktním message link operacemi z popisu WSDL

To jak spolu tyto prvky souvisí, vysvětluje obrázek 4.6.

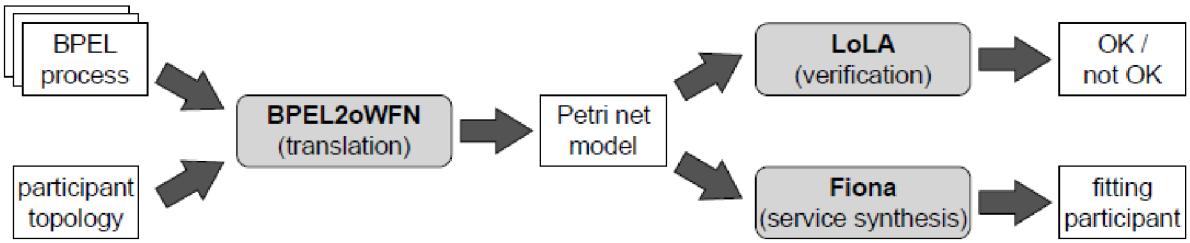


obr. 4.6: Struktura jazyka BPEL4CHOR, převzato z [14]

Při srovnání tohoto jazyka s WS-CDL je potřeba zdůraznit, že WS-CDL poskytuje model interakce, naopak BPEL4CHOR poskytuje model propojení (*interconnection*). Proto není možné popis v BPEL4CHOR verifikovat stejným způsobem jako ten v jazyce WS-CDL.

U verifikace se na chvíli zastavíme a řekneme si jakým způsobem je možné ji v případě BPEL4CHOR realizovat. Protože BPEL4CHOR není založený na žádném formálním modelu a na rozdíl od WS-CDL není ani s žádným příbuzný, není možné ho verifikovat přímo, ale je potřeba ho převést na nějaký formalizmus. Autoři jazyka k tomuto účelu používají formalizmus Petriho sítí[21]. Dle autorů z univerzity v Rostocku [14] je možné tímto způsobem při použití jistých optimalizačních metod verifikovat i choreografie až o tisících účastnících. Jako prostředník mezi Petriho sítěmi a BPEL4CHOR se využívá prostředek *open workflow nets* (oWFNs)[22], který vhodně zachycuje sémantiku BPEL4CHOR. Celý řetězec transformací, které vedou na verifikovaný model v Petriho sítích můžeme vidět níže na obrázku 4.7.

[21] O Petriho sítích pojednává kapitola 4.3.1.
[22] Více informací o oWFNs může čtenář nalézt na http://www.gnu.org/software/bpel2owfn/.



obr. 4.7: Proces validace choreografie za použití Petriho sítí, převzato z [14]

Ani tento jazyk zatím nemá žádnou podporu ze strany velkých hráčů v softwarových produktech. Popisy v tomto jazyce je však možné vytvářet za pomocí editoru k tomuto účelu vytvořenému[23]. Pro proces verifikace choreografií existuje také jistá podpora, a to v podobě open source nástroje GNU BPEL2oWFN[24], který převádí popis v BPEL do notace oWFNs. Ty je možné verifikovat a zkoumat podobně jako Petriho sítě.

4.3 Popisy založené na formálním modelu

V předchozích kapitolách jsme uvedli nejdříve diagramy pro popis choreografií, poté následovaly doménově specifické jazyky (DSL). Jako na poslední prostředky pro popis spolupráce služeb se zaměříme na matematické formální modely a jiné formalizmy. Vybereme takové formalizmy, které se pro popis choreografií a souběžných procesů hodí nejvíce a ty si náležitě popíšeme.

Cílem této kapitoly je ukázat další alternativu popisu choreografií služeb, a to popisy ve formálních jazycích. Formální jazyk je takový jazyk, který má matematicky plně definovanou syntaxi, sémantiku a důkazní systém (tj. lze matematicky dokázat, zda daný výraz do jazyka patří nebo nepatří). Nepopiratelnou výhodou formálních modelů je možnost aplikace formálních metod. Jak uvádí Marek Rychlý v [15], takovými metodami může být dokazatelnost formulí temporálních logik, dokazatelnost či vyvratitelnost některých kritických vlastností (existence uváznutí apod.), simulace v určitém prostředí, různé procházení stavového prostoru atd. Tyto metody nám zajišťují prostředek pro validaci softwarových architektur různých druhů a potažmo její bezpečnost.

Formálních jazyků existuje velké množství. Každý jazyk je však vhodný pro popis určitého typu systémů. Dle Marka Rychlého hraje důležitou roli při výběru vhodného jazyka mnohem více aspektů než by se mohlo zdát. V prvé řadě je samozřejmě potřeba zvážit vhodnost jazyka pro daný systém. Důležitá je také míra zkušeností návrháře s daným formalizmem a dostupnost nástrojů pro modelování ve zvoleném formalizmu. A konečně také to, zda existují prostředky pro navázání na další fáze ve vývojovém procesu. Především zda je možné z formalizmu generovat kostry kódu nebo zda ho lze převést na nějaký standardizovaný jazyk.

Mezi formální jazyky řadíme například:

- *CSP*
- *Linear temporal Logic*
- *Petriho sítě*

[23] Nástroj je dostupný na adrese http://www.bpel4chor.org/.
[24] Viz http://www.gnu.org/software/bpel2owfn/.

- *Π-kalkul*
- *RAISE*
- *SPIN*
- *Statecharts*
- a mnoho dalších…

V oblasti popisu choreografie služeb v SOA se nejvíce používají dva formalizmy, které si v této práci popíšeme. Mluvit budeme o Petriho sítích a procesní algebře π-kalkul.

4.3.1 Petriho sítě

Petriho sítě (PN) jsou formální grafický jazyk sloužící pro modelování systémů se souběžností a sdílením zdrojů. Často jsou používány pro popis distribuovaných systémů a ověřování jejich validity. Formalizmus Petriho sítí byl poprvé představen německým matematikem a informatikem Carlem Adamem Petrim počátkem 60. let v jeho disertační práci. Petriho síť je forma orientovaného bipartitního grafu s ohodnocením. Taková síť sestává ze dvou druhů uzlů, prvním jsou místa a druhým přechody. Orientované hrany pak spojují místa s přechody. Vzhledem k tomu, že se jedná o bipartitní graf, hrana nemůže nikdy spojovat místo s místem nebo přechod s jiným přechodem.

V dalším textu budeme pod pojmem Petriho síť uvažovat tzv. *P/T Petriho síť* (*Place/Transition*), která je formálně definována jako uspořádaná šestice *(P, T, F, M_0, W, K)*, kde:

- P je množina míst
- T je množina přechodů, přičemž platí $P \cap T = \emptyset$
- F je množina orientovaných hran mezi místy a přechody, $F \subseteq (P \times T) \cup (T \times P)$
- M_0 – počáteční ohodnocení je funkce $M_0: P \to N$, která přiřadí určitý počet tokenů (počáteční ohodnocení) každému místu
- W – váha hran je funkce $W: F \to N - \{0\}$, která přiřadí každé hraně kladné číslo udávající kolik tokenů je při provedení jednoho přechodu zkonzumováno nebo vygenerováno (vede-li hrana z přechodu do místa)
- K - kapacita je funkce $K: P \to N - \{0\}$, která přiřadí každému místu kladné číslo udávající maximální počet tokenů, které se v něm mohou najednou nacházet

Sémantiku Petriho sítí, tedy pravidla pro provádění přechodů (evoluční pravidla), vymezíme následujícími definicemi převzatými z [18]:

Definice 4.1 Značení Petriho sítě

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť. Zobrazení $M: P \to N \cup \{\varpi\}$ se nazývá značení (*marking*) Petriho sítě N, jestliže $\forall p \in P: M(p) \leq K(p)$.

Definice 4.2 Proveditelnost přechodu

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť. Přechod $t \in T$ je *proveditelný (enabled)* při značení M, stručněji *M-proveditelný (M-enabled)*, jestliže

$$\forall p \in t: M(p) \geq W(p, t)$$

$$\forall p \in t: M(p) \leq K(p) - W(t, p)$$

Definice 4.3 Provedení přechodu, následné značení

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť a M její značení. Je-li přechod $t \in T$ proveditelný při značení M, pak jeho *provedením* získáme následné značení M' ke značení M, které je definováno takto:

$$\forall p \in P: M'(p) = \begin{cases} M(p) - W(p,t), jestliže\ p\ \in t \setminus t \\ M(p) + W(t,p), jestliže\ p\ \in t \setminus t \\ M(p) - W(p,t) + W(t,p), jestliže\ p\ \in t \cap t \\ M(p), jinak \end{cases}$$

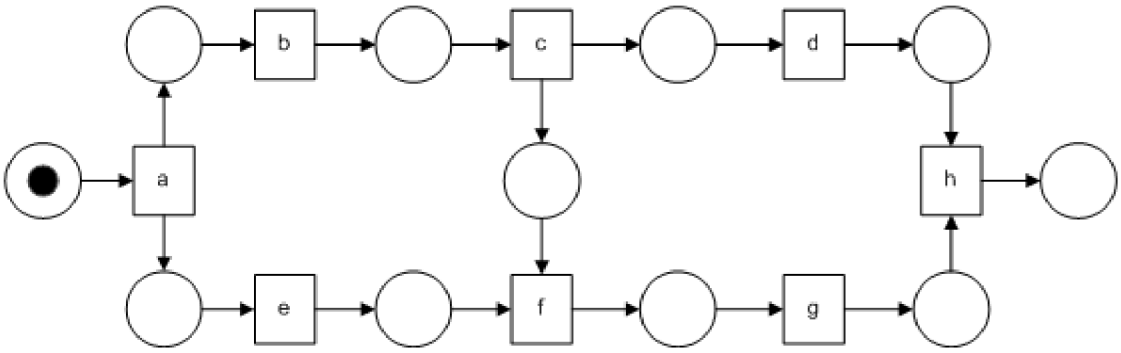
Definice 4.4 Množina dosažitelných značení

Nechť $N = (P, T, F, W, K, M_0)$ je Petriho síť a M její značení. Symbolem $[M\rangle$ označme nejmenší množinu různých značení sítě N takovou, že platí:

1. $M \in [M\rangle$
2. Je-li $M_1 \in [M\rangle$ a pro nějaké $t \in T$ platí $M_1[t\rangle M_2$, pak $M_2 \in [M\rangle$

Množina $[M\rangle$ se označuje jako *množina dosažitelných značení* (*reachibility set*) ze značení *M*. Množina $[M_0\rangle$ dosažitelných značení z počátečního značení se označuje jako množina dosažitelných značení (reachibility set) sítě *N*.

Na obrázku 4.8 můžeme vidět příklad jednoduchého modelu, který je popsán Petriho sítí. Prázdné čtverce značí přechody a kružnice značí místa.

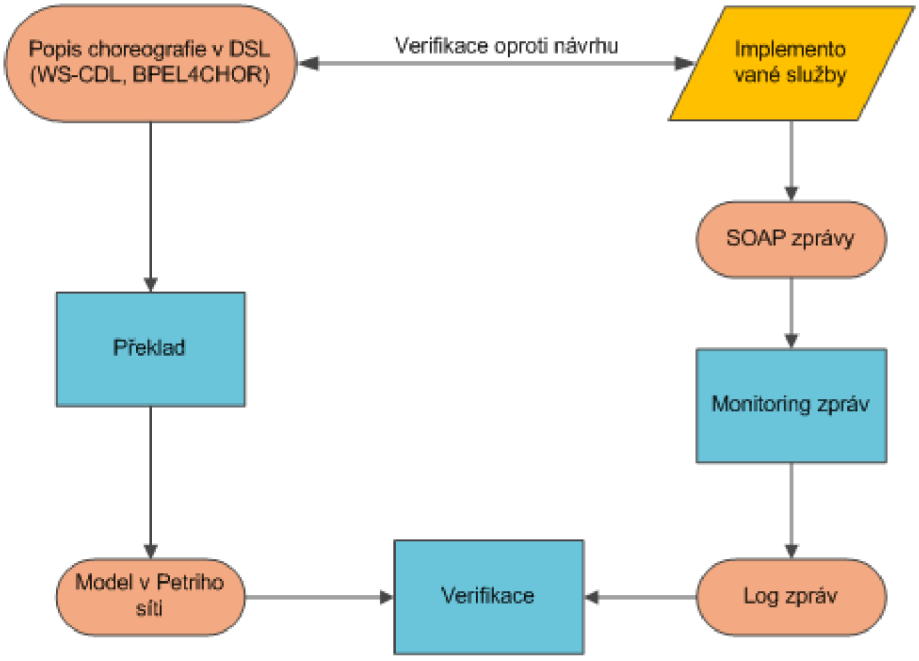


obr. 4.8: Jednoduchý model interakce vyjádřený Petriho sítí

Tento model ukazuje tzv. černobílou[25] Petriho síť s osmi přechody. Nejprve je vykonán přechod *a*, následuje souběžné vykonání *b* a *e*. *c* je vykonáno ihned po *b*, zatímco *f* musí čekat na dokončení obou, *c* i *e*. Takto se postupuje dále, až je vykonán přechod *h* a aktivita sítě skončí. Všechny přechody přitom byly vykonány právě jednou.

Petriho sítě se hojně využívají jako jazyk pro popis architektur (*Architecture description language*, ADL) nebo na nich takové jazyky staví. Pro popis choreografií je samozřejmě možné Petriho sítě použít také, častěji se však používají pro účely jejich validace a ověření správnosti implementace oproti návrhu. Takový princip jsme již nastínili v souvislosti s jazykem BPEL4CHOR (kapitola 4.2.2), nyní se na něj podíváme podrobněji.

[25] Označení černobílá Petriho síť znamená, že token se v místě buď nachází, nebo nenachází. Existuje pouze jeden typ tokenů. Rozlišujeme ještě barvené Petriho sítě, kde se mohou vyskytovat tokeny různých typů.



obr. 4.9: Proces validace choreografie za využití Petriho sítí

Jak můžeme vidět na diagramu z obrázku 4.9, Petriho sítí se při validaci implementované choreografie využívá následujícím způsobem:

- Pomocí monitorovacího nástroje se z běžící choreografie získá do logovacího souboru záznam volaných služeb a jejich metod.
- Tento soubor se pak použije jako trajektorie, po které se prochází Petriho síť vytvořená na základě návrhu v daném DSL jazyku.

Pokud trajektorie neodpovídá konfiguraci Petriho sítě nebo například nedochází k využití některých míst, neshoda mezi návrhem a implementací je odhalena.

Petriho sítě jsou tedy často využívány jako vyjadřovací prostředek, který se díky své formálnosti dobře verifikuje. Neslouží proto přímo k popisu choreografií, ale jako jejich zjednodušený, matematicky verifikovatelný model.

4.3.2 π-kalkul

Matematický formalizmus π-kalkul byl navržen jako prostředek pro popis komunikujících, souběžných procesů, jejichž uspořádání se může měnit. Tento formalizmus spadá pod obecný formální model procesní algebry, což je nástroj pro popis procesů na vysoké úrovni abstrakce. π-kalkul vznikl jako rozšíření procesní algebry od Robina Milnera – *Calculus of communicating systems* (CCS). V této kapitole vysvětlíme, jak je π-kalkul definován, jak se používá pro popis architektur a v čem jsou jeho hlavní přednosti.

Vzhledem k tomu, že π-kalkul spadá do množiny procesních algeber, stejně jako všechny procesní algebry sestává ze dvou elementárních prvků:

- *Proces* (agent) – zastupuje subjekt, který v systému komunikuje. V souvislosti s architekturou SOA a choreografií služeb bude proces většinou zastupovat jednotlivé služby. Proces může

být buď atomická entita, nebo může být dále dekomponován na subprocesy. Ve formálním zápisu se značí velkými písmeny, např. *P, Q*.

- *Jméno* (kanál, port) – element, který realizuje samotou komunikaci a zároveň je i předmětem této komunikace. V SOA jím je tedy zpráva, ale zároveň i pomyslný kanál, přes který zpráva putuje do cíle. Ve formálním zápisu se značí malými písmeny, např. *x, y*. Jak uvádí Robin Milner [17], je-li to výhodné, můžeme pomocí jmen rozlišovat odesílající a přijímací konec kanálu. V takovém případě mluvíme o *polaritě jmen*, kde např. jméno $\bar{x}$ nazýváme *co-name* (česky sdružené jméno) ke jménu x. $\bar{x}$ potom značí odesílající konec kanálu, naopak x značí konec přijímající.

Sémantiku π-kalkulu definujeme podle autora Robina Milnera [17]. Jako první definujeme proces 0, což je tzv. prázdný proces (*null process*). Mějme *P*, *Q*, což jsou procesy a *x*, *y*, jména procesní algebry π-kalkul. Potom následující výrazy jsou také procesy.

- $P|Q$ – značí paralelní kompozici procesů *P* a *Q*, tzv. *operátor paralelní kompozice*
- $P.Q$ – značí sekvenční kompozici procesů *P* a *Q*, kdy proces *Q* bude proveden až po skončení procesu *P*, tzv. operátor *sekvenční kompozice*
- $x(y).P$ – značí proces, který čeká na přijetí jména *y* po kanálu *x*, poté pokračuje jako proces *P*, tzv. *vstupní operátor*
- $x\langle y\rangle.P$ – značí proces, který nejprve odešle jméno *y* po kanálu *x*, až poté co je *y* přijato jiným procesem pokračuje jako *P*, tzv. *výstupní operátor*
- $(y)P$ – vytvoří *y* jako nový kanál platné pouze v kontextu *P* a pokračuje jako proces *P*, tzv. *operátor skrytí*
- $!P$ – vytvoří nekonečný počet paralelních kopií *P*, tzv. *operátor replikace*
- $P+Q$ – značí proces, který se chová buď jako *P* nebo *Q*, nedeterministický výběr

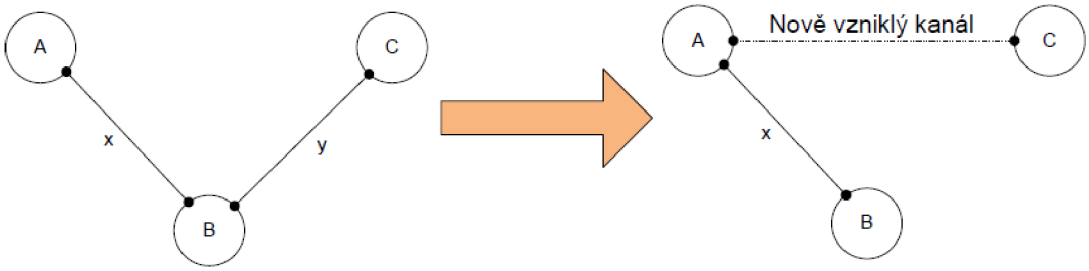
Vývoj komunikace v takto definované procesní algebře se popisuje pomocí redukčního pravidla definovaného takto:

$$(x(y).P)|(y\langle z\rangle.Q) \rightarrow \{z/y\}P|Q$$

Takový formální zápis značí proces $x(y).P$ (můžeme ho označit *P'*), který nejprve přijme po kanálu *x* jméno dále označované jako *y* a poté pokračuje jako proces $\{z/y\}P$. Zároveň zápis vyjadřuje proces $y\langle z\rangle.Q$ (můžeme ho označit *Q'*) paralelní k *P'*, který odesílá po kanálu *y* jméno *z* a dále pokračuje jako *Q*. Složená závorka před *P* značí, že přijímané jméno y bylo nahrazeno jménem z, které odesílal proces $y\langle z\rangle.Q$.

Rád bych na tomto místě zdůraznil, že π-kalkul popisuje pouze komunikaci, nepopisuje vnitřní chování jednotlivých procesů. Stejně tak nedefinuje typy zpráv. Zprávy rozlišuje pouze podle jejich názvu, který je identifikuje. Obecně komunikují procesy pomocí jmen, přes které jména zároveň posílají. To je velmi důležitá vlastnost i pro popis choreografií, protože umožňuje popsat tzv. *mobilitu architektury* (tj. v případě SOA mobilitu komunikačních kanálů). Pro lepší pochopení si ukážeme jednoduchý příklad.

Mějme procesy *A*, *B* a *C*. Procesy *A* a *B* spolu komunikují po kanále *x*, *B* a *C* sdílejí kanál *y*. Formalizmus π-kalkul dokáže vyjádřit událost, kdy proces *B* pošle po kanálu *x* procesu *A* „spojení“ na proces *C*, aby *A* mohl zahájit komunikaci s *C*. Jak se taková jednoduchá choreografie zapíše formálně si ukážeme záhy, grafickou reprezentaci můžeme vidět na obrázku 4.10.



obr. 4.10: Ukázka mobility komunikačního kanálu

Mobilitu komunikačního kanálu y znázorněnou na obrázku můžeme formálně zapsat takto:

$$A|B|C = x(z).z.A'(x,z)|x\langle y\rangle.B'(x)|C(y) \rightarrow y.A'(x,y)|B'(x)|C(y)$$

Mechanizmus mobility komunikačního kanálu přitom může být využit už i ve velmi jednoduché choreografii a je proto pro popis spolupráce nezbytným prvkem. Takovým příkladem může být situace, kde uživatel komunikuje s objednávkovým systémem a objednávkový systém zase se skladem. Dojde-li například k vyčerpání zásob, může služba skladu získat pomocí mobility komunikačního kanálu spojení se zákazníkem a přímo ho na vzniklou situaci upozornit. Tento princip bude také využíván a diskutován v praktické části práce.

Příklady obsahující mobilitu komunikačního kanálu i názornou ukázku použití π-kalkulu nalezne čtenář v kapitole 5.3.2.

Tímto jsme dokončili průvodce různými popisy choreografií a ukázali mnoho předností především formálních popisů. Nyní všechny definované poznatky použijeme v praxi při návrhu servisně orientovaného systému, který využívá principu choreografií.

5 Návrh ukázkové architektury SOA

V posledních dvou kapitolách této práce se budeme věnovat návrhu a implementaci ukázkové architektury SOA. S využitím nástrojů a teorie popsané v předchozích kapitolách budeme při komponování služeb klást hlavní důraz na choreografie. Tyto dvě kapitoly nás budou postupně provázet celým životním cyklem tvorby SOA, od definování požadavků na systém až po výsledný produkt.

5.1 Specifikace požadavků

Požadavky na výsledný systém byly stanoveny v souladu se zadáním této práce. Dle zadání je požadováno implementovat ukázkovou architekturu SOA pro internetový obchod, která bude zahrnovat části pro příjem objednávek, řízení skladu a odesílání faktur. Požadavky plynoucí z tématu práce jsou pak následující.

- Systém musí splňovat kritéria servisní orientace, komunikace mezi systémem a zákazníkem musí probíhat na základě definovaného kontraktu, popsaného standardním jazykem WSDL.
- Služby v systému musí vykazovat rysy choreografie, musí mezi sebou spolupracovat.
- Implementace bude vycházet z předem vytvořeného formálního modelu choreografie, který bude vytvořen pomocí nástroje PI4SOA[26] (bude popsán dále).

Základní požadavky na funkčnost architektury, které definuje zadání, byly rozšířeny a blíže specifikovány následujícím způsobem. Architektura bude podporovat dva hlavní byznys procesy. Prvním je vytvoření speciálního ceníku na míru s využitím údajů o zákazníkovi. A druhým je proces přijetí objednávky, vytvořené na základě tohoto ceníku, vyskladnění a odeslání zboží a vystavení faktury. Bylo navrženo, že zákazníci budou spadat do jednotlivých tříd dle jejich bonity a na základě příslušnosti k třídám jim budou stanoveny ceny za zboží.

Pro lepší názornost ukázkového příkladu byly specifikovány doplňující požadavky na implementaci. Pro komunikaci se systémem bude vytvořena jednoduchá klientská aplikace, která bude umožňovat komunikaci s obchodem jménem různých zákazníků. Dále bude možné prostřednictvím této aplikace vytvořit a odeslat objednávku. Pro lepší názornost bude zároveň zobrazován průběh komunikace v choreografii a samotný XML obsah odchozích a příchozích zpráv.

5.2 Návrh choreografie

Na základě specifikovaných požadavků bylo možné definovat účastníky choreografie a jejich spolupráci mezi sebou. Pro globální zachycení těchto prvků bylo v první fázi návrhu choreografie použito prostředku BPMN (viz kapitola 4.1.1) spolu s rozšířením pro tvorbu choreografií – Choreography-Model. Podle požadavků na funkčnost systému bylo stanoveno 8 účastníků choreografie, kteří budou v pozdějších fázích životního cyklu zastoupeni službami. Těmito účastníky jsou:

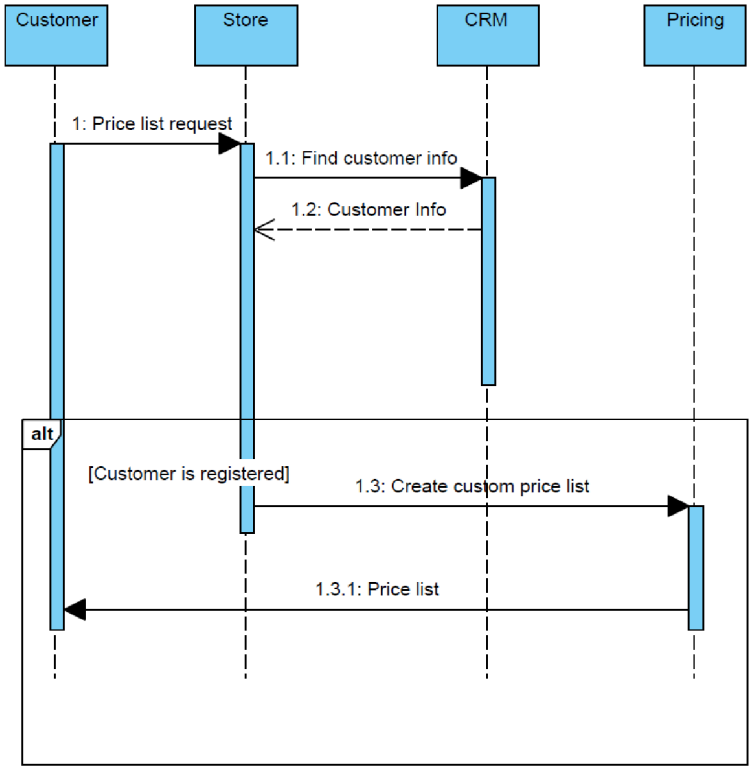
[26] Bližší informace o tomto nástroji naleznete na http://sourceforge.net/apps/trac/pi4soa/wiki.

- *Customer* – reprezentuje v systému zákazníka
- *Store* – skladové hospodářství
- *CRM* – systém správy zákazníků
- *Pricing* – tvorba ceníků dle údajů o zákaznícovi
- *Orders* – příjem objednávek a manipulace s nimi
- *Invoicing* – tvorba a odesílání faktur
- *Expedition* - expedice
- *Delivery service* – zásilková služba

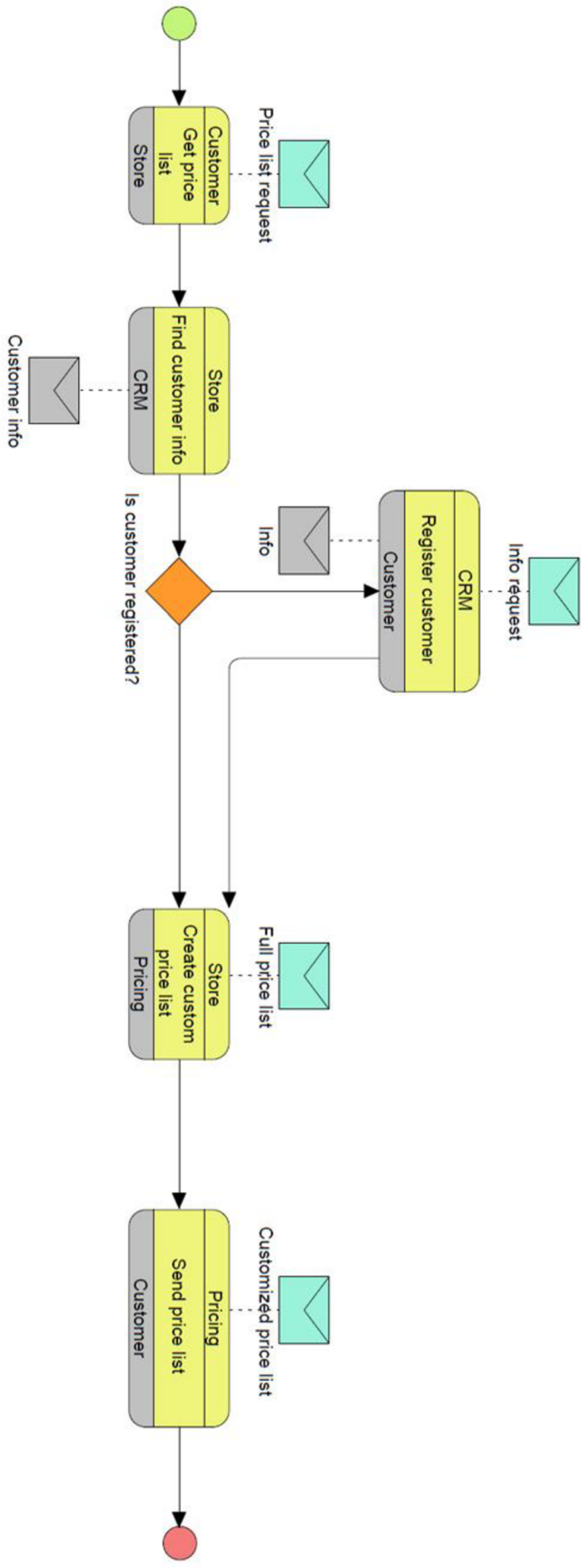
Nyní se podíváme na spolupráci účastníků v prvním byznys procesu tvorby ceníku. Obrázek 5.2, který čtenář nalezne na následující straně, zobrazuje diagram BPMN popisující tuto spolupráci. Vidíme, že choreografie je iniciována účastníkem *Customer*, který odesílá směrem ke *Store* požadavek na ceník – *Price list request*. Spolupráce dále pokračuje tím, že *Store* žádá od *CRM* detailní informace o zákaznícovi, které obdrží jako zprávu typu *Customer Info*. Podle výsledku této zprávy zjistí *Store*, zda je uživatel registrován a v kladném případě pokračuje spoluprácí s účastníkem *Pricing*, kterého žádá o vytvoření přizpůsobeného ceníku. V případě, že zákazník není registrován, *CRM* žádá zákazníka o registraci. Celá choreografie je ukončena odesláním přizpůsobeného ceníku směrem k zákazníkovi.

Diagram procesu přijetí objednávky a odeslání zboží nalezne čtenář v příloze.

Nyní se posuneme v návrhu dále a definujeme vývoj komunikace (spolupráce) v systému. Opět se zaměříme na první proces vytvoření ceníku. Pro návrh vývoje spolupráce v systému byl zvolen sekvenční diagram jazyka UML, znázorněn je na obrázku 5.1.



obr. 5.1: Vývoj spolupráce, je-li zákazník registrován



obr. 5.2: Návrh choreografie procesu vytvoření ceníku

Sekvenční diagram z obrázku 5.1 definuje posloupnost výměn zpráv mezi jednotlivými účastníky. Tento diagram je svým významem téměř totožný s diagramem BPMN, jedná se pouze o jinou formu zápisu. Proto není potřeba ho blíže popisovat. Je potřeba pouze říci, že diagram na obrázku popisuje pouze jednu vývojovou větev komunikace, kdy je zákazník již registrován. Naopak diagram BPMN vyjadřuje alternativy obě.

5.3 Modelování choreografie

Poté, co jsem definoval účastníky choreografie a to, jak se spolupráce mezi nimi vyvíjí, přistoupíme k vytváření formálního modelu choreografie. Ten vytvoříme pomocí formalizmu procesní algebry π-kalkul. Z praktických důvodů budeme používat nástroj na π-kalkulu postavený, který nese název PI4SOA[27]. Definici komunikace v čistém π-kalkulu si pak ukážeme pouze na jednom příkladu.

5.3.1 PI4SOA

Nástroj PI4SOA slouží k návrhu a testování choreografií služeb a pro tyto účely poskytuje grafické prostředí. Modely choreografií v něm vytvořené jsou svou strukturou podobné kombinaci jazyka WS-CDL a π-kalkulu. Nástroj je vytvořen v jazyku Java v prostředí Eclipse.

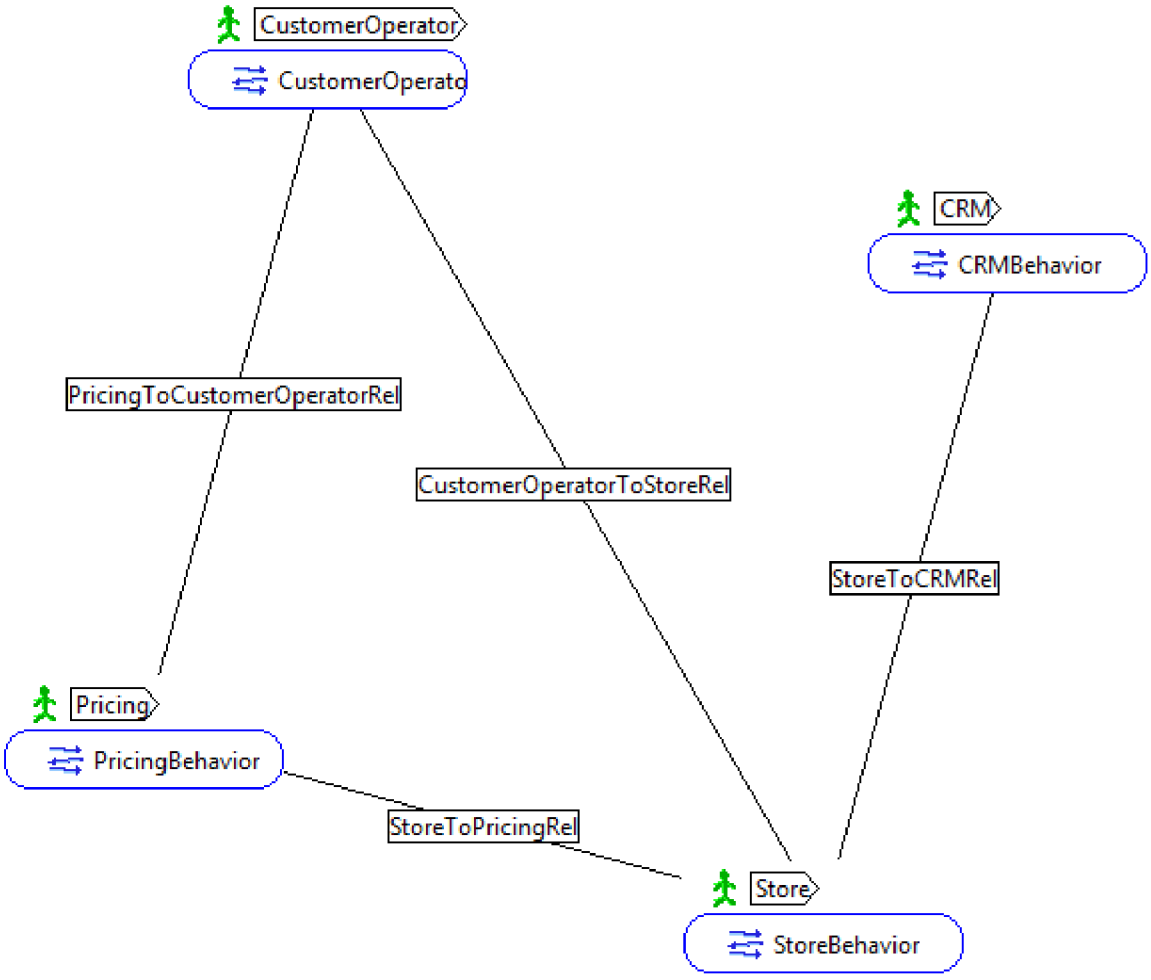
Tento nástroj zvládá export i import choreografií v jazyku WS-CDL. Dále je možné z choreografií vytvářet UML a BPMN diagramy, exportovat WSDL popisy služeb a vytvářet slovní popisy choreografií v HTML. Největšího využití tohoto nástroje dosáhneme ve spojení s jazykem Java, ve kterém je možné přímo exportovat šablony kódu služeb a urychlit tak vývoj. Nejrychlejší realizace choreografie pak dosáhneme exportem do vykonatelného BPEL, který je připraven přímo pro nasazení na aplikačním serveru JBoss[28].

Je zřejmé, že možnosti PI4SOA jsou celkem široké, především ve světě Java. Vzhledem k tomu, že pro tuto práci bylo zvoleno jako implementační prostředí .NET od společnosti Microsoft, byly využity jen exporty do standardizovaných formátů jako je WS-CDL, WSDL nebo BPMN. O návaznosti na tento nástroj při implementaci služeb bude pojednávat kapitola 6.

Nyní přistoupíme k vytváření formálního modelu choreografie v tomto nástroji. Jako první krok je potřeba definovat účastníky choreografie, v terminologii PI4SOA *Role*, a vztahy mezi nimi (*Relationship*). Opětovně budeme pracovat s byznys procesem žádosti o ceník, který byl představen v předchozí části. Model procesu přijetí objednávky nalezne čtenář v příloze.

Na obrázku 5.3 můžeme vidět definici účastníků a jejich vazeb mezi sebou v choreografii „Vyžádání ceníku“.

[27] Bližší informace o tomto nástroji naleznete na http://sourceforge.net/apps/trac/pi4soa/wiki.
[28] Bližší informace na http://www.jboss.org/.

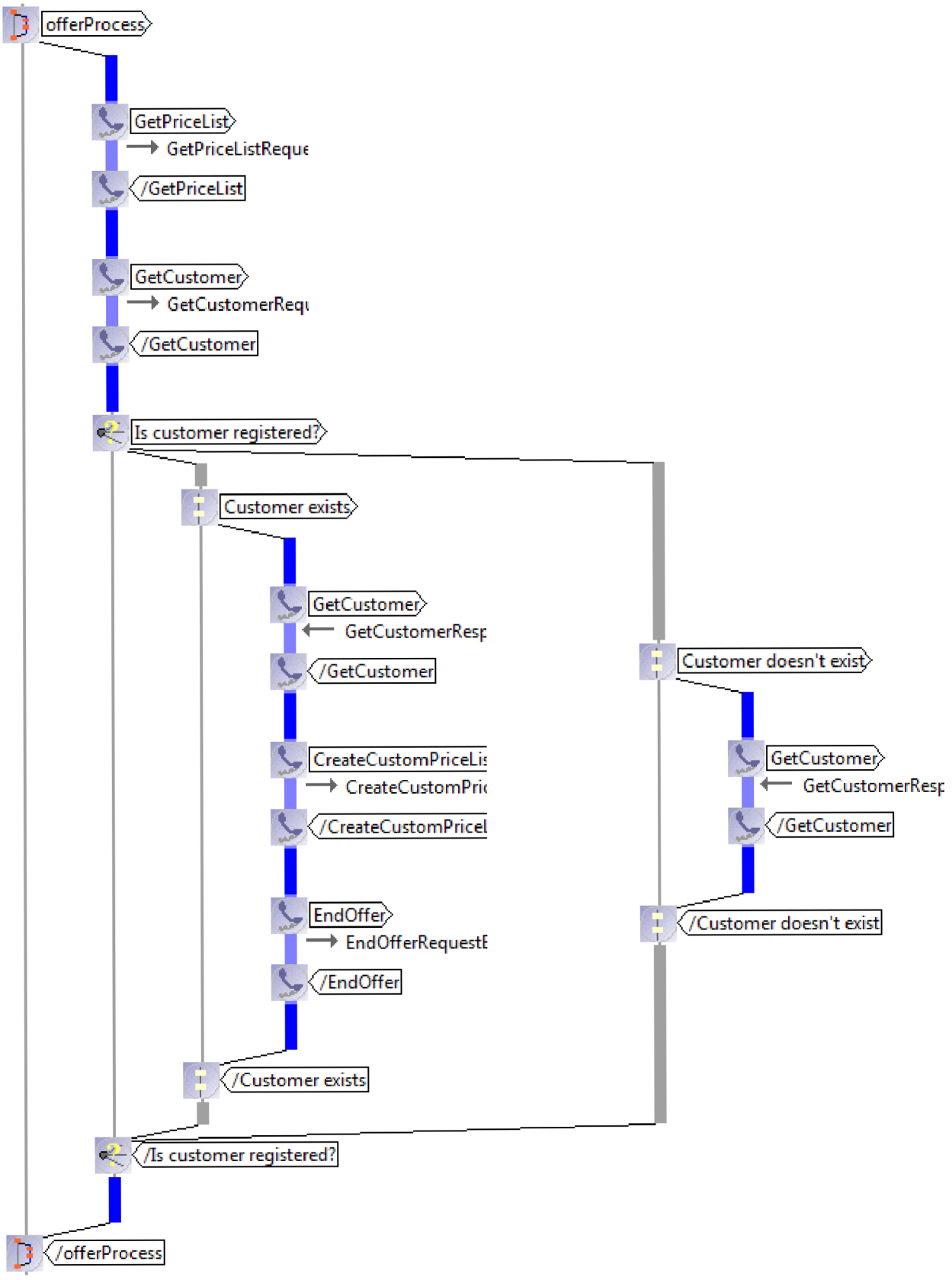


obr. 5.3: Definice *Roles* a *Relationships* v PI4SOA

Spolu s definováním vazeb je potřeba vytvořit i kanály (mají blízkou souvislost s kanály z π-kalkulu), které jsou v PI4SOA nazývány jako *Channel*. Ty mohou mít vlastní identifikátory, díky kterým je možné odlišovat různé instance choreografií. Jako poslední je potřeba definovat tzv. *Information types*, což jsou v podstatě typy zpráv, které se prostřednictvím kanálů posílají. V souvislosti s touto první choreografií jsem definoval tyto *Information types*:

- *CreateCustomPriceListRequest* – žádost o vytvoření upraveného ceníku
- *CustomPriceList* – upravený ceník
- *Customer* – objekt zákazníka
- *GetCustomerRequest* – žádost o zákazníka
- *GetPriceListRequest* – obecná žádost o ceník
- *URITokenType* – identifikátor kanálů

Nyní se přesuneme ke stěžejní části vytváření formálního modelu. Tou je vytvoření tzv. *Choreography Flow* neboli toku choreografie. Tento model popisuje to, jak se může choreografie vyvíjet s ohledem na příchozí a odchozí zprávy. Celý tok prvního byznys procesu můžeme vidět na obrázku 5.4.



obr. 5.4: Vývoj choreografie zapsaný v PI4SOA (*Choreography Flow*)

Nyní jsme dokončili všechny tři části tvorby modelu choreografie, tedy definici účastníků a vazeb, definici vyměňovaných typů a definici vývoje choreografie. Máme-li takový model dokončen, můžeme ho použít k testování choreografie, exportu kontraktů či získání standardního popisu WS-CDL.

5.3.2 Modelování choreografie v čistém π-kalkulu

Definovat formální model choreografie je samozřejmě možné i za přímého použití samotné procesní algebry. Z hlediska praktického použití je to však krajně nevhodné především z časových důvodů a při návrhu se používají spíše nástroje na formalizmech postavené. Pro účely této práce si takovou formu modelu ukážeme. V procesní algebře π-kalkul budeme definovat choreografii téhož byznys procesu, který nás provází od začátku této kapitoly, tedy „Žádost o ceník".

V tomto popisu budeme používat výraz podmínky, který v π-kalkulu zapíšeme uvnitř hranatých závorek.

Jako první definujeme proces účastníka, který v choreografii zastupuje zákazníka obchodu[29].

$$CustomerOperator'(PERSONAL\_ID, store) =$$

$$(priceListRequest, reply)($$

$$store\langle PERSONAL\_ID, priceListRequest, reply\rangle$$

$$.reply(customPriceList)$$

$$)$$

Tento proces nejprve odesílá po sdíleném kanálu skladu trojici, konstantu identifikátor zákazníka, objekt s požadavkem na ceník a nově vytvořený kanál, po kterém má přijít odpověď. Následně čeká na odpověď na tomto kanálu, která přijde v podobě ceníku.

Dalším je proces skladu.

$$Store'(store, crm, pricing) =$$

$$store(customerId, priceListRequest, replyCustomer)$$

$$.(reply)\big(crm\langle customerId, reply\rangle.reply(customerInfo)\big)$$

$$.[customer\ is\ registered](customPriceListRequest)$$

$$(pricing\langle customPriceListRequest, replyCustomer\rangle)$$

Proces skladu nejprve čeká na přijetí požadavku na ceník na sdíleném kanálu. Hned po přijetí odesílá žádost o údaje zákazníka na systém CRM a poté čeká na přijetí těchto údajů. Nyní je použita zmíněná podmínka. Pokud je v odpovědi obsažen zákazník, který je registrovaný, odesílá žádost o upravený ceník po sdíleném kanálu cenotvorby. Spolu se žádostí odesílá i soukromý kanál zákazníka, aby mu mohl být předán ceník. Podmínka je v tomto případě dosti abstrahovaná a v úplném popisu by bylo potřeba ji nahradit dalším procesem.

Nyní se dostáváme k procesu účastníka CRM.

---

[29] Jména psaná velkými písmeny značí konstanty.

$Crm'(crm) =$

$crm(customerId, replyStore)$

$.RetrieveCustomer(customerId))$

$.(customerInfo)(replyStore\langle customerInfo\rangle)$

Tento proces začíná rovněž čekáním na příchozí zprávu. Ve chvíli, kdy ji obdrží, spouští subproces získání údajů o zákazníkovi z databáze a poté tyto údaje odesílá po kanálu pro odpověď, který v úvodu obdržel. Subproces získání údajů o zákazníkovi zahrnuje i možnost, kdy zákazník dosud není registrován.

Posledním procesem je účastník Pricing.

$Pricing'(pricing) =$

$pricing(customPriceListRequest, replyCustomer)$

$.CreationOfPriceList(customPriceListRequest)$

$.(priceList)(replyCustomer\langle priceList\rangle)$

Tento proces dokončuje celou choreografii tím, že předává zákazníkovi ceník. Opět je iniciován příchozí zprávou na sdíleném kanálu. Poté pokračuje subprocesem vytvoření ceníku a končí odesláním tohoto ceníku po soukromém kanálu zákazníka.

Rád bych na tomto místě poznamenal, že zmíněné subprocesy mohou být (a v úplném popisu komunikujícího systému by měly být) dále definovány. Také je potřeba říci, že pokud chceme úplně popsat takovýto komunikující systém, musí být všechny definované procesy zahrnuty v paralelní kompozici například v nadřazeném procesu celého systému. Tím dosáhneme jejich nepřetržitého běhu a umožníme jim komunikovat mezi sebou. Cílem této části tedy nebylo poskytnout kompletní popis komunikujícího systému, ale ukázat, jak lze procesy zapisovat a zdůraznit, že využití procesní algebry ve spojení s nějakým nástrojem je o mnoho efektivnější.

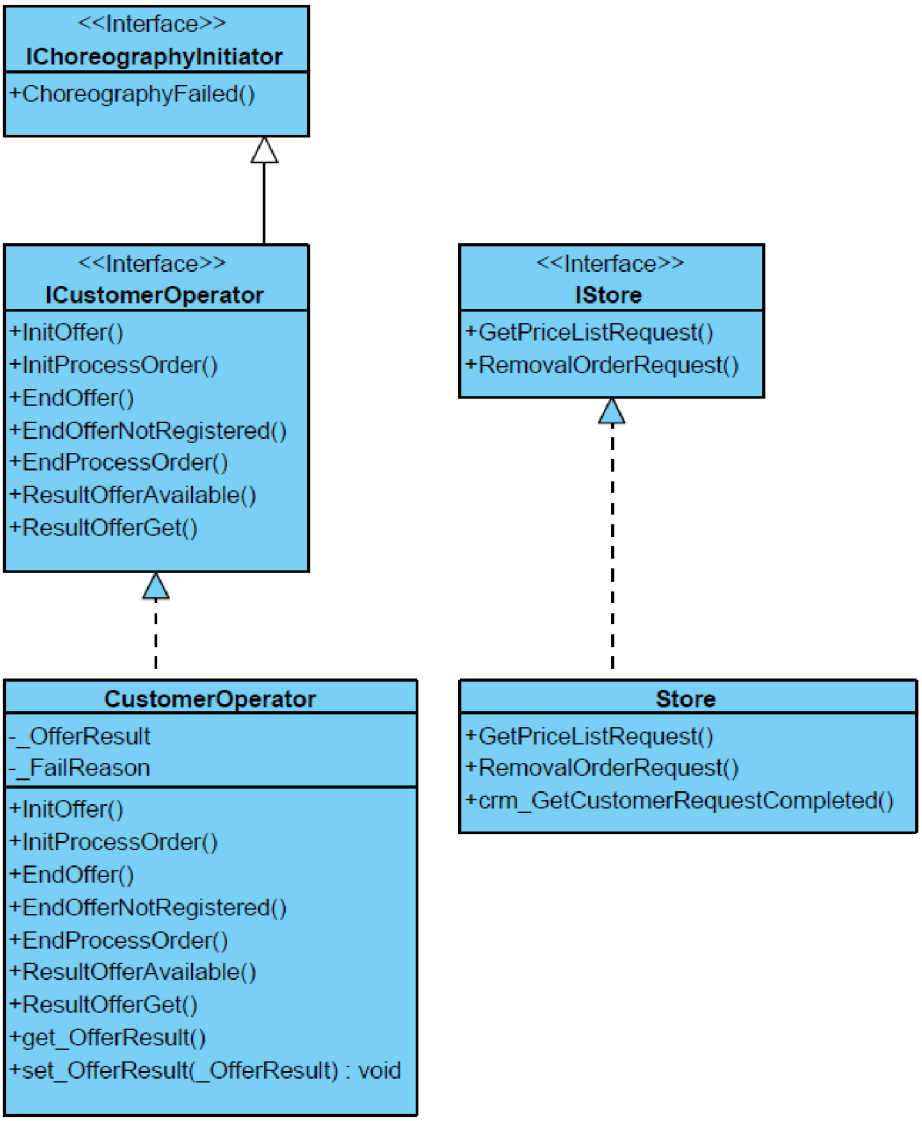
5.4 Návrh implementace služeb

Nyní již máme konceptuální model architektury v podobě BPMN diagramu i formální model choreografií v PI4SOA. Poslední částí, kterou při návrhu SOA není možné zanedbat, bude návrh samotné implementace. Nyní tedy definujeme tzv. "kandidátní služby" a specifikujeme jejich komunikační rozhraní.

Kandidátní služba bývá identifikována při návrhu a v budoucí implementaci má zapouzdřovat část funkcionality systému. Dá se říci, že v našem návrhu jsou kandidátní služby ekvivalentem pro účastníky choreografie a vyjmenovány jsou na začátku kapitoly 5.2. Mírná změna se týká pouze budoucí služby *Customer*, kde je potřeba vysvětlit, že tato služba nebude v systému zákazníka přímo reprezentovat (jak tomu bylo při návrhu choreografií), ale bude zprostředkovávat komunikaci mezi ním a servisně orientovaným systémem.

Nyní již máme definované služby a můžeme specifikovat jejich komunikační rozhraní. V tomto případě budeme vycházet z formálních modelů vytvořených v PI4SOA. Komunikační rozhraní služeb

je definováno pomocí třídního diagramu UML. Na obrázku 5.5 můžeme vidět výřez z třídního diagramu celého systému, kompletní diagram pak čtenář nalezne v příloze.



obr. 5.5: Výřez z třídního diagramu systému, definuje rozhraní služeb

Komunikační rozhraní (kontrakt) služby *CustomerOperator* je v diagramu definováno rozhaním *ICustomerOperator*, kontrakt služby *Store* pak pomocí rozhraní *IStore*. Zároveň tato část diagramu znázorňuje, že rozhraní *ICustomerOperator* rozšiřuje rozhraní *IChoreographyInitiator*, které určuje, že služba zastupující zákazníka bude zároveň iniciovat choreografii a budou jí předávány výjimky v choreografii vzniklé. O tomto řešení správy chyb a výjimek bude řeč v kapitole 6.

Komponentový diagram, který znázorňuje závislosti mezi službami, byl z důvodu svého rozsahu umístěn do příloh.

6 Implementace ukázkové architektury SOA

V této závěrečné kapitole se budeme věnovat samotné implementaci navržené architektury. Ukážeme si některé techniky používané při implementaci SOA v prostředí Microsoft .NET a také různé mechanizmy, jak postupovat, abychom ošetřili vzniklé chyby a výjimky.

Při implementaci budeme plně dodržovat specifikace, které vznikly ve fázi návrhu a za pomoci nástroje PI4SOA.

V reálném systému by bylo výhodnější některé části procesu modelovat pomocí orchestrací. S ohledem na téma práce a z důvodu předvedení choreografie jsou však všechny části procesu modelovány jako spolupráce služeb (choreografie). Implementoval jsem ty služby, které byly v návrhové fázi označeny jako kandidátní. V prvním procesu žádosti o ceník tedy budou spolupracovat tyto služby:

- *CustomerOperator* – prostředník mezi zákazníkem (příp. jeho systémem) a elektronickým obchodem
- *Store* – představuje službu skladového hospodářství
- *CRM* – služba, která zprostředkovává a udržuje údaje o zákaznících
- *Pricing* – služba cenotvorby, vyžaduje údaje o zákazníkovi, aby mu mohla vytvořit ceník na míru

Druhý proces přijetí objednávky je tvořen spoluprácí výše zmíněných služeb spolu s dalšími službami:

- *Orders* – služba, která přijímá objednávku a dále s ní operuje
- *Invoicing* – vytváří a spravuje faktury
- *Expedition* – stará se o vyskladnění zásob a o komunikaci se zásilkovou společností
- *DeliveryService* – služba zásilkové společnosti

Všech těchto osm služeb dohromady tvoří celý servisně orientovaný systém elektronického obchodu, který na základě zákazníkovy identity a bonity vytvoří ceník, ze kterého zákazník vytvoří objednávku. Pokud objednávka splňuje stanovené podmínky, systém ji přijme a vyexpeduje objednané zboží. To je prostřednictvím zásilkové společnosti odesláno zákazníkovi.

6.1 Implementační prostředí

Téma této práce není svou povahou vázáno na specifické prostředí od jednoho výrobce, naopak jednou z hlavních výhod tohoto přístupu k tvorbě software je jeho nezávislost na implementačním prostředí. I přesto, že je možné implementaci provést téměř v jakémkoliv programovacím jazyce, ne

všechny poskytují dostatečný komfort a podporu vývoje. Jako dvě nejpoužívanější platformy při vývoji SOA se uplatňuje Java EE[30] a framework .NET[31] od společnosti Microsoft.

Jako implementační prostředí byl zvolen framework .NET verze 4 a jazyk C#, a to z důvodu mých předchozích zkušeností s oběma. Tento framework poskytuje již od verze 3.0 (vydána roku 2006) knihovnu pro komunikaci po síti s názvem WCF (Windows Communication Foundation), která sjednocuje všechny nástroje pro vzdálenou komunikaci a je navržená s ohledem na servisně orientovaná řešení. Protože SOA je založená především na standardech, tak knihovna WCF z velké části implementuje standardy WS-I a WS-* (viz kapitola 2.2). Při implementaci jsem čerpal z [19].

Rozhraní služeb se v knihovně WCF specifikuje pomocí obvyklého rozhraní (*interface*), které známe z objektově orientovaného programování. Rozhraní samotné, stejně jako jeho metody, musí být navíc opatřeny atributy, které značí, že daná metoda je vystavena prostřednictvím webové služby žadatelům služby. Použití atributů při konfiguraci služeb je velkou výhodou této knihovny. Dosahuje se tím efektivního oddělení výkonného kódu od konfigurace služeb a komunikačních kanálů. Nejlepší bude malá ukázka takového přístupu.

```
[ServiceContract(SessionMode=SessionMode.Required)]
interface IStore
{
      [OperationContract(IsOneWay=true)]
      void GetPriceListRequest(GetPriceListRequest request);

      [OperationContract]
      RemovalOrderResponse RemovalOrderRequest(RemovalOrderRequest
request);
}
```

Zmiňované atributy jsou uvedené v hranatých závorkách. Atribut `ServiceContract` označuje dané rozhraní (ve smyslu objektového programování) jako rozhraní webové služby a nastavuje její chování. Jeho parametr `SessionMode` v tomto případě specifikuje, že každý kanál použitý pro komunikaci se službou musí podporovat *session*[32]. Session slouží pro korelaci mezi zprávami, které jsou pomocí tohoto mechanizmu shlukovány do konverzací. Je možné tak například nastavit, že chceme pro každého žadatele služby vytvořit novou instanci služby nebo naopak chceme, aby všichni žadatelé sdíleli jedinou instanci.

Atribut `OperationContract` značí, že dané metody spadají do rozhraní služby a jsou tedy zařazeny do kontraktu služby v jazyce WSDL, který je na základě tohoto rozhraní generován. I tento atribut může mít mnoho parametrů. V příkladu je uveden parametr, který dané metodě nastavuje, že nevrací žádnou hodnotu. Žadatel této metody tedy nečeká na její zpracování ani odpověď.

Implementací rozhraní opatřeného takovými parametry získáme v prostředí .NET implementaci webové služby, která je schopná o sobě zprostředkovat informace v podobě kontraktu WSDL.

---

[30] Java Platform, Entreprise Edition byla vytvořená společností Sun, více informací naleznete na http://download.oracle.com/javaee/.

[31] Více informací o frameworku .NET naleznete na http://msdn.microsoft.com/en-us/netframework/aa496123

[32] Session je v tomto případě mechanizmus pro zajištění korelace mezi zprávami. Nejedná se tedy o session, tak jak je známe v souvislosti s protokolem HTTP.

6.2 Popis implementace

Implementace architektury elektronického obchodu byla pojata jako ukázková, proto spíše než k praktickému použití slouží k demonstraci choreografie služeb a principu servisní orientace. Některé podrobnosti týkající se například samotné objednávky nebo údajů o zákaznících jsou schválně zanedbány, protože pro naše účely nemají smysl.

Architektura sestává z osmi služeb, které byly definovány na začátku této kapitoly. O komunikaci se zákazníkem se stará služba *CustomerOperator*. Tu může využívat buď přímo zákazníkův systém, protože služba vystavuje své neměnné servisně orientované rozhraní, které zahrnuje operace vyžádání ceníku a zpracování objednávky. Nebo je možné službu volat prostřednictvím aplikace s grafickým rozhraním, která byla vytvořena současně s ukázkovým příkladem. Aplikace nese název *Terminal* a volá v novém vlákně běžícím na pozadí metody služby *CustomerOperator* a zobrazuje výsledky. Více o této aplikaci se dozvíte v kapitole 6.3.

Služba *CustomerOperator* spojuje komunikaci mezi celou choreografií elektronického obchodu a samotným zákazníkem. Na začátku této kapitoly bylo řečeno, že vysvětlíme, jak byla řešena správa chyb a výjimek. To je v servisně orientovaném světě relativně náročný úkol, obzvláště pak u choreografií. Právě za tímto účelem je využita služba *CustomerOperator*, která potažmo představuje „orchestrátora" choreografie, protože celou choreografii iniciuje a zároveň je na ní delegována správa chyb a výjimek. Vznikne-li tedy v nějaké službě výjimka, je odeslána právě iniciátorovi choreografie a ten ji většinou zpracuje a reportuje dále uživateli, se kterým je v kontaktu. Stejně funguje i ukončení choreografie, kdy je výsledek předán iniciátorovi a ten ji předá uživateli.

Při implementaci jsem využíval výstupů z nástroje PI4SOA, ve kterém byl vytvořen formální model. Využíval jsem tzv. „Contract first" přístup, což znamená, že na základě kontraktu služeb (většinou v jazyce WSDL) implementujeme služby samotné. Kontrakty služeb WSDL jsem proto získal z modelu PI4SOA a použil je pro automatické vytvoření koster služeb.

Dále bylo potřeba implementovat tyto metody a datové typy, které představují objekty komunikace. Datové typy (potažmo typy zpráv), které si služby vyměňují, jsou definovány vždy na straně služby, ke které se vztahují. Například služba *Store* definuje typ *GetPriceListRequest*, který představuje zprávu s požadavkem na ceník. Má-li jiná služba používat tento datový typ (například ho chce odeslat ve zprávě), musí být ustanovena tzv. *service reference* na službu *Store*. To zajistí, že se na straně druhé služby automaticky (díky vývojovému prostředí) vygenerují všechny sdílené typy služby *Store*. V reálném servisně orientovaném řešení by však bylo výhodnější typy i rozhraní služby iniciátora definovat na straně centrálního registru služeb. O tomto možném rozšíření si povíme v podkapitole 6.4.

Některé služby (konkrétně *CRM* a *Pricing*) mají vlastní databázi[33], ze které získávají data. Na straně *CRM* je uložena databáze kontaktů zákazníků a jejich bonity a na straně služby *Pricing* jsou pak ceny jednotlivých položek. Přístup k databázím je zajištěn pomocí technologie *LINQ to SQL*, která podporuje jednoduché dotazování databáze bez nutnosti použití SQL syntaxe.

V této implementaci SOA je zahrnuta i mobilita komunikačního kanálu, o které jsme mluvili v souvislosti s π-kalkulem v kapitole 4.3.2. Některé typy zpráv jsou v hierarchii dědičnosti potomky

[33] Využívána je databáze MS SQL EXPRESS

typu *ChoreographyMessage*, který v sobě zahrnuje atribut odkazu na službu, která choreografii iniciovala. Toho se využívá právě v situacích, kdy dojde k selhání a je potřeba choreografii ukončit a na situaci upozornit i zákazníka prostřednictvím služby, skrze kterou se systémem komunikuje (tato služba je zároveň iniciátorem choreografie). Metody, které služba iniciátora (tj. orchestrátora) vystavuje, jsou popsány pomocí rozhraní *IChoreographyInitiator* (viz kapitola 5.4), které je v podobě knihovny distribuováno ke každé službě. Každá služba tedy v případě chyby ví, kam se má obrátit a jakou metodu volat.

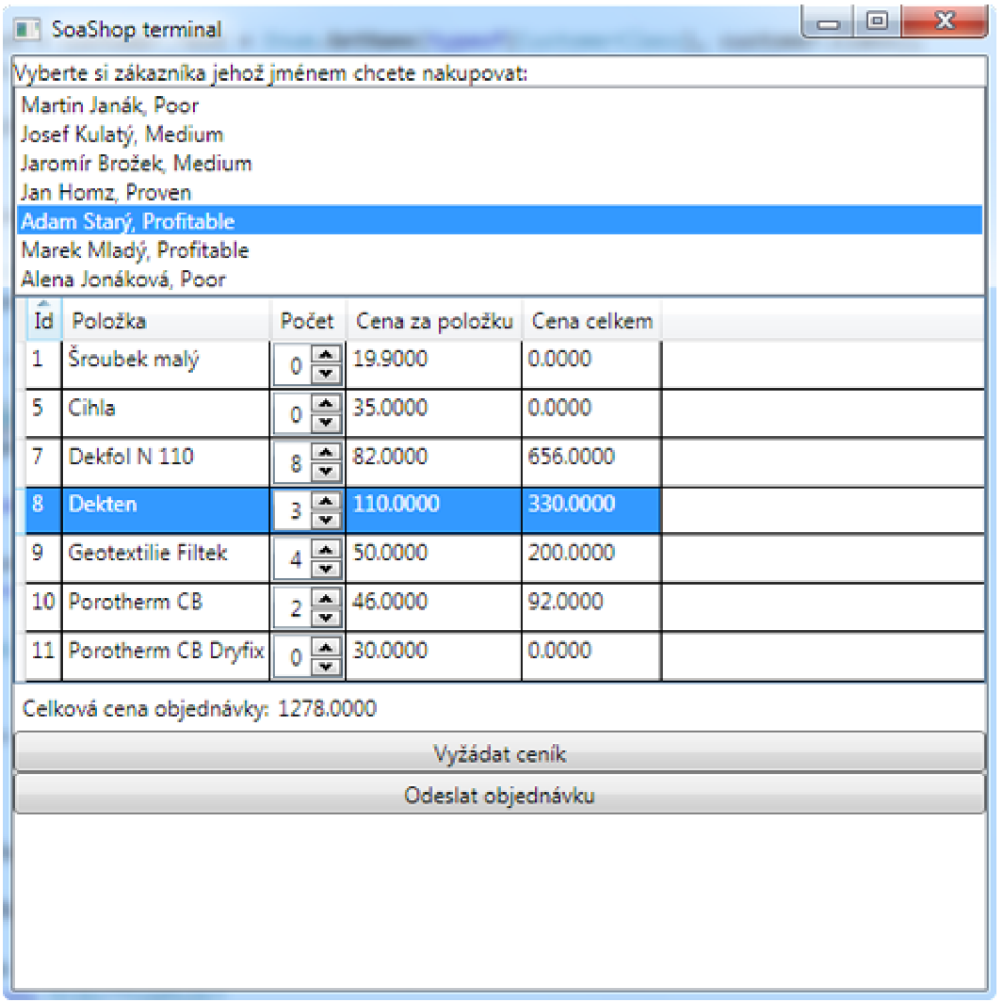
6.3 Aplikace Terminal

Účelem této aplikace je plně demonstrovat možnosti servisně orientované architektury. Tato aplikace poskytuje jednoduché uživatelské rozhraní pro žádost o ceník, tvorbu objednávky a její odeslání. Tyto operace může uživatel provádět jménem jednoho z několika přednastavených zákazníků a ověřit si tak, že zákazník obdrží ceny na základě svého zařazení do třídy. Zákazníka si vždy před zahájením byznys procesu vybere ze seznamu, kde je zobrazeno jeho jméno i s třídou bonity do které spadá. Zároveň aplikace poskytuje základní monitorovací funkce choreografie.

První takovou funkcí je sledování obsahu odchozích a příchozích zpráv v čistém XML. Ty se zobrazují ve vyhrazené záložce.

Druhá funkce pak uživateli umožňuje sledovat tok předávání zpráv mezi službami. To je zobrazeno v textovém režimu, kde formou log záznamů vypisuje předávání zpráv i s jejich typy.

Aby aplikace neblokovala uživatelské rozhraní, jsou všechna volání služeb realizována na novém vlákně. Některé choreografie totiž mohou trvat delší dobu. Obecně je však tento přístup více doporučovaný.



obr. 6.1: Ukázka tvorby objednávky v aplikaci Terminal

6.4 Možná rozšíření

V předchozích odstavcích jsem zmínil některá zjednodušení v implementaci. Jedním z nich byla definice typů zpráv, které jsou nyní definovány vždy pouze na straně služby, se kterou nějak souvisí a ostatní služby je získávají pomocí *service reference*. Je-li potřeba některé typy distribuovat ke všem službám, musíme tyto typy ke službě přidat ve formě knihovny.

V praxi to není ideální řešení, už jenom z důvodu komplikovanosti udržování aktuální verze sdílených knihoven u všech služeb. Ve větších SOA řešeních se za tímto účelem zavádí centrální registr služeb, který udržuje kontrakty všech služeb v systému a zároveň shromažďuje i definice typů v systému vyměňovaných. Ty jsou pak definovány většinou ve formě XSD schémat.

Implementace takového registru by umožnila i větší propojenost s nástrojem PI4SOA. Bylo by pouze potřeba implementovat rozšíření, které by zaznamenávalo vyměňované zprávy v daném formátu. PI4SOA totiž umožňuje takové záznamy importovat a na jejich základě automaticky verifikovat implementaci oproti modelu choreografie.

Závěr

Principy servisní orientace jsou dnes uplatňovány v mnoha organizacích, ať už ve formě jednoúčelových služeb nebo komplexních servisně orientovaných řešeních. Toho si jsou vědomi i výrobci software a velmi dobře tuto oblast podporují. Pro návrh a implementaci kompozic služeb tak existuje mnoho nástrojů. Ve většině případů je ale úsilí výrobců věnováno kompozicím typu orchestrace, které je možné popisovat pomocí workflow a je jednoduché je řídit. Méně pozornosti je pak věnováno kompozicím typu choreografie, které svým chováním mohou připomínat multiagentní systémy.

Tato práce představila servisně orientovanou architekturu a zmapovala dostupné prostředky pro popis choreografií v ní. Jejím dalším cílem bylo ukázat, že můžeme pomocí vhodných popisů, založených na matematickém formalizmu, choreografie verifikovat a dosahovat tak bezpečných a spolehlivých řešení. Na základě odborné literatury jsem čtenáři vysvětlil specifika servisně orientované architektury, a to hned z několika úhlů pohledu. Postupně jsem se zabýval hlavními komponentami SOA, jako jsou služby a zprávy. Dále jsem představil různé standardy, na kterých servisně orientovaná řešení stojí, až jsem se dostal k jádru práce, kterým byly kompozice služeb. Tato část práce přehledně představila prostředky pro popis choreografií služeb, rozdělené do tří skupin podle toho, jak kvalitní popis poskytují. Zabýval jsem se abstraktními popisy a popisy, které jsou de facto považovány za standardy a především popisy založenými na formálním matematickém modelu.

Právě poslední kategorii bylo věnováno více prostoru. Tato část práce vysvětlila vlastnosti dvou často používaných formalizmů pro popis souběžných procesů a na praktických příkladech demonstrovala jejich použití. Těmito formalizmy jsou Petriho sítě a procesní algebra π-kalkul. V této oblasti práce přinesla ukázku možných využití těchto formalizmů, jak pro návrh choreografií, tak i pro jejich verifikaci. Je zajímavé, že v oblasti SOA zůstává využití těchto prostředků pouze na experimentální a akademické úrovni, tedy alespoň zatím.

Všechny teoretické znalosti, nástroje a představené standardy byly aplikovány při návrhu a implementaci ukázkové architektury SOA, která implementuje byznys procesy objednávky zboží, fakturace a skladů elektronického obchodu. Implementace architektury má za hlavní cíl prakticky představit servisní orientaci a choreografii služeb. Služby jsou proto schválně navrhnuté jako spolupracující mezi sebou, i když by v reálné implementaci bylo výhodnější je řídit nadřazeným prvkem (tedy služby „orchestrovat“). Přínosem této implementace je využití formálního modelu při návrhu architektury, demonstrace celého vývojového procesu SOA a v neposlední řadě ukázka reálné implementace v prostředí Microsoft .NET.

Téma servisní orientace a servisně orientované architektury je tak obsáhlé, že se nabízí hned několik námětů na pokračování této práce. Při implementaci ukázkové architektury jsem zvažoval, zda v demonstračním příkladu realizovat rozšíření webových služeb WS-Transaction a implementovat tak transakční zpracování napříč choreografií. Přestože jsem od tohoto záměru nakonec upustil, myslím si, že samotné téma transakcí v SOA je velmi zajímavé a hodilo by se zpracovat v návaznosti na tuto práci.

Literatura

[1] Erl, T.: SOA Principles of Service Design, Upper Saddle River, Prentice Hall 2008, ISBN 978-013-2344-821

[2] Erl, T.: SOA Kompletní průvodce, Computer Press, Brno, 2009, ISBN 978-80-251-1886-3

[3] W3C: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer [online]. [cit. 2011-03-3]. URL: http://www.w3.org/TR/wsdl20-primer/

[4] W3C: SOAP Version 1.2 Part 1 [online]. [cit. 2011-03-3]. URL: http://www.w3.org/TR/soap12-part1/

[5] Weiss, P., Rychlý, M.: ARCHITEKTURA ORIENTOVANÁ NA SLUŽBY, NÁVRH ORIENTOVANÝ NA SLUŽBY, WEBOVÉ SLUŽBY [online]. [cit. 2011-03-18]. URL: http://www.fit.vutbr.cz/research/view_project.php?file=%2Fproj%2F378%2FSOA_SOAD_WS.pdf&id=378

[6] Knap, P.: Orchestrace a choreografie služeb [online]. [cit. 2011-03-10]. URL: www.cssi.cz/cssi/system/files/all/00knap.pdf

[7] Rosen, M.: Orchestration or Choreography? [online]. [cit. 2011-03-12]. URL: http://www.bptrends.com/publicationfiles/04-08-COL-BPMandSOA-OrchestrationorChoreography-%200804-Rosen%20v01%20_MR_final.doc.pdf

[8] Peltz, C.: Web Services Orchestration and Choreography [online]. [cit. 2011-03-12]. URL: http://www.computer.org/portal/web/csdl/abs/mags/co/2003/10/rx046abs.htm

[9] Vašíček, P.: Úvod do BPMN [online]. [cit. 2011-03-23]. URL: http://bpm-sme.blogspot.com/2008/03/3-uvod-do-bpmn.html

[10] Barros, A., Dumas, M., Oaks, P.: A Critical Overview of the Web Services Choreography Description Language (WS-CDL) [online]. [cit. 2011-03-25]. URL: http://www.bptrends.com/publicationfiles/03-05%20WP%20WS-CDL%20Barros%20et%20al.pdf

[11] BPELJ: BPEL for Java technology [online]. [cit. 2011-03-23]. URL: http://www.ibm.com/developerworks/library/specification/ws-bpelj/

[12] W3C: Web Services Choreography Description Language Version 1.0 [online]. [cit. 2011-03-23]. URL: http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

[13] Bouché, P.: WS-CDL and Pi-Calculus [online]. [cit. 2011-03-26]. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.5028&rep=rep1&type=pdf

[14] Lohmann, N., Kopp, O., Leymann, F., Reisig, W.: Analyzing BPEL4Chor: Verification and Participant Synthesis [online]. [cit. 2011-03-26]. URL: http://www2.informatik.hu-berlin.de/top/download/publications/LohmannKLR2007_wsfm.pdf

[15] Rychlý, M.: Formální specifikace architektur informačních systémů [online]. [cit. 2011-03-27]. URL: http://www.fit.vutbr.cz/~rychly/public/docs/formal-architectures/xhtml/formarch.xhtml

[16] Rychlý, M.: Formal-based Component Model with Support of Mobile Architecture, Brno, CZ, UIFS FIT VUT, 2010, s. 120. Dostupné z URL: http://www.fit.vutbr.cz/research/view_pub.php?file=%2Fpub%2F9174%2Fthesis.pdf&id=9174

[17] Milner, R.: The Polyadic π-Calculus: a Tutorial [online]. [cit. 2011-04-08]. URL: http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-180/ECS-LFCS-91-180.ps

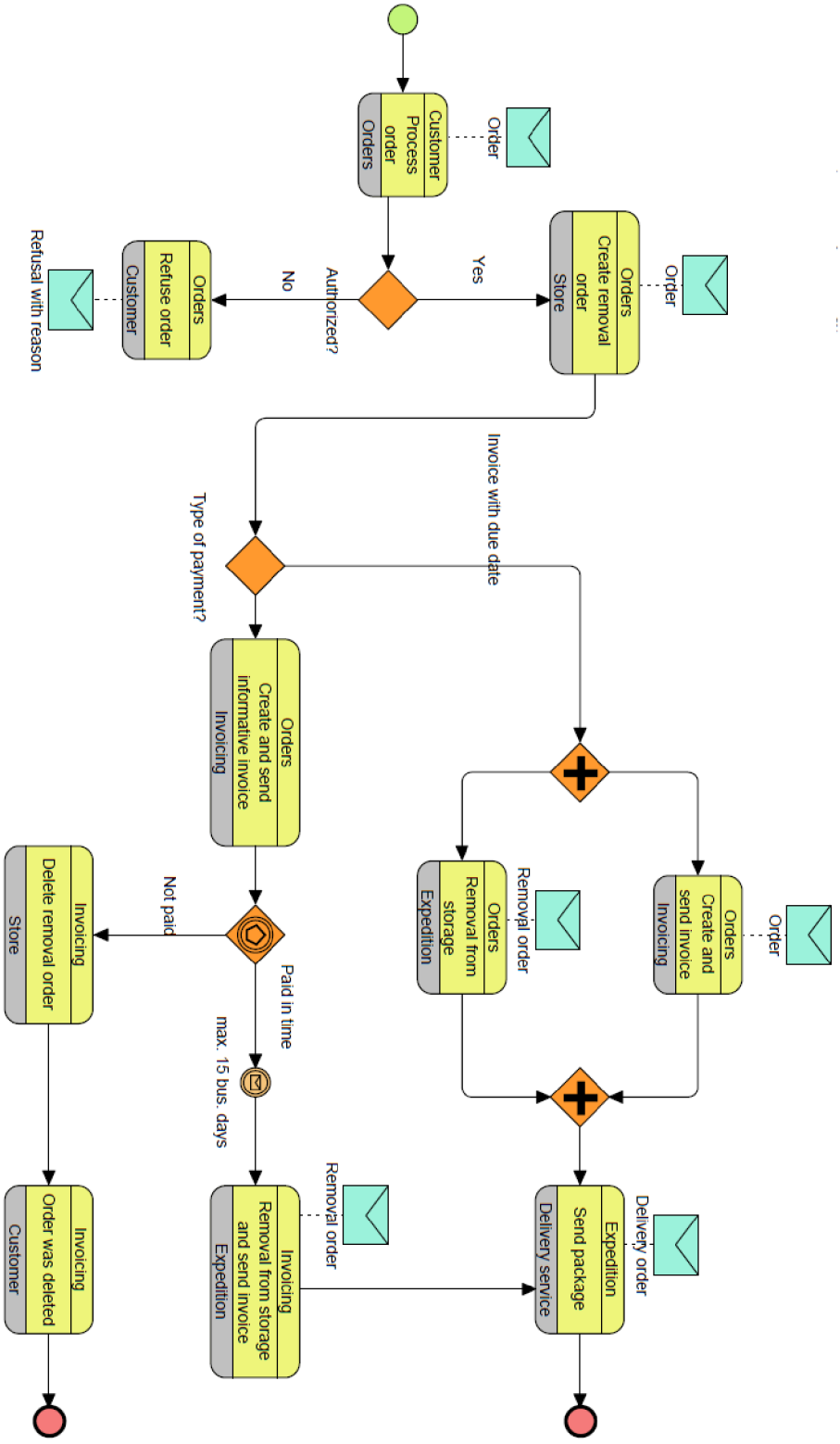
[18] Češka, M., Marek, V., Novosad, P., Vojnar, T.: Petriho sítě, Brno, CZ, UITS FIT VUT, 2009. Dostupné z URL: http://www.fit.vutbr.cz/study/courses/PES/public/Pomucky/PES_opora.pdf

[19] Pathak, N.: Pro WCF 4 Practical Microsoft SOA Implementation, Apress, 2011, ISBN 978-1-4302-4302-3368-5

Seznam příloh

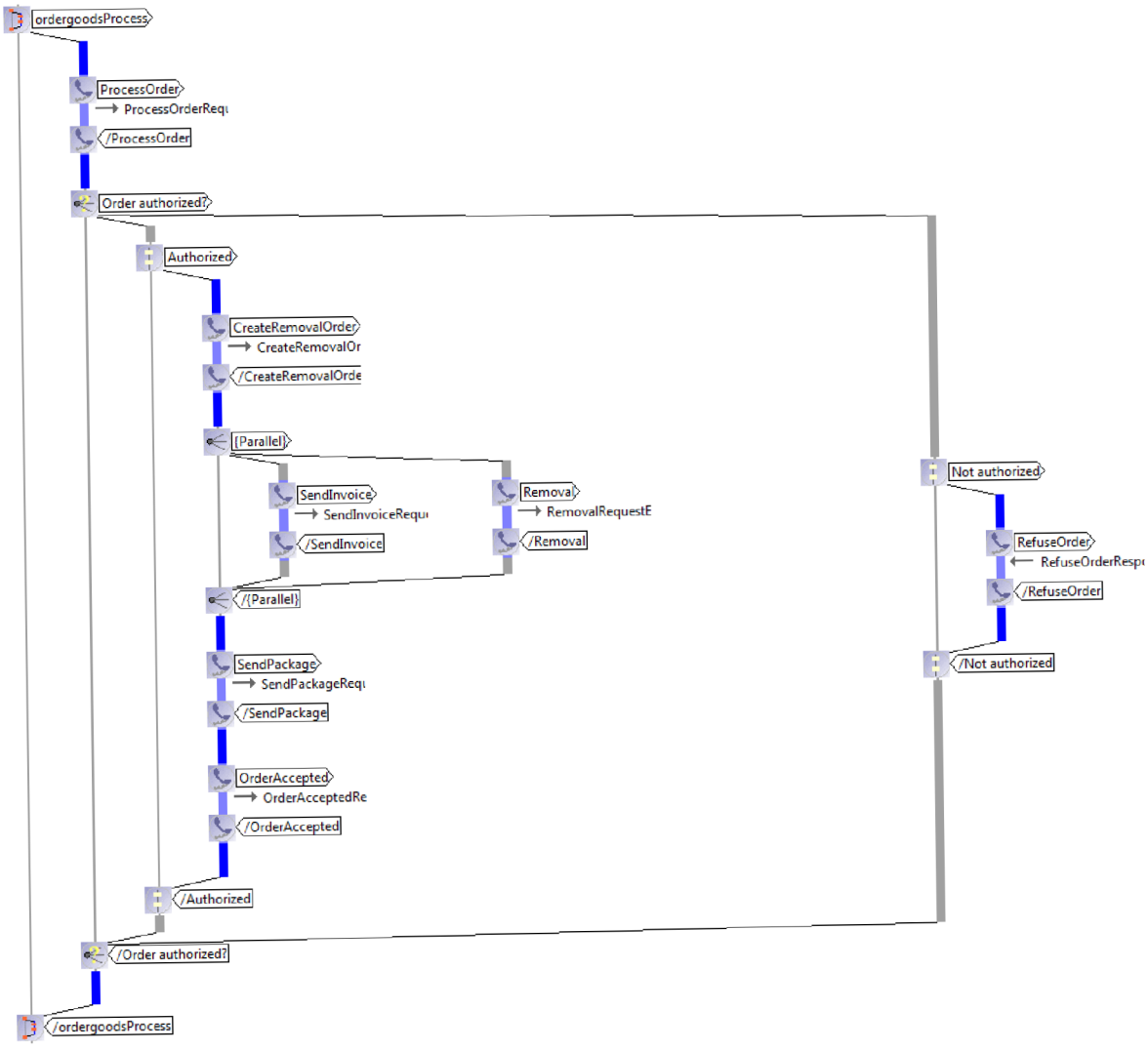
Přílohy

Příloha 2. BPMN diagram procesu přijetí objednávky



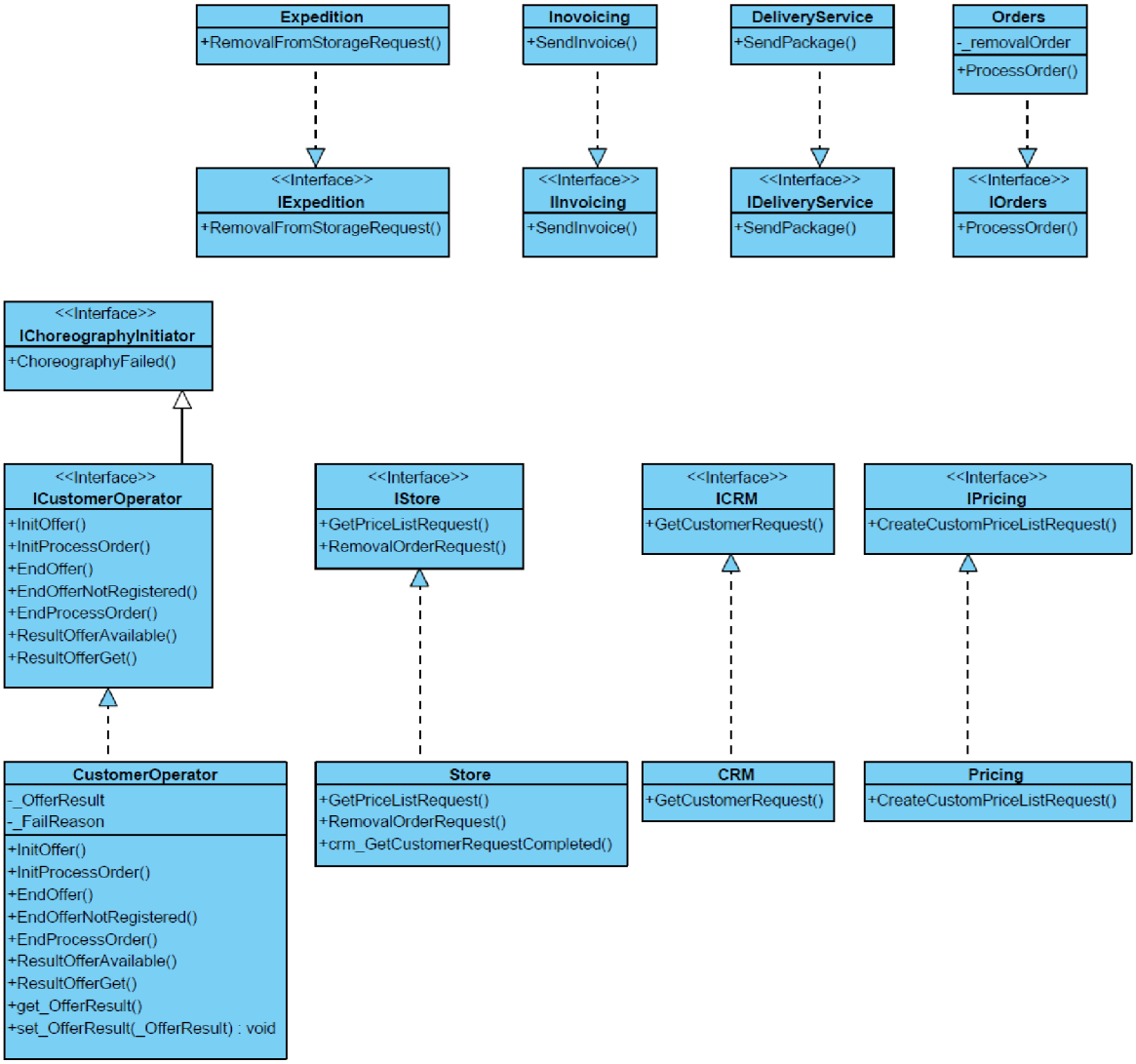
obr. 6.2: BPMN diagram procesu přijtí objednávky a odeslání zboží

Příloha 3. *Choreography flow* model procesu přijetí objednávky v PI4SOA



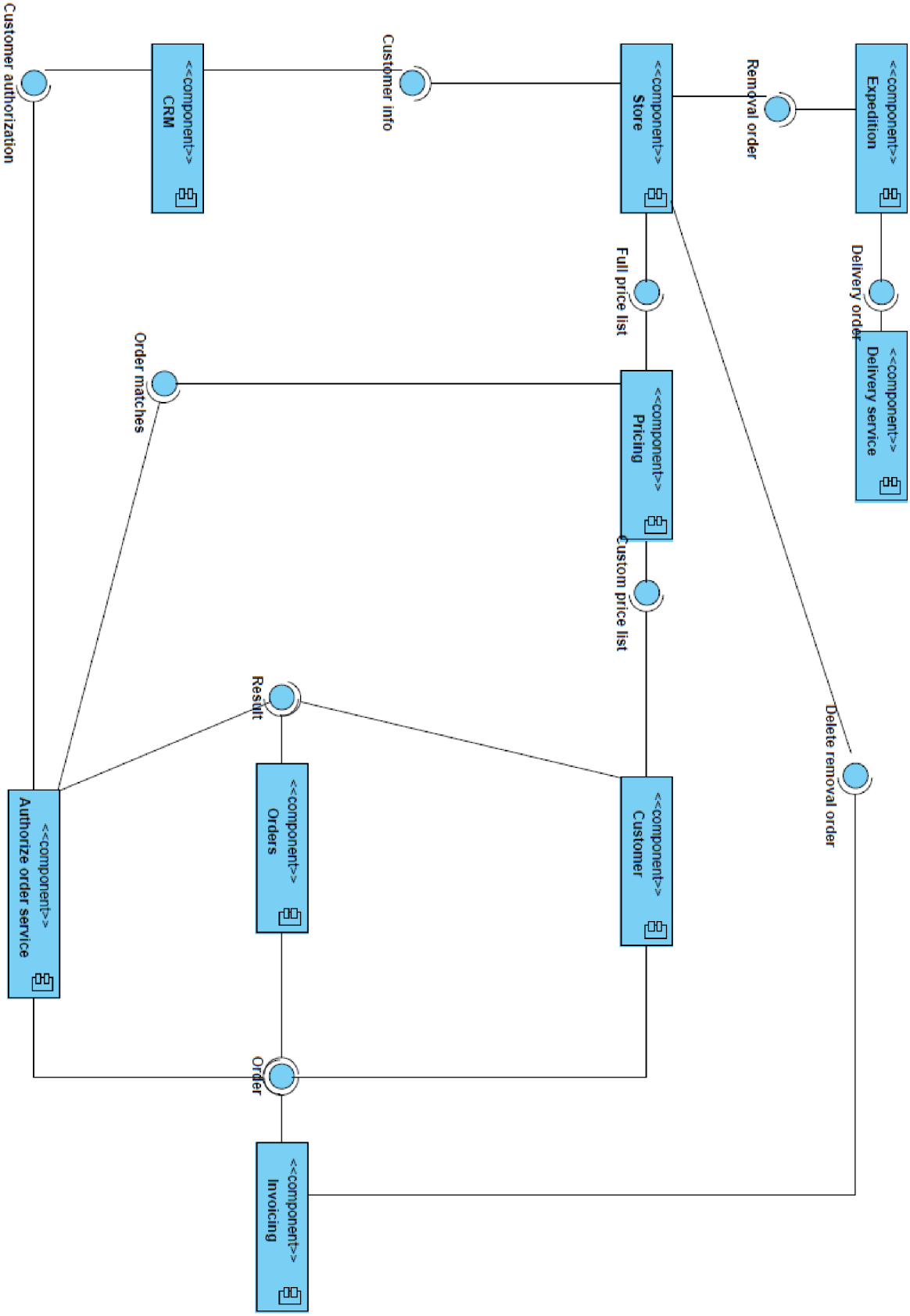
obr. 6.3: Formální model choreografie byznys procesu přijetí objednávky v PI4SOA

Příloha 4. Třídní diagram celého systému



obr. 6.4: Třídní diagram znázorňující rozhraní služeb

Příloha 5. Komponentový diagram celého systému



obr. 6.5: Komponentový diagram celého systému