UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Sebastjan Hrovat

RAZVOJ GENERIČNE STORITVE ZA PODPORO SESTAVLJENIH UPORABNIŠKIH OPRAVIL V POSLOVNIH PROCESIH

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Marjan Krisper

Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01616/2009
Datum: 15.11.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **SEBASTJAN HROVAT**

Naslov: **RAZVOJ GENERIČNE STORITVE ZA PODPORO SESTAVLJENIH UPORABNIŠKIH OPRAVIL V POSLOVNIH PROCESIH**

DEVELOPMENT OF GENERIC SERVICE FOR COMPLEX USER TASK SUPPORT IN BUSINESS PROCESSES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V okviru diplomske naloge obravnavajte področje avtomatizacije poslovnih procesov s sodobnimi pristopi in tehnologijami s povdarkom na storitveno usmerjeni arhitekturi. Predstavite opravila v poslovnih procesih s povdarkom na uporabniških sestavljenih opravilih. Identificirajte vzorce sestavljenih uporabniških pravil, ki jih je smiselno implementirati. Uporabite jezik BPEL. Izdelajte storitev, ki bo omogočala enostavno spreminjanje množice vzorcev.

Mentor:

prof. dr. Marjan Krisper

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Sebastjan Hrovat,

z vpisno številko 63010043,

sem avtor diplomskega dela z naslovom:

RAZVOJ GENERIČNE STORITVE ZA PODPORO SESTAVLJENIH UPORABNIŠKIH OPRAVIL V POSLOVNIH PROCESIH

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek)

  prof. dr. Marjana Krisperja

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 23. 11. 2009 Podpis avtorja: _____

Zahvala

Zahvaljujem se prof. dr. Marjanu Krisperju za mentorstvo, vodenje in usmerjanje pri izdelavi diplomskega dela.

Zahvaljujem se tudi dr. Ani Šaša, ki mi je pri delu pomagala s koristnimi strokovnimi nasveti.

Hvala staršem za podporo ter vsem, ki so me v času študija spodbujali in mi pomagali.

Kazalo

Seznam uporabljenih kratic in simbolov

ERP - Enterprise Resource Planning je sistem za upravljanje s funkcijami in informacijami v podjetju.
CRM - Customer Relationship Management je sistem, ki podpira interakcijo s strankami.
SCM - Supply Chain Management je sistem za upravljanje z oskrbovalno verigo v podjetjih.
WS-BPEL (ali krajše BPEL) - Web Services Business Process Execution Language je standard organizacije OASIS, ki specificira jezik za interakcije med spletnimi storitvami.
UDDI - Universal Description, Discovery and Integration je platformno neodvisen imenik za objavo in iskanje spletnih storitev.
XML - eXtensible Markup Language ali razširljiv označevalni jezik je nabor pravil za kodiranje elektronskih dokumentov.
BPMN - Business Process Modeling Notation je grafična notacija za modeliranje poslovnih procesov.
UML - Unified Modeling Language je grafični jezik za vizualizacijo, specifikacijo, konstruiranje in dokumentiranje programskih rešitev.
EPC - Event-Driven Process Chain je tehnika modeliranja poslovnih procesov.
eEPC - Extended Event-Driven Process Chain razširjena tehnika modeliranja poslovnih procesov EPC.
OASIS - Organization for the Advancement of Structured Information Standards je neprofitna organizacija, ki skrbi za razvoj in konvergenco odprtih standardov globalne informacijske družbe.
XPath - XML Path je jezik za naslavljanje XML dokumenta.
XSLT - eXtensible Stylesheet Language Transformations je jezik za transformacijo XML dokumentov v druge XML dokumente ali v (X)HTML dokumente.
XPointer - XML Pointer Language je jezik za naslavljanje delov XML dokumenta.
(X)HTML - eXtensible Hypertext Markup Language je vrsta jezika XML, ki razširja jezik HTML.
RESTful Web Services - Representational State Transfer Web Services je ena od vrst spletnih storitev.
SOAP - Simple Object Access Protocol je specifikacija za izmenjavo strukturiranih sporočil med spletnimi storitvami preko računalniškega omrežja.
WSDL - Web Services Description Language je jezik za opis spletne storitve. V verziji WSDL 1.1 je akronim pomenil Web Services Definition Language.
WS-* Web Services ali spletne storitve so ohlapno povezane programske komponente, ki komunicirajo prek Interneta in uporabljajo standardne tehnologije.
URI - Uniform Resource Identifier je niz znakov, ki identificirajo oziroma poimenujejo vir na Internetu.
XSD - XML Schema Definition je jezik za opis sheme XML dokumentov.
WS-HumanTask - je specifikacija, ki omogoča integracijo ljudi v storitveno usmerjene aplikacije.
BPEL4People - je specifikacija, ki predstavlja razširitev BPEL-a za podporo poslovnih procesov, ki vključujejo ljudi in njihovo posredovanje.

SOA - Service Oriented Architecture oziroma storitveno usmerjena arhitektura je arhitektura, ki razdeli funkcije v samostojne enote, ki jih imenujemo storitve.
BPM - BPEL Process Manager je ena od komponent Oraclovega SOA Suite-a, ki ponuja infrastrukturo za ustvarjanje, objavo in upravljanje z BPEL poslovnimi procesi.

Povzetek

V poslovnem okolju se pojavljajo vedno večje zahteve po zanesljivosti, učinkovitosti in fleksibilnosti poslovnih sistemov. Zaradi tega se vedno bolj razvija področje avtomatizacije poslovnih procesov. To pomeni, da je treba razviti aplikacije, ki nudijo podporo za vsako aktivnost v poslovnem procesu. Ker so poslovni procesi unikatni in se skozi čas spreminjajo, se je za reševanje tega problema uveljavila storitveno usmerjena arhitektura informacijske tehnologije.

Storitveno usmerjena arhitektura razdeli funkcije v samostojne enote, ki jih imenujemo storitve. Storitve so dostopne preko omrežja in tako jih lahko kombiniramo in ponovno uporabimo pri izdelavi aplikacij za podporo poslovnega procesa. Poslovni proces je implementiran z jezikom za izvajanje poslovnih procesov. Med temi jeziki je danes najbolj uveljavljen WS-BPEL.

Vsaka storitev implementira eno aktivnost v poslovnem procesu. Aktivnost je lahko avtomatizirana ali pa je pogojena z ročnim izvajanjem oziroma posredovanjem uporabnika. Slednje aktivnosti imenujemo uporabniška opravila. Opravila v poslovnem procesu je lahko elementarno ali sestavljeno. Sestavljeno opravilo je opravilo, ki je sestavljeno iz enega ali več podopravil. Podopravilo pa je spet lahko sestavljeno ali elementarno opravilo.

V diplomskem delu so obravnavani vzorci sestavljenih uporabniških opravil. Namen diplomskega dela je identifikacija vzorcev, ki so smiselni za implementacijo in tudi sama implementacija. Identificiranih je osem vzorcev. Prvih sedem je osnovnih in zelo splošnih, osmi vzorec pa je poljubna zaporedna kombinacija prvih sedmih. Vsi vzorci imajo visoko stopnjo ponovne uporabljivosti, pomembno pa je tudi, da posamezni klici storitve za izvajanje elementarnih opravil ne zahtevajo ohranjanja stanja.

Za implementacijo vzorcev je uporabljen jezik BPEL. Pri implementaciji je bil cilj tudi izdelava take storitve, kjer je enostavno dodajanje novih vzorcev oziroma spreminjanje množice vzorcev. Načela storitveno usmerjene arhitekture nam omogočajo implementacijo storitve v različnih okoljih in orodjih, v diplomskem delu pa je uporabljeno orodje Oracle, ki je zelo razširjeno.

Ključne besede:

storitveno usmerjena arhitektura, storitev, poslovni proces, uporabniško opravilo, vzorec sestavljenih uporabniških opravil, WS-BPEL

Abstract

Reliability, efficiency and flexibility demands of business systems in business environment are increasing. Therefore business process automation is rapidly developing, meaning we have to develop applications to support every activity in business process. Since business processes are unique and are changing over time, service oriented architecture is used as a solution to this problem.

Service oriented architecture divides functions in independent units called services. Services can be accessed over network, combined and reused when developing applications for business process support. Business process can be implemented with business process execution language. The most recognized among these languages is WS-BPEL.

Every service implements one of the activities in business process. Activity can be automated or manual. Manual activity is called user task. Business process can contain simple or complex tasks. A complex task is a task containing multiple subtasks. Every subtask can be either simple or complex task.

In my thesis I was dealing with complex user task patterns. One of the goals of this thesis is the identification of patterns which are suitable for implementation. Eight patterns have been identified. Seven of them are very basic and general, the eighth is an arbitrary sequence of any number of the first patterns. All of the patterns are highly reusable.

The other goal of this thesis is the implementation of identified and described patterns. BPEL is used for implementation. The aim of the implementation was also a service where patterns can be easily added or changed. Service oriented architecture enables the service to be implemented in different environments and software tools. Due to their popularity Oracle tools are used in thesis.

Key words:

service oriented architecture, service, business process, user task, complex user task pattern, WS-BPEL

1 Uvod

Integracija storitev, programov in podatkov v poslovnem okolju postaja čedalje pomembnejša, saj podjetja želijo vzpostaviti zanesljivo in učinkovito informacijsko podporo svojemu poslovanju. Integracijo aplikacij in storitev so v preteklosti ovirali predvsem visoki stroški, saj ni bilo na voljo ustrezne arhitekture za avtomatizacijo poslovnih procesov. Kot odgovor na te izzive se nam danes ponuja storitveno usmerjena arhitektura informacijske tehnologije.

V drugem poglavju diplomskega dela je predstavljena storitveno usmerjena arhitektura in tehnologije, jeziki ter specifikacije, ki se v storitveno usmerjeni arhitekturi najpogosteje uporabljajo. Predstavljene so spletne storitve, ki so ključen del storitveno usmerjene arhitekture, ter WS-* standardi, ki jim morajo spletne storitve ustrezati. Opisani so tudi dinamični klici spletnih storitev, kar je eden od ključnih izzivov pri implementaciji. Podan je tudi opis poslovnega procesa ter uporabniških opravil. Za integracijo uporabniških opravil v poslovni proces, ki je implementiran z jezikom BPEL, sta v uporabi specifikaciji WS-HumanTask in BPEL4People.

Namen pričujočega dela je identifikacija in implementacija vzorcev sestavljenih uporabniških opravil, zato so v četrtem poglavju predstavljeni vzorci. Identificiranih je sedem osnovnih vzorcev in vzorec, ki je poljubna zaporedna kombinacija teh sedmih vzorcev. Za vsak vzorec je podan tudi primer in grafičen prikaz vzorca. Vzorci morajo zadoščati pogojem, ki so našteti in opisani v tretjem poglavju.

V petem poglavju je predstavljena arhitektura implementacije, predstavljeni so ključni izzivi pri implementaciji ter opisi posameznih delov splošne implementacije vzorcev sestavljenih uporabniških opravil. Nato je predstavljena še konkretna implementacija storitve in vzorcev v okolju Oracle. Vzorci so implementirani kot spletne storitve in imajo enoten način klicanja, zato je opisana tudi XML shema, ki je uporabljena za klic generične storitve.

2 Predstavitev področja

2.1 Storitveno usmerjena arhitektura

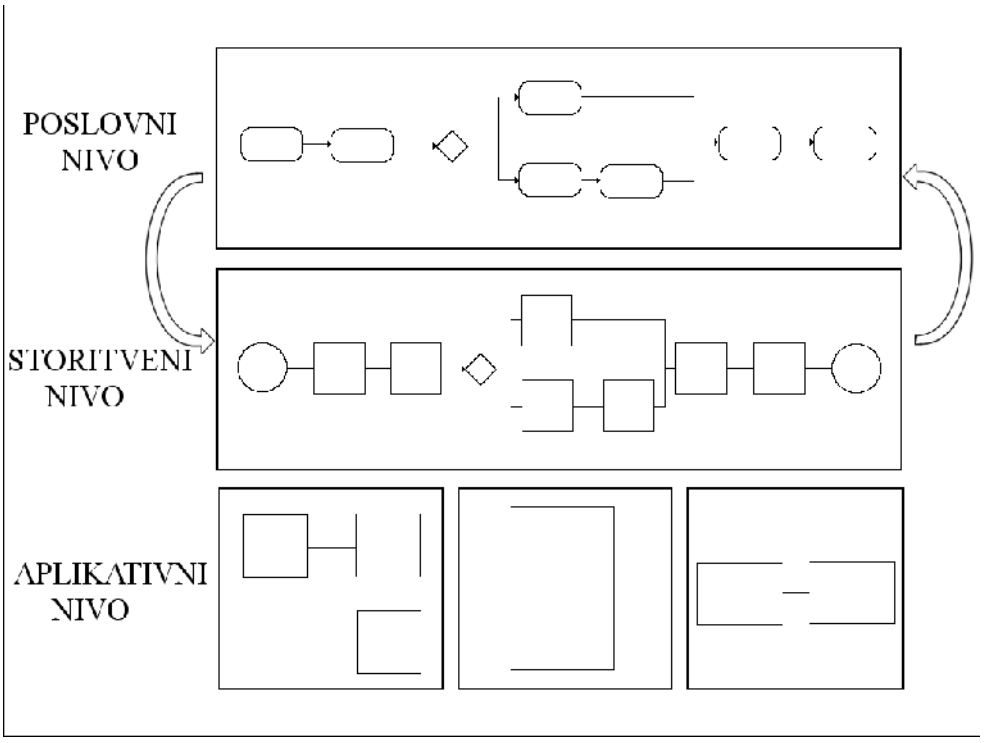
Glavni cilj informacijske tehnologije v poslovni informatiki je zagotavljanje podpore za poslovne operacije. Informacijska tehnologija je že uspešno avtomatizirala precej funkcij, kot so plačilne liste, glavna knjiga ter podobne. Nato se je nabor podprtih funkcij širil na ERP, CRM ter SCM. V naslednjem koraku pa želimo avtomatizirati poslovne procese. To pomeni, da je treba razviti aplikacije, ki nudijo podporo za vsako aktivnost v poslovnem procesu. Vendar pa so poslovni procesi v vsakem poslovnem sistemu unikatni, treba pa je upoštevati tudi dejstvo, da se s časom spreminjajo. Če se procesi skozi čas ne bi spreminjali, bi jih lahko podprli z aplikacijo, ki bi uporabljala močno povezane komponente, ki niso tako fleksibilne. Ker pa se procesi s časom spreminjajo, potrebujemo rešitev za fleksibilno integracijo, ki pa je v preteklosti ni bilo na voljo. Podjetja iz teh razlogov tudi ne morejo kupiti informacijske podpore za poslovne procese, ampak jo morajo razviti po meri za svoje potrebe [17].

Integracija storitev, programov in podatkov v poslovnem okolju postaja čedalje pomembnejša, saj podjetja želijo vzpostaviti zanesljivo in učinkovito informacijsko podporo svojemu poslovanju. Integracijo aplikacij in storitev so v preteklosti ovirali predvsem visoki stroški, saj ni bilo na voljo ustrezne arhitekture. Kot odgovor na te izzive se nam danes ponuja storitveno usmerjena arhitektura informacijske tehnologije [33].

Storitveno usmerjena arhitektura nam ponuja rešitve problemov, ki se pojavljajo v obstoječi arhitekturi. Storitveno usmerjena arhitektura omogoča razvoj bolj fleksibilnih aplikacij ter aplikacij, ki so bolj prilagodljive kot aplikacije razvite v tradicionalni arhitekturi. Storitveno usmerjena arhitektura nam omogoča boljšo povezanost med poslovnimi procesi in aplikacijami, saj zmanjša semantični prepad med modeli poslovnih procesov ter aplikacijami, ki podpirajo poslovne procese [17].

Storitveno usmerjena arhitektura razdeli funkcije v samostojne enote, ki jih imenujemo storitve. Storitve so dostopne preko omrežja in tako jih lahko kombiniramo in ponovno uporabimo pri izdelavi aplikacij. V svoji izvorni kodi nimajo implementiranih nobenih klicev med seboj. Namesto tega komunicirajo s pomočjo protokolov, ki določajo obliko sporočil ter način pošiljanja [10, 24].

Slika 2.1.1 prikazuje ključni koncept storitveno usmerjene arhitekture. Storitveno usmerjena arhitektura je sestavljena iz treh nivojev. Na vrhu je poslovni nivo, na dnu pa je aplikativni nivo. Med poslovnim in aplikativnim nivojem je storitveni nivo, ki je sestavljen iz storitev. Storitve pa so logične enote, ki ustrezajo načelom storitvene usmerjenosti. Ta načela so: ohlapna povezanost, storitvena pogodba, neodvisnost, abstrakcija, ponovna uporaba, storitve so brez stanja in so narejene tako, da jih lahko odkrivamo in uporabljamo s temu namenjenimi mehanizmi [10].



Slika 2.1.1: Koncept storitveno usmerjene arhitekture

Za določeno aplikacijo potrebujemo točno določene storitve, ki jih je treba klicati v pravem zaporedju. Ker storitve takega visokonivojskega nadzora nimajo, jih je treba orkestrirati. Orkestracija storitev je proces pri katerem storitve povežemo v nehierarhično obliko. Pri tem moramo poznati razpoložljive storitve ter njihove vmesnike. Z orkestracijo lahko povežemo med seboj tudi različne procese. Orkestracijo lahko naredimo na različne načine z različnimi orodji, v diplomskem delu pa je za orkestracijo storitev uporabljen jezik BPEL.

Cilj storitveno usmerjene arhitekture je izdelava ad hoc aplikacij, ki jih sestavimo skoraj izključno iz obstoječih storitev. Vsaka storitev naj bi zagotavljala čim več funkcionalnosti, saj je tako potrebnih manj storitev in zato tudi manj povezav med njimi. Vendar pa storitev ne sme imeti preveč funkcionalnosti, saj je s tem lahko otežena njihova ponovna uporabljivost. Zato moramo najti optimalno količino funkcionalnosti, ki naj jo storitev zagotavlja.

Pogoj za storitveno usmerjeno arhitekturo je ohlapna povezanost storitev. Ohlapna povezanost pomeni, da pri klicanju storitve nimamo konkretne implementacije, ampak samo vmesnik, ki opisuje storitev. Za tem vmesnikom pa se lahko skriva ena ali več konkretnih implementacij. Storitve so neodvisne ena od druge. Če določena storitev izpade, druge še vedno delujejo. Ohlapna povezanost je pomembna tudi pri spreminjanju storitev, saj minimalna odvisnost zagotavlja, da spremembe ene storitve ne vplivajo na druge storitve. Tak pristop izboljša robustnost, naredi sisteme bolj prožne in fleksibilne ob spremembah ter olajšuje ponovno uporabo storitev [17]. Tabela 2.1.1 podaja razlike med močno in ohlapno povezanostjo.

	MOČNA POVEZANOST	OHLAPNA POVEZANOST
FIZIČNA POVEZANOST	Zahteva neposredno fizično povezavo	Fizična povezava je lahko posredna
KOMUNIKACIJA	Sinhrona	Asinhrona
NAČIN INTERAKCIJE	Kompleksna objektna drevesa	Sporočila
KONTROLA PROCESNE LOGIKE	Centralna kontrola procesne logike	Porazdeljena kontrola procesne logike
ODKRIVANJE IN POVEZOVANJE STORITEV	Statično povezane storitve	Dinamično povezane storitve
PLATFORMNA ODVISNOST	Močna odvisnost od operacijskega sistema in programskega jezika	Neodvisnost od operacijskega sistema in programskega jezika

Tabela 2.1.1: Primerjava močne in ohlapne povezanosti sistemov [20]

S pomočjo storitveno usmerjene arhitekture lahko dosežemo večjo storilnost in prilagodljivost z gradnjo novih poslovnih aplikacij in procesov iz storitev, ki omogočajo ponovno uporabo. Sestavljanje aplikacij iz vnaprej pripravljenih storitev zahteva veliko manj časa in denarja, kot če bi jih razvili na novo. Značilnost sestavljenih aplikacij je tudi, da so posamezni deli med seboj neodvisni, lahko tečejo celo na različnih sistemih, povezave med temi deli pa so standardizirane.

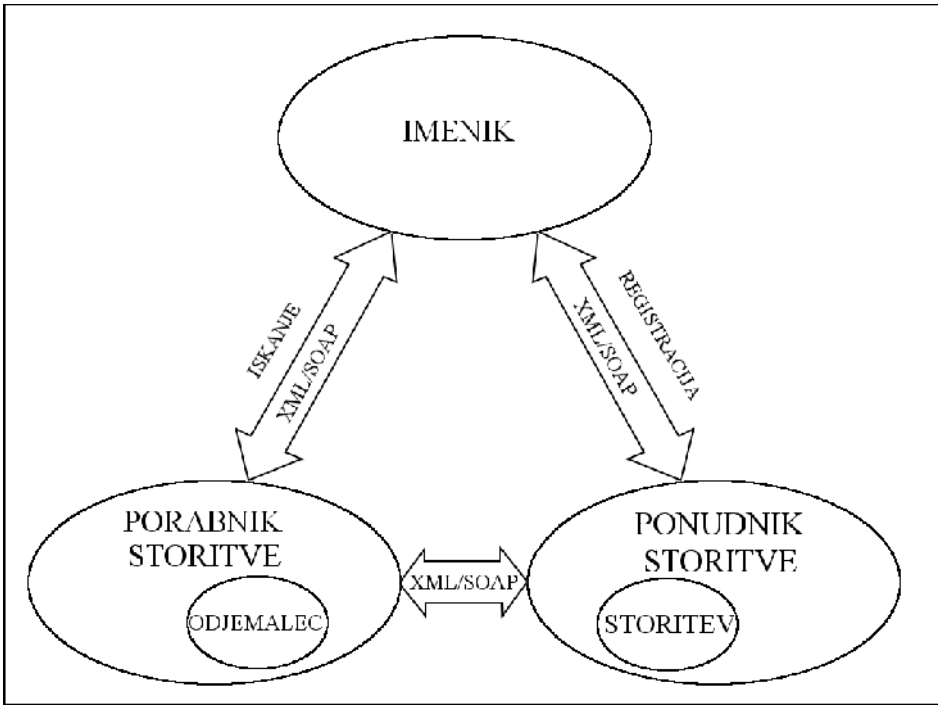
Storitveno usmerjene arhitekture postajajo vedno bolj pomemben pristop k povezovanju raznolikih računalniških sistemov v široko porazdeljenih omrežjih.

Storitve v storitveno usmerjeni arhitekturi lahko implementiramo v obliki spletnih storitev (Web Services).

Pri spletnih storitvah uporabljamo mehanizem UDDI (Universal Description, Discovery and Integration) za objavo in iskanje spletnih storitev v imeniku. UDDI je platformno neodvisen imenik, ki bazira na jeziku XML. Podjetjem po vsem svetu omogoča, da objavijo svoje storitve na Internetu. S pomočjo mehanizma UDDI podjetja svoje storitve objavijo, iščejo po imeniku spletnih storitev, ki so na voljo, ter definirajo na kakšen način lahko aplikacije komunicirajo s temi storitvami.

UDDI imenik sestavljajo trije deli: bele strani, rumene strani ter zelene strani. Bele strani vsebujejo ime, naslov in druge podatke podjetja, ki ponuja storitve. V rumenih straneh so storitve podjetij kategorizirane po določenih standardih in klasifikacijah. Ker eno podjetje lahko ponuja več storitev je z eno belo stranjo, ki predstavlja eno podjetje, lahko povezanih več rumenih strani, ki predstavljajo posamezne storitve. Storite so lahko tudi v različnih kategorijah. Zelene strani vsebujejo tehnične podatke o dostopnih storitvah posameznih podjetij ter opis povezav do storitev. Na teh straneh

je opisano kako dostopamo do storitve. Naveden je naslov storitve in parametri, ki so potrebni za klic storitve. Ker ima lahko spletna storitev več možnih povezav je lahko nanjo vezanih več zelenih strani [25].



Slika 2.1.2: Storitveno usmerjena arhitektura (SOA)

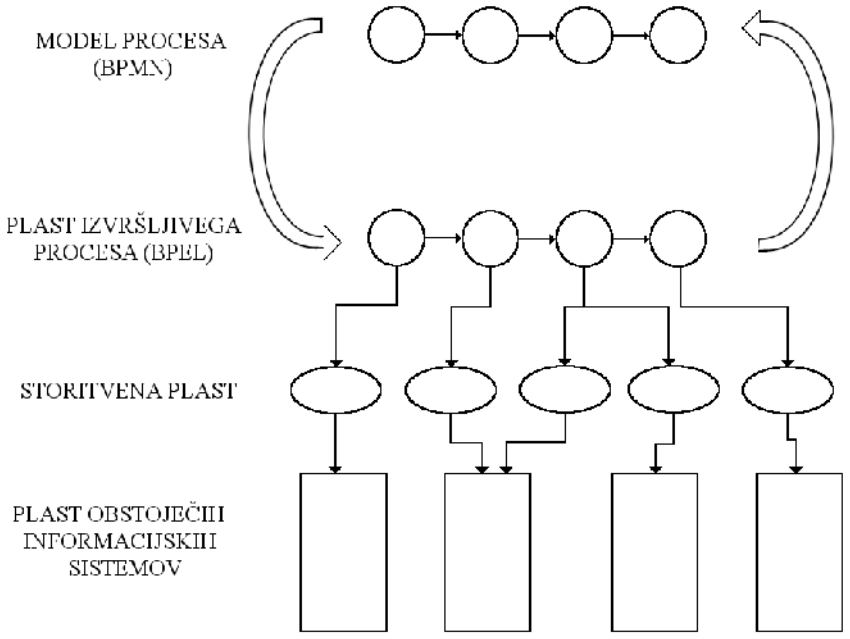
2.2 BPMN (Business Process Modeling Notation)

Business Process Modeling Notation je grafična notacija za modeliranje poslovnih procesov. S pomočjo te notacije lahko narišemo diagrame poslovnih procesov. Ti diagrami predstavljajo aktivnosti in opravila ter razmerja med njimi, uporabljajo pa koncepte diagramov poteka oziroma diagramov aktivnosti iz jezika Unified Modeling Language (UML) [17, 23]. Glavni namen BPMN je zagotavljanje podpore upravljanju poslovnih procesov tako za tehnične kot za poslovne uporabnike, saj je notacija intuitivna in enostavna, hkrati pa omogoča opis kompleksnih procesov in specifikacijo podrobnosti [17].

BPMN specifikacija določa grafične simbole za aktivnosti, odločitve, dogodke, vloge ter dokumente. Poleg tega določa pravila za povezovanje teh elementov. Zaporedje izvajanja aktivnosti določimo s povezavo zaporedja, ki je puščica s polno črto. Tok dokumentov pa določimo s sporočilno povezavo, ki je puščica s črtkano črto.

Specifikacija BPMN poleg grafičnih simbolov definira tudi preslikavo diagrama procesa v izvršljiv jezik BPEL. Poleg osnove za informacijski sistem pa nam opisi procesov v notaciji BPMN lahko služijo tudi kot delovna navodila za zaposlene ter del poslovnika kakovosti (ISO 9001:2000) [23].

Modele poslovnih procesov uporabljamo za lažje razumevanje procesov. Z njihovo pomočjo procese lažje spreminjamo, izboljšujemo in optimiziramo. Poleg notacije BPMN lahko za modeliranje poslovnih procesov uporabljamo tudi številne druge vizualne jezike. Najpogosteje so uporabljeni EPC (Event-Driven Process Chain), eEPC (Extended Event-Driven Process Chain) ter diagrami aktivnosti v UML [17].



Slika 2.2.1: Povezanost BPMN in BPEL ter umeščenost v storitveno usmerjeno arhitekturo

2.3 WS-BPEL (Web Services Business Process Execution Language)

WS-BPEL oziroma Web Services Business Process Execution Language ali krajše BPEL je standard organizacije OASIS, ki specificira jezik za interakcije med spletnimi storitvami [22]. BPEL je jezik, ki skrbi za orkestracijo spletnih storitev [16]. Omogoča nam formalen opis poslovnih procesov in razširja model spletnih storitev s podporo poslovnim transakcijam. Za razliko od BPMN notacije, s pomočjo katere modeliramo procese, je BPEL jezik za izvajanje procesov.

BPEL proces je lahko sinhronski ali asinhronski. Proces, ki v svojem toku kliče asinhronski proces, lahko s svojim izvajanjem nadaljuje vzporedno z izvajanjem asinhronskega procesa. Ko se asinhronski proces zaključi, pošlje odgovor v kličoči proces. Vmes lahko oba procesa izvajata operacije.

Proces, ki pa kliče sinhronski proces, mora svoje delovanje prekiniti in čakati na odgovor sinhronskega procesa. Šele ko se sinhronski proces izvede do konca in pošlje

svoj odgovor, prvi proces lahko nadaljuje s svojim izvajanjem. Pri klicanju sinhronskega procesa je torej onemogočeno vzporedno izvajanje obeh procesov.

BPEL bazira na XML-u. V oznaki <process> je navedeno ime procesa in našteti so imenski prostori, ki jih uporabljamo v procesu. To je hkrati tudi korenski element XML dokumenta, ki opisuje proces. Naslednja oznaka je <partnerLinks>, v kateri so našteti partnerji oziroma njihove storitve s katerimi proces komunicira. Interakcija s spletno storitvijo se smatra kot komunikacija s poslovnim partnerjem, ki je opisana s pomočjo <partnerLink> oznake. V vsaki oznaki <partnerLink> sta navedeni vlogi procesa in partnerja. Eden od njiju je ponudnik, drugi pa uporabnik storitve.

Zadnja v deklarativnem delu je oznaka <variables>, kjer so naštete globalne spremenljivke procesa. Vrednosti spremenljivk določajo stanje procesa med izvajanjem. Če želimo v procesu uporabljati tudi lokalne spremenljivke ter lokalne partnerske povezave, lahko to označimo z oznako <scope>. Spremenljivke in povezave potem veljajo samo v obsegu oznake <scope>. Za manipulacijo s spremenljivkami v jeziku BPEL uporabljamo jezik XPath.

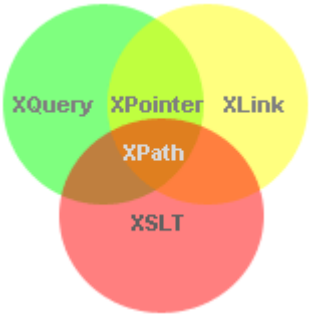
Dejanski potek procesa je opisan v oznaki <sequence>, kjer v oznaki <receive> proces prejme vhod in na koncu vrne odgovor v oznaki <reply>, če gre za sinhronski proces oziroma v oznaki <invoke>, če gre za asinhronski proces. Znotraj oznake <sequence> lahko dodajamo oznake, ki ustrezajo stavkom v drugih programskih jezikih, npr.: <if>, <while>, <repeatUntil>, za določanje vrednosti spremenljivkam pa uporabljamo oznako <assign>.

Pomembne oznake so še <invoke>, <receive> in <reply>, ki označujejo aktivnosti za izmenjavo sporočil s poslovnimi partnerji. Oznaki <invoke> ter <receive> v paru uporabljamo za klicanje asinhronskega procesa. V oznaki <invoke> navedemo katero storitev želimo klicati, operacijo, ki naj se izvede ter vhodne podatke za to storitev, v oznaki <receive> pa ime storitve, operacijo ter spremenljivko, v katero se bo zapisal odgovor. Za klicanje sinhronskega procesa uporabimo samo oznako <invoke> [5].

2.4 XML Path jezik (XPath)

XPath je jezik za naslavljanje delov XML dokumenta. Zasnovan je dovolj splošno, tako da ga lahko uporabljamo v XSLT (eXtensible Stylesheet Language Transformations) ter v jeziku XPointer (XML Pointer Language). Slika 2.4.1 prikazuje umeščenost jezika XPath v XML tehnologije. XPath uporablja kompaktno sintakso, ki ne bazira na XML-u. Za navigacijo po hierarhični strukturi XML dokumenta uporablja notacijo v obliki poti. XML dokument je v XPath jeziku predstavljen kot drevo z vozlišči. XPath podpira tudi XML imenski prostor, zato je ime vozlišča sestavljeno iz imenskega prostora ter lokalnega imena vozlišča. Primer imena vozlišča je ’/client:BPELProcessInputVariable/client:result’. V tem imenu je ’client’ imenski prostor, ’result’ pa je vozlišče poddrevesa ’BPELProcessInputVariable’.

Poleg naslavljanja vozlišč se XPath uporablja tudi za iskanje vozlišč, ki ustrezajo določenim pogojem. Ta funkcionalnost se uporablja v transformacijah jezika eXtensible Stylesheet Language (XSL). S pomočjo XSL transformacij (XSLT) lahko transformiramo XML dokument v drug XML dokument ali celo v (X)HTML dokument. Z XSLT lahko dodajamo in odstranjujemo elemente ter atribute v XML datoteki, lahko sortiramo elemente in jih filtriramo [6].



Slika 2.4.1: Umeščenost jezika XPath v XML tehnologije

2.5 Spletne storitve

Konzorcij W3C definira spletno storitev kot programski sistem, ki podpira interoperabilno interakcijo med računalniki preko omrežja [26]. Gre torej za različne sisteme, ki sodelujejo med seboj, si izmenjujejo informacije ter znajo izmenjane informacije tudi uporabiti. Odjemalčeva aplikacija komunicira s spletnimi storitvami s pomočjo XML dokumentov. Gartner je spletne storitve definiral kot ohlapno povezane programske komponente, ki delujejo prek Interneta in uporabljajo standardne tehnologije. Storitev je lahko razvita in lahko teče na poljubni platformi, za povezavo z drugimi storitvami pa uporablja vmesnik, iz katerega lahko tako človek kot računalnik razbereta na kakšen način je potrebno storitev klicati in kakšen odgovor lahko pričakujemo od nje. V vmesniku je naveden tudi naslov storitve.

Spletne storitve delimo v več skupin. Najbolj znane so tako imenovane Big Web Services, v zadnjem času pa se nekoliko pogosteje uporabljajo tudi Representational State Transfer (RESTful) Web Services. Storitve iz prve skupine za komunikacijo uporabljajo XML sporočila, ki ustrezajo SOAP standardom. Te storitve imajo svoj opis v obliki WSDL datoteke, so bolj fleksibilne in bolj ustrezajo zahtevam po kakovosti storitev. Te storitve se tudi najpogosteje uporabljajo v storitveno usmerjeni arhitekturi.

Storitve iz druge skupine, RESTful Web Services, so nekoliko bolj popularne postale v zadnjem času. Te storitve ne potrebujejo XML sporočil za komunikacijo ter nimajo nujno svojih opisov v obliki WSDL datotek. So bolj primerne za osnovno ad hoc integracijo, saj so precej enostavne za implementacijo, vendar pa niso tako fleksibilne kot Big Web Services in kot take niso primerne za zahtevnejše poslovne aplikacije [18].

V diplomskem delu so uporabljene Big Web Services, ki z ostalimi sistemi komunicirajo s pomočjo SOAP sporočil preko HTTP protokola ter morajo zadoščati WS-* standardom oziroma specifikacijam. Glavni standardi so WS-Coordination, WS-Security, WS-Addressing, WS-Reliability in WS-Transaction [26].

SOAP (Simple Object Access Protocol) je specifikacija za izmenjavo strukturiranih sporočil med spletnimi storitvami preko računalniškega omrežja. Sporočila so v XML formatu in za prenos ponavadi uporabljajo HTTP protokol, ki spada v aplikacijsko plast. Za prenos SOAP sporočil lahko uporabljamo tudi kakšen drug protokol iz aplikacijske plasti, vendar je HTTP najbolj pogost.

WS-Coordination skrbi za koordinacijo akcij porazdeljenih aplikacij ter konsistenco rezultatov porazdeljenih transakcij. Vendar pa ta specifikacija ni dovolj za koordinacijo spletnih storitev, potrebne so še druge specifikacije, kot na primer WS-Atomic Transaction [11, 29].

WS-Security skrbi za varnost spletnih storitev. Varnost zagotavlja z dodajanjem digitalnih podpisov in kriptiranja glave SOAP sporočil. Na ta način dobimo tako imenovano end-to-end varnost, saj deluje na aplikacijskem in ne na transportnem nivoju [15, 31].

WS-Addressing je standarden način za vstavljanje naslova v glavo SOAP sporočila. Omogoča uporabo asinhrone interakcije, saj je v glavi tudi naslov, kamor je potrebno poslati odgovor. Na ta način razdvojimo interakcijo na zahtevo in odgovor in ker imamo dve SOAP seji v vmesnem času ne obremenjujemo HTTP protokola. To nam omogoča, da interakcija traja poljubno dolgo časa [14, 7, 28].

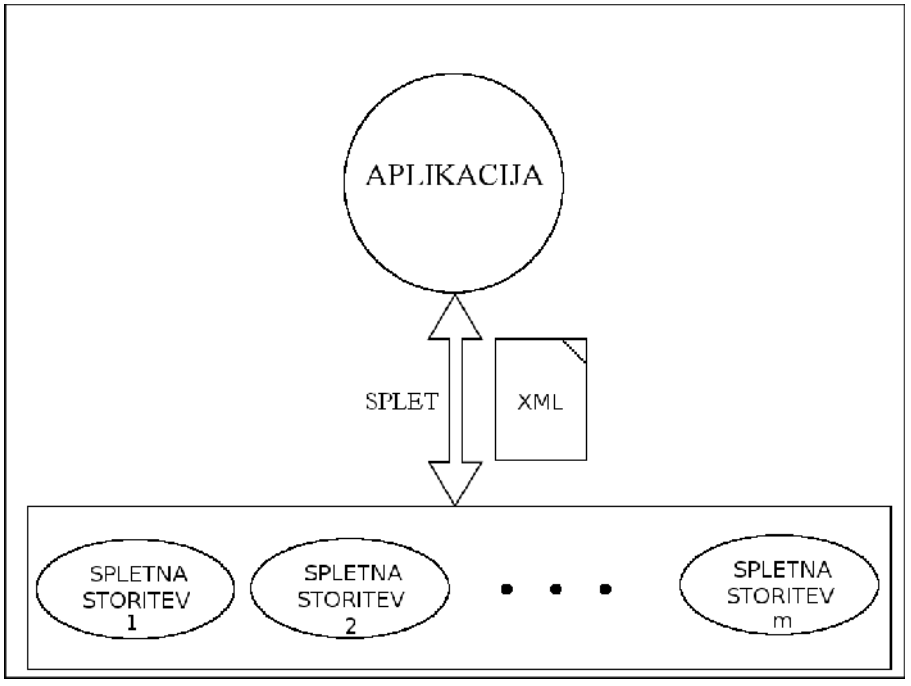
WS-Reliability zagotavlja zadostno raven zanesljivosti, ki je potrebna za določene spletne storitve. SOAP preko HTTP-ja namreč sam po sebi ne zagotavlja takšne zanesljivosti [8, 30].

WS-Transaction definira dva tipa koordinacije, ki jih uporabljamo skupaj z WS-Coordination specifikacijo. Ta dva tipa sta Atomic Transaction, ki se uporablja za individualne transakcije ter Business Activity za dalj časa trajajoče transakcije [19, 12, 32].

Storitve so vmesnik do poslovnih funkcij, ki jih je mogoče združiti in iz njih oblikovati celovito rešitev ali poslovni proces. S pomočjo storitveno usmerjene arhitekture postanejo te storitve neodvisne od obstoječega okolja, zato jih lahko ljudje, procesi in druge aplikacije preprosto ponovno uporabijo. Storitve so danes večinoma implementirane v obliki spletnih storitev. Spletne storitve lahko razvijemo na novo ali pa jih ovijemo okoli starih 'legacy' sistemov in tako te stare sisteme preko omrežja ponudimo drugim uporabnikom.

Vsaka enota storitveno usmerjene arhitekture lahko nastopa v več vlogah: v vlogi ponudnika storitve, v vlogi uporabnika storitve ali v obeh vlogah hkrati.

Ponudnik storitve ustvari spletno storitev ter njen vmesnik objavi v register spletnih storitev. Vsak ponudnik spletnih storitev se mora odločiti, katere storitve bo dal na voljo drugim uporabnikom, kako bo poskrbel za varnost in hkrati enostavno dostopnost spletnih storitev ter kako bo uporabnikom zaračunaval za njihovo uporabo. UDDI standard definira način objave ter iskanja spletnih storitev.
Uporabnik spletne storitve poišče storitev v registru in se nato poveže s ponudnikom storitve ter kliče ustrezno storitev.



Slika 2.5.1: Spletne storitve v okviru storitveno usmerjene arhitekture

2.5.1 Dinamični klic spletnih storitev

Storitveno usmerjena arhitektura skupaj s spletnimi storitvami torej omogoča poslovnim procesom enostavno povezovanje z drugimi poslovnimi procesi ter aplikacijami. Proces v BPEL-u komunicira z ostalimi storitvami s pomočjo partnerske povezave, kjer je definiran vmesnik za komunikacijo ter lokacija storitve, ki jo proces uporablja. V večini procesov so partnerske povezave statične in se nanašajo na storitev, ki jo je razvijalec procesa predvidel. Tak način povezave nas lahko omejuje, saj so v večjih sistemih procesi pogosto zelo kompleksni in komunicirajo z velikim številom partnerjev. Včasih ob razvoju BPEL procesa naslov storitve še ni znan, ali pa želimo na voljo več storitev na različnih naslovih. Lahko se nam namreč zgodi, da je določena storitev nedosegljiva, deluje pa enaka storitev na drugem naslovu. Vso to logiko za odločanje, katero storitev bomo uporabili, lahko integriramo v BPEL proces, ki pa na ta način postane večji in bolj kompleksen ter zapleten.

V te namene nam BPEL omogoča koncept dinamičnega povezovanja partnerskih povezav. Podatke o partnerskih povezavah lahko podamo v konfiguracijo procesa, ali pa jih podamo kot vhod procesa. Ta pristop nam omogoča, da se sistem prilagodi na pogoje, ki v času razvoja še niso znani.

V partnerski povezavi so navedene operacije ter tipi sporočil, ki ustvarijo vmesnik s storitvijo s pomočjo lastnosti portType v WSDL dokumentu. V statičnem procesu so ti podatki podani ob razvoju procesa. Ker pa v nekaterih primerih želimo te podatke podati med izvajanjem procesa, ali pa jih želimo spremeniti, jih lahko podamo dinamično.

Za primer vzemimo proces za odobritev kredita, ki se odloči za enega od kreditodajalcev na podlagi vhodnih podatkov, na primer zneska kredita. V tem primeru se je potrebno dinamično kreirati partnersko povezavo z izbranim partnerjem, ki ponuja storitev najema kredita pod temi pogoji.

Partnersko povezavo lahko dinamično kreiramo s pomočjo mehanizma, ki ga standard WS-Addressing imenuje endpoint reference. Na ta način lahko izberemo eno izmed storitev, ki so definirane v WSDL dokumentu, ali pa definiramo novo storitev med izvajanjem procesa. Poslovni proces je statično odvisen od vmesnika, ki je definiran v portType lastnosti, medtem ko lahko s pomočjo endpoint reference dinamično spremenimo naslov storitve. To pomeni, da mora biti vmesnik definiran že vnaprej in med izvajanjem procesa ne moremo dinamično določiti vhodnih podatkov in operacij. Določimo lahko samo ime in točen naslov storitve, ki jo želimo klicati in odgovarja definiranem vmesniku [9].

V endpoint reference moramo podati URI naslov storitve, ki je lahko mrežni ali logični naslov. Endpoint reference lahko vsebuje tudi vrednosti različnih lastnosti in parametrov, ki so potrebni pri klicu storitve, ime storitve ter politike, ki opisujejo obnašanje, zahteve in zmogljivosti storitve [10].

2.6 Metapodatki spletnih storitev

Metapodatki spletnih storitev podajajo informacije o storitvi oziroma opisujejo storitev. So sestavni del pogodbe o uporabi storitve (service contract), ki jo mora uporabnik storitve poznati in sprejeti za uspešno komunikacijo.

2.6.1 WSDL (Web Services Description Language)

Kot je omenjeno pri opisu storitveno usmerjene arhitekture, je pogoj zanjo ohlapna povezanost storitev, kar pomeni, da potrebujemo vmesnik, ki opisuje storitev. To je tudi eden ključnih konceptov storitveno usmerjene arhitekture. Vsaka storitev, ki jo želimo klicati, torej potrebuje dokument, ki jo opisuje. Najpogosteje se v ta namen uporabljajo WSDL dokumenti.

WSDL uporablja XML format za formalni opis vmesnika storitve. V WSDL dokumentu so abstraktni opisi sporočil ter operacij ter konkretni opisi povezav in storitev. Abstraktne opise lahko ponovno uporabimo, saj niso odvisni od platforme, na kateri teče spletna storitev. Tako se tudi po morebitni spremembi implementacije spletne storitve ta del ne spremeni.

Danes sta v uporabi dve verziji jezika WSDL. Starejša verzija je WSDL 1.1, novejša pa je bila najprej označena kot WSDL 1.2, vendar so jo kasneje zaradi nekaterih korenitih sprememb označili kot WSDL 2.0. WSDL 2.0 nudi boljšo podporo RESTful spletnim storitvam in je enostavnejši za implementacijo, vendar pa ni podprt v programih za razvoj programske opreme. Ti programi v glavnem nudijo možnost razvoja WSDL 1.1 dokumentov [27]. Zato je tudi v diplomskem delu za implementacijo uporabljen WSDL 1.1.

V verziji WSDL 1.1 so trije glavni deli abstraktnega dela opisa: tipi <types>, sporočilo <message> ter tip vrat <portType>. Tipi v WSDL dokumentu opisujejo podatke. V ta namen so uporabljene XML sheme, ki so lahko definirane znotraj WSDL dokumenta ali pa so uvožene iz drugih dokumentov. Element <portType> definira spletno storitev s tem, da so v njem navedene operacije <operation>, ki jih spletna storitev lahko izvede. Znotraj vsake operacije so navedena sporočila, ki jih operacije sprejemajo kot vhod <input> ter oddajo kot odgovor <output>.

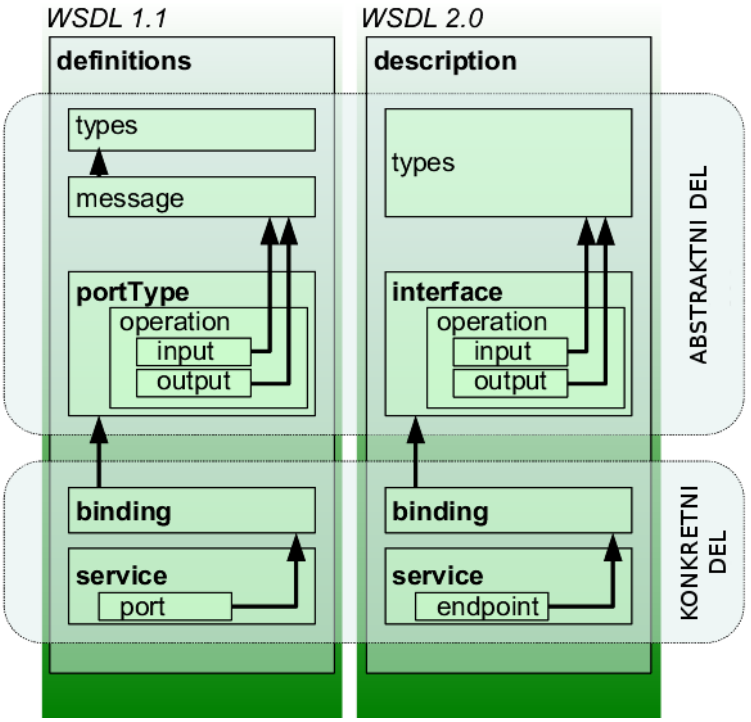
Sporočilo v WSDL dokumentu odgovarja določeni operaciji. Sporočilo vsebuje informacijo, ki je potrebna za izvedbo operacije. Za opis formata sporočil WSDL uporablja XML sheme. Vsako sporočilo je sestavljeno iz enega ali več logičnih delov [27].

Poleg abstraktnega dela je v dokumentu tudi konkreten del, ki vsebuje opis povezav. Tu je naveden fizični transportni protokol, ki je potreben za povezavo s spletno storitvijo. Ta del je sestavljen iz dveh večjih razdelkov: povezava <binding> in storitev <service>. Prvi razdelek opisuje fizično povezavo, ki je potrebna za komunikacijo s storitvijo. Najpogosteje se za povezavo uporablja SOAP protokol. Storitev je vozlišče, ki lahko vsebuje več vrat <port>, vrata pa predstavljajo fizični naslov preko katerega do storitve dostopamo z določenim protokolom. Fizični naslov je ponavadi podan z navadnim URL nizom [27].

V WSDL 2.0 v abstraktnem delu ni več elementa sporočilo <message>. V tej verziji za vhod in izhod operacij uporabljamo neposredno XML sheme. Element tip vrat <portType> je v verziji 2.0 preimenovan v vmesnik <interface>. Element vmesnik vsebuje enake definicije operacij kot element tip vrat. Slika 2.6.1.1 prikazuje strukturo WSDL 1.1 in WSDL 2.0 dokumentov.

Spletne storitve so lahko tudi neodvisne od WSDL dokumentov. Lahko se namreč zgodi, da se naslov spletne storitve pogosto spreminja in se želimo izogniti neprestanemu posodabljanju WSDL datotek. To lahko storimo tako, da proces med izvajanjem določi naslov endpoint reference. V WSDL datoteki še vedno

mora biti napisan naslov, ki ga proces uporabi, če v endpoint reference ni navedenega nobenega naslova.



Slika 2.6.1.1: Struktura WSDL 1.1 in WSDL 2.0

2.6.2 XSD (XML Schema Definition)

Jezik XSD je pomemben del XML tehnologije, saj se pogosto uporablja pri spletnih storitvah. Z njim opišemo shemo na podlagi katere potem lahko kreiramo XML dokumente. Vsak XML dokument je v bistvu instanca določene XSD sheme. Shema lahko obstaja kot samostojen dokument, lahko pa jo dodamo v WSDL dokument in tako neposredno pred opisom storitve definiramo format sporočil, ki jih storitev uporablja za komunikacijo. Sicer pa je potrebno uvoziti shemo iz XSD dokumenta.

Vsak podatek v shemi mora pripadati določenem podatkovnem tipu. Osnovni podatkovni tipi so že vnaprej definirani, lahko pa dodamo tudi svoje podatkovne tipe, ki so lahko razširjeni ali sestavljeni iz osnovnih tipov.

Vsaka shema vsebuje korenski element <schema>, kjer so definirani imenski prostori. Sledijo pa lahko še elementi <element>, <simpleType>, <complexType> ter <import> in <include>. V oznaki <element> navedemo ime ter tip podatkov, ki nastopajo v shemi. S pomočjo oznak <simpleType> in <complexType> lahko definiramo sestavljene podatkovne tipe s to razliko, da je <simpleType> enonivojski tip, pri <complexType> pa imamo lahko več nivojev: eden ali več elementov tega tipa je lahko sestavljen. Oznaki <import> in <include> uporabimo za uvoz shem, ki jih potem lahko razširimo z dodatnimi tipi in elementi. Če

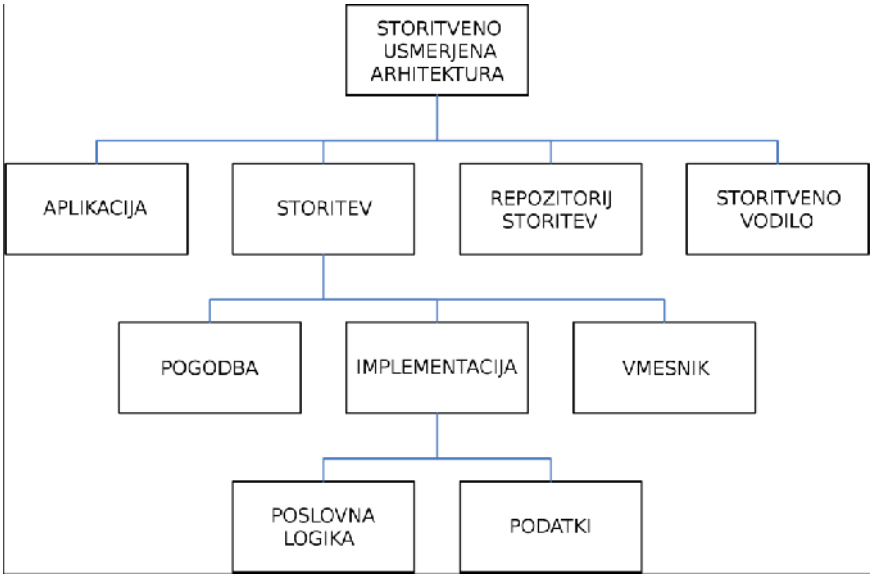
uporabimo <include>, je imenski prostor enak kot v uvoženi shemi, če pa uporabimo <import>, pa lahko uporabimo svoj imenski prostor [13].

Jezik XML Shema Definition je velik in kompleksen ter vsebuje še precej elementov in atributov, vendar ti niso tako pomembni in tudi v sami implementaciji diplomskega dela niso uporabljeni.

2.6.3 Politike

Vsako avtomatizirano poslovno opravilo mora upoštevati določena pravila ter omejitve, ki so lahko pojavijo zaradi zahtev poslovnih opravil, narave podatkov, ki jih izmenjujemo ali zaradi varnostnih meril organizacije. Vsaka storitev in izmenjano sporočilo ima drugačne značilnosti, ki jih morajo druge storitve poznati, da lahko komunicirajo s to storitvijo. Te značilnosti so na primer tehnične omejitve, kvaliteta storitve (QoS – Quality of Service) in podobne. Vse značilnosti storitve, ki so potrebne za komunikacijo z ostalimi storitvami, so opisane v politikah in se izmenjavajo skupaj z ostalimi metapodatki [10]. V spletnih storitvah politike implementiramo s pomočjo standarda WS-Policy, ki vsebuje še WS-PolicyAttachments in WS-PolicyAssertions specifikaciji.

Slika 2.6.1 prikazuje umeščenost storitev v storitveno usmerjeni arhitekturi.



Slika 2.6.1: Dekompozicijski diagram storitveno usmerjene arhitekture

2.7 Poslovni proces in uporabniška opravila

Poslovni proces je množica logično povezanih strukturiranih aktivnosti oziroma opravil, ki zagotavljajo storitev ali izdelavo izdelka za določeno stranko [21]. Lahko so avtomatizirana ali pa jih izvede določena oseba oziroma skupina oseb [17]. Poslovne procese delimo na ključne in podporne poslovne procese. Ključni poslovni procesi neposredno ustvarjajo poslovno vrednost, podporni poslovni procesi pa omogočajo izvajanje ključnih poslovnih procesov ali pa omogočajo bolj učinkovito

izvajanje ključnih poslovnih procesov [4]. Poslovne procese lahko delimo tudi na notranje in globalne. Notranji poslovni procesi potekajo v enem samem podjetju, globalni poslovni procesi pa povezujejo dve ali več podjetij. Obe vrsti poslovnih procesov sta pomembni in ju lahko modeliramo s pomočjo storitveno usmerjene arhitekture [17].

V interesu vsakega podjetja je, da so poslovni procesi učinkoviti ter da vsebujejo le tiste aktivnosti, ki so potrebne za sam potek procesa, saj je s tem potek hitrejši. Poslovne procese je torej potrebno optimizirati, saj je od njih odvisna učinkovitost podjetja. Optimizacija poslovnih procesov je za podjetja vedno bolj pomembna, saj zagotavlja konkurenčno prednost pred ostalimi podjetji. S pomočjo storitveno usmerjene arhitekture je optimizacija poslovnih procesov enostavnejša in hitrejša kot pri drugih klasičnih pristopih [17].

Poslovni proces sestavljajo opravila, ki so lahko avtomatizirana ali ročna. V diplomskem delu so obravnavana uporabniška opravila. Uporabniška opravila so tisti del opravil v poslovnih procesih, ki niso avtomatizirana. Pri teh opravilih je za nadaljevanje poslovnega procesa potrebno posredovanje uporabnika ali skupine uporabnikov.

Opravila v poslovnem procesu so lahko elementarna ali sestavljena. Sestavljeno opravilo je opravilo, ki je sestavljeno iz enega ali več podopravil. Podopravilo pa je spet lahko sestavljeno ali elementarno opravilo.

V zadnjem času se pospešeno razvijajo uporabniške interakcije v poslovnih procesih s ciljem standardizacije vključitve uporabniških opravil v BPEL procese. V ta namen sta bili razviti specifikaciji WS-HumanTask in BPEL4People.

2.7.1 Specifikacija WS-HumanTask

WS-HumanTask specifikacija omogoča integracijo ljudi v storitveno usmerjene aplikacije. Uporabniška opravila so storitve, ki jih opravljajo ljudje. Uporabniško opravilo ima dva vmesnika. Na eni strani je vmesnik, ki opisuje in ponuja storitev, ki jo ljudje opravljajo, na drugi strani pa je uporabniški vmesnik. Vsako uporabniško opravilo ima določene ljudi, ki lahko delajo na tem opravilu. S tem so določene vloge, ki jih uporabniki igrajo pri nekem opravilu.

Posebna vrsta uporabniških opravil so notifikacije. Notifikacije omogočajo pošiljanje informacij o pomembnih poslovnih dogodkih ljudem, ki jih to zadeva. Komunikacija pri notifikacijah je enosmerna. Opravilo dostavi sporočilo in ne čaka na odgovor.

Osnovni element WS-HumanTask specifikacije je <humanInteractions>, ki podpira opredelitev človeških interakcij. V tem elementu so opredeljena tako uporabniška opravila kot notifikacije, ima pa sledeče lastnosti:

- <expressionLanguage> je atribut, ki opredeljuje jezik za naslavljanje XML dokumentov. Privzet je jezik XPath.
- <queryLanguage> je atribut, ki opredeljuje jezik za poizvedbe v XML dokumentih. Tudi tukaj je privzet jezik XPath.
- <extensions> je element, ki opredeljuje imenske prostore atributov in elementov razširitev WS-HumanTask specifikacije. Ta element ni obvezen, če pa je prisoten, pa mora vsebovati vsaj en element <extension>.
- <import> je element, ki deklarira odvisnost od zunanjih WS-HumanTask in WSDL definicij. Ta element ima atribute <namespace>, <location> ter <importType>.
- <logicalPeopleGroups> je element, ki specificira vse logične skupine ljudi, ki jih uporabljajo spremljajoča uporabniška opravila. Vsebuje atributa name in reference ter element parameter.
- <tasks> je element, ki določa množico uporabniških opravil.
- <notifications> je element, ki določa množico notifikacij.

Element <humanInteractions> ne sme biti prazen, vsebovati mora vsaj en element [2].

2.7.2 Specifikacija BPEL4People

Specifikacija BPEL4People predstavlja razširitev BPEL-a za podporo večje množice scenarijev, ki v poslovnih procesih vključujejo ljudi in njihovo posredovanje. BPEL specifikacija se osredotoča na poslovne procese z avtomatiziranimi aktivnostmi, ki komunicirajo s spletnimi storitvami. Vendar pa je spekter aktivnosti, ki tvorijo splošen poslovni proces, precej širši. Ljudje pogosto sodelujejo v izvajanju poslovnega procesa, zato moramo računati na interakcijo med procesom in uporabniškim vmesnikom ter upoštevati človeški faktor.

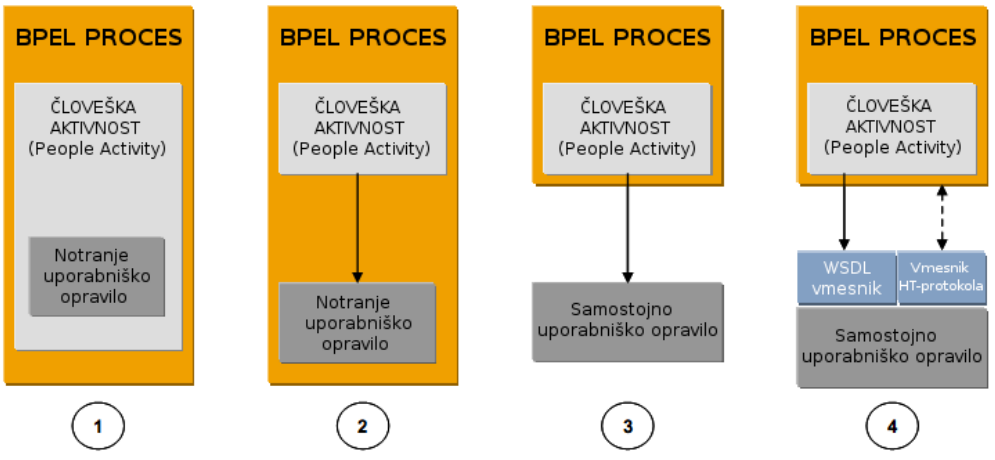
Ta specifikacija predstavlja množico elementov, ki razširjajo obstoječo množico standardnih BPEL elementov in omogočajo modeliranje človeških interakcij, ki lahko obsegajo tako enostavne potrditve kot kompleksne interakcije z ad-hoc podatki. Specifikacija vpelje človeško aktivnost kot nov tip aktivnosti, ki omogoča neposredno specificiranje človeške interakcije v procesu. Implementirana je lahko kot vgrajeno ali samostojno opravilo definirano z WS-HumanTask specifikacijo. BPEL4People specifikacija uporablja jezik XPath za dostop do vsebine procesa. Cilj te specifikacije je platformna neodvisnost in interoperabilnost (zmožnost komuniciranja različnih komponent).

BPEL4People proces mora uporabljati elemente iz razširitve BPEL4People ter elemente iz imenskega prostora WS-HumanTask. Element <humanInteractions> ni obvezen, če pa je prisoten, pa vsebuje deklaracije elementov iz imenskega prostora WS-HumanTask. To so elementi <logicalPeopleGroups>, <tasks> in <notifications>. Poleg elementa <humanInteractions> vsebuje še element <peopleAssignments> in tip <peopleActivity>. Element <peopleAssignments>

uporabimo za dodelitev uporabnikov v določene generične človeške vloge, ki so povezane s procesom. Tip <peopleActivity> je uporabljen za modeliranje človeških interakcij v BPEL procesu, ovit pa je v aktivnost z oznako <extensionActivity>.

Človeške interakcije so v BPEL proces lahko integrirane na več različnih načinov. Slika 2.7.2.1 prikazuje štiri primere. V prvih dveh primerih je uporabniško opravilo notranje, definirano je znotraj BPEL procesa. Uporabniško opravilo je lahko definirano kot del človeške aktivnosti. Tako opravilo prikazuje prvi primer, kjer je opravilo omejeno na komponento človeške aktivnosti, v kateri se nahaja. Uporabniško opravilo pa je lahko definirano tudi zunaj komponente človeške aktivnosti in na ta način lahko eno uporabniško opravilo uporablja več komponent človeških aktivnosti, kar je pomembno z vidika ponovne uporabe. Tretji primer prikazuje uporabo samostojnega uporabniškega opravila znotraj enega okolja brez specifikacije vmesnika. Zato je klic uporabniškega opravila odvisen od implementacije. Ta primer je podoben kot drugi primer, s tem da je uporabniško opravilo definirano zunaj BPEL procesa in je zato od njega neodvisno. Zaradi tega opravilo tudi nima neposrednega dostopa do vsebine procesa in njegovih podatkov. Četrti primer prikazuje uporabo samostojnega uporabniškega opravila v drugem okolju. Glavna razlika v primerjavi s tretjim primerom je, da ima tukaj uporabniško opravilo vmesnik, preko katerega ga lahko kličemo s pomočjo protokolov spletnih storitev. Za komunikacijo med procesom in uporabniškim opravilom je uporabljen koordinacijski protokol WS-HumanTask.

Prvi, drugi in četrti primer so prenosljivi med različnimi BPEL okolji, ki imajo implementirano specifikacijo BPEL4People. Precej konceptov v specifikaciji BPEL4People je podedovanih iz specifikacije WS-HumanTask, zato sta si specifikaciji precej podobni [3].

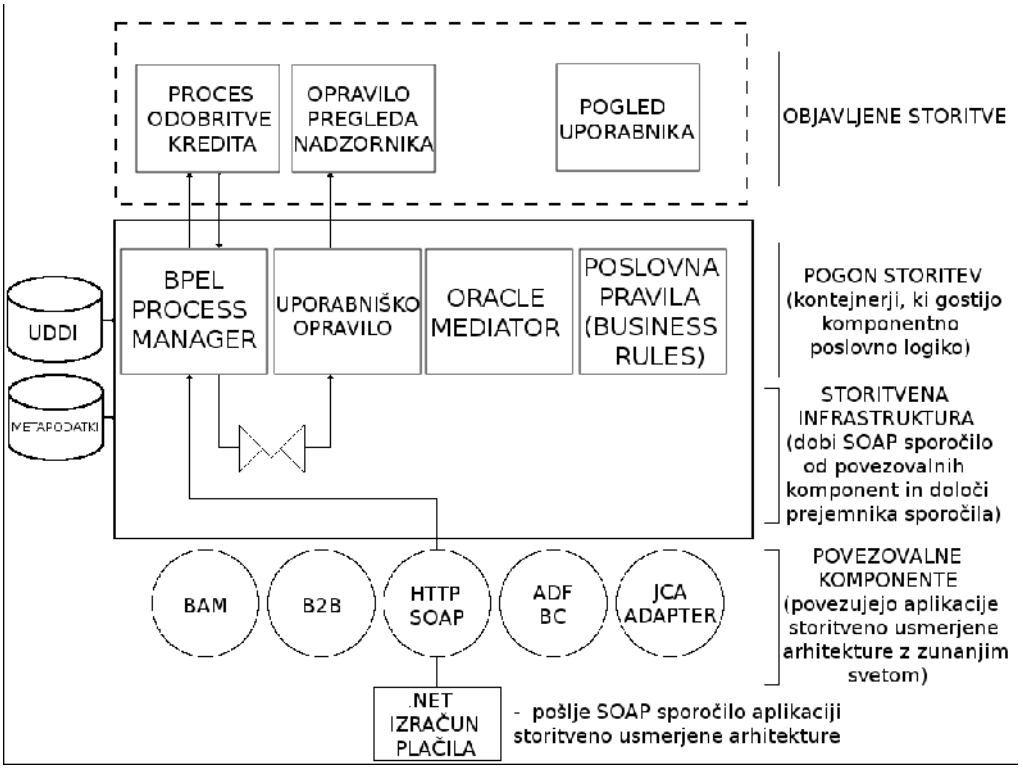


Slika 2.7.2.1: Načini integracije človeške interakcije

2.7.3 Uporabniška opravila v okviru Oraclove platforme

Implementacija diplomskega dela je realizirana na Oraclovi platformi z orodjem SOA Suite. Oracle ponuja SOA Suite kot nabor komponent za razvoj storitveno usmerjene arhitekture, objavo storitev na strežniku ter upravljanje z njimi. S pomočjo SOA Suite paketa storitve lahko izdelamo, upravljamo in orkestriramo v sestavljene aplikacije oziroma poslovne procese. Na ta način lahko sestavljamo aplikacije iz komponent, ki so razvite na različnih platformah in z različnimi tehnologijami. SOA Suite omogoča tudi postopno vpeljavo storitveno usmerjene arhitekture.

SOA Suite je sestavljen iz več komponent, ki imajo enoten način objave na strežniku, upravljanja, varnosti ter upravljanje metapodatkov. Zagotavlja nam tudi integrirano možnost pošiljanja sporočil, odkrivanja storitev, orkestracije, monitoringa, upravljanja ter varnosti spletnih storitev ter uporabo poslovnih pravil. Komponente Oracle SOA Suite so: BPEL Process Manager, Human Workflow Service, Business Rules, Business-to-Business Integration, Business Activity Monitoring, Enterprise Service Bus, Complex Event Processing ter JDeveloper.
Sestavljene aplikacije so sestavljene iz storitev, komponent storitev in referenc, ki so skupaj oblikovane in objavljene na strežniku kot ena aplikacija. Slika 2.7.3.1 predstavlja povezanost teh komponent med seboj in povezanost z drugimi sistemi. Z drugimi sistemi se lahko povezujejo s pomočjo povezovalnih komponent (binding components). Na sliki je predstavljen primer, ko zunanja aplikacija (v tem primeru .NET kalkulator) komunicira s sestavljeno aplikacijo [34].



Slika 2.7.3.1: Zunanja aplikacija komunicira s sestavljeno aplikacijo

Ena od komponent SOA Suite je Human Workflow Service. Ta komponenta skrbi za interakcijo poslovnega procesa z uporabniki in nam nudi več funkcionalnosti [34]:

- interakcijo poslovnega procesa z uporabniki, vključno z dodeljevanjem uporabnikov uporabniškim opravilom
- določanje rokov, obveščanje in preusmerjanje za pravočasno opravljanje uporabniških opravil
- obveščanje uporabnikov o opravilih na različne načine, najpomembnejša je aplikacija Oracle BPM Worklist
- organizacijo, filtriranje ter določanje prioritet, s pomočjo česar uporabniki bolj efektivno opravljajo svoja opravila
- poročila, preusmerjanje, enakomerno obremenjevanje uporabnikov za lažje razporejanje opravil med uporabnike

Human Workflow Service ima že vgrajene nekatere vzorce sestavljenih uporabniških opravil. Vendar so tu vzorci že določeni in definirani ter jih zato ne moremo spreminjati ali dodajati. V trenutno najnovejši verziji 11g so podprti enostavno uporabniško opravilo ter vzorci vzporedja, zaporedja in ad hoc.

3 Vsebina in namen diplomskega dela

V poslovnem svetu naraščajo zahteve po zanesljivosti in učinkovitosti, predvsem pa si poslovni uporabniki želijo fleksibilnih in čim bolj ekonomičnih poslovnih sistemov. Zaradi tega se pospešeno razvija področje avtomatizacije poslovnih procesov, za kar se vse pogosteje uporablja koncept storitveno usmerjene arhitekture. Poslovni proces je v storitveno usmerjeni arhitekturi sestavljen iz storitev, ki so med seboj neodvisne. Vsaka storitev predstavlja opravilo v poslovnem sistemu, ki je lahko avtomatizirano, ali pa pogojeno s posredovanjem uporabnika.

V diplomskem delu je pozornost posvečena vzorcem sestavljenih uporabniških opravil. Uporabniška opravila so spletne storitve, pri katerih je potrebno posredovanje uporabnika, kateremu je opravilo namenjeno. Za implementacijo je uporabljen jezik BPEL, ki je najbolj razširjen jezik za izvajanje poslovnih procesov. Implementirana je krovna storitev, ki kliče druge storitve, v katerih so implementirani vzorci sestavljenih uporabniških opravil. Te storitve pa potem komunicirajo s storitvami, ki implementirajo sama uporabniška opravila.

Namen diplomskega dela je identifikacija vzorcev, ki so smiselni in primerni za implementacijo, ter postavitev ustrezne arhitekture storitve. Pri identifikaciji množice vzorcev, ki so implementirani v storitvi, je upoštevano, da morajo imeti visoko stopnjo ponovne uporabljivosti in da posamezni klici storitve za izvajanje elementarnih opravil ne zahtevajo ohranjanja stanja. Pri postavitvi ustrezne arhitekture storitve pa so določene XML sheme ter vmesniki storitve in način na katerega generičen proces posameznega vzorca kliče različne storitve za posamezna uporabniška opravila.

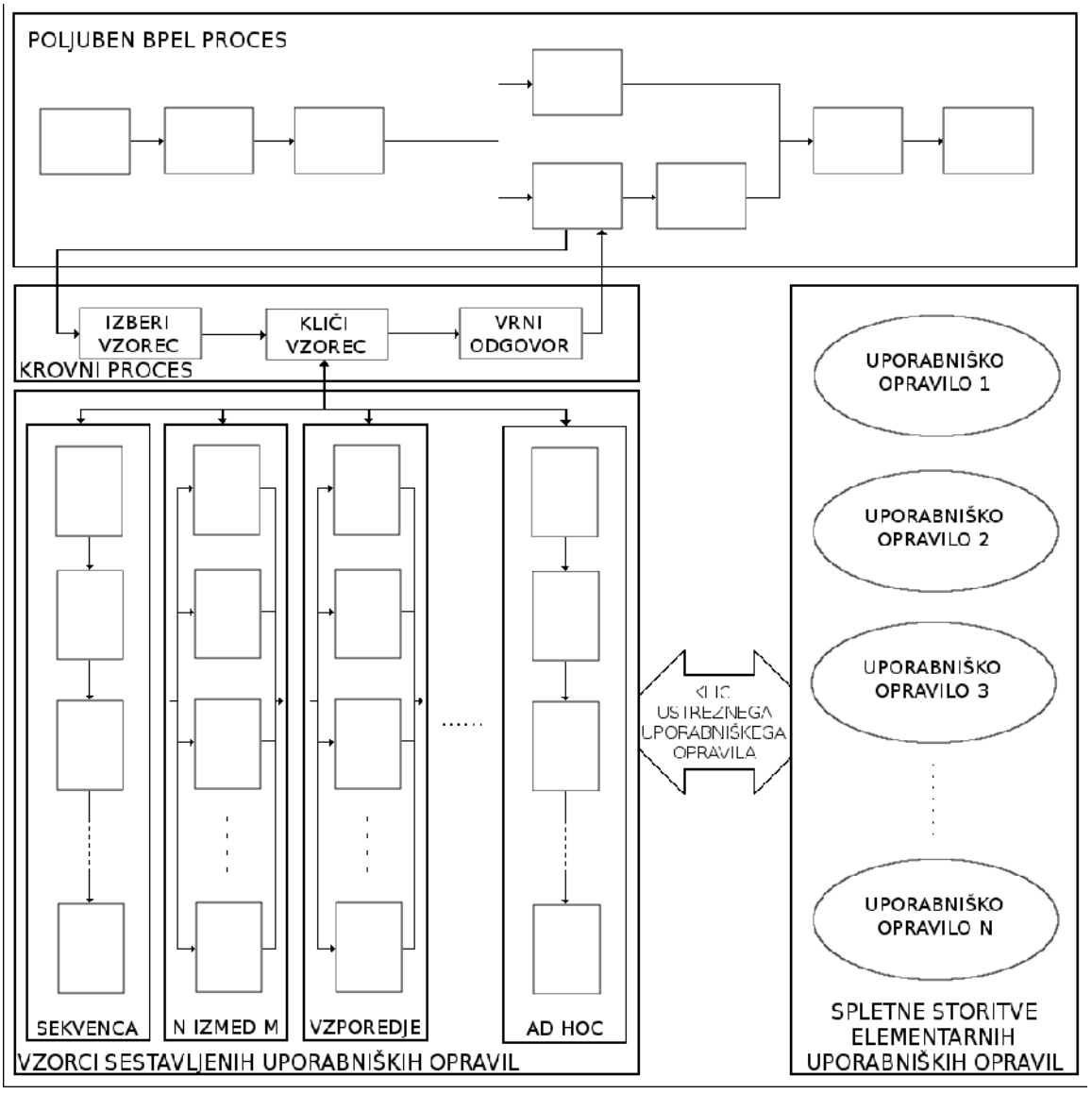
Poleg identifikacije vzorcev je namen diplomskega dela tudi implementacija storitve, ki podpira identificirane vzorce. Ker se določeni vzorci pogosto pojavljajo pri tokovih sestavljenih uporabniških opravil jih je smiselno prepoznati ter združiti v storitev, ki jih podpira, namesto da vedno znova kličemo določeno zaporedje sestavljenih uporabniških opravil. Množica identificiranih vzorcev ni popolna, saj se lahko spreminja od poslovnega sistema do poslovnega sistema. Zato je pri implementaciji storitve cilj tudi izdelava take storitve, kjer bo enostavno omogočeno dodajanje novih vzorcev oziroma spreminjanje množice vzorcev.

Arhitektura storitve je zasnovana tako, da se lahko implementira v obstoječih okoljih in orodjih, zato v poslovnih sistemih ni potrebno namestiti novih okolij. To nam omogočajo načela storitveno usmerjene arhitekture. V diplomskem delu sem se odločil za implementacijo storitve v okolju Oracle, ki je zelo razširjeno, lahko pa bi uporabil tudi katerega izmed množice ostalih orodij, ki so danes na voljo.

4 Vzorci sestavljenih uporabniških opravil

Zaradi avtomatizacije poslovnih procesov v poslovnih sistemih prihaja do integracije sistemov in ljudi, ki sodelujejo v teh sistemih. Poslovni procesi in z njimi povezani sistemi imajo določen življenjski cikel in določeno obnašanje. Uporabniki, ki sodelujejo v poslovnih procesih, imajo svoje vloge in z njimi povezana dovoljenja oziroma pooblastila, da opravljajo opravila v poslovnem procesu. S pomočjo sestavljenih uporabniških opravil lahko integriramo sisteme z več uporabniškimi opravili hkrati.

Kot vhod dobi krovni proces ime vzorca ter seznam uporabniških opravil. Seznam uporabniških opravil je lahko poljuben in tako imamo lahko ogromno kombinacij različnih uporabniških opravil v vsakem vzorcu. Vendar pa ne moremo izbirati povsem poljubnih uporabniških opravil, saj realizacija z dinamičnimi partnerskimi povezavami določa, da morajo imeti vse spletne storitve, ki jih kličemo z eno partnersko povezavo, enak vmesnik.



Slika 4.1: Arhitektura procesov vzorcev sestavljenih uporabniških opravil

Na sliki 4.1 je predstavljena arhitektura procesov, ki implementirajo vzorce sestavljenih uporabniških opravil. Na vrhu je poljuben BPEL proces, ki kliče Krovni proces. Ta nato kliče ustrezen vzorec sestavljenih uporabniških opravil. Klicani vzorec potem pošlje zahtevo ustreznim uporabniškim opravilom, ter čaka na njihov odgovor. Ko dobi odgovor, ga pošlje Krovnemu procesu, ta pa ga posreduje kot odgovor na zahtevo, ki jo je prejel na začetku.

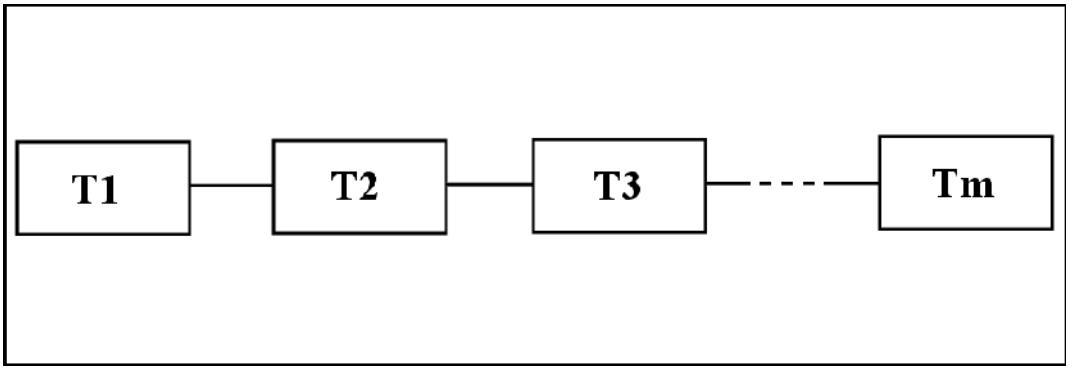
V nadaljevanju so opisani vzorci sestavljenih uporabniških opravil, ki so obdelani v diplomskem delu. Pri identifikaciji vzorcev sem si pomagal z vzorci opisanimi v [1] ter [35].

4.1 Sekvenca

Sekvenca je vzorec, pri katerem več uporabnikov, ali pa en sam uporabnik, opravi opravila v zaporedju. Gre za enostavno zaporedje, pri čemer se naslednje opravilo ne začne izvajati, dokler se prejšnje ne zaključi oziroma dokler ga uporabnik ne dokonča. V zaporedju je lahko poljubno mnogo opravil.

Primer vzorca:

Zaposleni pošlje zahtevek za nakup drobnega inventarja ali pisarniške opreme svojemu nadrejenemu. Nadrejeni presodi, če je zahtevek upravičen in ga lahko odobri ali zavrne. Če ga zavrne, napiše obrazložitev, če pa ga odobri, pa zahtevek avtomatsko potuje naprej do finančne službe, ki lahko odobri nakup, ali pa ga zavrne zaradi pomanjkanja sredstev. Odobren zahtevek potuje naprej do nabavne službe, ki zahtevane dobrine nabavi, ali pa sporoči, da niso dobavljive.



Slika 4.1.1: Vzorec sekvenca

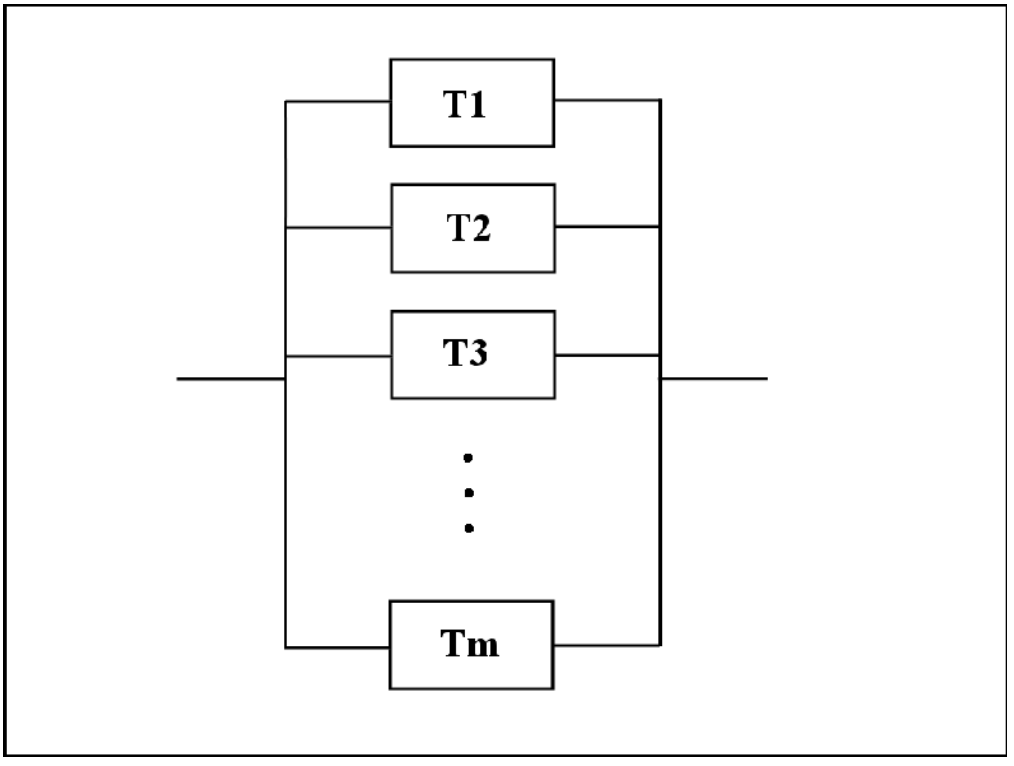
4.2 Vzporedni vzorci

4.2.1 Diskriminator

Pri diskriminatorju gre za vzporedni vzorec, pri katerem več uporabnikov hkrati dobi enako uporabniško opravilo. Vsi uporabniki začnejo opravljati opravilo in ko ga prvi konča, ostalim ni več potrebno nadaljevati, ampak prekinejo z delom. Opravilo lahko dobi poljubno mnogo uporabnikov.

Primer vzorca:

Nadrejeni v podjetju ima nujno opravilo, ki ga pošlje zaposlencem, ki so ga zmožni opraviti in imajo za to tudi pooblastilo, torej imajo določeno vlogo. Ker je opravilo nujno se mudi in več uporabnikov začne delati na tem opravilu. Kdor ga prvi opravi, pošlje odgovor nadrejenemu, ostali pa lahko prenehajo z opravljanjem.



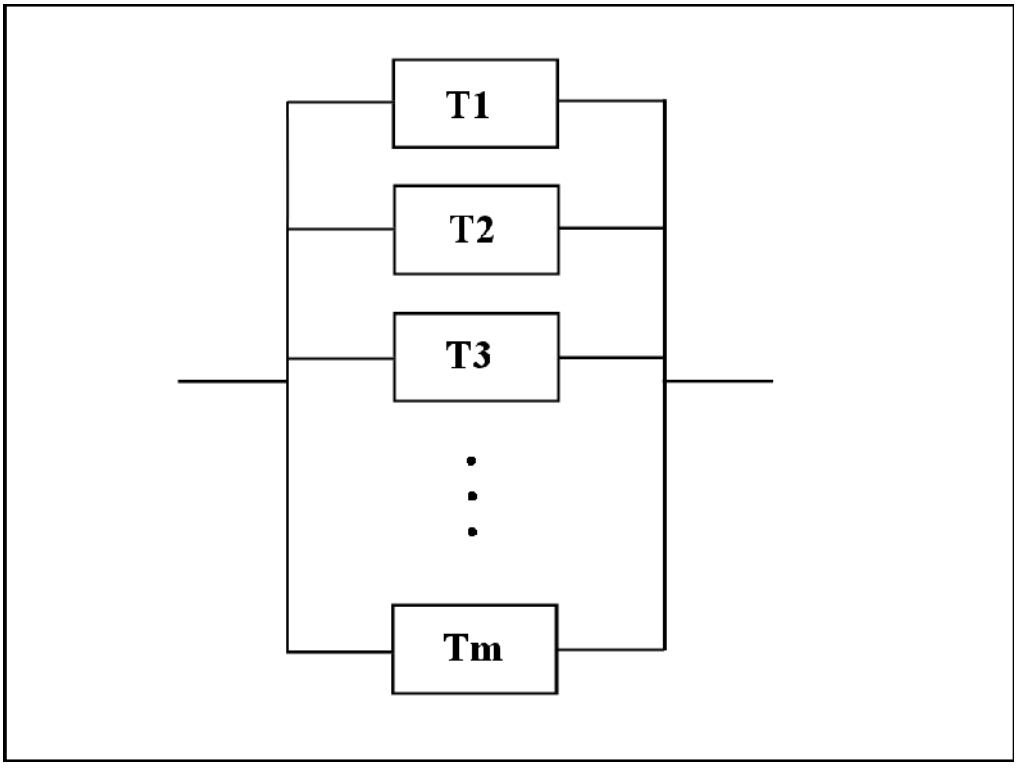
Slika 4.2.1.1: Vzorec diskriminator

4.2.2 N izmed M opravil

Ta vzorec je podoben primer kot diskriminator. Gre za vzporedni vzorec, pri katerem več uporabnikov dobi enako uporabniško opravilo. Število vseh uporabnikov je M. Za razliko od diskriminatorja, kjer samo en uporabnik dokonča opravilo, v tem vzorcu zahtevamo da N izmed M uporabnikov naredi dodeljeno opravilo. Lahko zahtevamo tudi, da vsi uporabniki naredijo opravilo. V tem primeru je N enako M.

Primer vzorca:

V poslovnem sistemu morajo odgovorni odločati o neki zadevi. Vsem odgovornim osebam pošljemo opravilo odločanja, pri čemer zahtevamo določeno udeležbo (N od M uporabnikov), da je odločanje veljavno. Ko dobimo N odgovorov, zaključimo z izvajanjem. Lahko pa tudi zahtevamo, da vsi uporabniki odgovorijo in sporočijo svojo odločitev. V tem primeru je N enako M.



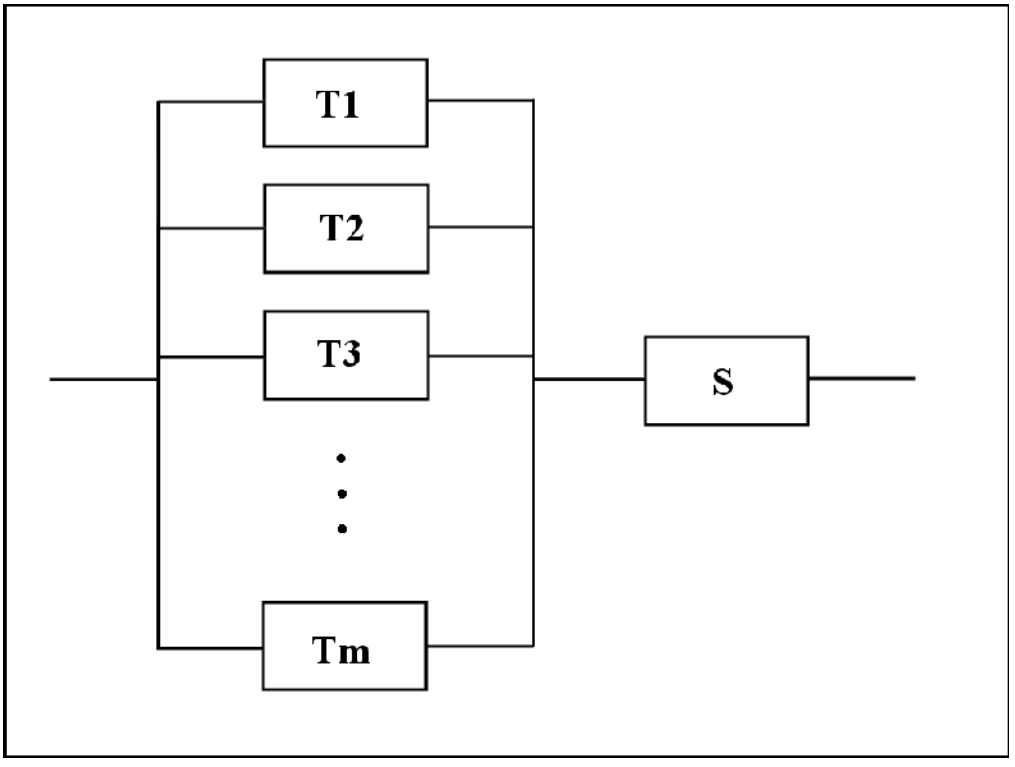
Slika 4.2.2.1: Vzorec N izmed M

4.2.3 Zlitje enakih

Zlitje enakih je vzporedni vzorec, pri katerem vsi uporabniki dobijo enako opravilo. Število vseh uporabnikov je M. Vzorec zahteva, da N izmed teh M uporabnikov dokonča opravilo. Lahko zahtevamo tudi, da vsi uporabniki dokončajo dodeljeno opravilo. Temu sklopu pa sledi še uporabniško opravilo, ki je poslano nadzorniku. Nadzornik preveri delo predhodnikov ter ga odobri oziroma zavrne, ali pa napiše kakšno opombo.

Primer vzorca:

V prejšnjem primeru smo odločanje izvajali med populacijo odgovornih v poslovnem sistemu, ki jim lahko zaupamo. Če pa odločanje razširimo na širšo populacijo, pa se lahko zgodi, da dobimo kakšne neresne, irelevantne odgovore. Zato na koncu odločanja uvedemo nadzornika, ki pregleda odgovore in ugotovi morebitne nepravilnosti.



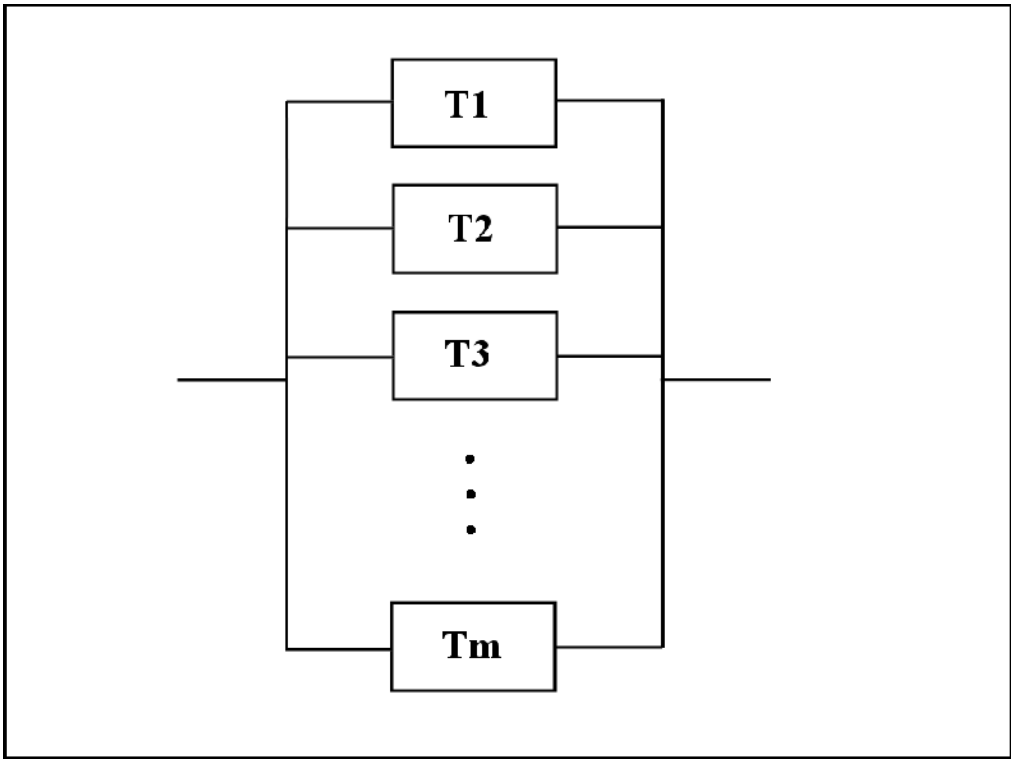
Slika 4.2.3.1: Vzorec zlitje enakih

4.2.4 Vzporedje

Vzporedje je tipičen vzporedni vzorec, pri katerem vsak od M uporabnikov dobi drugačno opravilo. Od vseh uporabnikov zahtevamo, da opravilo dokončajo.

Primer vzorca:

Kompleksno uporabniško opravilo lahko razbijemo na več elementarnih uporabniških opravil, ki jih uporabniki lahko delajo hkrati. Odgovorna oseba lahko na primer dobi več različnih prošenj, ki jih je treba pregledati ter odobriti ali zavrniti. Prošnje lahko poljubno razdeli med več uporabnikov sistema. Vsako prošnjo pošlje osebi, ki je specializirana za to področje. Ta nato pogleda če je prošnja popolna ter jo odobri ali zavrne.



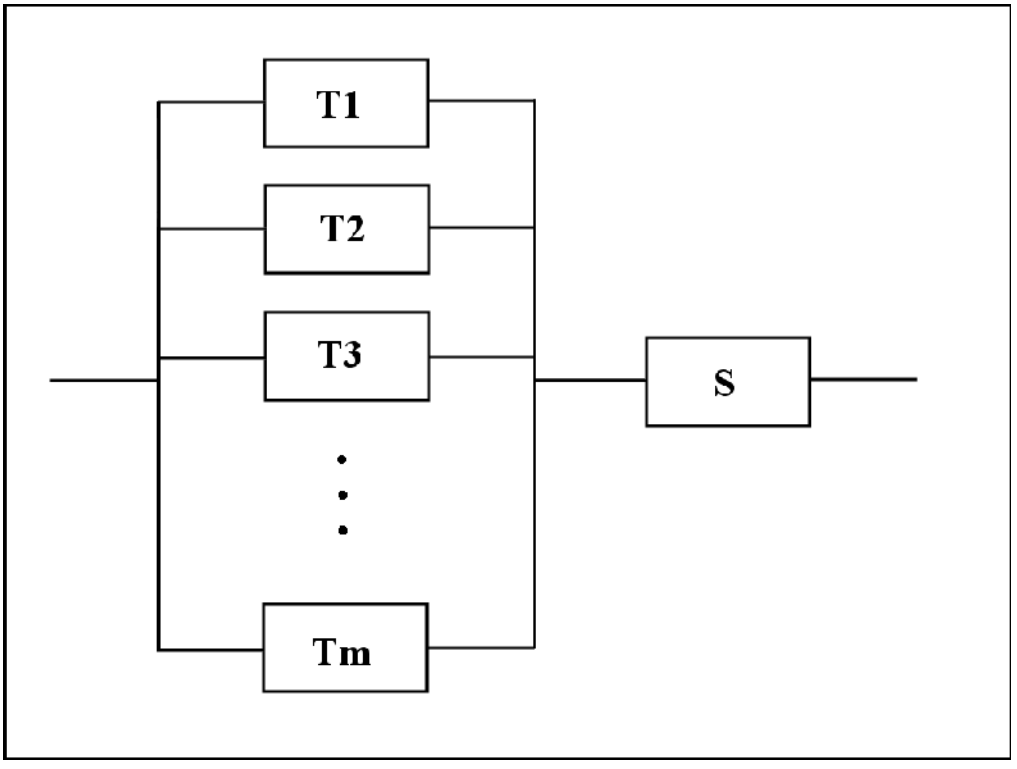
Slika 4.2.4.1: Vzorec vzporedje

4.2.5 Zlitje različnih

Zlitje različnih je vzporedni vzorec, pri katerem vsak uporabnik dobi v izvajanje drugačno opravilo. Število vseh uporabnikov je M. Pri tem vzorcu je zahtevano, da vsi uporabniki dokončajo dodeljena opravila. Temu sklopu sledi uporabniško opravilo, ki je poslano nadzorniku. Nadzornik preveri dokončana uporabniška opravila v vzorcu ter jih odobri oziroma zavrne, lahko pa napiše kakšno opombo.

Primer vzorca:

Podobno kot v prejšnjem primeru, pri vzorcu vzporedja, tudi tukaj kompleksno uporabniško opravilo razbijemo na več enostavnejših opravil. Tukaj vsak pogleda dodeljeno prošnjo ter ugotovi, če je popolna ter jo po potrebi dopolni. Nato da predlog o odobritvi ali zavrnitvi, nakar odgovorna oseba pogleda vse prošnje, ki so seveda popolne in se tudi s pomočjo predlogov podrejenih lahko hitro odloči katero bo odobrila in katero zavrnila.



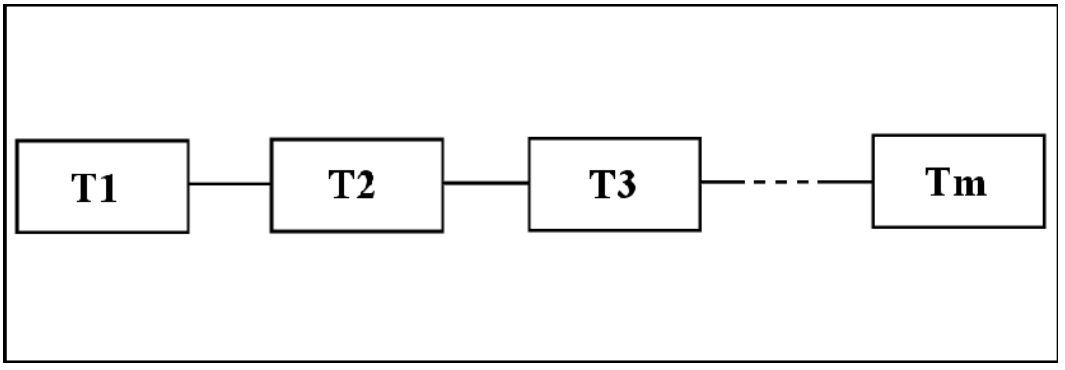
Slika 4.2.5.1: Vzorec zlitje različnih

4.3 Ad hoc ali dinamičen vzorec

Pri ad hoc oziroma dinamičnem vzorcu določimo zaporedje uporabniških opravil ter uporabnika, ki bo opravil prvo opravilo. Ta uporabnik nato izbere naslednika, ki bo opravil naslednje opravilo. Vzorec se nadaljuje tako, da vsak uporabnik izbere naslednjega uporabnika, dokler se ne izvedejo vsa uporabniška opravila.

Primer vzorca:

Tukaj gre za posebno vrsto sekvence. Če kompleksno uporabniško opravilo traja dalj časa, se lahko zgodi, da uporabnika, ki mu je eno od elementarnih opravil dodeljeno, ni na delovnem mestu. Bodisi je na dopustu bodisi so njegove delovne naloge začasno ali za stalno dodeljene drugemu uporabniku. Zato nadrejeni poleg vseh uporabniških opravil določi samo prvega uporabnika. Ko ta uporabnik konča svoje delo, ve kateremu uporabniku mora poslati naslednje uporabniško opravilo ter to stori. Tako vsak uporabnik razen zadnjega izbere svojega naslednika, ki je v tistem času na delovnem mestu in je odgovoren za področje naslednjega opravila.



Slika 4.3.1: Vzorec ad hoc

4.4 Nadaljevanje

Z nadaljevanjem oziroma kombinacijo vzorcev lahko pridemo do bolj kompleksnih vzorcev. Na ta način lahko zaporedno kombiniramo vse do sedaj opisane vzorce.

5 Implementacija

V diplomskem delu je za implementacijo vzorcev tokov sestavljenih uporabniških opravil uporabljen programski jezik BPEL. Vzorci so razviti z orodjem Oracle JDeveloper. Vsak vzorec je implementiran kot proces v jeziku BPEL. Ti procesi so potem na voljo na Oraclovem aplikacijskem strežniku kot spletne storitve.

5.1 Arhitektura implementacije

Implementacija je sestavljena iz več delov ter se izvaja na več nivojih. Osnovni del je BPEL proces, ki kliče BPEL procese na nižjem nivoju. Ti klici so izvedeni dinamično, saj je izbira procesa na nižjem nivoju odvisna od vhoda procesa na višjem nivoju.

Na vrhu je krovni proces, ki ga neposredno kličemo in poskrbi za dinamične klice ostalih BPEL procesov. Ta proces je na najvišjem nivoju in kliče procese na drugem nivoju. Nato čaka na njihove odgovore in ko mu zahtevano število procesov odgovori, se izvede do konca ter zaključi.

Na drugem nivoju so procesi, ki predstavljajo vzorce opisane v prejšnjem poglavju. Krovni proces ponavadi kliče samo enega izmed teh procesov, razen če gre za nadaljevanje procesov. V tem primeru lahko kličemo več procesov in s tem kombiniramo več vzorcev naenkrat ter tako dobimo bolj kompleksne vzorce.

Vsak izmed procesov na drugem nivoju kliče več storitev, ki predstavljajo uporabniška opravila. Te storitve kliče v različnih zaporedjih in vzorcih. Tudi tukaj so klici izvedeni dinamično, saj so klici uporabniških opravil odvisni od vhoda procesa na drugem nivoju.

5.2 Ključni izzivi

Pri izdelavi diplomskega dela sem najprej določil množico vzorcev sestavljenih uporabniških opravil. Za to množico velja, da so vzorci zadosti elementarni, da je možno njihovo kombiniranje za izdelavo kompleksnejših vzorcev ter njihova ponovna uporaba v čim večji meri. Pogoj je tudi, da posamezni klici storitve za izvajanje elementarnih uporabniških opravil ne zahtevajo ohranjanja stanja.

Nato sem identificirano množico vzorcev implementiral v programskem jeziku BPEL. Pri sami implementaciji vzorcev sestavljenih uporabniških opravil je bila potrebna uporaba dinamičnih klicev spletnih storitev. Vsak vzorec je namreč realiziran kot BPEL proces in se predstavlja kot spletna storitev.

5.3 Opisi posameznih delov

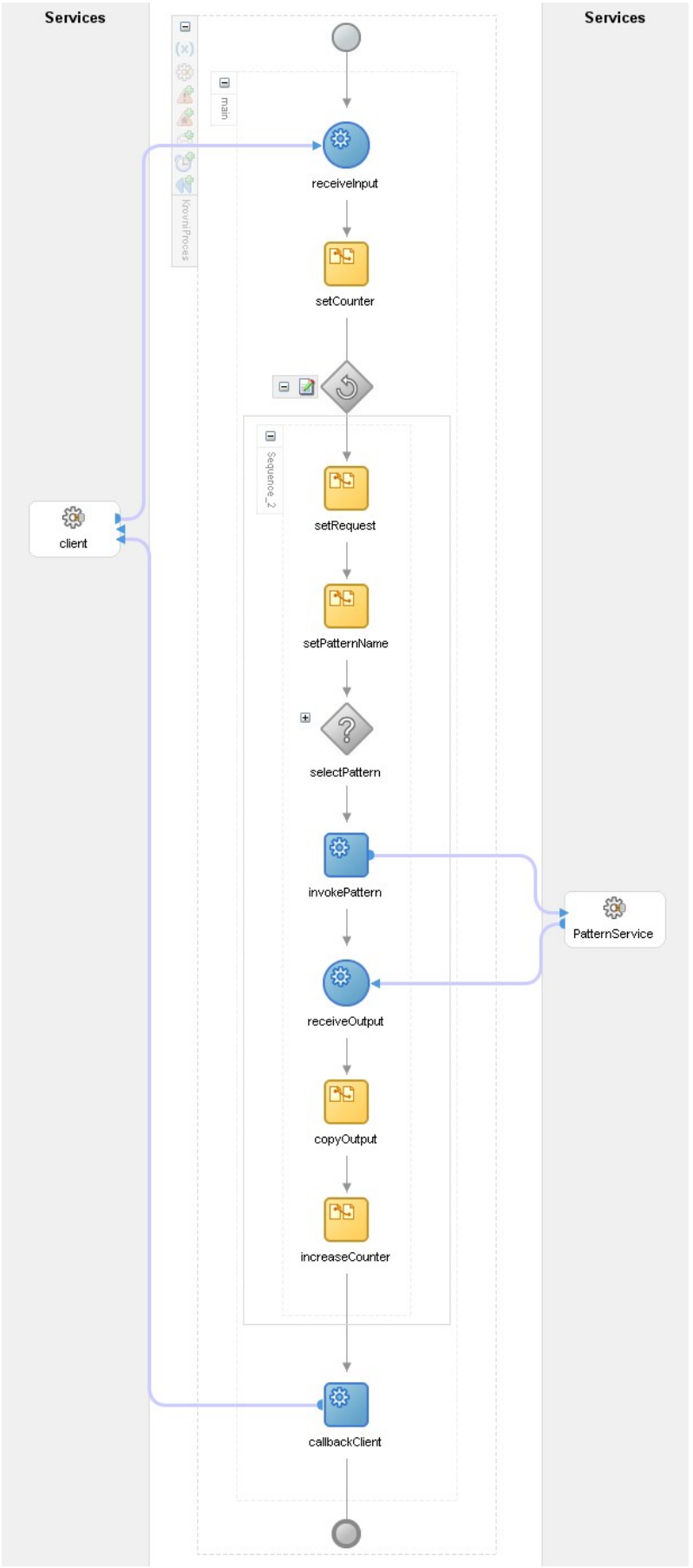
5.3.1 Krovni proces

Krovni proces je osnovna spletna storitev, s katero lahko uporabnik neposredno komunicira oziroma jo kliče. Kot vhod mora uporabnik storitvi posredovati podatkovni tip vzorci (patterns), ki vsebuje enega ali več vzorcev (pattern). V večini primerov gre samo za en vzorec, razen če želimo kombinirati več vzorcev. V tem primeru navedemo vse vzorce in jih pošljemo kot vhod krovnemu procesu.

Vsak podatkovni tip vzorec vsebuje zaporedno številko vzorca, ime vzorca, seznam uporabniških opravil, ki jih mora vzorec klicati, ter morebitno uporabniško opravilo nadzornika pri vzorcih zlitje enakih in zlitje različnih.

V procesu se za vsak vzorec kliče ustrezna spletna storitev, v kateri je implementiran proces tega vzorca. V case stavku selectPattern se v partner link PatternService zapiše naslov ter ime spletne storitve, ki jo proces v naslednjem koraku kliče. Ko proces dobi odgovor, ga zapiše v izhodno spremenljivko procesa. Če kličemo več vzorcev pa proces odgovor doda v izhodno spremenljivko poleg ostalih odgovorov.

Na sliki 5.3.1.1 je prikazan Krovni proces v orodju Oracle JDeveloper.



Slika 5.3.1.1: Slika krovnega procesa

5.3.2 Spletne storitve vzorcev

5.3.2.1 Sekvenca

Sekvenca je vzorec, ki uporabniška opravila kliče v zaporedju. Ko krovni proces pokliče to spletno storitev, ji kot vhod da vzorec, ki vsebuje seznam uporabniških opravil (userTasks). Vsako uporabniško opravilo (userTask) ima zaporedno številko, ime uporabniškega opravila ter vhod in izhod uporabniškega opravila.

V procesu se za vsako uporabniško opravilo kliče ustrezna spletna storitev v kateri je dejansko implementirano uporabniško opravilo. Proces počaka, da uporabnik dokonča svoje opravilo in pošlje odgovor. Ko proces dobi odgovor, ga doda v izhodno spremenljivko ter kliče naslednje uporabniško opravilo. Ko dobi odgovor od zadnjega uporabniškega opravila v vzorcu, se proces zaključi in pošlje celoten odgovor krovnemu procesu.

5.3.2.2 Diskriminator

Vzorec diskriminator kliče uporabniška opravila enega za drugim. Spletna storitev, v kateri je implementiran ta proces, kot vhod dobi od krovnega procesa vzorec, ki vsebuje seznam uporabniških opravil. Uporabniška opravila so oštevilčena in vsako uporabniško opravilo ima določen vhod ter spremenljivko, v katero se zapiše izhod.

Proces za vsako uporabniško opravilo na prejetem seznamu kliče ustrezno spletno storitev v kateri je implementirano uporabniško opravilo. Proces ne čaka na odgovore, ampak za vsako uporabniško opravilo na seznamu pokliče spletno storitev. Ko pošlje vse zahteve, čaka na odgovore vseh uporabniških opravil. Uporabniško opravilo, ki se prvo zaključi pošlje odgovor in ko ga proces dobi, ga zapiše v izhodno spremenljivko, ki jo kot odgovor pošlje krovnemu procesu in se s tem zaključi. Ostale odgovore uporabniških opravil ignorira, saj jih ne potrebuje.

5.3.2.3 N izmed M

V tem vzorcu proces zopet kliče enega za drugim vsa uporabniška opravila, ki jih spletna storitev dobi kot vhod od krovnega procesa. Tudi tukaj so uporabniška opravila na seznamu oštevilčena in vsako opravilo ima vhod ter spremenljivko, v katero se zapiše izhod.

Proces v spletni storitvi za vsako uporabniško opravilo na seznamu opravil kliče ustrezno spletno storitev v kateri je implementirano uporabniško opravilo. Postopek je podoben kot pri diskriminatorju. Tudi tukaj proces ne čaka odgovorov, ampak za vsako uporabniško opravilo na seznamu pokliče spletno storitev. Ko so vse zahteve poslane, pa proces čaka na odgovore. Tukaj se za razliko od diskriminatorja ne zadovolji samo z enim odgovorom, ampak čaka na določen odstotek odgovorov – od M poslanih zahtev mora

dobiti N odgovorov. Vsak prejet odgovor proces doda v izhodno spremenljivko in ko dobi N odgovorov pošlje odgovor krovnemu procesu in se zaključi. Ostalih M-N odgovorov ignorira.

5.3.2.4 Zlitje enakih

Ta vzorec je zadnji, pri katerem dobijo vsa uporabniška opravila enak vhod oziroma so vsa uporabniška opravila enaka, vendar poslana različnim uporabnikom. Proces v tem vzorcu kliče v zaporedju vsa uporabniška opravila, ki jih dobi kot vhod od krovnega procesa.

Proces v spletni storitvi za vsako uporabniško opravilo na prejetem seznamu pokliče ustrezno uporabniško opravilo, ki je implementirano kot spletna storitev. Proces ne čaka na odgovore, ampak pošlje zahteve vsem uporabniškim opravilom, ki so našteta na seznamu. Ko pošlje vse zahteve, čaka na odgovore. Do tega dela je proces enak kot proces v vzorcu N izmed M. Tudi tukaj proces čaka na določen odstotek odgovorov – N odgovorov mora prejeti od M poslanih zahtev. Odgovore dodaja v izhodno spremenljivko. Nato pa za razliko od prejšnjega vzorca kliče še nadzorno uporabniško opravilo, ki dobi kot vhod vse odgovore, ki jih je proces prejel. Nadzornik preveri, če je vse v redu in potrdi ali zavrne odgovore. Lahko pa zraven napiše kakšno opombo.

5.3.2.5 Vzporedje

V tem vzorcu so uporabniška opravila različna. Proces dobi od krovnega procesa vhod v katerem je seznam različnih uporabniških opravil, ki jih mora proces klicati vzporedno.

Proces ta uporabniška opravila kliče enega za drugim. Za vsako uporabniško opravilo na prejetem seznamu kliče spletno storitev v kateri je implementirano uporabniško opravilo. Ko konča cikel pošiljanja zahtev se začne cikel sprejemanja odgovorov. Ker so opravila na seznamu oštevilčena, je vrstni red prejemanja odgovorov lahko poljuben, saj s poznavanjem zaporedne številke opravila proces zapiše odgovor k pravi zahtevi. Proces mora počakati, da prejme toliko odgovorov kolikor je bilo poslanih zahtev. Odgovore proces dodaja v izhodno spremenljivko in šele ko prejme vse odgovore pošlje odgovor krovnemu procesu in se s tem zaključi.

5.3.2.6 Zlitje različnih

Vzorec zlitje različnih je podoben vzorcu vzporedja. Proces v tem vzorcu dobi od krovnega procesa vhod v katerem je seznam različnih uporabniških opravil, ki jih proces mora klicati.

Proces v spletni storitvi zaporedno kliče uporabniška opravila s prejetega seznama. Za vsako opravilo na seznamu proces kliče spletno storitev v kateri je implementirano uporabniško opravilo. Ko proces pošlje vse zahteve, čaka na

odgovore. Za razliko od zlitja enakih gre tukaj za različna uporabniška opravila, zato proces mora počakati na odgovore vseh uporabnikov, katerim je bila poslana zahteva za opravilo. Proces odgovore shrani v izhodno spremenljivko. Ko prejme vse odgovore jih pošlje nadzorniku. To stori s klicanjem nadzornega uporabniškega opravila. Nadzornik preveri, če so vsi odgovori v redu in jih potrdi ali zavrne. Zraven lahko doda opombo in vse skupaj pošlje nazaj procesu v vzorcu zlitja različnih. Ta pošlje odgovor krovnemu procesu ter se s tem zaključi.

5.3.2.7 Ad hoc

Ad hoc je vzorec, ki od krovnega procesa dobi seznam uporabniških opravil, ki jih mora klicati, ter uporabnika, ki dobi prvo uporabniško opravilo. Za ostala uporabniška opravila ni podanih uporabnikov. Ko prvi uporabnik zaključi opravilo, izbere uporabnika ki bo naredil naslednje uporabniško opravilo. Proces v ad hoc vzorcu dobi odgovor od uporabnika ter iz odgovora razbere naslednjega uporabnika. Temu uporabniku pošlje naslednje opravilo na seznamu. Vsak uporabnik, ki prejme zahtevo za uporabniško opravilo, mora izbrati tudi uporabnika, ki bo opravil naslednje opravilo. Edino zadnjemu uporabniku ni potrebno izbirati naslednika, ampak samo naredi zahtevano opravilo.

Proces shrani odgovor vsakega uporabnika ter iz njegovega odgovora razbere tudi kateremu uporabniku naj pošlje naslednje uporabniško opravilo. Ko dobi odgovor od zadnjega uporabnika, pošlje odgovor krovnemu procesu ter se zaključi.

5.3.2.8 Nadaljevanje

Vzorec nadaljevanje je implementiran drugače kot do sedaj opisani vzorci. Ker gre za poljubno kombinacijo vseh opisanih vzorcev, je potrebno na začetku krovnemu procesu podati to kombinacijo. Krovni proces je do sedaj vedno dobil kot vhod vzorec, kjer je bilo podano ime vzorca, seznam uporabniških opravil ter morebitno nadzorno uporabniško opravilo. Krovni proces pa lahko dobi kot vhod tudi seznam več vzorcev. Vsak vzorec ima zaporedno številko, ime vzorca ter seznam navadnih in nadzornih uporabniških opravil za ta vzorec.

Krovni proces kliče vzorce enega za drugim, ti pa kličejo uporabniška opravila s svojega seznama. Ko se proces v enem vzorcu zaključi, krovni proces kliče proces v naslednjem vzorcu. Na koncu dobi odgovore od vseh vzorcev in jih združi.

5.3.3 XML sheme

XML sheme vhodov vseh procesov so definirane v datoteki UserTaskTypes.xsd. Vhod krovnega procesa je podatkovnega tipa PatternsType, ki vsebuje enega ali več vnosov podatkovnega tipa PatternType.

V vsakem podatkovnem tipu PatternType so polja zaporedna številka vzorca oziroma patternNo, ki je tipa integer, ter ime vzorca oziroma patternName, ime nadzornega uporabniškega opravila oziroma supervisionTask in odgovor nazdornika oziroma supervisorReply, ki so tipa string. V podatkovnem tipu pattern je tudi en ali več vnosov podatkovnega tipa UserTaskType.

Podatkovni tip UserTaskType vsebuje polja zaporedna številka uporabniškega opravila oziroma taskNo, ki je tipa integer ter ime uporabniškega opravila oziroma taskName, ki je tipa string. Poleg teh dveh polj vsebuje podatkovni tip UserTaskType tudi polji userTaskInput in userTaskOutput, ki sta sestavljenih tipov UserTaskInputType in UserTaskOutputType.

Tip UserTypeInputType je sestavljen iz polj iniciator oziroma initiator, uporabnik oziroma user ter zadeva oziroma subject. Vsa ta polja so tipa string.

V tipu UserTypeOutputType pa so polja uporabnik oziroma user, odgovor oziroma reply, naslednji uporabnik oziroma nextUser ter napaka oziroma error. Tudi tukaj so vsa polja tipa string.

V nadaljevanju je predstavljena XML shema:

```
<?xml version="1.0"?>
<xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.autotask.com/ns/autotask"
        xmlns:tns="http://www.autotask.com/ns/autotask">
 <xsd:element name="userTaskInput" type="tns:UserTaskInputType"/>
 <xsd:element name="userTaskOutput" type="tns:UserTaskOutputType"/>
 <xsd:element name="userTask" type="tns:UserTaskType"/>
 <xsd:element name="pattern" type="tns:PatternType"/>
 <xsd:element name="patterns" type="tns:PatternsType"/>
 <xsd:complexType name="UserTaskType">
  <xsd:sequence>
   <xsd:element name="taskNo" type="xsd:integer"/>
   <xsd:element name="taskName" type="xsd:string"/>
   <xsd:element name="userTaskInput" type="tns:UserTaskInputType"/>
   <xsd:element name="userTaskOutput" type="tns:UserTaskOutputType"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="PatternsType">
  <xsd:sequence>
```

```
      <xsd:element name="pattern" type="tns:PatternType" minOccurs="1"
maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
 <xsd:complexType name="PatternType">
  <xsd:sequence>
   <xsd:element name="patternNo" type="xsd:integer"/>
   <xsd:element name="patternName" type="xsd:string"/>
    <xsd:element name="supervisionTask" type="xsd:string"/>
    <xsd:element name="superviserReply" type="xsd:string"/>
    <xsd:element name="userTasks" type="tns:UserTaskType" minOccurs="1"
maxOccurs="unbounded"/>
   </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="UserTaskInputType">
  <xsd:sequence>
   <xsd:element name="initiator" type="xsd:string" nillable="false"/>
   <xsd:element name="user" type="xsd:string" nillable="false"/>
   <xsd:element name="subject" type="xsd:string" nillable="false"
            minOccurs="1"/>
   </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="UserTaskOutputType">
  <xsd:sequence>
   <xsd:element name="user" type="xsd:string"/>
   <xsd:element name="reply" type="xsd:string"/>
   <xsd:element name="nextUser" type="xsd:string"/>
   <xsd:element name="error" type="xsd:string" default="""/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

5.4 Konkretna implementacija v okolju Oracle

Krovni proces in spletne storitve vzorcev so razvite v orodju Oracle JDeveloper Studio Edition Version 10.1.3.1.0.3984. Za objavo storitev potrebujemo strežnik Oracle Application Server iz SOA Suite-a, ki mora biti tudi verzije 10.1.3. Storitve sem poskusil objaviti tudi na novejši verziji strežnika, vendar so se že pri verziji 10.1.4. pojavili manjši problemi, zaradi česar procesi storitev niso delovali v celoti.

5.4.1 Krovni proces

Krovni proces je spletna storitev na prvem nivoju. To storitev uporabnik neposredno kliče. Vhod tega procesa je inputVariable, ki je tipa sporočilo (message type) KrovniProcesRequestMessage. Sporočilo KrovniProcesRequestMessage je definirano v WSDL datoteki KrovniProces.wsdl in ima podatkovno strukturo patterns, ki je definirana v podatkovni shemi UserTaskTypes.xsd, ki je opisana v poglavju 5.3.3.

Ko proces dobi vhod za vsak vzorec v vhodu enkrat izvede vsebino while zanke. Pogoj v zanki je:

bpws:getVariableData('counter') <= count(bpws:getVariableData('inputVariable','payload','/ns1:patterns/ns1:pattern'))

Na začetku iteracije je trenutni vzorec kopiran v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokePatterns:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/ns1:patterns/ns1:pattern[number(bpws:getVariableData('counter'))]"/>
   <to variable="request" part="payload"
      query="/ns1:pattern"/>
</copy>
```

V naslednjem koraku je ime trenutnega vzorca kopirano v spremenljivko patternName, na podlagi česar se v switch stavku izbere ustrezen vzorec, katerega storitev bo proces klical:

```
<case condition="bpws:getVariableData('patternName') = 'sekvenca'">
   <bpelx:annotation>
      <bpelx:pattern>Sekvenca</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setSekvenca">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
               <Address>http://domaci:8888/orabpel/default/Sekvenca</Address>
               <ServiceName
xmlns:ns1="http://samples.otn.com">ns1:Sekvenca</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="PatternService"/>
      </copy>
```

```
    </assign>
 </case>
```

Kot primer je naveden case stavek za vzorec sekvenca. V partnersko povezavo PatternService je potrebno kopirati endpoint referenco, ki vsebuje naslov storitve ter ime storitve. Če ni izbran noben vzorec oziroma je ime vzorca napačno, se v izhodno spremenljivko napiše napaka, spremenljivka se vrne kot izhod procesa, proces pa se zaključi:

```
<assign name="setError">
   <copy>
      <from expression="'Napaka pri izbiri vzorca'"/>
      <to variable="outputVariable" part="payload"
         query="/ns1:pattern/ns1:patternName"/>
   </copy>
  </assign>
<invoke name="reportError" partnerLink="client"
      portType="client:KrovniProcesCallback"
      operation="onResult"
      inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

V kolikor ne pride do napake, se proces nadaljuje in v nadaljevanju sledita aktivnosti invokePattern ter receiveOutput. Z aktivnostjo invokePattern proces kliče ustrezno storitev v kateri je implementiran zahtevan vzorec. Kot vhod za klic vzorca pošlje proces spremenljivko request, ki je definirana na začetku iteracije. Aktivnost receiveOutput pa sprejme odgovor klicane storitve, ki ga zapiše v spremenljivko response:

```
<invoke name="invokePattern" partnerLink="PatternService"
      portType="services:PatternService" operation="initiate"
      inputVariable="request"/>
 <receive name="receiveOutput" partnerLink="PatternService"
      portType="services:PatternServiceCallback"
      operation="onResult" variable="response"
      createInstance="no"/>
```

Na koncu proces kopira vrednost spremenljivke response v izhodno spremenljivko krovnega procesa outputVariable, ki je tipa sporočilo (message type) KrovniProcesResponseMessage. Sporočilo KrovniProcesResponseMessage je definirano v WSDL datoteki KrovniProces.wsdl in ima podatkovno strukturo patterns, ki je definirana v podatkovni shemi UserTaskTypes.xsd.

5.4.2 Sekvenca

Sekvenca je spletna storitev na drugem nivoju. To spletno storitev kliče Krovni proces. Proces v spletni storitvi Sekvenca dobi kot vhod spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Ta tip je skupen za vse spletne storitve na drugem nivoju, ki implementirajo vzorce sestavljenih uporabniških opravil. Definiran je v WSDL datoteki Sekvenca.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.
Na začetku proces kopira vrednost vhodne spremenljivke v izhodno spremenljivko outputVariable. Vhodnim podatkom v izhodni spremenljivki na koncu priključi še izhodne podatke in tako imamo zahtevo in odgovor v eni spremenljivki:

```
<copy>
   <from variable="inputVariable" part="payload"
      query="/auto:pattern"/>
   <to variable="outputVariable" part="payload"
      query="/auto:pattern"/>
</copy>
```

Nato proces v zanki izvede iteracijo za vsako uporabniško opravilo v vzorcu. Pogoj v zanki je:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/auto:pattern/auto:userTas
ks'))
```

Na začetku iteracije proces definira zahtevo za uporabniško opravilo. To stori tako, da kopira vrednost ustreznega uporabniškega opravila v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable"
query="/auto:pattern/auto:userTasks[number(bpws:getVariableData('counter'))]"
                  part="payload"/>
   <to variable="request" part="payload"
                  query="/auto:userTask"/>
</copy>
```

V naslednjem koraku proces v spremenljivko taskName zapiše ime trenutnega uporabniškega opravila in na podlagi te spremenljivke v switch stavku izbere ustrezno uporabniško opravilo, ki ga bo klical:

```
<case condition="bpws:getVariableData('taskName') = 'odlocanje'">
   <bpelx:annotation>
      <bpelx:pattern>Odlocanje</bpelx:pattern>
   </bpelx:annotation>
```

```
   <assign name="setOdlocanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
            <Address>http://domaci:8888/orabpel/default/Odlocanje</Address>
             <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Odlocanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="TaskService"/>
      </copy>
   </assign>
</case>
```

Naveden je primer klica uporabniškega opravila odločanje. V partnersko povezavo TaskService je potrebno kopirati endpoint referenco, ki vsebuje naslov storitve ter ime storitve. Če ni izbrano nobeno uporabniško opravilo oziroma je ime uporabniškega opravila napačno, se v izhodno spremenljivko napiše napaka, spremenljivka se vrne kot izhod procesa, proces pa se zaključi:

```
<assign name="setError">
        <copy>
            <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
            <to variable="outputVariable" part="payload"
query="/auto:pattern/auto:userTasks[number(bpws:getVariableData('counter'))]/
auto:userTaskOutput/auto:error"/>
        </copy>
</assign>
<invoke name="reportError" partnerLink="client"
        portType="tns:PatternServiceCallback"
        operation="onResult"
        inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

Če pri izbiri uporabniškega opravila ne pride do napake, se proces nadaljuje. Proces v nadaljevanju izvede aktivnosti invokeTask in receiveInput. Z aktivnostjo invokeTask pokliče storitev izbranega uporabniškega opravila. Kot vhod klica opravila proces uporabi spremenljivko request, ki je definirana na začetku iteracije. Z aktivnostjo receiveOutput pa proces sprejme odgovor uporabniškega opravila, ki ga zapiše v spremenljivko response:

```
<invoke name="invokeTask" partnerLink="TaskService"
      portType="services:TaskService" operation="initiate"
      inputVariable="request"/>
```

```
<receive name="receiveOutput" partnerLink="TaskService"
      portType="services:TaskServiceCallback"
      operation="onResult" variable="response"
      createInstance="no"/>
```

Proces na koncu kopira vrednost spremenljivke response v izhodno spremenljivko procesa Sekvenca outputVariable, ki je tipa sporočilo PatternServiceResponseMessage. Tip PatternServiceResponseMessage je definiran v WSDL datoteki Sekvenca.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd. Zadnji korak procesa je vračanje odgovora Krovnemu procesu v obliki spremenljivke outputVariable.

5.4.3 Diskriminator

Diskriminator je spletna storitev na drugem nivoju. Krovni proces jo kliče, če je izbran vzorec diskriminator. Kot vhod dobi proces v tej spletni storitvi spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Ta tip je definiran v datoteki Diskriminator.wsdl in ima podatkovno strukturo pattern. Podatkovna struktura pattern je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces na začetku kopira vrednost vhodne spremenljivke v izhodno spremenljivko outputVariable:

```
<copy>
  <from variable="inputVariable" part="payload"
     query="/auto:pattern"/>
  <to variable="outputVariable" part="payload"
    query="/auto:pattern"/>
</copy>
```

V tem procesu so vsa uporabniška opravila enaka, poslana pa so različnim uporabnikom. Ker so opravila enaka, proces pred zanko v switch stavku določi za katero uporabniško opravilo gre:

```
<case condition="bpws:getVariableData('inputVariable','payload',
'/auto:pattern/auto:userTasks[1]/auto:taskName') = 'glasovanje'">
  <bpelx:annotation>
    <bpelx:pattern>Glasovanje</bpelx:pattern>
  </bpelx:annotation>
  <assign name="setGlasovanje">
    <copy>
      <from>
        <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
          <Address>http://domaci:8888/orabpel/default/Glasovanje</Address>
```

```
            <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Glasovanje</ServiceName>
          </EndpointReference>
        </from>
        <to partnerLink="TaskService"/>
      </copy>
    </assign>
  </case>
```

V zgornjem primeru je naveden case stavek za uporabniško opravilo glasovanje. Proces v partnersko povezavo TaskService kopira endpoint referenco, ki vsebuje naslov storitve ter ime storitve. Če ni izbrano nobeno uporabniško opravilo oziroma je ime opravila napačno, proces v izhodno spremenljivko zapiše napako, jo vrne kot izhod ter se na tem mestu zaključi:

```
<assign name="setError">
   <copy>
     <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
     <to variable="outputVariable" part="payload"
         query="/auto:pattern/auto:userTasks[1]/auto:
           userTaskOutput/auto:error"/>
   </copy>
</assign>
<invoke name="reportError" partnerLink="client"
     portType="tns:PatternServiceCallback"
     operation="onResult"
     inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

Če do napake ne pride, se proces nadaljuje. Sledi while zanka s pogojem:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/auto:pattern/auto:userTas
ks'))
```

Zanka se izvede za vsako uporabniško opravilo v vzorcu. Na začetku iteracije proces kopira vrednost ustreznega uporabniškega opravila vhodne spremenljivke v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/auto:pattern/auto:userTasks[number(bpws:getVariableData('counter'))]"
/>
   <to variable="request" part="payload"
     query="/auto:userTask"/>
```

```
</copy>
```

V naslednjem koraku se izvede aktivnost invokeTask, ki kliče trenutno uporabniško opravilo. Kot vhod pošlje spremenljivko request:

```
<invoke name="invokeTask" partnerLink="TaskService"
   portType="services:TaskService" operation="initiate"
   inputVariable="request"/>
```

Tukaj se iteracija zaključi. Za while zanko je v procesu aktivnost receiveOutput, ki čaka na odgovore klicanih uporabniških opravil:

```
<receive name="receiveOutput" partnerLink="TaskService"
    portType="services:TaskServiceCallback" operation="onResult"
    variable="response" createInstance="no"/>
```

Ko proces prejme prvi odgovor od klicanih uporabniških opravil, kopira vrednost spremenljivke response v izhodno spremenljivko outputVariable, ki je tipa sporočilo PatternServiceResponseMessage. Ta tip je definiran v WSDL datoteki Diskriminator.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd. Na koncu proces vrne odgovor Krovnemu procesu v spremenljivki outputVariable.

5.4.4 N izmed M

Spletno storitev NizmedM kliče krovni proces, ko je izbran vzorec N izmed M. Tudi ta storitev je na drugem nivoju. Vhod procesa v storitvi je spremenljivka inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Tip je definiran v datoteki NizmedM.wsdl in ima podatkovno strukturo pattern. Podatkovna struktura pattern je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces najprej kopira vrednost vhodne spremenljivke inputVariable v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="inputVariable" part="payload"
       query="/auto:pattern"/>
  <to variable="outputVariable" part="payload"
      query="/auto:pattern"/>
</copy>
```

Tudi v tem procesu so, podobno kot v procesu spletne storitve Diskriminator, enaka uporabniška opravila poslana različnim uporabnikom. Zato tudi tukaj lahko proces že pred zanko določi za katero uporabniško opravilo gre:

```
<case condition="bpws:getVariableData('inputVariable','payload',
'/auto:pattern/auto:userTasks[1]/auto:taskName') = 'odlocanje'">
   <bpelx:annotation>
```

```
      <bpelx:pattern>Odlocanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setOdlocanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
               <Address>http://domaci:8888/orabpel/default/Odlocanje</Address>
               <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Odlocanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="TaskService"/>
      </copy>
   </assign>
</case>
```

Naveden case stavek je za primer, ko je izbrano uporabniško opravilo odločanje. Proces v partnersko povezavo TaskService kopira endpoint referenco, ki vsebuje naslov ter ime storitve. Če je ime opravila napačno oziroma ni izbrano nobeno opravilo, proces zapiše napako v izhodno spremenljivko in jo vrne kot izhod ter se zaključi:

```
<assign name="setError">
   <copy>
      <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
      <to variable="outputVariable" part="payload"

query="/auto:pattern/auto:userTasks[1]/auto:userTaskOutput/auto:error"/>
   </copy>
</assign>
<invoke name="reportError" partnerLink="client"
      portType="client:PatternServiceCallback"
      operation="onResult"
      inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

Če ne pride do napake, se proces nadaljuje. Sledi zanka s pogojem:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/auto:pattern/auto:userTas
ks'))
```

Zanka se izvede za vsako uporabniško opravilo v vzorcu. Število opravil je enako M. V vsaki iteraciji proces kopira vrednost trenutnega opravila v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/auto:pattern/auto:userTasks[number(bpws:getVariableData('counter'))]"
/>
   <to variable="request" part="payload"
      query="/auto:userTask"/>
</copy>
```

Zatem se v aktivnosti invokeTask izvede klic ustreznega uporabniškega opravila, ki kot vhod pošlje spremenljivko request:

```
<invoke name="invokeTask" partnerLink="TaskService"
   portType="services:TaskService" operation="initiate"
   inputVariable="request"/>
```

Na tem mestu se iteracija zaključi, prvi while zanki pa sledi druga while zanka s pogojem:

```
bpws:getVariableData('i') <= ceiling((bpws:getVariableData('counter')-1)  * 0.5)
```

Ta zanka se izvede N-krat. V danem primeru je N = M * 0.5, lahko pa bi N tudi spremenili, ali pa ga določili dinamično z vhodom v proces. V tej zanki proces v aktivnosti receiveOutput v vsaki iteraciji sprejme en odgovor od klicanih uporabniških opravil:

```
<receive name="receiveOutput" partnerLink="TaskService"
    portType="services:TaskServiceCallback"
    operation="onResult" variable="response"
    createInstance="no"/>
```

Na koncu iteracije vsak prejet odgovor proces kopira iz spremenljivke response v izhodno spremenljivko outputVariable, nato pa se zanka zaključi. Spremenljivka outputVariable je tipa sporočilo PatternServiceResponseMessage. Ta tip je definiran v datoteki NizmedM.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd. Proces vrne odgovor Krovnemu procesu v spremenljivki outputVariable.

5.4.5 Zlitje enakih

Ko je izbran vzorec zlitje enakih, Krovni proces kliče spletno storitev ZlitjeEnakih, ki je na drugem nivoju. Proces kot vhod dobi spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Tip je definiran v datoteki ZlitjeEnakih.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces na začetku kopira vrednost vhodne spremenljivke inputVariable v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="inputVariable" part="payload"
        query="/ns4:pattern"/>
  <to variable="outputVariable" part="payload"
      query="/ns4:pattern"/>
</copy>
```

Ta proces je zadnji proces, pri katerem so različnim uporabnikom poslana enaka uporabniška opravila. Proces zato v naslednjem koraku pred zanko določi katero uporabniško opravilo bo potrebno klicati:

```
<case condition="bpws:getVariableData('inputVariable','payload',
'/ns4:pattern/ns4:userTasks[1]/ns4:taskName') = 'odlocanje'">
   <bpelx:annotation>
      <bpelx:pattern>Odlocanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setOdlocanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
               <Address>http://domaci:8888/orabpel/default/Odlocanje</Address>
               <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Odlocanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="TaskService"/>
      </copy>
   </assign>
</case>
```

Naveden je case stavek za primer, ko je izbrano uporabniško opravilo odločanje. Proces v tem delu kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo TaskService. Če pri izbiri uporabniškega opravila pride do napake, proces zapiše napako v izhodno spremenljivko outputVariable, jo vrne kot odgovor ter se zaključi:

```
<assign name="setError">
   <copy>
      <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
      <to variable="outputVariable" part="payload"
         query="/ns4:pattern/ns4:userTasks[1]/ns4:userTaskOutput/ns4:error"/>
   </copy>
</assign>
<invoke name="reportError" partnerLink="client"
```

```
      portType="client:PatternServiceCallback"
      operation="onResult"
      inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

Če do napake ne pride, se proces nadaljuje. Izbiri uporabniškega opravila sledi zanka s pogojem:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/ns4:pattern/ns4:userTasks
'))
```

Zanka se izvede za vsako uporabniško opravilo v vzorcu. Število uporabniških opravil v vzorcu je enako M. Na začetku iteracije proces kopira vrednost trenutnega uporabniškega opravila v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/ns4:pattern/ns4:userTasks[number(bpws:getVariableData('counter'))]"/
>
   <to variable="request" part="payload"
      query="/ns4:userTask"/>
</copy>
```

Sledi aktivnost invokeTask, ki kliče ustrezno uporabniško opravilo:

```
<invoke name="invokeTask" partnerLink="TaskService"
   portType="ns1:TaskService" operation="initiate"
   inputVariable="request"/>
```

S tem korakom se iteracija zaključi. Ko proces izvede prvo zanko, sledi druga zanka s pogojem:

```
bpws:getVariableData('i') <= ceiling((bpws:getVariableData('counter')-1)  * 0.5)
```

Ta zanka se izvede N-krat. V zgornjem primeru je N = M * 0.5. Število N lahko tudi spremenimo ali pa ga določimo dinamično z vrednostjo vhoda v proces. V tej zanki proces v aktivnosti receiveOutput v vsaki iteraciji sprejme en odgovor klicanih opravil:

```
<receive name="receiveOutput" partnerLink="TaskService"
    portType="services:TaskServiceCallback"
    operation="onResult" variable="response"
    createInstance="no"/>
```

Na koncu vsake iteracije prejet odgovor proces kopira iz spremenljivke response v izhodno spremenljivko outputVariable. Do tega dela je proces enak procesu NizmedM. Ko pa se zanka zaključi, proces v switch stavku določi katero nadzorniško opravilo bo klical:

```
<case condition="bpws:getVariableData('inputVariable','payload',
'/ns4:pattern/ns4:supervisionTask') = 'preverjanje'">
   <bpelx:annotation>
      <bpelx:pattern>Preverjanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setPreverjanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">

<Address>http://domaci:8888/orabpel/default/Preverjanje</Address>
               <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Preverjanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="SupervisorService"/>
      </copy>
   </assign>
</case>
```

Naveden je case stavek za primer, ko je izbrano uporabniško opravilo preverjanje. Proces kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo SupervisorService. Če pri izbiri nadzorniškega opravila pride do napake, proces napako zapiše v izhodno spremenljivko outputVariable, jo vrne kot odgovor in se zaključi kot pri izbiri uporabniškega opravila.
Če ne pride do napake, proces kopira vrednost izhodne spremenljivke outputVariable, ki vsebuje zahteve in odgovore vseh uporabniških opravil, v spremenljivko supervisorRequest, ki je vhodna spremenljivka aktivnosti invokeSupervisor:

```
<copy>
   <from variable="outputVariable" part="payload"
      query="/ns4:pattern"/>
   <to variable="supervisorRequest" part="payload"
      query="/ns4:pattern"/>
</copy>
```

V nadaljevanju sledita aktivnosti invokeSupervisor ter receiveSupervision. Aktivnost invokeSupervisor kliče ustrezno nadzorniško opravilo, aktivnost

receiveSupervision pa dobi odgovor nadzornika v spremenljivki supervisorResponse:

```
<invoke name="invokeSupervisor" partnerLink="SupervisorService"
    portType="ns1:SupervisorService" operation="initiate"
    inputVariable="supervisorRequest"/>
<receive name="receiveSupervision" partnerLink="SupervisorService"
   portType="ns1:SupervisorServiceCallback" operation="onResult"
   variable="supervisorResponse" createInstance="no"/>
```

Na koncu proces v odgovor doda še odgovor nadzornika, torej kopira vrednost spremenljivke supervisorResponse v izhodno spremenljivko outputVariable:

```
<copy>
  <from variable="supervisorResponse" part="payload"
      query="/ns4:pattern"/>
  <to variable="outputVariable" part="payload"
     query="/ns4:pattern"/>
</copy>
```

Spremenljivka outputVariable je tipa sporočilo PatternServiceResponseMessage. Ta tip je definiran v datoteki ZlitjeEnakih.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd. Proces vrne odgovor Krovnemu procesu v spremenljivki outputVariable.

5.4.6 Vzporedje

Krovni proces na prvem nivoju kliče spletno storitev Vzporedje na drugem nivoju, ko je izbran vzorec vzporedje. Proces v spletni storitvi Vzporedje dobi kot vhod spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Tip je definiran v datoteki Vzporedje.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces najprej kopira vrednost vhodne spremenljivke inputVariable v izhodno spremenljivko outputVariable:

```
<copy>
  <from variable="inputVariable" part="payload"
     query="/ns1:pattern"/>
  <to variable="outputVariable" part="payload"
     query="/ns1:pattern"/>
</copy>
```

Pri tem procesu so uporabnikom poslana različna uporabniška opravila, zato je potrebno za vsako uporabniško opravilo v vzorcu posebej določiti, kakšen klic bo proces izvedel. Proces gre zato v naslednjem koraku že v zanko s pogojem:

bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/ns1:pattern/ns1:userTasks'))

Proces nato na začetku vsake iteracije kopira ime trenutnega uporabniškega opravila iz vhodne spremenljivke v spremenljivko taskName:

```
<copy>
   <from variable="inputVariable" part="payload"
  query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]/
  ns1:taskName"/>
   <to variable="taskName"/>
</copy>
```

Na podlagi spremenljivke taskName potem v switch stavku določi katero uporabniško opravilo bo klical:

```
<case condition="bpws:getVariableData('taskName') = 'glasovanje'">
   <bpelx:annotation>
      <bpelx:pattern>Glasovanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setGlasovanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">

<Address>http://domaci:8888/orabpel/default/Glasovanje</Address>
               <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Glasovanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="TaskService"/>
      </copy>
   </assign>
</case>
```

Naveden je case stavek za primer, ko je izbrano uporabniško opravilo glasovanje. Proces kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo TaskService. Če pri izbiri uporabniškega opravila pride do napake, proces zapiše napako v izhodno spremenljivko outputVariable, jo vrne kot odgovor Krovnemu procesu in se zaključi:

```
<assign name="setError">
   <copy>
      <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
```

```
      <to variable="outputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]/
       ns1:userTaskOutput/ns1:error"/>
   </copy>
</assign>
<invoke name="reportError" partnerLink="client"
      inputVariable="outputVariable"
      portType="client:PatternServiceCallback"
      operation="onResult"/>
<terminate name="Terminate"/>
```

Če ne pride do napake, se proces nadaljuje. V naslednji aktivnosti kopira vrednost trenutnega uporabniškega opravila v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]"/
>
   <to variable="request" part="payload"
      query="/ns1:userTask"/>
</copy>
```

Naslednja aktivnost je invokeTask, ki kliče ustrezno uporabniško opravilo:

```
<invoke name="invokeTask" partnerLink="TaskService"
   portType="ns2:TaskService" operation="initiate"
   inputVariable="request"/>
```

Na tem mestu se iteracija zaključi in prvi zanki sledi druga zanka s pogojem:

```
bpws:getVariableData('i') < bpws:getVariableData('counter')
```

Ta zanka se izvede tolikokrat kot prva zanka. V tej zanki proces v vsaki iteraciji v aktivnosti receiveOutput sprejme en odgovor klicanih opravil:

```
<receive name="receiveOutput" partnerLink="TaskService"
    portType="ns2:TaskServiceCallback" operation="onResult"
    variable="response" createInstance="no"/>
```

Na koncu vsake iteracije druge zanke proces kopira prejet odgovor iz spremenljivke response v izhodno spremenljivko outputVariable. Ko proces zaključi z izvajanjem druge zanke, pošlje odgovor Krovnemu procesu v spremenljivki outputVariable. Spremenljivka outputVariable je tipa sporočilo PatternServiceResponseMessage, ki je definiran v datoteki Vzporedje.wsdl in ima

podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.

5.4.7 Zlitje različnih

Proces Krovni proces na prvem nivoju kliče spletno storitev ZlitjeRazlicnih na drugem nivoju, ko je izbran vzorec zlitje različnih. Proces v spletni storitvi ZlitjeRazlicnih dobi kot vhod spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Tip je definiran v datoteki ZlitjeRazlicnih.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces najprej kopira vrednost vhodne spremenljivke inputVariable v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="inputVariable" part="payload"
      query="/ns1:pattern"/>
   <to variable="outputVariable" part="payload"
      query="/ns1:pattern"/>
</copy>
```

Tudi pri tem procesu so uporabnikom poslana različna uporabniška opravila, zato je potrebno za vsako uporabniško opravilo v vzorcu posebej določiti, kakšen klic bo proces izvedel. Proces gre zato v naslednjem koraku že v zanko s pogojem:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/ns1:pattern/ns1:userTasks
'))
```

Proces nato na začetku vsake iteracije kopira ime trenutnega uporabniškega opravila iz vhodne spremenljivke v spremenljivko taskName:

```
<copy>
   <from variable="inputVariable" part="payload"
  query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]/
  ns1:taskName"/>
   <to variable="taskName"/>
</copy>
```

Na podlagi spremenljivke taskName potem v switch stavku določi katero uporabniško opravilo bo klical:

```
<case condition="bpws:getVariableData('taskName') = 'glasovanje'">
   <bpelx:annotation>
      <bpelx:pattern>Glasovanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setGlasovanje">
```

```
      <copy>
        <from>
          <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">

<Address>http://domaci:8888/orabpel/default/Glasovanje</Address>
            <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Glasovanje</ServiceName>
          </EndpointReference>
        </from>
        <to partnerLink="TaskService"/>
      </copy>
    </assign>
</case>
```

Naveden je case stavek za primer, ko je izbrano uporabniško opravilo glasovanje. Proces kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo TaskService. Če pri izbiri uporabniškega opravila pride do napake, proces zapiše napako v izhodno spremenljivko outputVariable, jo vrne kot odgovor Krovnemu procesu in se zaključi:

```
<assign name="setError">
   <copy>
      <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
      <to variable="outputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]/
       ns1:userTaskOutput/ns1:error"/>
   </copy>
</assign>
<invoke name="reportError" partnerLink="client"
      inputVariable="outputVariable"
      portType="client:PatternServiceCallback"
      operation="onResult"/>
<terminate name="Terminate"/>
```

Če ne pride do napake, se proces nadaljuje. V naslednji aktivnosti kopira vrednost trenutnega uporabniškega opravila v spremenljivko request, ki je vhodna spremenljivka aktivnosti invokeTask:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]"/
>
```

```
   <to variable="request" part="payload"
      query="/ns1:userTask"/>
</copy>
```

Naslednja aktivnost je invokeTask, ki kliče ustrezno uporabniško opravilo:

```
<invoke name="invokeTask" partnerLink="TaskService"
   portType="ns2:TaskService" operation="initiate"
   inputVariable="request"/>
```

Na tem mestu se iteracija zaključi in prvi zanki sledi druga zanka s pogojem:

```
bpws:getVariableData('i') < bpws:getVariableData('counter')
```

Ta zanka se izvede tolikokrat kot prva zanka. V tej zanki proces v vsaki iteraciji v aktivnosti receiveOutput sprejme en odgovor klicanih opravil:

```
<receive name="receiveOutput" partnerLink="TaskService"
    portType="ns2:TaskServiceCallback" operation="onResult"
    variable="response" createInstance="no"/>
```

Na koncu vsake iteracije druge zanke proces kopira prejet odgovor iz spremenljivke response v izhodno spremenljivko outputVariable. Do tega dela je proces enak procesu Vzporedje. Ko pa se zanka zaključi, proces ZlitjeRazlicnih v switch stavku določi katero nadzorniško opravilo bo klical:

```
<case condition="bpws:getVariableData('inputVariable','payload',
'/ns4:pattern/ns4:supervisionTask') = 'preverjanje'">
   <bpelx:annotation>
      <bpelx:pattern>Preverjanje</bpelx:pattern>
   </bpelx:annotation>
   <assign name="setPreverjanje">
      <copy>
         <from>
            <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">

<Address>http://domaci:8888/orabpel/default/Preverjanje</Address>
               <ServiceName
xmlns:ns1="http://services.otn.com">ns1:Preverjanje</ServiceName>
            </EndpointReference>
         </from>
         <to partnerLink="SupervisorService"/>
      </copy>
   </assign>
</case>
```

Naveden je case stavek za primer, ko je izbrano uporabniško opravilo preverjanje. Proces kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo SupervisorService. Če pri izbiri nadzorniškega opravila pride do napake, proces napako zapiše v izhodno spremenljivko outputVariable, jo vrne kot odgovor in se zaključi kot pri izbiri uporabniškega opravila.
Če ne pride do napake, proces kopira vrednost izhodne spremenljivke outputVariable, ki vsebuje zahteve in odgovore vseh uporabniških opravil, v spremenljivko supervisorRequest, ki je vhodna spremenljivka aktivnosti invokeSupervisor:

```
<copy>
   <from variable="outputVariable" part="payload"
        query="/ns4:pattern"/>
   <to variable="supervisorRequest" part="payload"
      query="/ns4:pattern"/>
</copy>
```

V nadaljevanju sledita aktivnosti invokeSupervisor ter receiveSupervision. Aktivnost invokeSupervisor kliče ustrezno nadzorniško opravilo, aktivnost receiveSupervision pa dobi odgovor nadzornika v spremenljivki supervisorResponse:

```
<invoke name="invokeSupervisor" partnerLink="SupervisorService"
     portType="ns1:SupervisorService" operation="initiate"
     inputVariable="supervisorRequest"/>
<receive name="receiveSupervision" partnerLink="SupervisorService"
    portType="ns1:SupervisorServiceCallback" operation="onResult"
    variable="supervisorResponse" createInstance="no"/>
```

Na koncu proces v odgovor doda še odgovor nadzornika, torej kopira vrednost spremenljivke supervisorResponse v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="supervisorResponse" part="payload"
        query="/ns4:pattern"/>
   <to variable="outputVariable" part="payload"
      query="/ns4:pattern"/>
</copy>
```

Spremenljivka outputVariable je tipa sporočilo PatternServiceResponseMessage. Ta tip je definiran v datoteki ZlitjeRazlicnih.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd. Proces vrne odgovor Krovnemu procesu v spremenljivki outputVariable.

5.4.8 Ad hoc

Proces Krovni proces na prvem nivoju kliče spletno storitev AdHoc na drugem nivoju, ko je izbran vzorec ad hoc. Proces v spletni storitvi AdHoc dobi kot vhod spremenljivko inputVariable, ki je tipa sporočilo PatternServiceRequestMessage. Tip je definiran v datoteki AdHoc.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.
Proces najprej kopira vrednost vhodne spremenljivke inputVariable v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="inputVariable" part="payload"
       query="/ns1:pattern"/>
   <to variable="outputVariable" part="payload"
      query="/ns1:pattern"/>
</copy>
```

V naslednjem koraku proces določi prvega uporabnika, ki bo opravil uporabniško opravilo ter določil naslednjega uporabnika. Prvi uporabnik je določen v vhodni spremenljivki:

```
<copy>
   <from variable="inputVariable" part="payload"
       query="/ns1:pattern/ns1:userTasks[1]/ns1:userTaskInput/ns1:user"/>
   <to variable="userName"/>
</copy>
```

Ker je pri tem vzorcu vsako uporabniško opravilo drugačno, je treba določati uporabniška opravila v zanki. Prav tako so poslana različnim uporabnikom, ki so določeni v prejšnjem koraku, zato je v zanki treba določati tudi naslednjega uporabnika. Proces začne z izvajanjem zanke s pogojem:

```
bpws:getVariableData('counter') <=
count(bpws:getVariableData('inputVariable','payload','/ns1:pattern/ns1:userTasks
'))
```

Na začetku vsake iteracije proces določi katero uporabniško opravilo mora v tej iteraciji klicati in ime opravila kopira v spremenljivko taskName:

```
<copy>
   <from variable="inputVariable" part="payload"
query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]/ns
1:taskName"/>
   <to variable="taskName"/>
</copy>
```

Na podlagi vrednosti spremenljivke taskName proces v switch stavku določi katero uporabniško opravilo bo klical:

```
<case condition="bpws:getVariableData('taskName') = 'odlocanje'">
  <bpelx:annotation>
    <bpelx:pattern>Odlocanje</bpelx:pattern>
  </bpelx:annotation>
  <assign name="setOdlocanje">
    <copy>
      <from>
        <EndpointReference
xmlns="http://schemas.xmlsoap.org/ws/2003/03/addressing">
          <Address>http://domaci:8888/orabpel/default/Odlocanje</Address>
          <ServiceName xmlns:ns1="http://services.otn.com">
            ns1:Odlocanje</ServiceName>
        </EndpointReference>
      </from>
      <to partnerLink="TaskService"/>
    </copy>
  </assign>
</case>
```

Naveden je case stavek za primer, ko je klicano uporabniško opravilo odločanje. Proces kopira endpoint referenco, ki vsebuje naslov in ime storitve v partnersko povezavo TaskService. Če pri izbiri uporabniškega opravila pride do napake, proces zapiše napako v izhodno spremenljivko outputVariable, jo vrne kot odgovor Krovnemu procesu in se zaključi:

```
<assign name="setError">
  <copy>
    <from expression="'Napaka pri izbiri uporabniskega opravila'"/>
    <to variable="outputVariable" part="payload"
                    query="/ns1:pattern/ns1:userTasks[number
(bpws:getVariableData('counter'))]/ns1:userTaskOutput/ns1:error"/>
  </copy>
</assign>
<invoke name="reportError" partnerLink="client"
    portType="client:PatternServiceCallback"
    operation="onResult"
    inputVariable="outputVariable"/>
<terminate name="Terminate"/>
```

Če ne pride do napake, se proces nadaljuje. V naslednji aktivnosti kopira vrednost trenutnega uporabniškega opravila v spremenljivko request, ki je vhodna

spremenljivka aktivnosti invokeTask. Poleg tega v spremenljivko request zapiše še uporabnika, ki bo opravil to uporabniško opravilo:

```
<copy>
   <from variable="inputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('counter'))]"/
>
   <to variable="request" part="payload"
      query="/ns1:userTask"/>
</copy>
<copy>
   <from variable="userName"/>
   <to variable="request" part="payload"
      query="/ns1:userTask/ns1:userTaskInput/ns1:user"/>
</copy>
```

V naslednjem koraku sledita aktivnosti invokeTask in receiveOutput. Aktivnost invokeTask kliče ustrezno uporabniško opravilo, aktivnost receiveOutput pa čaka na odgovor:

```
<invoke name="invokeTask" partnerLink="TaskService"
      portType="ns2:TaskService" operation="initiate"
      inputVariable="request"/>
<receive name="receiveOutput" partnerLink="TaskService"
       portType="ns2:TaskServiceCallback" operation="onResult"
       variable="response" createInstance="no"/>
```

Ko dobi odgovor, ga iz spremenljivke response kopira v izhodno spremenljivko outputVariable:

```
<copy>
   <from variable="response" part="payload"
        query="/ns1:userTask"/>
   <to variable="outputVariable" part="payload"

query="/ns1:pattern/ns1:userTasks[number(bpws:getVariableData('response',
     'payload','/ns1:userTask/ns1:taskNo'))]"/>
</copy>
```

V spremenljivki response proces dobi tudi ime naslednjega uporabnika, ki ga kopira v spremenljivko userName:

```
<copy>
   <from variable="response" part="payload"
        query="/ns1:userTask/ns1:userTaskOutput/ns1:nextUser"/>
```

```
    <to variable="userName"/>
</copy>
```

Na tem mestu se iteracija zaključi. Ko proces zaključi z izvajanjem zanke, vrne odgovor Krovnemu procesu v obliki spremenljivke outputVariable. Spremenljivka outputVariable je tipa sporočilo PatternServiceResponseMessage, ki je definiran v datoteki Vzporedje.wsdl in ima podatkovno strukturo pattern, ki je definirana v podatkovni shemi UserTaskTypes.xsd.

5.4.9 Nadaljevanje

Vzorec nadaljevanje oziroma kombinacija različnih do sedaj naštetih vzorcev, je realizirana v Krovnem procesu. Krovni proces kot vhod dobi inputVariable, ki je tipa sporočilo KrovniProcesRequestMessage. Ta tip je definiran v datoteki KrovniProces.wsdl in ima podatkovno strukturo patterns. Ta struktura je sestavljena iz ene ali več podatkovnih struktur pattern, ki je vhod za poljubni vzorec. Krovni proces v zanki kliče več vzorcev zaporedno in s tem dobimo nadaljevanje oziroma kombinacijo različnih vzorcev.

5.4.10 WSDL datoteke

WSDL datoteke spletnih storitev so sestavljene iz petih delov. Vsi deli spadajo pod korensko oznako <definitions>. V tej oznaki so navedena imena imenskih prostorov ter ime procesa. V prvem delu so definicije tipov. Tukaj so definirani vsi tipi, ki jih uporablja storitev. Tipe lahko neposredno definiramo v tem delu, lahko pa uvozimo tipe iz XML Shema dokumentov. V drugem delu so definirana sporočila, ki jih spletna storitev dobi kot vhod ter pošlje kot odgovor. V tem delu lahko del sporočila določimo kot tip, ki je definiran v prvem delu. V tretjem delu so definicije tipov vrat, ki jih uporablja spletna storitev oziroma BPEL proces. Tukaj so definirane operacije, ki jih nudi spletna storitev. V definiciji vrat sta določena tudi vhodno sporočilo in odgovor teh operacij, ki sta definirana v prejšnjem delu. V četrtem delu je definicija povezave, na koncu pa je še definicija storitve.

5.4.10.1 WSDL datoteka Krovnega procesa

Na začetku datoteke je korenska oznaka <definitions> v kateri so vse ostale definicije. Prva je definicija tipov:

```
<types>
        <schema xmlns="http://www.w3.org/2001/XMLSchema">
                <import namespace="http://www.autotask.com/ns/autotask"
schemaLocation="UserTaskTypes.xsd" />
        </schema>
</types>
```

Podatkovni tipi so uvoženi iz datoteke UserTaskTypes.xsd. Sledita definiciji sporočil – zahteve in odgovora:

```
<message name="KrovniProcesRequestMessage">
        <part name="payload" element="ns1:patterns" />
</message>
<message name="KrovniProcesResponseMessage">
        <part name="payload" element="ns1:patterns" />
</message>
```

Zadnji del v abstraktnem delu so definicije tipov vrat:

```
<portType name="KrovniProces">
        <operation name="initiate">
                <input message="client:KrovniProcesRequestMessage"/>
        </operation>
</portType>
```

V tipu vrat KrovniProces je navedena operacija 'initiate', ki kot vhod uporablja sporočilo KrovniProcesRequestMessage. Tip vrat KrovniProcesCallback je namenjen za odgovor asinhronih klicev procesa:

```
<portType name="KrovniProcesCallback">
        <operation name="onResult">
                <input message="client:KrovniProcesResponseMessage"/>
        </operation>
</portType>
```

Na koncu je naveden tip partnerske povezave, ki povezuje tipa vrat KrovniProces in KrovniProcesCallback v asinhronsko komunikacijo:

```
<plnk:partnerLinkType name="KrovniProces">
        <plnk:role name="KrovniProcesProvider">
                <plnk:portType name="client:KrovniProces"/>
        </plnk:role>
        <plnk:role name="KrovniProcesRequester">
                <plnk:portType name="client:KrovniProcesCallback"/>
        </plnk:role>
</plnk:partnerLinkType>
```

5.4.10.2 Splošna WSDL datoteka spletnih storitev vzorcev

WSDL datoteka spletnih storitev vzorcev PatternService vsebuje poleg elementov, ki so našteti v WSDL datoteki KrovniProces tudi elemente povezave in storitev. Ta dva elementa tvorita konkretni del WSDL datoteke. Elementa povezava določata fizično povezavo:

```
<binding name="PatternServiceBinding" type="tns:PatternService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="initiate">
    <soap:operation soapAction="initiate" style="document"/>
    <input>
      <soap:header message="tns:MessageIDHeader" part="MessageID"
use="literal" required="false"/>
      <soap:header message="tns:ReplyToHeader" part="ReplyTo"
use="literal" required="false"/>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>

<binding name="PatternServiceCallbackBinding"
type="tns:PatternServiceCallback">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="onResult">
    <soap:operation soapAction="onResult" style="document"/>
    <input>
      <soap:header message="tns:RelatesToHeader" part="RelatesTo"
use="literal" required="false"/>
      <soap:body use="literal"/>
    </input>
  </operation>
</binding>
```

Sledijo elementi storitve. Na začetku je PatternServiceCallbackService, sledijo pa definicije vseh storitev, ki jih Krovni proces lahko kliče. To so spletne storitve, ki implementirajo vzorce. Prikazan je primer spletne storitve Sekvenca:

```
<service name="PatternServiceCallbackService">
  <port name="PatternServiceCallbackPort"
binding="tns:PatternServiceCallbackBinding">
    <soap:address location="http://openuri.org"/>
  </port>
</service>

<service name="Sekvenca">
  <port name="TaskServicePort" binding="tns:PatternServiceBinding">
    <soap:address location="http://domaci:8888/orabpel/default/Sekvenca"/>
```

```
    </port>
  </service>
```

Vsaka spletna storitev vzorca ima tudi svojo WSDL datoteko, ki je sestavljena podobno kot WSDL datoteka Krovnega procesa, zato niso dodatno opisane.

6 Zaključek

6.1 Diskusija

V okviru diplomskega dela so identificirani ter implementirani vzorci sestavljenih uporabniških opravil. Implementacija je realizirana večnivojsko, za komunikacijo med posameznimi nivoji so uporabljeni klici spletnih storitev. Ti klici so realizirani dinamično. Na prvem nivoju so dinamični klici uporabljeni zato, da vedno kličemo eno storitev z enakim vmesnikom in na enakem naslovu, namesto da bi bilo potrebno za vsak vzorec vnašati ime storitve v kateri je realiziran ter naslov storitve. Prednost je tudi ta, da se pri dodajanju novih vzorcev sam klic krovne storitve ne spremeni.

Tudi procesi, ki v spletnih storitvah predstavljajo vzorce, dinamično kličejo uporabniška opravila. Dinamični klici uporabniških opravil so smiselni, če imamo veliko različnih uporabniških opravil, saj pri vzorcih pride do klicev različnih kombinacij uporabniških opravil iz te množice.

Če pa bi tukaj šlo vedno za enako kombinacijo uporabniških opravil ali celo za klic enega samega uporabniškega opravila, bi bili vsekakor bolj primerni statični klici, ki so hitrejši in enostavnejši. Pri statičnih klicih namreč ni treba izvajati dodatnih operacij, ki vključujejo izbiro ustrezne storitve in pisanje naslova izbrane storitve v endpoint reference.

Semantični splet je splet na katerem so računalniki sposobni razumeti podatke, jih iskati in uporabljati. Za zdaj razvoj semantičnega spleta še ni na taki stopnji, da bi lahko dinamično klical poljubno spletno storitev, saj računalnik trenutno ni sposoben razumeti, kaj določena storitev ponuja ter kaj pomenijo posamezni podatki. Z razvojem semantičnega spleta pa bo prišlo do tega, da bodo računalniki znali sami poiskati ustrezno spletno storitev. Ko bo računalnik našel pravo spletno storitev, bo tudi vedel kakšne vhodne podatke zahteva ter bo znal uporabiti podatke, ki jih bo dobil kot odgovor. Zato bo z razvojem semantičnega spleta koncept vzorcev in dinamičnih klicev še bolj primeren za uporabo, saj dinamični klici ne bodo omejeni le na spletne storitve s povsem enakimi vmesniki, ampak bo pomembno kaj storitev ponuja. V zadnjih letih se precej razvija področje algoritmov za odkrivanje semantičnih spletnih storitev ter določanje ujemanja storitev.

V diplomskem delu sem identificiral ter implementiral osem vzorcev sestavljenih uporabniških opravil. Prvih sedem vzorcev je zelo osnovnih in splošnih, osmi vzorec pa je poljubna zaporedna kombinacija prvih sedmih. S tem sem po mojem mnenju pokril precejšen del uporabe vzorcev sestavljenih uporabniških opravil, če upoštevamo tudi pogostost uporabe vzorcev. Ti vzorci se namreč najpogosteje uporabljajo, z zadnjim vzorcem pa lahko dobimo ogromno število bolj specifičnih vzorcev. Obstaja še veliko drugih vzorcev, ki pa so še bolj specifični in se zato redkeje uporabljajo.

Pri identificiranih vzorcih je bilo potrebno upoštevati da morajo biti enostavno ponovno uporabljivi. Zato je nesmiselna implementacija velikega števila specifičnih

vzorcev, saj se zelo redko uporabljajo in niso v tolikšni meri ponovno uporabljivi. Poleg tega se bolj specifični vzorci pogosto uporabljajo z enakimi uporabniškimi opravili in jih je zato bolj smiselno implementirati s statičnimi klici uporabniških opravil in ne z dinamičnimi klici.

7 Viri

[1] W.M.P. van der Aalst, D. Edmond, A.H.M ter Hofstede, N. Russel, “Workflow Resource Patterns”, BETA Working Paper Series, Eindhoven, 2004, dostopno na: http://www.workflowpatterns.com/documentation/documents/Resource%20Patterns%20BETA%20TR.pdf

[2] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. Konig, F. Leymann, R. Muller, G. Pfau, K. Plosser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, M. Zeller, “Web Services Human Task (WS-HumanTask), Version 1.0 ”, junij 2007, dostopno na: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf

[3] A. Agrawal, M. Amend, M. Das, M. Ford, C. Keller, M. Kloppmann, D. Konig, F. Leymann, R. Muller, G. Pfau, K. Plosser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, M. Zeller, “WS-BPEL Extension for People (BPEL4People), Version 1.0 ”, junij 2007, dostopno na: http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf

[4] S. Alter, Information Systems: Foundation of e-business, Prentice Hall College, 2001.

[5] C. Barreto, V. Bullard, T. Erl, J. Evdemon, D. Jordan, K. Kand, D. Konig, S. Moser, R. Stout, R. Ten-Hove, I. Trickovic, D. van der Rijn, A. Yiu, “Web Services Business Process Execution Language Version 2.0”, maj 2007, dostopno na:http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm

[6] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Simeon, “ XML Path Language (XPath) Version 1.0”, januar 2007, dostopno na: http://www.w3.org/TR/xpath20/

[7] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, S. Winkler, “Web Services Addressing (WS-Addressing)”, avgust 2004, dostopno na: http://www.w3.org/Submission/ws-addressing/

[8] D. Bunting, J. Durand, K. Iwasa, S. Kunisetty, M. Peel, T. Rutt, “Web Services Reliable Messaging TC, WS-Reliability 1.1”, november 2004, dostopno na: http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

[9] S. Carey, SOA Best Practices: The BPEL Cookbook, Making BPEL Processes Dynamic, dostopno na: http://www.oracle.com/technology/pub/articles/bpel_cookbook/carey.html

[10] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, New Jersey: Prentice Hall PTR, 2005.

[11] M. Feingold, R. Jeyaraman, E. Newcomer, I. Robinson, "Web Services Coordination (WS-Coordination) Version 1.1", julij 2007, dostopno na: http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.1-spec/wstx-wscoor-1.1-spec.html

[12] T. Freund, M. Little, E. Newcomer, I. Robinson, "Web Services Business Activity (WS-BusinessActivity) Version 1.2", februar 2009, dostopno na: http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html

[13] S. Gao, C.M. Sperberg-McQueen, H.S. Thompson, "XML Shema Definition Language (XSD 1.1 Part1: Structures", april 2009, dostopno na: http://www.w3.org/TR/xmlschema11-1/

[14] M. Gudgin, M. Hadley, T. Rogers, "Web Services Addressing 1.0 – Core. W3C", maj 2006, dostopno na: http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/

[15] P. Hallam-Baker, C. Kaler, K. Lawrence, R. Monzillo, A. Nadalin, "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)", februar 2006, dostopno na: http://docs.oasis-open.org/wss/v1.1/wss-v1.1-spec-os-SOAPMessageSecurity.pdf

[16] M.B. Jurič, B. Mathew, P. Sarand, Business Process Execution Language for Web Services, Second Edition, Birmingham,UK: Packt Publishing, 2006.

[17] M.B. Jurič, K. Pant, Business Process Driven SOA using BPMN and BPEL: From Business Process Modeling to Orchestration and Service Oriented Architecture, Birmingham,UK: Packt Publishing, 2008.

[18] F. Leymann, C. Pautasso, O.Zimmermann, "RESTful Web Services vs. "Big" Web Services: Making the right architectural decision", v zborniku International World Wide Web Conference, Peking, Kitajska, april 2008, str. 805-814.

[19] M. Little, E. Newcomer, I. Robinson, A. Wilkinson, "Web Services Atomic Transaction (WS-AtomicTransaction) Version 1.2", februar 2009, dostopno na: http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec-os/wstx-wsat-1.2-spec-os.html

[20] M. Wright, "Web Services: yesterday's hype or tomorrow's promise", 2005, dostopno na: http://ausweb.scu.edu.au/aw06/papers/refereed/wright/paper.html

[21] Wikipedia, Business Process, dostopno na: http://en.wikipedia.org/wiki/Business_process

[22] Wikipedia, Business Process Execution Language, dostopno na: http://en.wikipedia.org/wiki/Bpel

[23] Wikipedia, Business Process Modeling Notation, dostopno na: http://en.wikipedia.org/wiki/BPMN

[24] Wikipedia, Service-oriented architecture dostopno na: http://en.wikipedia.org/wiki/Service_oriented_architecture

[25] Wikipedia, Universal Description Discovery and Integration, dostopno na: http://en.wikipedia.org/wiki/UDDI

[26] Wikipedia, Web service, dostopno na: http://en.wikipedia.org/wiki/Web_service

[27] Wikipedia, Web Services Description Language, dostopno na: http://en.wikipedia.org/wiki/Web_Services_Description_Language

[28] Wikipedia, WS-Addressing, dostopno na: http://en.wikipedia.org/wiki/WS-Addressing

[29] Wikipedia, WS-Coordination, dostopno na: http://en.wikipedia.org/wiki/WS-Coordination

[30] Wikipedia, WS-Reliability, dostopno na: http://en.wikipedia.org/wiki/WS-Reliability

[31] Wikipedia, WS-Security, dostopno na: http://en.wikipedia.org/wiki/WS-Security

[32] Wikipedia, WS-Transaction, dostopno na: http://en.wikipedia.org/wiki/WS-Transaction

[33] (2005) Microsoftove novice: Storitveno usmerjena arhitektura je prihodnost. Dostopno na: http://www.microsoft.com/slovenija/novinarji/novice/microsoft/2005/05/2005051901.mspx

[34] (2009) Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite, 11g Release 1 (11.1.1). Dostopno na: http://download.oracle.com/docs/cd/E12839_01/integration.1111/e10224/toc.htm

[35] (2006) Oracle BPEL Process Manager Developer's Guide, 10g Release 2 (10.1.2), pogl. 16, dostopno na: http://download-west.oracle.com/docs/cd/B14099_19/integrate.1012/b14448/workflow.htm