Debreceni Egyetem

Informatikai Kar

Szolgáltatásorientált szemlélet a programozásban

Témavezető:

Dr. Juhász István

egyetemi adjunktus

Készítette:

Kovács György

programtervező-matematikus

Debrecen

2007

Tartalomjegyzék

1. BEVEZETÉS 5

2. A SOA ELŐTT 6

2.1. Eljárásorientált programozás 6

2.2. Strukturált programtervezés 8

2.3. Objektumorientált programozás 9

2.4. Objektumorientált programtervezés 10

2.5. Komponensorientált programozás 11

3. A TECHNOLÓGIA - WEBSZOLGÁLTATÁSOK 14

3.1. SOAP 15

3.1.1. SOAP boríték specifikáció 16

3.1.2. Adatkódolási szabályok 19

3.1.3. RPC szabályok 21

3.1.4. SOAP implementációk 21

3.2. WSDL 22

3.3. UDDI 27

3.3.1. Adatmodell 28

3.3.2. UDDI API 33

3.3.3. UDDI UBR 33

3.4. ESB 33

3.5. BPEL4WS 36

3.6. Összefoglalás 38

4. KONCEPCIÓK 40

5. PROGRAMTERVEZÉS 43

5.1. Szolgáltatások azonosítása 43

6. SZÁMÍTÁSELMÉLETI HÁTTÉR 51

6.1. Service Centered Calculus 51

6.2. Service Oriented Computing Kernel 52

7. PÉLDA **56**

7.1. A feladat leírása 56

7.2. WSrendeles szolgáltatás Visual Studio fejlesztőkörnyezettel C# nyelven . . 56

7.3. WSholjar szolgáltatás létrehozása NetBeans 5.5 környezetben Java nyelven 59

8. ÖSSZEFOGLALÁS **63**

Köszönetnyilvánítás

Szeretnék köszönetet mondani Dr. Juhász István tanár Úrnak, aki elvállalta a szakdolgozatom témavezetését, és tanácsaival segítette annak elkészítését.

Szeretnék továbbá köszönetet mondani családomnak, akik minden körülményt biztosítottak ahhoz, hogy ezt a dolgozatot elkészíthessem.

1. BEVEZETÉS

Az utóbbi években divattá vált az üzleti szoftverekkel kapcsolatban szolgáltatásokról beszélni. A megrendelők webszolgáltatásokat akarnak, a szoftverfejlesző cégek webszolgáltatásokat készítenek, de hogy pontosan mit is takarnak a szolgáltatás, webszolgáltatás, szolgáltatásorientált programozás, szolgáltatásorientált architektúra kifejezések, arról kevés szó esik.

Legtöbben a szolgáltatásorientált architektúrát (SOA) webszolgáltatásokkal, és azok összekapcsolt rendszerével azonosítják. Ezzel szemben a webszolgáltatások csak egy megvalósítása a szolgáltatásorientált architektúrához kapcsolódó ajánlások egy részének. A szolgáltatásorientált architektúra sokkal több annál, mint jól definiált üzleti szolgáltatások készítése, publikálása (a nyilvánosság számára elérhetővé tétele), és felhasználása. Magában foglalja az új szemlélet módot, szoftver tervezési mintákat, módszertanokat, ezért talán sokkal találóbb lenne, ha egyszerűen csak szolgáltatásorientációnak neveznék.

Szeretném ebben a dolgozatban bemutatni a szolgáltatásorientált programozás kialakulásához vezető utat az üzleti szoftverkészítésban, a webszolgáltatások legfontosabb szabványait és megoldásait. Szeretném bemutatni a szolgáltatásorientált szemlélethez kötődő új programtervezési koncepciókat, a még fejlődő számításelméleti hátteret, majd egy példán keresztül bemutatni webszolgáltatások készítését C# nyelven Microsoft Visual Studio 2003 rendszerben és Java nyelven NetBeans 5.5 IDE segítségével.

2. A SOA ELŐTT

A kereskedelmi szoftverek létrehozásának céljára az imperatív programozási nyelvek, és ezáltal az imperatív programozási paradigma terjedt el, mivel ez állt a legközelebb az emberi gondolkodáshoz. Ez a megközelítés kiválóan modellezhető a matematikai/számításelméleti Turing-gépekkel, s így minden, a Turing-gépek értelmében algoritmusnak tekintett algoritmus megvalósítható az imperatív nyelvek segítségével. Kell lennie egy módosítható tárnak, amit a program használhat. Megváltoztatva a változók állapotát, vagyis azt, amit a tárban tárolunk, egy állapotváltozást érünk el a képzeletbeli automatában. A számítások ezután meghatározott utasítások egymásutáni végrehajtásával kivitelezhetők. (Lényegében ezek a Neumann-architektúra legfőbb jellemzői.)

Kezdetben, az első üzleti szoftverek megjelenésekor, a technológiai háttér, és a rendelkezésre álló programozási nyelvek és eszközök maihoz viszonyított szegényessége miatt nem beszélhetünk a programozási eszközökben manapság megszokott adat és funkcionális absztrakcióról. Bár az akkori szoftverek ugyanúgy üzleti problémákat oldottak meg, mint a mai megfelelőik, azok elkészítése különösebb, módszeres tervezés nélkül, a legelemibb programozási eszközök felhasználásával történt.

A kezdeti programozási nyelvek strukturálatlan programozást tettek csak lehetővé, ami azt jelenti, hogy a végrehajtható utasítások egymás után egyetlen blokkban következtek. Ezek a nyelvek (FORTRAN, ASSEMBLY) a vezérlési szerkezetek kialakításához a GOTO, vagy JUMP utasításokat használnak, amelyek bár némi gyorsaságot jelentenek a későbbi nyelvekben megjelent függvényhívásokhoz képest, azonban a kódot többnyire érthetetlenné és követhetetlenné teszik. Az ilyen un. spagettikódban nehéz a hibakeresés. Az utóbbi években megjelent programozási nyelvek nem is tartalmaznak ilyen eszközt.

2.1. Eljárásorientált programozás

Ha az imperatív programozást alprogramokkal, mint programozási eszközökkel egészítjük ki, azt procedurális (eljárásorientált) programozásnak nevezzük. Az eljárásorientált programozási nyelvek megjelenése óriási előre lépést jelentett. Az alprogramok, amelyeket

a különböző nyelvek különböző néven ismernek (rutin, szubrutin, metódus, függvény) nem azonosak a matematikai függvényekkel, (a matematikai függvényekhez hasonló alprogram eszközöket a funkcionális nyelvek használnak [LISP]) számítási lépések végrehajtható sorozatait tartalmazzák. Az alprogramoknak két típusa van: az eljárások a kód bármely pontján hívhatók, ahol végrehajtható utasítás állhat; a függvények a kód minden ponján és kifejezésekben is hívhatók. Az alprogramok használatával:

- megvalósítható a kód-újrafelhasználást
- a program menete könnyebben vezérelhető a GOTO, vagy JUMP utasításokkal szemben
- a kód struktúrált, ezáltal könnyen átlátható, megérthető

Saját típusok definiálásával adatabsztrakciót, saját függvények, eljárások definiálásával pedig procedurális absztrakciót lehet megvalósítani. Ezek segítségével már gyakorlatilag minden, az üzletmenethez hozzátartozó körülményt az emberi gondolkodással összeegyeztethető módon modellezni lehet. Az összetartozó adatokat egy egységként, összetett adattípusok segítségével (tömb, rekord) lehet kezelni, míg a rajtuk végezhető elemi és összetett műveleteket függvényekként lehet megvalósítani. Tehát az eljárás-orientált programozás lényege: az összetartozó elemi adatokat adatszerkezetekbe szervezzük, és ezeken az adatszerkezeteken műveleteket hajtunk végre. Az így elkészült szoftver működése a következő: a belépési pont egy kiemelt függvény, vagy eljárás, ami szekvenciálisan hajtja végre az utasításokat, amelyek újabb függvények és eljárások indítását jelenthetik. A probléma a nagyobb rendszerek fejlesztésekor adódott:

- Ez a modell nem túl jól skálázható.
- Ha megváltozik a tár egy, a szoftver több része által megosztva használt területe, nehéz kibogozni, hogy ez a változás pontosan mely részeit érinti a kódnak.
- Különböző adatstruktúrákon végezhető műveletek a kód legkülönbözőbb helyein fordultak elő. Ha módosítani kellett egy struktúrán, akkor nagyon sok helyen kellett módosítani a már működő kódot is. Ezeknek a helyeknek a megkeresése időigényes és nagy rendszereknél sziszifuszi munka volt.

Természetes igényként merült fel a kérdés, hogyan lehetne az összetartozó műveleteket és struktúrákat egy helyen kezelni. Ez kezdetben úgy történt, hogy önálló fordítási egységekbe, modulokba szervezték őket. A modulok külön láthatósági tartományt alkotnak. Kívülről csak azok az eszközök érhetők el, amelyeket úgy hoztunk létre. További problémát okozott, hogy a műveleteket nem biztos, hogy a megfelelő paraméterekkel, a megfelelő értelemben hívták meg, valamint a különböző struktúrák nem csak a megfelelő műveletekkel voltak módosíthatóak, hanem a memóriát közvetlenül elérve is.

2.2. Strukturált programtervezés

Az eljárás-orientált programozási nyelveken alapuló rendszerek tervezésének fő eszközei a strukturált módszertanok, ezek közül is az SSADM voltak. Az SSADM rendszer analizáló, és tervező módszertan. Az SSADM eljárás a következő modulokra bomlott:

- **megvalósíthatóság-elemzési modul**: költségek, források, várható haszon felmérése, és ezek alapján a kockázat megállapítása
- **követelmény-elemzési modul**: felhasználói követelmények felmérése
- **a jelenlegi helyzet vizsgálata**: az elemzők felmérik a jelenlegi működési környezetet, meghatározzák a jól működő dolgokat, dokumentálják a követelményeket, adatmodellek, és adatfolyam modellek készülnek
- **rendszerszervezési alternatívák**: a követelményeket kielégítő, de különböző jellegű és működésű alternatívákat kell felkínálni
- **követelmény specifikációs modul**: követelmények részletes meghatározása
- **logikai rendszerspecifikációs modul**: a cél, hogy olyan specifikáció álljon össze, amely alapján a fizikai tervezést és megvalósítást ki lehet adni szerződéses külső feleknek
- **rendszertechnikai alternatívák**: lehetőséget adnak a megvalósítási és üzemeltetési környezetak közti választásra.

- **logikai rendszertervezés**: pontosan specifikálni kell a feldolgozási folyamatokat. Leggyakrabban a Jackson strukturált programozás alapelveit és jelöléstechnikáját használja fel a módszer, kisebb kiegészítésekkel.
- **fizikai rendszertervezés**: a logikai rendszerspecifikációból és a technikai környezet leírásából kiindulva meg kell határozni az adatok és folyamatok fizikai részleteit. Itt végződik az SSADM módszer. A fizikai megvalósítás már nem tartozik ide.

2.3. Objektumorientált programozás

Az eljárás-orientált nyelvekben a modulok részben oldották meg azt a problémát, hogy az alprogramoktól nagyobb programozási egységekből, tágabb absztrakciós eszközök mentén építsük fel a szoftver rendszereket. Ennek a problémának a teljes megoldására született meg az absztrakt adattípus, mint programozási eszköz. Legfontosabb jellemzői:

- megvalósítja a procedurális és adatabsztrakciót
- elrejti állapotát a külvilág elől, az csak jól definiált metódusokon keresztül módosítható
- az adattípus kiterjeszthetőségével úgymond fejlődési lehetőséget biztosít,
- levetővé teszi hierarchiák kialakítását
- a polimorfizmus megvalósításával lehetővé teszi a program futása során a hierarchiában leszármazottai tulajdonságainak dinamikus felvételét.

Az absztrakt adattípus először a Simula 67-ben jelent meg osztályként, teljesen új absztrakciós szemléletmódot téve lehetővé: A világot az emberi képzeletnek sokkal jobban megfelelő osztályok, és objektumok rendszerében szemlélte. Ez teljesen összhangban van az ember világról alkotott képének, ahogy az a beszélt nyelvvel való párhuzam alapján is megnyilvánul. Az általunk használt fogalmak, absztrakciók az osztályoknak feleltethetőek meg, míg azok a dolgok, amelyek előtt határozott névelőt használunk, vagy tulajdonnevük van, konkrét objektumokra vonatkoznak. A problémák megoldása terén tehát az objektum-orienált tervezéssel készült szoftverek a megoldást az objektumok

terében keresik. Egy szoftver készítésekor a lehető legpontosabban modellezhetjük azt a környezetet, amelyben a problémával, feladattal találkoztunk, vagy a lényeges tulajdonságokat kiemelve egy leegyszerűsített, absztrakt világban gondolkodhatunk.

2.4. Objektumorientált programtervezés

Manapság objektum-orientált rendszerek feltérképezésére, tervezésére leginkább az UML (Unified Modelling Language) eszközeit használják. Az UML szöveges dokumentumokat és diagramokat alkalmaz a vizsgált, vagy éppen tervezett rendszer modelljének kezelésére. Nagyon fontos, hogy ne keverjük össze a modellt a diagramokkal. A diagram csak a modell bizonyos szempontból tekintett vetületének grafikus megjelenítése. A diagramokat 3 osztályra lehet bontani: szerkezet, vagy struktúra diagramok, viselkedés diagramok, és interakciós diagramok. Az UML specifikáció a különböző diagramokhoz külön terminológiát alkalmaz. Az UML segítségével történő rendszer tervezés lépései:

- **Követelmények rögzítése.** Fogalomszótárat hozunk létre, amelyben a kifejlesztendő rendszer szakmai terminológiáját rögzítjük. Az áttekintésben szövegesen egy összefoglalást adunk a kifejlesztendő rendszer működéséről. Kiterjesztési mechanizmussal testre szabhatjuk az UML jelöléseit. Ezek alapján létrehozzuk a használati eseteket, melyek a rendszer felhasználóinak különböző megközelítéseit tartalmazzák. A kifejlesztendő rendszert absztrakt, funkcionálisan elkülönülő csomagokra bontjuk. A rendszerek együttműködése diagramban beillesztjük a már meglévő rendszerkörnyezetbe. Ezek után létrehozzuk a követelményelemzés további szöveges dokumentumait, amelyek a felhasználók igényeit tartalmazzák, egységes forgatókönyvszerű formában, megtervezzük a felhasználói felületeket. Ezek után elvégezzük a követelményelemzést: Egyeztetünk a megrendelővel, meghatározzuk a kritikus pontokat a tervben, megbecsüljük a költségeket.

- **Osztálydiagramokat készítése.** Ez tartalmazza a létrehozandó osztályokat, azok attribútumait, és metódusait, valamint az osztályok között kapcsolatokat, függőségeket. Az alkalmazás strukturális szerkezetét írjuk le vele. Ez szolgál majd az implementáció létrehozásának alapjául.

- **Interakció diagramok.** Az interakció diagramokon az objektumok nem strukturális kapcsolatait a közöttük lezajló műveletekkel jellemezhetjük. Ide tartoznak a szekvencia diagramok, valamint az együttműködési diagramok.

- **Időben lezajló változás.** Az aktivitás diagramokkal a munkafolyamatot az aktív szempontból, a végrehajtandó tevékenységek sorrendjével írják le, míg az állapot-átmenet diagramok passzív oldalról, a vizsgált elemet ért külső hatásokra bekövetkező reakciók alapján vizsgálják.

- Ezek után létrehozzuk a **szakterületi modell**t, ami az alkalmazás belső logikai szerkezetét ábrázolja.

- **Implementációs diagramok.** Az implementációs diagramokon az alkalmazás fizikai szoftver-alkotóelemeit, azok összefüggéseit és a közöttük lévő függőségeket ábrázoljuk, valamint az alkalmazással kapcsolatban álló, annak működését biztosító fizikai hardvert, számítógépeket, illetve egyéb egységeket, és azok egymás között kapcsolatait.

- **Tervezés és megvalósítás.** A tervezés a szakterületi modellt informatikai közegbe ülteti át. Meghatározzuk az alkalmazási környezetet, az implementációs platformot, adatbázis-kezelőt, programozási nyelveket, és technológiákat. Megvalósítás során először a csomagokat külön-külön dolgozzuk ki, aztán egybe integráljuk, és teszteljük az együttes működésüket.

2.5. Komponensorientált programozás

1968-ban egy NATO konferencián nevezték el a szoftverfejlesztésben kialakult helyzetet szoftver-krízisnek. Felismerték, hogy az egyre nagyobb méretű szoftverek készítése nem folytatódhat úgy, ahogy eddig történt. A szoftverek tervezése egyre bonyolultabbá vált, és kialakult a programtervezés, mint önálló mérnöki tudomány. Hasonló fejlődés figyelhető meg a történelem során más területeken is. A 19. században a kezdeti céhes iparűzési formát felváltották a manufaktúrák, ahol a késztermékeket sok kicsi, standard, kicserélhető darabból állították össze. Másik példának tekinthető az elektrotechnikai szakterületen a kicsi diódák, és tranzisztorok használata, amelyek nem egy speciális fela-

dat ellátására születtek, hanem általánosan felhasználhatók különböző készülékekben. A szoftvertervezés is ezt az újrafelhasználásorientált szemléletmódot akarja alkalmazni. Az elképzelés az, hogy a szoftverrendszereket függetlenül tervezett komponensek halmazaként állítsuk elő. A szoftver-komponens fogalom kezdettől jelen van, bár eleinte csak a Fortran forráskód-modulokat jelentette.

A komponensek legfontosabb jellemzője, hogy elrejtik az implementációjukat. A komponensek fekete dobozoknak képzelhetőek: a programozó tudja, hogy néz ki, és mire képes, de nem tudja, hogy hogyan működik valójában. Ez korlátozásnak tűnhet, de nem az. Az alkalmazásfejlesztők számára hasznos lehet, ha kicserélhetnek egy komponenst egy másikra, feltéve, hogy ugyanazt az interfészt implementálja. A komponens készítője ezzel szemben megváltoztathatja a komponens implementációját, feltéve, hogy az interfész változatlan marad. Az interfész a komponens alapú szoftver fejlesztés (CBSD) nagyon fontos fogalma. Talán úgy definiálható legjobban, hogy: metódusok megnevezett gyűjteménye, amelyek egy funkcionalitást reprezentálnak. Az interfész egyfajta szerződésnek fogható fel az interfész kibocsájtója, és a felhasználója között. A komponensek könnyen átültethetőek egyik alkalmazás környezetből a másikba, ezért a komponenseknek ön-tartalmazó szoftver elemeknek kell lenniük, amelyek függetlenek minen más komponenstől. Mivel a komponensek kicserélhetőek, nem címezhetik egymást közvetlenül, csak egy indirekciós mechanizmuson keresztül, egy központi nyilvántartó egység segítségével. Valahányszor egy kliens meg akar hívni egy metódust, lekérdezi a nyilvántartó egységtől a szűkséges interfészt, mire az visszatér egy referenciával. A következő probléma, amelybe komponensek létrehozásakor ütközünk, hogy interoperábilisnek kell lenniük, ami azt jelent, hogy tudniuk kell kommunkálni más programozási nyelven, más hoszton futó komponensekkel. Átültethetőnek kell lenniük más operációs rendszerekre és hardverre. Ezek csak virtuális gépekkel, kereszt-fordítókkal, és middleware-szoftverekkel biztosíthatók, mint például a CORBA az OMG-től, vagy a DCOM a Microsoft-tól. A komponenseket nem önállóan használják, hanem egy szoftver architektúrához kapcsolódóan, ami meghatározza, hogy a komponensek interfészeinek hogyan kell kinézniük, a komponensek hogyan legyenek megvalósítva. A komponens modellek egy közös keretrendszert biztosítanak, amelyben a komponensek együttműködhetnek. A keretrendszeren kívül a komponens nem használ-

ható. Két fontos komponens modell a JavaBeans a Sun Microsystems-től, és az ActiveX a Microsoft-tól, amely a Component Object Model (COM) -on alapszik. A különbség az objektum-orientált szemlélethez képest, (bár az OO mentén épül fel), hogy a komponens-orientált programozás sokkal absztraktabb képet ad a szoftverrendszerről.

A JavaBeans komponens modell. Legfontosabb jellemzője: Egy JavaBean újrafelhasználható szoftver komponens, amely vizuális építő eszközből módosítható. A JavaBean-ek nem sokban különböznek a hagyományos Java osztályoktól, ez könnyen használhatóvá teszi a modellt. A fő JavaBean koncepciók:

- a különböző építő eszközök képesek a Bean jellemzőinek (tulajdonságok, metódusok, események) felderítésére az introspekciónak nevezett folyamatban
- az un. tulajdonságok a Bean megjelenési és viselkedési jellemzői, amelyek a tervezés során megváltozhatnak
- a Bean-ek eseményekkel kommunikálnak egymással
- a perzisztencia lehetővé teszi hogy letárolják, és visszaállítsák az állapotukat

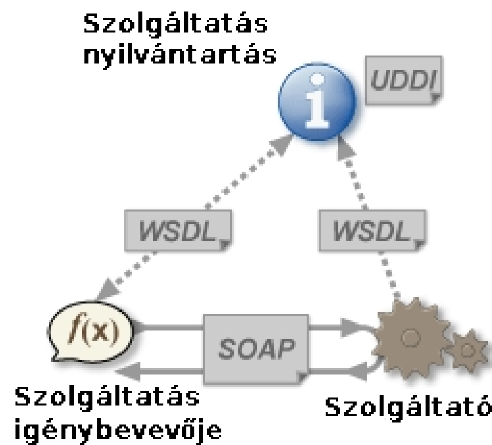
3. A TECHNOLÓGIA - WEBSZOLGÁLTATÁSOK

A '90-es évek végén sokan azt gondolták, hogy a DCOM és a CORBA párharcából fog kikerülni az a technológia, amely irányt mutat az üzleti szoftverek fejlesztésében, biztosítva az internetes kódújrafelhasználás lehetőségét, azonban egyik technológia sem váltotta be a hozzá fűzött reményeket. Használatuk nehézkes volt, és nehezen boldogultak az internetes tűzfalakon keresztül történő kommunikációval, a távoli gépek biztonságossága is megkérdőjelezhető volt. A fő probléma az volt, hogy nem globális, minden aspektusra kiterjedő szabványokon alapultak, sok volt a technológia függő dolog. Kettőjük között a webszolgáltatások emelkedtek ki győztesként.

A szolgáltatás, mint fogalom, hosszú idő óta jelen van, azonban az informatikában csak a 2000-es évek elején kezdték használni. Az internet széleskörű elterjedésével, és az interneten végezhető üzleti tevékenységek (elektronikus vásárlás, stb.) megjelenésével jött divatba. Az üzleti gondolkodásnak elengedhetetlen része a szolgáltatás. A vállalatok, cégek, szolgáltatásokat nyújtanak, és azokhoz más szolgáltatásokat használnak fel. Ha egy vállalat kellően nagy, és részekre, osztályokra tagolódik, azok külön, belső szolgáltatást nyújthatnak egymásnak.

Az világhálón megjelenő üzleti szolgáltatásokat kezdetben csak web-böngészőn keresztül érhettük el. Azonban az egyre újabb és újabb szolgáltatások kezdtek más, HTTP alapú interfészeket biztosítani, amelyeken keresztül különböző információkhoz juthattunk, műveleteket tehettünk anélkül, hogy az adott szolgáltatáshoz tartozó web-oldalt megnyitottuk volna. Ezeket az információkat, műveleteket aztán felhasználhattuk akár saját szolgáltatásaink létrehozásához. (Gondoljunk csak a különböző bankok által biztosított interneten megadott átutalási megbízásokra, amely lehetőség aztán a bankokon keresztül minden elektronikus vásárlással foglalkozó portálra beépült.) Az így kialakult szolgáltatások hálózatát, rendszerét a különböző internetes szabványokkal foglalkozó intézmények igyekeztek standardekkel és protokollokkal kézben tartani, létrehozva ezzel a szabványos webszolgáltatásokat.

A webszolgáltatás olyan szoftverrendszer, amely interoperábilis gép-gép közötti interakciókat tesz lehetővé számítógép hálózaton keresztül.



1. ábra. Webszolgáltatások architektúrája

A webszolgáltatások alapját három szabvány képzi: SOAP, WSDL, UDDI. A SOAP a platformfüggetlen üzenetváltást teszi lehetővé, a WSDL a webszolgáltatások interfészeinek szabványos leírását, az UDDI, pedig a webszolgáltatások közétételét, és a felderíthetőségét biztosítja, központi, interneten elérhető regiszterek segítségével.

3.1. SOAP

A SOAP mozaikszó a Simple Object Access Protocol kifejezésből ered, azonban mára elszakadt a jelentésétől, és a szolgáltatás-orientált környezetben megjelenő XML alapú üzenetküldő protokoll önálló megnevezésévé vált. A SOAP protokoll használatával teremtenek kapcsolatot a különböző webszolgáltatások egymás között. A SOAP üzenet alapjában véve egy egyirányú információ továbbítás SOAP csomópontok között, amelyet gépek generálnak, összetett interakciós minták megvalósítására. Maga a SOAP üzenet nem más, mint egy szöveges dokumentum, amely az XML jellegének köszönhetően tagekkel szabdalt adatokat tartalmaz. Szöveges mivoltának köszönhetően teljesen platformfüggetlen. A W3C által közétett 1.2 verziójú SOAP ajánlás 3 fő részből áll:

- **SOAP boríték specifikáció** - előírásokat tartalmaz a SOAP üzenet tagolására, formájára.
- **Adatkódolási szabályok** - az SOAP üzenetekben megjelenő adattípusok definícióját, formáját, és leírását adja.
- **RPC szabályok** - a távoli metódushívással kapcsolatos megoldásokat specifikálja.

3.1.1. SOAP boríték specifikáció

Az SOAP üzenet 4 fő részből áll:

- `<envelope>` (boríték) - kötelező, az üzenet gyökér eleme, mintegy bekeretezi az egész üzenetet
- `<header>` (fejrész) - opcionális, az üzenet útvonalára tartalmazhat információkat
- `<body>` (törzs) - kötelező, az üzenet elsődleges információtartalmát hordozza. A SOAP tervezésének egyik fő célja az volt, hogy adatokat, és azok pontos specifikációját lehessen megjeleníteni az üzenetben, míg a másik fő cél, hogy távoli eljárás hívásokat (Remote Procedure Call) lehessen vele becsomagolni. Az előbbi esetben a tetszőleges adatok megfelelő XML szintaktikával kerülhetnek bele a törzsbe. Az utóbbi esetben azonban, egy RPC megvalósításához az üzenetnek tartalmaznia kell a következő információkat:
  - a cél SOAP csomópont címét,
  - a metódus, vagy eljárás nevét,
  - a paraméterként átadott argumentumok azonosítóját és értékét, valamint a visszatérési értéket, és az output paramétereket.
  - az üzenetváltási mintát, amelyet az RPC végrehajtásánál alkalmazni kell.
- `<fault>` (hiba) - hiba esetén a `<body>` elem tartalmazhat egy `<fault>` tag-et, amely megadja a hiba kódját, valamint tartalmazhat egy olyan részt (`<reason>`-tag), amely emberi feldolgozás céljára szövegesen leírja a hiba lehetséges okait. A `<fault>` elem megjelenése és tartalma erősen függ attól, hogy a SOAP rendszer hogyan kötődik az alatta fekvő tényleges végrehajtásért felelős réteghez.

Egy lehetséges SOAP kérés:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

Egy lehetséges SOAP válasz:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

A fenti SOAP üzenet egy RPC-kérést tartalmaz HTTP protokollon keresztül, egy input paraméterrel. Megfigyelhető az `<Envelope>` elemben a szabványos `envelope` névtér megadása. Szintén az `<Envelope>` tag-ban található a szerializációs szabályokat megadó `encodingStyle` attribútum. A `<Body>` elemben található a metódushívás specifikációja, és szintén a `<Body>` elem attribútuma a metódusnév elemet definiáló névtér. A metódus neve `getStockPrice`, melynek egyetlen paramétere a `StockName`, melyek azonos nevű elemekben vannak megadva. A SOAP válasz felépítése azonos.

Egy SOAP üzenet feldolgozása a következő módon történik: Első lépésként, a feldolgozó SOAP csomópont megvizsgálja a SOAP-üzenet szintaktikailag helyes-e, megfelel-e a hozzá tartozó DTD előírásnak, és elemzi a SOAP specifikus részeket: az `<envelope>`, a `<header>`, és a `<body>` tag-ek tartalmát. A `<header>`-rész blokkokból állhat, amelyek azt határozzák meg, hogy a cél SOAP csomópontig vezető úton az egyes csomópontok

mit tegyenek az üzenettel. A blokkok tartalmaznak egy `role` attribútumot, amelynek értéke egy sztring, ami egy szerepkört jelent. Ha van olyan `header` blokk, amelyhez tartozó szerep ellátására az adott csomópont képes, akkor a megfelelő tevékenységet végre hajtahatja. Három standard szerep van, amelyet minden SOAP csomópontnak ismernie kell:

- `none` (üres, nincs szerep)
- `next` (tovább)
- `ultimateReceiver` (célÁllomás).

Az ajánlás nem szól arról, hogy pontosan milyen tevékenységeket kell végrehajtania az adott csomópontnak. Egy `header` feldolgozása jelentheti az üzenet újragenerálását módosított `header`-rel. A `header` blokkokban megjelenhet egy további attribútum, mégpedig a `mustUnderstand`. Ha ez az argumentum egy `header` blokkban `true` értékkel szerepel, az azt jelenti, hogy a blokkhoz tartozó szereppel rendelkező csomópontnak kötelező jelleggel fel kell dolgoznia a blokkot. Ha nem teszi, akkor `fault` hiba generálódik. Ezzel az attribútummal biztosítható, hogy minden szűkséges művelet végrehajtódjon az üzeneten. A `header` blokkoknak létezhet egy további attribútuma, amely logikai értéket vehet fel. A `relay` attribúum értéke azt jelzi, hogy amennyiben a `header` blokk a megfelelő szereppel rendelkező csomóponthoz ér, és a csomópont nem dolgozza fel a blokkot, kerüljön-e továbbításra a blokk, vagy sem. Alapértelmezésben ha egy blokk megfelelő szerepű csomóponthoz ér, kikerül a generált, továbbított üzenetből, függetlenül attól, hogy a csomópont feldolgozta-e a blokkot, vagy sem.

```
<q:anotherBlock xmlns:q="http://example.com"
  env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
  env:relay="true">
...
...
</q:anotherBlock>
```

Ha egy üzenet olyan csomóponthoz ér, amely `next` szereppel rendelkezik, akkor a szerepnek megfelelő blokk akár feldolgozásra kerül, akár nem, a csomópont továbbítja az

üzenetet, azonban a blokkot eltávolítja az üzenetből. Ha azonban (mint a fenti példában is), true értékkel szerepel a blokkban a relay attribútum, a csomópontnak továbbítania kell az `anotherBlock` blokkot is.

A SOAP üzenetek különböző alsóbb rétegbeli protokollok segítségével juthatnak egyik csomópontról a másikra. Az erre vonatkozó specifikációt `SOAP-binding`-nek nevezzük. A `SOAP-binding` az üzenet `<Envelop>` elemével közrezárt információhalmaz szerializációját jelenti olyan formába, amelyből a következő csomópont információveszteség nélkül rekonstruálni tudja az eredeti információkat. Több különböző `binding` protokoll létezik. A leggyakrabban használt a HTTP. A HTTP összeköttetési, és üzenetküldési modellje a következő: A kliens azonosítja a szervert egy URI-val. Kapcsolódik hozzá, kihasználva az alatta lévő TCP/IP hálózatot, és egy HTTP-kérés üzenetet intéz, majd egy HTTP-válasz üzenetet fogad ugyanazon a TCP kapcsolaton. A HTTP egyértelműen megfelelteti a válaszokat a kéréseknek. Ezek alapján egy HTTP-kérésben küldött SOAP üzenetnek megfeleltethető a HTTP-válaszban kapott SOAP üzenet. Továbbá a HTTP a kiszolgálókat URI-val azonsítja, amely használható egy SOAP csomópont azonosítójaként is. A SOAP-üzenetek továbbítására a HTTP protokollon kívül gyakran használják még az e-mail küldés lehetőségét (SMTP).

3.1.2. Adatkódolási szabályok

A SOAP üzeneteknek platformfüggetlennek kell lenniük, annak érdekében, hogy a különböző programozási nyelveken írt, különböző keretrendszereken futó, és különböző architektúrákon működő rendszerek egyaránt feldolgozhassák őket. A platformfüggetlenség egyik, talán legsarkalatosabb pontja, hogy hogyan feleltessük meg a különböző adattípusokat egymásnak. A különböző nyelvek az egymásnak megfelelő adattípusokat sokszor nem úgy ábrázolják (például a lebegőpontos számokat nem olyan pontossággal), valamint előfordulhat, hogy a legelemibb típusok egyikének egyáltalán nincs megfelelője egy másik nyelvben. A probléma megoldására a SOAP egy saját, minden nyelvtől független típus, és kódolási rendszert használ, amit aztán a különböző implementációk már pontosan fordíthatnak a megfelelő típusra.

A SOAP adattípusai az XML Schema specifikációból erednek. (Az XML semmiféle megkötést nem tartalmaz az adattípusokra vonatkozóan) A SOAP adattípusokat két nagy csoportra oszthatjuk. A skalár típusok pontosan egy, atomi értékkel rendelkeznek, míg az összetett típusok összetett értéket vehetnek fel. Az összetett típusokat szintén tovább bonthatjuk: tömbre, és struct típusra. A tömbben minden elemhez egy pozíció tartozik, míg a struct típus elemeire egy-egy névvel hivatkozunk. Az XML Schema-ban minden atomi adattípus vagy egyszerű, vagy származtatott. Az egyszerű adattípus más adattípusok által nem kifejezhető. Az egyszerű adattípusokból hozhatóak létre a származtatott típusok, melyeket az egyszerű típusok tartományainak szűkítésével nyerünk. Az így létrejött összetett típushierarchia olyan módon definiálja a `string`, `float`, `boolean`, `URI`, vagy `time` típusokat, hogy minden alkalmazásplatform alkalmas lesz a megértésükre. Az összetett típusok közül a tömb esetén annak létrehozásakor ki kell jelölnünk a tömb, és az elemtípus méretét is, majd a tömb elemeit az `<item>`-tag -ekkel adhatjuk meg. `struct` típus esetén minden egyes mezőhöz külön kiegészítő elemet kell kijelölnünk.

Tömb szerializációjánál látható, hogy a tömb elemei sorfolytonosan kerülnek megadásra:

```
<SOAP-ENC:Array id="array-2" SOAP-ENC:arrayType="xsd:string[2]">
   <item>r1c1</item>
   <item>r1c2</item>
</SOAP-ENC:Array>
```

Struktúra szerializációja esetén mielőtt megadnánk a struktúra aktuális elemeit, definiálnunk kell a struktúra szerkezetét a megfelelő XML-Schema dokumentumban:

```
<e:Book>
   <author>Henry Ford</author>
   <preface>Prefatory text</preface>
   <intro>This is a book.</intro>
</e:Book>
```

Ezesetben a struktúrát leíró Schema részlet a következő:

```
<element name="Book">
<complexType>
  <element name="author" type="xsd:string"/>
  <element name="preface" type="xsd:string"/>
   <element name="intro" type="xsd:string"/>
</complexType>
</e:Book>
```

3.1.3. RPC szabályok

A SOAP egyik fő feladata, hogy távoli eljáráshívásokat lehessen leírni/kódolni az üzenetekben. Ezek a leírások függetlenek minden programozási nyelvtől, a cél csomópontnak kell gondoskodnia róla, hogy hogyan végzi el a tényleges metódushívást. A World Wide Web az erőforrásokat URI -kkal azonsoítja. Bár a SOAP ajánlás nem írja elő, célszerű a címző URI-ba minden olyan azonosítót, és argumentumot belevenni, amely segíthet egyértelműen azonosítani a távoli eljárást. Egy távoli eljáráshívás a következő módon kerül kódolásra:

- A hívást egy egyszerű struktúra reprezentálja, amely minden egyes in vagy in/out paraméterhez tartalmaz egy `tag`-et.
- A struktúra neve azonos az eljárás, vagy metódus névvel.
- Minden paramétert tartalmazó `tag` címkéje a paraméter nevét tartalmazza.

Az RPC válaszok a modellezése a következő:

- A választ szintén egy egyszerű struktúra tartalmazza, amelyben van egy `tag` a visszatérési érték, és minden egyes `out`, vagy `in/out` paraméter számára.
- A struktúra neve nincs meghatározva, általában a meghívott eljárás neve a `Response` utótaggal.
- Az elem neve, amely a visszatérési értéket reprezentálja, szintén nincs meghatározva, de a válasz elemen belül az első helyen kell szerepelnie, és utána az `out` és `in/out` paraméterek.

Az RPC hívások a standard SOAP hibakódokat további, specifikus hibakódokkal egészítik ki.

3.1.4. SOAP implementációk

A W3C SOAP specifikáció egy ajánlás, melynek százas nagyságrendű implementációja van. Ezek közül a leggyakrabban használt az Apache SOAP, és a Microsoft SOAP implementáció. Bár mind a W3C ajánlást valósítja meg, az ajánlásban nyitva hagyott kérdések

miatt a megvalósítások eltérőek, és ezért gyakran nem teljesen kompatibilisek.

Bár a SOAP üzenetek használata első pillantásra összetett és nehéz feladatnak tűnhet, aki webszolgáltatásokat akar készíteni, nem kell SOAP üzenetekkel közvetlenül találkoznia. A különböző fejlesztő rendszerek és implementációk automatikus eszközöket biztosítanak a SOAP-üzenetek generálására, és feldolgozására. Az üzenetek pontos felépítésének, és jelentésének ismerete csak akkor szűkséges, ha az üzenetek alapján akarunk hibakeresést, nyomkövetést végezni.

3.2. WSDL

A `WSDL (Web Services Description Language)` eszközeivel írhatjuk le a webszolgáltatásokat egy közös, `XML` alapú nyelven. A `WSDL` állományból különböző információkat tudhatunk meg a webszolgáltatásról:

- interfész információk
- adattípusokra vonatkozó információk
- összekapcsolódási (binding) információk
- címzési információk

A WSDL leírás, és egy egyszerű interfész között az a különbség, hogy a WSDL platform- és nyelvfüggetlen, elsősorban SOAP alapú szolgáltatások leírására használatos. A WSDL-leírás segítségével meg tudjuk találni a webszolgáltatást, kapcsolódni tudunk hozzá, illetve a leírásból tudjuk, hogy a szolgáltatás milyen formában fog válaszolni. A WSDL XML -t használ a webszolgáltatások leírásához. A specifikáció hat főelemre bontja a WSDL-t:

- `<definitions>` - ez a gyökér elem, szerepelnie kell minden WSDL dokumentumban.
- `<types>` - a kliens és a szerver között alkalmazott típusleírást tartalmazza. Hasonlóan a SOAP-hoz, az XML Schema specifikációt használja. Ha a szolgáltatás csak az XML Schema egyszerű típusait használja, nincs szűkség a types elemre.
- `<message>` - egyirányú üzenetet ír le, ami lehet kérés, vagy válasz. Definálja az üzenet nevét, valamint a paramétereit, vagy visszaadott értékeit.

- <portType> - a portType elem a message elemek kombinációjából formál egyirányú, vagy kétirányú műveleteket. Például egy portType művelet egyesíthet egy kérés és egy válasz üzenetet egy önálló kérés/válasz műveletté.
- <binding> - a szolgáltatás konkrét megvalósítását írja le.
- <service> - a meghívandó szogláltatás helyét határozza meg. Általtában egy URL-t foglal magában a SOAP szolgáltatás eléréséhez.

További két elemet alkalmazhatunk egy WSDL leírásban:

- <documentation> - ember által olvasható magyarázó szöveget tartalmaz, bármely WSDL elem tartalmazhatja.
- <import> - más WSDL dokumentum, vagy XML Schema importálására alkalmazhatjuk.

```
<?xml version="1.0"?>

<!-- WSDL description of the Google Web APIs.
     The Google Web APIs are in beta release. All interfaces are subject to
     change as we refine and extend our APIs. Please see the terms of use
     for more information. -->

<!-- Revision 2002-08-16 -->

<definitions name="GoogleSearch"
             targetNamespace="urn:GoogleSearch"
             xmlns:typens="urn:GoogleSearch"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <!-- Types for search - result elements, directory categories -->

  <types>
    <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="urn:GoogleSearch">

      <xsd:complexType name="GoogleSearchResult">
        <xsd:all>
    <xsd:element name="documentFiltering"           type="xsd:boolean"/>
    <xsd:element name="searchComments"              type="xsd:string"/>
    <xsd:element name="estimatedTotalResultsCount"  type="xsd:int"/>
    <xsd:element name="estimateIsExact"             type="xsd:boolean"/>
    <xsd:element name="resultElements"              type="typens:ResultElementArray"/>
    <xsd:element name="searchQuery"                 type="xsd:string"/>
    <xsd:element name="startIndex"                  type="xsd:int"/>
    <xsd:element name="endIndex"                    type="xsd:int"/>
    <xsd:element name="searchTips"                  type="xsd:string"/>
    <xsd:element name="directoryCategories"         type="typens:DirectoryCategoryArray"/>
    <xsd:element name="searchTime"                  type="xsd:double"/>
        </xsd:all>
      </xsd:complexType>
```

```
    <xsd:complexType name="ResultElement">
      <xsd:all>
  <xsd:element name="summary" type="xsd:string"/>
  <xsd:element name="URL" type="xsd:string"/>
  <xsd:element name="snippet" type="xsd:string"/>
  <xsd:element name="title" type="xsd:string"/>
  <xsd:element name="cachedSize" type="xsd:string"/>
  <xsd:element name="relatedInformationPresent" type="xsd:boolean"/>
  <xsd:element name="hostName" type="xsd:string"/>
  <xsd:element name="directoryCategory" type="typens:DirectoryCategory"/>
  <xsd:element name="directoryTitle" type="xsd:string"/>
      </xsd:all>
    </xsd:complexType>

    <xsd:complexType name="ResultElementArray">
      <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="typens:ResultElement[]"/>
        </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="DirectoryCategoryArray">
      <xsd:complexContent>
        <xsd:restriction base="soapenc:Array">
  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="typens:DirectoryCategory[]"/>
        </xsd:restriction>
      </xsd:complexContent>
    </xsd:complexType>

    <xsd:complexType name="DirectoryCategory">
      <xsd:all>
        <xsd:element name="fullViewableName" type="xsd:string"/>
        <xsd:element name="specialEncoding" type="xsd:string"/>
      </xsd:all>
    </xsd:complexType>

  </xsd:schema>
</types>

<!-- Messages for Google Web APIs - cached page, search, spelling. -->

<message name="doGetCachedPage">
  <part name="key"            type="xsd:string"/>
  <part name="url"            type="xsd:string"/>
</message>

<message name="doGetCachedPageResponse">
  <part name="return"         type="xsd:base64Binary"/>
</message>

<message name="doSpellingSuggestion">
  <part name="key"            type="xsd:string"/>
  <part name="phrase"         type="xsd:string"/>
</message>

<message name="doSpellingSuggestionResponse">
  <part name="return"         type="xsd:string"/>
</message>

<!-- note, ie and oe are ignored by server; all traffic is UTF-8. -->

<message name="doGoogleSearch">
  <part name="key"            type="xsd:string"/>
  <part name="q"              type="xsd:string"/>
  <part name="start"          type="xsd:int"/>
  <part name="maxResults"     type="xsd:int"/>
  <part name="filter"         type="xsd:boolean"/>
  <part name="restrict"       type="xsd:string"/>
  <part name="safeSearch"     type="xsd:boolean"/>
```

```
  <part name="lr"              type="xsd:string"/>
  <part name="ie"              type="xsd:string"/>
  <part name="oe"              type="xsd:string"/>
</message>

<message name="doGoogleSearchResponse">
  <part name="return"          type="typens:GoogleSearchResult"/>
</message>

<!-- Port for Google Web APIs, "GoogleSearch" -->

<portType name="GoogleSearchPort">

  <operation name="doGetCachedPage">
    <input message="typens:doGetCachedPage"/>
    <output message="typens:doGetCachedPageResponse"/>
  </operation>

  <operation name="doSpellingSuggestion">
    <input message="typens:doSpellingSuggestion"/>
    <output message="typens:doSpellingSuggestionResponse"/>
  </operation>

  <operation name="doGoogleSearch">
    <input message="typens:doGoogleSearch"/>
    <output message="typens:doGoogleSearchResponse"/>
  </operation>

</portType>


<!-- Binding for Google Web APIs - RPC, SOAP over HTTP -->

<binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
  <soap:binding style="rpc"
               transport="http://schemas.xmlsoap.org/soap/http"/>

  <operation name="doGetCachedPage">
    <soap:operation soapAction="urn:GoogleSearchAction"/>
    <input>
      <soap:body use="encoded"
                namespace="urn:GoogleSearch"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded"
                namespace="urn:GoogleSearch"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>

  <operation name="doSpellingSuggestion">
    <soap:operation soapAction="urn:GoogleSearchAction"/>
    <input>
      <soap:body use="encoded"
                namespace="urn:GoogleSearch"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded"
                namespace="urn:GoogleSearch"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>

  <operation name="doGoogleSearch">
    <soap:operation soapAction="urn:GoogleSearchAction"/>
    <input>
      <soap:body use="encoded"
                namespace="urn:GoogleSearch"
```

```
                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
                  namespace="urn:GoogleSearch"
                  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>

  <!-- Endpoint for Google Web APIs -->
  <service name="GoogleSearchService">
    <port name="GoogleSearchPort" binding="typens:GoogleSearchBinding">
      <soap:address location="http://api.google.com/search/beta2"/>
    </port>
  </service>

</definitions>
```

Példaként vizsgáljuk meg a `Google Search API WSDL` állományt (google.wsdl). Az egész leírást mintegy bekeretezi a `<definitions>` tag, melyben megadjuk a névtér specifikációkat.

A `<types>` elemben saját összetett típusokat definiálunk XML Schema típusdefiníciójának megfelelő módon: a

- `GoogleSearchResult`,
- `ResultElement` ,
- `ResultElementArray` ,
- `DirectoryCategoryArray` és
- `DirectoryCategory`

típusokat.

Ezet követően a `<massage>` elemekben definiáljuk az egyirányú üzeneteket, amelyek a szolgáltatással történő kommunikációban használunk majd. Külön definiáljuk a lekérdező üzeneteket, és az azokra adott válasz üzeneteket:

- `doGetCachedPage` ,
- `doGetCachedPageResponse` ,
- `doSpellingSuggestion` ,

- `doSpellingSuggestionResponse` ,
- `doGoogleSearch` ,
- `doGoogleSearchResponse`

Az egyirányú üzenetekből a `<operations>` tag-ekkel a `<portType>` elemen belül műveleteket definiálunk. Az így létrejött `GoogleSearchPort` szemléletesen azt jelenti, hogy ezen a porton kapcsolódva a szolgáltatáshoz a `<portType>` elemben definiált `doGetCachedPage` , `doSpellingSuggestion` , `doGoogleSearch` műveleteket hajthatjuk végre, amelyek mindegyike jelen esetben egy kérés (input) és egy válasz (output) üzenetküldést jelent.

Ezután a `<binding>` elemen belül definiáljuk a fentebb definiált műveletek (operations) kapcsolódását a konkrét SOAP megvalósításhoz: rögtön a WSDL `<binding>` eleme után következik a `<soap:binding>` elem, amelyben specikiáljuk, hogy a műveletek `rpc` távoli metódus hívások és a `transport` attribútummal jelezzük, hogy HTTP protokollon keresztül kommunikálunk. Ezt követően minden egyes művelet input és output üzenetére megadják a SOAP szerializáció típusát az `encodingType` attribútumban.

A `<service>` elemben adjuk meg a webszolgáltatás elérhetőségét, egy port, binding és cím összerendeléssel.

A WSDL állományokat elkészíthetjük kézzel is, de természetesen erre is automatikus eszközök állnak rendelkezésünkre, melyek a különböző platformokon különböző módon működnek, de közös bennük, hogy a webszolgáltatás kódjából generálják a WSDL állományt. Java környezetben például egy megfelelő interfészt implementáló metódusok kerülnek bele a WSDL állományba, míg .NET rendszerben a megfelelő attribútummal rendelkező metódusok.

3.3. UDDI

Az UDDI (Universal Discovery Description and Integration) 3.0 szolgáltatások regisztrálására, felkutatására és integrálására használt technikai specifikáció. Az UDDI a különböző ipari és üzleti ágazatokon átívelő, web szolgáltatásokat támogató specifikációk rendszere.

Ezek a specifikációk meghatározzák, hogyan írjuk le szolgáltatásokat, üzleteket, mindezt standard XML, HTTP, IP technológiákon alapuló eszközökkel. Leírhatunk XML alapú és nem XML alapú szolgáltatásokat is. Az UDDI 3 fő részből áll:

- **UDDI adatmodell** - ez egy XML Schema definíció üzleti vállalkozások, szolgáltatások, webszolgáltatások és azok kapcsolatainak leírására.
- **UDDI API** - SOAP alapú üzenetküldés API, az UDDI-ben adatok keresésére és közétételére szolgál.
- **UDDI URB** - (UDDI Business Registry) Az UDDI specifikáció megvalósulásai, weboldalak, melyeken UDDI-hez kapcsolódó műveleteket lehet végezni.

Az UDDI-ben tárolt adatokat 3 csoportba soroljuk:

- **Fehér oldalak** - a vállalkozásokkal, és azok szolgáltatásaival kapcsolatban tartalmaznak információkat.
- **Sárga oldalak** - a szolgáltatások kategorizálva jelennek meg benne.
- **Zöld oldalak** - a szolgáltatások eléréséhez szűkséges technikai információkat tartalmazzák.

3.3.1. Adatmodell

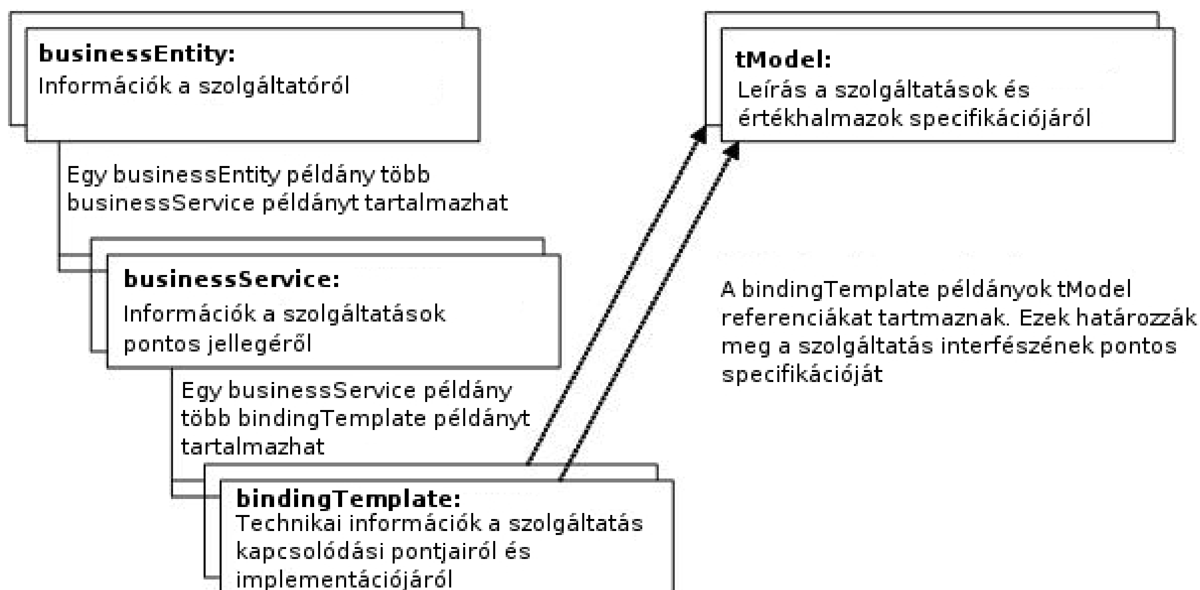
Az UDDI négy alap adatszerkezetet definiál az adatok tárolására:

- `businessEntity`
- `businessService`
- `bindingTemplate`
- `tModel`

A fenti adatszerkezetek segítségével helyezhetünk el adatokat egy UDDI nyilvántartóban.

A businessEntity adatszerkezet

Minden egyes `businessEntity` példányt egyértelműen azonosít a `<businessKey>` attribútuma. Ezt az attribútumot megadhatjuk mi is, de ha nem adjuk meg, akkor az nyilvántartó generál egyet.



2. ábra. Az adatszerkezetek kapcsolata

Az adatszerkezet elemei:

- `<discoveryURLs>` - olyan URL-ek listája, amelyek más fájl-alapú, szolgáltatás kereső rendszerekre mutatnak.
- `<name>` - az üzleti entitás neve, szöveges formátumban. Több `<name>` elem is szerepelhet, melyek más-más nyelven tartalmazzák az információt.
- `<description>` - szöveges leírása az üzleti entitásnak, szintén több ilyen elem szerepelhet, különböző nyelven.
- `<contacts>` - csak egy ilyen elem lehet, mely az üzleti entitás elérhetőségét tartalmazza.
- `<businessServices>` - az üzleti entitás által nyújtott szolgáltatások listája.
- `<identifierBag>` - további azonosítókat tartalmaz, melyek más rendszerekben azonosítják az üzleti entitást.
- `<categoryBag>` - az üzleti entitás tevékenységi köreit adja meg.

A businessService adatszerkezet

Egy `businessService`-t példányt egyértelműen azonosít egy `serviceKey` attribútum. Ha

nem adunk meg `serviceKey` attribútumot, akkor a nyilvántartó generál egyet. Minden üzleti szolgáltatás pontosan egy üzleti entitáshoz tartozik, a `businessKey` attribútum ennek az üzleti entitásnak az azonosítóját tartalmazza.

Az adatszerkezet elemei:

- `<name>` - az üzleti szolgáltatás példány nevét adja meg, több is szerepelhet belőle
- `<description>` - az üzleti szolgáltatás szöveges leírását adja, több ilyen elem is szerepelhet, különböző nyelven
- `<categoryBag>` - azokat az üzleti ágazatokat azonosítja, amelyekhez az üzleti szolgáltatás tartozik (a sárga oldalak generálásánál van szerepe)
- `<bindingTemplates>` - elem a biztosított webszolgáltatások technikai leírása.

A bindingTemplate adatszerkezet

Minden egyes `bindingTemplate` példányt egyértelműen azonosít a `<bindingKey>`. A `<serviceKey>` elem a szolgáltatást azonosítja, amelyhez a `bindingTemplate` tartozik.

Az adatszerkezet elemei:

- `<description>` - jelentése azonos a korábbiakkal, rövid leírást adhatunk a példányról, akár több nyelven is.
- `<accessPoint>` - a szolgáltatás használatához szükséges információt tartalmazó sztring. Nincs megkötés az információ mibenlétére. Általában egy hálózati cím, de lehet e-mail-cím vagy akár egy telefonszám is.
- `<hostingRedirector>` - elavult, funkcióját az `<accessPoint>` hordozza magában.
- `<tModelInstanceDetails>` - egy vagy több `tModelInstanceInfo` példányt tartalmaz. A `tModelInstanceInfo` példányokban lévő `tModelKey` attribútumok segítségével kereshetünk kompatibilis webszolgáltatásokat.
- `<categoryBag>` - elem hasonló a korábban leírt azonos nevű elemhez: annak a webszolgáltatásnak tartalmazza a felhasználási területeit, amelyet a `bindingTemplate` ír le.

```
<businessEntity businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
                operator="www.ibm.com/services/uddi"
                authorizedName="0100001QS1">
  <discoveryURLs>
    <discoveryURL useType="businessEntity">http://www.ibm.com/services/uddi/uddiget?
    businessKey=BA744ED0-3AAF-11D5-80DC-002035229C64</discoveryURL>
  </discoveryURLs>
  <name>XMethods</name>
  <description xml:lang="en">Web services resource site</description>
  <contacts>
    <contact useType="Founder">
      <personName>Tony Hong</personName>
      <phone useType="Founder" />
      <email useType="Founder">thong@xmethods.net</email>
    </contact>
  </contacts>
  <businessServices>
    <businessService serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
                     businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
      <name>XMethods Delayed Stock Quotes</name>
      <description xml:lang="en">20-minute delayed stock quotes</description>
      <bindingTemplates>
        <bindingTemplate bindingKey="d594a970-3e16-11d5-98bf-002035229c64"
                         serviceKey="d5921160-3e16-11d5-98bf-002035229c64">
          <description xml:lang="en">SOAP binding for delayed stock quotes service</description>
          <accessPoint URLType="http">http://services.xmethods.net:80/soap</accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
    </businessService>
  </businessServices>
</businessEntity>
```

A fenti példában egy egyszerű `businessEntity` példányt láthatunk. Megfigyelhető a `businessKey` attribútum, amelyet mi határoztunk meg, valamint a `<name>`, `<description>` és `<contacts>` elemek. A `<businessServices>` elemen belül a `businessService` adatszerkezet egy példánya van megadva, melyen belül a látható a `<bindingTemplate>` és annak a `tModelInstanceDetails` eleme, mely a kulcsával hivatkozik arra `tModel` példányra, amelyet a szolgáltatás megvalósít.

A tModel adatszerkezet

Az UDDI-nek nagyon fontos tulajdonsága, hogy a webszolgáltatásokat úgy írhatjuk le benne, hogy a leírások elég információt tartalmazzanak ahhoz, hogy kereséseket tudjunk végezni benne. További fontos tervezési szempont volt, hogy a leírások kellő mennyiségű információt adjanak arról, hogy hogyan lehet ezeket a webszolgáltatásokat elérni, és dolgozni velük. Ehhez valamilyen módon le kell tudni írni azt, hogy hogyan viselkedik, milyen konvenciókat követ, milyen specifikációkhoz ragaszkodik. Ezt a leírást hivatott megadni a `tModel` adatszerkezet. Minden egyes tModel példány egy kulcsal rendelkező egyed az UDDI-ben. Általános esetben a tModel példányok célja egy absztrakción alapuló referencia rendszer biztosítása. Az egyik gyakori használata a `tModel` példányoknak a

technikai specifikációk vagy konvenciók megadása. A `tModel` példányok az UDDI nyilvántartóban valójában kulccsal rendelkező önálló techinkai specifikációk, leírások, melyeket a kulcsukra hivatkozó webszolgáltatásoknak meg kell valósítaniuk. Ilymódon egy `tModel` példány technikai ujjlenyomatnak tekinthető. Minden egyes `tModel` egyedet egyértelműen azonosít a `tModelKey` attribútum.

Az adatszerkezet elemei

- `name` , `description` - használatuk és céljuk megegyezeik a korábban tárgyalt, azonos nevű elemekkel.
- `overviewDox` - hivatkozásokat adhatunk meg további leírásokra, és előírásokra, a `tModel` példány használatát illetően.
- `identifierBag` , `categoryBag` - használatuk, és céljuk ugyanaz, mint ahogy azt korábban láttuk a `businessEntity` struktúránál.

Példa `tModel` példányra:

```
<tModel tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
        operator="www.ibm.com/services/uddi"
        authorizedName="0100001QS1">
  <name>XMethods Simple Stock Quote</name>
  <description xml:lang="en">Simple stock quote interface</description>
  <overviewDoc>
    <description xml:lang="en">wsdl link</description>
    <overviewURL>http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl</overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
                    keyName="uddi-org:types"
                    keyValue="wsdlSpec" />
  </categoryBag>
</tModel>
```

publisherAssertion adatszerkezet

Sok olyan vállalat és szervezet van, amelyet nem lehet leírni egyetlen `businessEntity` elemmel. Ennek általában az az oka, hogy a leírásaik, és ennek megfelelően a profiljaik szerteágazóak. A legegyszerűbb megoldás több üzleti entitást regisztrálni. Ebben az esetben viszont valamilyen módon jelölni kell, hogy ezek az entitások egymáshoz kapcsolódnak. Erre szolgálnak a `publisherAssertion` struktúra példányai.

Az adatszerekezet elemei:

- `<fromKey>` , `<toKey>` - azokat az üzleti entitásokat adja meg, amelyek egymással valamilyen kapcsolatban vannak.
- `<keyedReference>` - hivetkozás a kapcsolatot leíró adatszerkezetre.

A két `businessEntity` példányt, amelyek között fennáll a kapcsolat, egyértelműen azonosítják a `fromKey` és a `toKey` elemek. Magát a kapcsolatot a `keyedReference` tag-en belül írhatjuk le.

3.3.2. UDDI API

Az UDDI API specifikáció SOAP alapú leírása olyan interfészeknek, melyeken keresztül az UDDI nyilvántartókban lehet műveleteket végezni: Leírásokat lehet benne elhelyezni, illetve kereséseket végrehajtani.

Az UDDI API-nek több implementációja van, amely lehetővé teszi a nagyobb szolgáltatók UDDI nyilvántartóinak elírését. Ilyen implementáció például a GUI-s felülettel rendelkező JAXR Registry Browser.

3.3.3. UDDI UBR

Az UDDI Business Registry egy kezdetben egy nyilvános weboldal, amelyet a legnagyobb cégek (Microsoft, SAP) együttesen működtettek 2006-ig, amikorra bebizonyosodott, hogy az UDDI 3.0 specifikáció elég robusztus, hogy a webszolgáltatások alapjául szolgáljon.

A nyilvános UBR üzemeltetése abbamaradt, azonban a különböző gyártók saját termékeikben továbbra is implementálják az UDDI lehetőségeket. UDDI API-t implementáló gyártók: Acumen Technology, Apache.org, BEA, Bindingpoint, Cape Clear Software, Fujitsu, IBM, Infravio, IONA, Microsoft, Novell, Oracle, SAP, Select Business Solutions, SOA Software, Sun Microsystems, Inc, Systinet, UDDI4J.org, webMethods.

3.4. ESB

Az önálló webszolgáltatások még nem nevezhetőek architektúrának. Ahhoz, hogy architektúrát alkossanak, további funkciókat kell megvalósítani a szoftverrendszerekben. A

mai üzleti alkalmazások elosztott architektúrájúak. Ahhoz, hogy egy tetszőleges platformon működő, és tetszőleges nyelven megírt alkalmazás alkalmazás egy másikkal hálózaton keresztül optimálisan tudjon együttműködni, szolgáltatásorientált és eseményvezérelt architektúrákat kell létrehozni. Az Enterprise Service Bus (ESB) egy olyan middleware, amely magában foglalja az integrációs technológiákat és a futásideji szolgáltatásokat, lehetővé téve ezzel, hogy az üzleti szolgáltatás széles körben felhasználható legyen, és ezzel a legjobb megoldást nyújtja az üzleti alkalmazások integrációjának problémájára. Valójában az ESB biztosítja egy webszolgáltatásokon alapuló szolgáltatásorientált architektúra megvalósításában az architektúrát. A probléma az, hogy több óriásvállalat is készített ESB megoldásokat. Néhány a SOAP/HTTP protokollcsaládra szorítkozik, míg mások multi-protokollt támogató rendszerek. Az ESB-k listája a legtöbb lehetőséget nyújtó applikációs szerverektől a kisebb, specifikus ESB integrációt biztosító eszközökig terjed. Vizsgáljuk meg, milyen tulajdonságokkal kell rendelkeznie egy minden igényt kielégítő, üzleti alkalmazások egyszerű és gyors létrehozására alkalmas ESB-nek!

Az ESB olyan infrastruktúrát biztosít, amely megszünteti a közvetlen összeköttetést a szolgáltatások igénybevevői, és szolgáltatók között. Az igénybevevők a buszt érik el, és nem a szolgáltatót, amely az éppen szükséges szolgáltatást biztosítja. A busz további hasznos képességekkel rendelkezhet: különböző biztonsági protokollokat alkalmazhat, és biztosíthatja az üzenetek megérkezését ahelyett, hogy ezeket az alkalmazásokban kellene implementálni. A szolgáltatások ilyetén kezelése növeli a SOA rugalmasságát, és kezelhetőségét.

Bár a szolgáltatás használóját és a szolgáltatót szétválasztja, a különböző technológiákat összeköti, lehetővé téve .Net, Java, Delphi és sok más technológiával készült webszolgáltatások számára, hogy egymással kommunikáljanak.

A különböző szolgáltatások által használt, egymással nem kompatibilis protokollokat, adatokat, kommunikációs mintákat azonos formára hozza, adattranszformációkat alkalmazva (amelyet alapjában az XML tesz lehetővé). Ennek másik fontos előnye, hogy a szolgáltatásoknak nem kell egymással teljesen kompatibilisnek lenniük a kommunikációban és

a felhasznált adatszerkezetekben. Az összeegyeztetés lehetősége ilymódon nagyobb teret hagy a szolgáltatásoknak, megszüntetve a köztük lévő függőségeket. A legnagyobb előnye az összeegyeztetés műveletnek, hogy a már meglévő szolgáltatásokat módosítások nélkül használhatjuk különböző követelményekkel rendelkező területeken. Ezzel a rugalmassággal, az összeegyeztetett szolgáltatásokkal lehetővé válik a szolgáltatások automatikusan működő üzleti folyamatokká szervezése.

Ahogy a szolgáltatásorientált architektúra, és a benne lévő szolgáltatások száma növekszik, úgy válik egyre bonyolultabbá a szolgáltatások menedzselése. A legfontosabb feladat a szolgáltatások installálása. Az ESB konfiguráció-vezérelt, ami azt jelenti, hogy kezdetben deklarálhatjuk a különböző szolgáltatásokat, és azok összeegyeztetésének módjait, és később ez a konfiguráció megváltoztatható, anélkül, hogy a szolgáltaításokat újra kellene fordítani, vagy telepíteni. Emellett az ESB rengeteg futási idejű szolgáltatást nyújt, amelyek segítik az architektúra felügyeletét, és a hibakeresést.

A szolgáltatások közötti üzenetek megbízható szállítása nem csak a garantált célbaérést jelenti, titkosítási igények is felmerülhetnek. A különböző szolgáltatások megvalósításakor felesleges ilyen titkosítási protokollokat implementálni, ha a szolgáltatások mögött van egy ESB, ami többek között a szolgáltatások között szállított üzenetek titkosítását is biztosítja.

Ahhoz, hogy a szolgáltatások megtalálják egymást, és az üzenetek eljussanak a megfelelő célszolgáltatáshoz, a szolgáltatások allat lévő architektúrának, vagyis az ESB-nek lehetővé kell tennie a szolgáltatások címzését és az üzenetek irányítását, vagyis a routing-ot. Léteznie kell egy üzleti szolgáltatások leírásait és elérhetőségét tartalmazó nyilvántartásnak (Service Routing Directory). Ez a nyilvántartás a webszolgáltatások esetén az UDDI nyilvántartó, amely a szolgáltatások dinamikus keresését, felderítését teszi lehetővé.

Az ESB lehetővé teszi összetett üzleti folyamatok definiálását erre a célra alkotott XML alapú leírónyelvek segítségével. Webszolgáltatások esetén ilyen lehetőségeket biztosító leírónyelv a BPEL4WS (Business Process Execution Language for Web Services).

Összefoglalva tehát az ESB használata a következő előnyökkel jár:

- Csökkenti az új üzleti folyamatok létrehozásának költségeit, mivel a korábbi, már létező szolgáltatások megfelelő összehangolásával felhasználhatjuk a már meglévő szoftvereket.
- Növeli a rendszer rugalmasságát, mivel csökkenti a komponensek, tehát a szolgáltatások közötti függőségek számát.
- Az üzenetek megbízhatóan kerülnek kiszállításra, hardveres vagy szoftveres meghibásodás esetén is.
- Szolgáltatások központi kezelésére alkalmas infrastruktúrát biztosít, annak ellenére, hogy maga a rendszer elosztott módon működik.

Az ESB fenti jellemzői a webszolgáltatásokhoz kapcsolódó specifikációk alapján a következő módon valósulnak meg: URL címzést használva, a létező HTTP és DNS infrastruktúra biztosítja a routing-ot és a webszolgáltatások elhelyezéskedésének átlátszóságát. SOAP/HTTP támogatja a kérdés/válasz üzenetküldési mintát. A HTTP transzport protokoll széleskörűen elérhető. A SOAP és a WSDL nyílt, implementáció független üzenetküldő és interfész leíró modellek.

3.5. BPEL4WS

A BPEL4WS leírónyelv a WSDL specifikációra épül. Legfontosabb eszköze a WSDL leírásokon alapuló webszolgáltatások közötti peer-to-peer üzenetváltások és interakciók leírása. A WSDL-hez hasonlóan az BPEL4WS is az absztrakt `portType` és `operation` interfészeket használja az üzleti folyamatok szervezésére a szolgáltatásokra hivatkozó konkrét referenciák helyett. A következő példa BPEL4WS dokumentum 4 fő részből áll:

- A `<partnerLinks>` elemen belül különböző linkeket azonosítunk, melyekben meghatározzuk a saját szerepkörünk és a webszolgáltatásét, amelyhez kapcsolódunk.
- A `<variables>` elemen belül definiáljuk azokat az adatváltozókat, amelyekben a beérkező adatokat kezeljük.

- Ezt követően a `<faultHandler>` elemen belül megadjuk a hibakezelés módját, a kommunikációk során előforduló különböző típusú hibákra azonosítóval hivatkozhatunk.

- A külső `<sequence>` elemen belül látható, hogy a szolgáltatás három fő lépésből áll: az első, hogy fogajuk a szolgáltatás kérést. Ez a `<receive>` elemen belül kerül specifikálásra, az utolsó lépésként specifikáljuk a választ a `<reply>` elemen belül. Ezek gyakorlatilag a szolgáltatás input és output műveleteinek képzelhetőek el. A két elem között történik a végrehajtás menetének megadása a `<flow>` elemen belül. A három újabb `<sequence>` elem három különböző lépést specifikál, amelyek újabb szolgáltatás indításokat tartalmaznak (`<invoke>` elemek). A három lépés közötti sorrend a `link` attribútum segítségével kerül definiálásra.

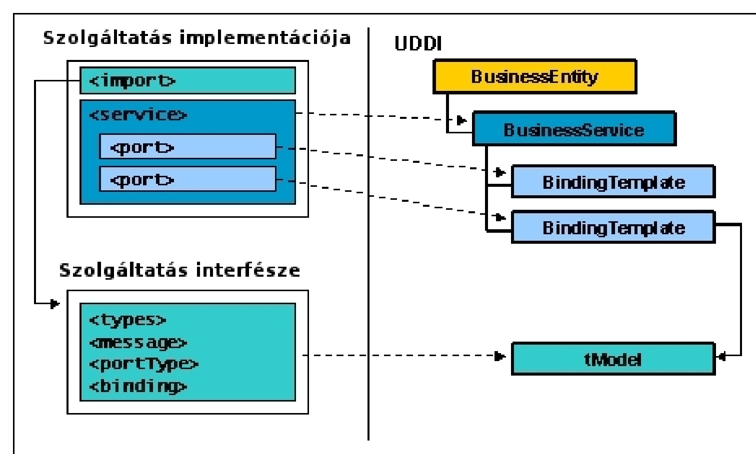
```
<process name="purchaseOrderProcess"
targetNamespace="http://acme.com/ws-bp/purchase"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
xmlns:lns="http://manufacturing.org/wsdl/purchase">
   <partnerLinks>
      <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT" myRole="purchaseService"/>
      <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"myRole="invoiceRequester"
          partnerRole="invoiceService"/>
      <partnerLink name="shipping" partnerLinkType="lns:shippingLT" myRole="shippingRequester"
          partnerRole="shippingService"/>
      <partnerLink name="scheduling" partnerLinkType="lns:schedulingLT"
          partnerRole="schedulingService"/>
      </partnerLinks>
   <variables>
      <variable name="PO" messageType="lns:POMessage"/>
      <variable name="Invoice" messageType="lns:InvMessage"/>
      <variable name="POFault" messageType="lns:orderFaultType"/>
      <variable name="shippingRequest" messageType="lns:shippingRequestMessage"/>
      <variable name="shippingInfo" messageType="lns:shippingInfoMessage"/>
      <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
   </variables>
   <faultHandlers>
      <catch faultName="lns:cannotCompleteOrder" faultVariable="POFault">
           <reply partnerLink="purchasing" portType="lns:purchaseOrderPT" operation="sendPurchaseOrder"
               variable="POFault" faultName="cannotCompleteOrder"/>
      </catch>
   </faultHandlers>

   <sequence>
      <receive partnerLink="purchasing" portType="lns:purchaseOrderPT" operation="sendPurchaseOrder"
         variable="PO">
      </receive>
      <flow>
         <links>
            <link name="ship-to-invoice"/>
            <link name="ship-to-scheduling"/>
         </links>
      <sequence>
         <assign>
            <copy>
               <from variable="PO" part="customerInfo"/>
               <to variable="shippingRequest" part="customerInfo"/>
            </copy>
         </assign>
```

```
      <invoke partnerLink="shipping" portType="lns:shippingPT" operation="requestShipping"
         inputVariable="shippingRequest" outputVariable="shippingInfo">
      <source linkName="ship-to-invoice"/>
      </invoke>
      <receive partnerLink="shipping" portType="lns:shippingCallbackPT" operation="sendSchedule"
           variable="shippingSchedule">
         <source linkName="ship-to-scheduling"/>
      </receive>
   </sequence>
   <sequence>
      <invoke partnerLink="invoicing" portType="lns:computePricePT" operation="initiatePriceCalculation"
         inputVariable="PO">
      </invoke>
      <invoke partnerLink="invoicing" portType="lns:computePricePT" operation="sendShippingPrice"
         inputVariable="shippingInfo">
         <target linkName="ship-to-invoice"/>
      </invoke>
      <receive partnerLink="invoicing" portType="lns:invoiceCallbackPT" operation="sendInvoice"
         variable="Invoice"/>
   </sequence>
   <sequence>
      <invoke partnerLink="scheduling" portType="lns:schedulingPT" operation="requestProductionScheduling"
         inputVariable="PO">
      </invoke>
      <invoke partnerLink="scheduling" portType="lns:schedulingPT" operation="sendShippingSchedule"
         inputVariable="shippingSchedule">
         <target linkName="ship-to-scheduling"/>
      </invoke>
   </sequence>
   </flow>
  <reply partnerLink="purchasing" portType="lns:purchaseOrderPT" operation="sendPurchaseOrder"
     variable="Invoice"/>
  </sequence>
</process>
```

A BPEL4WS képezi majd az alapját a későbbiekben bemutatott, szolgáltatások együttműködését modellező formális kalkulusoknak is.

3.6. Összefoglalás



3. ábra. A WSDL és az UDDI kapcsolata

A SOAP tehát egy olyan XML alapú protokoll, amely lehetővé teszi különböző adatok

szöveges formalizását, valamint távoli metódushívások leírását. Mindezek mellett olyan információkat is tartalmaz, amelyek a cél csomópontban a saját feldolgozását vezérlik. A WSDL a webszolgáltatások nyilvános interfészeinek leírását tartalmazza, szintén XML alapú leírónyelven. A SOAP üzenetekben található távoli metódushívások a WSDL állományokban definiált interfészek metódusaira vonatkoznak. A SOAP üzeneteket ilymódon a célszolgáltatás WSDL állománya alapján hozzák létre megfelelő automatikus eszközök. Az UDDI nyilvántartók cégek, szolgáltatások, és ha webszolgáltatásokról van szó, akkor azok interfészeinek specifikációt tartalmazza. Az UDDI kapcsolata a WSDL-el ott jelenik meg, hogy a WSDL dokumentumban a `<service>` elemen belül definiált szolgáltatás megfelel az UDDI `businessService` adatszerkezetének, a `<port>`-ok definiálása a `bindingTemplate` adatszerkezetnek, valamint a `<type>`, `<messages>`, `<binding>`, `<portType>` elemek a `tModell` adatszerkezetnek, amely azokat az technikai specifikációkat tartalmazza, amelyeket egy szolgáltatás megvalósít. Az ESB biztosítja azokat a szolgáltatásokat, amelyek az üzenetek biztonságos célbajutását garantálják, s ezenkívül az ESB biztosítja azt az egységes felületet, amelyet a különböző gyártók szolgáltatásai elérhetnek. A BPEL4WS az ESB-hez kapcsolódó specifikáció, amely a webszolgáltatásokból üzleti folyamatok definiálását teszi lehetővé, szintén XML alapon. Az ESB, és a BPEL azok az szabványok, amelyek az individuális webszolgáltatásokat üzleti folyamatokká kapcsolják össze, s ezzel létrehozzák a webszolgáltatások architektúráját.

4. KONCEPCIÓK

A webszolgáltatások egymással összefüggű, összetett rendszert alkotnak. Ahhoz, hogy megfelelő módon használhatóak legyenek, számos szabvány definiálására volt szűkség: a SOAP üzenetek szabványa, az interfészek leírását lehetővé tevő WSDL, vagy a publikálást megvalósító UDDI specifikáció. Ezek, és még nagyon sok további szabvány alkotja azt az architektúrát, amely lehetővé teszi webszolgáltatások egymáshoz kapcsolódó rendszerének a használatát. Ha a webszolgáltatások kapcsán felmerült problémákat és megoldásaikat általánosítjuk, megfogalmazhatjuk a szolgáltatásorientált architektúra definícióját:

Szolgáltatásorientált architektúra: (SOA) Az eljárások, gyakorlatok, keretrendszerek, amelyek lehetővé teszik, hogy az alkalmazások funkcionalitása biztosítható, és elérhető legyen a szolgáltatást igénylők számára. A szolgáltatások felhasználhatók, publikálhatók, felfedezhetők, és absztraktak, minden implementációtól függetlenek, az interfészeknek egy standard formáját biztosítva. (CBDI)

A szolgáltatásorientált architektúra fő koncepciói tehát:

1. **szolgáltatás:** jól definiált, önálló üzleti függvény, mely teljesen független a felhasználási környezetétől, és jól definiálható interfészen keresztül érhető el.

2. **szolgáltatásleírás**: egy szolgáltatás tulajdonságainak és interfészeinek valamilyen szabvány szerinti leírása.

3. **hirdetés, és felderítés**: a szolgáltatás szabványos leírását publikus közé kell tenni, hogy azokat a szolgáltatás későbbi felhasználói megtalálhassák.

A szolgáltatásorientált architektúra víziója, hogy a jövőben egy vállalat által igényelt IT szolgáltatásokat nagymértékben automatikusan működő szoftverek a vállalat saját, belső szolgáltatásaiból, és más vállalatok által közétett publikus szolgáltatásokból, mint építőelemekből, az elkészített specifikáció alapján automatikusan összeállítják, ezzel az üzleti szoftverek fejlesztését a deklaratív felé mozdítva el: megfelelő leírónyelven leírjuk, hogy milyen szolgáltatásokra van szűkségünk, és az automatikus eszközök minimális emberi beavatkozással összeállítják.

A szolgáltatásorientált architektúra sok szempontból nagyon hasonlít az objektum-technológiához, és a komponens alapú programozáshoz: Akárcsak az objektumok, és komponensek, a szolgáltatások is építőkockákat reprezentálnak, amelyekből felépíthetjük a számunkra éppen megfelelő szoftverrendszert. A szolgáltatás egy olyan építő elem, amely egyben kezeli az információt, és a viselkedést, elrejti a belső felépítését, és egy viszonylag egyszerű interfészt állít rendelkezésünkre, amelyen keresztül használhatjuk. Amíg az objektumok egy létező dolgot szimbolizálnak, addig a szolgáltatások egy való világbeli cselekvést reprezentálnak, így az objektumorientált tervezés, és a szolgáltatásorientált architektúra tökéletesen kiegészítik egymást.

A szolgáltatások használatával valódi megfeleltetés jön létre az üzletmenet, és az információ technológiai implementáció között. Évekkel ezelőtt az üzletemberek nem igazán értették az IT architektúrát, az nem felelt meg kellőképpen az üzletmenetnek. Jól megtervezett szolgáltatásokkal azonban radikálisan gyorsíthatjuk az üzlet menetek, és az információs folyamatok közötti konvergenciát. A jól megtervezett szolgáltatások az üzleti szolgáltatásokhoz hasonló kezelhetőséget biztosítanak számunkra. Ha egy szolgáltatás függetlenítve van az implementációjától, annak felhasználására sok különböző alternatív lehetőség adódik.

Egy szolgáltatás létrehozásának menete:

1. tradicionális tervezés
2. programozás
3. ha szűkséges, integrálás más, már létező szolgáltatásokhoz

A szolgáltatásorientált architektúra lényege tehát az, hogy képesek legyünk a szolgáltatásokat mint első rendű termékeket kezelni. A szolgáltatás legyen a kapcsolat a szolgáltató és a szolgáltatás igénybevevője között. Ehhez szűkségünk van egy szolgáltatás architektúrára, amely biztosítja, hogy a szolgáltatások ne süllyedjenek az interfészek szintjére, hanem saját identitásuk legyen, és függetlenül, és halmazokban is kezelhetőek legyenek. A CBDI megtervezte a Business Service Bus koncepciót (Enterprise Service

Bus), amely pontosan kielégíti ezeket az igényeket. Ahelyett, hogy a fejlesztőkre maradna a szolgáltatások megkeresésének, kiválasztásának és azonos környezetbe helyezésének feladata, a BSB (ESB) válik kiinduló pontul, amely segíti őket, hogy szolgáltatásoknak egy olyan koherens halmazát állítsák össze, amely pontosan a megoldandó feladat elemeit tartalmazza. Tehát a BSB célja, hogy a közös specifikációk, eljárások, ne az egyes szolgáltatások szintjén létezzenek, hanem a BSB szinten legyenek biztosítva.

5. PROGRAMTERVEZÉS

A szolgáltatásorientált koncepció és szemlélet megjelenése új programtervezési megközelítések megszületését vonta maga után együtt. A szolgáltatásorientált tervezéssel az üzletmenethez sokkal jobban illeszkedő szoftvereket készíthetünk, olymódon, hogy a tervezést nem a technológiai megvalósítás aspektusainak vizsgálatával kezdjük, hanem fentről lefelé haladó tervezéssel, a valós üzletmenet dekompozíciójával és IT-re történő átültetésével végezzük. Ennek megfelelően az első és legfontosabb lépés, ami a korábbi programtervezési módszereknek nem volt része, az üzleti folyamatokban résztvevő szolgáltatások azonosítása, és ezzel az üzleti folyamat dekompozíciója jól definiált szolgáltatásokra szolgáltatásokra. Az egyes szolgáltatások tervezése eztuán történhet a hagyományosnak mondható objektumorientált módszerek segítségével.

5.1. Szolgáltatások azonosítása

Mi legyen egy szolgáltatás? Melyek azok az üzleti tevékenységek, funkciók, amelyeket szolgáltatásoknak nevezhetünk, és ezek közül melyek azok, amelyeket webszolgáltatásokként kell megvalósítanunk? Nagyon fontos, hogy a szolgáltatásokat pontosan azonosítsuk. Több megközelítésből is megvizsgálhatjuk az üzleti tevékenységet annak érdekében, hogy azonosítsuk a szolgáltatásokat. Kiindulhatunk a legritkábban használt funkcióktól, és azokat integrálhatjuk a gyakrabban használtakba. Egy másik, sokkal absztraktabb megközelítés a kommunikációs minták vizsgálatán alapszik. Ennek bemutatásához első lépésként megvizsgáljuk, hogy az egy szervezetben dolgozó emberek hogyan kommunikálnak egymással, miközben az üzlethez kötődő tevékenységeiket végzik. Ezek után bevezetjük az üzleti tranzakció fogalmát, mint kommunikációs mintát, majd megvizsgáljuk, hogy az üzleti tranzakciókból hogyan épül fel egy üzleti tevékenység, és az üzleti tranzakciók hogyan teszik lehetővé az üzleti szerepek azonosítását. Minden üzleti szerep egy jól definiált szolgáltatást jelent más üzleti szerepeket betöltő entitások felé, legyen az vállalaton belüli, vagy kívüli.

Szervezet. Szervezetnek nevezzük embereknek közös céljaik elérése érdekében együttműködő csoportját. Hogy növeljék az együttműködés hatásfokát, szerepeket határoznak

meg, és osztanak szét egymás között, és a különböző szerepekhez tartozó munkák koordinálására közös megegyezés alapján eljárásokat, szabályokat dolgoznak ki. (Akárcsak a történelmi manufaktúrákban amelyek az egyedi, céhes termelést tették hatékonyabbá.)

Üzleti tevékenység. Üzleti tevékenységnek nevezzük valamely szervezeti szerepet betöltő entitás által végrehajtott cselekvések rendezett sorozatát. Minden cselekvés tovább részcselekvésekre bontható.

Egy személy több szerepet is betölthet egy szervezeten belül. Minden szerephez tartozik egy elemi hatáskör és felelősség, amellyel a személynek rendelkeznie kell, hogy egy adott termelési tevékenységet végrehajtson. A gyakorlatban a szerepek általában nem feleltethetők meg közvetlenül a szervezeti funkcióknak.
Az üzleti szerepeket betöltő entitások két különböző tevékenységet végezhetnek:

- **Termelő tevékenység:** A szervezet környezetéhez kötődő javak létrehozásához, vagy szolgáltatások nyújtásához kapcsolódnak.
- **Koordinációs tevékenység:** Termelő tevékenységek indításához, és befejezéséhez kapcsolódnak. Koordinációs tevékenységek végrehajtásakor az emberek kötelezettségeket vállalnak egymás irányába, az egyes termelő tevékenységek végrehajtásával kapcsolatban.

Egy termelő tevékenység sikeres végrehajtása termelési tényt jelent, s hasonlóan egy sikeres koordinációs tevékenység végrehajtása koordinációs tényként fogható fel (egy kötelezettség születése, vagy teljesítése).

Egy koordinációs tevékenységet mindig egy szerepkör indít (a kezdeményező), és mindig egy másik szerepkör fogad (a címzett). A kommunikáció mindig üzenetcserével történik. Kommunikálva az emberek próbálják befolyásolni egymás viselkedését. A kommunikációban használt üzenetek legfontosabb tulajdonsága a formája, jelentése és érvényességi ideje. Az üzeneteket ezen tulajdonságaiktól függően 5 fő csoportba sorolhatjuk:

- állítás (a beszélő kijelent valamilyen tényt)

- utasítás (a beszélő a hallgatót valamilyen cselekvés elvégzésére készteti)
- kötelezettség (a beszélő valamit tenni fog a jövőben, és ezért felelősséget is vállal)
- deklaráció (bevezet valamilyen új fogalmat)
- kifejezés (kifejezi a beszélő érzéseit, gondolatait)

Egy kommunikációt akkor tekintünk sikeresen befejezettnek, ha a beszélő mentális állapota, és a hallgató mentális állapota megfelel egymásnak. A megértés elérése érdekében a hallgató magyarázatot kérhet a beszélőtől. A kommunikáció csak akkor teljesül, ha a hallgató megerősíti, hogy értette a beszélőt. A korábbi emberek közötti kommunikációs modellel ellentétben (ahol az kommunikációt egymástól független üzenetváltásokként modelleztük) tehát a beszélgetés valójában kommunikációs szerepek váltogatásának sorozata két ember között. A termelő célú beszélgetések mindig arról szólnak, hogy valaki tegyen valamit. Az informatív beszélgetés célja, hogy már létező tudást osszunk meg valakivel. A koordinációs tevékenységek kommunikációs tevékenységek, amelyeket a koordinációs tevékenység indítójától annak címzettjéhez irányítunk. Mivel a koordinációs tevékenységek a következő szabványos módon adhatók meg:

```
<kezdeményező>: <indíttatás>: <címzett>: <tény>: <érvényességi idő>
```

például:

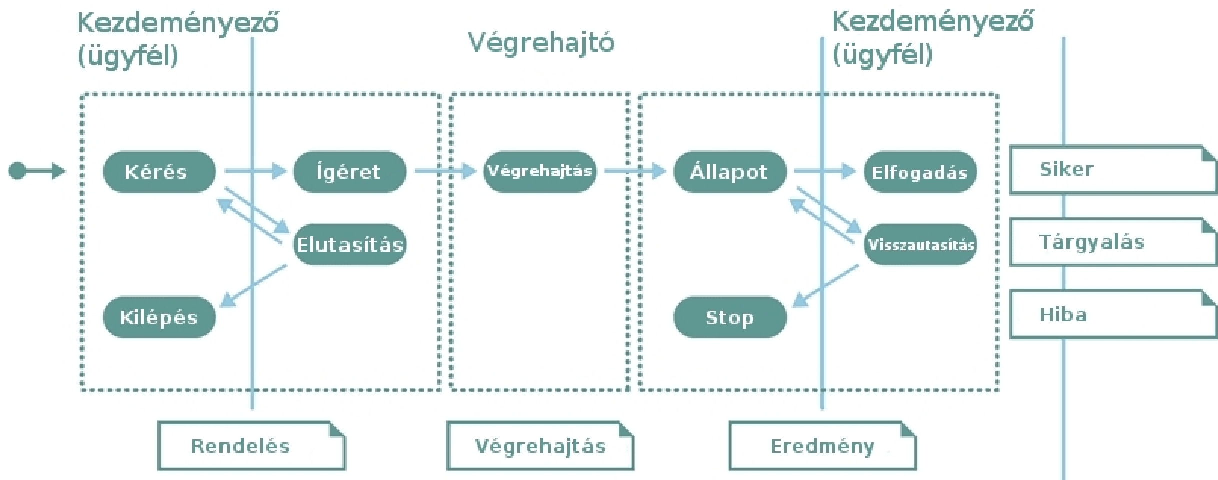
```
Sanyi: kérés: Eszter: hitel-limit megnövelése 100000HUF-ra: holnap
```

Egy koordinációs tevékenység mindig egy termelő tevékenységhez kapcsolódik, pontosabban egy termelési tényhez, amelynek létrehozása a termelő tevékenység sikeres végrehajtását jelenti.

Az üzleti tranzakció minta. Amikor egymás cselekvését koordinálni szándékozó emberek közötti beszélgetéseket vizsgálunk, a kommunikációs tevékenységek, amelyek a beszélgetést alkotják, ugyanazt a mintát követik, az üzleti környezettől függetlenül. Ezt a mintát nevezzük üzleti tranzakció kommunikációs mintának.

> Elsőként megállapodnak arról, hogy mi a cél. Aztán megtörténik ennek végrehajtása a termékek világában, azonban amíg ezt mindkét fél el nem fogadja,

nem ér véget az előállító kötelezettsége. Ez a 3 lépés alkotja az üzleti tranzakciót.



4. ábra. Az üzleti tranzakció kommunikációs minta

Az üzleti tranzakció két szerepet vezet be: az ügyfél szerep indítja az üzleti tranzakciót, míg a végrehajtó szerep végrehajtja a kívánt cselekvést. Az első, megrendelő fázisban egy termelő beszélgetés zajlik, ami az ügyfél kérésével indul, és az kiszolgáló ígéretével fejeződik be. A végrehajtási fázisban a kiszolgáló végrehajtja a kívánt cselekvést. A tranzakció szintén egy termelő beszélgetéssel zárul az eredmény fázisban, amely egy állítással kezdődik, mely szerint a kiszolgáltó teljesítette a kérést, és az ügyfél elfogadásával zárul.

Az üzleti tranzakció mindhárom fázisában újabb üzleti tranzakciók indulhatnak, melyek sikeres befejeződése nélkül nem folytatódhat az eredeti tranzakció. Ily módon egymásba ágyazott és egymás után láncolt tranzakciók összetett struktúráját lehet megalkotni. Minden üzleti tranzakcióhoz tartozik egy jól definiált üzleti szerep, amely a tranzakciót végrehajtó személy elemi hatáskörét reprezentálja. Minden üzleti tranzakció végrehajtásáért felelős üzleti szerep egy elemi és jól definiált üzleti szolgáltatást nyújt, és kapcsolatban állhat egy, vagy több másik elemi üzleti szolgáltatással, amelyet más üzleti szerepek nyújtanak.

Most nézzük meg, mit csinálnak az dolgozók a koordinációs tevékenységek között. Minden üzleti szerephez tartozik egy munka lista, amely prioritással rendelkező bejegyzéseket

tartalmaz azokhoz a koordinációs tényekhez, amelyekre a szerepnek válaszolnia kell. Hogy a koordinációs tény munkában végződik-e a végrehajtó, vagy az indító számára, az a koordinációs tény jellegétől függ. A eseménykezelő szabályok meghatározzák, hogy egy adott üzleti szerepnek mit kell válaszolnia egy koordinációs tényre. Bár a szabályok leírják az üzletmenetet, a szerepek felelősek maradnak a döntéshozatalért, akkor is, ha ez az eljárások figyelmen kívül hagyását jelentik! Ez az alapvető oka annak, amiért nem lehet az eseménykezelő szabályok végrehajtását teljesen automatizálni. Egy üzleti szerepet ellátó dolgozó mindennapi munkája úgy telik, hogy kiválasztja a legnagyobb prioritású munkát a listájából, és végrehajtja az alkalmazható szabályt. Egy tevékenység szabály végrehajtása mindig egy, vagy több koordinációs szerep igénybevételével végződik.

Egy üzleti tevékenységben lezajló eseményeket három absztrakciós szinten szemlélhetjük: az üzleti, az információs és az infrastrukturális szinten. Eddig az üzleti szintre fókuszáltunk, ezen a szinten a szerepeket szociális szereplők látják el. A szociális szereplők felhasználják a valós szereplők szolgáltatásait, hogy a tudást tárolják, és üzeneteket váltsanak. Ezek logikai műveleteket hajtanak végre az információkon, de nem tudják, hogy az valójában mit elent. Ők az információs szinten működnek. Láthatjuk, hogy milyen információkat tárolnak, de hogy hogyan, azt nem. A valós szereplők a formális, vagy fizikai szereplőktől függnek, akik előállítják, közéteszik, tárolják, másolják, és megsemmisítik az adatokat, vagy dokumentumokat, amelyek a tudást tárolják. Az üzleti, információs és infrastrukturális szineteken működő szereplők 3 féle szolgáltatást biztosítanak, amelyek egymásra rétegződnek. A három réteg szolgáltatásainak absztrakciós szintja a teljesen manuálistól a teljesen automatizáltig terjedhet. Míg az emberek mindhárom szinten betölthetnek szerepeket, addig a gépek csak az infrastrukturális, és információs szinten alkalmazhatók. Egy emberi szereplő, aki egy adott üzleti szerepben dolgozik, a végrehajtandó üzleti tevékenység részleteit kioszthatja gépeknek, de a végén mindig ő lesz a felelős a végrehajtásért.

Az üzletmenet üzleti, információs és infrastrukturális szolgáltatásokra bontása, és a köztük lévő kapcsolatok definiálása független bármilyen technológiától, és ezért az előállt szolgáltatások is technológia-függetlenek, s ez lehetővé teszi, hogy bármely szolgáltatást bármi-

lyen technológiával implementálhassunk, és azt bármikor kicserélhessük akár egy emberi szereplőre.

Tehát egy adott üzletmenet feltérképezésekor az első lépés az üzletben lezajló kommunikációk vizsgálta. Ez alapján meghatározzuk az alapvető üzleti tranzakciókat, amelyekből az üzletmenet felépül. Kialakítjuk a tranzakciók közötti összetett kapcsolatokat. A tranzakciók alapján meghatározzuk az üzleti szerepeket, amelyeket absztrakt, technológia független szolgáltatásoknak tekinthetünk.

Ha az üzleti folyamatokat üzleti tranzakciók egymásból kiinduló hierarchiájaként írjuk le, egyúttal a tranzakciókat végrahajtó üzleti szolgáltatások láncolatát modellezzük. Miután sikerült azonosítani a fő szolgáltatásokat, azok megvalósítása történhet a bevezetésben tárgyalt objektumorientált módszerek segítségével, szem előtt tartava természetesen a szolgáltatás jelleget, tehát a megvalósításhoz választott technológia a SOAP, WSDL, UDDI specifikációkon alapuló webszolgáltatások kell hogy legyen.

Példa.

Tekintsünk egy okmányirodában lejátszódó szokványos kommunikációt:

- Miben segíthetek?

- Útlevelet szeretnék csináltatni.

- Illetékbélyeget vásárolt?

- Igen.

- Személyigazolványa itt van?

- Igen.

- Megkérem fáradjon az utolsó ablakhoz, ahol a kollega készít egy fényképet.

- Rendben.

Az Ügyfél visszatér.

- Kérem töltse ki a személyi adatait tartalmazó űrlapot.

- Tessék, kitöltöttem.

- Köszönöm.

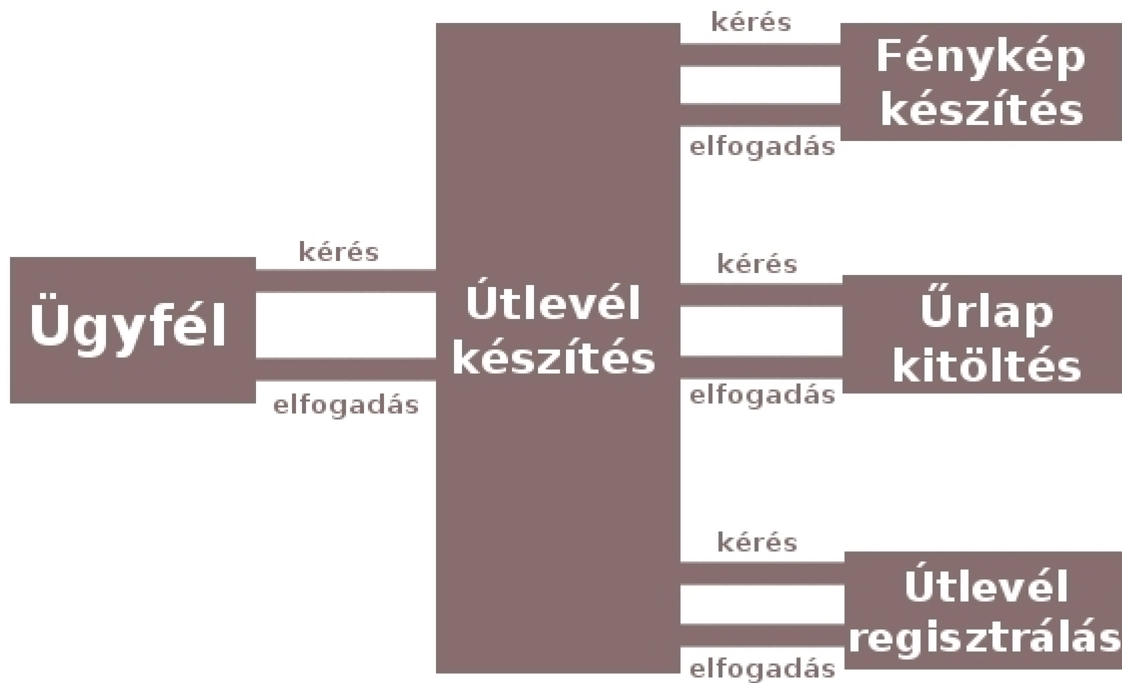
- Az útlevele néhány percen belül elkészül.

- Köszönöm.

- Tessék itt az útlevele.

- Köszönöm.

Formalizálva:



5. ábra. Az útlevél készítés folyamatának üzleti tranzakciói

termelő tevékenység: `Ügyfél: kér: Ügyintéző: Útlevelet készíteni.`

– informatív tevékenység: `Ügyintéző: kér: Ügyfél: Illetékbélyeg.`

– informatív tevékenység: `Ügyintéző: kér: Ügyfél: Személyigazolvány.`

– termelő tevékenység: `Ügyintéző: kér: Ügyfél: fényképezős ablakhoz menni.`

– – ...

– termelő tevékenység: `Ügyintéző: kér: Ügyfél: Űrlap kitöltése.`

– – ...

termelő tevékenység: `Ügyintéző: ígér: Ügyfél: Néhány percen belül készen lesz.`

– termelő tevékenység: `Ügyintéző: kér: Központi szerver: útlevél regisztrálása.`

– – ...

termelő tevékenység: `Ügyintéző: állít: Ügyfél: Készen van.`

termelő tevékenység: `Ügyfél: elfogad: Ügyintéző: Köszönöm.`

A fenti példában a külső szintű kommunikációk tartoznak az ‘útlevélLétrehozás’ üzleti tranzakcióhoz, míg az belsőbb szintűek (–) közül a termelő tevékenységek további tranzakciók indítását jelentik. Tehát az ‘útlevélLétrehozás’ üzleti tranzakció kapcsolatban áll, pontosabban rendezett sorban a következő tranzakciók beágyazott végrehajtását jelenti: ‘fényképKészítés’; ‘űrlapKitöltés’; ‘útlevélRegisztrálás’. Ez azt jelenti, hogy az ‘útlevélLétrehozás’ tranzakció (szerepkör) által definiált szolgáltatás igénybe veszi a ‘fényképKészítés’, ‘űrlapKitöltés’, és ‘útlevélRegisztrálás’ szolgáltatásokat. A ‘fényképKészítés’ lehet automatizált, vagy ember által végrehajtott. Az ‘űrlapKitöltés’ papíron történik, ezért mindenképpen ember által végrehajtott, míg az ‘útlevélRegisztrálás’ lehet a megfelelő kormányhivatal számítógépes szolgáltatása.

6. SZÁMÍTÁSELMÉLETI HÁTTÉR

A programozási paradigmák fejlődésével párhuzamosan alakulnak ki azok a számítástudományi modellek, amelyekkel a különböző paradigmák koncepcióit és az azt megvalósító programozási nyelveket modellezni lehet. Természetesen mindent lehet modellezni a Turing-gépek vagy az azzal azonos számítási erejű λ -kalkulus eszközeivel. Turing-gépekkel tulajdonképpen direkt módon minden leírható, ami a számíógépben történik, a λ -kalkulus önmagában a tisztán funkcionális nyelvek számítástudományi háttere, azonban alkalmas kiterjesztésével leírhatóak az eljárásorientált és objektum-orienált nyelvek is. Ez azért előnyös, mert ha sikerül elkészíteni a paradigma matematikai - számítástudományi modelljét, akkor meg lehet vizsgálni, hogy a programozási nyelvek közül melyik milyen szinten tartalmazza a paradigma eszközeit. Az utóbbi években nagyon felgyorsult a fejlődés: alig néhány évvel azután, hogy megfogalmazták a komponensorientált programozás lényegét, és megvalósították az első ilyen rendszereket, már újabb szemlélet jelent meg, mégpedig a szolgáltatásorientált programozás. Minél nagyobb a modellek absztrakciós szintje, annál nehezebb megfelelő matematikai modellt építeni mögéjük. Mivel a szolgáltatásorientált szemlélet még új az informatikában, ezért csak most kezdenek megjelenni az azokat leíró formális rendszerek.

6.1. Service Centered Calculus

A kalkulus az Cook és Misra által alkotott Orc (, szolgáltatások rendezésére (Orchestration) létrehozott) nyelv alapján készített formális kalkulus. Az Orc programozási modellt egyszerűsége miatt választották az SCC alapjául, mivel a benne lévő három alapvető, szolgáltatások kompozíciójára szolgáló operátorral (szekvenciális, szimmetrikus párhuzamos, asszimmetrikus párhuzamos) bármilyen szolgáltatás együttműködési minta megvalósítható.

Az SCC kalkulus a szolgáltatások definiálását, meghívását és a kétirányú kommunikációt alkalmazó munkameneteket hivatott modellezni. Az SCC-ben a szolgáltatások bizonyos fajta függvénynek tekinthetők, amelyeket a kliensek meghívhatnak. A szolgáltatások definíciója $s \Rightarrow (x)P$, ahol s a szolgáltatás neve, x a formális paraméter, és P a szol-

gáltatás egy implementációja. A szolgáltatás hívások formája a következő: $s(x)P \Leftarrow Q$, ahol minden egyes, Q által előállított értékre meghívódik az s szolgáltatás egy példánya, ezt követően x-et helyettesíti a hívás aktuális paramétere, és az s szolgáltatás befejeződése után a hívó oldalon lefut a P folyamat, melyet a hívás visszatérése triggerel. Például:

$succ \Rightarrow (x)x + 1$

$s\{(x)(y)\ return\ y\} \Leftarrow 5$

A fenti *succ* szolgáltatás megnöveli a paramétere értékét 1-el. A szolgáltatás befejeződésénél a hívási környezetben az x helyettesítődik 5-el, és amit a szolgáltatás visszaad (y), az lesz a szolgáltatáshívás értéke.

Egy szolgáltatáshívás minden esetben új munkamenet indítását jelenti. Az új munkameneteket két új azonosítóval jelöljük a szerver és kliens oldalon: r és $\overline{r}$. Például a fenti hívás a következő munkamenetben jelenik meg:

$(vr)(r \triangleright 5 + 1 \mid \overline{r} \triangleright (y)return\ y)$ A munkamenetek definiálása azért szűkséges, hogy ne csak az egyes szolgáltatásokat, hanem különböző munkamenetekhez tartozó, különböző paraméterekkel meghívott szolgáltatáspéldányokat tudjuk modellezni.

6.2. Service Oriented Computing Kernel

A SOCK egy három rétegből álló kalkulus, melynek célja a szolgáltatások viselkedésének, deklarációjának és kompzíciójának modellezése. A viselkedés a szolgáltatás, menetére utal, a deklaráció a különböző futási módok megadását jelenti, míg a kompozíciós rétegben a önálló szolgáltatásokból szolgáltatások rendszerét hozhatjuk létre. A kalkulus szemantikája a címkézett átírási rendszerek kifejezéseivel van megadva.

A legalső szintű réteg a *service behaviour calculus*. Ebben a rétegben definiálhatjuk a szolgáltatás műveleteit, valamint minden művelethez rendelhetünk egy modalitást. Ezek a modalitások a webszolgáltatások jellemző kommunikációs mintáinak felelnek meg:

Bemeneti műveletek, amelyek szolgáltatás funkcionalitást látnak el:

- kérés fogadása
- kérés-válasz: kérés fogadása, amely egy válasz üzenet küldését vonja maga után

Kimeneti műveletek, amelyek szolgáltatás funkcionalitást igényelnek:

- kérés küldése
- kérésre-válasz: kérés üzenet küldése, amely egy válasz fogadását maga után

Formálisan egy folyamat jellegét, viselkedését a következő szabálynak megfelelően adhatjuk meg:

$$P, Q ::= 0 \mid \overline{\epsilon} \mid \epsilon \mid x := e \mid \chi?P : Q \mid P; P \mid P|P \mid \sum_{i \in W}^{+} \epsilon_i; P_i \mid \chi \rightleftharpoons P$$

$$\epsilon ::= s \mid o(\vec{x}) \mid o_r(\vec{x}, \vec{y}, P)$$

$$\overline{\epsilon} ::= \overline{s} \mid \overline{o}k(\vec{x}) \mid \overline{o_r}k(\vec{x}, \vec{y})$$

A definíció szerint tehát a P, Q folyamatok lehetnek rendre: a 0 a nullfolyamat. Az ϵ rendre a bemeneti műveleteket: jelzés fogadása, egyirányú (kérés) üzenet fogadása, és kérés fogadása majd válaszadás kommunikációs műveleteket jelenti. A $\overline{\epsilon}$ rendre a jelzés, figyelmeztetés küldés, vagy kérésre adott válasz műveleteket jelentik. Az $x := e$ a szokásos értékadás művelet, míg azt követi az elágaztató utasítás, ahol χ a feltétel, majd a folyamatok szekvenciális $P; Q$ és párhuzamos $P|Q$ kompozíciója. A $\sum_{i \in W}^{+} \epsilon_i; P_i$ az ϵ-tól függő nemdeterminisztikus választást, míg az utolsó lehetőség $\chi \rightleftharpoons P$ az iterációt jelenti.

Nézzük meg a következő példát, melyben egy olyan szolgáltatás viselkesését írjuk le a *servicebehaviourcalculus* eszközeivel, amely egy futball meccs eredményét tartja nyilván:

$MatchMonitor ::= update(team); (team == TA)?scoreTA := scoreTA+1 : scoreTB := scoreTB + 1$

$+$

$cres(\langle\rangle, \langle scoreTA, scoreTB\rangle, 0)$

$+$

$reset(\langle\rangle); scoreTA := 0; scoreTB := 0$

Látható, hogy a szolgáltatásnak 3 művelete van, az *update* művelet egyetlen paramétere egy csapatnév. A művelet a csapatnévhez tartozó pontszámot megnöveli eggyel. A második művelet a *cres*, amellyel az eredményt lehet lekérdezni, míg a harmadik, a *reset*, amellyel alapállapotba lehet állítani a számlálókat, s ezzel a szolgáltatást. Természetesen a formális szemantikának megfelelően minden egyes művelethez meg lehet adni egy levezetési szabályt. A levezetési szabályok, és formális behelyettesítési műveletek segítségével aztán formálisan végre tudjuk hajtani, ki tudjuk próbálni az általunk definiált szolgáltatásokat.

A *service engine calculus* a fenti, *service behaviour calculus* eszközein felül bevezeti az state (állapot) és a correlation set (kapcsolódó változók halmaza) koncepciókat. Ezek segítségével tényleges munkameneteket modellezhetünk, továbbá a *service engine calculus* eszközeivel adhatjuk meg azt is, hogy a különböző munkamenetek szekvenciálisan, avagy párhuzamosan fussanak. A *service engine calculus* szintaktikája:

$U ::= P_x | P_\bullet \qquad W ::= c \triangleright U \qquad D ::= !W | W^*$

A fenti jelölésekben a kiindulási P folyamat egy, a *service behaviour calculus* alapján definiált szolgáltatás viselkedés. A P_x jelentése, hogy a P-nek nincs perzisztens, tehát megmaradó állapota, míg a $P_\bullet$ jelentése, hogy perzisztens állapota van. A c a kapcsolatos változók halmazát jelenti, míg a $!W$ arra utal, hogy a W-n a műveletek párhuzamosan is végrehajthatók, ellentétben a W^* esettel, ami a műveletek szekvenciális végrehajtását jelenti. A *service engine calculus*-sal megadott definíció minden esetben egy teljesértékű szolgáltatás deklarációt jelent.

Az alábbi példában az előző $MatchMonitor$ szolgáltatás viselkedés alapján készítünk egy szolgáltatás deklarációt a service engine calculus eszközeivel:

$$MatchMonitorDec \ ::= \ \{\} \triangleright MatchMonitor^{*}_{\bullet}$$

A $\{\}\triangleright$ jelentése, hogy üres korrelációs halmaz tartozik a szolgáltatáshoz. A $*$ jelentése, hogy az állapota perzisztens, tehát megőrzi az állapotát, míg a $\bullet$ arra utal, hogy a szolgáltatáson munkameneteket szekvenciálisan kell végrehajtani.

A SOCK harmadik, legfelső szintű rétege a *service system calculus* pedig arra való, hogy a különböző *service engine*-ek kompozícióját, mint rendszert írhassuk le, és vizsgálhassuk vele.

7. PÉLDA

7.1. A feladat leírása

Egy autóalkatrészeket forgalmazó kereskedés bővíteni akarja elérési lehetőségeit 2 internetes szolgáltatással. Azért lehet előnyös ilyen szolgáltatások létrehozása, mert a későbbiekben bármilyen webáruház felveheti a beszállítói közé a céget a rendeléseket egyszerűen a webes interfészen keresztül egyenesen a cégnek továbbítva. (Természetesen ahhoz bonyolultabb, több műveletet biztosító szolgáltatásra van szűkség, most a példa kedvéért egyszerűsítettem az esetet.) A szolgáltatások azonosítása már megtörtént, két webszolgáltatásra lenne szükségük. Az első kap egy rendelést ami egy termék azonosítót és egy mennyiséget jelent, a megrendelő azonosítójával. Amennyiben nincs a raktáron, a 0 numerikus értéket adja vissza. Ha kevesebb van raktáron, mint amennyire szűkség lenne, akkor azt a számot adja vissza, ami raktáron van. Ha van elég akkor a rendelési mennyiséget. Közben csökkenti a raktár adatbázisban a készletet. A másik szolgáltatás a már leadott rendelések adatbázisából lekérdezi, hogy egy adott azonosítójú rendelésnek mi az állapota, azaz hol tart a kiszállításban. Az adatbázisok más szoftverekkel is elérhetőek.

7.2. WSrendeles szolgáltatás Visual Studio fejlesztőkörnyezettel C# nyelven

A *File* menü *New* menüjében válasszuk a *web site* menüpontot. Ezután válasszuk ki az *ASP.NET service* pontot.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Data;
using System.Data.OleDb;

[WebService(Namespace = "http://peldaceg.hu/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    public Service () {
    }

/* a [WebMethod} attribútummal rendelkező függények szolgálnak majd a webszolgáltatás
interészeként. */

    [WebMethod]
    public int rendeles(string rendelo, int mennyiseg, string termekKod) {
```

```
        int termMennyiseg=0;

        /* Microsoft Access adatbázis elérési útja, amelyben a raktar adatait tartjuk nyilván. */
        string raktar = "Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=E:\\prog\\database\\raktar.mdb";

        /* A rendelések adatait tartalmazó Microsoft Access adatbázis elérési útja. */
        string rendeles = "Provider=Microsoft.Jet.OLEDB.4.0;" + "Data Source=E:\\prog\\database\\rendeles.mdb";

        /* Az adatbázisok megnyitása. */
        OleDbConnection adatbRaktar = new OleDbConnection(raktar);
        OleDbConnection adatbRendeles = new OleDbConnection(rendeles);

        /* SQL kérés, mellyel a raktáron lévő mennyiséget kérdezzük le vele. */
        string parancs = "SELECT mennyiseg FROM raktar WHERE termek_kod = " + termekKod;

        OleDbCommand adatbParancs = new OleDbCommand(parancs, adatbRaktar);
        OleDbDataReader adatok = null;

      /* Adatbázis megnyitása. */
        adatbRaktar.Open();

        /* Lekérdezés végrehajtása. */
        adatok = adatbParancs.ExecuteReader();

        /* Az eredményül kapott rekordhoz tartozó termékmennyiség kinyerése. */
        adatok.Read();
        termMennyiseg = adatok.GetInt32(0);

        if (termMennyiseg < mennyiseg)
        {
            /* A mennyiség nullázása, ha kevesebb van, mint amennyit a megrendelő szeretne. */
            parancs = "UPDATE raktar SET mennyiseg = 0 WHERE termek_kod = " + termekKod;

            /* A parancs végrehajtása. */
            adatbParancs = new OleDbCommand(parancs, adatbRaktar);
            adatbParancs.ExecuteNonQuery();

            mennyiseg = termMennyiseg;
        }
        else
        {   /* A raktárban lévő mennyiség csökkentése a megrendelés mennyiségével. */
            parancs = "UPDATE raktar SET mennyiseg = (SELECT mennyiseg FROM rendeles
            WHERE termek_kod = " + termekKod + " ) - " + mennyiseg + " WHERE termek_kod = " + termekKod;
            adatbParancs = new OleDbCommand(parancs, adatbRaktar);

            /* A parancs végrehajtása. */
            adatbParancs.ExecuteNonQuery();
        }

        if (mennyiseg > 0)
        {
            /* A megrendelés rögzítése. */
            parancs = "INSERT INTO rendeles VALUES megrendelo = " + rendelo +
            ", mennyiseg = " + mennyiseg + ", termek_kod = " + termekKod;

            /* A rögzítés végrehajtása. */
            adatbParancs = new OleDbCommand(parancs, adatbRendeles);
            adatbParancs.ExecuteNonQuery();
        }

        return mennyiseg;
    }

}
```

A kódban a `rendeles` függvény egy szokásos C# függvény, annyi különbséggel, hogy van egy `[WebMethod]` attribútuma, amivel jelezzük a futtató rendszernek, hogy a függvény a webszolgáltatás interészéhez tartozik. Egy lehetséges SOAP kérés:

```
POST /WSrendeles/Service.asmx HTTP/1.1
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <rendeles xmlns="http://peldaceg.org/">
      <rendelo>aaa123</rendelo>
      <mennyiseg>12</mennyiseg>
      <termekKod>bwtz76</termekKod>
    </rendeles>
  </soap12:Body>
</soap12:Envelope>
```

Megfigyelhető a SOAP kérésben a távoli metódushívás szerializációja, aholis a `rendeles` a metódus neve, míg a a `rendelo`, `termekkod`, és `mennyiseg` elemek a metódushívás aktuális paramétereit azonosítják. Ezek az elemek mind a `rendeles` elemben hivatkozott `http://peldaceg.org` XML névtérben vannak specifikálva. Egy lehetséges SOAP válasz:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <rendelesResponse xmlns="http://peldaceg.org/">
      <rendelesResult>12</rendelesResult>
    </rendelesResponse>
  </soap12:Body>
</soap12:Envelope>
```

A szolgáltatást leíró WSDL állomány:

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:mime="http://
schemas.xmlsoap.org/wsdl/mime/" xmlns:tns="http://tempuri.org/"
xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:soap12=
"http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" targetNamespace="http://peldaceg.org/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://peldaceg.org/">
      <s:element name="rendeles">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="rendelo" type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="mennyiseg" type="s:int" />
            <s:element minOccurs="0" maxOccurs="1" name="termekKod" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="rendelesResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="rendelesResult" type="s:int" />
          </s:sequence>
        </s:complexType>
```

```
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="rendelesSoapIn">
    <wsdl:part name="parameters" element="tns:rendeles" />
  </wsdl:message>
  <wsdl:message name="rendelesSoapOut">
    <wsdl:part name="parameters" element="tns:rendelesResponse" />
  </wsdl:message>
  <wsdl:portType name="ServiceSoap">
    <wsdl:operation name="rendeles">
      <wsdl:input message="tns:rendelesSoapIn" />
      <wsdl:output message="tns:rendelesSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ServiceSoap" type="tns:ServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="rendeles">
      <soap:operation soapAction="http://tempuri.org/rendeles" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="ServiceSoap12" type="tns:ServiceSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="rendeles">
      <soap12:operation soapAction="http://tempuri.org/rendeles" style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Service">
    <wsdl:port name="ServiceSoap" binding="tns:ServiceSoap">
      <soap:address location="http://localhost:3637/WSrendeles/Service.asmx" />
    </wsdl:port>
    <wsdl:port name="ServiceSoap12" binding="tns:ServiceSoap12">
      <soap12:address location="http://localhost:3637/WSrendeles/Service.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

7.3. WSholjar szolgáltatás létrehozása NetBeans 5.5 környezetben Java nyelven

NetBeans környezetben hasonló módon az *File* menü *New project* menüpontjának kiválasztásával indul a fejlesztés. A szolgáltatás kódjában fontos, hogy importáljuk a `WebMethod`, `WebParam`, `WebService` interfészeket:

```
package org.peldaceg.ws;

import javax.jws.WebMethod;
```

```
import javax.jws.WebParam;
import javax.jws.WebService;
import java.sql.*;

/**
 *
 * @author Kovács György
 */
@WebService()
public class WSholjar {
    /**
     * Web service operation
     */

    @WebMethod
    public String holjar(@WebParam(name = "rendelesKod") String rendelesKod) {

        try{
            /* adatbázis kapcsolat létesítése */
            Connection kapcsolat = DriverManager.getConnection("jdbc:odbc:rendeles");

            /* parancs objektum létrehozása */
            Statement parancs = kapcsolat.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
             ResultSet.CONCUR_READ_ONLY);

            /* lekérdezés végrehajtása a parancs objektum segítségével */
            ResultSet eredmeny = parancs.executeQuery("SELECT allapot FROM rendelesek
             WHERE rendeles_kod = "+rendelesKod);

            /* az eredmények feldolgozása */
            eredmeny.next();
            if(eredmeny.next())
                return eredmeny.getString("allapot");
            else
                return "nincs ilyen rendeles";

        }
        catch(Exception e){
            System.out.println("Hiba: "+e);
        }

        return "nincs ilyen rendeles";
    }

}
```

A `@WebService` külső interfész jelzi, hogy webszolgáltatást implementálunk, a `@WebMethod` külső interfész adja meg a szolgáltatás interfész metódusait, és a `@WebParam` külső interfésszel adhatjuk meg azokat a paramétereket, amelyeket a `@WebMethod` paraméterként kap majd a SOAP protokollon keresztül. A SOAP üzenetek generálására és feldolgozására általában a különböző rendszerek létrehoznak egy proxy osztályt, és a szoftverünk tulajdonképpen annak egy példányával kommunikál, és azon keresztül éri el a NetBeans és Visual Studio környezetkbe beépített ESB-t. A kliens alkalmlazásokban ugyanígy létrejön egy proxy osztály a WSDL dokumentum alapján, és ezen keresztül érhetjük el a webszolgáltatást Így Egy lehetséges SOAP kérés:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://ws.peldaceg.org/">
```

```
    <soapenv:Body>
        <ns1:holjar>
            <rendelesKod>aaa1234</rendelesKod>
        </ns1:holjar>
    </soapenv:Body>
</soapenv:Envelope>
```

És a lehetséges SOAP válasz:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://ws.peldaceg.org/">
    <soapenv:Body>
        <ns1:holjarResponse>
            <return>nincs ilyen rendeles</return>
        </ns1:holjarResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

A WSDL állományt a NetBeans 5.5 IDE automatikusan generálta. Ezt használhatjuk fel a szolgáltatás publikálására:

```
<?xml version="1.0" encoding="UTF-8"?><definitions xmlns="http://
schemas.xmlsoap.org/wsdl/" xmlns:tns="http://ws.peldaceg.org/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://
schemas.xmlsoap.org/wsdl/soap/" targetNamespace="http://ws.peldaceg.org/"
 name="WSholjarService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://ws.peldaceg.org/" schemaLocation="http://
      THUNDER:8080/CalculatorWSApplication/WSholjarService/
      __container$publishing$subctx/WEB-INF/wsdl/WSholjarService_schema1.xsd"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:soap12=
       "http://schemas.xmlsoap.org/wsdl/soap12/"/>
    </xsd:schema>
  </types>
  <message name="holjar">
    <part name="parameters" element="tns:holjar"/>
  </message>

  <message name="holjarResponse">
    <part name="parameters" element="tns:holjarResponse"/>
  </message>
  <portType name="WSholjar">
    <operation name="holjar">
      <input message="tns:holjar"/>
      <output message="tns:holjarResponse"/>
    </operation>
  </portType>

  <binding name="WSholjarPortBinding" type="tns:WSholjar">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="holjar">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>

      </output>
    </operation>
  </binding>
  <service name="WSholjarService">
    <port name="WSholjarPort" binding="tns:WSholjarPortBinding">
      <soap:address location="http://THUNDER:8080/CalculatorWSApplication/WSholjarService"
       xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
```

```
      xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"/>
    </port>
  </service>
</definitions>
```

8. ÖSSZEFOGLALÁS

Annak ellenére, hogy a webszolgáltatások és a szolgáltatásorientált programozás csak néhány éve van jelen, az üzleti szoftverek gyors és rugalmas fejlesztésében mégis meghatározó szerepe van, ezért nagyon fontos, hogy aki webszolgáltatásokkal és azok fejlesztésével kezd foglalkozni, tisztában legyen webszolgáltatás technológia jelentősségével, mivel úgy tűnik ez az első, igazán széles körben elterjedt technológia, amely nagy, akár üzleti szintű komponensek felhasználását teszi lehetővé akár az interneten keresztül, s így függetlenül attól, hogy az egyes szoftverkomponensek hol találhatók. Mindezt az XML-alapú szabványainak köszönheti, melyek szöveges megjelenésük miatt platformfüggetlen módon ugyanazt jelentik bármilyen architektúrán, és szintén ez az a pont, ahol a webszolgáltatások általánosabbak a DCOM, vagy CORBA komponenseinél.

Ugyanakkor a webszolgáltatások alapján meghatározták az szolgáltatásorientált architektúra általános jellemzőit, amely mint ahogy a gyakorlat is bizonyítja, a legalkalmas elosztott, multiplatformos szoftverrendszerek megvalósítására. Nem kizárt tehát, hogy a jövőben további, a webszolgáltatásoktól különböző szolgáltatásorientált architektúra technológiák látnak napvilágot.

A szolgáltatásorientált programtervezés legfontosabb kérdései, hogy pontosan mi legyen egy szolgáltatás, tehát a megvalósítandő összetett üzleti folyamatot hogyan bontsuk kisebb, jól definiált szolgáltatásokra, valamint hogy a különböző szolgáltatásoknak hogyan tevezzük meg az együttműködését, milyen kommunikációs mintákat valósítsunk meg közöttük. Mivel magukat a szolgáltatásokat valamilyen objektumorientált technológia segítségével szoktuk megvalósítani, ezért a szoftvertervezés szolgáltatásokhoz képest alacsonyabb szinjén nem volt szűkség új módszerek bevezetésére, az UML megfelel a célra, és mivel a szolgáltatások azonosítása és összeszervezése inuitívan is megoldható, ezért nem alakult még ki általános, szolgáltatásorientált programtervezési módszerek. Az utóbbi években azonban egyre több olyan eszköz jelent meg, amely a webszolgáltatások együttműködésének tervezését teszi lehetővé (pl. az Orc nyelv) ezért várható, hogy előbb utóbb általános módszer jelenik meg, amely egyszerre válaszolja meg a szolgáltatások szintjéhez tartozó

összes programtervezési kérdést.

Mindeközben folyamatos kutatások folynak olyan formális módszerek kifejlesztésére, amelyek lehetővé teszik a szolgáltatások matematikai modellezését. Hamarosan ezek a modellek lehetővé teszik majd szolgáltatások együttműködési hatékénységának vizsgálatát, s ezzel olyan eszközt biztosítanak majd, amellyel az elérhető szolgáltatásokból közel automatikus módon a lehető legoptimálisabb összetett szolgáltatást állíthatjuk majd elő saját üzleti szolgáltatásunk IT-re történő leképezéseként.

Megpróbáltam ebben a dolgozatban áttekintést nyújtani az informatikában minden területén megjelenő szolgáltatásorientált szemléletről, remélem elnyerte tetszését és segítségére vált a szolgáltatásorientált megközelítéssel most ismerkedő olvasónak.

Hivatkozások

[1] Gottdank Tibor, *Webszolgáltatások*, ComputerBooks, 2005.
ISBN 963 618 305 8

[2] Végh Csaba, *Alkalmazásfejlesztés*, Logos, 2000.
ISBN 963 037 660 1

[3] *Microsoft Architecture Journal*, 2004-2007.

[4] M. Boreale, R. Bruni, R De Nicola, *SCC: Service Centered Calculus*, 2006.

[5] C. Guidi, R. Lucchi, G. Zavattaro, *SOCK: a calculus for service oriented computing.*

[6] Kincses László, *SSADM*, MTA, 1993.

[7] W3C, *SOAP specifikáció*, http://www.w3.org/TR/soap/

[8] W3C, *WSDL specifikáció*, http://www.w3.org/TR/wsdl

[9] OASIS, *UDDI specifikáció*,
http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[10] IBM, *Implementing an ESB using IBM WebSphere*,
http://www.redbooks.ibm.com/redbooks/pdfs/sg247335.pdf