Debreceni Egyetem

Informatikai Kar

KORSZERŰ FELHASZNÁLÓI FELÜLETEK GENERÁLÁSA

Témavezető:
Kollár Lajos
egyetemi tanársegéd

Készítette:
Győri József
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

1. BEVEZETÉS

Szakdolgozatomban egy korszerű felhasználói felület generálásának főbb mozzanatait, illetve a hozzájuk szükséges technológiákat mutatom be. Korszerű felhasználói felület alatt tipikusan a népszerű webes felületet értjük a fejlesztés során. Egy UML osztály modellből kiindulva tehát egy olyan felhasználó felületet generálásának lépéseit kívánom leírni, mely mögött egy létező adatbázis van, és a legenerált felületen megadott inputok az adatbázisban entitásként tárolódnak.

A projekt inputja egy megfelelő UML szerkesztő segítségével előállított UML osztály diagramból importált XMI 2.1-es XML dokumentum. Következő lépésként az előálló dokumentumból ki kell nyerni a szükséges elemeket. Ehhez az XSLT transzformáció nyújt segítséget, melyet elvégezhetünk egy arra alkalmas XSLT transzformátorral, amely a komolyabb XML szerkesztők már beépítve tartalmaznak. A transzformálás után a létrejövő Java osztályokat és XHTML oldalakat, akár manuálisan, akár az XSLT transzformátorral a mellékelt web alkalmazás projektünkbe másoljuk. A kódok összeforrasztása után az előállított war fájlt az alkalmazás szerverére felhelyezzük, és máris elérhetővé válik az általunk előállított felhasználói felület.

2. FELHASZNÁLÓI FELÜLETEKRŐL RÖVIDEN

A szakdolgozatom célja – többek között – egy körszerű felhasználói felületet előállítása, ezért fontos tisztázni, mit is értünk pontosan a fogalom alatt. A felhasználói felület (angolul user interface, röviden UI) egy berendezés (például a számítógép), vagy egy számítógépes program (például egy operációs rendszer) azon elemeinek összessége, amelyek a felhasználóval való kommunikációért felelősek, és a berendezés vagy program irányítását, vezérlését lehetővé teszik.
A leggyakrabban előforduló felhasználói felület típusai:

- Parancssoros felhasználói felület (CLI – Command Line Interface): a parancsbevitel billentyűzettel történik, az üzenetkijelzés a monitoron, szintén szöveges formában.
- Szöveges felhasználói felület (TUI – Text User Interface): a monitoron szöveges feliratú karaktercellák helyettesítik a nyomógombokat és egyéb grafikus elemeket.
- Grafikus felhasználói felület (GUI – Graphic User Interface): a képernyőn szöveges és grafikus elemek együttesen jelennek meg.

A mai korban a korszerű felhasználói felületeknél követelmény a grafikus jelleg. Ilyen esetben a számítógép és a felhasználó közötti kapcsolat érdekében szöveges és rajzos elemek együttesen alkotnak egy jól megtervezett felületet. A grafikus felületeknél fontos szerepe van az interakciós eszközöknek és annak, hogy egy felület minél könnyebben átlátható és kezelhető legyen. Általánosságban a mai informatikai rendszerekben menük, ablakok, gombok és a hasonló funkcionalitással bíró eszközök alkotják a grafikus felületeket.

A grafikus felületek egyik legnépszerűbb fajtája a webes felület, amely a dolgozat fő irányvonala is egyben. A felhasználó a böngészőn keresztül jeleníti meg a kódból generált felületet. Általános irányvonalként itt is törekedni kell, hogy átlátható és praktikus legyen, és minél kevesebb adatforgalmat generáljanak ezek.

A projektben egy ún. CRUD felületet hozunk létre, amely a Create, Read, Update és Delete alapműveleteket valósítja meg egy felületen bizonyos entitásokkal kapcsolatban. A CRUD műveletek nélkül nem tekinthető teljesnek a tartalom és az adatkezelő rendszerek, ugyanis ezekből származnak a komplexebb manipulációk is.

3. TECHNOLÓGIAI ÁTTEKINTÉS

Ez a fejezet a projekt során felhasznált technológiák rövid ismertetését tartalmazza. A szakdolgozat kiinduló technológiája az XML, ezért érdemes részletesebben is foglalkozni vele és a hozzá szorosan kapcsolódó további technológiákkal.

3.1 XML (EXTENSIBLE MARKUP LANGUAGE)[1] [2]

3.1.1 RÖVID ISMERTETÉS

Az XML (eXtensible Markup Language) a W3C által ajánlott általános célú leíró nyelv, speciális célú leíró nyelvek létrehozására. Az elsődleges célja strukturált adatszerkezetek tárolása, de használható más nyelvek definiálására is. Az XML dokumentum önmagában egy úgynevezett fa-adatstruktúrát reprezentál. Az adatok reprezentálása az ún. 'tag'-ek közt történik, melyekhez további információ hordozására attribútumokat is csatolhatunk.

Egy elemhez tetszőleges számú attribútumot csatolhatunk. Az elem tartalma lehet szöveges (karakteres) adat, vagy másik elem(ek), ily módon valósítva meg a fa-struktúrát. Egy dokumentum csak akkor tekinthető XML dokumentumnak, ha jól strukturált, azaz megfelel az XML specifikáció által leírt szintaxisnak.

3.1.2 FONTOSABB SZINTAKTIKAI SZABÁLYOK:

- Egyetlen gyökérelem lehet a dokumentumban, ezt csak XML deklaráció, feldolgozó utasítások és megjegyzések előzhetik meg.
- Az üres elemek megjelölhetők üres elem (önlezáró) tag-gel, pl.:
`<tag_neve/>`
- Minden attribútumérték szimpla (') vagy dupla (") idézőjelek között van. Szimpla idézőjel szimpla idézőjelet, dupla idézőjel dupla idézőjelet zár.
- Az elemek egymásba ágyazhatók, de nem lehetnek átfedők. Így minden elemet egy másik elem kell, hogy magába foglaljon, a gyökérelemet természetesen nem.
- Az elem nevek kisbetű–nagybetű érzékenyek.
- A dokumentum karakterkészletének meg kell felelnie a benne leírt karakterkódolásnak. A karakterkódolás általában az XML deklarációban van meghatározva.
- Mivel a többek közt a '<', '>', '& karaktereknek speciális jelentésük van, ezért ha a szövegben használni kívánjuk őket, 'entity'-ket kell helyettük használnunk.

A '<' jel helyett a következőt kell használni : '<'
A '>' jel helyett a következőt kell használni : '>'
A '&' jel helyett a következőt kell használni : ' &' .

- Lehetőség van névterek használatára is:
  A namespace-ek (minősítő nevek/névtér) lehetővé teszik, hogy az elemek nevei – egy prefixummal ellátva – egyértelművé váljanak. A prefixum az adott namespace egyedi URI-je. A felhasznált namespace prefixumot deklarálni kell egy xmlns prefixumú attribútumban vagy valamelyik szülőnél pl.:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

3.1.3 XML DOKUMENTUMOK HELYESSÉGE

Ahhoz, hogy egy XML dokumentum helyes legyen, a következő követelményeknek kell megfelelnie:

- *Helyesen formázottság*. Egy helyesen formázott XML dokumentum megfelel minden XML szintaxis-szabálynak. Például: ha egy nem üres elem rendelkezik ugyan nyitó tag-gel, de záró 'tag'-gel nem, akkor nem helyesen formázott. Az a dokumentum, ami nem helyesen formázott, nem tekinthető XML-nek. Az elemzőnek meg kell tagadnia a feldolgozását.
- *Érvényesség*. Egy érvényes dokumentum olyan adatot tárol, ami megfelel a felhasználó által definiált tartalmi szabálynak, ami leírja a helyes adatértékeket és -helyeket. Például: ha a dokumentum egy elemének egy egész számként értelmezhető szöveget kell tartalmaznia, ám ehelyett a szöveg egy string, üres vagy más elemeket foglal magába, akkor a dokumentum nem érvényes.

3.1.4 ÉRVÉNYES XML DOKUMENTUMOK

A helyesen formázott és egy adott sémának megfelelő XML dokumentum érvényesnek nevezhető. Egy XML séma az XML dokumentum típusának leírása, jellemzően az XML megkötésein túli korlátozásokat tartalmaz az adott típus struktúrájára és tartalmára. Nagy mennyiségű szabványos és szabadalmazott XML séma jelent meg, hogy leírja ezeket a megkötéseket, és ezen sémák egy része maga is XML-alapú.

Az általánosított adatleíró nyelvek (mint például az SGML és az XML) megjelenése előtt a szoftvertervezőknek speciális fájlformátumokat vagy kis nyelveket kellett kifejleszteniük, hogy adatot tudjanak megosztani több alkalmazás között. Ez részletes specifikációk megírását, valamint speciális célú elemzők és írók megalkotását követelte meg.

Az XML szabályos struktúrája és szigorú elemzési szabályrendszere képessé teszi a szoftvertervezőket, hogy az elemzést szabványos eszközökre bízzák, és mivel az XML egy általános, adatmodell-orientált keretrendszert biztosít az alkalmazás-specifikus nyelvek fejlesztőinek, a szoftverfejlesztőknek csak a szabályrendszer és az adat kifejlesztésére kell koncentrálni, viszonylag magas absztrakciós szintet elérve.

Alaposan tesztelt eszközök állnak rendelkezésre, hogy az XML dokumentumot a sémával szemben validálják: az eszköz automatikusan ellenőrzi, hogy a dokumentum megfelel-e a sémában kifejtett megkötéseknek. Vannak olyan eszközök, melyek az XML elemzőbe építve érhetők el, illetve megjelentek a külön használhatók is.

A sémák más felhasználása is létezik: az XML szerkesztők például a szerkesztés során fel tudják használni a sémákat a hatékonyság és a kényelem növelése érdekében.

3.1.5 DTD (Document Type Definition)

A legrégebbi XML séma a Document Type Definition (DTD, dokumentum típus definíció), ami az SGML[3]-ből származik. Mivel a DTD az XML 1.0 szabvány része, így mindenhol támogatott. Azonban megvannak a maga hátrányai:

- nem támogatja az XML újabb képességeit, például a névtereket
- hiányzik a szemléletessége, az XML több formális képessége nem használható a DTD-ben
- az SGML-ből származtatott, nem XML szintaxist használ a sémák leírására

3.1.6 XML Schema

Az XML Schema egy újabb keletű XML sémanyelv, amit a W3C és a DTD utódaként definiáltak. Gyakran XSD (XML Schema Definition) néven is szokták emlegetni. Az XSD lényegesen többre képes a DTD-nél az XML nyelvek leírása terén. Sokoldalú adattípus rendszert használ, ami részletesebb megkötéseket tesz lehetővé az XML dokumentum logikai szintjén, és emiatt sokkal robusztusabb érvényesítő keretrendszert követel meg. Ráadásul az XSD XML-alapú formátumon alapul, minek következtében szokványos XML eszközöket lehet használni a létrehozásához és feldolgozásához. Az implementációk azonban sokkal többet kívánnak, mint az egyszerű XML olvasási képesség.
Az XML Schema-t érő kritikák közül néhány:

- A specifikáció nagyon nagy, ami nehézzé teszi a megértést és az implementálást
- Az XML-alapú szintaxis bőbeszédűséghez vezet a sémadefinícióban, ami nehezíti az XSD olvasását és írását

3.1.7 Más séma nyelvek

Néhány séma nyelv nemcsak egy adott XML formátum struktúráját írja le, hanem korlátozott lehetőséget is biztosít az egyedi – szabványak megfelelő – XML fájlok feldolgozásának befolyásolására. A DTD és XSD szintén rendelkezik ezzel a képességgel: alapértelmezett attribútum-definíciók meghatározását is elvégzik.

3.2 UML (UNIFIED MODELLING LANGUAGE)[4]

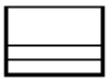
Az UML az objektumorientált-tervezés fontos eszköze. Implementációtól független tervezést tesz lehetővé, a teljes szoftverfejlesztési életciklust támogatja. A tervezés alatt álló rendszert diagramok segítségével írja le. (Pl.: Use Case, Class, Object, Component, Sequence, Collaboration, …, stb.).

A projekt fejlesztése során egy UML osztálydiagramból indulunk ki, ám fontos először is tisztázni az ide vonatkozó fogalmakat:

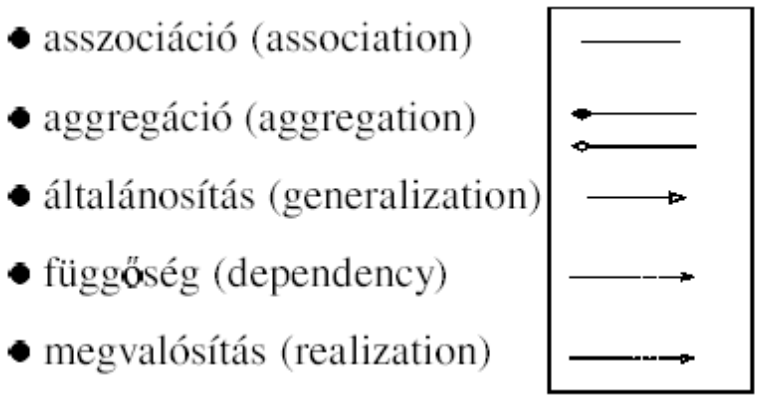
- **Osztály:** Az osztály nem más, mint egy absztrakt adattípus: egységbezárás. Tulajdonságok (állapotok) + viselkedések együttese. (Tulajdonságok = Attribútumok, Viselkedések = Metódusok)
- **Objektum:** Az osztály egy példánya. Minden objektum rendelkezik állapottal (az attribútumok határozzák meg), viselkedéssel (az operációk határozzák meg) és identitással (egyedi).
- **Öröklődés:** Egy osztály létrehozásához egy már meglévő osztályt használunk fel. Az új osztály örökli az ősosztály attribútumait, metódusait, de emellett lehetősége van új attribútumok, metódusok definiálására, a meglévő metódusok felüldefiniálására.
  - Általánosítás: Több osztály közös részstruktúráját és – viselkedését egy közös ősosztályba helyezzük.
  - Specializálás: Származtatott osztályokat hozunk létre, melyek finomítják az ősosztályt.

Ezután vizsgáljuk meg a fejlesztés során is használt osztálydiagramot, melyet az osztályok és a közöttük fennálló kapcsolatok reprezentálására használunk.

- Az osztály UML jele:



- Alapvető kapcsolatok:



- Asszociáció-kapcsolat:
  - Osztályok közötti kétirányú összeköttetés (a navigálási irány megadható – üzenet iránya).
  - „Használati kapcsolat”: létük általában egymástól független, de legalább az egyik ismeri és/vagy használja a másikat.
  - Gyakorlatilag az osztályokból létrejövő objektumok között van kapcsolat.

- Aggregáció kapcsolat:
  - Az asszociáció egy speciális formája.
  - „Rész-egész” kapcsolat (erősebb, mint az asszociáció).
  - Az egyik objektum fizikailag tartalmazza vagy birtokolja a másikat.
  - A rész objektumok léte az egészobjektumtól függ.
  - UML jelölésben egy rombusz található a kapcsolatvonalon, a tartalmazó oldalán
    - Kétféle:
      - ***erős tartalmazás***: ha a tartalmazott objektum nem vehető ki a tartalmazóból.
      - ***gyenge tartalmazás***: ha a tartalmazott objektum kivehető a tartalmazóból.
- Kompozíció**:** olyan tartalmazás, ahol az egész objektum minden részével erős tartalmazási kapcsolatban áll.

- Kapcsolatok nevei:
  - Az asszociáció-kapcsolatnak opcionálisan lehet neve.
  - Ez általában egy igés kifejezés (A kapcsolatvonalra tesszük, nagy kezdőbetűvel, aláhúzás nélkül).
  - A név egy irányt is meghatároz (három szög a név mellett).
  - Az aggregációnak általában nincs neve, mert egyébként is oda kell érteni, hogy ***has***, vagy ***contains.***

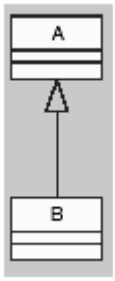
- Szerepek nevei:
  - Az asszociáció/aggregáció végpontját szerepnek nevezzük.
  - Általában ez egy főnév, mely a kapcsolat adott végpontján levő osztály szerepét írja le az adott kapcsolatban.
  - Helyettesítheti a kapcsolat nevét.

- Multiplicitás:
  - A kapcsolatok végpontjainál definiáljuk a hozzátartozó osztályokat.

- A gyakorlatban a kapcsolat megvalósulásánál résztvevő objektumoknak a számát szabja meg.
- A kapcsolat mindkét végén lehet.
- Példák: 1..* számosság.

- Öröklődés kapcsolat:

UML jelölés:



Az öröklődési kapcsolatnak nincs neve, illetve nincsenek használva szerepnevek és multiplicitás sem. Az öröklődés feltárása az osztályok vizsgálatával történik. Megnézzük, hogy van-e lehetőség általánosításra vagy specializálásra.

3.3 XMI (XML METADATA INTERCHANGE)[5]

Az XMI szintén OMG technológia, mely az objektum modellek és egyéb strukturált OMG eszközök interneten való hordozhatóságáért jött létre. Tehát az XMI lényege a meta-adat leírása, és ezáltal hordozhatóvá tétele, különösen az UML alapú modellező eszközök között. Az XMI már nevéből fakadóan egy speciális XML dokumentum, így ugyancsak rendelkezik az XML által nyújtott számtalan előnnyel.

3.4 XPATH (XML PATH LANGUAGE)[6][7]

3.4.1 LEGFŐBB JELLEMZŐK

Az XPath a W3C ajánlása, mely az XSLT és XPointer definiálásának eredményeképpen jött létre. Elsődleges célja az XML dokumentumok alkotórészeinek megcímzése, de ezen felül tartalmaz még néhány alapműveletet is. Az XPath nem XML szintakszisú, mert használhatónak kell lennie XML dokumentumon belül, például attribútumokban. Egy egyszerű Xpath kifejezés a következőképpen néz ki:

```
/html/head/meta[@name='Peter']/@age
```

Ez a kifejezés a html tag unokájának a 'meta' nevű, olyan csomópontjának az 'age' attribútumát jelöli ki, melynek a 'name'-attribútuma: 'Peter'. Összefoglalóan az Xpath-ra jellemző, hogy:

- Tömör, nem-XML szintaxis (XPath kifejezések).
- XPath kifejezések használata URI-kban es attribútumok értékeiben.
- XML dokumentumokhoz a DOM-hoz hasonló módon logikai modell definiálása, azonban eltérések a DOM-modelltől.
- Fa-adatmodell, amelyben minden csomóponthoz definiált annak sztringértéke.
- Az XPath kifejezések egy részhalmaza alkalmas mintaillesztésre, éppen erre használja az XSLT.
- Az elnevezés abból származik, hogy a '/' karakter olyan , mintha hierarchikus könyvtárba navigálnánk.

3.4.2 LÉPÉSEK

Akárcsak a fájlrendszerek esetén, itt is kétféle útvonalleírás létezik. Az egyik a relatív, ahol az elérési út kiinduló pontja az aktuális pozíció. A másik az abszolútleírás, ahol a leírás egy „/” jellel kezdődik, illetve a kiindulási pont az XML dokumentum gyökere (root).

Egy elérési út által elérhető csomópontok körének meghatározásakor az elérési út értelmezése mindig balról jobbra történik. Az első lépés során a lépés kiindulópontja az aktuális – vagy abszolútleírás esetén a gyökér – elem, és a lépés során kiszámítódik mindazon csomópontok köre, amelyet az adott lépés leír.

Az irány a lépéssel megjelölni kívánt csomópontok és az aktuális elem viszonyát írja le. A következők lehetnek:

- *child*:: Gyermek. Az aktuális elem gyermekeleme. Amennyiben nincs irány megadva, akkor ez az alapértelmezés.
- *descendant*:: Leszármazott. Az aktuális elem összes leszármazottja (a gyermekei és azoknak is a gyermekei rekurzívan).
- *parent*:: Szülő. Az aktuális elem szülő elemét jelöli ki.
- *ancestor*:: Felmenő. Az aktuális elem összes felmenőit jelöli ki. (A saját szülőjét és annak a szülőjét is rekurzívan, így természetesen szükségszerűen mindig a gyökér elemet is.)
- *following-sibling::* Következő testvér. Az aktuális elem következő testvérét jelöli ki. Ha az aktuális csomópont tulajdonság vagy névtér, a következő elem halmaz üres .
- *preceding-sibling::* Előző testvér. Az aktuális elem előző testvérét jelöli ki. Ha az aktuális csomópont tulajdonság vagy névtér, az előző elem halmaz üres.
- *following::* Következők. Az aktuális elemet követő minden elem a dokumentum sorrendjében.
- *preceding::* Megelőzők. Az aktuális elemet megelőző minden elem a dokumentum sorrendjében.
- *attribute::* Tulajdonság. Az aktuális elem tulajdonságait jelöli ki. Üres, ha az aktuális elem nem egy csomópont.

- *namespace::* Névtér. Az aktuális elem névtereit jelöli ki. Üres a halmaz, ha az aktuális elem nem egy csomópont.
- *self::* Önmaga. Ez az aktuális elemet jelöli ki.
- *descendant-or-self::* Leszármazottak vagy önmaga. Az aktuális elemet és a leszármazottait jelöli.
- *ancestor-or-self::* Felmenő vagy önmaga. Az aktuális elemet és a felmenőit jelöli.

3.4.3 KIFEJEZÉSEK, FELTÉTELEK.

A feltételes kifejezés egy tetszőlegesen összetett kifejezés, amely a csomópontok körének meghatározására egyéb XPath kifejezéseket, függvényeket és a csomópontok tartalmát is felhasználhatja. A kifejezés kiértékelése előtt létrejön egy az irány és a típus által meghatározott csomóponthalmaz. Ezután az értelmező végigmegy a halmaz elemein, és minden egyes elemre kiértékeli a feltételes kifejezést. Ha a kifejezés az adott elemre igaz értékű, akkor a csomópont bekerül az XPath lépés eredményhalmazába. Egy feltételes kifejezés egy vagy több kifejezésrészből épül fel, ezek között logikai műveletek teremtenek kapcsolatot. A kifejezéshez a következő műveleti jelek használhatók fel, amelyek szabványosnak mondhatóak az adott területen:

- **or** (logikai vagy)
- **and** (logikai és)
- = (egyenlőség vizsgálat), **!**= (nem egyenlőség vizsgálat)
- <= (kisebb vagy egyenlő), < (kisebb), >= (nagyobb vagy egyenlő), > (nagyobb)

Egy összetettebb példa egy ilyen kifejezésre, amely egy XMI dokumentum csomópontjából jelöl ki elemeket:

```
$document//mdOwnedViews/mdElement[@elementClass='Association' and
child::elementID/@xmi:idref=fc:classInWhichAssociation($document,$node)]
/linkSecondEndID/@xmi:idref=$tempID
```

3.5 XSLT (EXTENSIBLE STYLESHEET LANGUAGE TRANSFORMATION)[89]

Egy XSLT dokumentum feladata a transzformáció. A meglévő XML dokumentum adatait használja fel arra, hogy egy akár az eredetitől teljesen különböző adathalmazt és adatstruktúrát állítson elő. Teljesen lényegtelen, hogy azután ezekből az adatokból egy másik XML dokumentum vagy egy HTML keletkezik. Az XSLT-t az úgynevezett XSLT transformátor végzi, amely inputként megkapja a forrásdokumentumot – általában valamilyen XML-hez köthető dokumentum –, továbbá az XSL dokumentumot, illetve opcionálisan megadható egy kimeneti útvonal is. Az eredménye a legenerált egy vagy több dokumentum lesz.

3.5.1 FELÉPÍTÉS, ESZKÖZÖK

Egy XSL dokumentum – ez esetben mindegy, hogy XSLT vagy XSLFO –, miután egy XML dokumentum is egyben, természetesen az xml vezérlési utasítással és a verzió megadásával kezdődik. A dokumentum gyökere a stylesheet elem, ahol rögtön megadásra kerül a használt névtér és a verzió is. Egy XSLT dokumentum alapeleme sablon. Sablonokat rendelhetünk egy XML dokumentum valamely csomópontjához, sőt akár minden csomópontjához, hogy azon a ponton található adatokra, illetve az onnan elérhető gyermekelemekre vonatkozó manipulációt végrehajthassuk.

```
<xsl:template match="/" fc:name="Crud">
```

A feltüntetett példa a sablont az egész inputként megadott dokumentum összes csomópontjára fogja értelmezni. XPath kifejezéseket lehet megadni a match attribútum értékeinek, így lehet szűkíteni az érintett csomópontok számát, melyre a sablon alkalmazva lesz. Az XSLT végigjárja az XML fát, végigmegy annak minden elemén, majd az adott aktuális elemhez veszi a megfelelő sablont, és végrehajtja azt. Fontos, hogy ha egy csomóponthoz talál megfelelő sablont, akkor annak a csomópontnak a gyermekein már nem megy végig, vagyis nem keres hozzájuk sablont. Egészen pontosan csak akkor foglalkozik velük, ha erre az adott szülősablon külön kéri. Az értelmező úgy tekinti, hogy egy sablon végrehajtásával – hacsak az másképpen nem rendelkezik – a gyermekelemek transzformációja is befejezettnek tekinthető. Az *apply-templates* kulcsszó segítségével lehet kerőszakolni a gyermekek transzformációját.

Számos transzformáció során szükségünk van az egyes csomóponthoz tartozó attribútumok pillanatnyi értékeire, hogy azoktól függően végezzünk tevékenységeket. Ehhez az eszköz a *value-of* kulccsszó. A következőképpen használjuk:

```
<xsl:value-of select="$className"/>
```

Az előző példában a *$className* változó pillanatnyi értékét kapjuk meg. Itt felmerül a kérdés, hogy az adott transzformáció eredménye miként tárolódik. Erre ad megoldást az xsl-*output* parancs, melyben definiálhatunk output-típusokat. Egy szabványos HTML kimenet a következőképpen néz ki:

```
<xsl:output name="html" method="html" version="4.0"
encoding="utf-8" indent="yes" /> <xd:doc scope="stylesheet">
.
.
<xsl:result-document href="webapp/new{$className}.xhtml" format="html">
```

Az xsl:output segítségével létrehozunk egy kimeneti típust, amit *html*-nek nevezünk, majd a *result-document* kulcsszó segítségével *format* attribútumába – a *html* értéket megadva – a *</xsl:result-document>* kulcsszóig egy kimenetbe megy a két tag közötti műveletek eredménye. Tetszőleges számú ilyen tag adható meg dokumentumonként, illetve sablononként. Itt jön képbe a XSLT nyújtotta ciklusutasítás: a *for-each*, melynek segítségével több dokumentumot gyárthatunk le akár egy adott csomóponthalmazból. Természetesen számos más esetben használható is. Tekintsük a következő kódrészletet:

```
<xsl:for-each select="fc:connections(root(.))">
<xsl:valueof select="fc:connectionID(current())"/>
</xsl:foreach>
```

A ciklus segítségével a connections függvény által visszaadott csomópontokon futunk végig, és az iteráció során az éppen megfogott csomópontot a current() kulcsszó segítségével érjük el. A példából is látható, hogy a transzformációk alatt az átláthatóság és az újrafelhasználhatóság érdekében létrehozhatunk függvényeket. Az xsl:function kulcsszó segítségével az alábbi módon:

```
<xsl:function name="fc:lower" fc:type="string">
```

Ezeket a függvényeket akár külön XSL dokumentumba is tárolhatjuk, és az *xsl-include* parancs segítségével hivatkozhatóvá tehetjük őket. A névütközés elkerülése végett a névterek használata függvényneveknél kifejezetten ajánlott.

3.6 XHTML (EXTENSIBLE HYPERTEXT MARKUP LANGUAGE)[10]

A HTML (HyperText Markup Language) szabványt sok kritika éri amiatt, hogy a tartalmi jelölésre szolgáló elemek mellett nagyon sok formai, a megjelenítést meghatározó elemet tartalmaz. Így erősen összemosódik a tartalom és annak megjelenése.
A HTML 4.0 szabványban ezért sok – a korábbi verziókban előszeretettel használt – elemet érvénytelenítettek, és a megjelenésre, design-célra a stíluslapok használatát javasolták. A HTML egyik nagy hátrányát azonban ez a szabvány sem tudta megváltoztatni, továbbra is egy statikus, előre definiált elemeket és paramétereket tartalmazó jelölőnyelv maradt. Az új követelményeknek megfelelően hozták létre az XHTML-t. Az XHTML családba tartozó dokumentumtípusok XML alapúak, azonban a megfelelő szabványok betartása mellett hagyományos – HTML 4.0 szabványt ismerő – böngészővel is olvashatók maradnak.

Vizsgáljuk meg, hogy milyen feltételeknek kell teljesülnie ahhoz, hogy egy XHTML dokumentum valid legyen:

3.6.1 VALIDÁCIÓS SZABÁLYOK[11]

- Az elemek egymásba ágyazásánál ügyelnünk kell a sorrendre, vagyis az elemek megnyitásának fordított sorrendjében kell elhelyezni a záróelemeket.
- Jól formázott dokumentumot kell létrehozni, tehát minden elemet a <html> elemen belül kell elhelyezni. Minden elemnek lehetnek további beágyazott elemei. Ezek az elemek páronként kerülnek megadásra, és ügyelni kell arra, hogy a szülőelembe szabályosan kerüljön beágyazásra. A jelenlegi HTML böngészők a szintaktikai hibák felett elsiklanak, illetve megpróbálják kitalálni, hogy mi volt a dokumentum létrehozójának szándéka. Az eredmény szempontjából mindegy, hogy a HTML kódok sokszor átláthatatlanok vagy feldolgozhatatlanok a helytelenül strukturált elemek miatt. Ezzel szemben az XML alapú böngészők ha nem jól formázott dokumentummal találkoznak, nem tudják feldolgozni azt.

- A tagneveket kis betűkkel kell írnunk. Mivel az XML szabvány megkülönbözteti a kis- és nagybetűket, a
 és
 tag két különböző dolgot jelölhet. Minden XHTML elemet kötelező lezárni. Érdemes itt megjegyezni, hogy az olyan html elemeket, melyeknek nincsen külön zárópárjuk az eredeti HTML szabványban, kötelezően szóköz és / jellel kell zárni pl.
.
- A paramétereket kisbetűvel kell írnunk.
- A paraméterek értékeit "" jelek közé kell zárnunk.
- Az attribútumok minimalizálása, rövidítése nem megengedett.
- A name attribútumok helyett id attribútumot kell használni.
- Az xml:lang elem szerepeltetése.
- Minden XHTML dokumentumnak rendelkezni kell DOCTYPE deklarációval. A html, head és body elemeknek szerepelniük kell, a title elemnek pedig a head elemen belül kell elhelyezkednie.

3.6.2 DOCTYPE

A DOCTYPE definiálja a dokumentum típusát. Az 1.0-s szabványban három dokumentumtípus létezik.

1. XHTML 1.0 Strict

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Ezt akkor ajánlatos használni, amikor tiszta – prezentációs lehetőségektől mentes – kódot állítunk elő, és stíluslapokat használunk.

2. XHTML 1.0 Transitional

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
```

Ezt akkor érdemes alkalmazni, amikor ki akarjuk használni a HTML prezentációs lehetőségeit, illetve amikor olyan böngészőre fejlesztünk, amely nem ismeri a stíluslapokat.

3. XHTML 1.0 Frameset

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

A nevéből is következik, hogy ezt akkor használjuk, ha az ablakot több keretre osztjuk fel.

3.7 JSF (JAVASERVER FACES)[12]

3.7.1 ÁTTEKINTÉS

Az összetett web alapú felhasználói felületek kialakítása továbbra is sok fejlesztői munkát igényelnek. Ki kell bontani a kérés paramétereit, ellenőrizni és feldolgozni azokat, visszaképezni HTML vezérlőkbe, stb. A megvalósítás fárasztó és hibaforrásban bővelkedő. A web-alkalmazásokra általában a vezérlők korlátozott halmaza áll rendelkezésre – korlátozott interaktivitással –, de egyéb funkcionalitásban nem különböznek a desktop alkalmazásoktól. Természetesen megpróbálták adaptálni a desktop alkalmazásnál használt modellt a szerveroldali fejlesztésekbe is. Ezt a megoldást választotta a JSF szakértői csapat. Mivel a desktop alkalmazások mégis valamennyire különböznek a szerveroldali alkalmazásoktól, új modellt kellett kidolgozniuk. Többek között a JSF is az MVC[13] architektúrát követi. Egy vezérlő servlet-et nyújt(Faces-Servlet), és renderelőkkel, más néven leképezőkkel megengedi a nézet különválasztását, és egy modellt, ami komponens osztályokat szolgáltat, és egy esemény alapú mechanizmust. A JSF-el az MVC architektúra a komponensek szintjén dolgozik.A JSF komponensek összeszerkesztéséhez Java kód írása helyett – mint ahogyan a Swing esetében is – a JSF fájlokba XML jelölésekkel határozhatja meg a fejlesztő a komponensek összefüggéseit. A JSF és a JSP kapcsolata opcionális, de a JSF van előtérben. A JSF általános komponensek halmazát szállítja a Standard JSF Tag Library-ban. Ez lehetővé teszi, hogy kiküszöbölje az összetett felhasználói felületek fejlesztésekor a kódolást, mivel könnyebbé teszi a grafikus eszközök alkalmazását.

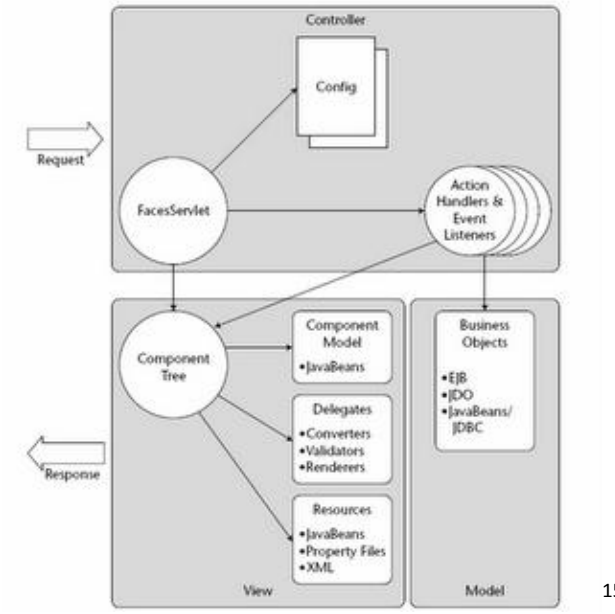
A JavaServer Faces egy szerver oldali komponens-keretrendszer Java technológia alapú web alkalmazások fejlesztéséhez. A következő elemekből áll ez a technológia:

- API, komponensek kezelésére és állapotaik módosítására; eseménykezelő, szerver oldali validátor; adatkonvertáló; és a felsorolt elemek kiterjesztését, bővítését is támogatja.
- Tag könyvtárak webes komponensek létrehozására és azok szerver oldali objektumokkal való összekötésére.

Egy tipikus Enterprise Java alkalmazásnál legalább három rétegről beszélhetünk :

- Front-End : ez a felel a megjelenítésért, tipikusan Web technológiákat soroljuk ezen réteg alá.
- Alkalmazás réteg: az üzleti logikát és az egyéb kontroll tevékenységeket értünk ezen réteg alatt.
- Adatbázis réteg: ez biztosítja az alkalmazás perziszenciáját, adatbázisban való tárolását.

3.7.2 A JSF MVC ARCHITEKTÚRA RÉSZEI[14]



15

1. ábra

1. View

A JSF alapvetően Tag Library gyűjtemény, ebből adódóan fastruktúrában tudjuk leképezni a weboldalakat. Minden egyes HTML tag-nek van JSF megfelelője, így akár az oldalak statikus összetevői is interaktívvá, programozhatóvá válhatnak. A JSF komponensek egyfajta önálló életet élnek az oldalon, a saját – kérésen érkező – értékeiket feldolgozzák, kiértékelik, érvényesítik, s ennek megfelelően teszik közzé a törzsüket, vagy irányítják egy másik oldalra a böngészőt. A JSF komponensfával kivétel nélkül JSP oldalakon belül találkozhatunk, ugyanis erősen épít a JSP technológiákra.

2. Model

A renderelt weboldalak adatmodellje egy vagy több POJO Bean (Plain Old Java Object). Ezek olyan egyszerű Bean-ek, amelyek a beállított tulajdonságuk révén bizonyos élettartamig adatokat tárolnak. A JSF oldalt bármelyik POJO Bean bármelyik tulajdonságával össze tudjuk kötni, erre használjuk a data-binding kifejezést. A projekt során Entity Beaneket alkalmazunk. Ezeket ebben az esetben Backing Beaneknek is nevezhetjük, hiszen a webes felületen lévő input egy ilyen Entity Bean-re képződik le. Ezt a Controller kezeli és az EJB osztály biztosítja, hogy megfelelően legyen tárolva.

3. Control

A vezérlést mindenképpen fel kell bontani több részre, ugyanis a vezérlés befolyásolására

- képes egy JSF Tag Library is, érvénytelen adatot észlelve ugyanis meggátolhatja a következő oldalra való átlépést;

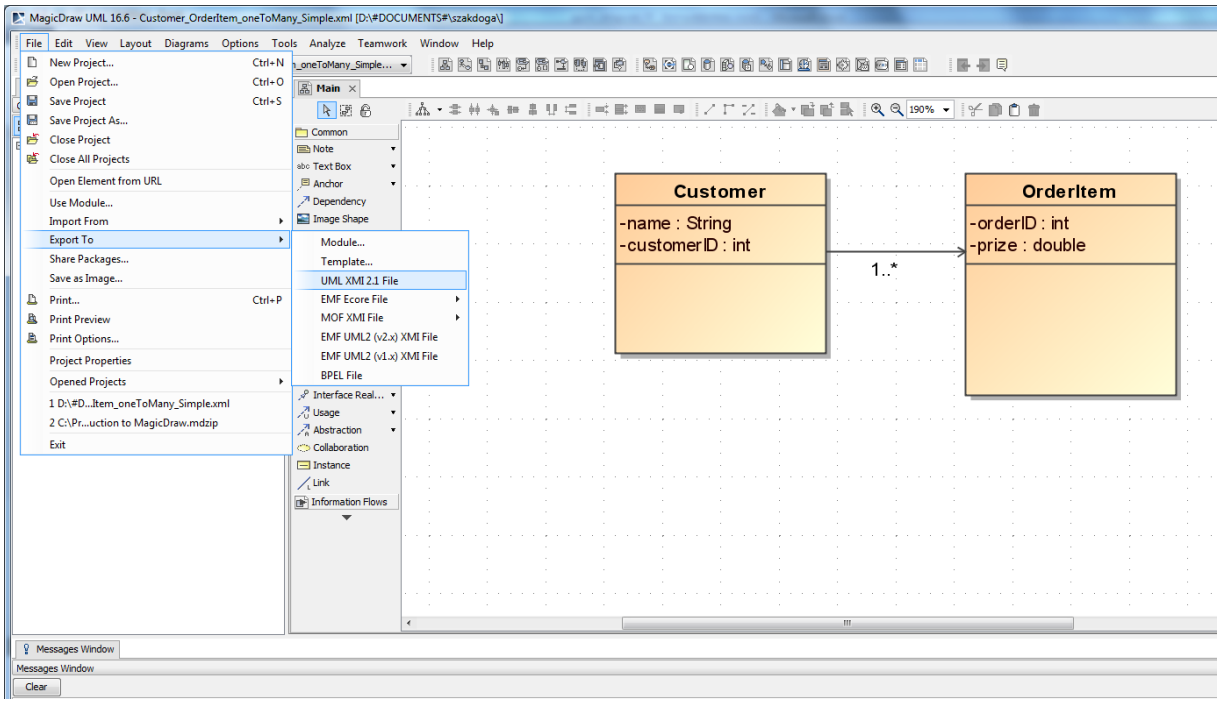
- képes egy Managed Bean is, a kért művelet eredménytelensége folytán hibaoldalra léptetheti a böngészőt;
- képes a JSF mechanizmus a meghatározott navigációs célok elérése esetén.

4. Fejlesztési eszközök

Ebben a fejezetben pontosan ismertetem a projekt során felhasznált szoftvereszközöket.

4.1 MagicDraw UML 16.6[16]

A MagicDraw egy többszörös díjnyertes UML üzleti folyamat, architektúra, szoftver illetve rendszermodellező eszköz, mely munkacsoport támogatást is nyújt. Támogatja a legújabb UML szabványokat.

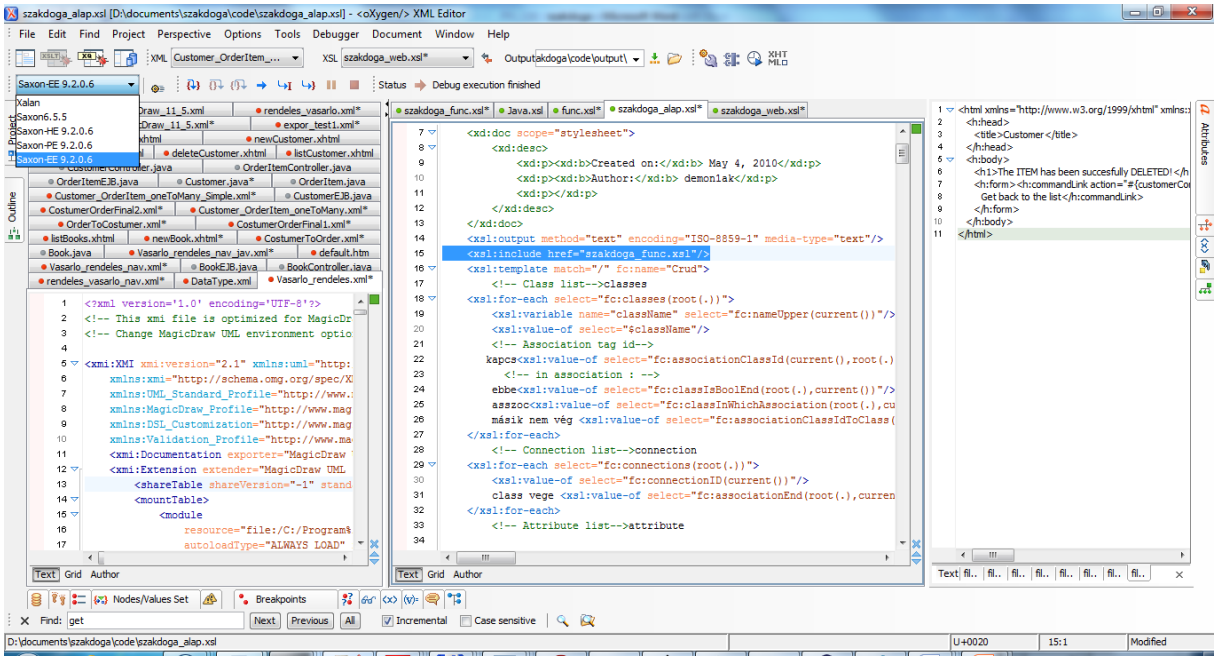


2. ábra

A képen éppen az osztálydiagram exportálásának opciói vannak feltüntetve. A projekt során egy XMI 2.1 verziójú dokumentumot dolgozunk fel, amelyet támogat ez a modellező eszköz.

4.2 OXYGEN XML EDITOR 11.2

A nevében is benne van, hogy egy XML szerkesztőprogram: tetszetős felületet biztosít az XML-re épülő dokumentumok szerkesztésére. Tartalmaz beépített XPath generátort, ami hasznosnak tűnhet egy óriás méretű XMI dokumentum feldolgozásakor, illetve egy a dokumentum-összehasonlító funkciót is, mely kijelzi a két dokumentum sorai közötti különbséget. Van beépített Debugger is a programban, de a számunkra legfontosabb funkciója a beépített XSL Transzformátor.



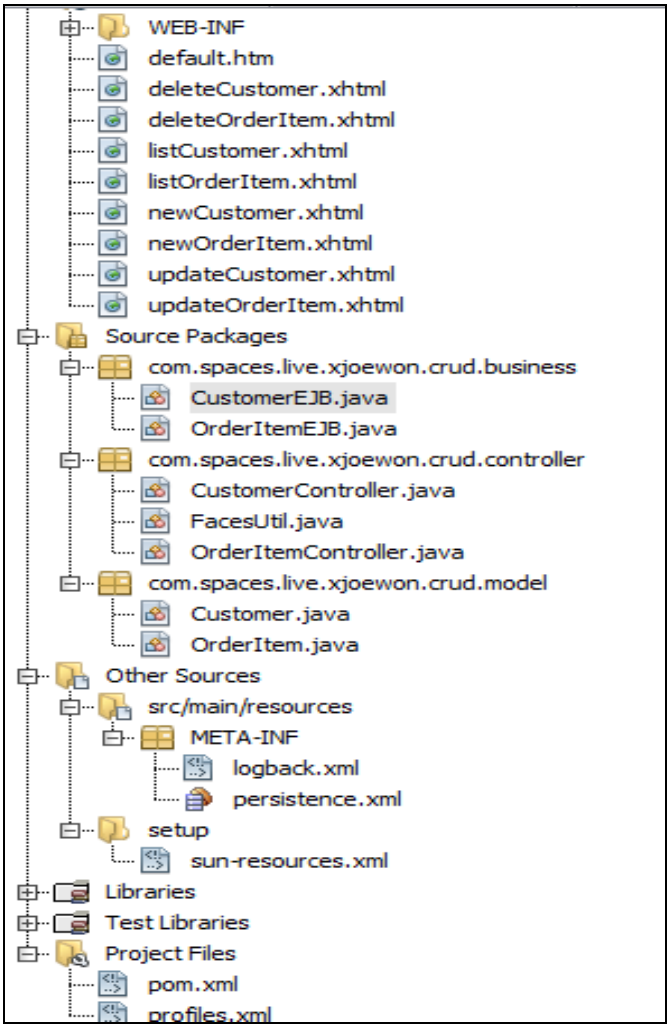
3. ábra

Egy kiválasztott Saxon-EE 9.2.0.6 típusú generátorral dolgozunk. A képen bal oldalt találhatóak az XMI fájlok, középen az XSLT dokumentumok, míg jobbra jelennek meg a transzformátor által előállított dokumentumok. A felső menüknél adhatjuk meg a feldolgozandó XMI fájl nevét, a használandó XSLT-t és egy könyvtárat, ahova a kimenetet szeretnénk legenerálni. Ez az eszköz nagyon hasznosnak bizonyult a projekt során.

4.3 NETBEANS IDE 6.8 (GLASSFISH, MAVEN ÉS JAVADB)[17]

A projektben szükségünk van egy Java szerkesztőre és egy webes alkalmazás szerverre. A Netbeans egy Java-ban íródott integrált fejlesztői környezet. A Glassfish[18]alkalmazás szerver külön is, de akár a Netbeans-el integrálva is letölthető. A Maven egy projektkezelő és fordításautomatizáló eszköz, mely a legújabb Netbeans verzióba van integrálva, illetve letölthető saját honlapjáról is. JavaDB[19]-vel hasonló a helyzet, a Netbeans integrálva tartalmazza. A transzformációk segítségével legenerált fájlokat az előre elkészített Netbeans

projekt megfelelő könyvtáraiba kell másolni, vagy a transzformátor kimenetét kell beállítani a projekt könyvtárára. (A szakdolgozathoz mellékelve van ez a Netbeans projekt.) Mivel Maven projektet készítünk, így a pom.xml fájlba az alkalmazott technológiákat, mint függőségeket kell feltüntetni. Ezután a Maven automatikusan kezeli ezeket, és a fordítás megteszi a szükséges lépéseket, hogy megfelelően lefusson a projekt. A Maven és a Glassfish integráció miatt, ha a kész projektet lefuttatjuk, akkor az előállított WAR fájlt a Glassfish automatikusan Deploy-olja, azaz felhelyezi az alkalmazás szerverre. Ezt elérhetjük a böngészőnkből. Futás közben a Netbeans is lehetővé teszi a Glassfish logok olvasását, innen megtudhatjuk, hogy webalkalmazásunkkal pontosan mi történik. Ezt azonban a Glassfish kezelésére elérhető webes kezelőfelületen is megtehetjük. A projektünk könyvtárstruktúrájának hasonlónak kell lennie a következővel:



4. ábra

A fenti (4.ábra) mutatja be tehát a könyvtárstruktúránkat. A web-inf könyvtár tartalmazza opcionálisan a faces-config fájlt, amivel a navigációt és egyes Bean-ek viselkedését lehet megszabni. Projektem során a navigációt faces-config nélkül oldom meg, a Bean-eket meg az

újonnan bevezetett annotációkkal látom el. Ez a mappa tartalmazza a web.xml fájlt, ami a kódunk futtatásához elengedhetetlenül szükséges Faces Servlet technológiát rendeli a projektünkhöz, gyakorlatilag ez működteti a programunkat:

```
<servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
```

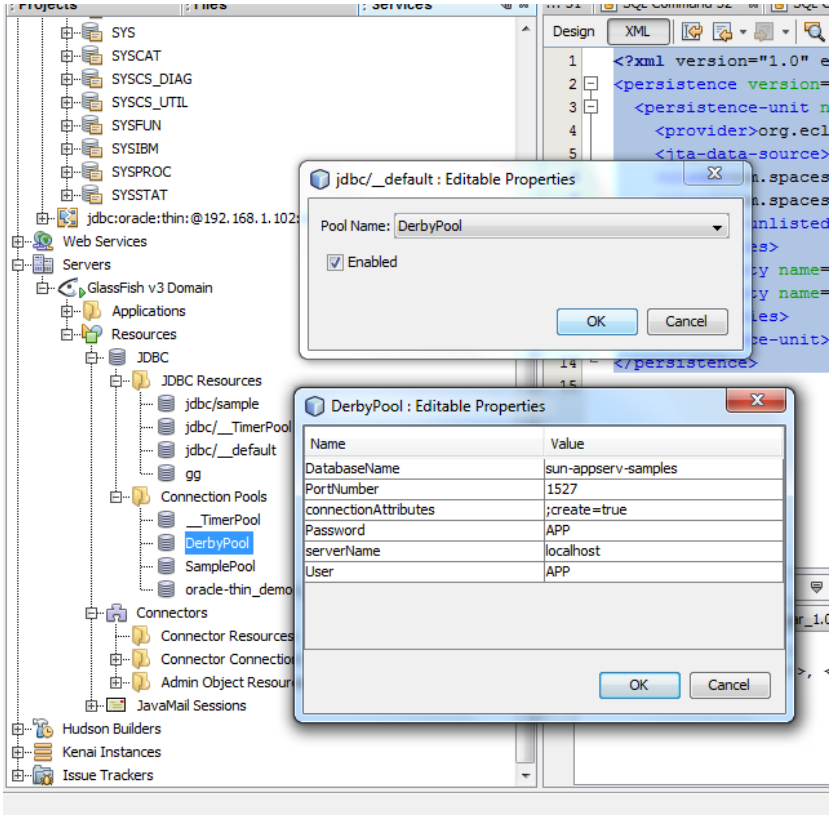
A web-app (web-pages) könyvtár tartalmazza az előállított xhtml, html fájlokat. Az src könyvtárban (Source Packages) helyezkedik el a Model alkönyvtár, melyben megtalálhatóak az ún. Backing Bean-ek. Ezek az inputok mögött állnak, és azok értékeit veszik fel a megfelelő attribútumaikban. A Controller alkönyvtárba a vezérlő fájlok kerülnek. Míg a Business könyvtárba az üzleti logikáért felelős fájlok, a projektünk esetén az adatbázissal történő kommunikációért felelősek.

A resources mappában helyezkedik el a logback.xml, ami opcionális és a logger konfigurációját tartalmazza. Ez tesztelésnél, debugging esetén segítséget nyújt a web alkalmazások nyomon követéséhez. A mappa másik eleme a persistence.xml, amely a következőképpen néz ki:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="crud" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <class>com.spaces.live.xjoewon.crud.model.OrderItem</class>
    <class>com.spaces.live.xjoewon.crud.model.Customer</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.logging.level" value="FINE"/>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Ez megfelel az XML szabvány által megkövetelt megszorításoknak. Ebben a fájlban állítjuk be, hogy a projekt során milyen eszközt használunk a perzisztencia céljából. A *transaction-type="JTA"* határozza meg a tranzakciókezelő API-t. A provider tag-ek között a projekt szempontjából fontos JPA (Java Persistence API) kerül deklarálásra. A JPA[20] implementációk

(és elődjük - a Hibernate) nagyon sok felesleges munkát levettek az ember válláról, s az egyszerűbb ORM műveletek mellé idővel – az adatbázisokkal való munka során felmerült problémák megoldásaként – egyre több eszköz került. Ilyen eszköz az Entity és/vagy Query Cache, amely a már beolvasott entitást adta vissza az adatbázishoz való kérdés nélkül, illetve a már lefuttatott kérdésre az adatbázisból egyszer már elkért halmazt hozza vissza ugyanilyen módon. A jta-data-source tag állítja be a használni kívánt tényleges adatbázisrendszert.



A Glassfish JDBC erőforrásai közül kiválasztott *JDBC/_default* egy DeryPool-t hivatkozik, mely egy JavaDB adatbázis kapcsolatot jelent. Java SE 6 része a JavaDB, egy teljes egészében a Javaban implementált relációs adatbázis-kezelő rendszer. Adatbázisos Java alkalmazások fejlesztésénél a JavaDB nagyszerűen használható teszteléshez, így ezt a tulajdonságát a projekt során is ki lehet használni. Az alkalmazás mögött álló adatbázist ezen érték átírásával cserélni lehet másik adatbázisra úgy, hogy az alkalmazás többi rétegére ez a változtatás teljesen közömbös lesz. A sun-resources.xml dokumentumba pedig megadhatunk projekt specifikus adatbázis kapcsolatot és a hozzátartozó Connection Pool-t.

5. A PROJEKT TERVEZÉSE

Ebben a fejezetben a fejlesztés lépéseit tárgyaljuk a már korábban bemutatott technológiák segítségével és az ismertetett fejlesztési eszközök által.

5.1 Elképzelés, cél

Cél egy webes felület létrehozása, mely lehetővé teszi, hogy az UML diagram által reprezentált osztályoknak megfelelő weblapok készüljenek Formokkal ellátva, ahol az osztály attribútumai lesznek a megfelelő inputok. Az inputok bevitele után az osztály mint Entitás fog viselkedni, melyet a megfelelő eszközökkel az adatbázisban is tárolni fogunk. A navigációt az osztályok közötti kapcsolatok feltárása után a már ismertetett technológiával valósítjuk meg.

5.2 Tervezett fejlesztési lépések

Az eddig ismertetetteket összefoglalva a következőképpen vázolhatjuk fel a fejlesztéshez szükséges lépéseket:

- UML osztálydiagram elkészítése. Bizonyos egyedi megszorításoknak meg kell felelnie.
- Az UML diagram importálása XMI 2.1 XML formátumba.
- XSL dokumentumok létrehozása az XMI dokumentum feldolgozása céljából.
- Az XSL Transzformáció végrehajtása az előbb legenerált XMI dokumentumokon.
- A kinyert Java osztályokat és HTML lapokat a dolgozathoz csatolt projekt könyvtárba bemásoljuk a már ismertetett hierarchia szerint.
- Netbeansben a Maven automatikus eszköz segítségével lefordítjuk a projekt tartalmát, majd azt a Maven automatikusan elhelyezi az alkalmazás szerveren.
- Ha sikeres volt a Deploy, akkor az alapértelmezett böngészőben már meg is jelenik a végeredmény, a program használatra kész.

5.3 Megszorítások, egyszerűsítések

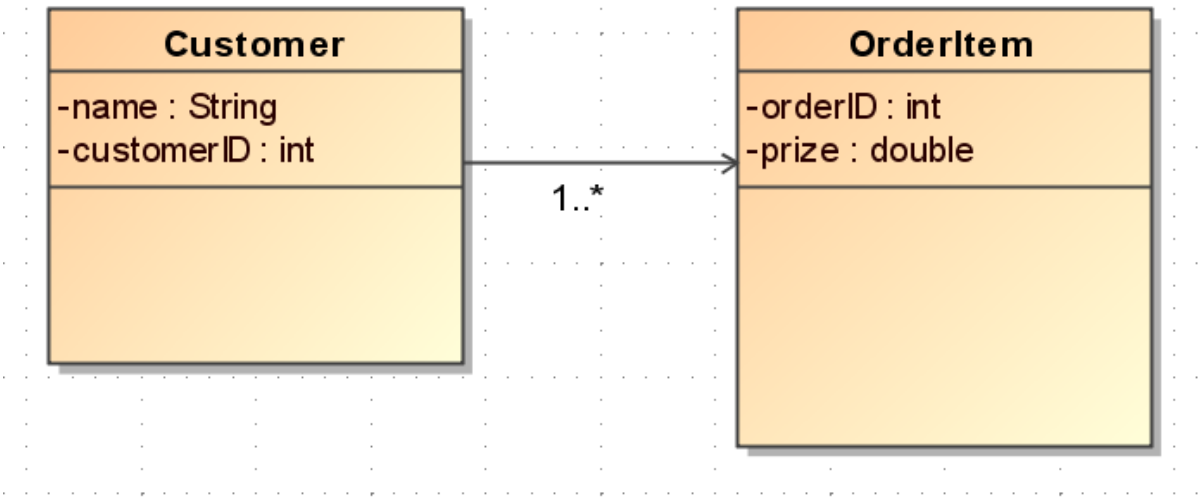
- A kiinduló UML diagramra pár kiinduló megszorítás érvényes, melyet be kell tartani, ha sikeresen akarunk generálni a kifejlesztett eszközeinkkel.
    - Megköveteljük, hogy minden osztálynak legyen egy ID szóra végződő attribútuma. Ez az adatbázisban való tárolásnál fontos, mert ezen attribútum lesz az elsődleges kulcs.
- Használjunk 1:* asszociációt. A CRUD felhasználói felületünkön az ilyen asszociációkkal tudjuk a legjobban demonstrálni a technológia előnyeit, a @OneToMany annotáció felel meg ennek. Legyen navigálható az asszociáció.
- Az generátoraink nincsenek felkészítve többszörös kapcsolatokra és egy felhasználói felülettel összeegyezhetetlen egyéb elemekre, tehát törekedni kell az egyszerű diagramokra.

6. A FEJLESZTÉS MEGVALÓSÍTÁSA

Ebben a részben az előzőleg tárgyalt megkötéseket figyelembe véve, egy konkrét UML diagramból történő CRUD felhasználói felületet generáló eszközrendszer elkészítését célozzuk meg.

6.1 UML DIAGRAM ELKÉSZÍTÉSE ÉS LEGENERÁLÁSA

A MagicDraw modellezőeszközünkkel létrehozunk egy UML osztály diagramot, hasonlót az alábbihoz:



Képezzük le a megszerkesztett Vásárló és Rendelési elem közötti kapcsolatot reprezentáló UML osztály diagramot XMI 2.1 formátumra!

6.2 AZ XMI FELDOLGOZÓ DOKUMENTUMOK ELKÉSZÍTÉSE

Először az adott XMI dokumentumot kell feltérképezni, mivel egy ilyen egyszerű diagram XMI-re leképezve több mint 8000 sorból áll. Az adott dokumentum nagyobb részét nem használjuk, mert teljesen irreleváns pl. MagicDraw-hoz köthető grafikus reprezentációs elemekre vonatkoznak.

Ezek után ki kell nyerni az osztályokat reprezentáló csomópontokat az XMI fájlból, ezekhez egy külön függvényeket tartalmazó XSL fájlt fejlesztettem *szakdoga_func.xsl.*

6.2.1 AZ XMI FELDOLGOZÓ FÜGGVÉNYEK ELKÉSZÍTÉSE

Az XSL függvények tehát egy külön fájlban vannak, melyre a fő XSL fájl *include* segítségével hivatkozik. A függvény névtere a következő:

```
xmlns:fc="http://www.xjoewon.spaces.live.com/">
```

Vizsgáljuk meg ezt a függvényeket tartalmazó fájlt.

```
<xsl:function name="fc:classes" fc:type="Class[]" fc:doc="Class Selection">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:sequence select="$document//packagedElement[@xmi:type='uml:Class' and
./ownedAttribute]"/>
</xsl:function>
```

A fenti függvény az XMI dokumentumból a *packagedElement* elemeket fogja meg, melyeknek az @xmi:type='uml:Class' és van ownedAttribute gyermekük. Az eredeti diagramot nézve világos hogy ez a függvény az összes attribútummal rendelkező osztályt ragadja meg, és rakja be egy szekvenciába. A szekvencia egy csomóponthalmaz, melyben a csomópontok a dokumentumban való előfordulásuk szerint helyezkednek el.

```
<xsl:function name="fc:connections" fc:type="Association.connection[]"
fc:doc="Select all association connections between classes">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:sequence select="$document//packagedElement[@xmi:type='uml:Association']"/>
</xsl:function>
```

Ez az előző függvényhez hasonlóan az összes packagedElement csomópontot kijelöli, és ezeket UML asszociációk és szekvencia formájában adja vissza a függvényt meghívó programrésznek.

Következő lépésként szükségünk van az egyes osztályokat reprezentáló csomópontok attributumainak kinyerésére. A következő módon valósítjuk meg ezt:

```
<xsl:function name="fc:attributes" fc:type="ownedAttribute[]" fc:doc="Select all
non-static attributes">
<xsl:param name="node" fc:type="Class" fc:doc="current node: current()"/>
<xsl:sequence select="$node//ownedAttribute[@xmi:type='uml:Property' and
parent::packagedElement[@xmi:type='uml:Class'] and not (@association)] "/>
</xsl:function>
```

A paraméterként megkapott $node változó tartalmazza az osztályt reprezentáló csomópontot. Ennek az *ownedAttribute* gyermekét kell kinyerni, mely *xmi:type* paraméterének értkénének az *uml:Property* –nek kell lennie és nyilván magának a $node –nak is *Uml:Class* –nak kell lennie. Mivel az attribútum nem lehet asszociáció, ezért van szükség a *not @association* elemre.

Tehát van egy szekvenciánk, mely az attribútum-csomópontokat tartalmazza. Szükségünk van az egyes attribútumok nevére, illetve típusára. A következő függvény gondoskodik egy attribútum nevének kinyeréséről :

```
<xsl:function name="fc:attributeName" fc:type="string" fc:doc="connection id">
        <xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
        <xsl:value-of select="$node/@name"/>
</xsl:function>
```

Ez a viszonylag összetett függvény pedig visszaadja az egyes attribútumkhoz tartozó típusokat:

```
  <xsl:function name="fc:attributeType" fc:type="string" fc:doc="connection id">
        <xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
        <xsl:variable name="type"
select="$node/type/xmi:Extension/referenceExtension/@referentPath"/>
        <xsl:variable name="tokens" select="tokenize($type, '::')"/>
        <xsl:variable name="typeUml" select="$tokens[last()]"/>
        <xsl:value-of select="$typeUml"/>
    </xsl:function>
```

A függvény egy attribútum-csomópontot kap meg, ám annak a *referenceExtension* nevü leszármazott elemének *@referentPath* attribútumában találjuk meg ténylegesen az eredeti attribútumhoz tartozó típust. Ez a *@referentPath* –ban található kifejezés egy összekonkatenált string, melynek legutolsó eleme – mivel máshol nem találtam erre utoló nyomot – tartalmazza a tényleges típust. A kinyert stringet a *tokenize* XSL utasítással string tömbre bontjuk a *::* elhatároló jelek mentén, majd a *last[]* kulcsszó segítségével belerakjuk a legutolsó elemet, vagyis éppen amire szükségünk van. Ezután a *value-of* utasítás segítségével visszaadjuk a kinyert típust. A biztonág kedvéért írtam egy függvényt is, mely az UML típusokat át tudja alakítani más típusokká. Egyfajta típus mappingról van itt szó, melyet az esetleges bővítéseknél felhasználhatunk. (Alkalmazható akár a primitív java típusok átalakítására, de erre rendszerint nincs már szükség.)

```
<xsl:function name="fc:umlToJava" fc:type="string" fc:doc="uml to java type">
<xsl:param name="from" fc:type="string" fc:doc="string to convert"/>
<xsl:value-of select="document('DataType.xml')/mappings/mapping[from=$from]/to"/>
</xsl:function>
```

A *DataType.xml* a következő strukúrát veszi fel :

```
<mappings name="DataType">
    <mapping>
        <from>string</from>
        <to>String</to>
    </mapping>
</mappings>
```

Ismerjük tehát az osztályokat és azoknak attribútumait, így a következő lépés az osztályok közötti asszociációk felkutatása lesz.

```
<xsl:function name="fc:associationNav" fc:type="mdElement[]" fc:doc="Select all
navigable connection">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:sequence
select="$document//mdOwnedViews/mdElement[@elementClass='Association']"/>
</xsl:function>
```

Ez a függvény visszaadja az asszociációk szekvencialistáját.

A generátor a kinyert osztályok szekvenciáján megy végig, ezért szükségünk van olyan függvényekre, amelyek megmondják, hogy egy osztály milyen asszociációban vesz részt, és ott az asszociáció melyik oldalán áll, illetve mely másik osztály szerepel ugyanebbe az asszociációba.

Elöször is az osztálycsomópontoknak van egyedi azonosításra szolgáló attribútumuk. Az asszociációkban azonban nem ezekkel az azonosítókkal fordulnak elő, ezért össze kell őket kötni a következő függvény segítségével. Bementeknek megkap egy osztály csomópont azonosítót és kimenetre visszadja az asszociációkban előforduló annak megfelelő azonosítót. A következőképpen néz ki ez a függvény:

```
<xsl:function name="fc:associationClassId" fc:type="string" fc:doc="connection id">
   <xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
   <xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
   <xsl:variable name="classID" select="$node/@xmi:id"/>
   <xsl:value-of select="$document//mdOwnedViews/mdElement[@elementClass='Class'
and child::elementID[@xmi:idref=$classID]]/@xmi:id"/>
</xsl:function>
```

A paraméterként megadott osztály csomópontok *packagedElement* –ek, de más azonosítóval rájuk hivatkoznak az *mdOwnedViews* csomópontok is. Ezt az azonosítót nyerjük ki ezzel a függvénnyel, mert a későbbiek szükség lesz az *mdElement* csomópontokból kinyert információkra, ugyanis ezek írják le az asszociációk számos fontos tulajdonságát. A következő függvény, felhasználja az előző azonosító kinyerést :

```
<xsl:function name="fc:classInWhichAssociation" fc:type="mdElement" fc:doc="Class
Selection">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
<xsl:value-of select="$document//mdOwnedViews/mdElement[@elementClass='Association'
and (child::linkFirstEndID[@xmi:idref=fc:associationClassId($node,$document)] or
child::linkSecondEndID[@xmi:idref=fc:associationClassId($node,$document)])]/element
ID/@xmi:idref"/></xsl:function>
```

Ez a függvény már az említett *associationClassId* függvény segítségével meghatározza, hogy egy adott *packagedElement* csomópont – mely ugye az osztályunknak felel meg –, milyen asszociációban szerepel, illetve visszaadja az asszociáció azonosítóját is.

Ezen információk birtokában az egyes osztályokon végigmenve meg tudjuk állapítani, hogy az adott osztály kezdő vagy végpontja-e ez asszociációnak, illetve egy adott osztály mellett melyik másik osztály szerepel az asszociációban. Ezek előtt azonban szükséges egy függvény, mely paraméterül megkap egy osztály csomópontot, és visszaadja a csomóponthoz tartozó asszociáció végpontjának azonosítóját.

```
<xsl:function name="fc:associationEnd" fc:type="string" fc:doc="Class Selection">
```

```
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
<xsl:value-of select="$document//packagedElement[@xmi:type='uml:Association' and
@xmi:id=((fc:classInWhichAssociation($document,$node)))]/ownedEnd/@xmi:id"/>
</xsl:function>
```

Ennek ismeretében az előbb tárgyalt függvényt értelmezzük, mely megmondja, hogy egy adott osztály végpont-e vagy sem:

```
<xsl:function name="fc:classIsBoolEnd" fc:type="boolean" fc:doc="Class Selection">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:param name="node" fc:type="packagedElement[@xmi:type='uml:Association']"
fc:doc="current node: current()"/>
<xsl:variable name="tempID"
select="$document//mdOwnedViews/mdElement[@elementClass='Role' and
child::elementID/@xmi:idref=fc:associationEnd($document,$node)]/@xmi:id"/>
<xsl:choose>
<xsl:when test="$document//mdOwnedViews/mdElement[@elementClass='Association']
/associationFirstEndID/@xmi:idref=$tempID and
$document//mdOwnedViews/mdElement[@elementClass='Association']/
linkFirstEndID/@xmi:idref=fc:associationClassId($node,$document)">
<xsl:value-of select="boolean(1)"/>
</xsl:when>
<xsl:when test="$document//mdOwnedViews/mdElement[@elementClass='Association']/
associationSecondEndID/@xmi:idref=$tempID and
$document//mdOwnedViews/mdElement[@elementClass='Association']/
linkSecondEndID/@xmi:idref=fc:associationClassId($node,$document)">
<xsl:value-of select="boolean(1)"/>
</xsl:when><xsl:otherwise><xsl:value-of
select="boolean(0)"/></xsl:otherwise></xsl:choose></xsl:function>
```

Erről a függvényről annyit érdemes tudni, hogy feltételekkel vizsgálja az asszociáció elemeit, és megállapítja, hogy az inputként megkapott osztály csomópont kezdőpont-e vagy sem. Igaz értéket ad vissza, ha az adott csomópont kezdőpont, hamisat, ha az adott csomópont végpont.

A következő függvény megmondja egy osztály csomópont párját az asszociációban :

```
<xsl:function name="fc:getOtherClass" fc:type="mdElement" fc:doc="Class Selection">
<xsl:param name="document" fc:type="root" fc:doc="model xmi document: root(.)"/>
<xsl:param name="node" fc:type="any" fc:doc="current node: current()"/>
<xsl:variable name="tempID"  select="fc:associationClassId($node,$document)"/>
<xsl:choose><xsl:when
test="$document//mdOwnedViews/mdElement[@elementClass='Association'
and child::elementID/@xmi:idref=fc:classInWhichAssociation($document,$node)]/
linkFirstEndID/@xmi:idref=$tempID">
<xsl:value-of select="$document//mdOwnedViews/mdElement[@elementClass='Association'
and child::elementID/@xmi:idref=fc:classInWhichAssociation($document,$node)]/
linkSecondEndID/@xmi:idref"/>
</xsl:when>
<xsl:when test="$document//mdOwnedViews/mdElement[@elementClass='Association'
and child::elementID/@xmi:idref=fc:classInWhichAssociation($document,$node)]
/linkSecondEndID/@xmi:idref=$tempID">
```

```
<xsl:value-of select="$document//mdOwnedViews/mdElement[@elementClass='Association'
and child::elementID/@xmi:idref=fc:classInWhichAssociation($document,$node)]/
linkFirstEndID/@xmi:idref"/></xsl:when></xsl:choose></xsl:function>
```

Ezzel áttekintettük a függvényeket tartalmazó dokumentumot.

6.2.2 A JAVA ELEMEK GENERÁLÁSA

Az előzőekben már vizsgáltuk, hogy egy JSF keretrendszer milyen fájlokat követel meg. Ebben a fejezetben ezeknek az előállításáról lesz szó, melyben felhasználjuk az előző fejezetben tárgyalt függvénykészletet.
A *szakdoga_alap.xsl* fájl-t fogjuk az XSLT transzformátornak megadni, mely legyártja a megfelelő Java elemeket. Ezen XSL fájl egy *for-each* segítségével végigmegy az XMI dokumentum összes osztályán, és minden osztályhoz legenerálja a következő fájlokat:

- [osztálynév].java, az osztály entitása, modellje.
- [osztálynév]Controller.java, az osztály vezérlője.
- [osztálynév]EJB.java, ami a perzisztenciát kezeli.

Érdemes először a *MODEL* elkészítését megvizsgálni. A szokásos *package* és *import* után ellátjuk az osztályt az @ENTITY annotációval, mely jelzi a JPA-nak, hogy egyeddel van dolga, és így lehetővé tesszük az adatbázisban való tárolását. Ezután egy @NamedQuery-t adunk meg, amely hasonlít a megszokott SQL utasításokhoz. Ez egy előre definiált lekérdezés, mely össze van kötve a neki megfelelő entitással. Az EntityManager hívja meg ezt a leképezést, mely az EJB részben inicializálunk, és a perzisztens objektumok kezelésére szolgál. A következő lépésként az attribútumokat generáljuk le a függvényeink segítségével. Amint már az előfeltételeknél feljegyeztük, az ID végződésű attribútumok @ID annotációt kapnak:

```
<xsl:for-each select="fc:attributes(current())">
 <xsl:variable name="attrName" select="fc:attributeName(current())"/>
 <xsl:variable name="attrType" select="fc:umlToJava(fc:attributeType(current()))"/>
          <xsl:choose>
               <!-- @id tag if ends with ID -->
                <xsl:when test="ends-with($attrName,'ID')">
@Id</xsl:when>
              <!-- association declaration -->
          </xsl:choose>
private <xsl:value-of select="concat($attrType,' ') "/>
<xsl:value-of select="$attrName"/>;
</xsl:for-each>
```

A következő lépés az asszociációkhoz tartozó annotációknak és a hozzátartozó attribútumok legenerálása. A fejlesztésünk során az 1:* asszociáció feltárása során a kiinduló osztály modelljében a @OneToMany annotációt használjuk, míg az asszociáció másik végén lévő osztály egy @ManyToOne annotációt kap meg. A következő kódrészlet illusztrálja:

```
@OneToMany(fetch = FetchType.EAGER)
public List<xsl:value-of select="'&lt;'" disable-output-escaping="yes"/>
<xsl:value-of select="fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name"/>
<xsl:value-of select="'&gt; '" disable-output-escaping="yes"/> get
<xsl:value-of select="fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name"/>s(){
return  <xsl:value-of select="fc:lower(fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name)"/>s;}
```

Ahogy korábban említettem, az XSL nem engedi meg, hogy < és > jelet használjunk az elemek között. Ennek megkerülése végett tehát az *xsl:value-of* elemet a *disable-output-escaping="yes"* attribútummal egészítjük ki, és a *select*-ben rendre < és > írunk. Ezzel a modellrész komplexebb területét áttekintettük. A getter, setter metódusok és a Java osztály struktúrájának a legenerálása nem okoz komolyabb gondot, ezekkel részletesebben nem foglalkozunk.

A következő lépés a *CONTROLLER* rész előállítása. Ez végzi a vezérlést, a XHTML oldalak ennek a metódusait hívják meg, illetve ennek a feladata, hogy a megfelelő felhasználó input esetén átnavigáltassa a felhasználót más XHTML oldalra. Mivel ez a kontroller osztály meglehetősen bonyolult, az asszociációk kezdő- és végtagját jelentő osztályokhoz egyedire kell generálni. Emiatt külön *choose* ágba kerül a két kontroller-típus generálását elvégző kódrészlet. A @ManagedBean @SessionScoped [21]annotációkat kell bevezetni. Az előbbi annotáció jelzi, hogy a model részben létrehozott Bean-t fogja kezelni. Az utóbbi az élettartamára ad utalást, ugyanis a SessionScoped annotáció azt jelenti, hogy a Bean tartalma több kérés után is megmarad. Ezt a technikát előszeretettel használják webes áruházak esetében is. A rendelkezésre álló idő után a Bean állapota törlődik. A felhasználói felületünkhöz SessionScoped Bean-t kell használnunk, mert az alkalmazandó – HTML táblának megfelelő – JSF *datatable* komponens ezt kívánja meg. Részlet az XSL kódból:

```
@ManagedBean
@SessionScoped
public class <xsl:value-of select="$className"/>Controller {
private static Logger logger = LoggerFactory.getLogger(<xsl:value-of
select="$className"/>Controller .class);
@EJB
private <xsl:value-of select="$className"/>EJB <xsl:value-of select="concat('
',fc:lower($className))"/>EJB;
private <xsl:value-of
select="fc:associationClassIdToClass(root(.),fc:getOtherClass(root(.),current()))/@
name"/>EJB <xsl:value-of
select="fc:lower(fc:associationClassIdToClass(root(.),fc:getOtherClass(root(.),curr
ent()))/@name)"/>EJB;
private <xsl:value-of
select="fc:associationClassIdToClass(root(.),fc:getOtherClass(root(.),current()))/@
```

```
public String doNew() {
        customer = new Customer();
        return "newCustomer.xhtml";
    }
    public String doCreateCustomer() {
        customer = customerEJB.createCustomer(customer);
        customerList = customerEJB.findCustomer();
        customer = new Customer();
        logger.info("create");
        return "listCustomer.xhtml";
    }
    public String doDeleteCustomer() {
        customer = (Customer) dataTable.getRowData();
        customerEJB.delete(customer);
        return "deleteCustomer.xhtml";
    }
    public String getCurrentList() {
        customerList = customerEJB.findCustomer();
        logger.info("refresh table");
        return "listCustomer.xhtml";
    }
    public String UpdateCustomer() {
        customer = (Customer) dataTable.getRowData();
        return "updateCustomer.xhtml";
    }
    public String doUpdateCustomer() {
        customerEJB.update(customer);
        customerList = customerEJB.findCustomer();
        logger.info("update");
        return "listCustomer.xhtml";
    }
    public String doCreateOrderItem() {
        customer = (Customer) dataTable.getRowData();
        FacesUtil.setApplicationMapValue("customer", customer);
        return "newOrderItem.xhtml";
    }
```

A kódot vizsgálva feltűnhet a *customer = (Customer) dataTable.getRowData()* kódrészlet, amely a későbbiekben ismertetett *datatable* egy adott sorát, mint a model részben definiált objektumot adja vissza. Ezt átadhatjuk az EJB-nek, hogy mentse el. Importálunk kell a *javax.faces.component.html.HtmlDataTable*, hogy használni tudjuk. Az egyes metódusok *return* része valósítja meg a navigációt. Ezt a navigációs lépést eredetileg a WEB-INF könyvtár *faces-config.xml dokumentumába* kellett regisztrálni, de megengedett ilyen módon return kifejezés használatával is. Az annotációkat is eredetileg külön fájlba kellett regisztrálni, de a Java 6 újítása ez a beágyazott annotáció, melyet mi is használunk a projekt során (@EJB)

A *FacesUtil.setApplicationMapValue("customer", customer);* kódrészlet, egy *customer* objektumot rak be egy MAP-be, melyet bárhonnan elérhetünk a továbbiakban.

Legutoljára maradt az EJB Bean legenerálása. Ez a Bean *@stateless*, azaz nem őrzi meg az állapotát. Az EntityManagert ez a Bean hívja meg, és elhelyezi a metódusain keresztül

megkapott Entitásokat az adatbázisba. Ez a megfelelő perzisztens rétegen keresztül történik. A projektünk során az EclipseLink JPA perzisztencia kezelőt használjuk, mely számtalan új funkciót nyújt, és kiemelten támogatja a vezető Java technológiákat és relációs adatbázisokat. Következzen a teljes osztály (importok és package nélkül):

```
@Stateless
public class CustomerEJB {
             // ====================================
             // =             Attributes           =
             // ====================================

@PersistenceContext(unitName = "crud")
private EntityManager em;
             // ====================================
             // =            Methods          =
             // ====================================

         public List<Customer> findCustomer(){
         Query query = em.createNamedQuery("findAllCustomer");
         return query.getResultList();}
         public Customer createCustomer(Customer customer) {
         em.persist(customer);
         return customer;}
         public Customer create(Customer t) {
         this.em.persist(t);
         return t;
         }
         public void delete(Customer t) {
         t = this.em.merge(t);
         this.em.remove(t);
         }
         public Customer find(Integer id) {
         return this.em.find(Customer.class, id);
         }
         public Customer update(Customer t) {
         return this.em.merge(t);
         }}
```

A *@PersistenceContext(unitName= "crud")* a perzisztencia kontextusa, gyakorlatilag ennek meg kell egyeznie a *persistence.xml* –ben definiált <persistence-unit name="crud" transaction-type="JTA"> name attribútumával. Ez a példaalkalmazásunkban meg is egyezik. Az EntityManager-t deklarálni kell.

A *findCustomer* függvény az Entity Bean-ben definiált NamedQuery-t az Entity Manager segítségével végrehatja, és a függvény visszaadja az eredményhalmazt a meghívó kontrollernek. Ezt a későbbiekben egy táblázat kereteibe illeszti be. A következő négy metódus a CRUD műveleteket valósítják meg az Entity Manager segítségével, a kontroller hívja meg ezeket a metódusokat, és adja át az adatbázisban tárolandó Entitásokat. Ennek a legenerálása nem okozhat különösebb gondot, ha már a többit sikeresen előállítottuk.

6.2.3 A HTML ALAPÚ ELEMEK GENERÁLÁSA

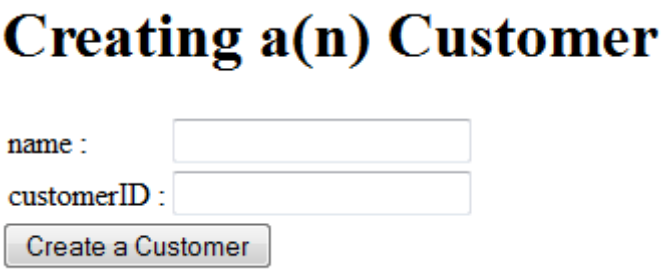
A felhasználói felület megjelenését ezek az elemek határozzák meg. A felhasználó a weblapokon keresztül kommunikál a kontrollerrel a JSF komponensek segítségével. Tekintsük meg a generálásukhoz elkészített XSLT fájlt. A Java fájl generátor text formátumba generált, ezzel szemben most HTML formátumú kimentet fogunk generálni, melyet az ismert módon állítunk be:

```
<xsl:output name="html" method="html" version="4.0" encoding="utf-8" indent="yes"/>
```

Elsőnek vizsgáljuk, meg a **new[Osztálynév].XHTML** fájlt, mely az adatbevitelt valósítja meg. Ezt az XSLT fájlunk a web-app könyvtárba generálja a többi HTML szabvány fájllal együtt. A következő HTML lapot állítja elő a generátor:

```
<html xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
xmlns:fc="http://www.xjoewon.spaces.live.com/"
xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core">
<h:head>
      <title>Customer</title>
</h:head>
 <h:body>
  <h1>Creating a(n) Customer</h1>
   <h:form>
     <table border="0">
        <tr>
           <td><h:outputLabel value="name : "/></td>
           <td><h:inputText value="#{customerController.customer.name}" /></td>
        </tr>
        <tr>
           <td><h:outputLabel value="customerID : "/></td>
           <td><h:inputText value="#{customerController.customer.customerID}/>
           </td>
        </tr>
</table><h:commandButton value="Create a Customer"
action="#{customerController.doCreateCustomer}"/>
      </h:form>
   </h:body>
</html>
```

Ez a következőképpen jelenik meg:



Elsőnek a névtereket kell beállítani:

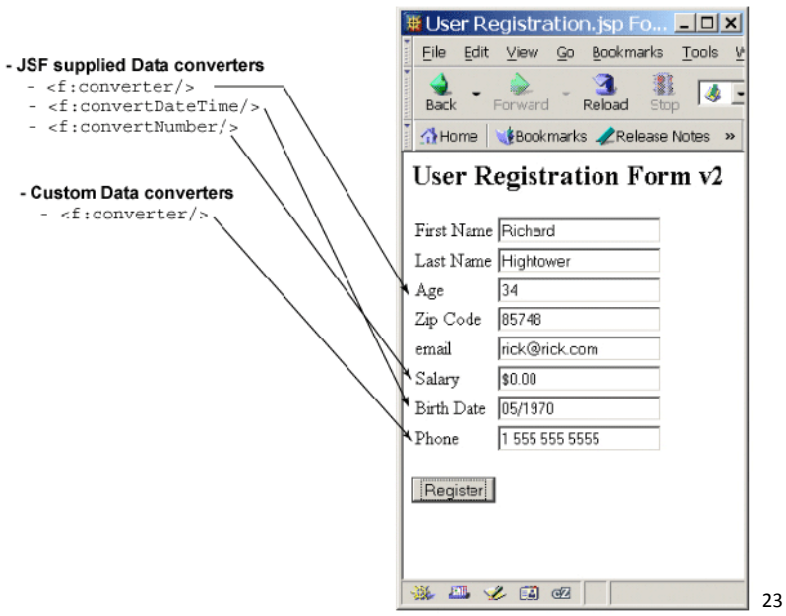
- `xmlns:h="http://java.sun.com/jsf/html"`
- `xmlns:f="http://java.sun.com/jsf/core"`

Ezután lehetőségünk van JSF html tagokat, illetve core tagokat használni. A h:form-on belül tudunk inputot átküldeni a kontrollernek.

A *<h:inputText value="#{customerController.customer.name}">* tag teszi lehetővé, hogy inputot írjunk be, és az a value részen belül megadott EntityBean attribútumhoz rendelődjön hozzá az input. A commandButton aktiválása után a vezérlés átkerül a *value* részében megnevezett metódushoz, és véglegesítődik a kötés az inputként megkapott InputText értékek és az EntityBean között. A kontrollerben ezek után kezelhetjük a létrejött Entitást. Mivel a JSF technológia rendkívül korszerű, ezért számos lehetőséget nyújt az adatok manipulálására, a beérkező adatok konverziójára és nem utolsó sorban az input szerver oldali validációjára. Használhatunk beépített validátorokat, illetve létrehozhatunk magunknak az oldal BackingBean-jébe (Entity). Léteznek ugyancsak beépített konverterek, ám saját magunk is létrehozhatunk egyedi konvertereket. Tekintsünk egy beépített validátor példakódját:

```
<h:inputText id="firstName" value="#{UserRegistration.user.firstName}">
<f:validateLength minimum="2" maximum="25" /> </h:inputText>[22]
```

Ez a firstName ID-val rendelkező inputText-nek az input hosszát szabja meg 2 és 25 karakter közé.
A következő ábra bemutatja a beépített konverzió használatát:



[23]

5. ábra

Nézzünk egy kód példát erre is:

```
<h:inputText id="phone" value="#{UserRegistration.user.phone}">
        <f:converter  converterId="arcmind.PhoneConverter" />
</h:inputText>
```

A következő HTML elem, melyet vizsgálunk a **list[Osztálynév].XHTML** elem. Ez a lap megjeleníti az előzőleg inputként megadott Entitásokat egy *h:dataTable* soraként.

Ez a generált oldal az entitások soronkénti megjelenítése mellett lehetőséget nyújt az egyes elemek törlésére, módosítására. Ezen kívül az asszociációknak megfelelően pl. a vásárlónak adhatunk megrendeléseket, illetve visszanavigálhatunk és felvehetünk új Entitásokat. Mivel osztályonként generáljuk a weblapokat, ezért az asszociációk kezdő- és végelemeit itt is külön kell kezelni. Értelemszerűen az asszociáció végpontjához tartozó osztályból generált lapnak nem lehetnek hozzáadó funkciói, ezért a *h:dataTable* –nek kevesebb oszlopa lesz . Továbbá egy asszociáció végpontját jelentő lapból kell egy navigációs lehetőség az asszociáció kezdőpontja felé is.
Nézzük az előbbi eset generáló kódjának részletét:

```
<xsl:choose>
<xsl:when test="fc:classIsBoolEnd(root(.),current())=boolean(1)">
<h:column> <f:facet name="header">
<xsl:value-of select="'&lt;h:outputText value='" disable-output
escaping="yes"/>"ADD "/
<xsl:value-of select="'&gt;'" disable-output-escaping="yes"/>
</f:facet>  <xsl:value-of select="fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name"/><xsl:value-of
select="'&lt;h:commandLink action='" disable-output-escaping="yes"/>"#{<xsl:value-
ofselect="fc:lower($className)"/>Controller.doCreate<xsl:value-of
select="fc:associationClassIdToClass(root(.),fc:getOtherClass(root(.),current()))/@
name"/>}"<xsl:value-of select="'&gt;'" disable-output-escaping="yes"/>
click  <xsl:value-of select="'&lt;/h:commandLink&gt;'" disable-output-
escaping="yes"/></h:column>
```

Az *xsl:when* biztosítja, hogy csak az asszociációk kezdőpontjai tartalmazzák ezt az oszlopot. Érdemes még megfigyelni, hogy itt is használni kell a < és > helyett a már említett megoldásokat. Ez a kód rész pedig a visszanavigálási linket teszi feltételessé :

```
<xsl:choose>
<xsl:when test="fc:classIsBoolEnd(root(.),current())=boolean(0)">
<br></br>
<xsl:value-of select="'&lt;h:commandLink action='" disable-output-
escaping="yes"/>"#{<xsl:value-of select="fc:lower($className)"/>
Controller.backTo<xsl:value-of select="fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name"/>}"
<xsl:value-of select="'&gt;'" disable-output-escaping="yes"/>
Back to <xsl:value-of select="fc:associationClassIdToClass(root(.),
fc:getOtherClass(root(.),current()))/@name"/>
<xsl:value-of select="'&lt;/h:commandLink&gt;'" disable-output-escaping="yes"/>
</xsl:when> </xsl:choose>
```

Annak érdekében, hogy a kilistázott entitásokat manipulálni tudjuk, a sorokat át kell tudni adni a kontrollernek. A *h:dataTable* lehetőséget ad a lista tartalmának egy kontrollerbeli változóhoz való hozzárendelésére:

```
<h:dataTable value="#{customerController.customerList}" var="entity" border="1"
binding="#{customerController.dataTable}" >
```

Ennek segítségével, ha egy adott sor oszlopában kattintunk a *CommandLink*-re, akkor az adott oszlopot mint Entity-t kinyerhetjük a kontrollerben a már tárgyalt módon:
customer = (Customer) dataTable.getRowData();

Az XSLT fájl ezeken túl legenerál még osztályonként két lapot, melyek a navigációt segítik – ezek ismertetése viszont már nem nyújt újdonságot –, illetve előállít egy default.htm fájlt, mely a webalkalmazás elindításakor a böngészőt átirányítja automatikusan a mi általunk legenerált lapokra.

7. Összegzés

A fejlesztés során sikerült megvalósítani az eltervezett célt: egy olyan webes felületet hoztunk létre, mely korszerű és az igények szerint könnyen módosítható. A fejlesztés során előállított CRUD interfész – a mögötte álló perzisztens technológiák segítségével – biztos adattárolást nyújt, és a tranzakciók kezelése is biztosított. Lehetőségünk van beérkező adatok szűrésére, átalakítására, illetve szükség esetén komolyabb üzleti logika is implementálható. Az előállt webes felület viszonylag egyszerű, de az előállításukra felhasznált technológiák lehetővé teszik a későbbi továbbfejlesztést. A bemutatott fejlesztés nem kezel minden eshetőséget, ami a kiindulási UML Diagramnál előfordulhat pl. többszörös asszociációk, öröklődések, illetve az előálló kezelőfelülethez sem biztosít alap konvertáló és validáló eszközöket. Egy következő projekt témája lehetne a felhasználói felületet átalakítása valamely ismert JSF kiegészítő komponens könyvtár alkalmazásával, mint pl. OpenFaces, RichFaces. A közkedvelt AJAX technológia bevezetése is meglehetősen javítaná a felhasználói felületünk megjelenését és funkcionalitását.

8. FELHASZNÁLT IRODALOM:

Könyvek :

- Antonio Goncalves, Beginning Java EE 6 Platform with Glassfish 3 from Novice to Professional, Apress, 2009.
- O'Reilly Media , Mastering XML Transformations , O'Reilly, 2008
- Wiley Publishing, Mastering JavaServer Faces, Wiley, 2004

Internetes források:

[1] http://hu.wikipedia.org/wiki/XML
[2] http://www.w3.org/TR/xml11/
[3] http://tldp.fsf.hu/Forditas-HOGYAN/sgmlinfo.htm
[4] Polgári Ferenc - Objektumorientált tervezés: UML osztálydiagramok
[5] http://xml.coverpages.org/wrightson-xminotes.html
[6] ww.inf.unideb.hu/~jeszy/download/xml/xpath-2x2.pdf
[7] http://avalon.aut.bme.hu/education/szakirany/infote/infmegj/xml.pdf
[8] http://www.w3schools.com/xsl/
[9] http://avalon.aut.bme.hu/education/szakirany/infote/infmegj/xml.pdf
[10] http://www.sulinet.hu/tart/fncikk/Kacn/0/24761/index.html
[11] http://en.wikipedia.org/wiki/XHTML#XHTML_2.0
[12] http://infokristaly.hu/hu/node/531
[13] http://www.roseindia.net/struts/mvc-architecture.shtml
[14] http://www.javaforum.hu/javaforum/7/cikkek/cikkek/9/show/jsf/netdiaghu/8/pm/edit
[15] http://3.bp.blogspot.com/_zMrw6uqe_Ig/Rh_LEGXBM-I/AAAAAAAAABc/5vTVEj5Cozc/s320/jsf2.png
[16] http://www.magicdraw.com
[17] http://netbeans.org
[18] https://glassfish.dev.java.net/
[19] http://db.apache.org/derby/
[20] http://www.javaforum.hu/javaforum/7/tippek/action/tippek_trukkok_praktikak/tippek_trukkok_praktikak/72/action/news.Detail/show/eclipselink_cache_coordination_jms_segitsegevel
[21] http://www.oracle.com/technology/tech/java/newsletter/articles/jsf_pojo/index.html
[22] http://www.ibm.com/developerworks/java/library/j-jsf3/
[23] http://www.ibm.com/developerworks/java/library/j-jsf3/fig8-sample-app.gif