Web alapú Információs Rendszerek modellezése

Doktori (PhD) értekezés

Adamkó Attila

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika és Számítástudományok Doktori Iskola

Debrecen, 2008

Web alapú Információs Rendszerek modellezése

Doktori (PhD) értekezés

Adamkó Attila

Debreceni Egyetem
Természettudományi Doktori Tanács
Matematika és Számítástudományok Doktori Iskola

Debrecen, 2008. március

Ezen értekezést a Debreceni Egyetem Természettudományi Doktori Tanács Matematika és Számítástudományok Doktori Iskola Informatika programja keretében készítettem a Debreceni Egyetem természettudományi doktori (PhD) fokozatának elnyerése céljából.

Debrecen, 2008. március 2.

Adamkó Attila
doktorjelölt

Tanúsítom, hogy Adamkó Attila doktorjelölt a fent megnevezett Doktori Iskola Informatika programjának keretében irányításommal végezte munkáját. Az értekezésben foglalt eredményekhez a jelölt önálló alkotó tevékenységével meghatározóan hozzájárult. Az értekezés elfogadását javasolom.

Debrecen, 2008. március 2.

Dr. Fazekas Gábor
témavezető

Tartalomjegyzék

Bevezetés

Az Internet rövid idő alatt nagyon széles körben terjedt el, bekapcsolódott az élet számos területére, egy új irányvonalat hozott létre az alkalmazásfejlesztésben. A növekedés jelentős hatással van az üzleti életre, az iparra, az oktatásra és személyes életünkre is. A világháló előreláthatóan környezetünk egyik meghatározó tényezője lesz, és nagy valószínűséggel egyre szélesebb körű lesz a weben[1] történő megjelenés.

A weboldalak fokozatosan alakulnak át az egyszerű, csak böngészhető információforrásokból web alapú elosztott alkalmazásokká. A hagyományos weboldalakkal összevetve a webalkalmazások már nemcsak a böngészést támogatják, hanem olyan műveleteket is biztosítanak, amelyek befolyásolják a megjelenítendő tartalmat, a navigációt, illetve a továbbiakban végezhető műveleteket.

A világháló a kiindulási formájához képest egy teljesen új környezetté alakult át. Az 1989-es indulásakor még az információ (kutatási eredmények, dokumentációk,…) megosztásának az eszközének számított. Az információ egyszeres szöveges állományokból állt, melyeket hiperhivatkozásokkal kapcsoltak össze. Mára azonban egy olyan platform lett, amely számos különböző típusú alkalmazás felhasználási jellegét ötvözi. Ezen alkalmazások az egészen apró méretektől a nagyon komplex vállalati szintű együttműködési méretekig húzódhatnak. Az Internet és a ráépülő technológiák mára meghatározó szerepet töltenek be az információszerzés és a tartalomszolgáltatás területén is. Ma már kevesen vannak, akik még nem kerültek kapcsolatba a világhálóval. A web technológiák pedig lehetővé teszik a felhasználók számára, hogy csupán egy böngésző segítségével elérhessék a számukra szükséges információkat. Ehhez viszont szükséges a kiszolgáló oldali támogatottság, azaz az információnak elérhetőnek kell lennie, amelyhez az információs rendszerek nyújtanak hatékony támogatást.

[1] Web alatt a világhálót értjük, de helyette a magyar nyelvben többé-kevésbé meghonosodott és praktikusabb „web” elnevezést fogjuk használni mind főnévi, mind jelzős szerkezetekben.

Az egyre komplexebb informatikai rendszerek pedig egyre hatékonyabb kezelhetőséget biztosítanak. Platform független szabványok alkalmazásával szinte teljes mértékben el lehet rejteni a kliensek elől az implementációs részleteket. A klienseknek nem kell tudniuk, hogy a rendszert milyen módon fejlesztették, milyen operációs rendszer alatt fut, nekik csak az internetcímet kell ismerniük a szolgáltatás eléréséhez. Ezen célok elérése már komolyabb feladat. A szoftverrendszerek fejlesztésének számos irányzata van, amelyek a hatékony és jó minőségű szoftverek előállítását célozzák meg.

Ezek közül mára ez egyik legjelentősebb irányzattá nőtte ki magát a modelleken alapuló rendszerfejlesztési irányzat. Természetesen ezen belül is széles skálán lehet mozogni a különböző megközelítések között, azonban mindegyikben közös, hogy valamilyen módon a vizsgált problémakör szakterületi modelljét állítják elő.

A fejlesztés lépései kapcsolódnak a hagyományos rendszerfejlesztésnél megismert jelentősebb lépésekhez, azonban már magában a kivitelezésben a terület sokrétűsége miatt eltérnek. Ennek a területnek a vizsgálata azonban még közel sem teljes. A webalkalmazások esetén pedig ez a kijelentés még határozottabban igaz.

A webalkalmazások fejlesztése számos olyan kérdést is felvet, amelyekkel a hagyományos szoftverfejlesztés során nem kell számolnunk. Ezek közül az egyik legjelentősebb az állandó technológiai változásoknak a hatása. Nagyon gyorsan alakulnak ki és terjednek el egyre újabb és újabb technológiák, amelyek még hatékonyabb alkalmazásfejlesztést tesznek lehetővé. Ezért a webfejlesztéseknek fel kell készülniük a változásokra. A másik fontos szempont a fejlesztésre szánható idő mennyisége. Ez webalkalmazások esetén legfeljebb három, esetleg négy hónap lehet. A harmadik a fejlesztéshez szükséges szakismeret. Miután a webalkalmazások számos technológiát használnak a működésükhöz, a fejlesztéshez ezen irányzatok mindegyikét ismerni kell, ami általában számos különböző területről érkező ember együttműködését kívánja meg. A negyedik pedig a felhasználók széles köre. Míg a hagyományos szoftverek esetén előre meghatározható, hogy a terméket kik fogják használni, addig a webalkalmazások esetén ez már nehezebben megfogható.

Ezen indokok alapján már érezhető, hogy miért van szükség a webalkalmazások fejlesztésének a hatékony támogatására. Természetesen a webalkalmazások is több kategóriába sorolhatóak. A dolgozat alapjául ezen irányzatok egyike, a web alapú információs rendszerek modell alapú fejlesztése szolgál.

Célkitűzés

Célunk egy olyan módszertan kidolgozása, amely hatékonyan tudja támogatni és segíteni a kis- és közepes méretű web alapú információs rendszerek fejlesztését és előállítását. Ez a terület még igen fiatalnak számít a szoftverrendszerek fejlesztésének területén, ezért a fejlesztését támogató módszerek iránt mind nagyobb igény jelentkezik. Hatékony segítségre van szükség a navigáció, a prezentáció, illetve a felhasználói felület tervezésénél, amely még a nagy, vállalati szintű alkalmazások többrétegű architektúrájához kapcsolódó alkalmazásszerver szintű támogatottságból is hiányzik.

A web alapú rendszerek fejlesztése többnyire még mindig ad hoc jellegű, valamint a szoros határidők miatt kiesik a fejlesztés módszeres megközelítése, hiányoznak a rendszertervek, problematikus a dokumentálás, illetve hiányos a dokumentáció. A gyors tervezésnek és implementációnak pedig többnyire káros, csak a későbbiekben, a használat során jelentkező hatásai vannak.

A dolgozatban elsősorban az adatorientált webalkalmazások tervezéséhez, modellezéséhez szükséges technológiák, tervezési minták kerülnek áttekintésre, rámutatva a jelenlegi módszerekben és fejlesztési irányokban rejlő problémákra. Támpontokat adunk a fejlesztés lépéseihez, illetve a modellek elkészítéséhez.

Bemutatunk egy olyan tervezési stratégiát, amely a modell alapú szoftverfejlesztés paradigmáját követve hatékonyan segíti az adatorientált webalkalmazások modellezését, valamint a modellekből történő automatikus kódgenerálást. Ehhez a tervezés során az UML, az implementálás során pedig az XML technológiákat alkalmazzuk. Ezek segítségével eredményesen tudjuk támogatni az újrafelhasználhatóságot mind a modellezés, mind pedig a program elkészítése során.

Motiváció

A kiindulási rendszer

Pár évvel ezelőtt a Debreceni Egyetem Matematikai és Számítástudományok Doktori Iskolája elhatározta, hogy a nyilvános adatok weben keresztül történő elérhetőségét megvalósítja. A kialakítandó webalkalmazással szemben támasztott követelmények között szerepelt a Doktori Iskola oktatóinak és hallgatóinak nyilvántartási rendszerbe vétele, a nyomon követése, illetve a doktori programok és kurzusok menedzselhetősége. Mindamellett, hogy lehetővé vált az adatok rendszerezett formában történő elérése, különböző statisztikai kimutatások készítésére is lehetőség nyílt. Továbbá a hallgatók és az oktatók számára rendelkezésre állt a publikációk egyszerűsített kezelése is. A fentebb szereplő táblázat alapján ezt a rendszert az információs rendszerek közé sorolhatjuk, mert interaktívan használható a különböző feladatok ellátására, melyek magukban foglalják az adatfeltöltést, a módosítást és a lekérdezéseket.

A rendszer alapját egy PostgreSQL adatbáziskezelő rendszer szolgáltatta, míg az elérhetőségét a HTML oldalakba ágyazott Perl nyelven írodott CGI szkriptek biztosították. Az alkalmazásréteg több rétegből állt, kihasználva a PL/pgSQL tárolt eljárásokat az adatbáziskezelőn keresztül. Mára ezen technológiák már nagyon elhalványultak, ami annak az eredménye, hogy nem támogatják az adat szerkezetének és megjelenítésének a magas szinten történő kettéválasztását.

Problémák

A rendszer üzemeltetése során új követelmények fogalmazódtak meg. Ezek egy része könnyen implementálható volt, viszont megjelent olyan követelmény is, amely a teljes adatstruktúra újratervezését igényelte volna. Ezek többsége a helyi szabályozások változására és a szervezeti átalakulásra vezethető vissza. Az adatmodell változásával azonban további problémák is megjelentek, amelyek a rendszer többi részére is kihatottak. Ezek alapján körvonalazódott, hogy a rendszer karbantartása és további fejlesztése nagyon összetett és nehezen megvalósítható. Ennek következményeképp fogalmazódott meg a teljes rendszer újragondolása.

Gyakori eset, hogy a problémák jelentős része a követelmények fejlődésével (evolúciójával) jelenik meg. A nehézségek már a karbantartásnál jelentkezhetnek, mert a fiatal fejlesztők nem feltétlenül rendelkeznek a régebbi technológiákhoz szükséges ismeretekkel, de ha mégis, akkor a változtatások átvezetése könnyedén vezethet nem megfelelő tervezési és/vagy kódolási stratégiához. Mindezek alapján döntöttünk úgy, hogy egy modell alapú tervezési stratégiát alkalmazunk a fejlesztési idő és a karbantartási költségek csökkentése érdekében. A dolgozatban az egyes témaköröknél ennek a rendszernek a példáján keresztül fogjuk bemutatni a fejlesztési fázisokat.

Áttekintés

Web Engineering

A gyenge teljesítmény és minőség elkerülésére, a kockázatok minimalizálására, illetve a webalkalmazások sikeresebb fejlesztéséhez szükségünk van hatékonyabb módszerekre és szemléletmódokra a web alapú rendszerek fejlesztésének és implementációjának minden fázisában. Természetesen a hagyományos és a webes szoftverfejlesztés között számos eltérés van, gondoljunk csak arra, hogy egy webalkalmazás több, különböző tudományágat átfogó irányzatot ötvöz (mint például rendszerelemzés és tervezés, hipermédia és hipertext, követelmény elemzés, ember-gép kapcsolat, felhasználói felület tervezés, tesztelés, modellezés) [1].

A web alapú rendszerek fejlesztésénél, karbantartásánál, illetve a kialakult irányzatok és alapelvek alkalmazása során szerzett tudást és tapasztalat összefogása révén alakult ki a Web Engineering fogalma. Definíciójaként pedig a következőt találjuk a Web Enginerering közösségi portálján [2]:

„*A Web Engineering a szisztematikus és követhető irányzatok alkalmazása a kiváló minőségű Világhálós alkalmazások költség-hatékony fejlesztéséhez és fejlődéséhez.*"

Jól érthető módszertanok iránti igény egyre nagyobb lett, mind több és több szervezet használja a világhálót, mint üzleti eszközt. Természetesen a webalkalmazások igen komplexek, és számos sajátossággal bírnak, így univerzális megoldást találni jelenleg nem – és talán a későbbiekben sem – lehet. Az elkészítendő webalkalmazás alapvető karakterisztikája viszont a fejlesztés elején már meghatározható, és hét különböző kategóriába sorolható:

- Informális
- Interaktív
- Kereskedelmi
- Munkafolyamat kezelő
- Együttműködési környezet
- Közösségi
- Portál

Bár egy kifejlesztendő webalkalmazás jellege alapján egyidejűleg több kategóriába is besorolható, valamint a későbbiekben még új funkcionalitásokkal is bővülhet, így összességében egy webalkalmazást több kategória fog együttesen jellemezni. Továbbá az esetek túlnyomó részénél a fejlesztési folyamat kezdetén még nem lehetséges pontosan definiálni, hogy a weboldalnak mit és hogyan kell tartalmazniuk, mert struktúrájuk és funkcionalitásuk az idő előrehaladtával fog kialakulni. Ez természetes, mert az ilyen alkalmazásokat többnyire a felhasználók olyan csoportja fogja használni, amelyeket még nem ismerünk, így nem tudjuk, milyen elvárásaik vannak a jövőbeli rendszerrel szemben. Másrészt egy internetes weboldalnak előnyére válik, ha bizonyos időközönként fejlődik, új funkciókkal bővül, megújul.

Web alapú Információs Rendszerek

A Web alapú Információs Rendszerek (*WIS*) a webalkalmazások egy speciális fajtája, amely a fenti felsorolást alapul véve a következő kategóriákba sorolható: információs, interaktív és kereskedelemmel kapcsolatos. Jellemzésére pedig a következők igazak:

- Információs Rendszer,
- elosztott alkalmazás,
- a kliens/szerver alkalmazások speciális esete, melyben a funkcionalitás nagy része a szerver-oldalra kerül,
- a weben, mint alkalmazási infrastruktúrán alapszik,
- a felhasználói kommunikációk webes interfészeken keresztül történnek.

A fent említett jellemzők előrevetítik, hogy egy Web alapú Információs Rendszer tervezése, fejlesztése és végül implementálása mennyire eltér a hagyományos szoftverfejlesztéstől. A modell elkészítéshez több különböző megközelítés is létezik – a maguk előnyeivel és hátrányaival. Van ahol a modell a hangsúlyos, van ahol a fejlesztési folyamat egyes fázisai, és van ahol a felhasználó a központi elem.

A mi szempontunkból egy Web alapú Információs Rendszer egy számítógép által támogatott információs rendszer, amely a világháló nyújtotta lehetőségeket kihasználja. Ezen rendszerek a felhasználóval folytatott kommunikáció és információ jellege alapján a következő kategóriákba sorolhatjuk:

- Információ szolgáltató
- Információs rendszer
- Közösségi
- Hirdetési.

A fenti szempontok alapján pedig a következő kis táblázatot állíthatjuk össze:

	Aszimmetrikus kommunikáció	Szimmetrikus kommunikáció
Objektív információ	**Információ szolgáltató**	**Információs rendszer**
Befolyásoló információ	**Hirdetési**	**Közösségi**

1. ábra: Az Információs rendszerek perspektívái

Miután az általunk vizsgált rendszerek elsődleges célja az információ szolgáltatása és az adatkezelést támogató feladatok támogatása, a továbbiakban a ténylegesen információs rendszerként jellemzett ággal foglalkozunk. A korábbiakban ugyan erre az ágra adat-orientált webalkalmazásként is hivatkoztunk.

A dolgozat következő részei ezen rendszerek modellezési szemléletmódjairól nyújtanak rövid áttekintést, bemutatva a webalkalmazások általános architektúráját és a fejlesztési folyamatokat, valamint a jelenleg elérhető módszereket és tervezési stratégiákat, kiemelve az architektúra és a tervezési stratégiák között található problémákat. Céljaink könnyebb eléréséhez (mint például modularitás, újrafelhasználhatóság, környezetfüggő megjelenítés) az implementáció során alkalmazhatjuk az XML technológiákat – bár az XML dokumentumok tárolása további problémákat vet fel. Az architektúra és a tervezési minták összekapcsolására pedig bemutatunk egy új szemszöget, miként célszerű az alkalmazás logikát (más néven az üzleti logikát) tartalmazó réteget két részre osztani: a tiszta alkalmazás logikára, amely a munkafolyamatok felügyeletére hivatott, valamint a tartományi logikára, amely felelős a szakterületi modell működtetéséért.

Webalkalmazások általános architektúrája

A webalkalmazások több feladatot kapcsolnak össze (szoftverfejlesztés, adatbázis modellezés, hálózati programozás, hatékony felhasználói felület tervezés) és számos, különböző típusú információforrást kezelnek (szöveg, grafika, hang, videó), melyek feldolgozása, tárolása, megjelenítése is feladatuk. Architektúrájukat pedig talán úgy jellemezhetjük a legjobban, hogy a kliens/szerver modellek egy speciális formája, ahol a kliensek „vékonyak", és a szerveroldal felelős az alkalmazás teljes funkcionalitásának a kezeléséért.

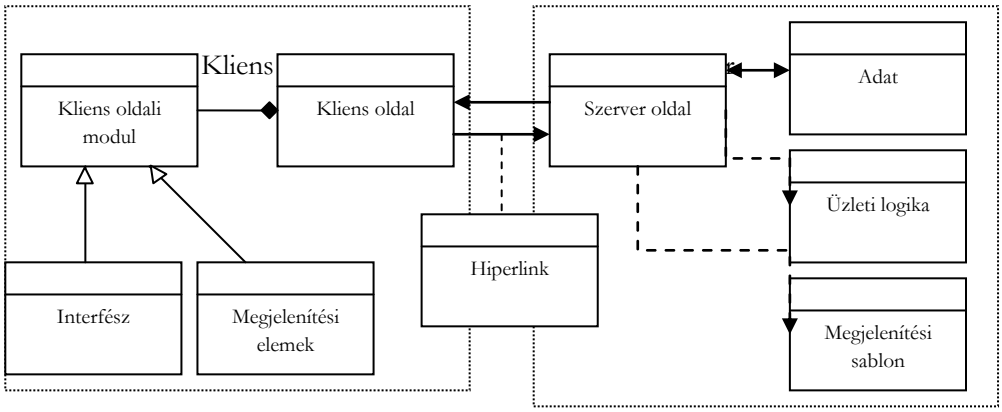
A felhasználóval webes felületeken keresztül történik a kommunikáció, amely három lépésre bontható:

- Kérés: a felhasználó elküldi a kérést a szervernek, többnyire egy weboldalon keresztül.
- Feldolgozás: a szerver fogadja a kérést, és különböző műveleteket hajt végre. Ezt követően az eredmény (általában egy új oldal) továbbítódik a klienshez.
- Válasz: a felhasználó böngészője megjeleníti a kérés eredményét.

Kezdetben a weboldalak **statikus**ak voltak, a szerver lényegében csak a beérkező kéréseknek megfelelő, előre elkészített, HTML nyelven íródott oldalak kiszolgálását végezte. Ez a módszer kimondottan rugalmatlanul kezeli például az azonos tartalom több, különböző oldalon történő megjelenítését. A cél eléréshez a tartalmat mindegyik oldalon meg kellett ismételni, ezzel redundánssá, és nehezen karbantarthatóvá téve az adatokat, valamint magukat az oldalakat.

A megoldást a **dinamikus modell** kidolgozása jelentette, további feladatot róva a szerveroldalra. Az új weboldalak továbbra is HTML nyelven íródnak, de már tartalmazhatnak olyan utasításokat, amelyeket a szerver az oldal kiszolgálása során értelmez és végrehajt. Elérhetővé vált, hogy a tartalom egy központi adatbázisból származzon, megkönnyítve az adatok karbantartását, kiiktatva a redundanciát. További előnye a dinamikus modellnek, hogy egy weboldal több, kisebb részből állítható össze, így például elegendő egy-egy állományban elhelyezni a weboldalak egységes megjelenését biztosító fejlécét, láblécét, amelyeket az oldalak beemelnek a kiszolgálás során, tovább csökkentve a redundanciát.

A következő ábrán ennek a dinamikus modellnek – nem minden részletre kiterjedő – sematikus rajza látható.



2. ábra: Webalkalmazások egy absztrakt modellje

A modellnek megfelelően a jelenlegi webalkalmazások architektúráját ezért több réteg alkotja. Legalsó szinten található az adathozzáférést biztosító **Adatelérési réteg**, amelynek alapvető funkcionalitása az adatok tárolása és kiszolgálása az alkalmazás (üzleti) logika rétegnek. A középen elhelyezkedő A**lkalmazás (üzleti) logika** felelős a szükséges ellenőrzések, számítások elvégzéséért, a munkafolyamatok felügyeletéért, és az adatok továbbításáért a Prezentációs rétegnek. Legvégül az oldalak – dinamikus – előállításáért, és a felhasználótól érkező adatok fogadásáért a **Prezentációs réteg** felelős.

Tervezési stratégiák

Az absztrakt modell kellően általános ahhoz, hogy megfelelő rálátást nyújtson a webalkalmazások viselkedésére, de túl általános ahhoz, hogy megmutassa az egyes kategóriákba tartozó webalkalmazások jellemzőit. Ráadásul egy fejlesztési folyamat kezdetén általában nem lehet teljes mértékben specifikálni, hogy az alkalmazás pontosan miként fog működni, mert struktúrája és funkcionalitása csak az idő előrehaladtával alakul ki. Továbbá egy webalkalmazás által tartalmazott és megjelenített információ szintén változhat. Így célszerű a fejlesztési folyamatot

kisebb, jól menedzselhető részekre bontani, amelyek segítségével a fejlesztők könnyebben átláthatják és kezelhetik a projektet. A legnagyobb kihívás és kockázat viszont akkor jelentkezik, amikor gyorsan kell a webalkalmazásokat elkészíteni. Ennek az eredménye általában a tervezésre, fejlesztésre, tesztelésre fordított idő redukálásában áll.

A megfelelő és hatékony fejlesztési technológiák kiválasztása ezért sokat segíthet a fejlesztőknek a web alapú rendszerek komplexitásának megítélésében, a kockázat minimalizálásában, valamint a határidőre történő szállításban. Ezenfelül a skálázhatóság szintén kulcsfontosságú tényező, mely eléréshez egy web alapú Információs Rendszernél szükséges, hogy olyan komponens-alapú architektúrával rendelkezzen, amely egy könnyen felállítható és méretezhető rendszert eredményez. Ezen céljaink eléréséhez egy útmutatót találhatunk a [3] cikkben – tíz lépésben összefoglalva a sikeres fejlesztés kulcsmozzanatait.

Az alkalmazás megtervezésének folyamatában általában a tervezési minták nyújtanak segítséget, míg az alkalmazás teljes életciklusára a módszertanok próbálnak megfelelő modelleket nyújtani. Természetesen ránk van bízva, hogy:

- saját modelleket készítünk,
- tervezési mintákat használunk,
- esetleg az elérhető módszertanok egyikét alkalmazzuk.

Tervezési minták

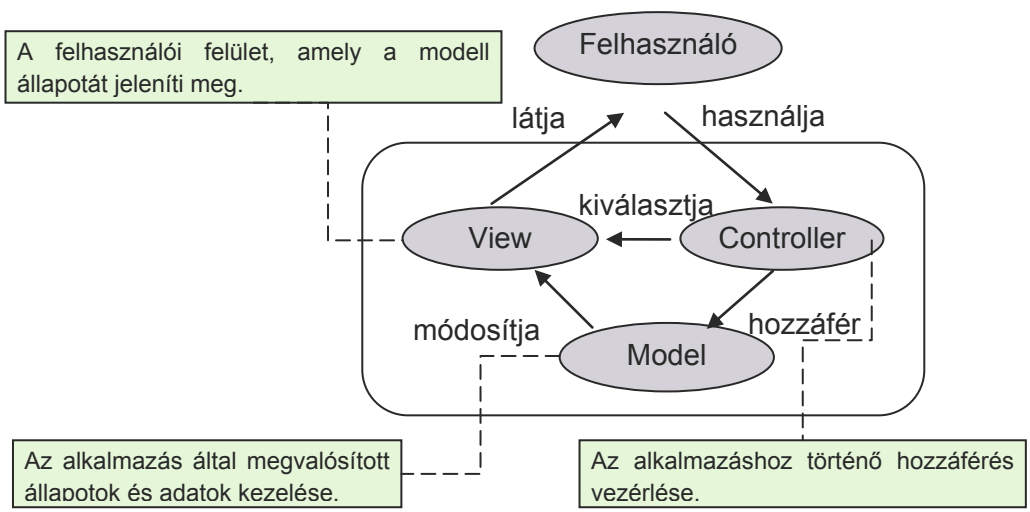
A tervezési minták alapötlete, hogy próbáljuk meg definiálni, majd kiemelni a magasabb szintű összefüggéseket, a komponensek közötti kapcsolatokat, majd ezeket alkalmazásról alkalmazásra újra és újra felhasználni. Egészen pontosan a következő definíciót értjük tervezési minta alatt: „*egymással együttműködő objektumok és osztályok leírásai, amelyek testreszabott formában valamilyen általános tervezési problémát oldanak meg egy bizonyos összefüggésben*” [4]. Ezenfelül segítenek tisztázni, hogy miként képzeljük el a fejlesztés alatt álló alkalmazás megvalósítását, illetve segítenek meghatározni az egymástól jól elkülönülő modulokat és objektumokat, valamint azok kapcsolatát.

Model-View-Controller (MVC)

A web alapú alkalmazások fejlesztésénél használt legelterjedtebb tervezési minta a Model-View-Controller (Model-Nézet-Vezérlő). Az MVC – mint módszertani alap – gyökerei a Smalltalkig nyúlnak vissza, ahol eredetileg a hagyományos „input - feldolgozás – output" folyamat grafikus felület segítségével történő alkalmazására használták. Talán ezért is alkalmas ez a koncepció a több rétegű webalkalmazások terén is.

Hatékonyan növeli az alkalmazás felhasználhatóságát, segít megérteni, illetve tisztázni a program pontos működését. Az MVC tervezési minta nagyon egyszerű, éppen ezért rendkívül hatékony, és az eredményeként kapott kód újrafelhasználható.

- **Model**: A Model reprezentálja az egész alkalmazói rendszer működését, az adatokat és azon üzleti szabályokat, amelyek ezen adatokhoz történő hozzáférést és módosítást felügyelik.
- **View**: A View jeleníti meg a tartalmat.
- **Controller**: A Controller fordítja le a View-n keresztüli műveleteket a Model által végrehajtandó utasításokra.



3. ábra: Az MVC architektúra

Egy webalkalmazás esetén a felhasználói interakciók (kliens – szerver kommunikáció) hiperlinkek segítségével történnek. Ezen hiperlinkek aktiválása együtt járhat adatok továbbításával a klienstől, mint például űrlapok (post) vagy URL kódolt (get) kérés. A szerver-oldalon a Controller fogadja a kéréseket, továbbítja a Model-nek, amely végrehajtja a kért feladatok – magában foglalva az üzleti szabályok használatát, illetve a Model állapotváltoztatását. A felhasználói utasításoknak, illetve a Model eredményének megfelelően célszerű a Controller-re bízni a megfelelő View kiválasztását.

Mindezek alapján az MVC használatával a következőket érhetjük el:

- *Modularitás*: lehetővé teszi bármely komponens kicserélését a felhasználó vagy a program igényei szerint. A program egy moduljának cseréje nem kötődik a többi modulhoz.
- *Újrafelhasználhatóság*: a korábban készült kódok újrafelhasználását támogatja, ha megfelelően körültekintően terveztünk.
- *Egyszerű bővíthetőség*: a Controller és a View a modellel együtt bővülhet.
- *Központosított vezérlő*: a Controller segítségével a menedzselhetőség még egyszerűbbé válik.

Vitathatatlan, hogy az MVC egy hasznos utat mutat webalkalmazások tervezéséhez, de nem ez az egyetlen út. Az MVC tervezési minta jelentősége abban rejlik, hogy segít elérni a különböző célokat, illetve az egyes funkcionális szintek elkülönítését. Így például külön modul gondoskodik az adatok kezeléséről, egy másik modul felelős az alkalmazás vezérléséért, és egy harmadik modul a megjelenítésért. Ezt az utat követve könnyen emelhetünk ki, és cserélhetünk ki különböző elemeket, bízva abban, hogy a többi rész érintetlenül maradhat.

Természetesen a webalkalmazások esetében kizárólagosan csak az MVC tervezési mintára támaszkodva könnyedén megfeledkezhetünk az alkalmazás számos más, fontos karakterisztikájáról. Ennek eredménye várhatóan egy igencsak „törékeny" megoldás, amelyet a későbbiekben nehéz karbantartani és továbbfejleszteni.

Kapcsolódó módszertanok

A tervezési minták hasznos segítséget nyújtanak a webalkalmazások elkészítéséhez, de alkalmazhatóságuk előfeltétele, hogy már rendelkezzünk kész tervekkel, amelyeket a minta segítségével hatékonyabban tudunk implementálni. A **tervek elkészítéséhez** viszont nem nyújtanak támogatást, ehhez egy, a számunkra megfelelő módszertan alkalmazása szükséges.

A ’90-es évektől kezdődően számos modellezési technika, módszertan került publikálásra, melyek a korai szakaszban még csak a hipermédiával való kapcsolaton alapultak, később már bekerültek a modern szoftverfejlesztési megközelítések, így egyre komplexebb webalkalmazások modellezésére is lehetőség nyílt.

Jelenleg több módszertan közül is választhatunk, eltérő előnyökkel és hátrányokkal. Vannak melyek a modellezés jelölésrendszerére összpontosítanak – mint például a Web Application Extension (WAE) [5], amely egy bővítés az Unified Modeling Language(UML)-hez –, és vannak amelyek a fejlesztési folyamatokra koncentrálnak, – mint például a Web Modeling Language (WebML) [6] és a Web Site Design Method (WSDM) [7]. A módszerek többnyire olyan kutatásokon alapulnak, amelyek szorosan kapcsolódtak az információ modellezéshez, illetve a hatékony navigációs struktúrák tervezéséhez, amelyhez jelentős támogatást adtak az adatbázis tervező és modellező technikák. A módszerek alapjaiban hasonló elképzelésekkel rendelkeznek egy tipikus Web alapú Információs Rendszer tervezéséről, az eltérések viszont a különböző szempontokból adódnak – hogyan és miként tekintenek egy webalkalmazásra. A különböző megközelítések **különböző modelleket** használnak, attól függően, hogy egy webalkalmazás mely részeire összpontosítanak.

A WebML például egy adat-orientált megközelítést használ olyan jelölésekkel, mint az ER modell és az UML osztály diagramok. Különbséget tesz szerkezeti modell, kompozíciós modell (oldalak, melyek az információs egységekből álló hipertextet felépítik), navigációs modell (az oldalak közötti kapcsolatok leírása) és prezentációs modell között. Ezenfelül tartalmaz elemeket, amelyekkel személyre szabhatóak az oldalak a különböző szerepköröknek megfelelően.

A WSDM ezzel ellentétben egy felhasználó központú megközelítést használ, a hozzáférhető adatok modellje a várható látogatók információ lekérdezésein alapul. A különböző felhasználói osztályokhoz különböző tevékenységek tartoznak, a

koncepcionális és a navigációs modellek ezek alapján készülnek. A modellezéshez saját jelölésrendszert használ.

A WAE pedig a programozó szemszögéből közelít, az UML nyelv bővíthetőségét kihasználva olyan komponenseket hoz létre, mint például weboldal és űrlap. Az egyes oldalakhoz külön osztálydiagram készíthető, amely tartalmazza az oldal segítségével végrehajtható funkciókat is. Majd ezen oldalakból felépíthető az alkalmazás teljes koncepcionális és logikai modellje.

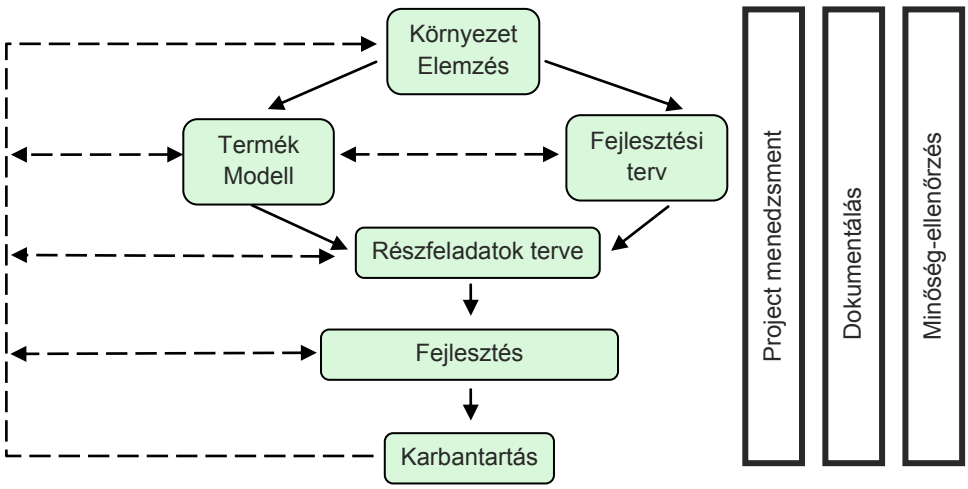
Ezeken felül természetesen további irányzatok is léteznek, amelyekhez a kutatók számos modellt és eszközt készítettek, hogy gyorsítsák a webalkalmazások különböző fejlesztési fázisaiban a tervezést és a fejlesztést [8].

Modellezési fázisok

Általában elmondható, hogy a módszertanok a tervezési folyamatot több feldolgozási folyamatra, illetve azok eredményeire – többnyire modellekre – bontják. Abban azonban alapvetően megegyeznek, hogy egy tipikus WIS tervezése a következő fázisokból áll:

- **Követelmény / környezet analízis**: a főbb célok összefoglalása, megértése, megvalósíthatósága, és a szükséges források specifikálása.
- **Koncepcionális tervezés**: a strukturális modell elkészítése a követelmény-analízisen alapulva, rámutatva az egyes komponensek közötti viszonyra.
- **Navigációs terv**: az alkalmazás navigációs modelljének elkészítése.
- **Prezentációs terv**: a navigációs, illetve strukturális elemek megjelenésének, valamint a felhasználói eseményekre való viselkedésüknek meghatározása.
- **Implementáció**

Láthatjuk, hogy egy Web alapú Információs Rendszer fejlesztése során számos különböző kérdést kell megértenünk, és megoldanunk. A tervezési folyamat egyes fázisai közötti kapcsolatot és összefüggést a következő ábrán láthatjuk.



4. ábra: Web alapú rendszerek egy lehetséges tervezési folyamata

A fejlesztés alatt az együttműködő csoportok közötti kommunikáció hatékony támogatását teszi lehetővé, ha szabványos jelölésrendszereket használunk. Erre a célra a web alapú rendszerek fejlesztésénél az UML megfelelő kiindulási alapot nyújt, elsősorban kihasználva a bővíthetőségét. Az egyes lépésekhez felhasználhatóak az UML különböző diagramjai, bár a modellezendő terület eltérése miatt jelentős kiegészítésekre van szükség, amelyek alapból nem találhatóak meg az UML-ben. Ilyen bővítéseket találhatunk a [5], [9], [10] cikkekben.

Jellemző problémák és megoldások

Az elérhető tervezési minták és módszertanok számos lehetőséget biztosítanak komplex webalkalmazások tervezéséhez, de a többségre jellemző, hogy mindegyik egy speciális esetet, vagy speciális megközelítést használ ezen rendszerek modellezéséhez. Így a minden igényt kielégítő, teljes webalkalmazások generálásához és működtetéséhez egyik sem nyújt megfelelő támogatást. Számos szempontot, mint például a testreszabhatóság vagy a tranzakció kezelés, nem mindig vesznek számításba.

A másik problematikus pont az **implementáció** megvalósítása, többségük csak a koncepciós és logikai szintre összpontosít, érintetlenül hagyva a fejlesztési folyamat eme utolsó fázisát. A webalkalmazások és a Web alapú Információs Rendszerek pedig mind-mind dinamikus weboldalakon alapulnak, amelyeket általában szkriptek állítanak elő (Php, Perl, JSP, esetleg ASP nyelven megírva). A szkriptek viszont a fejlesztési folyamatban egy újabb problémát eredményeznek, mert az üzleti (alkalmazás) logika és a megjelenítésért felelős kódrészleteket általában egy programmodulba „mossák" össze, amint azt a következő kódrészlet is mutatja.

```
<HTML>
<BODY>
Welcome
<%        Session = Application.RestoreSession()
          Response.Write "Session("UserName")"

          if Application("LastUserLogon") = "" then
                    Response.Write "<P>You are the first user to log on."
          else
                    Response.Write "<P>The last user was user <B>"
                    Response.Write Application("LastUserLogon")
                    Response.Write "</b>"
          end if
%>
</P>
<P>There have been <B><%=Session("Visitors")%></B> total visits to this
site.
</BODY>
</HTML>
```

5. ábra: A megjelenítés és a logika keveredése

Az eltérő funkcionalitásokért felelős kódok egy állományba történő összedolgozása az alkalmazás működése szempontjából önmagában még nem jelentene kezelhetetlen problémát. Viszont egy webes rendszer fejlesztésén általában több különálló csoport dolgozik, a rendszernek egy-egy részletéért felelve. Miután így ugyanazt az állományt többen is szerkesztik, ráadásul az alkalmazás különböző funkcionalitása kapcsán, ez könnyen futtathatatlan, szerencsétlenebb esetben működő, de hibás szkripthez vezethet. A problémát a következő példa is jól demonstrálja: a látványtervező módosítani szeretné – mélyebb szerver-oldali programozási ismeretek nélkül – a weboldal kinézetét a megjelenítendő információk sorrendjének (a látogatások számának és az üdvözlés) felcserélésével. A látványtervező szemszögéből nézve semmilyen probléma sem jelentkezik, mert a célját elérte, viszont a csere következtében az alkalmazás logikája hibásan fog működni, mivel az adatok visszatöltéséért felelős utasítást már megelőzi egy olyan utasítás, ami a betöltött adatokra hivatkozik.

A különböző funkcionalitású kódok keveredése további nehézségeket vet fel, ha egy webalkalmazás különböző felületeket nyújt az Internet és az intranet felé. Ebben az esetben az üzleti logikát szükségszerűen két helyen is implementálni kell, ami természetesen nehezen karbantartható programot eredményez. Ha változtatni kell logikai szinten, akkor a módosításokat a redundancia miatt több helyen is el kell végezni, ami plusz figyelmet követel.

Hasonlóan problematikus pont lehet, ha az említett módszertanok köré épített webalkalmazás generátorokat használjuk, mert a generált oldalak – általában ASP vagy JSP fájlok – ha el is különítik az üzleti logikát a megjelenítéstől, de a tartalmat és a megjelenítést már nem. Továbbá bármely kézzel végzett – nem modellezhető bővítés – a modell legapróbb változása során az oldalak automatikus (újra)generálásával azonnal elvész.

A fenti problémák jól tükrözik, hogy miért fontos **különálló modulokba szervezni** az üzleti logika, a tartalom, illetve a megjelenítésért felelős részeket, egyértelműen felosztva a felelősséget a grafikus és a fejlesztő között. A szétválasztás természetéből adódó további előny, hogy lehetővé teszi mind a logika, mind a megjelenítendő elemek későbbiekben történő újrafelhasználhatóságát. (Az MVC tervezési minta pontosan ezért választja külön a megjelenítésért felelős Nézet komponenst az alkalmazás funkcionális részétől.)

Hasonló, de nem teljesen analóg a helyzet a webalkalmazások réteges architektúrájában szereplő Üzleti (Alkalmazás) logikát tartalmazó réteg és az Adatelérést biztosító réteg között is. A fejlesztőknek ezen szemszög alapján is egy ésszerű felosztásra kell törekedniük az Adatelérési és az Üzleti logika rétege között. Pontosabban a cél az lenne, hogy az Adatelérési réteg ne, vagy csak minimálisan tartalmazzon validálást, üzleti logikát, mert funkcionalitásuk alapján ezen feladatok a másik réteghez tartoznak. Viszont kiiktatni mindennemű ellenőrzést az adatokat kezelő rétegből nem feltétlenül a legjobb megközelítés. Az adatbázissémákban megtalálható külső kulcsokra, illetve „not null" megszorításokra gondolhatunk úgy is, hogy elég ha csak az üzleti logika szintjén ismerjük, bár a többség valószínűleg azzal a megközelítéssel fog egyetérteni, hogy ezen egyszerű megszorításokat tároljuk az adatbázisban, és módosítsuk ha az üzleti szabályok változnak.

Ezt a megközelítést folytatva célszerű a fejlesztés során meghatározni azt a finom elválasztó vonalat az üzleti rétegben, amely kettéosztja **alkalmazás logikára**, illetve **tárolási logikára**. Az alkalmazás logika a munkafolyamatokért lesz felelős, az adatok tárolásáért pedig a tárolási logika. Adatbázis szinten implementálásra kerül a tárolási logika, így az üzleti modulokban már csak a tiszta alkalmazás logikára kell koncentrálnunk. Természetesen az elválasztó vonal kijelölése nem egy magától értetődő feladat, mert elég széles határok között mozoghat. A korábban már említett külső kulcsok és „not null" megszorítások kezelésén felül az adatbáziskezelő rendszerek számos további lehetőséget nyújtanak, ami a határok kitolódását eredményezi. A fejlesztés során dönteni kell, hogy az adatbáziskezelő rendszerek szolgáltatásaiból mennyit kívánunk igénybe venni, hogy a határt pontosan meg tudjuk határozni. A következő szempontok segíthetnek a döntésben:

- **Adatorientált alkalmazások** esetében célszerű kihasználni az adatbáziskezelők nyújtotta lehetőségeket, mert így egy helyen fognak szerepelni az adatok közötti összefüggéseket felügyelő kódok. Gondoljunk például az adattáblák oszlopainál elhelyezhető *ellenőrző feltételekre*, illetve a bonyolultabb kapcsolatokat is kezelni képes *tárolt eljárásokra*, melyek a logikai modellnek megfelelően korlátozzák a felvihető adatokat, illetve további (pl. számítási) feladatokat láthatnak el. Az adatok megjelenítésében hasznos segítség lehet, hogy a különböző igényeknek megfelelően eltérő nézetek hozhatóak létre (*nézettáblák* kialakításával, amelyek egyes adatbáziskezelők esetében az adatok módosítására is használhatóak).

- Portál és **egyéb alkalmazások** esetében az adatbázis többnyire ténylegesen csak az adatok központi tárolásáért felelős, nem léteznek bonyolult összefüggések az egyes táblák között, így az ellenőrző rész felkerülhet az üzleti logika rétegébe.

Természetesen minden egyes webalkalmazás más és más, általános érvényű szabályt létrehozni nem lehet, de mindig célszerű a tárolt eljárások használata, mert mindig elősegítik:

- a rendszer modularitását,
- az adatbázis rugalmasságát,
- a karbantartást.

A következő példa rámutat arra, hogy néhány sor kód – amely az adatok konzisztenciája miatt fontos, de idegen az alkalmazás vezérlő logikájától – lehetővé teszi az alkalmazás logika számára, hogy a továbbhaladásról az adatbázis válasza alapján döntsön.

```
create function check_membership() returns opaque as '
          m_id INTEGER;
          begin
          -- select disabled member ID
          select into m_id member_id from members where
                                                    memeber_id=NEW.m_id;
          if (m_id IS NULL) then
                    raise exception''Could not find a disabled member!'';
          else if (NEW.judge_id IS NULL OR m_id=NEW.judge_id) then
                    raise exception ''Member could not be there! / or no
                                        judge'';
               end if;
          end if;
          return new;
          end;'
Language 'plpgsql';
```

6. ábra: Tárolt eljárások alkalmazása

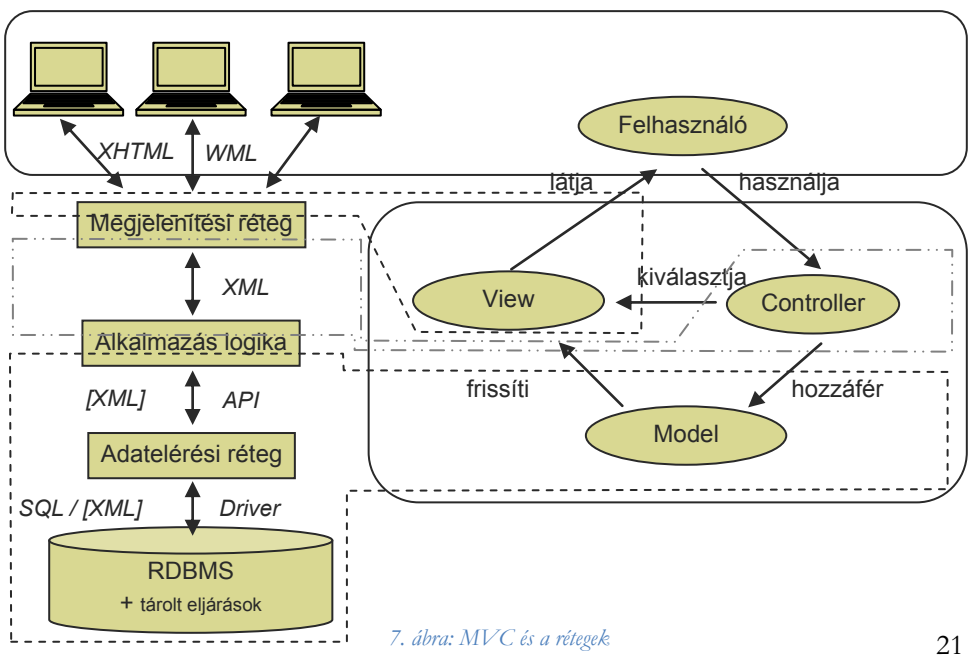
A módszertanok többsége sajnos nem használja ki se a nézettáblák nyújtotta lehetőséget (vagy ha igen, akkor csak adatkinyerésre), se a tárolt eljárásokat.

A webalkalmazások réteges szerkezete és az MVC tervezési minta

Az MVC egy jól ismert és széles körben használt tervezési minta, de a kérdés az, hogy miként tudjuk ezt társítani a webalkalmazások réteges szerkezetével?

Első ránézésre a válasz elég egyszerűnek tűnhet, mert három modulunk és három rétegünk van, a View komponens megfelelhet a Prezentációs rétegnek, a Model komponens az Adatelérési rétegnek, és végül a Controller pedig az Üzleti logika rétegének.

Viszont ha jobban megnézzük a Model és a Controller funkcionalitását, úgy találhatjuk, hogy a Model tartalmazza az üzleti szabályokat, a Controller pedig a felhasználói műveletek kezeléséért felelős. Ezek alapján érezhető, hogy a Controller csak egy része az Üzleti Logika rétegének, és a Model alkotja a másikat. A Prezentációs réteg esetén sincs egyszerűbb dolgunk, mert az általa megjelenítendő információ kiszolgálása a View komponens feladata, viszont a beérkező adatok kezelését a Controller látja el. Egy webalkalmazás fejlesztésénél ezért figyelembe kell vennünk e rétegek **heterogén összetételének** lehetőségét, és a fejlesztés során ezek ismeretében kell eljárnunk.



7. ábra: MVC és a rétegek

Az XML Technológiák szerepe

Az XML technológiák széles körben használatosak a webes világban a tartalom és az eszközfüggő-megjelenítés egyszerűbb és könnyebb újrafelhasználhatósága miatt. A tartalom reprezentálható, mint XML dokumentum, és a kiválasztott tartalom megjelenítése XSLT stíluslapokkal és transzformátorokkal megoldható. Az univerzális kliens hozzáférést a különböző XML formátumok – mint például az asztali böngészők számára XHTML, a WAP-os telefonok számára WML, … – előállításával lehet elérni.

További előnyökre tehetünk szert, ha a megjelenítendő információkat hordozó entitásokat a koncepcionális szinten adatbázis objektumokként kezeljük, és a különböző környezetfüggő nézeteknek megfelelően dinamikusan állítjuk elő ezen XML dokumentumokat. Ez a megközelítés megkönnyíti a szükséges változtatások egyszerűbb kivitelezését, az inkonzisztencia elkerülését, illetve támogatja a modularitásnak és újrafelhasználhatóságnak a további formáit. Ehhez természetesen szükséges, hogy definiáljuk saját belső XML formátumainkat az egyes rétegek közötti információátadáshoz. A definíciók továbbá lehetővé tehetik a gép-gép közötti kommunikációit is webszolgáltatások létrehozásával.

Miután elkészültek ezen XML formátumok – például XML Schema használatával – szükséges, hogy megteremtsük a kapcsolatot az adatbázis táblái és az XML fájlok között. A kérdés az, hogy miként konvertáljuk a relációs adatokat XML-re, illetve XML-t relációs adattá?

A strukturált XML adatok és a relációs adatbáziskezelők között számos különbség van, ahogyan az 5. ábrán is látható.

Ha relációs adatbáziskezelőt használunk az adatok tárolására szükséges, hogy:

- Normalizáljuk az XML struktúrákat.
- További kódrészletek segítségével osszuk fel az XML dokumentumokat a táblák között.
- A lekérdezéseket transzformálni kell, valamint az eredményt több, alsóbb szinten található adatbázis struktúrából kell összerakni.

XML	RDBMS
■ Az adatok egyszerű hierarchikus szerkezetűek ■ A csomópontok elemeket és/vagy attribútumokat tartalmaznak ■ Az elemek egymásba ágyazhatóak ■ Az elemek sorrendje adott ■ Opcionális séma ■ Lekérdezés XML szabványokkal	■ Az adatok több táblában vannak elhelyezve ■ A cellákban egyszerű értékek szerepelnek ■ A cellákban csak atomi értékek ■ Sor/Oszlop sorrend nem definiált ■ Séma szükséges ■ Lekérdezés SQL nyelvvel ■ Összekapcsolások szükségesek

8. ábra: Az XML és az RDBMS jellemzői

Miután elkészítettük a megfelelő XML Schema-kat, a sémák hatékonyan használhatóak a felhasználói adatok – XML dokumentumokká történő konvertálását követően – ellenőrzésre. Amennyiben a validálás sikeres, az XML dokumentumot továbbadhatjuk az Adatelérési rétegnek, ha pedig sikertelen, akkor a megfelelő hibaüzenetet tartalmazó oldal generálódik.

Egy másik érdekes lehetőség az XML Schema-k webes űrlapokká történő alakítása XSLT segítségével. Az adatok rögzítésére, illetve módosítására szolgáló űrlapok viszonylag egyértelműen származtathatóak a sémák XML természetéből adódóan az XForms segítségével. Bár a szükséges transzformáció az összetett típusok és az egymásba ágyazhatóság miatt nem magától értetődő művelet.

Összegezve, ezek a technológiák a hasznosságuk mellett segítenek a webes Információs Rendszerek különböző fejlesztési folyamatainak könnyebb átláthatóságában és megértésében, bár nem a legkönnyebb és legegyszerűbb utat kínálva a céljaink eléréséhez.

Elméleti háttér

A szoftverek használata napjainkban már teljesen elfogadott, munkánk során állandó jelleggel használjuk ezeket a termékeket, hogy a modern életünket könnyebbé tegyük. Gondoljunk például a szövegszerkesztőkre, a levelező programokra vagy akár a kikapcsolódást szolgáló alkalmazásokra. Mind arra készült, hogy egy adott feladatot, tevékenységet hatékonyabban, könnyebben tudjunk elvégezni. Mindezek mellett pedig gyakran hasznunk különféle keresőket is. A témaválasztásunk alapját képező Információs Rendszerek rendelkeznek ilyen jellegű funkcionalitással, és ha egy nyilvántartási rendszerként képzeljük el, akkor különböző szempontok alapján kell majd tárolni és rendszerezni az adatokat. A hagyományos megoldás a papír alapú kartotékolás volt, ahol a katalógus kialakítói az adott területnek megfelelően különböző stratégiák segítségével kidolgozták a megfelelő tárolási rendszert. Ugyan-úgy, mint egy papír alapú nyilvántartás kialakításánál, a számítógépes megoldásnál is figyelembe kell venni, hogy hova készül az a rendszer. Meg kell ismerni az adott területet, a követelményeket és be kell illeszteni abba a környezetbe. Természetesen ez fordítva is igaz, egy szoftvert nem lehet csak úgy kiragadni a működési köréből, abból a környezetből, ahová készült.

A szoftvereknek ezért praktikusnak és hasznosnak kell lenniük, elválaszthatatlanul illeszkedniük kell abban a környezetbe, ahol működtetjük őket. A szoftver tervezése éppen ezért egy bonyolult és komplex művelet, nem lehet úgy elképzelni, mint egy precíz tudományt, ahol jól definiált formulák és elméletek léteznek. Természetesen felfedezhetünk technikákat és irányelveket, amelyek hasznosak lehetnek a szoftver fejlesztési folyamatában, de az épületekhez hasonlóan, a szoftvereknél is nehéz két egyforma terméket találni, mert a tervezés és a fejlesztés lépései mindig magukon fogják hordozni a tervező személyes adottságait.

A szoftverek tervezésére – talán épp ezért is – számos különböző megközelítés létezik. A szoftveripar az elmúlt évtizedek alatt megismert és alkalmazott többféle módszert a termékek előállítására. Mindazonáltal mindegyik esetében megvoltak a maga előnyei és a velejáró hiányosságai.

Tartomány alapú tervezés

A szoftverfejlesztés általában a valós világban létező folyamatok automatizálására vagy valamilyen üzleti feladatra nyújtanak megoldást. Legyen szó bármelyikről is, már a fejlesztés elejétől kezdve figyelembe kell venni, hogy hova készül és milyen kapcsolata lesz az adott területtel.

A szoftver pedig programkódot fog tartalmazni. Ehhez viszont nem tudunk úgy nekikezdeni, hogy leülünk és már gépeljük is – eltekintve persze a triviális feladatoktól, amelyekre már lehetnek jól bevált módszereink. Ahhoz, hogy jó szoftvert készítsünk, tudnunk kell, hogy milyen célra is készül a rendszer. A példánknál maradva, nem tudunk hatékony információs rendszert készíteni anélkül, hogy ne ismernénk meg azt a területet ahová szánjuk.

Amikor elkezdjük egy szoftverrendszer fejlesztését, összpontosítanunk kell arra a szakterületre, amelyen működni fog. A szoftver célja az lesz, hogy az adott területet még hatékonyabbá tegye. Ennek érdekében arra a területre tökéletesen be kell tudnia illeszkedni, különben nem várt viselkedéssel, félreértésekkel és akár károkkal is szembesülhetünk.

Annak érdekében, hogy a szoftver a szakterület megfelelő leképezéseként tudjon működni, szükséges, hogy az alapvető elemeket, koncepciókat és az azok között található kapcsolatokat tökéletesen egyesítse. A szoftvernek a szakterületet kell modelleznie. Ez természetesen lehetetlen anélkül, hogy megfelelő ismereteket ne szereznénk magáról a területről.

De mit is takar a szakterület? A valós világban bonyolult összefüggések állnak fent. Ahhoz, hogy a számunkra szükséges információkat összegyűjtsük a szakterületnek egy absztrakciójára lesz szükségünk. A vizsgált területről a legtöbbet a szakterület szakértőitől (*domain expert*) tudhatunk meg. Azonban az információk birtokában nem tudunk elkezdeni azonnal szoftvereket összeállítani. Először csak ötleteink lesznek, amelyek szépen átalakulnak majd tervekké. Azonban a szakterületi modellt nem egy diagramként kell elképzelni, hanem olyan összefüggésekként, amelyeket a diagramoknak majd hordozniuk kell. Ez már közelíteni fogja a szakterületről képzett absztrakciónkat. A diagramok pedig segítenek az ötleteink kommunikálásában.

Természetesen a világunkban túl sok összefüggés van az egyes részek között, melyekből az absztrakció során bizonyosokat elhagyunk. Annak eldöntése, hogy mi fontos és mi felesleges, a tartomány modellezésénél mindig kihívás. Vannak azért egyszerűbb esetek is, mint például az általunk vizsgált nyilvántartási rendszerben a személyek lakcíme minden bizonnyal fontos lesz, míg a szemük színe már nem feltétlenül. Az információt rendszereznünk kell, a bizonyos koncepciók mentén összetartozó részek segítségével pedig feloszthatjuk a szakterületet. Ezen darabokat pedig a későbbiekben a logikai modellünk felépítéséhez használhatjuk fel.

A modellek nélkül nehezen tudnánk átlátni a komplex összefüggéseket, a modell fogja képviselni a szakterületről alkotott képünket. Azonban ezen elképzeléseinket át kell adnunk a tervezőknek, a fejlesztőknek. Ehhez kommunikációra van szükségünk. A modellünket tudni kell kommunikálni. Ha rendelkezünk a modellel, akkor elkezdhetünk tervezni, majd fejleszteni. A szoftvertervezésnél azonban nem a kódolás a legfontosabb rész. Míg a kódolási hibákat viszonylag könnyen javíthatjuk, addig a szakterületi modellben vétett tervezési hibák javítása már sokkal bonyolultabb, és általában költségesebb művelet. Természetesen a végén elkészülő rendszer jól megtervezett kód nélkül nem sokat ér. Ennek a folyamatára többféle módszertan is létezik.

Ezek közül az egyik legismertebb a vízesés modell, amely számos lépésen keresztül vezet el a végső szoftvertermékhez. A szakterületi elemző felállítja a követelményeket, amelyeket a tervező átvesz és elkészíti ezen követelményekből a modellt. Ezután a fejlesztők a modell alapján elkészítik a szoftvert. Azonban ez az információ átadásának egy igencsak egyirányú megközelítése. A fő probléma, hogy az egyes lépésekből nem érkezik visszajelzés, mondjuk a tervezőtől a szakértő felé, vagy a kódolótól a tervezőhöz.

Ennek teljesen ellentmondanak az agilis programozási módszertanok, mint például az extrém programozás (*XP*). Ebben az esetben a fejlesztés kezdetekor nem készítik el az összes követelményt, hanem a hangsúlyt a változáskezelésre helyezik. Mindig elkészül egy kisebb szelet, amit a következő lépésben tovább finomítanak. Ennek azonban számos negatív hatása is lehet. Miután nem ismerjük az összes követelményt az elején, a tervezők nem feltétlenül hoznak megfelelő döntéseket. Ennek következményeként jelentősen megnő a refactoringnak a szerepe, ráadásul a szakterületi elemző állandó jelenlétére is szükség van.

A tartomány alapú fejlesztésnél ezért először mindig a szakterület feltérképezésével és megértésével kell kezdeni. Össze kell gyűjtetnünk a szükséges információkat és le kell szűrnünk belőle a lényeget. Ehhez igénybe vehetjük a szakterületet ismerő emberek segítségét, miután a tervezők nem ismerhetik a világban megtalálható összes területet.

Ez a lépés leginkább a beszélgetések időszaka. Míg a tervező osztályokat, objektumokat, metódusokat és változókat lát, addig a szakterületi elemzők a terület összefüggéseit ismerik. Annak érdekében, hogy a modelljeinket el tudjuk készíteni, szükséges, hogy egy közös nyelven alapuló kommunikáció alakuljon ki. Ennek a nyelvnek az alapját a szakterületnek kell szolgáltatnia. Ennek a nyelvnek a segítségével tudjuk majd összekapcsolni a szakterületet a tervekkel.

A beszélgetések során kis ábrákat készíthetünk, amely az egyes részeket ábrázolja az azonosított koncepciókkal és összefüggéseikkel. A fejlesztés során pedig az ábrákon látott elemneveket használhatjuk például az egyes osztályok neveként, hogy a kapcsolat látható maradjon a modell és kód között. Az ábrák rajzolására használhatunk már ismert modellező eszközöket is, de egy kívülálló számára egy nagy UML ábra már nem sok jelentéssel bír. Amíg öt-hat doboz és pár vonal van rajta, addig bárki átlátja a felrajzolt összefüggéseket, de már egy közepes méretű projekt esetén is túl összetett diagramok készülnek, amelyeket néha még a hozzáértőknek is időbe telik átlátni. Ezért ennél a lépésnél inkább a kicsi ábrákra és szöveges magyarázatokra kell törekedni, hogy kialakíthassuk a megfelelő koncepciókat.

Az eddigiek alapján áthatjuk annak a jelentőségét, hogy mennyire fontos egy olyan modellnek a kialakítása, amelynek a gyökerei szorosan kötődnek a vizsgált szakterülethez, annak részleteit és összefüggéseit nagy pontossággal jeleníti meg. A cél egy jó modellnek az elkészítése, amely hasznos segítség lehet a kód előállításához. A modell elkészítés hosszú időbe is telhet, de nem ér sokat, ha nem tudjuk megfelelően átalakítani a szoftver nyelvére. További problémát jelenthet, ha fejlesztők a modellben található összefüggéseket megpróbálják átalakítani, kibővíteni és kifejezni saját diagramjaikkal. Eltávolítva egymástól az eredeti tervet és azt, ami kódolásra kerül. Ez számos kérdést fog felvetni a későbbiekben a bővíthetőség és a karbantartás kapcsán.

A szakterület tervezésének építőkövei

Amikor szoftverrendszereket tervezünk, a rendszer bizonyos része egyáltalán nem kötődik a szakterülethez, de szükséges az infrastruktúra használatához vagy épp magának a szoftvernek a működéséhez. Ebből az is következik, hogy a szakterületért felelős rész akár csak egy kisebb részét teszi majd ki a rendszernek, hiszen egy alkalmazás tartalmaz kódot az adatbázis eléréshez, fájlok vagy hálózati erőforrások igénybevételéhez, felhasználói felülethez és még számos ehhez hasonló részhez.

Egy objektumorientált programban az üzleti objektumok gyakran tartalmaznak olyan kódot, amely éppen az előbb említett funkcionalitásokhoz szükségesek. Gyakori, hogy a felhasználói felület megjelenítéséért felelős kód beágyazott üzleti logikát tartalmaz, amint ez az előző fejezetben a problémák között említettem. Ennek oka gyakran az igények gyors kielégítésében keresendő, mert sokkal gyorsabb lehet így elkészíteni a rendszert, mint egy szisztematikus tervezésre időt áldozni.

Mindezeken felül a szakterületi kód összeolvasztása a többi funkcionalitással nagyon hamar megnehezíti a rendszer átláthatóságát. A felhasználói felületen végzett módosítások megváltoztathatják az üzleti logikát, amely tovább gyűrűzhet az adatbázis hozzáférési részig vagy más programrészhez. Ezentúl a tesztelése is meglehetősen nehézzé válik egy így felépített programnak. Ezek alapján fontos, hogy a programok szerkezetét rétegekre bontsuk. Ezzel kapcsolatban a bevezető szakasz MVC mintájánál már vizsgáltuk ennek a szükségességét.

A program rétegre bontása során minden egyes réteget úgy kell megtervezni, hogy az összefüggő és csak az alatta elhelyezkedő rétegektől függjön. A szerkezeti mintákat követve könnyen megvalósíthatjuk a rétegek szabványos és egymástól jól definiált felületektől függő felépítését. Ennek alapján a szakterület modellel kapcsolatos ismereteinket egy rétegbe tudjuk összefogni, elválasztva a felhasználói felületért, az alkalmazás vagy éppen az infrastruktúra használatáért felelős kódtól. Ez lehetővé teszi, hogy a modellünk a későbbiekben is fejlődhessen, tisztán, összefogottan tartalmazva a szakterületről alkotott elképzeléseinket.

A tervezés során a következő általános, 4 koncepcionális réteget tartalmazó architektúrát szokták alkalmazni:

- ***Felhasználói felület:*** A felhasználó számára az információk megjelenítéséért és a felhasználói utasítások fogadásáért felelős.
- ***Alkalmazás réteg:*** Vékony réteg, amely az alkalmazás működését koordinálja. Nem tartalmaz üzleti logikát, nem tárolja az üzleti objektumok állapotát, de az egyes felhasználói feladatok folyamatát nyomon követi.
- ***Tartományi réteg:*** Ez a réteg tartalmazza a szakterületről alkotott modellünket. Az üzleti logika lelke. Ismeri az üzleti objektumok állapotát, azonban a perzisztencia fenntartását és az állapotok tárolását az alatta található réteghez delegálja.
- ***Infrastruktúra:*** A felette található rétegek alapvető szolgáltatásainak a támogatására szolgál. Ilyen például a már említett perzisztencia, de ide tartozik a felhasználói felület létrehozásához szükséges függvénykönyvtár is.

Az alkalmazás rétegre bontásával és a réteg közötti együttműködés szabályainak kidolgozásával elérhetjük, hogy a szoftverrendszerünk ne váljon átláthatatlanná és nehezen karbantarthatóvá. Ellenkező esetben egy apró változtatás végiggyűrűzhet a teljes alkalmazáson megjósolhatatlan következményekkel. A tartományi réteg bevezetésével azonban egy helyre összpontosíthatjuk a szakterületi ismereteinket, amely független az infrastruktúrát érintő tevékenységektől. Alkalmazás rétegre pedig a legtöbb esetben szükség van, hogy az üzleti logikához kapcsolódjon egy vezérlő, aki felügyeli és koordinálja az alkalmazás működését.

A következőkben pedig nézzük meg, hogy milyen fogalmak terjedtek el a tartományalapú tervezés során.

Entitás

Az objektumoknak létezik egy olyan osztálya, amelynél szükséges, hogy rendelkezzenek identitással, amely a végig azonos marad a rendszerben. Itt most nem az attribútumokra kell gondolni, hanem arra, hogy ezen objektumoknak meg kell őrizniük a folytonosságukat és identitásukat a rendszer működése során. Ezeket az objektumokat nevezik *entitásoknak*.

Az objektumorientált nyelvekben megjelenő objektumok és azok azonosítására használt referenciák ugyan egyediek a programon belül egy adott pillanatban, de egy szerializáció, egy memóriába történő ki- és belapozás során ez változhat. Ez nem az az identitás fogalom amire most nekünk szükségünk van.

Gondoljunk például egy nyilvántartási rendszerre, melyben emberekről kell adatokat tárolnunk (név, születési idő, …). Ezek külön-külön nem teszik lehetővé az objektumok egyediségét, nem tudunk megkülönböztetni két azonos névvel rendelkező személyt. Pedig számunkra az entitások során ez a legfontosabb. Még akkor is meg kell tudni különböztetni őket, ha azonos attribútumokkal rendelkeznek. Egy hibás azonosítás számos nem kívánatos eseményt válthat ki.

Éppen ezért az entitások implementálása együtt jár az azonosító létrehozásával is. Az előbb vizsgált személy esetén ez akár lehet az attribútumok egy együttese, vagy épp egy általunk létrehozott azonosító, vagy akár valamilyen személyes azonosítószám. Ha ezt az egyediséget nem tudjuk biztosítani, akkor az egész rendszer inkonzisztenssé válhat. Ennek érdekében létrehozhatunk külön modulokat, amelyek az egyedi azonosítók előállításáért felelősek.

Amikor egy objektumot az attribútumai helyett az azonosítója alapján különböztetünk meg, a modellünkben elsődlegesen az entitások közé sorolhatjuk. Az osztály definícióját próbáljuk meg a lehető legegyszerűbben megfogalmazni, és vegyük figyelembe a rendszerben végbemenő életciklusát. Hozzunk létre olyan műveleteket, amelyek lehetővé teszik az objektumok egyedi azonosítását. Ennek az egyedi azonosságnak a definiálásáért a modell a felelős és modellezés kezdeti fázisától számításba kell ezt vennünk. Természetesen az is fontos, hogy egy objektumnak szükséges-e egyediséggel rendelkeznie vagy sem.

Érték objektumok

A kérdés, hogy mi alapján határozhatjuk meg, hogy egy objektumnál szükséges az egyediség vagy sem. Az entitásokat nyomon tudjuk követni, az alkalmazáson belül csak egy példányban létezhetnek és emiatt nem lehet őket újrafelhasználni. Ennek az a velejárója, hogy az egyediség fenntartása költségekkel jár. Amit ha rosszul tervezünk, akkor akár a teljesítmény csökkenését is eredményezheti a nagyszámú entitás fenntartása miatt.

Ezek alapján ésszerű, ha megpróbáljuk különválasztani azon objektumokat, amelyek az alkalmazás működése során akár többször is újrafelhasználhatóak és nem kell önálló egyediséggel rendelkezniük. Ezen modellezési elemeket hívjuk érték objektumoknak (*value object*). Ebből az is következik, hogy meg is oszthatóak a hatékonyabb működés érdekében, aminek viszont így szoros velejárója, hogy megváltoztathatatlannak kell lenniük. Amikor egyszer létrejöttek egy adott értékkel, akkor a program további működése során az már nem változtatható meg. Ha új értékre van szükségünk, akkor egyszerűen kicseréljük egy másikra.

Nézzük újra a személy osztály példáját. Az előbbiekben már láttuk, hogy a személyeknek szükséges egyediséggel rendelkezniük. Azonban mennyire szoros jellemzője egy személynek a lakcíme, amely tartalmazza az utcát, a házszámot és az országot. Célszerű-e ezen információkat kiemelni? Egy személynek különböző leíró attribútumai nem feltétlenül hordozzák azt a jelentéstartalmat, amit szeretnénk kifejezni. Viszont kiemelve ezeket egy új osztályba, mondjuk Lakcím néven, akkor már az előbbi logikai összetartozást ki tudjuk fejezni.

Mennyire kell ezen információknak egyediséggel rendelkezniük? Használjuk-e a címeket önállóan azonosításra?
Ha ezen kérdésekre megpróbálunk válaszolni, akkor valószínűleg arra a döntésre jutunk, hogy nyugodtan létrehozhatjuk érték objektumként a lakcímet, és a személy objektum majd hivatkozik rá. Több személy lakhat ugyanazon a címen, megosztva egymás közt ugyanazon objektumot, amelyek így külön koncepcionális összefüggést írnak le.

Szolgáltatások

A tartomány elemzése során eddig érintettük az entitásokat és az érték objektumokat. Az objektumokat úgy képzeltük el, mint amelyek rendelkeznek attribútumokkal, saját maga által kezelt belső állapottal, amelyhez még kapcsolódik a viselkedése. Az elemzés során többnyire a főnevekből képezzük az osztályokat, míg az igékből a viselkedés leírásához szükséges tevékenységeket. Azonban találhatunk olyan igéket, amelyek nem feltétlenül kapcsolódnak egyetlenegy osztályhoz sem, viszont a tartomány egy fontos működését írják le.

Amikor ilyen esettel találkozunk, akkor azokat célszerű úgy elképzelni, mint a tartomány által nyújtott szolgáltatást. A *szolgáltatások* nem rendelkeznek belső állapottal, hanem a céljuk a tartomány funkcionalitásának a biztosítása. Ennek értelmében szintén a tartomány egy koncepciójának a leírására szolgálnak.

A szolgáltatások olyanok, mint az interfészek, amelyek tevékenységeket tesznek lehetővé. Amikor egy meghatározó tevékenység nem sorolható egy entitás vagy egy értékobjektum elvárható felelősségi körébe, akkor létrehozzuk a szolgáltatást a modell szinten.

A tervezés során fontos, hogy a szolgáltatásainkat jól körülhatároljuk, elválasztva a többi rétegtől. Azonban nem mindig egyszerű megkülönböztetni, hogy egy szolgáltatás a tartományhoz vagy az alkalmazás réteghez tartozik, hiszen mindkettő az entitásokra és az értékobjektumokra épülve végzi a működését. Vegyük az alábbi példát. Egy webes rendszer kimutatásokat tud készíteni különböző sablonok segítségével, amelyet a felhasználók a böngészőjükben tudnak megtekinteni.

A felhasználói felület rétege állítja össze a megfelelő oldalakat a bejelentkezéshez és a jelentésekhez is. Az alkalmazás egy vékony réteg, amely a felhasználói felület és a tartomány modell között található. A tartományi objektumok kötődnek a jelentések elkészítéséhez, mondjuk Report és Template néven. Amikor a felhasználó kiválaszt a listából egy sablont és megadja mondjuk a lekérdezési feltételeket, akkor ezen adatok az alkalmazás rétegen keresztül átadódnak a tartományi réteghez. Ez a réteg felelős a jelentés elkészítéséért és annak visszaadásáért. Miután a jelentések a sablonoktól függnek, szükséges egy szolgáltatás, amely az adott sablont szolgáltatja, mert ez nem a sablon feladata. A szolgáltatás az infrastruktúra segítségével megkeresheti a szükséges sablonfájlt akár adatbázisból, akár fájlból.

Modulok

Nagy és összetett alkalmazások esetén a modell mérete idővel egyre csak nagyobb és nagyobb lesz. Egy ponton pedig eléri azt a méretet, ami után már nehéz úgy gondolni rá, mint egy nagy egészre, mert az egyes részek közötti összefüggések átláthatatlanná válnak. Ennek kapcsán a modellünket modulokba szervezhetjük, amelyek az összetartozó koncepciók egybefogását teszik lehetővé.

Sokkal könnyebb egy nagy modellt elképzelni, ha látjuk milyen részekből áll és milyen az azok közötti kapcsolat. Miután megértettük a modulok közötti együttműködést, nekiláthatunk az egyes modulok részleteinek a megismeréséhez. A modulok kialakításához pedig az egyes osztályok egymáshoz viszonyított kapcsolatának erőssége alapján juthatunk el. Ennek vizsgálatához célszerű megnézni, hogy milyen adatokon dolgoznak. Ugyanazon adatokon dolgozó osztályok között szorosabb összefüggés mutatkozhat, mint az eltérő adatokkal dolgozók esetében.

A modulok kialakítása során jól definiált interfészeket kell létrehoznunk, amelyeken keresztül a többi modullal a kapcsolatot tarthatják. Az interfészek segítenek a kapcsolatok számának csökkentésében és ezzel együtt a komplexitás redukálásában is. Sokkal hamarabb átlátjuk a rendszer működését ha kevés számú kapcsolattal találkozunk.

A modulok központi részét teszik ki a tervezésnek, mert a határvonalak és szerepkörök eldöntésében segítenek. Ezek a későbbiekben már jellemzően nem változnak, ezzel szemben a modulok tartalma az idő előre haladásával jelentősen átalakulhat. A moduljaink így együtt tudnak fejlődni a projekttel.

További elemek

Az eddigi részekben áttekintettük a modell vezérelt tervezés alapvető elemeinek a listáját. A továbbiakban olyan mintákat említünk meg, amelyek a különböző modellezési feladatokhoz nyújtanak hatékony segítséget.

Ezek közül az első az aggregátorok (*aggregate*), melyek segítenek az objektumok határainak és egymástól való függőségüknek a kezelésében. A tárolók (*repository*) és a gyárak (*factory*) pedig az objektumok tárolásában és előállításában segítenek nekünk.

Az aggregátorok azon objektumok összefogására szolgálnak, amelyeket egy egésznek tekinthetünk. A korábbi példák esetében a személy és a lakcíme közötti összefüggés mutat erre példát. Az aggregátum rendelkezik egy központi elemmel. Ez az egyetlen elem, amely kívülről elérhető. A központi elem tartalmazza a megfelelő hivatkozásokat a többi elemre. Ezen kapcsolódó elemekkel pedig csak a központi elemen keresztül lehet műveleteket végezni, ezzel biztosítva a felügyelhető működést. Amennyiben adatbázisban kerülnek tárolásra, akkor szintén csak a központi elemnek szabad elérhetőnek lennie a lekérdezések során.

Miután ezen aggregált objektumok gyakran nagyok és komplexek, az előállításukhoz részletes ismeretekre van szükségünk. Ennek a kezelésére szolgálnak a gyárak. Ezáltal elérhető, hogy csak a gyárnak kelljen ismernie a szükséges összefüggéseket. Ha szeretnénk egy ilyen aggregált objektumot előállítani, akkor elegendő a gyárhoz fordulni.

Az objektumok előállítása után elkezdődik saját életciklusuk, amely során vándorolhatnak, külső tárolóra kerülhetnek és még hasonlók. Ahhoz, hogy egy objektumot használni tudjunk szükséges, hogy ismerjük a referenciáját. Azonban nagy rendszerek esetén ez túl sok referencia tárolását követelheti meg az egyes objektumoktól még akkor is, ha átmenetileg nincs szükségük egy adott objektumra. Erre kínálnak megoldást a tárolók. A tárolók célja, hogy magukba zárják az objektum referenciák eléréséhez szükséges ismereteket. A külvilág oldaláról nem kell semmilyen plusz ismeret, hogy hol van éppen a kérdéses objektum (pl. adatbázisban, fájlban, …). Elegendő a tárolóhoz fordulni. Ebből az is következik, hogy az újonnan létrejövő objektumok a tárolóban kerüljenek elhelyezésre.

Ezen eszközökkel már könnyedén meg tudjuk valósítani a szakterületi modell leírását, a továbbiakban pedig áttekintjük a modell vezérelt szoftverfejlesztés alapjait.

Modell vezérelt szoftverfejlesztés

Napjainkban a modell vezérelt szoftverfejlesztés (*Model driven Software Engineering - MDE*) [11] egyre nagyobb szerepet játszik és a szoftverrendszerek fejlesztésében az egyik legígéretesebb paradigmának tűnik. Az MDE használatával a szoftverfejlesztés elsődleges eszközei a modellek, amelyek végigkísérik az alkalmazásfejlesztés fázisait. Elsőbbséget élveznek a programkóddal szemben, a tervezőknek elsődlegesen a probléma területnek a modelljét kell elkészíteniük, nem pedig a platform specifikus leképzését. Ennek eredményeképp az utóbbi években egyre nagyobb lett a modell vezérelt fejlesztést támogató eszközök száma is.

A MDE legjelentősebb iránya az Object Management Group (*OMG*) által kidolgozott modell vezérelt architektúra (*Model Driven Architecture - MDA*) [12] [13]. Az alkalmazások platform-független szinten kerülnek modellezésre melyeket modell transzformáció(k) során alakítanak át platform specifikus modellekké.

Egy másik megközelítés szerint a fentebb említett tartományhoz szorosan kapcsolódó tartomány specifikus nyelvet (*Domain Specific Language - DSL*) [14] kell kialakítani. Ennek során az adott problémakörhöz legjobban illeszkedő nyelv kerül kialakításra, amely az általános célú nyelvekkel szemben várhatóan jobban vissza tudja adni a szakterületi fogalmakat. Egy DSL nyelv lehet belső vagy külső. A külső DSL-ek a fő programozási nyelvtől különböző nyelven íródnak és valamilyen fordító vagy interpreter segítségével kerülnek feldolgozásra. A Unix segédprogramjainak a nyelve, mint például az awk, vagy maguk a konfigurációs fájljai mind-mind külső DSL-re példák. Belső vagy beágyazott DSL nyelvek az alkalmazás fő programozási nyelvébe ágyazódnak be. A dinamikus programozási nyelvek rendelkeznek ilyen tulajdonságokkal (például Lisp, Smalltalk, Ruby).

Természetesen felfedezhetünk bizonyos analógiát a tartomány specifikus nyelvek és az MDA megközelítés között. Egy DSL megfelel egy meta-modellnek és a meta-meta- modellnek a megfelelője a magát a DSL-t leíró nyelvtan. A legtöbb esetben egy DSL kifejezhető, mint egy *Meta Object Facility* (*MOF*) [15] metamodell, vagy épp egy UML profil.

Éppen ezért nyílik lehetőség a DSL irányzatoknak az MDA megközelítéssel történő integrálására. Ez a webalkalmazások fejlesztését támogató keretrendszerek esetén kimondottan hasznos lehet a konfigurációs fájlok nyelvének kialakításához. A fejlesztés során mi is kialakítottunk egy tartomány specifikus nyelvet, amelyet a továbbiakban majd az MDA által használt platform független modellé alakítunk át. Mielőtt azonban erre rátérnénk, tekintsük át röviden az MDA alapjait.

Model Driven Architecture

A model vezérelt architektúra (*MDA*), mint láttuk a modell vezérelt szoftverfejlesztés egyik irányzata. Az alapját az elosztott rendszereknél használt Object Management Architecure (*OMA*) jelentette. [16]
Az MDA három alapelve a hordozhatóság, az együttműködés és az újrafelhasználás a koncepciók architekturális elválasztásán keresztül. Ezért a rendszert az architekturális szempontok figyelmen kívül hagyásával kell megtervezni. A terveket ezután a választott platformnak megfelelően, a sajátosságok figyelembe vételével át kell alakítani az alkalmazás használatát biztosító platformra. Ezt hívják MDA mintának.

Az MDA irányzat számos OMG szabványon alapul, ilyen például a már említett Meta Object Facility (MOF) és a Unified Modeling Language (*UML*) [17], amelyek a következő részben kerülnek ismertetésre. Annak ellenére, hogy az MDA a transzformációkon alapul, nem köti meg semmilyen specifikus transzformációs nyelvnek a használatát sem. Azonban már nagyon közel van a szabvánnyá váláshoz az OMG-nek a MOF 2.0 Query/View/Transformations (QVT) transzformációs nyelve [18].

Modell típusok

Az MDA a következő modelleket különbözteti meg:

- számítás független modell (*Computation Independent Model – CIM*)
- platform független modell (*Platform Independent Model – PIM*)
- platform specifikus modell (*Platform Specific Model – PSM*)
- platform modell (*Platform Model – PM*)

Az első három modell reprezentálja a rendszer különböző nézeteit a különböző szempontok és absztrakciós szintek szerint. Ezek a hagyományos szoftverfejlesztés elemzési, tervezési és implementációs szakaszoknak felelnek meg.

Számítás független modell (CIM)

Az MDA útmutató szerint a számítás független modell a rendszer környezetét és a vele szemben támasztott követelményeket tartalmazza. Ennek a modellnek szokott a szinonimája lenni az elemzési modell, szakterületi modell vagy üzleti modell. A CIM modell feladata megteremteni a kapcsolatot a szakterületi elemzők és rendszertervezők között.

Platform független modell (PIM)

A platform független modell lényegében az első konkrét modell, ami létrejön. Az MDA terminológia szerint egy olyan magas fokú absztrakciós szint, amely figyelmen kívül hagyja mind a tervezett platformot, mind pedig az implementációs részleteket. A PIM modellek leírhatják magát a rendszert, illetve a rendszer által megvalósítandó üzleti logikát is.

Platform specifikus modell (PSM)

A platform specifikus modell már egy adott platformra készül, figyelembe véve annak sajátosságait. A PIM modellből származtatható transzformációk segítségével. A céljától függően több-kevesebb részletet tartalmazhat. Amennyiben annyi részletet tartalmaz, hogy az már elegendő az implementáció automatikus generálásához, akkor már az implementáció platform specifikus modelljét ábrázoltuk. Másrészt viszont, ha ehhez nem elegendő információt tartalmaz a PSM, akkor manuális beavatkozásra van szükség az implementáció előállításához.

Platform modell (PM)

Az MDA útmutatóban a platform modell megfogalmazása egy picit bizonytalan. Egyrészt jelenti a technikai koncepciókat, megjelenítve a platform egyes építőelemeit és az azok által nyújtott szolgáltatásokat. Másrészt, a platform specifikus modellben használható elemeket írja, hasonlóan, mint egy metamodell, csak itt a platform specifikus modellekhez igazodva.

Az OMG meta-architektúrája

Az MDA nem egy különálló szabvány, hanem inkább a többi OMG szabványra épül. A következő rész bemutatja az MDA alapját jelentő metamodell hierarchiát, majd pedig a platform független modellezéshez használt UML nyelv rövid áttekintése következik.

Metamodell szintek

Amikor modellekkel dolgozunk, meg kell különböztetnünk a metamodellt, a modellt és a modell példányt. A metamodell határozza meg a modellek létrehozására használható nyelvet, és a modell a metamodell egy példánya. Miután a metamodell is egy modell, ezért ez a modell-példány kapcsolat a végtelenségig egymásra építhető. Ha a modellünk a hierarchia i.-ik szintjén van, akkor a neki megfelelő metamodell az i+1.-ik szinten.

Az OMG egy négyszintes metamodell hierarciát határozott meg a szabványaikhoz. A metamodell hierarchia gyökere a harmadik szinten elhelyezkedő meta-metamodell. Ezt nevezik Meta Object Facility-nek (*MOF*) és a metamodelleket leíró nyelvnek tekintjük. A MOF visszaható, ezért önmaga definiálására is alkalmas, így nincs szükség újabb szint bevezetésére. Az UML pedig egy példa a második szinten elhelyezkedő metamodellre, azaz azon nyelvekre, amely az első szinten használható modellek leírására használhatóak. A hierarchia legalján pedig az első szinten definiált modellelemeknek a példányai találhatóak – azaz egy Személy osztály esetén egy személy objektum.

A következőkben összegezhetjük az OMG metamodell hierarchiáját:

M3: meta-metamodell = metamodell definíciós nyelv = MOF

M2: metamodell = modell nyelv specifikáció (pl. UML)

M1: modell

M0: modell példány

Meta Object Facility

A Meta Object Facility (*MOF*) az MDA metaadat kezelési rendszerének az alapját jelenti. Arra az igényre adott válaszként alakult ki, amely az alkalmazások – leginkább fejlesztő eszközök – közötti adatcsere kapcsán alakult ki. A Metaadat az adat struktúrájának és jelentésének a specifikálására szolgál. A MOF megteremti a lehetőségét ezen metamodellek meghatározására (azaz a meta-metamodelleknek).

A MOF leképzések különösen fontosak a metadatok kicseréléséhez és manipulációjához, és ebből kifolyólag magának az MDA megközelítésnek a sikeréhez. Az eszközök közötti együttműködéshez, gondoljuk itt az UML eszközökre, pedig az XML Metadata Interchange (*XMI*) formátum leképzése elengedhetetlen fontosságú. Ezenfelül természetes számos más leképzés is létezik, például a Java Metadata Interface, vagy XML Document Type Definition.

A MOF specifikáció két részből áll, Essential MOF és Complete MOF. Az EMOF a legalapvetőbb metamodellezési eszközöket tartalmazza, leginkább az objektum-orientált szemléletmódhoz szükséges elemekkel. Ezzel szemben a CMOF metamodellek EMOF metamodellek leírására szolgálnak az EMOF bővíthetőségét kihasználva. Ez leginkább az UML nyelvnek a profilokkal történő kiterjesztésére hasonlít.

Unified Modeling Language (UML)

Az UML egy széles körben használt általános célú nyelv a szoftverrendszerek modellezésére. Az UML absztrakt szintaxisa egy MOF metamodellként érhető el, amelyhez társul egy konkrét grafikus szintaxis is. Az UML alkalmas a rendszerek mind statikus, mind dinamikus jellegének a modellezésére.

Az első meghatározó UML verzió a ’90-es évek végén jelent meg a „három amigó” (Grady Booch, Ivar Jacobson és James Rumbaugh) által kifejlesztve, főleg az akkori objektum-orientált szemléletek egységesítésére. Azt, amit ma UML-ként vagyis „Unifi ed Modeling Language”-ként ismerünk, eredetileg „Unified Method”-nak hívták. Ezt a nevet azonban félrevezetőnek találták, mivel egyfelől az UML jelölések egész rendszerét kínálja, másfelől pedig nem rögzít technikákat vagy szoftverfolyamatokat. Egy módszer egy szoftverfolyamat egyes lépéseinek szisztematikus kidolgozását támogatja. Egy módszer egy jelölésből és egy technikából

áll. A jelölések lehetnek szövegesek vagy grafikusak, és csupán a modell leírására szolgálnak. A technika ezzel szemben egy részletes eljárásmód, egy vázlatos útmutatás a jelölés alkalmazásához, vagyis a modell elkészítéséhez.

A második meghatározó verzió, az UML2 már az OMG kezdeményezésére jött létre. Ennek a végleges változata 2006-ban jelent meg. A legnagyobb változás az UML első és második változata között a nyelv bővíthetőségének a továbbfejlesztése és a modellezési lehetőségeknek a kiterjesztése új diagramok bevezetésével. A bővíthetősége réven ki tudunk fejleszteni egy metamodellt kimondottan az általunk vizsgált web alapú információs rendszereknek a modellezésére.

Az UML kiterjesztése

Az UML kiterjesztései lehetnek ún. „könnyűsúlyúak" (lightweight) és „nehézsúlyúk" (heavyweight). A nehézsúlyú kiterjesztések az UML metamodeljének a módosításán alapulnak, ami maga után vonja, hogy modellezési elemek szemantikájának a változását ezáltal elveszti a kompatibilitását az UML eszközökkel. A könnyűsúlyú kiterjesztéseket UML profiloknak nevezik és az UML kiterjesztési mechanizmusán alapszanak. A profilok általában sztereotípiákat, kulcsszavas értékeket és megszorításokat tartalmaznak, amelyek az UML metaosztályok kiterjesztését és megszorítását hivatottak ellátni annak érdekében, hogy egy adott problémakör modellezhető legyen.

A nehézsúlyú kiterjesztéseket „profilozhatónak" nevezik, ha a metamodell egy UML profilra leképezhető. Az UML modellező eszközök általában lehetővé szokták tenni a profilok alkalmazását, illetve a profil alapján a metamodell bővítését, de ez a ritkább eset. Az általánosabb, hogy a metamodellben végrehajtott kiegészítéseket egy profilra leképezik, és a modellező eszközökben alkalmazzák.

Mi ezt a szemléletet fogjuk követni annak érdekében, hogy az általunk modellezni kívánt Web alapú Információs Rendszereket a modell– és tartomány alapú modellezési technikák segítségével az MDA irányelveit követve a platform független modellekből platform specifikus modellekhez jussunk. Ehhez először szükségünk lesz a saját tartomány-specifikus jelölésrendszerünknek a kidolgozására.

A DSL létrehozása

A jelölésrendszer kidolgozása során megtartjuk az alapvető koncepciókat, amelyet a tartomány alapú tervezés során érintettünk. Amivel tovább bővítjük a jelölést, az a modellezendő terület sajátosságából ered.

A webalkalmazások esetén a modellben az entitások azon kívül, hogy egyediséggel kell, hogy rendelkezzenek, bizonyos esetben szükséges, hogy különböző szerepköröket is ellássanak. Vegyünk például egy oktatási intézményt, ahol oktatók és hallgatók is dolgoznak. A kiindulás állapot szerint valaki vagy oktató, vagy hallgató és ennek megfelelően rendelkezik a rá jellemző leíró attribútumokkal. Mind a két azonosított entitáshoz létrehozzuk a megfelelő kis dobozt az ábránkon és röviden jellemezzük is őket. A tervezés során feltűnik újra a lakcím problémája, de a korábbi ismereteink alapján már tudjuk, hogy ez egy értékobjektum lesz. Eddig rendben is vagyunk. Problémák majd akkor merülnek fel, ha egy hallgató végez és esetleg alkalmazásra kerül az általunk modellezett oktatási intézményben. Kérdés, hogy milyen formában jelenítsük meg?

Az első megoldás lehetne, hogy miután entitás, vegyünk fel majd neki egy új objektumot és úgy jó lesz. Viszont mit tegyünk, ha az adataiban változás áll be? A hallgató objektumunk és az oktató objektumunk azonos alapadatokkal rendelkezik. Ha majd hallgatóként keresem, mint végzett diák, akkor az akkor aktuális értékeket kapjuk, ha pedig oktatóként, akkor már a módosítottat. Érezhetően nem lesz ez így tökéletes. A legegyszerűbben úgy oldhatjuk meg a kérdést, hogy az entitásoknál megengedjük a szerepkörök használatát, ami a viselkedésükre vonatkozik.

Ehhez csupán az entitás definícióját kell bővítenünk. A korábbiakban azt mondtuk, hogy az ezen objektumoknak meg kell őrizniük a folytonosságukat és identitásukat a rendszer működése során. Most ezt kiegészítjük azzal, hogy lehetőséget adunk egy objektumnak eltérő szerepkörökben történő elérésére. Miután jelenleg még az MDA filozófia szerint számítás független modell (CIM) létrehozásánál járunk, semmilyen több információt nem kell rögzítenünk.

A tartomány alapú tervezés következő építőeleme az érték objektum volt. Ehhez szerencsére nem kell semmilyen plusz jelentést társítanunk.

A tartomány által leírt terület határainak a kijelöléséhez a szolgáltatás elemet fogjuk használni. Ez egy jól definiált felületet határoz meg az elérhető tevékenységekkel együtt. A DSL nyelvünkben a szolgáltatásokat egy általános programozási nyelv szintaktikájával fogjuk leírni. Megadhatjuk a várt visszatérési érték típusát, illetve a szükséges paramétereket.

A tartományi objektumok tárolását és elérését a tárolók (*repository*) segítségével fogjuk megvalósítani. A tárolók lesznek a felelősek az új objektumok létrehozásáért és a törlésükért is.

A rendszer alapvető komponenseinek a leírása pedig a modul értelmezését fogjuk felhasználni, hiszen ennek a feladata a valamilyen logika vagy szempont szerint összetartozó objektumokat egybefogni. Az interfészek megadásához a szolgáltatásoknál használt szintakszist fogjuk alkalmazni. Ezek után pedig álljon itt egy példa, amely bemutatja a tartomány specifikus nyelvünk alkalmazását.

```
Application OktatásiIntézmény {
    basePackage = hu.myIntémzény

    Module Intézmény {

        Service IntézményService {
          keressKurzusNév delegates to
                          KurzusRepository.findKurzusByName;
          mentIntézmény delegates to IntézményRepository.save;
          keressOktatóNév delegates to
                          SzemélyRepository.findOktatóByName;
          keressHallgatóNév delegates to
                          SzemélyRepository.findHallgatóByName;
        }
        Entity Intézmény{
          String name key
          reference Set<@Movie> movies
          Repository IntézményRepository {
            save;
            @Intézmény findIntézményByName(String name);
          }
        }
        Entity Kurzus {
          String kurzusNév not changeable
          Integer kredit
          Repository KurzusRepository {
            List<@Kurzus> findByName(Long intezmenyId, String nev);
          }
        }
    }
```

```
Module Személy {
 Service SzemélyService {
         keressSzemélyNév delegates to
                  SzemélyRepository.fingSzemélyByName;
         keressHallgatóNév delegates to
                  SzemélyRepository.findHallgatóByName;
         keressOktatóNév delegates to
                  SzemélyRepository.findOktatóByName;

  }

  Entity Személy {
    String szem.szám key length="20";
    Integer kor;
    String nev;

         EntityRole Oktató{
              String szakmai_érdeklődés;
         }

         EntityRole Doktorandusz{
              String neptun_kod;
         }

      Repository SzemélyRepository {
       findById;
       List<@Személy> fingSzemélyByName(String név);
       save;
       findByQuery;
       findByExample;
       delete;
  }
 }
}
```

9. ábra: DSL példa

Fejlesztési folyamat

A webalkalmazások készítése eltérő fejlesztési lépéseket igényel, mint amit a hagyományos szoftverfejlesztési megközelítések kínálnak a „hagyományos" értelemben vett szoftverek fejlesztéséhez. A Web Engineering egy új és fejlődő tudományág, ami épp ennek a fejlesztési folyamatnak az átfogására alakult ki.

Természetesen a szoftver rendszerek fejlesztésénél már kialakult és bevált módszerek nem feltétlenül alkalmatlanok webalkalmazások fejlesztésére, csak bizonyos szemléletmód váltásra van szükség, hogy alkalmazkodhassanak a világháló által nyújtott lehetőségekhez. Az alapok hasonlóak, ugyanúgy egy modellezésre, egy absztrakcióra van szükség, aminek a segítségével a terméket el kell tudni helyezni a környezetében. Ehhez hasznos segítséget nyújt a szoftverfejlesztésben már jól bevált fejlesztési folyamat a „Unified Process" [19]. Ez a fejlesztési folyamat *átöleli az alkalmazások teljes életciklusát*, melyet fázisokra bont, mint például *elemzés, kialakítás, elkészítés, átvitel és karbantartás*. Minden egyes fázisnál adott, hogy kik vesznek részt benne, milyen tevékenységeket végeznek, és milyen eredményeket állítanak elő. Az eredmények közé tartoznak a különböző modellek, a kódrészletek és a dokumentáció.

Az általunk ajánlott módszer segítséget nyújt a modellek lépésenkénti, szisztematikus elkészítéséhez, ami egy iteratív és növekményes tervezési folyamatot eredményez. A modellezés során a következő lépéseket végezzük el: követelményelemzés, koncepcionális-, navigációs- és prezentációs tervezés. A modellezés során az UML (Unified Modeling Language [17]) nyelvet használjuk, melyben a megszorítások megadhatóak mind természetes nyelven, mind pedig OCL (Object Constraint Language [20]) kifejezések segítségével.

A célunk nem egy teljesen új módszertan meghatározása az alapoktól kezdve, hanem a webalkalmazások fejlesztésénél már jól bevált módszertanok különböző szemléletmódjainak az összefogása és ezen eredményeknek új ötletekkel történő

kiegészítése, mint például a szerepkörök modellezése vagy az oldalak absztrakt megtervezése komponens alapokon.

Mindezek alapján a megközelítésünk a következő főbb irányelveket használja:

- szabványos jelölés használata (*UML*),
- szabványos kiterjesztések alkalmazása (*UML profilok*),
- az egyes modellek elkészítésének pontos leírása,
- a kritériumok megadása.

Az UML formális definíciókat használ a modellek leírásához, amely a metamodell szintjén bővíthető. Ennek alkalmazásával a rendszer strukturális szempontjait az alapeszközökkel modellezhetjük. Az ábrázoláshoz osztálydiagramokat használunk, amelyek biztosítani tudják a későbbiekben a platform specifikus modellek generálását a szerkezeti jellemzőkkel együtt. A modellezés során a navigáció kialakításához a szerkezeti modellt használjuk fel, mert az egyes elemek közötti asszociációk biztosítják a meghatározó kapcsolatokat adatorientált esetben.

A különböző lépésekhez tartozó szakterületi modellek létrehozásának a legáltalánosabb módja az UML szemantikájának a kibővítése. Ehhez használhatjuk UML profilokat, amelyek sztereotípiákat és kulcsszavas értékeket adnak a modellekhez. Ezen profilokra úgy is tekinthetünk, mint amelyek a metamodellben helyezkednek el, hogy különféle UML „dialektusokat" határozzanak meg, amelyeket a modellinkben felhasználhatunk. Ez a mechanizmus mind a platform független, mind a platform specifikus modellek elkészítéséhez hasznos segítséget nyújt.

A következőkben a fejlesztési folyamat egyes fázisaihoz tartozó lépéseket, az ott előálló modelleket és diagramokat tekintjük át.

A folyamat fázisai

A kezdeti lépés az *elemzési fázis*, ami egy rendszer iránti igény megfogalmazásával indul. A fázis végére az ötlettől eljutunk, a célrendszer általános elképzelésén át, a megvalósítandó, többnyire a rendszer üzleti szerepének a meghatározásáig. Kialakulnak az elsődleges elvárások, a várható architektúrák, illetve a várható költségek és a fejlesztés tervezett időigénye.

A *kialakítási fázisban* már meghatározzuk a rendszer konkrét architektúráját. A projektvezető kialakítja a megvalósítandó tevékenységek terveit, várható erőforrásigényeiket. Fontos, hogy kockázatelemzés is készüljön, ami kiterjed mind a várható csúszásokra, mind a módosítási igényekre. Ezen tervek nélkül nem léphetünk át a következő fázisba.

Az *elkészítési fázisban* a rendszer kifejlesztésére összpontosítunk, mindamellett, hogy kiegészítő követelmények beillesztése és kisebb módosítások még végrehajthatóak. A fázis végére pedig az összes használati eset megvalósításra került.

Az *átviteli fázis* öleli át a rendszer tesztelését (az elkészült termékeket szokták béta kiadásoknak nevezni). Ebben a fázisban a felhasználók egy kisebb csoportja megismerkedik a rendszerrel, elkezdődik a betanításuk, és a támogatási szolgáltatás kialakítása. Továbbá a fellépő hibák javítása is ebbe a fázisba tartozik.

A *karbantartási (evolúciós) fázis* akkor kezdődik, amikor a rendszer első verziója leszállításra került, és célja, hogy a terméket adaptálja a környezethez. Előfordulhat, hogy javításokat is kell végezni, de elsődlegesen a finomhangolás a cél – mint például egy képernyőn elhelyezkedő információk átrendezése.

Mindegyik fázis több mérföldkőből áll, amelyek bejárásával lehet átlépni a következő fázisba. A mérföldkövek tartalmazzák az életciklus céljait, szerkezeteit, a kezdeti működési feltételeket, a termék kiadását és megszüntetését.

Tervezési lépések

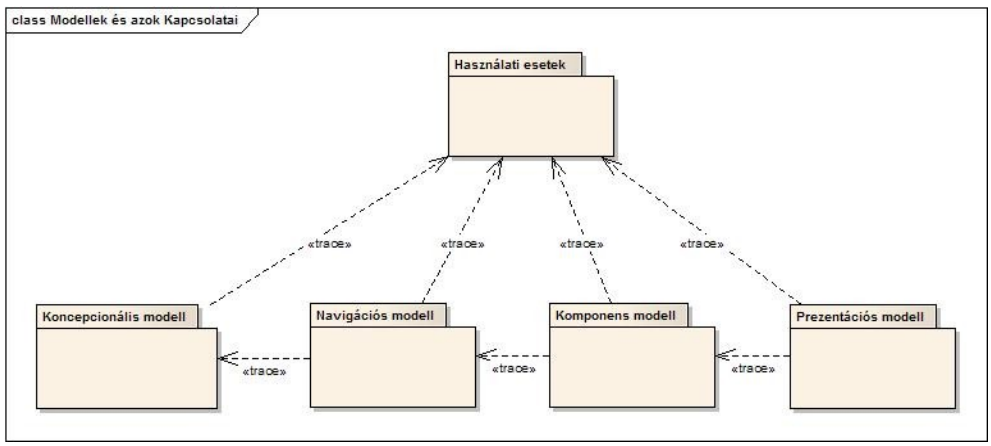
A tervezési folyamat több összetett munkafolyamatból áll, amely magában foglalja a *követelmény meghatározást*, az *elemzést*, a *tervezést*, az *implementálást* és a *tesztelést*. A mi esetünkben az elemzés és a tervezés két nagyon szorosan összekapcsolódó tevékenység, amelyeket talán nem is érdemes most kettébontani. A tervezés a mi esetünkben az elemzés egy további finomításának tekinthető, ahol az elemzés során kapott modelleket pontosítjuk. Éppen ezért az elemzési modellek a tervezési modellek absztrakciójának tekinthetőek, úgy használhatóak, mint egy kevésbé részletes tervezési modell.

A tesztelés egy alapvető fontosságú munkafolyamat, amely a minőségbiztosítási követelmények szem előtt tartásánál fontosak. A minőségbiztosítási folyamat magában foglalja a validálást, a verifikációt és magát a tesztelési folyamatot. A webalkalmazások esetében mind a tesztelés, mind pedig a minőség másképp értelmezhető, mint a hagyományos szoftverek esetében. A jól ismert metrikák (mint például a kódsorok száma – LoC, vagy az osztályok száma) nem feltétlenül szolgálnak mérvadó információkkal. Ezen a területen jelenleg számos kutatás folyik, annak reményében, hogy sikerül olyan mérhető adatokat találni, amelyek jól tudnak jellemezni egy modellvezérelt fejlesztés során előállt alkalmazást mind használhatóság, mind megbízhatóság tekintetében [21] [22].

A követelmény meghatározás és az elemzési/tervezési lépések során a következő modellek készülnek el (az irodalomban gyűjtő néven: *artifact* – magyarul talán termék – néven hivatkoznak ezen modellekre összességében):

- használati esetek (*use case models*)
- koncepcionális modell (*conceptual model*)
- navigációs modell (*navigation model*)
- prezentációs modell (*presentation model*)

Ezen modellek egy iteratív folyamat során egyre részletesebben fogják tartalmazni a szakterületi követelményeket. A következő ábrán a modellek egymásra épülését, egymásra történő hatását szemlélteti.



10. ábra: A létrehozandó modellek és azok kapcsolatai

A követelményelemzés célja a létrehozandó webalkalmazás funkcionális követelményeinek a meghatározása és azok használati esetekként történő szemléltetése.

A koncepcionális modell az elemzés és tervezési lépések során az alkalmazás szakterületi modelljét szemlélteti, amely már figyelembe veszi a használati esetek által rögzített követelményeket is. Ehhez a hagyományos objektumorientált fejlesztési technikát alkalmazzuk, ami az osztályok és azok kapcsolatainak a leírására szolgál. A koncepcionális modell egy hagyományos osztálydiagram formájában készül el.

A navigációs modell ezen koncepcionális modellen alapul, és célja, hogy meghatározza a webalkalmazás navigációs struktúráját. Ehhez bevezetünk új sztereotípiákat és navigációs elemeket, amelyeket a navigáció leírásához fogunk használni. Az eredmény itt is egy osztálydiagram lesz, amely már az alkalmazáson belüli lehetséges navigációs utakat tartalmazza.

A prezentációs modell egy absztrakt felhasználói felület elkészítését segíti, amelyeken keresztül majd a felhasználók elérhetik az alkalmazást. Ennél a lépésnél

bevezetjük az oldal darabkák fogalmát, amely egy összetartozó, adott kontextuson belüli információ halmazt jelöl.

A dolgozat további részében példaként a téma alapjául szolgáló webalkalmazás szolgáltatja a fejlesztési folyamat egyes lépéseinek a bemutatását.

Példa: Doktori Adatbázis – ***Elemzési fázis***
A Doktori Adatbázisnak, mint web alapú információs rendszernek a víziója: *Információt szolgáltat a Doktori Iskola programjairól, hallgatóiról, oktatóiról, kurzusairól, illetve hírekről, konferenciákról.*

Követelményelemzés

A követelményelemzés az a folyamat, ahol meghatározzuk, illetve feltárjuk, hogy milyen alkalmazásnak kell elkészülnie. A követelmény egy feltétel vagy képesség, aminek az alkalmazásnak meg kell felelnie.

A követelményelemzés nem tartozik az egyszerűbb feladatok közé, amely a hagyományos szoftverekhez képest a webalkalmazások esetén további nehézségeket jelent. Ilyenek például a következőek:

- egy webalkalmazásnak *egynél több belépési pontja* is lehet,
- a megrendelők többnyire csak *részleges információkkal* tudnak szolgálni az elvégzendő feladatokkal szemben – csak a vízióval,
- rendkívül *gyorsan változnak a körülmények*, a technológiák, az erőforrások,
- *nincs kialakított, szisztematikus fejlesztési irányelv*.

A használati esetek azonban segítenek, hogy feltárhassuk a rendszerrel szemben támasztott követelményeket, illetve, hogy biztosítsuk azok teljességét és helytállását. A követelményeket épp ezért ezen a formális és technikai információktól mentes nyelven kell leírni, amit a megrendelő – aki általában nem járatos az alkalmazásfejlesztésben – is megért. A követelményeknek két kategóriáját különböztetjük meg: funkcionális és nem-funkcionális követelmények.

A *funkcionális követelmények* rögzítik azon jellemzőket, amelyeket a rendszernek képesnek kell lennie végrehajtani. Általában a rendszer viselkedését határozzuk meg vele a különböző felhasználói események kapcsán, ami kapcsolódhat a tartalomhoz, a szerkezethez, a megjelenítéshez vagy akár a felhasználói profilhoz is.

A *nem-funkcionális követelmények* rögzítik az olyan elvárásokat, mint például a teljesítmény, megbízhatóság, bővíthetőség vagy például a működési környezet.

Példa: Doktori Adatbázis – ***Elemzési fázis – Követelményelemzés***
A Doktori Adatbázisnak felhasználói információkat szeretnének kapni a programokról, hallgatókról, oktatókról és kurzusokról. Az egyes elemek megtekintése során szeretnének látni kapcsolódó információkat konferenciákról és publikációkról. *A felhasználók végezhetnek kereséseket is.* *Az alkalmazásnak a leggyakrabban használt böngészőkre kell optimalizáltnak lennie.* *Van-e valamilyen megjelenítési irányelve az oldalaknak?*

A követelmények összetettsége és sokszínűsége miatt az elemzést többnyire nem egyedül egy személy végzi, hanem többen is részt vesznek benne. Egyrészt az adott terület szakértője, aki a modellezendő területet ismeri – esetünkben mondjuk a Doktori Iskola titkára, vagy vezetője. A szerkezetért felelős tervező, aki járatos a szoftverfejlesztésben és vezeti a továbbiakban a teljes folyamatot. A navigációért felelős tervező, aki átlátja a használati esetek alapján elképzelhető fellépő hozzáférési és navigációs igényeket. Valamint a felhasználói felület tervezője, aki a vizuális modellezésért lesz felelős.

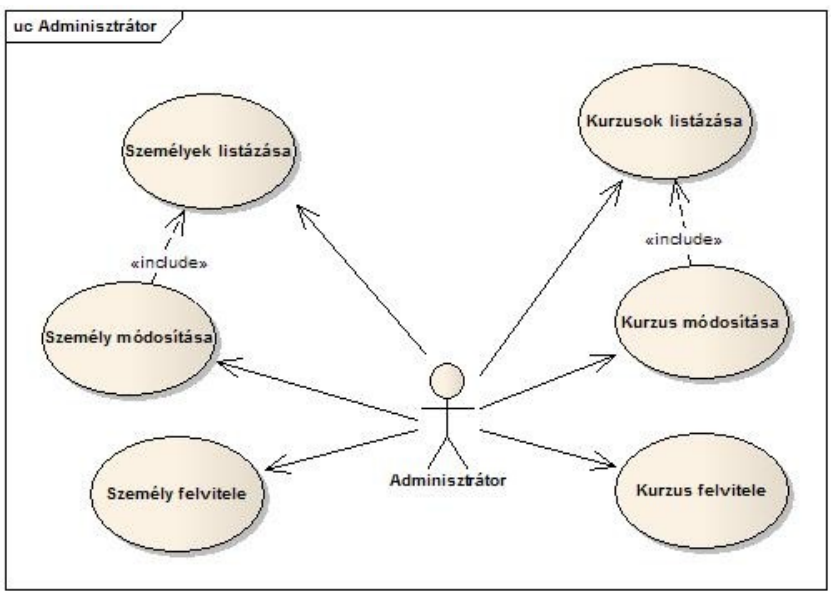
A használati esetek segítségével az alkalmazás felhasználó központú kapcsolatait tudjuk kifejezni, egyúttal meghatározva azon résztvevőket (aktorokat) akik az alkalmazást használni fogják, illetve, hogy az alkalmazásnak milyen funkcionalitásokkal kell rendelkeznie az egyes aktorok esetében.

A legjobb módszer arra, hogy megtaláljuk a megfelelő használati eseteket, hogy ha megvizsgáljuk, hogy az egyes aktorok mit várnak el a rendszertől. Ide leginkább a funkcionális követelmények – mint például az információ tartalom, vagy a navigáció – tartoznak.

Példa: Doktori Adatbázis – ***Elemzési fázis – Követelményelemzés – Aktorok***

A szöveges elvárások átolvasása után a következő aktorok azonosíthatóak:

- *felhasználó – aki a nyilvános felületen böngészik*
- *oktató (témavezető)*
- *adminisztrátor*
- *titkár és vezető*
- *hallgató (opcionális)*



11. ábra: Adminisztrátor használati esetek - részlet

A webalkalmazás használati eseteinek létrehozásához és a követelményelemzés során alkalmazható lépésekről számos modellezési módszert találhatunk az irodalomban [23] [24].

Az ajánlott lépések:

- Aktorok megkeresése.
- Az aktorok által végezhető feladatok megkeresése.
- A tevékenységek használati esetekbe csoportosítása.
- Az aktorok és a használati esetek között kapcsolatok kialakítása.
- Határozzuk meg a beágyazó («include») és kiterjesztő («extend») használati eseteket.
- Egyszerűsítsük a modellt az aktorok és a használati esetek közötti öröklődésekkel.
- Készítsük el az egyes használati esetek rövid, vázlatpontokból álló részletezését.

Ezen lépéseken felül természetesen léteznek még további lehetőségek is, mint például a használati esetek fontossági sorrendjének megállapítása, hogy a fejlesztés során melyekkel foglalkozzunk először. A pontosabb érthetőség érdekében célszerű még egy „szótár" elkészítését is beiktatni, amely tartalmazza a rendszer szakterületéhez tartozó kifejezéseket és azok jelentését.

Elemzés és tervezés

Az elemzési és tervezési munkafolyamatok célja, hogy elkészítsük a követelményelemzés eredményein alapuló terveket, amelyek a webalkalmazás alapjául szolgálhatnak. Az elemzés során a funkcionális követelményeket kell figyelembe vennünk, kihagyva mind a nem-funkcionális és mind az implementációs megszorításokat. A tervezés során, ami tekinthető az elemzés finomítási folyamatának, már figyelembe vehetőek a nem-funkcionális követelmények is, de az implementációs megfontolások még továbbra sem. (A továbbiakban az elemzés és a tervezés együtt kerül tárgyalásra, és a tervezés kifejezés az elemzést is magában fogja foglalni.)

A tervezés során a modell vezérelt fejlesztés (Model Driven Development – MDA) irányelveit fogjuk követni az elképzelt webalkalmazás megtervezéshez. A tervezés lépései során a következő lépéseket hajtjuk végre, illetve modelleket állítjuk elő:

- architekturális tervezés és modell
- szerkezeti (koncepcionális) tervezés és modell
- navigációs tervezés és modell
- prezentációs tervezés és modell

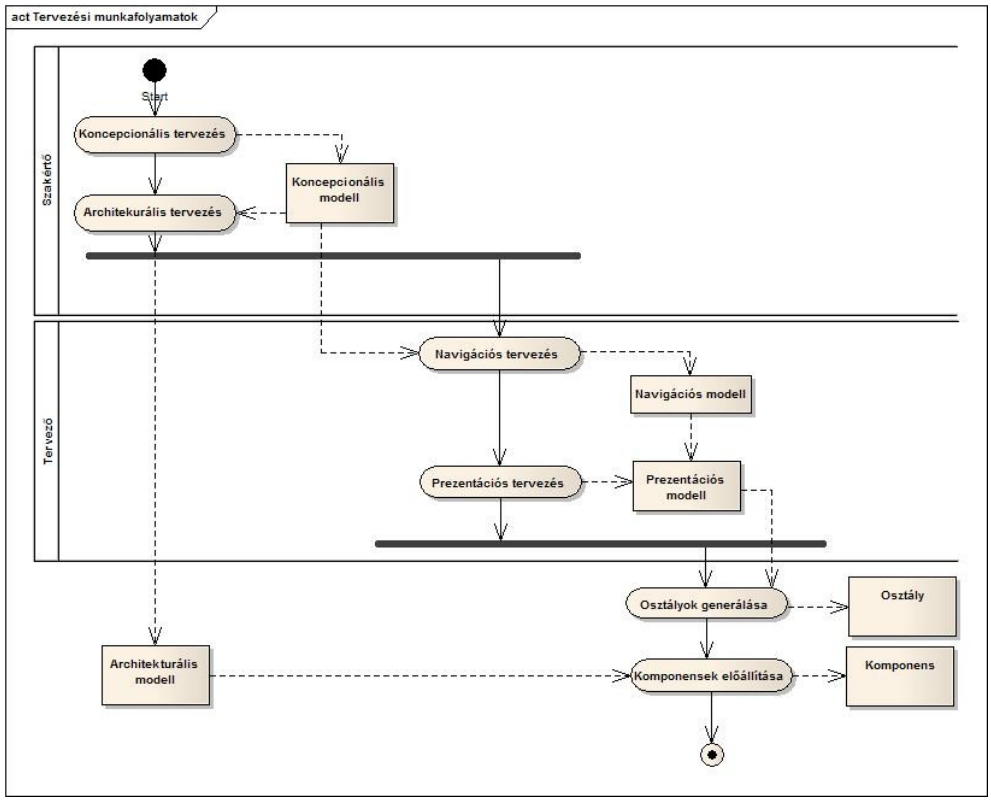
Az egyes lépések során a legfontosabb UML eszköz az osztálydiagram lesz, amelyet a kiterjesztési mechanizmusa segítségével a szakterülethez készített profil segítségével alkalmazunk.

Az *architekturális tervezés* célja, hogy meghatározza az igénybe venni kívánt architektúrára vonatkozó kihelyezési modelleket. Ehhez meg kell határoznunk az alrendszereket, az interfészeiket, a hálózati kapcsolatukat valamint az olyan speciális követelményeket, mint a perzisztencia, elosztottság és teljesítmény.

A *koncepcionális tervezés* során a cél a rendszer szakterületi modelljének elkészítése és megjelenítése, mint egy objektum-orientált osztálymodell. Itt felhasználjuk azokat bővítéseket, amelyeket az UML metamodelljében végeztünk, hogy a szakterületre legjobban illeszkedő speciális kifejezéseket használhassuk. A továbbiakban ez a modell fog a rákövetkező lépések alapjául szolgálni.

A *navigációs tervezés* során meghatározzuk azt a szerkezetet, amelyen az alkalmazáson belüli navigáció alapulni fog. Alapja a koncepcionális modell osztálydiagramja és az eredmény egy újabb osztálydiagram lesz, ami a koncepcionális modell egyfajta nézetének is tekinthető. A diagramban természetesen újabb osztályok is megjelenhetnek, illetve bizonyosak hiányozhatnak is, attól függően, hogy az optimális navigáció megvalósítható legyen.

A felhasználó felület különböző aspektusai a prezentációs modell segítségével kerülnek kifejezésre. A navigációs diagram alapján és egy általunk meghatározott algoritmus segítségével az oldalak statikus szerkezete egy adott kontextus esetén könnyedén előállítható, ami tartalmazza a megjelenítendő információt és a hozzá kapcsolódó menüszerkezetet. Ezen felül elkészíthető egy dinamikus modell is az állapotátmenet diagramok segítségével, amely kifejezheti ezen navigációs modell dinamikus viselkedését is.



12. ábra: A tervezés munkafolyamatai

A továbbiakban az egyes lépéseket és az ott előálló modelleket fogjuk megvizsgálni.

Koncepcionális tervezés

A koncepcionális tervezés folyamatában a cél egy olyan modell létrehozása, amely az alkalmazás – és a követelményelemzési fázisban azonosított felhasználók – szempontjából meghatározó koncepciókat tartalmazza. A legfontosabb a szakterület összefüggéseinek és jelentéstartalmainak az összefogása úgy, hogy még a lehető legkisebb mértékben sem vesszük figyelembe a navigációs, a prezentációs vagy akár a felhasználási szempontokat. Annak eldöntése, hogy az egyes összefüggő részek miként alkothatnak egy oldalt, vagy miként tudunk ide eljutni az alkalmazásban majd csak a későbbi fázisokban alakul ki. Ezen tartományi összefüggéseket tartalmazó modelleket *Számítógép-független modelleknek* (Computational Independent Model – CIM) szokták nevezni a modell-vezérelt fejlesztési folyamatokban.

A szakterület szerkezeti modelljének elkészítéséhez a már jól ismert objektum-orientált fejlesztési módszereket követhetjünk, mint például:

- osztályok azonosítása, mint például a *Személy* és a *Kurzus*,
- legfontosabb attribútumok és műveletek meghatározása,
- az osztályok közötti kapcsolatok feltárása,
- összetartozások és függőségek keresése, azaz *Publikáció* nem létezhet önmagában, csak *Személyhez* kapcsolódóan,
- öröklődési hierarchiák feltérképezése, mint például a *Személy különböző szerepkörei*,
- megszorítások meghatározása, mint például egy *Hallgató* esetén az abszolutórium megszerzésétől számított három éven belül jelentkezhet fokozatszerzésre.

*Példa: Doktori Adatbázis – **Elemzési fázis – Elemzés és tervezés – Koncepcionális terv***
A szöveges elvárások átolvasása után a következő osztályok azonosíthatóak: Személy, Oktató, Hallgató, Publikáció, Tudományos Fokozat, Munkahely, Kurzus, … … *A személyek a következő attribútumokkal rendelkezhetnek: Név, Születési dátum, …* … *A Publikáció kulcsszavakat tartalmazhat, és önállóan nem létezhet.* *Egy személy többféle szerepkörben is feltűnhet.* *Egy hallgatónak legfeljebb két témavezetője lehet.* *A Bizottságokban csak Tudományos fokozattal rendelkező személyek vehetnek részt.* …

A szöveges követelmények és az első körben kialakuló osztályok között megfigyelhetjük, hogy egy adott személy attól függően, hogy épp *Oktatóként* vagy épp *Hallgatóként* vesz részt egy együttműködésben eltérő viselkedéssel rendelkezik. A modell elkészítése során ezért figyelembe kell vennünk ezt az eltérő viselkedési módot.

Szerepkörök modellezése koncepcionális szinten

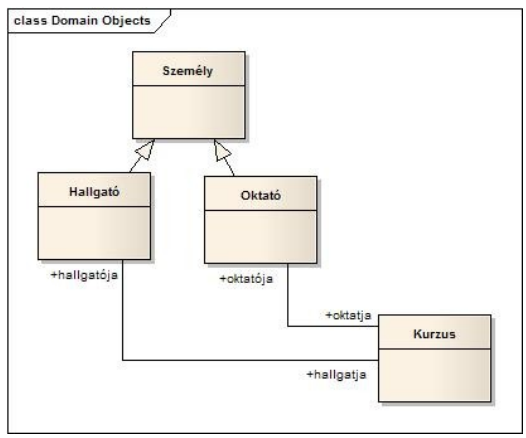
A vizsgált példánkhoz visszatérve, egy személy jelentkezésekor, majd felvételét követően hallgatóként fog szerepelni. Ez azt jelenti, hogy az objektumnak képesnek kell lennie a hallgató objektumok közötti üzenetek kezelésére és értelmezésére. Később, ugyanez a személy oktatóként alkalmazásra kerül, így az objektumnak képessé kell válnia, hogy az oktatók közötti üzenetváltásokat tudja kezelni. Ráadásul az is előfordulhat, hogy egyidejűleg hallgató is és oktató is ugyanazon a személy. Ennek megfelelően az adott objektumnak tudnia kell viselkedni, mint oktató, és mint hallgató is, annak ellenére, hogy két különböző osztály tagja (egyidejűleg).

Hasonló viselkedésbeli különbségeket fedezhetünk fel a kereskedelmi webalkalmazásokban is, ahol az egyes termékek attól függően mutatnak más és más viselkedést, hogy éppen a számlázási alrendszerben egy rendelési tételként jelenik meg, vagy éppen a napi ajánlatok oldalán szerepel, esetleg egy ajándékként kerül megjelenítésre.

Mindezek alapján láthatjuk, hogy az objektumok annak megfelelően változtatják a jellemzőiket, hogy épp egymással kapcsolatba lépnek, együttműködnek vagy éppen milyen irányból értük el. Ezért a célunk az, hogy képesek legyünk megvalósítani az objektumoknak az eltérő kontextusok alapján történő különböző viselkedésüket (mind adat, mind tevékenységek szintjén). Ez az eltérő viselkedés azonban nem csak akkor jön elő, amikor az adott kontextusnak megfelelő speciális üzeneteket küldünk az objektumoknak, hanem akkor is, amikor navigálunk közöttük a webalkalmazásban.

A szerepkört a következőképpen írhatjuk le: azon jellemzők halmaza, amelyek egy objektum számára ahhoz szükségesek, hogy egy adott kontextusban elérhető legyen. Amikor pedig egy objektum úgy viselkedik, ahogyan azt a kontextus megköveteli, akkor azt mondhatjuk, hogy az objektum az adott szerepkörben szerepel.

Tekintsük a következő egyszerű példát a Doktori Iskola esetében: a hallgatók kurzusokat vesznek fel, amelyeket az oktatók tartanak. A koncepcionális elemzés során a hagyományos objektumorientált szemlélet alapján már az elemzés fázisában kiderül, hogy szükség lesz Hallgató és Oktató osztályokra, és tudjuk, hogy mindkettő a Személy osztályon alapul. A tervezés eredményeképp a következő ábrát kapjuk.



13. ábra: Hagyományos OO koncepcionális modell

Azonban ez a modell több problémát is felvet. Egyfelől, az *Oktató* és a *Hallgató* is a *Személy* osztály leszármazottja. Mi történik akkor, ha egy hallgató sikeresen befejezte tanulmányait és oktató lesz? Egyszerre lehet-e hallgatója is és oktatója is a tárgynak? Ez utóbbi kérdésre még tudunk is választ adni egy megszorítás segítségével, de az első kérdés esetében a modell nem működik megfelelően. Másrészt a kurzusok idővel megszűnhetnek, ami a felvétel szempontjából fontos lehet. Az együttműködések során általában azokra a kurzusokra van szükségünk, amelyek éppen léteznek. Viszont a már lezárt kurzusok esetében további jellemzőket is megadhatunk, mint például a lezárás dátuma, vagy a lezárás oka. Ezek viszont olyan attribútumok, amelyek csak akkor szükségesek, ha ilyen kontextusban keresünk kurzusokat. Az alapesetben a kontextus az aktív kurzusokra vonatkozik.

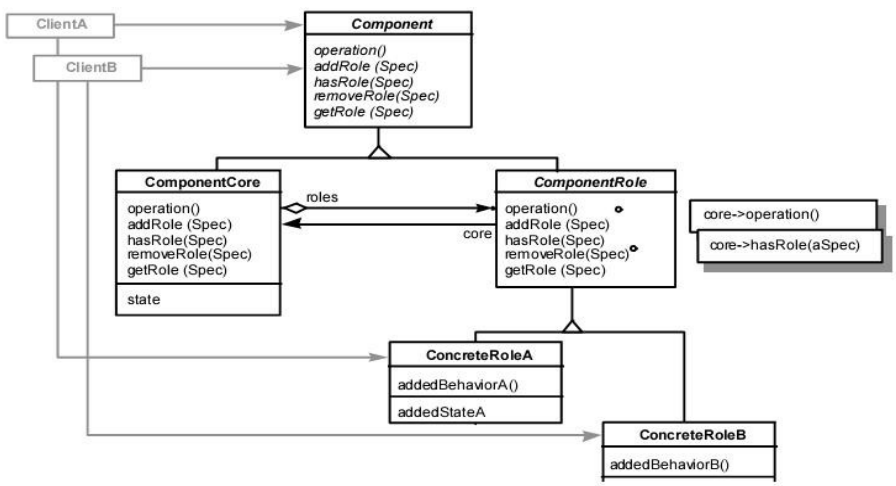
Az egyik megoldás lehetne erre, hogy a kurzus esetében bevezetünk további alosztályokat, mint például *AktívKurzus* és *LezártKurzus*. Azonban ez a megközelítés nem adja a megfelelő megoldást, miután előfordulhat, hogy egy objektumnak idővel az egyikből át kell kerülnie a másikba, ami az OO szemléletmódban nem tartozik a támogatott módosítások körébe. Alternatív megoldásként számításba vehetjük, hogy

nyilvántartjuk az egyes objektum aktuális állapotát, mint például aktív vagy lezárt. Ebben az esetben támaszkodhatunk a tervezési minták között Állapot (State) [4] néven ismert módszert, azonban ez a lehetőség nem számol azzal, hogy egy objektum egy időben akár több állapotban is lehet.

Az objektum-orientált paradigmában nincs „magától értetődő" szerkezet az objektumoknak a különböző kontextusok esetén a természetükből eredő jellegzetességek elválasztására. Ez a problémakör azonban eltérő módon tud megjelenni a koncepcionális, és eltérő módon a navigációs tervezés során. Ráadásul ez a probléma nem csak a webalkalmazások sajátossága, hanem egy általános szoftvertervezési probléma.

A megoldás a szerepkörök általános értelmezése biztosíthatja, amit ha beemelünk a koncepcionális modellbe, akkor biztosítjuk annak lehetőségét, hogy az alkalmazásunk a későbbiek, mondjuk egy addig előre nem látott típus megjelenése során is bővíthető marad. Az irodalomban többen is foglalkoztak a szerepkörök modellezésével [25] egyesek pedig saját jelöléseket is bevezettek a diagramokon [26].

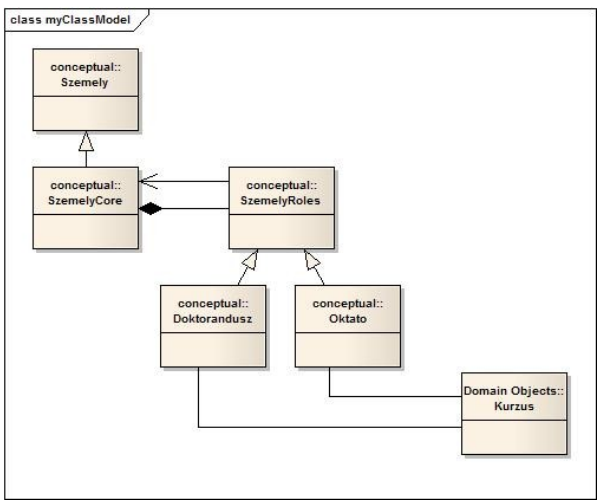
A továbbiakban egy szerkezeti modellt fogunk alkalmazni, amely kompozíció segítségével próbál megoldást nyújtani az objektumok által betölthető szerepek kezelésére, ráadásul figyelembe véve, hogy egyidejűleg akár több szerepkört is felvehetnek. A koncepció az alábbi ábrán látható:



14. ábra: Szerepkörök modellezése [27]

A szerepköröknek, mint elsődleges osztályoknak a bevezetése lehetővé teszi, hogy a fentebb vázolt problémákat a koncepcionális tervezés során figyelembe tudjuk venni. Ennek segítségével akár azt is ki tudjuk fejezni, hogy egy objektum egy adott időben több másik objektummal más és más szerepkör kapcsán áll összefüggésben. Ezáltal meg tudjuk különböztetni azokat az attribútumokat, amelyek egy adott szerepkör betöltéséhez szükségesek, azoktól, amelyek az objektum természetéből eredő attribútumok és mindig változatlanok maradnak függetlenül attól, hogy az objektum épp milyen szerepben van.

Ezek alapján az áttervezett koncepcionális modell a következőképpen alakul:



15. ábra: Példa a szerepkörre

Mindezek után kanyarodjunk vissza az eredeti irányhoz, a koncepcionális tervezés során előállítandó szerkezeti modell elkészítéséhez, ami a szakterület modellje figyelmen kívül hagyva minden más szempontot. Ebből következően hasonló lesz a hagyományos szoftverfejlesztési technológiák során előálló szakterületi modellekhez. A tevékenység eredményeképp egy UML osztálydiagramot kapunk, ami a szakterületi modellünket fogja tartalmazni a tartományunkhoz meghatározott tartomány specifikus információtartalommal.

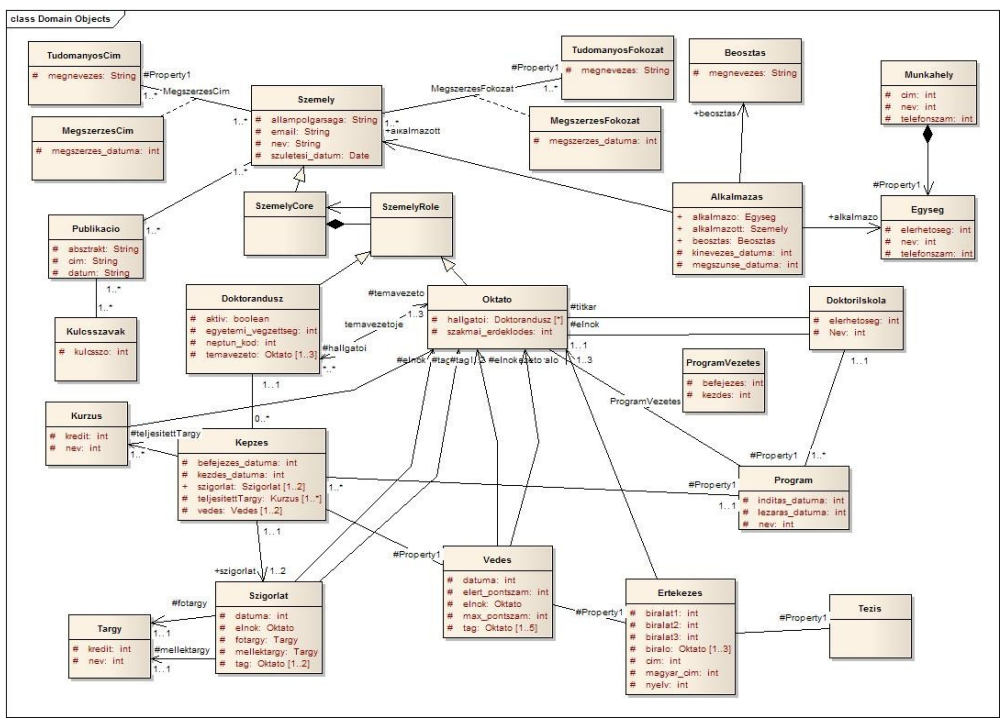
Előfordulhat, hogy túl sok osztály jönne létre az egyes szakterületek esetén, ilyenkor célszerű a valamilyen szempontok szerint kapcsolódó osztályokat *csomagokba* összefogni. A létrejött diagram pedig a továbbiakban fontos szerepet fog betölteni,

mert mind a navigációs, mind a prezentációs modellek elkészítésénél ebből fogunk kiindulni. Az osztályok közötti asszociációk például a navigáció alapjául szolgáló hivatkozások létrehozásához használhatjuk fel.

Szerkezeti modell

A szerkezeti modellek elkészítésénél a leggyakrabban alkalmazott eszközök az osztályok, az asszociációk és a csomagok. Egy általános webalkalmazás esetén a használati esetek és a tevékenység diagramok szolgáltatják az alapot. Az adatközpontú webalkalmazások esetén (amelyekkel foglalkozunk) a szakterület szerkezeti modellje sokkal összetettebb, mint a korábban bemutatott ábrák. Nagyon fontos, hogy az osztálydiagram megfelelő részletességgel és minőségben mutassa be a modellezett területet, mert a továbbiakban erre fogunk építeni.

A példánk a Doktori Iskola szakterületéhez kapcsolódó szerkezeti modell előállítása. Az elkészítés alapját a Doktori Iskola szabályzata nyújtotta.



16. ábra: Szerkezeti modell - Doktori Iskola

Navigáció tervezése

A navigáció megtervezése egy webalkalmazás esetében döntő fontosságú lépés. Még a legegyszerűbb, kis mélységű hierarchiával rendelkező esetekben is nagyon összetetté válhat a modell az újonnan hozzáadott linkek miatt. Egyrészről a hozzáadott linkek felvételével megkönnyíthetjük az egyes információs elemek elérhetőségét, másrészt viszont megnehezíthetik a diagramok átláthatóságát. Összességében egy jól strukturált navigáció kialakítását teszi lehetővé, így a navigációs modell nem csak a dokumentáció elkészítésében lesz nagyon hasznos, hanem az alapvető bejárási lehetőségek feltérképezésében is.

A navigációs tervezés során a webalkalmazásunk szerkezetét állítjuk elő, ahol a tervezőnek meghatározó fontosságú döntéseket kell hoznia, amelyekben rögzíti, hogy mely osztályoknak kell szerepelniük (amit természetesen befolyásolhat, hogy mely aktorhoz kapcsolódó nézetet vizsgálunk), illetve milyen navigációs utakat állít fel annak érdekében, hogy a termék biztosítani tudja az elvárt funkcionalitást. Ezeket a döntéseket a tervező a koncepcionális modellen és a használati esetekben rögzített követelmények alapján állítja fel.

A navigációs modell elkészítését célszerű két lépésre bontani. Az elsőben azon osztályokat határozzuk meg, amelyek a navigáció során közvetlenül elérhetőek lesznek. Ebben a lépésben a navigációs osztálydiagram készül el, amely specializált UML osztályokat és asszociációkat használ. A második lépés, hogy ezt a diagramot kiegészítjük a navigáció során alkalmazandó elérési struktúrákkal és menüszerkezettel, amivel megkapjuk a navigációs szerkezeti diagramot. Ez a diagram mutatja meg ténylegesen, hogy az egyes elemek hogyan érhetőek el az alkalmazáson belül.

Az elérhető módszertanok közös vonása, hogy osztályokat (csomópontokat) és azokat összekötő hivatkozásokat (linkeket) használnak a navigáció kifejezésére. Azonban az egyes módszertanok eltérő módon vélekednek a navigációs diagram által megjelenített tartalomról. A WebML például a navigációt összekapcsolja a feldolgozással, és ennek megfelelően a linkek szemantikáját is másképp értelmezi. A diagramon weboldalakat jelenít meg, mint navigációs célokat, amelyek együtt tartalmazzák a koncepciós modell entitásait és a rajtuk végezhető műveleteket. Ezen felül a linkek esetén szükséges megjelölni, hogy melyek tartoznak egy adott elem módosításához, hogy a technikai szempontból fontos adatbázis azonosítókat át tudja

adni az egyes lépések között. A mi esetünkben erre nincs szükség, mert a diagram nem ebből a célból készül, másrészt ez az alkalmazás logika feladata lesz.

Az általunk használt példánál maradva a webalkalmazásunk navigációs modelljének a célja, hogy meghatározza az egyes lépések során mely koncepcionális osztályról mely másikra tudunk eljutni a meghatározott asszociációk mentén. A különböző típusú felhasználókhoz eltérő navigációs utak is tartozhatnak, mert eltérő feladatokat hajthatnak végre a rendszer használata során. Egy oktató esetén mondjuk a hozzá tartozó doktoranduszok, az általa oktatott kurzusok fognak megjelenni, mint lehetséges navigációs célok, egy adminisztrátor esetén ezen információk mellett még megjelenhetnek például a karbantartó utasításokhoz szükséges linkek is. Azonban az első lépésnél még csak azt akarjuk elérni, hogy a navigáció szempontjából fontos osztályokat meghatározzuk.

Navigációs osztály modell

A navigációs osztály modell a szerkezeti modell egy részgráfjának tekinthető, amelyből elhagytuk azon osztályokat, amelyek nem relevánsak a navigációs során és más osztályok származtatott attribútumaiként jelennek meg. A mi példánk esetében ez a publikációkhoz kapcsolódó kulcsszavak esetén fedezhető fel, mert külön a kulcsszavakat, mint navigációs célt nem fogjuk érinteni, viszont az egyes publikációk esetén szeretnénk látni a kapcsolódó kulcsszavakat.

A modell felépítéséhez a meta-modellben definiált navigációs csomópont és link elemeket fogjuk használni, amelyek az osztály és az asszociáció leszármazottai. Megadhatjuk továbbá egy speciális attribútummal azon csomópontokat, amelyek az alkalmazás belépési pontjait szolgáltatják, illetve azokat is, amelyeknek bárhonnan elérhetőnek kell lenniük explicit linkek nélkül is. A már említett linkek esetén meghatározhatjuk azokat, amelyeket a rendszernek magától követnie kell, hogy a későbbiekben meghatározandó navigációs kontextus automatikusan előálljon. A navigáció kifejezésre használt linkek (az asszociáció pontosításai) itt már minden esetben irányítottnak kell lennie az egyértelmű haladási irány kifejezésre. A koncepcionális modell eredetileg irányítatlan vagy kétirányú asszociációit fel kell bontani két irányított linkre.

A navigációs osztály által reprezentált osztályok lesznek azok, amelyek példányai érinthetőek a navigáció során. A navigációs osztályok nevei megegyeznek a hozzá

kapcsolódó koncepcionális osztályéval és a diagramokon ezen osztályok saját sztereotípiával lesznek megjelölve. A navigációs osztályok ezen felül tartalmazhatnak további (származtatott) attribútumokat is (melyeket a nevük előtt találhat / jel vezet be). A származtatási feltételek bármilyen nyelvi kifejezéssel megadhatóak.

A következő lépések segíthetnek a navigációs osztály modell elkészítésében:

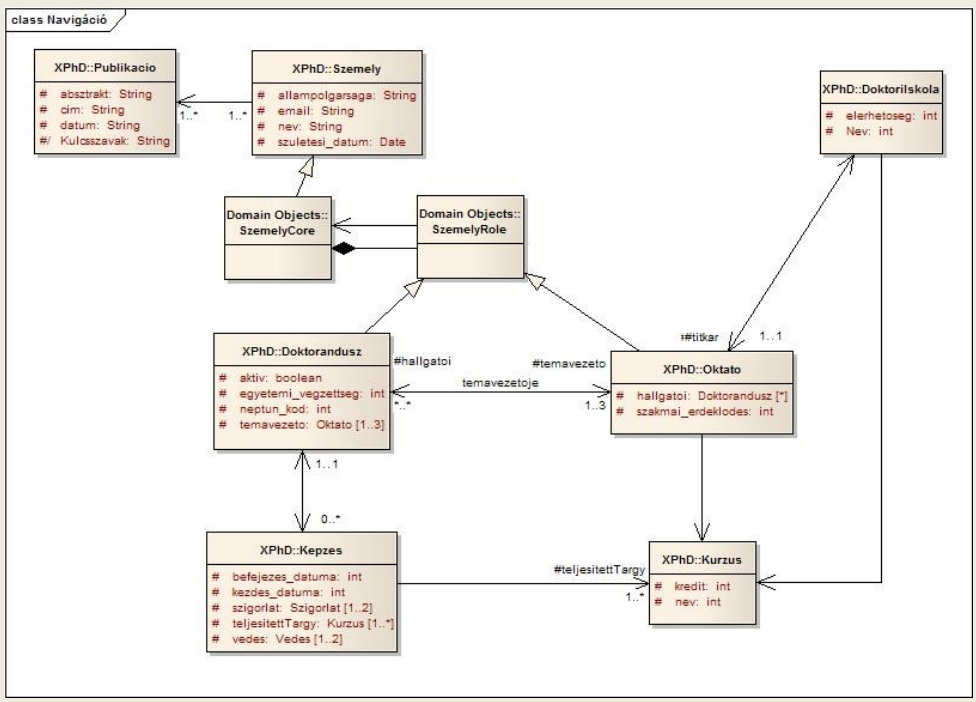
- A navigációs osztályoknak megfeleltethetőek a koncepcionális osztályok.
- A navigációs osztály egy attribútuma megfeleltethető egy, a navigáció szempontjából irreleváns, de a megjelenítés szempontjából fontos koncepcionális osztálynak.
- A navigációs linkek közvetlenül a koncepcionális asszociációkat veszik alapul.
- Az asszociációkat át kell alakítani navigálható linkekké, egyértelműen kifejezve, melyik a forrás és melyik a cél.
- Ezen koncepcionális asszociációk közül elhagyhatóak azok, amelyeknél a résztvevő osztályt származtatott attribútumként emeltünk be.
- A navigáció osztályok attribútumai megfeleltethetőek a koncepcionális osztályok attribútumainak.
- Újabb linkek vehetőek fel, hogy a navigációt elősegítsük.

A modellezés eredményeképp egy újabb osztálydiagramot kapunk, amelyen a navigációs osztályok tartalmazhatnak további attribútumokat illetve metódusokat. A diagramról továbbá eltávolításra kerülnek az asszociációkhoz tartozó asszociációs osztályok is, mert a navigáció kialakításában nem hordoznak mérvadó információkat. Ezzel szemben a prezentációs modellben, ahol az összetartozó információkat (kontextust) kell meghatározni, már fontos szerephez jutnak, mert a segítségükkel releváns információkat tudunk elhelyezni a képernyőn megjelenítendő tartalomban.

Példa: Doktori Adatbázis – ***Elemzési fázis – Elemzés és tervezés – Navigációs osztályterv***

A koncepcionális modell alapján elkészült a navigáció megtervezésének első lépéseként a navigációs osztálydiagram.
Meghatároztuk a navigáció szempontjából fontos osztályokat, ennek eredményeképp a Kulcsszavak osztály elhagyásra került. Értékeit a Publikáció osztály fogja egy származtatott attribútum segítségével megjeleníteni.
Az alapvető navigációs utak is feltérképezésre kerültek, mely során egy új irány is felvételre került a Doktori Iskola és a Kurzus között.

Az eredményül kapott navigációs modell az alább található ábrán látható.



17. ábra: Navigációs osztály diagram

Az ábrán jól megfigyelhető, hogy a Kurzus csomóponthoz több különböző úton is el lehet jutni, attól függően, hogy egy hallgató, egy oktató, vagy éppen a Doktori Iskolán keresztül szeretnénk róla információkat elérni.

Egy jól megtervezett webalkalmazás esetén ezeket az eltérő navigáción keresztül történő eléréseket ajánlatos lenne figyelembe venni. Ami a mi esetünkben ez a következőt jelentheti a Kurzus csomópont elérése során:

- a hallgató irányából megközelítve hasznos lenne ha megjelennének alapvető információk, mint például ki oktatja, mi a tematikája, illetve ami kimondottan csak a hallgató esetén jöhet számításba – a kurzus felvételére vonatkozó adatok és ennek a lehetőségét biztosító tevékenység linkek megjelenítése,
- az oktatók szempontjából az előbbiek már nem relevánsak, az ő esetükben viszont a kurzus meghirdetése merülhet fel, mint igény,
- a Doktori Iskola oldaláról szemlélve pedig a fentebb említett plusz szolgáltatások egyike sem mérvadó.

Szerepkörök modellezése a navigáció során

A problémát jobban megfigyelve azt tapasztaljuk, hogy nagyon hasonló a koncepcionális tervezés során felmerült kérdéshez: hogyan valósítható meg, hogy egy objektum több szerepkört is betöltsön?

Azonban a mostani esetben az információ alapjául szolgáló osztály a koncepcionális modellben egymaga szerepel, azon a szinten nem szükséges megkülönböztetni a fentebb említett szempontok alapján. Ezzel szemben azt szeretnénk elérni, hogy a csomópontunk eltérő módon viselkedjen annak függvényében, hogy mely navigációs úton keresztül értük el. Ennek eredményeképp nemcsak a megjelenő információt, hanem a továbbhaladási irányokat is módosítani lehetne, mondjuk további kifelé mutató linkek felvételével.

Egy lehetséges megoldás lehetne erre, ha a probléma feldolgozását az alkalmazás logika szintjére helyezzük át és a csomópont elérésekor „programozottan” hajtanánk végre a csomópont kibővítését az adott navigációs iránynak megfelelően. Egy másik megoldás pedig talán az lehetne, hogy több, különböző Kurzus csomópontot veszünk fel, annak megfelelően, hogy milyen „nézetre” van szükségünk. Ezzel viszont az eredeti kurzus alapinformációit megsokszoroznánk, ami karbantartási nehézségekhez vezethet.

Mindezeket egybevetve a következő tervezési problémákra szeretnénk megoldást találni:

- Hogyan tudjuk egyértelműen jelezni, hogy *egy csomópont eltérő információkat szolgáltathat* annak függvényében, hogy milyen navigációs úton értük el?
- Hogyan jelezzük azt, hogy *egy csomópont eltérő továbbhaladási linkeket kínál fel* annak függvényében, hogy azt milyen úton értük el?

Azonban az OO világban az objektumok a viselkedésükben nem tesznek különbséget attól függően, hogy az üzenetet melyik objektumtól kapta, mindig ugyan úgy reagálnak. A navigáció során pedig épp ezt a viselkedést szeretnénk az objektumainktól elvárni.

Az eltérő viselkedés eredhet magából a navigációból, azaz az eltérő forrás csomópontok eltérő módon szeretnék „látni" a cél csomópontot, illetve eredhet az alkalmazást felhasználó aktorok eltérő szerepköréből (felhasználói profiljából – adminisztrátor vagy vendég). A tervezésnél ezeket a szempontokat kell figyelembe vennünk, és ezek alapján kell elkészítenünk a módosított navigációs diagramokat.

Természetesen a navigációs tervezés során a különböző szerepkörök meghatározásánál elsődlegesen a koncepcionális modellben kialakított szerepek kerülnek alkalmazásra a navigációs csomópontok származtatásánál. A mi példánk esetében ez a Személy osztály két eltérő szerepköre esetén a Hallgató és az Oktató szerepeket fogja jelenteni. Miután ez már a koncepcionális tervezés során kialakult, a navigáció során automatikusan felhasználhatjuk ezeket a különböző navigációs célcsomópontok különböző nézetének a kialakításához. Mindemellett figyelembe kell venni még azokat a lehetőségeket is, amelyek a követelményelemzés fázisában, mint felhasználói profil alakultak ki. Az egyes profilok is eltérő viselkedést várhatnak el a navigációs diagramunk csomópontjaitól. Ezen szerepkörök az alaposztály jellemzőin kívül megkövetelhetnek további attribútumok és metódusok felvételét is, hogy az az adott szempont szerinti viselkedését el tudja látni. A helyzet pedig csak bonyolódhat, ha egy ilyen szerepkör egybeeshet egy, a koncepcionális tervezés során kialakított osztály szerepkörével.

Mindezek mellett kialakulhatnak olyan esetek is, amikor a koncepcionális modellben egy osztály rendelkezik különböző szerepkörökkel, de a navigációs modellben már nem jelennek meg. Ilyen akkor fordulhat elő, ha pusztán viselkedésbeli szerepekről van szó és nincs megfelelőjük a navigációs modellben.

Ennek az ellenkezője is adódhat, amikor a koncepcionális osztály nem rendelkezik szerepkörrel, de a neki megfelelő navigációs csomópont már igen. Ilyenre példa, amikor új utakkal bővül a modell és ezzel együtt az új út új viselkedést is magával hozhat, miután a koncepcionális szinten nem szükséges ezen információknak a kifejezése.

Harmadrészt, a legösszetettebb esetben, előfordulhat olyan is, hogy egy koncepcionálisan már szerepkörökkel rendelkező osztályt a navigációs modellben további – navigációs – szerepekkel is fel kell ruházni. Erre példa lehet a Személy osztály Hallgató szerepkörének a viselkedése az eltérő navigációs utak során. Más és más viselkedést várhatunk el tőle, ha a Doktori Iskola úton érjük, vagy ha a Témavezető ágon érkezünk el hozzá. A viselkedéseken felül természetesen az utólagosan hozzáadott információ tartalom is változhat az elérési útnak megfelelően.

Összességében kijelenthetjük, hogy a szerepkörök bevezetése a navigációs diagramokon növeli az átfogó funkcionalitását az alkalmazásunknak. A szerepkörök megléte egy egyszerű és intuitív lehetőség arra, hogy kifejezzük az egyes csomópontok eltérő karakterisztikáját, amikor különböző linkeken keresztül érjük el azokat. Segítségükkel elérhetjük a következőket:

- A felhasználó navigációs útjának megfelelően ugyanazon objektum eltérő sajátosságokat mutathat, gondoljunk csak arra, hogy a kurzus a hallgató esetében plusz információkat is mutat, míg az oktató irányból megközelítve ezek már nem jelennek meg.
- A csomópont a hozzáférési irány alapján különböző linkeket kínál fel, a kurzus esetében a hallgatónál a tantárgy felvételét, míg az oktató esetében a tárgy meghirdetését.

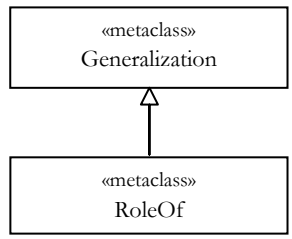
A szerepkörök bevezetésére már a koncepcionális modellezésnél láttuk, hogy az irodalomban többféle lehetőséget is kidolgoztak az objektumok eltérő viselkedésének a leírására. Ezt a szemlélet módot kell most megjeleníteni a navigációs modellben is. Ami már láttunk, hogy az OO alapelemek nem elegendőek a dinamikus viselkedés kifejezésére, mert nem tudjuk elválasztani a természetes és a környezettől függő tulajdonságokat. Az öröklődés sem biztosítja a megfelelő megoldást, mert az objektumok nem tudják megváltoztatni az osztályukat.

A tervezési minták [4] közül az Állapot (*State*) és a Díszítő (*Decorator*) minták rendelkeznek olyan tulajdonságokkal, amelyek segíthetnek a célunk elérésében. A mi megközelítésünkhöz viszont a Díszítő egy specializált változata áll a legközelebb, a Szerepkör minta (*Role pattern*) [27].

A szerepkörök használata meghatározó fontosságú a webalkalmazások szerkezeti struktúráinak modularitásában, és éppen ezért jelentősen elősegíthetik a fejlődést, a karbantartást és az újrafelhasználást. Miután az alaptípusokról leválasztottuk a szerepeket, a továbbiakban tőlük függetlenül képesek a fejlődésre, miközben az alaptípusok is megtartják alapvető jelentőségeiket. Ezzel válik lehetővé, hogy az alaptípusok esetén újabb és újabb szerepköröket vehessünk fel az eltérő viselkedésük kifejezésére. Mindezt anélkül, hogy a közös részhez hozzá kellene nyúlnunk, vagy bármit módosítanunk kellene az alaposztály absztrakcióján.

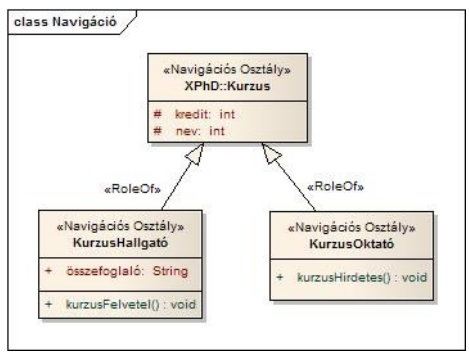
Az irodalomban több módszertan is elérhető, viszont közülük csak kevés foglalkozik a szerepközök témakörével. Az OOHDM (Object-Oriented Hypermedia Design Method) [28] esetén a diagramokat saját jelölésekkel egészítik ki a szerepkörök leírására, bár a jelölésrendszerében az UML-hez hasonló kifejezéseket használ. A mi szemléletmódunkban azonban a cél az, hogy az UML saját jelöléseit felhasználva fejezzük ki a szerepköröket. Ehhez leginkább az UWE [29] által ajánlott koncepció áll a legközelebb.

A különböző viselkedési módok bevezetésére az osztályokat a pontosítás metamodell szinten történő kiegészítésével valósítják meg, melyet a „RoleOf" névvel láttak el. Ezzel a modellek ugyan új osztályokkal bővülnek, de ezek kapcsolata az eredeti osztállyal az öröklődés kapcsán jól átlátható és érthető marad. Miután egy csomópont a navigáció során többféle szerepet is betölthet, ezért a modellben ez az osztály annyi új leszármazottal fog rendelkezni, amennyi különböző szerepet el kell látnia. Az alosztályok így öröklik a szerepkörtől független jellemzőiket és viselkedésüket és ezen felül meghatározhatják saját speciális viselkedésüket is.



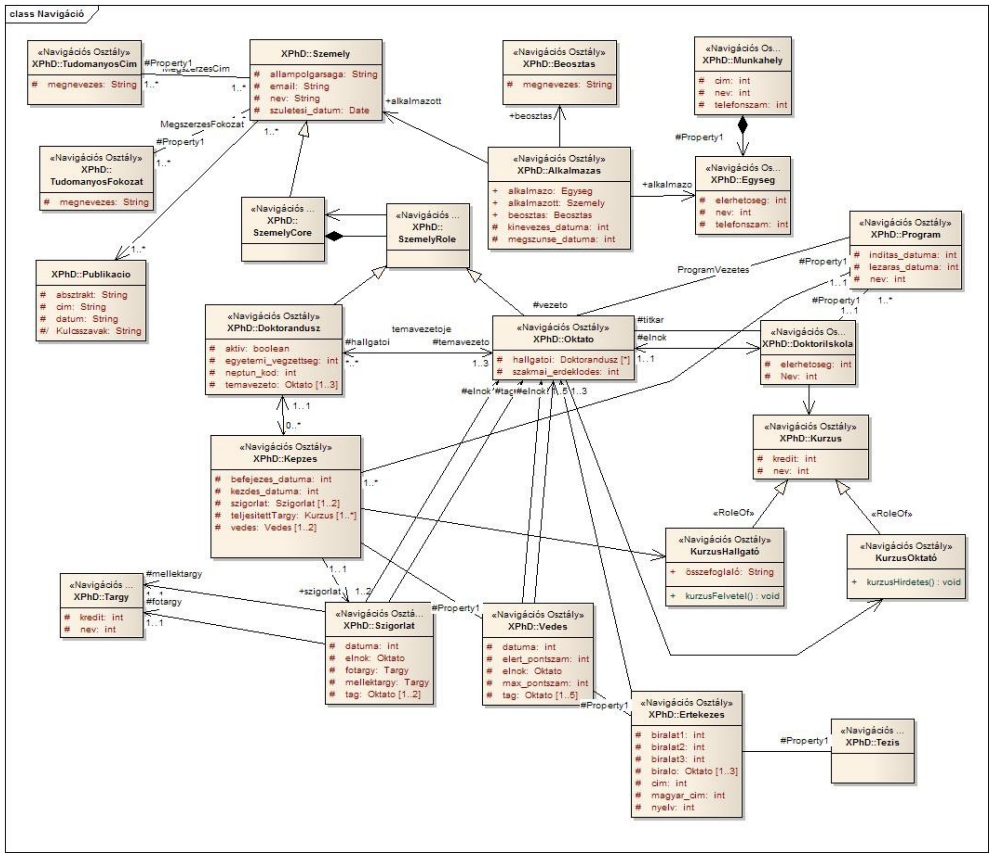
18. ábra: UML metamodell bővítése a szerepkörrel

Ennek az új osztálynak a navigációs kontextusát megadhatja az az útvonal, amelyen keresztül elértük, illetve a kontextust meghatározhatja magának a navigációt végző felhasználónak a rendszerben betöltött szerepe is. A szerepkör-osztályokból származó példányok élettartama pedig közvetlenül az ősosztályukból származó példányok élettartalmától függ, amelyhez a RoleOf linken keresztül kapcsolódik. Ebből természetesen az is következik, hogy a szerepkör-osztály példányok hozzáférhetnek mindenhez, amihez az ősosztálynak is hozzáférése van.



19. ábra: Szerepkörök a Kurzus osztályhoz

Ezen irányelvek mentén készült el a következő navigációs diagram, ami a példaként használt Doktori Iskola navigációs vázát mutatja.



20. ábra: Navigációs diagram

A koncepcionális modellel összehasonlítva a navigációs osztálydiagram számottevően nem lett bonyolultabb, bár az új jelölések bevezetése és a plusz navigációs linkek felvételével egy kicsit zsúfoltabbá tette a diagramot, de az olvashatósága nem romlott le.

Miután elkészültünk ezzel a diagrammal, következik a navigációs szerkezeti modell elkészítése, amely már tartalmazni fogja a navigáció során használt menüszerkezetet is.

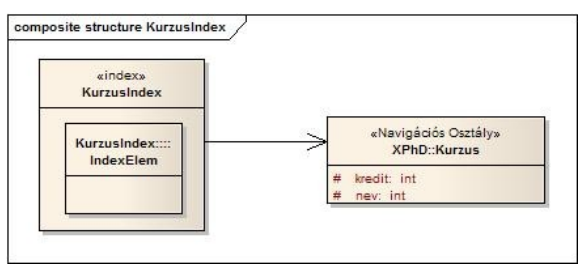
Navigációs szerkezeti modell

Ebben a lépésben a navigációs szerkezeti modellt fogjuk előállítani a navigációs osztálydiagramot alapul véve. Meghatározzuk az egyes csomópontok elérési módját, amit az alapvető navigálási szerkezetek felvételével tudunk elérni. Az ismertebb modellezési technikák alapján a legfontosabb elemek közé tartozik az index, a lekérdezés és a menü. Ezekkel a szerkezetekkel nagyon egyszerűen ki tudjuk egészíteni a már meglévő diagramunkat úgy, hogy a haladási irányok továbbra is adottak maradnak, de már sokkal jobban fog hasonlítani egy képzeletbeli oldalrészletre. Technikailag ez ugyancsak egy osztálydiagramot fog eredményezni, annyi eltéréssel, hogy még tovább részleteztük a csomópontok közötti haladási és elérési lehetőségeket.

Elemi navigációs szerkezetek

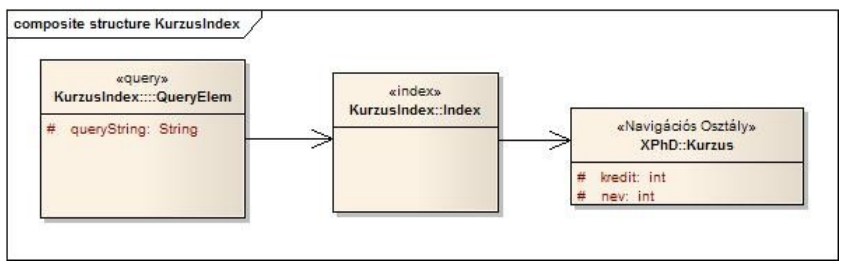
Az egyes navigációs csomópontok elérési módjait alapvetően két hozzáférési elem segítségével tudjuk megvalósítani: *index* és *lekérdezés*. A megadásukat a meta-modellben a navigációs csomópont egy pontosításaiként adtuk meg, amelyek a diagramjainkon így sztereotipizált osztályokként fognak megjelenni.

- Index: közvetlen hozzáférést biztosít egy navigációs osztály példányaihoz. A modellezéséhez egy kompozit osztályt használhatunk, amely korlátlan számú index elemet tartalmazhat. Ezek az index elemek azonosítják és hivatkozzák az egyes példányait az adott navigációs osztálynak. Az index osztály jelölésére az *«index»* sztereotípiát fogjuk használni. Segítségükkel már ki tudjuk fejezni, hogy az olyan navigációs linkek, amelyek nem egy-az-egyhez számosságú navigációs csomópontokat kötöttek össze, miként tudjuk elérni az egyes elemeket közvetlenül. Az indexek elhelyezés szinte minden ilyen esetben szükséges, hacsak az osztály nem szerepel a navigációban, mint célcsomópont.



21. ábra: Index osztály a navigáció során

- Lekérdezés: közvetlen hozzáférést biztosít egy navigációs osztály példányaihoz, de az indextől eltérően, már nem az összes elemet akarjuk elérni, hanem csak azon példányokat, amelyek eleget tesznek egy általunk (felhasználó) megadott szűrőfeltételnek. A lekérdezést tekinthetjük egyfajta speciális indexnek is, ezért az ábrázolás során is kifejezzük, hogy a lekérdezés eredményeképp mindig egy *«index»* típusú osztályhoz jutunk, amely már csak a kérdéses elemek hivatkozásait tartalmazza. A modellezéséhez szintén egy osztályt használunk, amit most a *«query»* sztereotípiával látunk el, ráadásul rendelkezik még egy attribútummal is, amely magát a lekérdezést tartalmazza. Természetesen előfordulhat olyan eset is, amikor a lekérdezésnek pontosan egy eredménye lesz, ilyenkor felmerülhetne az igény, hogy az index kihagyásával közvetlenül érjük el a megfelelő elemet, de ez plusz navigációs linkeknek a felvételét követelné meg, amivel csak „feleslegesen" szaporodnának a diagramon szereplő linkek száma.



22. ábra: Lekérdező osztály a navigáció során

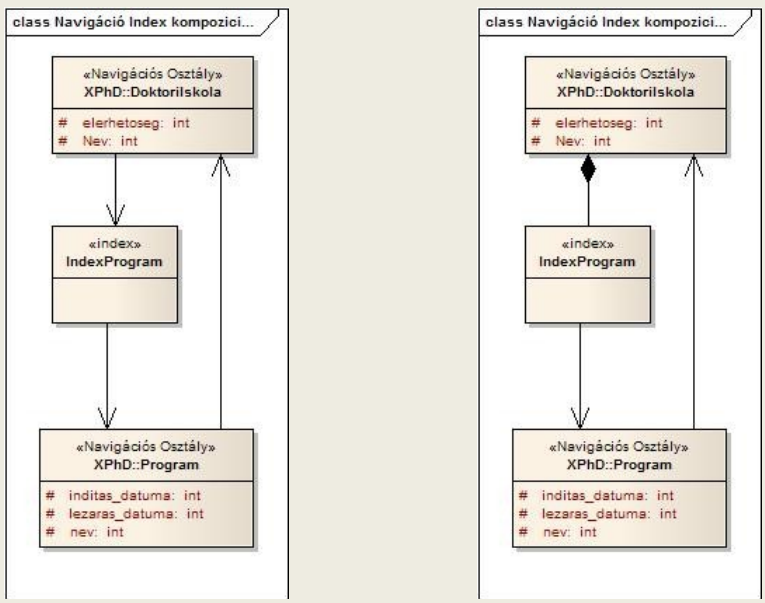
Ezen a két alapvető navigálási struktúraelemen kívül tetszőlegesen sok, hasonló jelentéssel bíró elemet is ki lehet fejezni. Ilyen lehet például fotóalbumban található képek szép egymásutánjában történő elérése. Ebben az esetben erre tekinthetünk úgy, mint egy speciális indexre, ahol az indexoldal helyett előre és visszafelé hivatkozó linkek találhatóak. Elképzelhető akár ennek és a lekérdezésnek a kombinációja is, amikor a bejárandó elemekhez egy lekérdezés segítségével jutunk el.

A navigációs osztálydiagram átalakítása során törekedjünk arra, hogy lehetőség szerint egy-a-sokhoz számosságú linkek maradjanak, melyek között a navigációt a fentebb felsorolt elemi navigációs struktúrákkal biztosítsuk. Amennyiben egy csomópontból többféle szempont szerint is elérhetünk egy másikat, akkor ennek megfelelően annyi különböző indexet hozhatunk létre, amennyit a helyzet megkíván.

Mindazonáltal előfordulhat az is, hogy a célunk nem a navigációs utak számának a növelése, hanem épp a fordítottja. Az indexek esetében ez azt jelentheti, hogy a kiindulási navigációs osztályból nem linken keresztül jutunk el az indexhez, hanem a kompozíció segítségével hozzákapcsoljuk magához a navigációs osztályhoz. Ezzel csökken a navigációs lépések száma és egy picit elősegítettük a megjelenítési modell kapcsán az adott oldal szerkezeti felépítésének a kialakítását is. Például a Doktori Iskola kapcsán a nyitó oldalon nem egy hivatkozást helyezünk el az elérhető képzési programok listájára, hanem már magán a nyitó oldalon szerepeltetjük ezeket az információkat.

Példa: Doktori Adatbázis – ***Elemzési fázis – Elemzés és tervezés – Navigációs osztályterv***

A Doktori Iskola nyitólapján szerepeljenek közvetlenül az működő képzési programok.



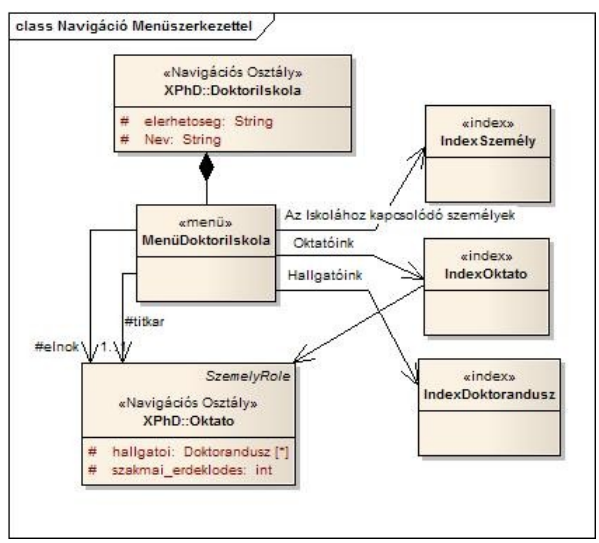
23. ábra: Indexelem kompozícióval

Miután kialakítottuk az egyes csomópontok között vezető navigációs utakat az indexek segítségével, a következő lépésben ezen utak egységekbe szervezését fogjuk elvégezni – kialakítjuk az egyes csomópontok elérési lehetőségeit menük segítségével.

Navigációs szerkezetetek egységbe fogása

A menü egy kiegészítő navigációs szerkezeti egység, amely az előző lépésben kialakított, indexekkel bővített navigációs szerkezet hatékonyabb összefogását teszi lehetővé. A menü kifejezésére az előbbiekhez hasonlóan egy sztereotipizált osztályt fogunk használni.

- Menü: összetett elem, amely a fentebb említett elemeket tartalmazhatja, mint például index, lekérdezés, navigációs osztály vagy akár egy másik menüt is. Egy olyan kompozit osztályként kell elképzelni, amelyben az elemek száma rögzített. Minden egyes elemnek van neve és tartalmaz linket egy navigációs osztályra vagy valamelyik szerkezeti elemre. A modellezésére itt is egy sztereotipizált osztály használunk a *«menü»* sztereotípiával megjelölve. Célszerű minden olyan navigációs osztály esetében kialakítani, amely tartalmaz kifelé mutató link(ek)et. A menüt az osztályhoz kompozíció segítségével kapcsoljuk hozzá, ezzel is kifejezve a csomópontok és a hozzá tartozó navigációs szerkezetnek a szoros és elválaszthatatlan összetartozását. Miután a menü osztály a meta-modellben a navigációs osztály leszármazottaként került definiálásra, így akár önállóan is szerepelhet. Erre az ad magyarázatot, hogy a webalkalmazások esetén általában létezik egy főmenü, amely minden oldalon szerepelhet.



24. ábra: Navigációs menüelem

Az ábrán az egyszerűbb átláthatóság érdekében a menü osztályt belső felépítését elhagytuk, helyette a belső menüpontok neveit a linkeken helyeztük el. A kompozit szerkezetet az index elemhez hasonlóan képzelhetjük el, csak itt indexelemek helyett menüelemek szerepelnek.

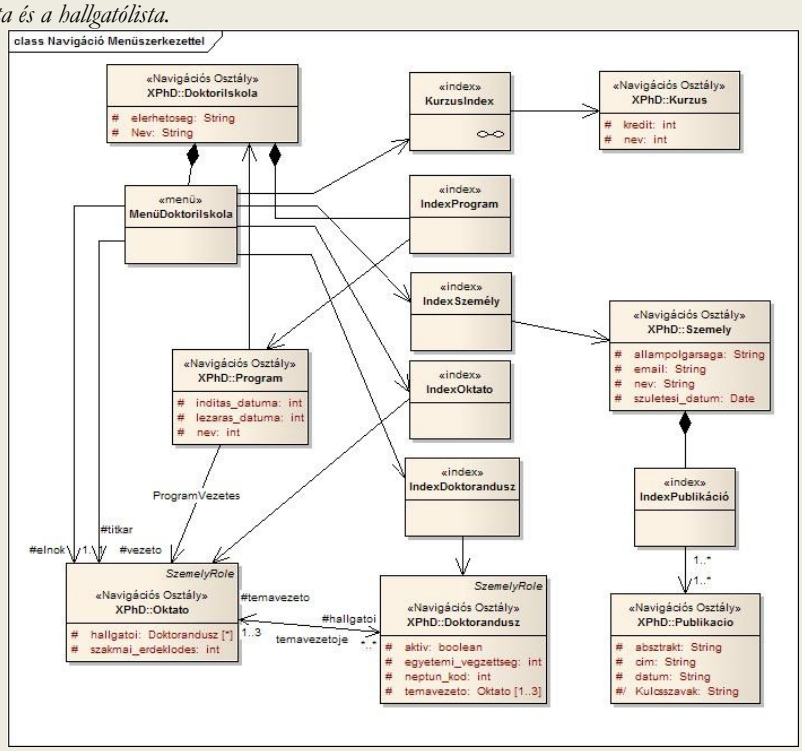
A menüszerkezet előállítása során a következő lépésekre figyeljünk:

- minden kimenő linkkel rendelkező osztály esetén készítsünk menüt,
- a menüt kompozícióval kapcsoljuk hozzá a csomóponthoz,
- szükség esetén a menüt bontsuk fel további almenükre,
- cseréljük le a navigációs osztály kimenő linkjeit a menühöz.

Az alábbiakban pedig a példakánt használt Doktori Iskola navigációs szerkezetének egy részlete látható:

Példa: Doktori Adatbázis – ***Elemzési fázis – Elemzés és tervezés – Navigációs menüterv***

A doktori Iskola főmenüjében szerepeljen közvetlen hivatkozásként a kurzuslista, a személylista, az oktatólista és a hallgatólista.



A navigációs kontextus

Az tervezési folyamat eddigi lépései során elsőként előállítottuk a koncepciós modellt, amely a szakterület szerkezeti felépítését mutatja finom részletességgel. Ez kiemelten fontos, mert a modellezés és a tartomány-specifikus nyelvünk kialakítása során a hangsúlyt az adatorientált feldolgozásra helyeztük és ahhoz, hogy a továbbiakban felhasználható legyen, mint az Információs rendszerünk alapja, szükséges a megfelelő részletezettség. Miután ezzel elkészültünk, kiegészítettük a modellt a navigációs szerepkörökkel, ami az eltérő navigációs utak és eltérő felhasználói szintek miatt szükséges. Ezt folytattunk a navigációs szerkezet meghatározásával, amelynek eredményeképp előálló modell már jól közelíti a webalkalmazásokról alkotott elképzeléseinket.

Ami viszont még hiányérzetet okozhat a megjelenítési szempontok mellett, az annak a kérdése, hogy a navigáció során az egyes csomópontok között végrehajtott lépések alatt mi történik, illetve pontosabban mi változik. Példaként vegyük megint a Doktori Iskolánkat. A nyitóoldalon az alapvető információkon kívül találunk hivatkozásokat az oktatókra, a hallgatókra, a kurzusokra. Innen átléphetünk az index segítségével például az oktatók listájára, majd kiválaszthatjuk az érdeklődésünknek megfelelő személyt. Az eddig lépéseink során nem fogunk meglepődni, hogy mindig egy új oldalt kaptunk, amely a kérdéses információt tartalmazta. Ezután, ha az oktatónk által tartott kurzusokra is kíváncsiak vagyunk, akkor a navigációs modellen látható kurzusindex segítségével azokat meg is tekinthetjük. Egy újabb listát kapunk egy új oldalon. Ezzel eddig rendben is lennénk. Tegyük fel, hogy ennél a pontnál el kell mennünk és valami oknál fogva csak később térünk vissza ehhez az oldalhoz. Az oldalon látni fogunk egy szép listát, ami kurzusneveket tartalmaz. Jogosan merül fel a kérdés: tudjuk-e még, hogy honnan (mely navigációból) származik ez a lista?

A példából érezhető, hogy az egyes csomópontok közötti linkek követése során szükséges lehet olyan – lényeges – információknak a megtartására, amely segítséget nyújthatnak abban, hogy el tudjuk helyezni az aktuális információt a megfelelő környezetben. Hiszen ezen információk nélkül a kapott oldalon található adatok nem feltétlenül értelmezhetőek pontosan.

Ezen szempontok alapján mindenképp szükséges, hogy lehetőséget biztosítsunk az egyes lépések során létrejövő csomópontváltások mellett a navigációs környezetnek (más néven a kontextusnak) a meghatározására is.

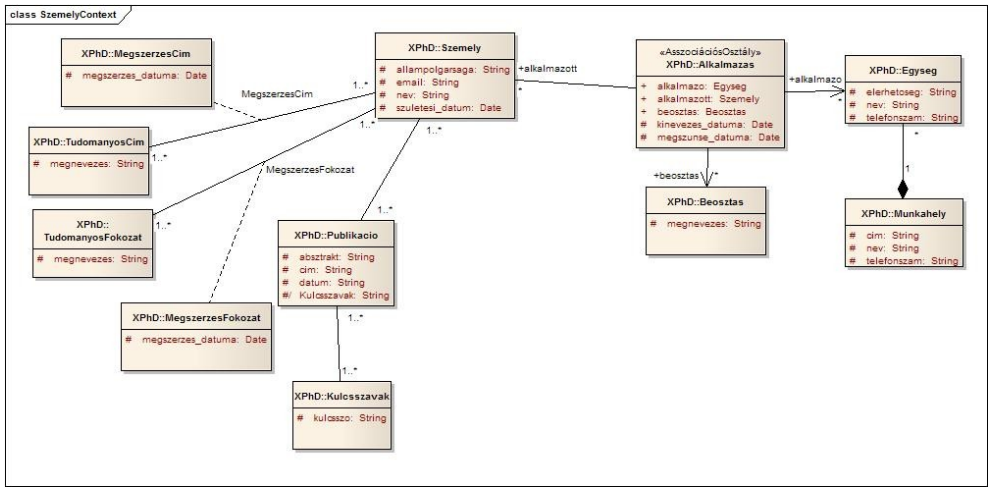
A *navigációs kontextus* kialakításánál többféle szempontot is meghatározhatunk, amely az Információs rendszerek szempontjából elsődlegesen a statikus osztályszerkezeten alapul, illetve az egyes tevékenységek során magának a tevékenységnek a végrehajtáshoz szükséges osztályokat is figyelembe veheti. Adat-orientált szempontból vizsgálva a navigációs kontextus kérdéstét a legkézen-fekvőbb megoldást az összetartozó csomópontok meghatározásához egyértelműen *az asszociációk mentén egymással kapcsolatban álló koncepcionális osztályok jelentik.*

A navigációs kontextus ebből kifolyólag a mi olvasatunkban egy koncepcionális osztály és az ahhoz a koncepcionális diagramon asszociációkkal kapcsolódó osztályok között fennálló viszony. A kontextus meghatározásához azért a koncepcionális modell kerül felhasználásra, mert a navigációs modellben egy adott osztály esetén számos további, kiegészítő link kerülhet felvételre, amelyek az eredeti szakterületi modellen nem szerepeltek. Ezen kiegészítő kapcsolatok a menüszerkezet segítségével természetesen elérhetőek lesznek, de a kontextus pontos meghatározásához nem szükségesek.

A kapcsolat az egyes koncepcionális osztályok között jellemzően nem egy-az-egyhez számosságú, ezért egy adott osztály esetén a kiindulási kontextus információ tartalmának meghatározásához támpontokra lesz szükségünk. Ebben segítségünkre lehetnek az asszociációhoz kapcsolódó asszociációs osztályok, illetve a navigációs modellben elkészült indexek, de ennek pontosítása prezentációs réteg feladata.

Vegyünk példaként a Doktori Iskolával kapcsolatban álló személyeket. A navigáció kontextus definíciója szerint végig kell néznünk a koncepcionális modellben a Személy osztály asszociációit. A kapcsolatok feltérképezése során négy asszociációt találunk. Ezeket az alábbi ábrán áthatjuk.

Példa: Doktori Adatbázis – ***Elemzési fázis – Elemzés és tervezés – Navigációs kontextus***
Személy osztály kontextusa: TudományosCím, TudományosFokozat, Publikáció és Alkalmazás. *<u>Kiegészítés:</u> A szakterület elemzése során azt tapasztaltuk, hogy szükséges lesz nyilvántartani, hogy ki, mikor, hol és milyen beosztásban dolgozott.* *Ezt a legegyszerűbben egy ternális asszociációval tudjuk megoldani, ahol a személy, a beosztás és a munkahely kerül egymással kapcsolatba. Az alkalmazás kezdetének és befejezésének a dátuma pedig egy asszociációs osztály segítségével modellezhető, mert az egyik előbbi osztálynak sem a jellemzője.* *Az eredményül kapott kontextus ábra az alább található képen látható.*



25. ábra: Személy osztály navigációs kontextusa

Az ábrához azonban egy apró kiegészítést kell tennünk, amire a modellező eszközök ternális, illetve annál nagyobb fokú kapcsolatok mostoha kezelése miatt kényszerülünk. A fentebb említett Alkalmazás asszociáció szemléltetését a jelenleg elérhető modellező eszközök nemes egyszerűséggel figyelmen kívül hagyják. Ennek kiküszöbölésére felvettük az *«AsszociációsOsztály»* sztereotípiát, amely az előbb említett kapcsolat leírására szolgál. Ezután a rövid kitérő után pedig lássuk a navigációs kontextust.

A Személy osztály esetében négy asszociációt találunk, amelyek mentén a Személy osztályt, mint kontextust meg kell tartani. Ez azt jelenti, hogy ha egy személy esetén például megtekintjük a publikációit, akkor a navigációs link követésével együtt jár az is, hogy a személyre vonatkozó (prezentációs modell szinten szabályozott) információk megmaradnak. Ez biztosítja a kontextus fenntartását, ami akár mint visszafelé mutató link is megjelenhet, hogy a szabad mozgást lehetővé tegyük magán a kontextuson belül. Érdemes megfigyelni, hogy a Személy az Alkalmazás asszociáción keresztül a Részleg osztállyal áll kapcsolatban, ami viszont a Munkahelyhez kompozíciós kapcsolattal kötődik. Ilyen esetben a kontextus célszerű kiterjeszteni a tartalmazó osztály felé is.

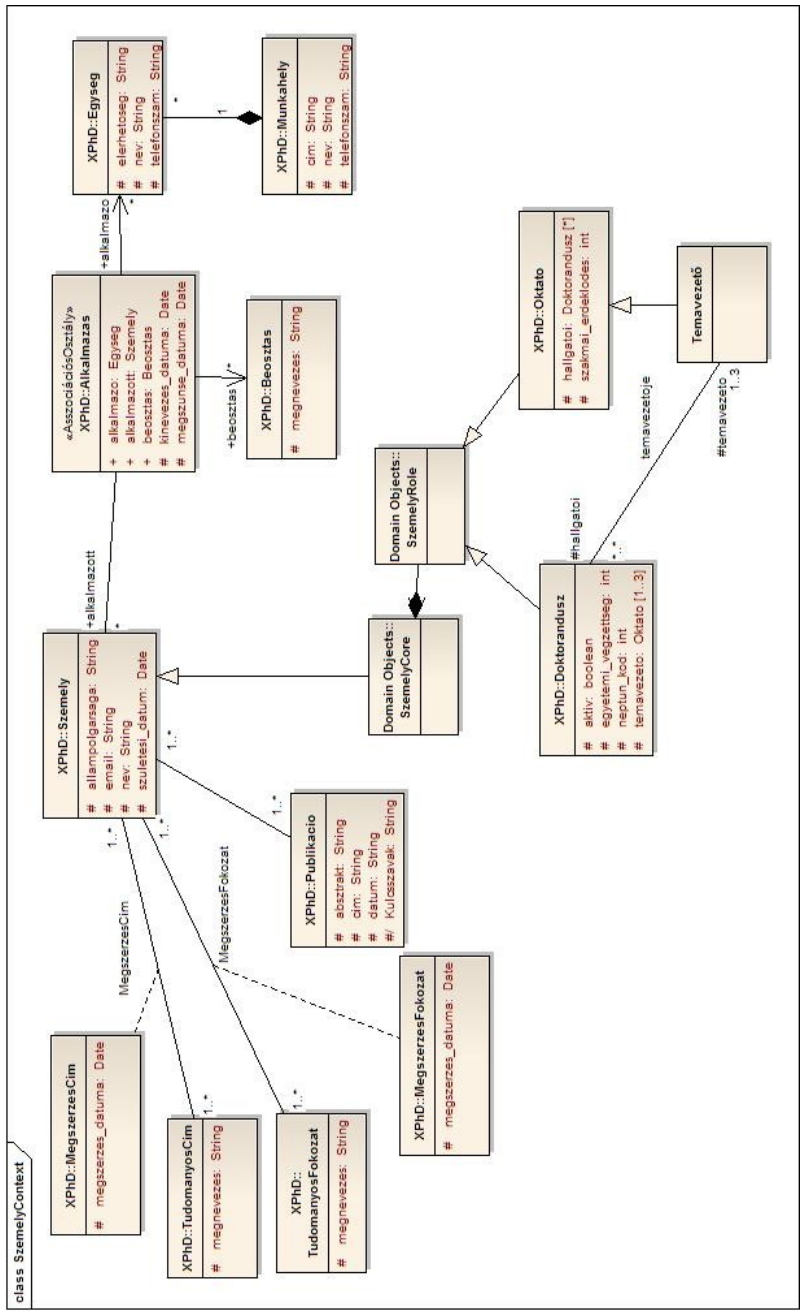
A másik irányból nézve, a kapcsolatban résztvevő osztályok esetén is hasonló állításokat fogalmazhatunk meg. Ha a navigáció feléjük halad tovább, akkor az adott személy kontextusán belül maradunk, ezért bizonyos (a környezet egyértelmű azonosítására alkalmas) információ(k)at meg kell tartaniuk. A mi esetünkben ez

például lehet a név, mint leíró attribútum átvitele, vagy akár a közöttük fennálló asszociáció szerepkörének a megnevezés is. Ennek az eldöntése azonban a már említett prezentációs modell feladata lesz. Hasonlóan, a prezentációs modell lesz a felelős a navigációs linkek elnevezéséért is.

Azonban nem minden koncepcionális osztály válik automatikusan egy navigációs kontextus kiindulási pontjává. Ennek eldöntését a tervezési folyamat során a navigációs szerkezeti diagram kialakítását követően kell elvégezni. A kiválasztott osztályok esetén egy kulcsszavas érték (tagged value) felvételével (pl. *isContextNode*) tudjuk jelezni, hogy navigációs kontextus csomópontról van szó.

Miután nem minden csomópont lesz kontextus kiindulási pont, a kontextus információknak ezért csak addig lesz jelentőségük, amíg egy újabb, saját navigációs kontextussal rendelkező csomóponthoz nem jutunk. Innentől kezdve már ez az újabb csomópont fogja szolgáltatni a kontextus a további navigáció során.

Érdemes még kiemelni a koncepcionális szinten szerepkörökkel rendelkező osztályok különböző szerepköreihez tartozó kontextus meghatározását. Ebben az estben a kapcsolatok feltérképezéséhez az öröklődési hierarchiában alatta és felette elhelyezkedő osztályok, valamint a szerepkör alapjául szolgáló osztály kapcsolatait kell figyelembe venni. Az alább látható ábrán egy oktató esetén a Témavezető, mint szerepkör és a Személy, mint alaposztály is részt vesz a kontextus meghatározásában, azaz a velük kapcsolatban álló osztályokra is kiterjed egy oktató kontextusa.



26. ábra: A Személy osztály kontextusa

Komponens modell

Az eddigi lépések során elértük, hogy az elkészítendő webalkalmazásunknak a lehető legabsztraktabb modelljét hozzuk létre. Kialakítottuk a tartomány specifikus nyelvünk egyed (*entity*) definíciója és a követelményelemzés során feltárt ismérvek alapján a koncepcionális modell osztályszerkezetét, amely figyelembe veszi az egyes egyedek viselkedés alapú megkülönböztetését is. Definiáltuk ezen elemek egymáshoz viszonyított elérhetőségét a navigációs szerkezeti modellel, amely az egyedek természetes szerepkörei mellett figyelembe veszi a navigáció során felmerülő, a navigációhoz kötődő esetleges újabb szerepek kialakulását.

Ami eddig nem került modellezésre, az a tartomány specifikus nyelvünkben megadott modul (*module*) definíció. A modul nem más, mint szorosan összefüggő osztályok (koncepciók) és feladatok együttese. Ennek az összetartozásnak is több fajtája létezik. A *kommunikációs összetartozás* azt fejezi ki, hogy a modul részei ugyanazon az adaton dolgoznak, míg a *funkcionális összetartozás* a modulok együttműködése egy adott cél megvalósításának érdekében. Ami olvasatunkban a kommunikációs összetartozás UML kifejező eszköze a komponens modell lesz. Ebben a modellben már nem vezetünk be újabb struktúrákat, hanem a meglévőket fogjuk össze komponensekbe. Az UML-ben a komponensek igazán univerzális egységek. Egy egységbezárt, önálló, teljes és ezáltal cserélhető egység. Az UML meta-modellben a komponens az osztályok egy alosztálya, tehát rendelkezik azok összes jellemzőjével, különösen kiemelve itt az összetett struktúrákét (*StructuredClassifier*). Azaz a komponenseknek is vannak részeik, attribútumai, metódusai. Ezenfelül rendelkezhet csatlakozókkal (*Port*) és interfészekkel.

A komponensek minden olyan helyen használhatóak, ahol az osztályok szerepelnek. A komponensek közötti összeköttetést a leghatékonyabban az összekötők (*Connector*) segítségével tehetjük meg. A kapcsolatot egymás között az összekötők az interfészek segítségével tartják fent. Ezeket pedig nyújtott és megkövetelt interfészeknek nevezik. A komponens által biztosított szolgáltatásokat a nyújtott interfész tartalmazza, míg a működéséhez szükséges szolgáltatásokat a megkövetelt interfészekkel fejezhetjük ki.

Miután a DSL segítségével leírt szakterületről az entitásokat tartalmazó koncepcionális modell készült el, amely a navigáció modellezése során tovább bővült,

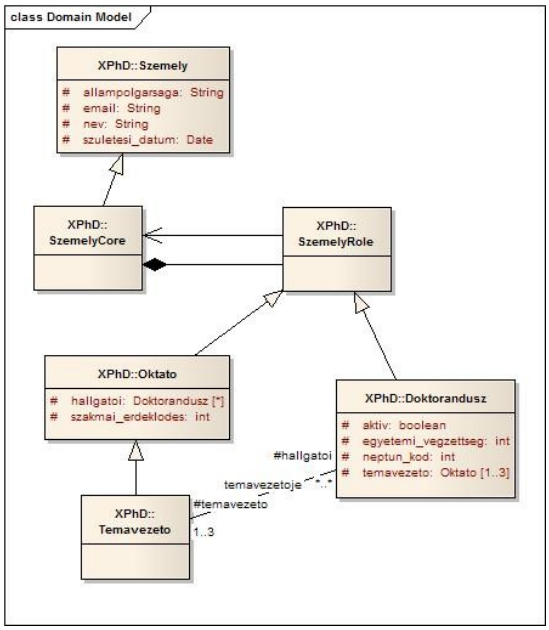
a komponens modellel pedig már ezt a bővített modellt fogjuk a szolgáltatásokhoz szükséges (részletesebb) felületekkel rendelkező komponensekké alakítani.

Nézzük a következő egyszerű személy entitás definícióját:

```
Module személy {
        Service SzemélyService {
          keressSzemélyNév delegates to
SzemélyRepository.keressSzemélyNév;
          # … további szolgáltatások …
        }
        Entity Személy {
          String szem.szám key length="20";
          Integer kor;
          String nev;

        EntityRole Oktató{
             String szakmai_érdeklődés;
        }
        EntityRole Doktorandusz{
             String neptun_kod;
        }
          Repository SzemélyRepository {
              findById;
              List<@Személy> keressSzemélyNév(String név);
              save;
              findByQuery;
              findByExample;
              delete;
          }
        }
    }
```

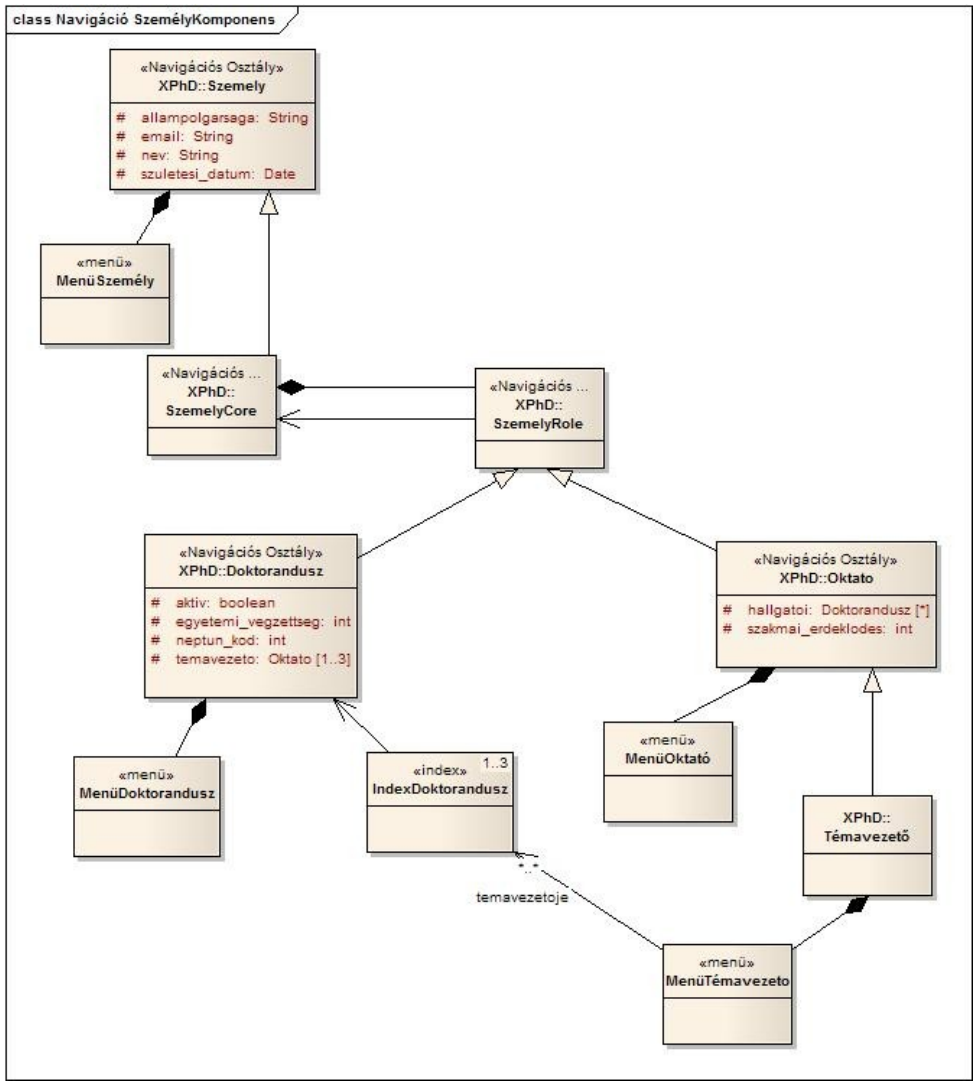
27. ábra: DSL definíciós példa - Személy modul

Ennek megfelelően az első lépésben, a koncepcionális modellben a következő osztályszerkezetet kaptuk:



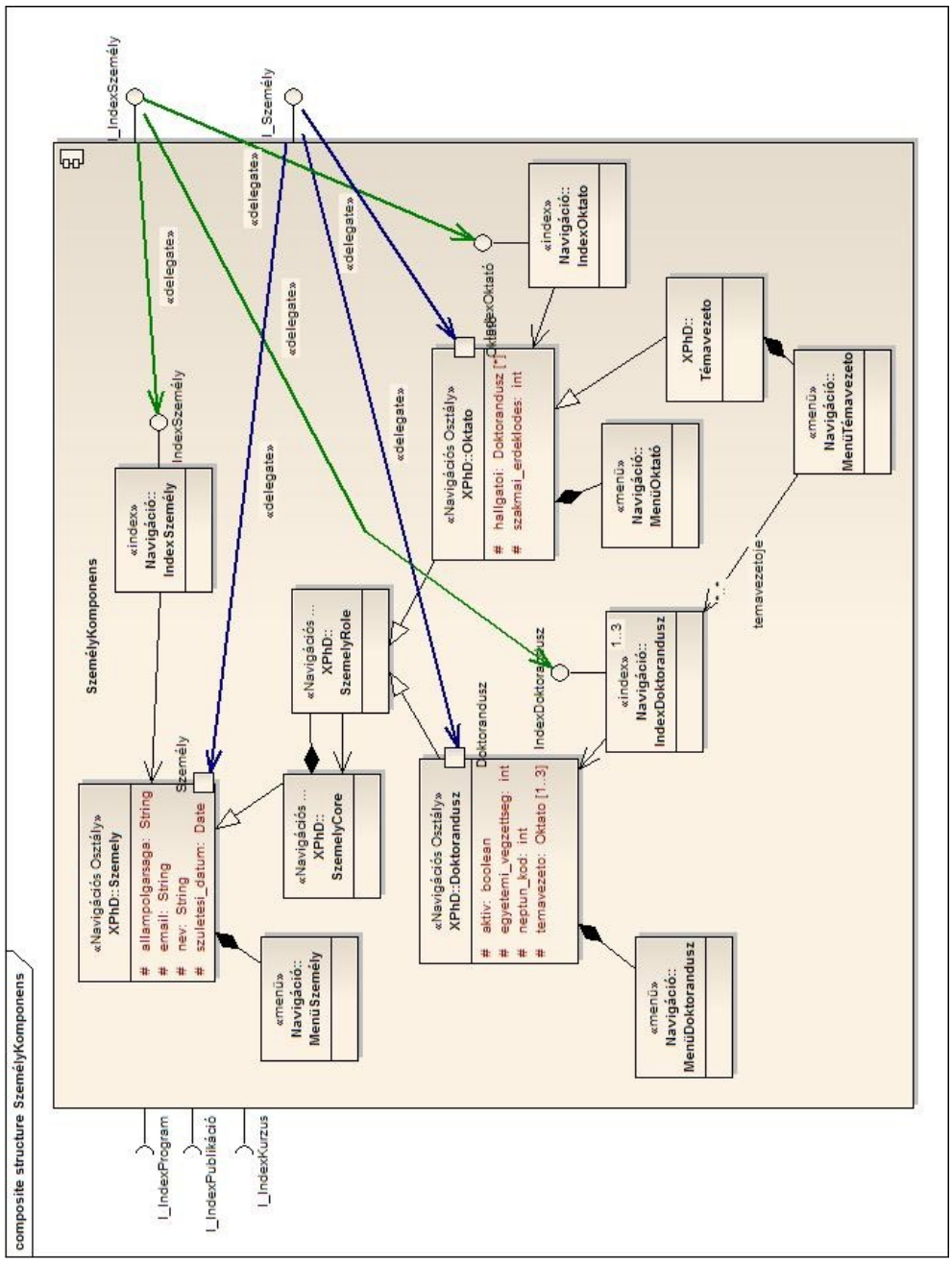
28. ábra: A Személy modulnak megfelelő osztályok

A feladat ebben a modellezési lépésben nyilvánvalóan az, hogy az így előálló osztályszerkezeteket a navigációs struktúrákkal (index, menü), illetve az esetleges navigációs szerepkörökkel bővítve elhelyezzük egy-egy komponensben. A komponens által alapvetően nyújtandó szolgáltatás definíciója megtalálható a DSL alapú szöveges állományunk Szolgáltatás (*Service*) szakaszában, míg a navigáció szempontjából szükséges és megkövetelt felületeket a navigációs diagram és a navigációs kontextus alapján tudjuk meghatározni minden egyes különálló elemnél. A következő ábrán a Személy modulnak komponensé történő alakításához szükséges osztályok teljes listája látható. Miután számos menü szerepel benne, és többféle szerepkör is látszik, a komponensnek egy sereg interfészt kell biztosítania, ami az egyes szerepköröknek megfelelő elérést fogják biztosítani, ezenfelül a menük miatt viszont több interfészt meg is kell követelnie a működéséhez.



29. ábra: Személykomponens tervezet

Ezen információk birtokában már el lehet készíteni a Személy komponenst, amely a következő ábrán látható. Vastagabb vonallal lettek szedve a nyújtott interfészek által biztosított szolgáltatások.



30. ábra: A Személy komponens

Szerepkörök a komponensekben

Amennyiben *egy komponens több szerepkörben is feltűnhet a* ***koncepcionális modell alapján***, akkor ez nem jelenti egyidejűleg annyi különböző interfésznek a megjelenését, hanem az általános *entitás által nyújtott interfészt kell bővíteni* a megfelelő viselkedéshez szükséges metódusokkal. A kapcsolódó komponensek pedig a – korábbi fejezetekben bemutatott – metamodellben meghatározott *hasRole()* és *getRole()* metódusok segítségével tudják lekérdezni a szerepköröket, illetve egy adott szerepkörhöz tartozó példányt elérni/lekérni.

A komponensen belül ilyenkor több delegáló összeköttetést használunk, amelyek továbbítják a hívást a megfelelő résznek. Ezzel is utalva arra, hogy az összkomponens szolgáltatását végeredményben a részek szolgáltatásai teszik ki. És ez fordítva is igaz. Lehetővé teszik a részek számára, hogy a hívásokat az összkomponens hívásaiként tüntessék fel. Mindkét esetben a küldő, illetve a fogadó rész felelős a hívás kezeléséért, nem pedig az összkomponens. A delegáló összekötőknek nincs saját viselkedésük, csupán egy „hosszabbító zsinórként" funkcionálnak.

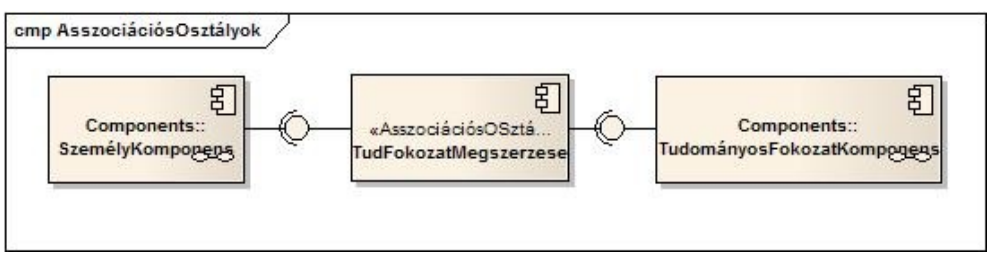
Ugyan ez vonatkozik arra az esetre is, ha egy *koncepcionális osztály még nem rendelkezik szerepkörökkel, de a* ***navigációs modell kialakítása során*** *erre igény lép fel.* A navigációs osztályok pontosításával kapott új osztályok és a hozzájuk kapcsolódó viselkedésnek a komponens által történő megvalósításához szintén *az általános – koncepcionális modellből származó – entitás nyújtott interfészét szükséges bővíteni*. Ebben az esetben is a kapcsolódó komponensek a működésükhöz az eltérő viselkedést a fentebb említett módszer segítségével tudják felhasználni. A mi esetünkben ilyen magatartással találkozhattunk a Kurzus osztály esetében.

Ami viszont eltérés a koncepcionális szinten szerepkörökkel rendelkező osztályok és a navigáció szintjén szerepkörökkel rendelkező osztályok között, hogy míg a koncepcionális szerepköröket az *Object Role* tervezési minta alapján modellezzük, addig a navigációs szerepköröknél erre a származtatást használtunk. Miután az Object Role lehetővé teszi az alaposztállyal történő kapcsolattartást mindkét irányban, addig a származtatás már nem. A korábbi szakaszokban ezzel már foglalkoztunk részletesebben. Ebből következően, *a navigációs szerepkörökre úgy tekinthetünk, mint dinamikus nézetekre egy adott osztály (komponens) esetén*. Ezért a megvalósítása során célszerű az Építő (*Builder*) tervezési mintát alkalmazni, amely lehetővé teszi, hogy a különböző szerepkörök különböző módon jelenjenek meg.

Asszociációs osztályok

A szerepkörök vizsgálata mellett fontos kiemelni a háttérbe szorult asszociációs osztályok jelentőségét is. Az asszociációs osztályokat azzal a céllal hoztuk létre, hogy olyan információkat hordozzanak, amely magának a két (vagy több) osztálynak egymással kialakított kapcsolatának a jellemzőit tartalmazza. Gondoljuk vissza a már említett alkalmazási információk, vagy éppen a tudományos fokozatok megszerzésére vonatkozó információkra. Ezek olyan jellegű adatok, amely egyik osztálynál sem illenének a leíró attribútumok közé. A koncepcionális tervezést követően a navigációnál már le is hagytuk az ábrákról, mert a navigáció jellegét nem befolyásolják. Azonban a megjelenítéshez már szorosan fognak kapcsolódni, mert a ***segítségükkel oldható meg, hogy egy adott nézetben az osztályok közötti kapcsolatot jellemző információk is megjelenjenek***.

Ennek érdekében az asszociációs osztályokat, mint összekötőket (összekötő komponensként) kell elképzelni a kapcsolatban résztvevő osztályoknak megfelelő komponensek között. Ezen osztályokat megvalósító komponensek lesznek ***felelősek a kapcsolattartásért***, miután *a hívások rajtuk keresztül fognak haladni*. A hívások helyes kezelésére olyan interfészeket kell szolgáltatniuk, mint amilyet az üzenetet küldő komponens megkövetel, hogy a kapcsolatban résztvevő többi komponenssel együtt el tudják látni a feladatukat. Talán úgy lehetne még leírni ezt a viszonyt, mint az adatmodellezés kapcsán felmerülő sok-a-sokhoz kapcsolat relációs modellre történő átültetésekor létrejövő kapcsoló táblák.



31. ábra: Asszociációs osztályok a komponensek között

Megjelenítési modell

A webalkalmazásunk tervezése során mindig szem előtt tartottuk, hogy a fejlesztés egyes lépéseinél mindig a lehető nagyobb mértékben próbáljuk meg függetleníteni az egyes szakaszokat mind egymástól, mind pedig az adott lépésben leválasztható szempontoktól. Ennek eredményeképp a koncepcionális modellünk átalakult egy komponens modellé, amelyben az összetartozó entitásaink egy független komponensként funkcionálnak. Ezzel a módszerrel lehetővé tesszük, hogy amikor információt szeretnénk kinyerni a rendszerből, akkor már csak a komponensek megfelelő összekapcsolásával, egy jól definiálható lépés sorozat mentén elérjük a célunkat.

Ez természetesen azt jelenti, hogy a felhasználó felület megtervezését egy absztrakt felhasználói felületként kell elképzelni, amely csak azt mondja meg, hogy milyen információkat szeretnénk látni. Ezen információk pedig az előbb említett komponens összekapcsolásokkal kifejezhetőek. Ennek az a feltétele, hogy az előző lépésben előálló komponensek tudják szolgáltatni a megfelelő részletességű információkat. De ha visszagondolunk, akkor a komponensek felületének tervezésénél pont ezeket a szempontokat vettük figyelembe. Mindezek mellett a komponensek biztosítják azokat a szolgáltatásokat is, amelyeket a DSL részben szolgáltatásként jelöltünk meg.

A megjelenítendő információ mennyisége és módja a komponens modellen felül a követelményelemzés fázisában felállított elvárásoktól is függ. Ezek többnyire már inkább csak a megjelenítés módját és részletességét taglalják. A mi modellünk esetében viszont a cél az, hogy ezeket az aspektusokat valamelyest leválasszuk, és olyan módon készítsük el a szükséges információkat tartalmazó adathalmazt, hogy azt a kliens oldali megjelenítőre bízzuk – esetleg még a kiszolgáló oldalon egy utolsó transzformációs lépésben átalakítjuk a kliens nyelvére. Gondoljunk itt a mára elterjedté vált mobileszközökre, okos telefonokra, PDA-kra. Ezen eszközök megjelenítési képességeik valamennyire korlátozottabbak, mint egy asztali számítógép böngészője, de nem elképzeletlen, hogy egy ilyen eszköz segítségével szeretnénk a rendszerrel kapcsolatot teremteni. Míg az számítógépes böngészők összetett feldolgozási képességgel rendelkeznek, addig ezek a kisebb eszközök már nem. Éppen ezért fontos, hogy a megjelenítés során csak egy absztrakt felületet használjunk annak érdekében, hogy az egyes nézetekben szükséges információhalmazt kialakítsuk.

A komponensek talán úgy a legegyszerűbb elképzelni, mint apró kis oldal darabkák, amelyek tudják szolgáltatni a szükséges információkat, legyen az a menüszerkezet, egy indexoldal, vagy éppen a tényleges tartalom. Ezen ki darabkákból össze tudunk építeni nagyobbakat, azokat meg nagyobbakba, mindaddig, amíg el nem érjük a megjeleníteni kívánt információtartalmat.

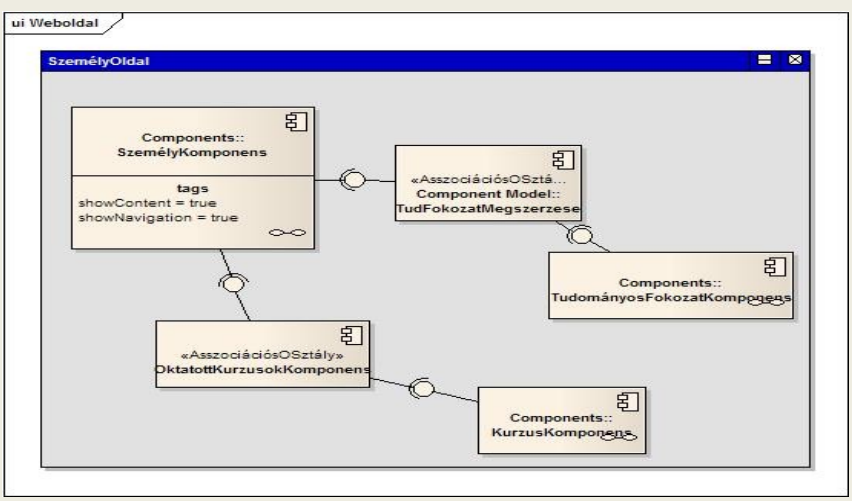
A megjelenítés során azonban figyelembe kell venni az olyan szempontokat is, mint például a navigációs szerkezet kialakításánál az indexelem kompozícióval történő hozzácsatolása (23. ábra: Indexelem kompozícióval), vagy a navigációs kontextus meghatározása. Ezen információkat azonban a komponenseink a megfelelően kialakított interfészek segítségével egymás között tudják egyeztetni.

A módszertanunk által így annak a lehetőségét biztosítjuk, hogy egy adott, a követelményelemzés fázisában kialakított nézethez szükséges információtartalmat elő tudjuk állítani absztrakt módón. A modellezés során nem térünk ki a dinamikus viselkedés leírására, azaz mi történjen gombnyomásra, beágyazott média elindításakor, vagy éppen egy form kitöltésekor. Ezek leírására további UML diagramok felvételére lenne szükség, mint például állapotátmenet diagram, vagy interakció diagram, amelyek túlmutatnak a felhasználói felület absztrakt leírásán.

Példa: Doktori Adatbázis – ***Elemzési fázis – Elemzés és tervezés – Prezentációs modell***

A Személyek oldalán a következő információk legyenek elérhetőek:

- *részletes adatok, navigációs szerkezet, oktatott kurzusok.*



32. ábra: Absztrakt megjelenítési modell

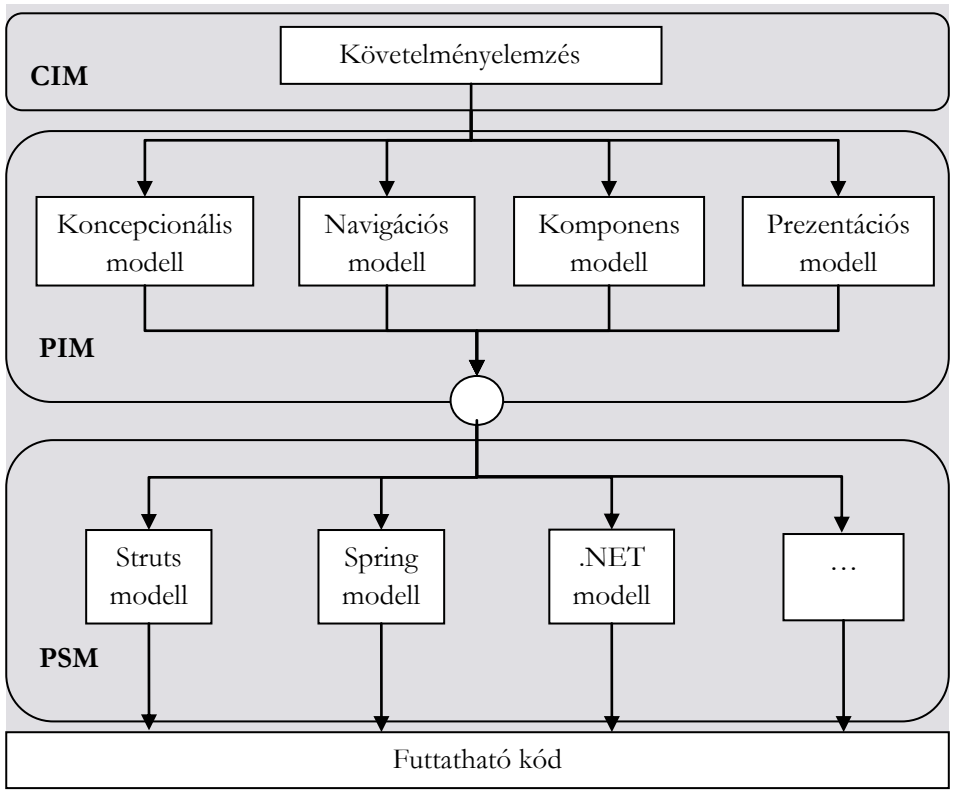
A modellek életre keltése

A fejlesztési folyamat során előállított modellek hasznos segítséget nyújtanak a szakterület átlátásában. Mind koncepcionális, mind navigációs, mind pedig komponens szinten átláthatjuk a modellezendő területet. A modellek ezen felül még komplexebb támogatást nyújthatnak, ha képesek vagyunk belőlük a tervezett webalkalmazásnak egy működő prototípusát előállítani. A kódgenerálás során különféle szempontokat is figyelembe vehetünk, amely nem csak egy váz generálását biztosíthatja, hanem lehetővé teszi annak bizonyos szintű használatát, vagy akár különféle külső csatlakozási felületek automatikus létrehozását.

Az MDA szemléletmódját követve ez a lépés a platform független modellek (*PIM*) platform specifikus modellekre (*PSM*) történő leképzésének a programkódban történő szerializációja. A platform független modellek leképzése több fázisban történik, annak megfelelően, hogy a webalkalmazás koncepcionális, navigációs esetleg prezentációs szempontjait vizsgáljuk. Az utóbbi kettőt azonban a komponens modell bevezetésével összefoghatjuk egyetlen transzformációvá. Minden egyes transzformációnak az a célja, hogy egy specifikus, kezeléséért felelős webplatform (vagy technológia) számára elérhetővé tegye. Adott platformok esetében az egyes részek akár ki is cserélhetőek más technológián alapuló részekre, anélkül, hogy a működését befolyásolná. Ilyen esetekben csak az adott rész áttranszformációjáért felelős modellrészt kell cserélni.

A teljes fejlesztési folyamat mindig a követelmények elemzésével kezdődik, amely többnyire a funkcionális követelményeket, az aktorokat és az általuk elvégzendő feladatokat tartalmazza – többnyire használati esetek formájában megfogalmazva. Ezt a részt tekinthetjük a CIM (*Computational Independent Model*) elkészítésének. Ebből képezzük le a koncepcionális modellünket, amely a szakterület szempontjából meghatározó összefüggéseket leírja. Ezt tekinthetjük már egy platform független modellnek (*PIM*), mert semmilyen specifikus technológiát nem vesz figyelembe a modellezés során. Az ebből származtatott navigációs és komponens modell szintén PIM modelleknek tekinthető, mert továbbra sem alkalmazunk akár technológia, akár implementációs szempontokat figyelembe vevő lépéseket. Azonban *a fejlesztés ezen szakaszában rendelkezésre álló modellek továbbtranszformálása már pontosan azt a célt fogja szolgálni, hogy a modelljeinket egy adott platform számára értelmezhetővé tegyük.* Ez az a pont,

amikor a modellek kiegészülnek a speciális technológiai és implementációs részletekkel. A folyamatot a következőképpen képzelhetjük el:



33. ábra: MDA fejlesztési lépések

A *platformot* úgy kell elképzelnünk, mint egy környezetet, amely a hozzá készült alkalmazások futtatását teszi lehetővé. A platformokkal ma leginkább keretrendszerek képében találkozunk, amelyek gyakran más platformokra épülnek. A web platformok pedig kimondottan a webalkalmazások futtatását teszik lehetővé, amelyekre gyakran (web)konténer néven is hivatkoznak.

Azonban a világháló kialakulásának kezdetén a webalkalmazások leggyakrabban egyedileg készültek, közvetlenül megvalósítva az összes szükséges protokollt. A dolgozatban példaként szereplő Doktori Adatbázis első verziója is még ebben a szemléletmódban készült Perl nyelven íródott szkriptek segítségével.

Manapság viszont már a fejlesztések mindig egy adott típusú platformra készülnek. Ezen platformokból jelenleg nagyon sok létezik és a fejlesztő feladata ebből kiválasztani a projekthez leginkább megfelelőt. A legtöbb egy specifikus programnyelv vagy virtuális gép köré épül, mint például a J2EE a Java nyelv köré, míg az ASP.NET a .Net platformra.

Ezen platformok is több osztályba sorolhatóak jellegüket tekintve. Az *első* osztályba azok tartoznak, amelyek átfogó, nagy rendszerként működve minden szükséges eszközt egyben tartalmaznak. Ilyen például az ASP.NET, amely ráadásul még operációs rendszerhez és webszerverhez is kötött. A *második* kategória jelenti a „könnyűsúlyú", többnyire nyílt forrású alkalmazás szervereket, amilyen például a Tomcat és a ráépülő keretrendszerek, mint a Struts, Spring vagy Cocoon. A J2EE platform egy átfogó kibővítése a Java Servlet/JSP technológiáknak, amely az ASP.NET-re adott válasznak tekinthető leginkább. A *harmadik* kategóriába az olyan platformok és keretrendszerek tartoznak, amelyek a gyors fejlesztés és az agilis programfejlesztési módszertant támogatják, mint például a Ruby on Rails.

A *mi célunk* azonban nem ezen keretrendszerek – és a hozzátartozó komponensek a modell vezérelt fejlesztési technikájának a támogatása, hanem egy *olyan koncepciónak a kidolgozása, amely a későbbiekben bármely platformra átültethető a szükséges transzformátorok elkészítésével.* Ebből a szempontból a céljainkhoz sokkal közelebb állnak a „könnyűsúlyú" alkalmazásszerverek, amelyek különböző komponensek és szolgáltatások használatát teszik lehetővé.

A továbbiakban egy olyan platformot fogunk használni, amely a dolgozat elején már bemutatott Model/View/Controller (MVC) tervezési mintán alapul, melyben a webalkalmazás értelmezése megfelel *a modell (tartalom), nézet (prezentáció) és vezérlő (navigáció és feldolgozás)* szabályainak. Ezzel lehetővé válik, hogy az egyes platform specifikus modelleknek megfelelő transzformációk elkülöníthetőek legyenek egymástól. Egy konkrét modellezési technikához (pl. JavaBeans) vagy megjelenítési technológiához (pl. Java Server Pages) így csak a megfelelő transzformációkat kell elkészíteni. Ezáltal ***biztosíthatjuk***, hogy például ***a modell résznek megfelelő technológia teljesen független a megjelenítésért felelős technológiától,*** így bármilyen más nézet is előállítható.

Alkalmazott keretrendszer

Spring

A keretrendszer, amely megfelel ezen elvárásoknak, a Spring. A Spring egy általános célú keretrendszer, amely a Java platformon alapul. Annak ellenére, hogy nagyon összetett, kompakt webes támogatással rendelkezik, lehetőséget ad a saját környezetünk kialakítására is. Számos integrációs eszközzel rendelkezik a különböző technológia területekhez, mint például a perzisztencia vagy a tranzakciókezelés. A moduláris architektúrája pedig biztosítja a bővíthetőségét. A következő modulokat tartalmazza:

- Web keretrendszer
- Beans
- Köztes rétegek támogatása (CORBA, SOAP, webszolgáltatások)
- Direct Access Objects (DAO, adatbázis absztrakciós réteg)
- Objektumrelációs leképzés (ORM, pl. JDO, Hibernate)
- Tranzakciókezelés
- Aspektusorientált programozás (AOP, pl. AspectJ)

A Spring keretrendszer a már említett MVC tervezési mintán [30] alapul. A modell rész fogja magába zárni a szükséges alkalmazás adatokat és ezek funkcionalitását, a nézet fogja megjeleníteni a modelltől származó adatokat, a vezérlő pedig fogadja a felhasználói kéréseket, elvégzi a modell változtatásait és frissíti a nézetet. A webalkalmazások fejlesztésénél azonban az eredeti MVC modellnek egy kötöttebb változatát használjuk, amelyet a Sun MVC2-nek nevezett el [31] és a http protokoll által meghatározott kérés/válasz alapú működésnek megfelelően lett átalakítva. Ennek értelmében a nézet kizárólag felhasználói kérések esetén frissül, nincs lehetőség arra, hogy a modell előidézhesse a nézet frissítését.
Működését tekintve a következő lépések zajlanak le egy kérés esetén. A felhasználó a webalkalmazással a böngészőjében megadott URL lekérésével léphet kapcsolatban. Ekkor a böngésző egy http kérést indít a kiszolgáló felé, ami az esetünkben lehet egy Tomcat alkalmazásszerver. A szerveroldalon létrejön egy objektum, ami a kérést tartalmazza, majd a Tomcat ezt a kérést átadja a sajátmagunk által létrehozott

feldolgozónak, amely majd egy válaszobjektum létrehozásával reagál. Ezt az objektumot kell majd a nézet résznek megjelenítenie, ami többnyire egy új weboldal létrehozását eredményezi. Ahhoz, hogy ezt az egyszerű folyamatot átlássuk, nézzük meg az egyes modulok működését.

Modell: A koncepcionális modellünk minden nehézség nélkül átalakítható a kiválasztott technológia platformnak megfelelő platform specifikus implementációra. Nem szükséges semmilyen speciális ősosztály vagy interfész megvalósítása sem, mert a modell részben bármilyen Java objektum (POJO – Plain Old Java Object) használható. A nézet, illetve a vezérlő rész a modell állapotához csak a megfelelő *get-* és *set-* metódusokon keresztül férhet hozzá. Ennek a követelménynek pedig számos specifikus technológia eleget tesz, ezért itt nagy szabadsági okkal lehet a későbbiekben is cseréket végezni. Legrosszabb esetben is un. közvetítő osztályokat használhatunk (illetve generáltathatunk), amelyek általában az objektumok adatbázisban történő perzisztens tárolásához szükségesek. Ezt legegyszerűbben úgy érhetjük el, ha felhasználunk a már kidolgozott Enterprise Java Bean-ek (*EJB*) adatbázisra történő leképzését, mondjuk a Hibernate keretrendszer segítségével.

Nézet: A Spring keretrendszer lehetővé teszi, hogy a vezérlőről elválasszuk a megjelenítésért felelős részt, ezzel tetszőleges technológiát alkalmazhatunk a megjelenítéshez. A modell pedig semmilyen módon sem függ a nézettől. Ezek alapján a következő technológiákat alkalmazhatjuk a megjelenítéshez (amelyek akár még kombinálhatóak is):

- Java Servep Pages (JSP)
- Tiles (Struts)
- Velocity és Freemaker
- XML és XSLT
- Jasper jelentéskészítő
- Portletek
- Java Server Faces

Az egyes nézetek a vezérlő szempontjából csak egy névvel azonosítható hivatkozás, amelyből majd a keretrendszer készíti el a konkrét weboldalakat. A navigációhoz pedig arra lesz szükségünk, hogy az alkalmazás szerver konfigurációjában beállítsuk, hogy az adott oldalakat a saját feldolgozónkon továbbítsa, amelyet általában a webcímekhez adott saját kiterjesztéssel érünk el.

Vezérlő: A Spring Keretrendszerbe a vezérlő egy Java osztály, amely a *Controller* interfészt implementálja. A keretrendszer a kéréseket egy *doService* elnevezésű metódus segítségével adja át a megfelelő feldolgozóknak. Ennek megfelelően a vezérlőnknek rendelkeznie kell egy *handleRequest* metódussal, hogy a keretrendszer felől érkező kéréseket fogadni tudjuk. A metódusnak egy előre megadott, *ModelAndView* típusú objektumot kell visszaadnia, amely egy segítőosztály, hogy mind a modell, mind a nézet példányt egyben tudjuk visszaadni. A modell ebben az esetben azon adatokat jelenti, amelyek az adott weboldal megjelenítéséhez szükségesek. A mi modellezési technikánk esetében ez a komponens modellnél kifejtett elképzelésnek megfelelően állítható össze.
Ennek megvalósításához először a modellt kell létrehozni a keretrendszer elvárásainak megfelelően. Ez a nézetnek megfelelően az általunk modellezett komponensek megfelelő metódusaink a meghívásával érhető el. Miután megvan mind a modell, mind a nézet (az absztrakt oldalunk), a keretrendszer meghívja ennek a segítőosztálynak a *render* metódusát a weboldal elkészítéséhez. Ezen a metódushíváson belül történik a hívás tovább delegálása a konkrét nézet implementációhoz. Ez a nézet megkapja a modellt és előállítja a kiválasztott technológiával a megjelenítendő oldalt.

Azonban ahhoz, hogy ez a rendszer működjön szükséges a keretrendszer saját alkalmazásunkhoz történő konfigurálása. Láttuk, hogy nagyon egyszerűen lehet a rendszerhez csatlakozni, csak *a vezérlés átadását kell kérni a saját vezérlő objektumunkhoz*, amely tetszőleges típusú objektum (POJO) lehet. Ezt a vezérlőt a keretrendszer automatikusan példányosítja, de ehhez ismernie kell a szerkezetét. Ezt egy egyszerű XML konfigurációs fájllal tehetjük meg, amelynek helyes szerkezetét egy DTD sémával szembeni validációval éri el a rendszer. A következő ábra azt mutatja be, hogy miképp tudjuk a Spring futtatókörnyezetet a saját igényeik szerint kialakítani.

```
<beans>
     <import resource="content-conf.xml" />
     <import resource="navigation-conf.xml" />

   <bean id="urlMapping" class="org.springframework.web.servlet.handler.
                              SimpleUrlHandlerMapping">
     <property name="mappings">
      <props>
       <prop key="/*.do">mainController</prop>
      </props>
     </property>
   </bean>

   <bean id="viewResolver" class="org.springframework.web.servlet.view
                                    .InternalResourceViewResolver">
     <property name="viewClass">
      <value>org.springframework.web.servlet.view.JstlView</value>
     </property>
     <property name="prefix"><value>/WEB-INF/jsp/</value></property>
     <property name="suffix"><value>.jsp</value></property>
   </bean>

   <bean id="mainController" class="myApp.runtime.MainController">
         <property name="rootObject">
           <ref bean="rootObject"/>
     </property>
     <property name="navigationClassInfos">
      <bean id="navigationClassInfos.list"  class="org.springframework.
                       beans.factory.config.PropertyPathFactoryBean"/>
     </property>
   </bean>
   </beans>
```

34. ábra: Spring konfigurációs fájl

A konfigurációs fájlunk alapján a Spring az összes *.do* kiterjesztésű laphivatkozást átadja a saját *mainController* osztályunknak. A megjelenítéshez a Java Server Pages technológia lett beállítva, míg a navigáció kezeléséért a *navigationClassInfos* osztály fog felelni. Ezt azért fontos kiemelni, mert a célunk a webalkalmazásunk egy működő prototípusának az automatikus generálása, amelyhez a navigációs modellünket természetesen fel tudjuk használni.

Ahhoz, hogy ez az egész összeálljon egy működő rendszerré szükséges, hogy átlássuk, hogy a rendszertől mit várunk el, illetve, hogy az egyes modulokhoz milyen kódot, esetleg kódrészletet tudunk előállítani. Ehhez tekintsünk vissza a célunkhoz, ami web alapú információs rendszerek modellezése. Ezen belül is a hangsúlyt az adattartalom szolgáltatására és kezelésére fektettük. Ennek hatékony megvalósításához a következő részben bemutatandó, *általunk kidolgozott* XML alapú kommunikáció segíthet.

Az XML alapú kommunikáció megtervezése

Az adatok leírására napjainkba az eszközfüggetlenség és a hordozhatóság jegyében az XML formátumot használjuk. *A rendszerünket úgy képzelhetjük el, mint amely képes a kéréseknek megfelelő XML tartalmat előállítani és szükség esetén a legkülönbözőbb formátumokban megjeleníteni is.* Természetesen ennél tovább is mehetünk, mert a leghatékonyabb akkor lesz a rendszer, ha ezen adatokat képes kezelni is.

Ennek érdekében az alkalmazásunkat fel kell készíteni a XML tartalmak előállítására, és azok fogadására is. Az első esetben, a tartalom előállításához a komponens modell során kialakult komponenseinket készítjük fel az XML adatok küldésére.
Miután az XML nyelv saját nyelvtanok kialakítását teszi lehetővé, szabad kezet kaptunk ahhoz, hogy a tartalmat a saját igényeinkhez igazodva állíthassuk elő. Amennyiben ezen adattartalomhoz még a validáláshoz szükséges sémákat is előállítjuk, akkor a külső forrásokból érkező (gondoljunk például egy webszolgáltatásól származó) adatok feldolgozását is el tudjuk végezni. A mi szempontunkból *ez tovább erősíti a tartalom és a megjelenítés egymástól történő különválasztásának a fontosságát.*

A modellezés során előállított különböző modellek eltérő alkalmazási lehetőségeket kínálnak. A koncepcionális modell például megfelelő ahhoz, hogy magát az adattartalmat elérhetővé tegyük. Azonban itt nem csak az elérhetőséget biztosíthatjuk, hanem a feldolgozást is. *A komponensek* az adatokat tetszőleges belső formátumban tárolhatják, de *a külvilág felé mindig XML dokumentumokkal fognak válaszolni.* Ennek létrehozásához az általunk kialakított XML szerkezetet fogják használni. Ha már ez az irány adott, akkor a létező XML dokumentumokból történő adatfeldolgozás már nem jelent extra megterhelést. Hiszen ebben az esetben az adatoknak csupán egy más formátumban történő megjelenéséről van szó, amelyet most az előbbi lépéssel ellentétes irányú átalakítást igényel, miután a szerkezetet ismerjük.

Amennyiben a tartalmat nem közvetlen feldolgozásra akarjuk használni, hanem az alkalmazáson belüli munkafolyamatokhoz, akkor szükségessé válik a navigációs szerkezeteknek a hozzácsatolása is. Ebben a navigációs modell nyújt segítséget,

hiszen a navigációs utakat ott határoztuk meg. Ennek eredményeképp egy újabb, de már bővebb XML dokumentumot kapunk, amely már alkalmas a megfelelő nézet segítségével a felhasználó számára megjeleníteni a kérdéses tartalmat.

Ennek eléréséhez XML transzformációra van szükségünk, amely a navigációs struktúrát is tartalmazó fájlunkat lefordítja a kliens által használt értelmezhető formátumra. A Spring keretrendszer esetében a nézet modulnál láthattuk, hogy alkalmas az XML dokumentumok XSLT stíluslapok segítségével történő átalakítására. Miután a hangsúly számunkra elsősorban az adattartalom szolgáltatásán van, a megfelelő nézet kialakítását bármikor egy adott rendszer igényeihez lehet hangolni (gondoljunk vissza a korábban említett mobil eszközök támogatására).

Miután az alkalmazáson belül is XML kommunikációt szeretnénk megvalósítani, a platform specifikus modellek előállítása mellett ezen modelleket szükséges, hogy kiegészítsük az XML adatok küldésére és fogadására is. A modell réteg esetén ez lehetőséget a maximális függetlenség elérésére az összes többi rétegtől. Szemléletesen szólva, bármely másik réteget anélkül tudjuk kicserélni, hogy az a modell szempontjából semmilyen változtatást nem igényel. Bárhonnan származó érkező (a megfelelő formátumban lévő) inputot elfogad.
Ha a vezérlő szempontjából nézzük, akkor a modelltől érkező XML válasz hasonló lehetőségeket rejt magában. Azaz a modellt megvalósító platform specifikus részt tudjuk bárminemű kockázat nélkül kicserélni egy másikra, ráadásul zökkenőmentesen. A legfontosabb, hogy a kommunikációhoz használt felület változatlan maradjon. ***A komponens alapú modellezésünk*** ezért ***erre rendkívül megfelelő lehetőségeket nyújt***. A vezérlő mindig csak a koncepcionális modellből származtatható adatokra és a navigációt lehetővé tevő navigációs struktúrára fog támaszkodni.

Ennek a szemléletmódnak a megvalósításához szükséges, hogy a *modelljeinket át tudjuk alakítani a számunkra megfelelő formátumokra.* Ehhez első lépésben az UML modellek transzformációjára van szükség, amelyhez az Object Management Group (*OMG*) által létrehozott XML Metadata Interchange (*XMI*) formátumot fogjuk alkalmazni. Az XMI formátum teszi lehetővé, hogy a modelljeinket egy szabványos nyelven kifejezhessük. Ezen felül alkalmas arra is, hogy a fejlesztés egyes szakaszaiban az egyes fejlesztők által használt eltérő modellező eszközök között átvigyük/átadjuk a modelljeinket.

Az XML struktúra kialakítása

Az eddigiekben áttekintettük, hogy az XML alapú kommunikáció és egy XML struktúra kialakítása hatékonyan segíthet a lehető legnagyobb fokú absztrakciós szint elérésében. A modelljeink szolgáltatják ehhez a megfelelő támogatást, hiszen a kialakításuk során mindig arra törekedtünk, hogy az adott lépésben csak a fontos szempontokat vegyük figyelembe.

Az XML alapú kommunikációnk megvalósításához a koncepcionális modellünkhöz kell visszanyúlni, miután ebben a modellben foglaltuk össze a vizsgált szakterület alapvető és meghatározó összefüggéseit. Ha pedig a Spring rendszerre gondolunk, akkor az MVC mintának a modell részéhez kapcsolódunk. Az előző részben kifejtettük, hogy miért is hasznos egy saját XML struktúra. Most ennek a struktúrának a kialakításához tekintjük át a szükséges koncepciókat.

A modellezés során a célunk egy adott területet a lehető legátfogóbban, a felesleges részletektől mentesen leírni. Ehhez használjuk az absztrakciót, mint eszközt, most pedig ennek az absztrakciónak az eredményét szeretnénk leírni úgy, hogy azt a programjaink is fel tudják használni. Ennek első lépéseként már rendelkezünk a szükséges osztálydiagramokkal. Ezen diagramokat pedig le tudjuk képezni egyszerű szöveges (XML) fájlokká a fentebb említett XMI formátum segítségével. Ahhoz, hogy tovább folytassuk a gondolatmenetet, ***az UML modellünkre úgy kell gondolni, mint kellően részletes kiindulási állapotra, amelyet át kell alakítani egy másik, a későbbiekben az adatkezelés céljából felhasználható formátumra***. Ebben a sémanyelvek nyújtanak segítséget, amelyek lehetőséget adnak az adatokat leíró szerkezet, a tartalom és a szemantika meghatározására XML dokumentumok esetén. Nagy mennyiségű szabványos és szabadalmazott XML séma jelent meg, hogy leírja ezeket a megkötéseket, és ezen sémák egy része maga is XML alapú. Az XML szabályos struktúrája és szigorú elemzési szabályrendszere képessé teszi a szoftvertervezőket, hogy az elemzést szabványos eszközökre bízzák, és mivel az XML egy általános, adatmodellorientált keretrendszert biztosít az alkalmazás specifikus nyelvek fejlesztőinek, a szoftverfejlesztőknek csak a szabályrendszer és az adat kifejlesztésére kell koncentrálni, így viszonylag magas absztrakciós szintet elérve. Természetesen sémanyelvből is több féle létezik, a mi elsődleges választásunk a W3C (World Wide Web Consortium) által ajánlásként elfogadott XML Schema nyelvre esett.

XML Schema

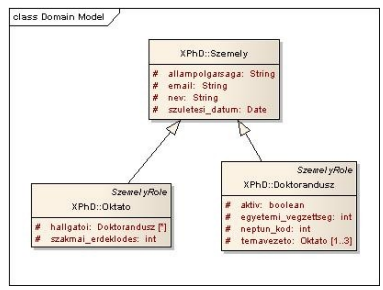
Az XML sémaleíró nyelvek célja, hogy formális leírást adjanak XML dokumentumosztályok definiálására. Másképpen megfogalmazva nyelvtani leírások, amelyek segítségével az XML struktúrák által ábrázolni kívánt üzleti szabályokat definiálhatjuk. De mit is ír le egy séma?

- Definiálja milyen elemek és attribútumok lehetnek egy dokumentumban.
- Definiálja az elemek egymásba ágyazását.
- Definiálja a gyermekelemek sorrendjét.
- Definiálja a gyermekelemek számát.
- Definiálja, hogy egy elem üres, szöveget, vagy további gyermekelemeket tartalmaz.
- Definiálja az elemek és attribútumok adattípusát.
- Alapértelmezett értéket definiál elemekhez és attribútumokhoz.

Egyszerűbben megfogalmazva a dokumentum struktúráját, valamint a benne található adok típusát és értelmezési tartományait definiálja. A sémák egyik fő célja, hogy miután formális leírást adtak egy dokumentumosztályról, ezek után programozottan meg lehessen mondani, hogy egy konkrét XML dokumentumpéldány megfelel-e a sémának? Ezt a folyamatot nevezzük validálásnak, és az olyan XML dokumentumot, ami egy sémának megfelelő validnak vagy érvényesnek. Az XML Schema használatával ezek alapján biztosítani tudjuk mindazt, amit az előbbi részekben a modularizáció és az egyéb forrásokból érkező adatok feldolgozásról mondtunk.

Az XML Schema-ra tekinthetünk úgy, mint egy speciális nézete egy kellően részletes UML osztálydiagrammal kifejezett modellnek. Ilyennel pedig rendelkezünk a koncepcionális modell révén. Ráadásul ez a sémanyelv tartalmaz beépített típusokat is, amelyek igen közel állnak az UML által használt alapvető típusokhoz. Mindazonáltal biztosítja annak lehetőségét is, hogy saját típusokat definiáljunk, ha szükséges. Ami még szintén hasznos, hogy az osztálydiagram által kifejezett asszociációkat is képesek kifejezni. Itt egy apró megjegyzés a kódgeneráláshoz, hogy az UML modellben nem különböztetjük meg az asszociációk sorrendjét, viszont ha az XML dokumentumokban fontos a kapcsolódó elem sorrendje, akkor egy apró trükkel ezt is megoldhatjuk. Nevezetesen az asszociációkhoz az UML bővíthetőségét kihasználva csatolhatunk egy *pozíció* elemet, amelyet a transzformálás során majd figyelembe veszünk.

A ***transzformáció*** pedig nem más, mint ***az UML metamodelljének elemeit leképezzük az XML Schema szabvány által használható elemekre***. Ez egy automatizálható folyamat, csak a szabályokat kell rögzítenünk, amelyek segítségével az UML metamodelljének koncepcióit átvisszük a séma nyelvére. A transzformációhoz közbenső lépést a diagram XMI formátumban történő kimentése jelenti. Ehhez kell elkészítenünk a megfelelő XSLT stíluslapokat, amelyek elvégzik a transzformációkat. Álljon itt egy példa, ami bemutatja ennek a folyamatnak a lépéseit. Elsőként vegyünk egy egyszerű osztályt, amelyen demonstráljuk az átalakítást.



35. ábra: A Személy osztály UML diagramja

Ebből kapjuk a következő XMI fájlt:

```
<UML:Class xmi.id = 'Iba1efm1' name = 'Szemely' visibility = 'public'>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id='Iba1efm2' name='nev' visibility='private' >
      <UML:StructuralFeature.type>
        <UML:Class xmi.idref = 'I15ba1efm3'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
    <UML:Attribute xmi.id='Iba1efm6' name='email' visibility='private'>
      <UML:StructuralFeature.type>
        <UML:Class xmi.idref = 'Iba1efm10'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
   <UML:Attribute xmi.id='Iba1efm6' name='allampolg' visibility='private'>
      <UML:StructuralFeature.type>
        <UML:Class xmi.idref = 'Iba1efm10'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
   <UML:Attribute xmi.id='Iba1efm6' name='szul_datum' visibility='private'>
      <UML:StructuralFeature.type>
        <UML:Class xmi.idref = 'Iba1efm10'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
  </UML:Classifier.feature>
</UML:Class>
```

36. ábra: A Személy osztály XMI reprezentációja

A kapott XMI fájlt *az alábbi* ***általunk készített*** *XSLT stíluslappal* tudjuk áttranszformálni az általunk várt XML Schema formátumra: (*részlet*)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
    <xsl:template match="/">
      <xs:schema>
       <xs:element
name="{xmi:XMI/uml:Package/packagedElement/packagedElement/packagedElement/
packagedElement/@name}">
          <xs:complexType>
            <xs:sequence>
               <xsl:for-each
select="xmi:XMI/uml:Package/packagedElement/packagedElement/packagedElement
/packagedElement/packagedElement[@xmi:type='uml:Class']">
       <xsl:if test="substring(@name, 1, 1)!='$'">
        <xs:element name="{@name}" type="{@name}Type" minOccurs="0"
maxOccurs="unbounded">
         </xs:element>
       </xsl:if>
      </xsl:for-each>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xsl:for-each
select="xmi:XMI/uml:Package/packagedElement/packagedElement/packagedElement
/packagedElement/packagedElement[@xmi:type='uml:Class']">
    <xsl:if test="substring(@name, 1, 1)!='$'">
       <xsl:call-template name="class"></xsl:call-template>
    </xsl:if>
   </xsl:for-each>
  </xs:schema>
 </xsl:template>

 <xsl:template name="class">
  <xs:complexType name="{@name}Type">
   <xsl:variable name="ClassID" select="@xmi:id">
       </xsl:variable>
       <xsl:if test="generalization">
       <xsl:call-template name="generalization">
       <xsl:with-param name="childClassID" select="$ClassID" />
       </xsl:call-template>
      </xsl:if>
     <xsl:choose>
       <xsl:when test="generalization"></xsl:when>
       <xsl:otherwise>
        <xsl:call-template name="attributes">
         <xsl:with-param name="ClassID" select="$ClassID"></xsl:with-param>
        </xsl:call-template>
        <xs:attribute name="id" type="xs:ID"></xs:attribute>
       </xsl:otherwise>
     </xsl:choose>
       </xs:complexType>
    </xsl:template>
```

A transzformáció eredmény pedig a következő séma fájl lesz:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:uml="http://schema.omg.org/spec/UML/2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
   <xs:element name="XPhD">
 <xs:complexType name="SzemelyType">
      <xs:attribute name="nev"/>
      <xs:attribute name="szuletesi_datum"/>
      <xs:attribute name="allampolgarsaga"/>
      <xs:attribute name="email"/>
      <xs:attribute name="id" type="xs:ID"/>
   </xs:complexType>
   <xs:complexType name="DoktoranduszType">
      <xs:complexContent>
         <xs:extension base="SzemelyType">
            <xs:attribute name="neptun_kod"/>
            <xs:attribute name="aktiv"/>
            <xs:attribute name="egyetemi_vegzettseg"/>
            <xs:attribute name="temavezeto"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="OktatoType">
      <xs:complexContent>
         <xs:extension base="SzemelyType">
            <xs:attribute name="szakmai_erdeklodes"/>
            <xs:attribute name="hallgatoi"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
</xs:schema
```

37. ábra: A generált séma

Ennek *a transzformációnak a segítségével elértük, hogy a* ***rendszerünk a szakterületi modellből képes automatikusan előállítani*** *a webalkalmazásunk modell részéhez kapcsolódó adatvalidációs sémát.* A jelentősége többrétű ennek a folyamatnak. Egyrészt megnyitja az utat újabb kommunikációs felületek felé, másrészt az adatkezelés egyszerűsítésére is alkalmas. Ehhez egy újabb W3C ajánlást fogunk felhasználni, amely a felhasználói felületen történő adatmegadás és validáció területén nyújt igen újszerű megoldást, melynek a lényege, hogy már a kliensoldalon megtörténik a felhasználó által megadott adatokból az XML dokumentum összeállítása, és csak akkor kerül elküldésre, ha maradéktalanul megfelel az általunk most létrehozott sémának.

Azonban az XML séma sem nyújt tökéletes megoldást a szakterületi modell maradéktalan leírására. Jellegéből adódóan ez a sémanyelv az XML dokumentumok architektúrális felépítését ellenőrzi. Olyan szemantika szabályok, mint például, ha egy

személy neme férfi, akkor az angolszász elnevezési konvenció szerint Mr. előnévvel kell rendelkeznie. Ehhez egy másik sémanyelvre lesz szükségünk.

A Schematron sémanyelv

A Schematron a legtöbb sémanyelvtől különbözik, mert nyelvtanok helyett szabályrendszereket használ. A Schematron sémában un. kijelentéseket (assertion) lehet megadni egy adott kontextusra vonatkozóan. Ha egy általunk megadott kijelentés nem teljesül, akkor egy – a fejlesztő által megadott – üzenettel leáll. A nyelv egyik előnye, hogy a kijelentések megadásához az angol nyelv mondataihoz hasonló szerkezeteket kell megadni. A kontextus kijelöléséhez pedig az XSLT által is használt XPath jelölést használja.

A Schematron szerkezete nagyon egyszerű. Mintákat adunk meg, amelyek a szabályok összefogására szolgálnak. A mintákat láthatjuk el névvel, amely az ellenőrzés során megjelenítésre kerül, így ha egy szabály sérül, akkor látjuk, hogy melyik mintánál történt a hiba. A szabályok rendelkeznek a kontextust kijelölő XPath kifejezéssel, amelyen a szabályon belüli kijelentéseket meg kell vizsgálni. A következő példa a fentebb felvetett probléma ellenőrzését mutatja be:

```
<pattern name="Check structure">
   <rule context="Person">
      <assert test="@Title">The element Person must have a Title.</assert>
      <assert test="count(*) = 2 and count(Name) = 1 and count(Sex) = 1">
The element Person should have the child elements Name and Sex.</assert>
      <assert test="*[1] = Name">The element Name must appear before
element Age.</assert>
   </rule>
</pattern>
<pattern name="Check co-occurrence constraints">
   <rule context="Person">
      <assert test="(@Title = 'Mr' and Sex = 'Male') or @Title != 'Mr'">
If the Title is "Mr" then the sex of the person must be "Male".</assert>
   </rule>
</pattern>
```

38. ábra: Schematron példa

A Schematron másik nagy előnye, hogy eleve úgy lett a kezdetektől fogva kifejlesztve, hogy az XML Schema bővíthetőségi lehetőségét kihasználva, magába az XML Schema dokumentumban lehet elhelyezni, ezáltal kombinálva a két nyelv előnyeit. A feldolgozás két lépcsőben zajlik, először lefut a Schematron validálás, ehhez a nyelv alkotói biztosítanak egy XSLT stíluslapot, majd pedig ha ez hiba nélkül

lezajlott, akkor jöhet az eredetileg tervezett XML Schema szerinti validáció is. A következő ábra az egymásba ágyazásra mutat egy példát.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="Person">
      <xs:annotation>
         <xs:appinfo>
            <sch:pattern name="Co-occurrence constraint on attribute Title"
xmlns:sch="http://www.ascc.net/xml/schematron">
               <sch:rule context="Person[@Title='Mr']">
                  <sch:assert test="Sex = 'Male'">If the Title is "Mr" then
the sex of the person must be "Male".</sch:assert>
               </sch:rule>
            </sch:pattern>
         </xs:appinfo>
      </xs:annotation>
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Sex">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:enumeration value="Male"/>
                     <xs:enumeration value="Female"/>
                  </xs:restriction>
               </xs:simpleType>
            </xs:element>
         </xs:sequence>
         <xs:attribute name="Title" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

39. ábra: Schematron beágyazása más sémanyelvbe

Összességében elmondhatjuk, hogy a Schematron tökéletes kiegészítője az XML Schema-nak, hogy az olyan összetett szakterületi megszorításokat is ki tudjuk fejezni, amire csupán a szerkezeti megszorítások nem elegendőek. Nagyon kevés olyan eset léphet fel, amely ezzel a két, különböző validálási szempont valamelyikével nem lenne kifejezhető. Segítségükkel az alkalmazáslogikát tehermentesíthetjük a validációs feladatoktól, még átláthatóbbá téve az alkalmazás egyes rétegei és funkcionalitása közötti határvonalat. Ennek az automatizálása azonban a jelenlegi fázisban még nem megvalósított, további fejlesztést igényel, hogy az UML modelljeinkbe miként tudjuk beemelni *ezeket a szabályokat* az UML által támogatott OCL kifejezésekkel szemben. Első nekifutásra megjegyzésként kerülnek elhelyezésre a diagramokon, ezáltal *a transzformáció során automatikusan átvehetjük*, eltekintve az OCL kifejezésektől.

Az adatmódosítás támogatása

Az eddigi lépések során arra kerestük a választ, hogy milyen lehetőségek nyílnak az adatok elérésre a modellből, illetve, hogy milyen szabványos felületen keresztül lehet a bevitelt megvalósítani. Erre az ismertetett XML technológiák megfelelő keretet biztosítottak. Ha a beérkező adatokból össze tudjuk állítani a megfelelő XML dokumentumot, akkor a generált séma segítségével annak érvényessége ellenőrizhető. Viszont ezen adatok felhasználói felületen keresztül történő bevitele mindig is plusz feladatot jelentett a programozók számára. Ha szerették volna elkerülni a hibás adatok elküldését, akkor a különböző böngészők számára el kellett készíteni a specifikus ellenőrző szkripteket, amelyek többnyire VisualBasic vagy Javascript nyelven íródtak az *űrlapok adatainak ellenőrzésére*. Ez hosszadalmas és meglehetősen gépies feladat csak azért, hogy a felhasználó már az adatok bevitele közben értesüljön a hibáról. Természetesen meg lehetne azt is tenni, hogy megvárjuk míg begépeli, elküldi, kiértékeljük, és ha hiba történt, akkor tájékoztatjuk annak okáról.

A másik része a problémának, hogy magát *a beviteli mezőket tartalmazó oldalt* is *létre kell hozni*, minden egyes attribútumhoz a neki megfelelő típusú vezérlőelemmel. Az oldalak szerkezete ebből kifolyólag eléggé jól körbehatárolható, de ez is sok időt vesz igénybe a fejlesztési oldalon. Ha pedig valamiért a koncepcionális modellen változtatást kell eszközölni, akkor annak az adatmódosító felületekre is kihatása van, ami csak a hibalehetőségek számát növeli.

Sajnos ennek kiküszöbölésére *az ismert módszertanok nem nyújtanak támogatást*. Az adatmódosítás igen elhanyagolt terület és az általános támogatottsága igen alacsony. Amely módszertanhoz pedig készült speciális támogató eszköz (pl. WebML esetén a WebRatio program), ott az adatmódosítás kötött felhasználói felülethez van rendelve, adott technológia használatára kényszerítve a felhasználót.

A mi ***célunk olyan módszer kidolgozása***, amely a koncepcionális modellen alapulva ***automatikusan lehetővé teszi*** a rendszerben tárolt ***adatok könnyű és hatékony kezelését***, anélkül, hogy a rendszer tervezőinek és fejlesztőinek kézzel kellene ehhez kódot írniuk. Miután a tervezés során rögzítettük a szakterület követelményeit, azokat le is írtuk és formalizáltuk, jogosan vetődik fel a kérdés, hogy miért ne lehetne ezt a lépést is automatikussá tenni.

A megoldáshoz az adatok megadására használt űrlapokat kell megvizsgálni. Ezek leírására manapság leggyakrabban a HTML nyelv által biztosított űrlapokat (formokat) használjuk. Ez egy kötött eszközkészlet, amely képes a beviteli mezőkben megadott adatokat összefogni, és egy http kérés során eljuttatni a szerverhez. A kiszolgáló oldalán pedig kezdődik a kiértékelés. A folyamat jól ismert és sok mindent már nem lehet rajta változtatni. Viszont ha a korábbi felvetéseinket megnézzük, akkor abban a felhasználó számára előállított űrlap elkészítésére és validálására tettünk észrevételeket.

A *felmerült ötlet megvalósíthatóságának alapját* a korábbi részben említett XML Schema generálása fogja jelenteni. Ezt ***a sémát fogjuk felhasználni a felhasználói felület automatikus előállításához***. Ehhez készíthetnénk saját transzformátorokat, de ennél szerencsésebb helyzetben vagyunk, mert a W3C szervezet elkészítette az űrlapok hatékonyabb kezelését célzó XForms ajánlását.

XForms

Az XForms dinamikus, rendszer és egyben szkriptnyelv független, valamint teljesen szabványos. Nem egy különálló dokumentumtípus, hanem más nyelvekbe integrálódik, mint például az XHTML vagy SVG. Sokat tanult a HTML-beli űrlapok jó és rossz tulajdonságából. Az XForms követhetőbbé és tisztábbá teszi azt, hogy milyen adatot is küldtünk el, és hová. Könnyen újra felhasználhatóvá teszi az űrlapokat, amik már nem szorosan kötődnek ahhoz az oldalhoz, amelyiken használják.

A klasszikus HTML webes űrlapok nem különítik el az űrlap funkcióját a megjelenítéstől. Egyben tartalmazzák mindezt a *form* elemben. Ezzel ellentétben az XForms elkülöníti az űrlap kezelését, a megjelenítését és az adatait. Így rugalmas megjelenítési beállításokat tesz lehetővé. Az XForms egyik fontos elve, hogy az űrlapja adatainak tárolásához egy XML dokumentumot használ, melyet adat-példánynak nevezzük. Bármilyen módosítást viszünk fel az űrlapon, az az XML példányra fog vonatkozni. Fontos kiemelni, hogy az adatpéldány adja meg az XForms modellben használandó adatstruktúrát. Ehhez kiegészítésként hozzá lehet csatolni egy XML Schema-t, amely segítségével automatikusan ellenőrzi a bevitt adatok helyességét. Az XForms mindaddig nem engedi az adatok elküldését, amíg az nem felel meg a séma által meghatározott követelményeknek.

Az XForms előnyei:

- MVC modell kialakítása,
- deklaratív programozási nyelv, amit könnyű tanulni, és a nyelvvel készített programokat egyszerű karbantartani, belőni,
- kompatibilis XML szabványokkal, úgy mint CSS, XML Schema és XPath,
- kiterjeszthetőség,
- nemzetköziesítés.

Az XForms még nem terjedt el olyan szinten, hogy a böngészőkben natívan implementálva lenne, talán ezt emelhetjük ki egyedüli hátrányaként. Azonban a kipróbálásához könnyen találhatunk megfelelő eszközöket. Manapság az XForms-ot majdnem minden webböngésző tudja futtatni valamilyen letölthető kiegészítő (plugin) használatával. Az alábbi ábrán egy egyszerű Xforms űrlapot használó oldalt láthatunk. Ami jól észrevehető, hogy az űrlap megadása nem a megjelenítendő (body) részben van, hanem még az oldal fejlécében kerül meghatározásra. Itt helyezkedik el az adatok tárolásáért felelős XML dokumentum is.

```
<html     xmlns="http://www.w3.org/1999/xhtml"
          xmlns:xf="http://www.w3.org/2002/xforms">
<head>
<title>Search</title>
<xf:model schema="instance.xsd">
  <xf:instance> <!-- Az adatok az „istance” elembe kerülnek. -->
   <data xmlns="">
    <q/>
   </data>
  </xf:instance>
 <xf:submission action="http://example.com/search" method="get" id="s"/>
</xf:model>
</head>
<body>
<p>
 <xf:input ref="q">
  <xf:label>Find</xf:label>
 </xf:input>
 Az „input” elem definiál egy szövegbeviteli mezőt.
 <xf:submit submission="s">
  <xf:label>Go</xf:label>
 </xf:submit>
 A „submit” elem olyan gombot ír le amivel be lehet vinni az adatot.
</p>
</body>
</html>
```

40. ábra: XForms példa

A példából az is látható, hogy az űrlap megjelenítésével nem sokat foglalkoztunk. Egyszerűen a lap törzsén annyit mondtunk, hogy most egy adott elemét az XML dokumentumunknak szeretnénk megjeleníteni. A XForms ilyenkor megnézi a megadott elem típusát és annak megfelelően fog elhelyezni egy megfelelő bemeneti mezőt. Az adott elem típusát megadhatjuk külön részben is, de a legegyszerűbb egy XML Schema-t átadni, amivel szemben egyrészt validáltatni is szeretnénk, másrészt az elem típusát is meg tudja ebből határozni. Speciálisan meg lehet adni egy címke (*label*) elemet is, ami a beviteli mező előtt fog elhelyezkedni. Ehhez hasonlóan csatolható hozzá még egy súgó mező is, illetve egy figyelmeztető üzenetet tartalmazó szöveg is, amely akkor jelenik meg, ha hibás inputot adunk meg.

Ezeknek az automatikus előállításához a kiindulási (koncepcionális) modellünket kell egy picit átalakítani úgy, hogy ezen információkat is elhelyezzük rajta. Ez történhet megjegyzésként, vagy az UML kiterjeszthetősége révén kulcsszavas értékekkel is. Az Xforms ezeken felül még számos lehetőséget rejt, amelyeknek a kihasználása még könnyebben használhatóvá teheti az alkalmazásunkat. Ilyen például a beviteli mezők lapokra (fülekre - *tabs*) történő csoportosítása, annak érdekében, hogy ne egy óriási űrlappal szembesüljünk, hanem egy logikusan tagolt felülettel.

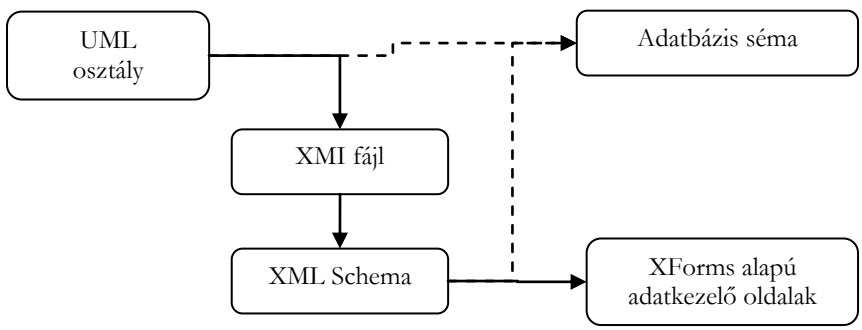
41. ábra: Xforms több lappal

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:xforms="http://www.w3.org/2002/xforms">

    <xsl:output method="xhtml"/>
    <xsl:param name="class" />

    <xsl:function name="fn:getType">
        <xsl:param name="node"/>
        <xsl:for-each
       select="$node/xsd:complexType/xsd:sequence/xsd:element">
            <xsl:if test="@name=$class">
                <xsl:value-of select="@type"/>
            </xsl:if>
        </xsl:for-each>
    </xsl:function>
    <xsl:template match="xsd:schema">
        <html xmlns="http://www.w3.org/1999/xhtml"
            xmlns:xforms="http://www.w3.org/2002/xforms"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <head>
                <xforms:model
       schema="http://localhost/project/uml2xsd/schema.xsd">
                    <xsl:attribute name="id">
                      <xsl:value-of select="$class"/></xsl:attribute>
                    <xforms:instance>
                       <xsl:variable name="root"
select="xsd:element/@name"/>
                        <xsl:element name="{$root}">
                           <xsl:apply-templates mode="model">
                             <xsl:with-param name="type"
select="fn:getType(xsd:element)"/>
                              </xsl:apply-templates>
                        </xsl:element>
                    </xforms:instance>
   <xforms:submission id="submit"
action="http://localhost/project/navigation/show.php" method="post"/>
 </xforms:model>
            </head>
            <body>
                <h1>
                    <xsl:value-of select="$class"/>
                </h1>
                <xsl:apply-templates mode="form">
                    <xsl:with-param name="type
                      select="fn:getType(xsd:element)"/>
                </xsl:apply-templates>
                <table><tr><td align="center">
                    <xforms:submit submission="submit">
                        <xforms:label>Save</xforms:label>
                    </xforms:submit>
                </td></tr></table>
            </body>
            </html>
    </xsl:template>
```

42. ábra: XSD2XForm transzformátor

Az előző oldalon látható példa annak a ***transzformáció****hoz használt stíluslapunknak* egy részlete, *amely* ***megvalósítja, az automatikus kódgenerálást*** *a korábbi lépések során, a koncepcionális modellünkből származtatott XML Schema fájlból.* A példa szemlélteti az egyes osztályok esetén a megjelenítendő mezőket feltérképezését és az XForms űrlap modell részében található XML példány felépítését. Az eredetileg az UML osztálydiagramon attribútumként szereplő mezők így alakulnak át interaktív beviteli mezőkké. Ezzel sikerült megvalósítani a célunkat, ami lehetővé teszi a modellezett Web alapú Információs Rendszerünk adatkezelési funkcióinak az automatizálást.

Az ***általunk kidolgozott kódgenerálási folyamat*** a következőképen szemléltethető:



43. ábra: UML és XML technológiák használata a kódgeneráláshoz

Az ábrán folytonos vonallal jelöltük az általunk végrehajtott transzformációs lépéseket, amelynek eredményeképp a platform független modellünkből kialakult több platform specifikus modell is. *Ezen* ***modelljeink lehetővé teszik a webalkalmazásunk egy működő prototípusának gyors előállítását****.* Az ábrán szaggatott nyíllal jelöltük azokat az opcionális utakat, amelyet a módszer még magában rejt. Ha az XML Schema-ból szeretnénk az adatok tárolásáért felelős réteget előállítani, akkor választhatjuk a séma relációs modellé történő áttranszformálást. Erre számos megoldással találkozhatunk. Viszont az XML sémánk magán hordozza annak a lehetőségét, hogy az adatbázisként valamilyen natív XML adatbáziskezelő rendszert alkalmazzunk. Ezen rendszerek jelenleg még nem terjedtek el meghatározóan, de a későbbiekben számolni kell ezzel is. A másik szaggatott nyíl pedig magából az osztálydiagramból indul, ami annak a lehetőségét jelenti, hogy a korábban már említett MVC modell részhez Java osztályok származtathatóak bármilyen nehézség nélkül, amelyek egy objektumrelációs fordító segítségével könnyedén tárolhatóak relációs adatbázisokban (ilyen például a Hibernate keretrendszer).

További lehetőségek

A fejezetben bemutatott technológiák és keretrendszerek természetesen további lehetőségeket is biztosítanak a webalkalmazások modell vezérelt fejlesztésére. Számos irányzat létezik, amely ezen rendszereknek más aspektusát veszi figyelembe, szemben a mi általunk választott adat-orientált szemléletmóddal.

A kialakított kódgenerálási módszerrel azonban gyorsan és hatékonyan tudjuk elkészíteni webalkalmazásainkat és a modelljeinkből származtatni tudjuk a működő rendszerünket. Nyilvánvalóan eme statikus szemlélete az alkalmazásainknak megköveteli a dinamikus oldal vizsgálatát is. Az üzleti folyamatok UML diagramok és egyéb jelölőnyelvekkel történő modellezése igen aktívan fejlődő ágat képvisel a tudományban. A dinamizmus modellezésére az UML-nek is megvan a saját kifejező eszköze az aktivitásdiagramok, az interakció diagramok és állapotátmenet diagramok formájában. A rendszerben lezajlódó változásokat ezekkel le tudjuk írni, amelyet a későbbiekben felhasználhatunk a felhasználói tevékenységeknek megfelelő funkcionalitás programozott származtatásához.

A végrehajtható tevékenységek egy része jellemzően olyan, amely az üzleti kapcsolatok során átvezet a webszolgáltatások és a szolgáltatás-orientált architektúrák világába. A modellezési technikánk ennek támogatását a tartomány specifikus nyelvünkben található *Szolgáltatás* formájában tartalmazza. Ezek implementálása során a webszolgáltatásokhoz szükséges WSDL (*WebService Description Language*) fájl létrehozása könnyedén beemelhető a módszertanunkba.

Ráadásul a bemutatott XForms technológia nem csak az általunk előállított séma és alkalmazáslogikával tudja a kapcsolatot tartani, hanem az eseménykezelésének köszönhetően lehetőség ad arra is, hogy az űrlap a tartalmát dinamikusan módosítsa például egy webszolgáltatástól kapott információval. De az adatorientált szemléletmódot figyelembe véve az adatok megjelenítésénél is lehet még találni új utakat. Gondoljunk itt például a skálázható vektorgrafikára (*Scalable Vector Graphics - SVG*), amely egy xml alapú nyelv, és tökéletesen illeszkedhet mondjuk az információs rendszerből kinyerhető statisztikák grafikus megjelenítésére. A lehetőségek és a technológiák tárháza korlátlan, kimeríthetetlen forrását nyújtva a továbbfejlődésnek.

Összefoglalás

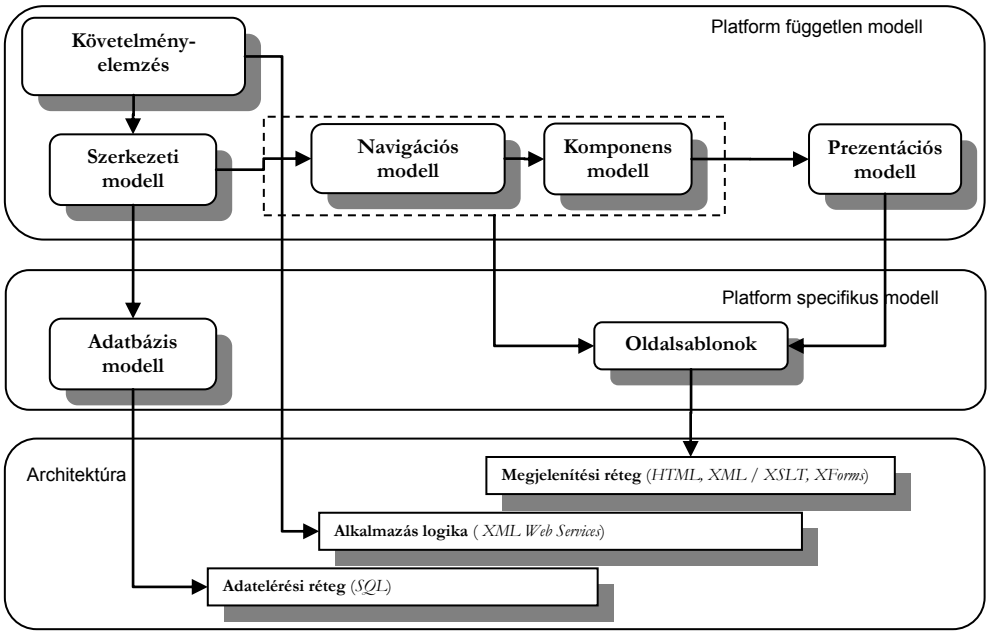
A dolgozatban a webalkalmazások fejlesztéséhez kínáltunk egy ***új, szisztematikus tervezési módszertant***, amely ezeket az alkalmazásokat *adatorientált szempontból* vizsgálja. Bemutattunk irányelveket, amelyek az adott szakterülethez kapcsolódó *tartomány alapú modellek* fejlesztését segítették. A módszerünk az UML metamodelljének kiterjesztésén alapul, amelyhez felhasználtuk a tartomány alapú tervezés irányelveit. A metamodell leírása részletesen megtalálható a dolgozat függelékében.

A szakterület leírásához ***elkészítettünk egy új tartomány specifikus nyelvet*** (*DSL*), amely bevezeti a *szerepkörök* fogalmát. Fontosnak tartottuk már a szakterület feltérképezésekor kiemelni az egyes *entitások különböző szerepkörökben történő viselkedésének* vizsgálatát. A nyelvünk segítségével meghatározott modellelemek kerültek leképzésre a módszertanunk által kínált ***szerkezeti modell*** metamodelljére, majd ebből *származtattunk egy UML profilt*.

Az általunk ajánlott fejlesztési folyamat második lépése a ***navigációs modell*** elkészítése. Kidolgoztunk egy módszert, ami lehetővé teszi a webalkalmazás navigációs modelljének *származtatását a szerkezeti modellből*. Ezt követően, az UML2 újdonságait felhasználva, a navigációs elemekből ***komponenseket alakítunk ki*** annak érdekében, hogy koncepcionális információszolgáltató elemek jöjjenek létre. Ezen elemek interfészek segítségével kapcsolhatóak egymáshoz, amely kapcsolatok a szükséges információt tartalmazó *absztrakt oldalak* kialakítását teszik lehetővé. A prezentációs réteg ezen absztrakt oldalakat alakítja át a kliensek számára megfelelő formátumra, így biztosítva a különböző eszközök által történő hozzáférést.

Az MDE fejlesztési irányzatban rejlő lehetőségek kiaknázásához ***UML és XML technológiákat alkalmazunk,*** amelyek segítségével a ***modelljeinkből futtatható kódot állítunk elő***. A szabványok (*OMG XMI formátum*) és ajánlások (*W3C XML Schema, XForms*) használatával további *lehetőség nyílik az adatmanipulációs oldalak automatikus generálására is*. Így a módszerünk hatékony támogatást biztosít az

adatorientált webalkalmazások fejlesztéséhez. Az *általunk kialakított* módszer az alábbi ábrán látható.



44. ábra: Fejlesztési fázisok

A bemutatott módszer a konferenciákon és a közvetlen környezetemben is komoly szakmai visszhangot váltott ki. Több fiatal kolléga érdeklődését is sikerült felkeltenem, közöttük többen vannak olyanok, akik a tudományos munkájukat ebben az irányban szándékoznak folytatni [21] [22] [32].

Summary

1 Introduction

The evolution of Web technologies increased the presence of the Web in our everyday life starting from personal home pages through corporate portals and Web shops up to Web applications implementing complex business processes. This diversity of applications makes the selection of the appropriate technology and platform harder, in particular, when these applications should collaborate with each other.

Some of them might be suitable in a given case, which are sometimes alternatives to each other, while they should be combined in other cases. As these technologies evolve very fast, they might become obsolete soon. It is very hard to predict, which technologies will be out "tomorrow" so business logic should be as independent as possible from technologies to allow change in the technology without affecting business processes and rules.

The modeling of such systems is a very complex process since (in an ideal case) it should take both existing and future technologies into consideration in order to create a flexible system which allows further development. Despite changing of the technology, models of the application domain remain almost the same since they capture business model and processes. The best practice tells us to keep the modeling of the application domain and the technological details separated.

1.1 Web Information System Development

In our research, we are mainly considering a subset of the aforementioned applications of the Web: the Web Information Systems (WISs). "*A WIS is an Information System providing facilities to access complex data and interactive services via the Web*" [33]. These "*represent a sub-category of mass information systems that are typically support on-line information retrieval and routine task by way of self-service for a large number (thousands or millions) of occasional users who are spread over many locations*" [34].

Table 1 shows a categorization of these systems depending on the direction of communication and the characteristics of information [35].

	Asymmetrical communication	Symmetrical communication
Objective information	Information provider	Information system
Persuasive information	Advertisement	Community

Table 1: Four perspectives of WISs.

The focus of information providers is clearly on one-way distribution of information, from the system to its external users. A news portal is an example of such a system. A WIS can also be seen as a marketing channel, on purpose to distribute promotional messages to an external audience: a home page can be considered like an "advertisement" system. A WIS can be used to create and develop a virtual community whereof forums are examples. The fourth perspective is also known as the development of data-intensive Web applications. Typical examples of such WISs are corporate portals, on-line flight booking systems, etc. Their focus is on the data structure and the information flow between the system and the users to support their work by the help of complex business processes which both require and provide information from/to the users.

In order to save time, development groups often disregard methodological approaches during the modeling of such systems. In most of the cases, this leads to an improper design and individual (not reusable) modeling practices. These make dynamic responses to emerging needs just on time almost impossible since new requirements might cause complete redesign of some system parts or even the whole system. The re-implementation of the affected system parts could be a very time-consuming operation. Its reason is that there is a gap between the model and the implementation of a system. Even when applying traditional software design methodology, once a model is created, it is "thrown away" after the first implementation and is neither used again nor updated. There is no direct connection between the model and the code so changes in the model are not reflected in the code, and contrary, changes in the code are not reflected in the model. This means that after any of them are modified, they are not synchronized anymore; therefore, the model cannot fulfill one of its primary purposes, namely, to tell relevant information about the system.

2 Motivation

The legacy system

Several years ago the author was involved in a project whose aim was to develop a Web-based application for the Doctoral School of Mathematics and Computer Science at the University of Debrecen. The requirements against the system were to keep track of students and teachers of the Doctoral School and, manage data of doctoral programs and courses as well. Besides the usefulness of having these data collected, it also allows the creation of some statistical reports. Moreover, teachers and students are able to have their list of publications collected. According to Table 1, this system can be considered as an information system since it is interactively used to accomplish their tasks including querying, modifying and inserting data.

The back-end of the system was a PostgreSQL DBMS while the front-end consisted of HTML pages combined with CGI scripts written in Perl. The application logic is multi-layered: utilizing PL/pgSQL stored procedures (executed inside the DBMS) and Perl.

Nowadays, almost all of these technologies are relatively rarely used as they do not support a high level of separation between the structure and the presentation of data.

Problems

After deploying the system new requirements arose. Some of them were easy to implement but a few of them required the redesign of the whole data model. This was partly needed due to the changes of the legal regulation and organizational restructuralization. On the other hand, most of the problems arose when the underlying data model had to be changed. This caused several other changes since it influenced other system parts as well.

These issues outlined that the maintenance and further development of the system is complicated therefore hard to accomplish. As a consequence the importance of the reconsideration of the whole system became apparent.

In general, problems are arising when requirements are evolving (which is more often the case), since maintenance might be very problematic as new developers are probably not so familiar with the older technologies and even if they are, this may lead to an improper design and/or coding strategy. For that reason, we considered to apply a model-based solution to reduce development time and maintenance costs.

3 Theoretical background

3.1 Domain Driven Design

Domain Driven Design is an approach for dealing with highly complex domains which is based on making the domain itself the main focus of the project, and maintaining a software model that reflects a deep understanding of the domain. We need to understand from the beginning where the software is originated from and which domain (environment) is related to. Domain Driven Design helps us to understand and to focus on the problem domain and to build a domain model which captures the basic concepts. During the construction of the domain knowledge we have the possibility to extract the essential information and to generalize it.

Domain Driven Design uses basic building blocks to define the domain itself. These blocks are entities, value objects, services and modules. Services act as interfaces which provide operations. The purpose of service is to simply provide functionality for the domain. Using modules in design is a way to increase cohesion and decrease coupling. Moreover, it uses well known design patterns like Aggregate, Builder, Factory and Repository to decrease complexity. Aggregate is a domain pattern used to define object ownership and boundaries. Factories and Repositories are two design patterns which help us deal with object creation and storage.

3.2 Domain Specific Languages

A DSL is a language specially geared to working within a particular area of interest: it might be a vertical domain such as telephone design, or a horizontal one like workflow. Some well-known examples of DSLs are HTML, SQL and UNIX mini languages. What is radically new is the idea of creating your own DSL for your

own project. Domain specific languages have important design goals that contrast with those of general-purpose languages:

- domain specific languages are less comprehensive.
- domain specific languages are much more expressive in their domain.
- domain specific languages should exhibit minimum redundancy.

In Model Driven Engineering many examples of domain-specific languages may be found: like OCL, a language for decorating models with assertions or QVT, a domain specific transformation language. However, languages like UML are typically general purpose modeling languages.

3.3 Model Driven Architecture

OMG's Model Driven Architecture (MDA) [13] provides a framework which allows applications to be described in a platform-independent manner. In MDA, artifacts are formal models describing a given aspect of the system. MDA uses UML as a standard modeling language in order to achieve a common understanding among stakeholders.

During model-driven development, a platform-independent model (PIM) is created first. This model should have a high level of abstraction hence implementation details—database system, application server platform, etc.—should be hidden. PIM models give viewpoints on how the system will support the business. The next step is a transformation of a PIM into one or more platform-dependent models called Platform Specific Models (PSMs).

The transformation process will deal with a specific set of technologies. For each specific technology a separate PSM is generated because most of today's systems span several technologies. The final step in the development process is the transformation of each PSM to code. Because a PSM fits its technology rather closely this transformation could be done relatively easy.

In case of a technological change (or evolution), a PIM is not subject to change as it models an application in a platform-independent manner. All what we need is to create a PSM in order to show how platform-independent components manifest on the new platform or technology. A PSM metamodel should be defined (or refined) which can describe the new platform in an abstract way to make the creation of platform-specific models easier. Transformation rules are also needed for mapping

PIM metamodel elements to PSM metamodel elements. Applying them to an existing PIM will result PSMs for the new platform.

3.4 UML as Modeling Language

The MDA approach requires a language which uses formal definitions so the tools will be able to transform these models automatically. OMG recommends the use of UML to construct platform-independent models. The power of UML in our case is the modeling of the structural aspects of a system. This is done through the use of class diagrams which enables the generation of PSMs with all structural features.

3.5 UML Profiles

The semantics of UML models can be extended by applying UML profiles by adding stereotypes and tagged values which is a common way for building UML models for particular domains. These profiles are considered to reside in the metamodel layer in order to define UML "dialects" that can be used by models in the model layer. This mechanism is useful for both PIMs and PSMs.

For PIMs, a new profile should be defined to support a given design methodology. There are several profiles for that purpose [36] [10] [5], but they might not be appropriate for a different design strategy.

Many popular UML Profiles for specific platforms (e.g., UML Profile for EJB, UML Profile for CORBA, and so on) also exist but due to the extensibility mechanism new ones may also be created. This is how new platforms and technologies should be handled.

Therefore, UML Profiles might serve as a basis for defining a PIM–PSM transformation. This should be defined using the concepts of the metamodels used for describing PIM (source) and PSM (target).

4 New Results

We proposed a systematic design method for Web applications which takes into account the data-oriented aspects of these applications. We presented guidelines for the development of domain models adopted for this problem domain. Our method is based on an extension of the UML metamodel which is influenced by the paradigm of Domain Driven Design [37]. For the problem domain a custom Domain Specific Language is created [38] utilizing the concepts of roles. These domain model elements are mapped to a metamodel for our methodology`s structural model. An UML profile is derived by means of stereotypes and tagged values. The next step in our suggested development process is the construction of the navigational model. We provide a method to derive the navigation model from the structural model of a Web application based on the UWE [10] method. Using UML2 features we construct components from navigational entities in order to realize conceptual information provider elements. These elements can be connected with each other through interfaces to form an abstract page containing the necessary information. These pages could be transformed by the presentational layer to the desired format achieving universal client access.

To exploit the MDE development approaches [11] we utilize UML and XML technologies to realize code template generation from our models. Using standards (like OMG's XMI format) and recommendations from W3C (like XML Schema, XSLT and XForms) we are able to generate the data manipulation pages as well. The created artifacts make our method valuable for data-oriented Web application development.

4.1 Definition of a custom DSL

Following the guiding principle of the Domain Driven Design methodology we have created our custom Doman Specific Language for data-oriented Web Information Systems. The main advantage of our specific language is the ability to capture the concept of different roles in the problem domain. Using roles in the domain model is essential to outline the different behavior of the entities. If we deal with this characteristic we have the opportunity to precondition our design phase of the development process for the proper handling of modeling artifacts. We have

realized this behavior by extending the definition of the entity term. However, this extension requires the definition of the appropriate services in the entity as well.

The service and module definition is not modified, but one should think about the module as a boundary of coherent concepts. If we introduce roles in entities it is recommended to create a separate module for them with the required services for each aspect.

4.2 UML metamodel extension and Profile derivation

A metamodel provides a precise definition of concepts used in the modeling steps, including modeling elements, their relationships and well-formedness rules. Our goal is to extend the UML metamodel with the concepts of the problem domain. This is achieved by deriving custom classes and associations from the UML metamodel elements. We have established the relationship between the structural elements using the Role Object pattern. Moreover, a navigational metamodel is created by adopting the modeling elements of the UWE method.

In order to keep CASE tool support we derive the proper UML Profiles from these metamodels. The profiles are comprises the definition of stereotypes, tagged values and constraint about their usage. The modeling tools can check that a model conforms to a given profile.

4.3 Development Process

The first step during system design is the analysis of requirements to gather and formalize user requests. Using use cases and activity diagrams, we could determine the outline of the system and describe its fundamental functional aspects from the different users' viewpoints.

Actors represent different roles of users of the Web application. “Use cases” are used to describe available operations which are used for determining the users' activities. This is important because they serve as a bird's eye view of the possible actions appearing on the start page of the related user groups which might be performed by users belonging to the group.

Activity diagrams are based on use cases and they are used to describe business logic. Moreover, they could be used to derive program modules and code skeletons, as well. Of course, our case is much easier as the structural model is deeply bound to the data structure, and our focus is on the data management and manipulation tasks which should be performed.

The second step is the conceptual modeling of data structure and access paths of the application. This is achieved by UML2 class diagrams which apply our UML profile. At conceptual level, structural, navigational, component and presentational models are created. This approach is well- known in the literature. Several design methods (OO-HDM [28], WebML [6], UWE [29], Jim Conallen's WAE [23], etc.) follow more or less the same way.

After a platform-independent model has been developed, it is subject to a model transformation which results in a platform-specific model which serves as a starting point for code generation. Our method is illustrated in the next Figure.

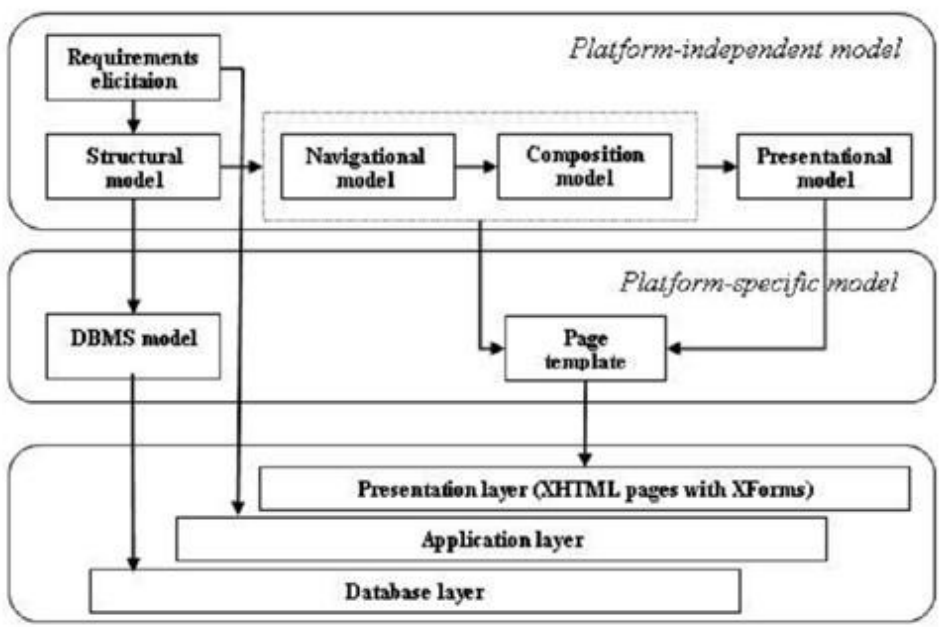


Figure 1:Phases of development

4.4 Models of the Design Phase

4.4.1 Structural Model

Main modeling elements of the structural model are classes, associations and packages. For a common Web application, use cases and activity diagrams are the base of the conceptual design of the domain.

However, in data-centric applications (which we consider), the structural model of the application domain is typically much more complex than any other models that have been mentioned previously. It is very important to create a class diagram which is of a good quality since many other activities (such as generating some primitive navigational elements like information access along associations) are relying on it.

4.4.1.1 The concept of roles

One of the advantages of our approach is the introduction of the roles in the structural model. There are situations when an object must provide different services (data, behavior) according to the context in which they are accessed. Like an academic accounting system which needs to track students and academics but a particular person may act sometimes as a student or sometimes as an academic. We use the role concept as a set of properties which are important for an object to behave in a particular context. In order to describe the different roles we have made use of the Role Object design pattern.

4.4.2 Navigational Model

The next stage in the development process is the navigational design. The navigational model specifies the elements of the structural model can be accessed from other parts of the application. In the navigation model's building process, the developer takes crucial design decisions such as what navigation paths are required to ensure the application's functionality. These decisions are based on the component model, use case diagrams and navigational requirements that the application needs to satisfy.

The navigational diagram is strongly connected to the structural one since it defines the navigational paths among structural model elements. In general, new

associations might be added for direct navigation to avoid navigation paths of length greater than one. However, there may be some classes in the structural model that are not subject to be a visiting target. Therefore, they should be omitted from the navigational diagram. The navigation inside the application mostly occurs along associations which are used to describe the relation between structural elements. Typically, these associations appear as either hyperlinks or menus in the user interface.

The navigational diagram supplements the structural model with some additional associations and classes representing different access structures such as menus, queries and indices.

4.4.3 Component model

The Component model is intended to define the basic modules of the system. These modules are the concepts that are described our domain specific language with the module definition. Modules may have services defined in the DSL, so they will expose the necessary interfaces to provide their functionality. Moreover of the entity definition, we use modules to join together the navigational classes with their navigation structures (like menus and indexes). This implies modules will expose the required interfaces for the navigation. These interfaces are determined by the navigational context. Naturally, the behaviors for the introduced navigational roles are also expressed by the components.

These components will serve the basis for the presentational model mapping between concepts of the structural model and Web pages. One should think of a component as an object which is associated with the entities of the structural model. Hence, a page can arbitrarily intermix information coming from multiple entities. Moreover, it is possible to extend the content with derived attributes or relationships.

An element (object) of the component model could be considered as the content of a page which should be rendered somehow (described by the presentational model) along with several navigational elements (given by using the navigational model). This actually provides a view over the structural model elements. It can be

extremely useful when different user groups are subject to access different page contents since several views might be defined on the same structural elements.

However, there could be more complex compositions such as when a supervisor's page should hold some pieces of information about his/her supervisees along with some data having statistical character (e.g., the number of the supervisor's refereed publications in the past 5 years). In such cases, the generated skeleton must be complemented with additional associations (to the appropriate classes of the structural model).

4.4.4 Presentational Model

The main goal of presentational modeling is to map the elements of the component model to some well-known GUI primitives. This task can also be applied at a conceptual level since it does not hold information about particular implementation. After the PIM–PSM transformation, some of those primitives might be replaced by others if the target platform does not support the given one (e.g., tree view of some hierarchical data might be replaced by a list with indented sublists). However, there are several presentational frameworks available we are not treated presentational aspects in this thesis except the generation of the data manipulation pages.

4.5 PIM–PSM transformations

Following the MDA principles we use several transformations to create PSMs from PIMs. For instance, at the structural model we use the "Hibernate" framework which provides an implicit PSM (relational model) so we does not deal with issues of this kind of model transformation. Composition and navigational models are used to formalize page templates along with presentational model. These templates are used to generate concrete pages using W3C's XForms standard for handling user inputs. The PSMs are represented with XML documents which allow XSLT to be used for the page generation from templates.

4.6 Code Generation

We use the Eclipse Modeling Framework to create the conceptual models. These models will serve the basis for the XML-based communication in the deployed application. When a user interacts with the system by filling and submitting forms, an XML document is created and passed to the server side. However, user-supplied data should be validated against the data model. Since XML documents' validation are based on any of the well-known XML schema languages (e.g., DTD, XML Schema, RelaxNG, Schematron), a schema was needed. A freely available Eclipse plug-in called hyperModel (http://www.xmlmodeling.com/hypermodel) is able to transform a UML2 class diagram into an XML Schema, therefore we applied it in our process. The generated XML Schema is one of the most important parts of our architecture since it is used as the base of the generation of XForms pages. On the other hand, it is applied for validating all the XML documents which are used for intra-system communication, as well. XForms pages might be embedded in more complex XHTML pages which conform to the component and navigational model defined at the conceptual modeling phase.

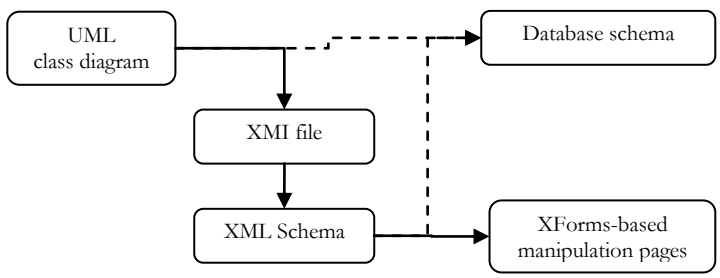


Figure 2: Code generation process

For creating models, OMG's XML Metadata Interchange (XMI) gives the possibility of a kind of tool-independence. As XMI is being an industrial standard for exchanging metadata of UML models in XML format, it is supported by all the major modeling tools so—in a collaborative environment—teams might use different tools. First of all, UML diagrams should be saved in an XMI-compliant format. Afterwards, XMI should be imported to Eclipse and let hyperModel do the transformation to XML Schema (.xsd). XForms pages are generated directly from the .xsd using an XSLT.

5 Conclusion

The presented approach describes a method to design and develop data-driven Web applications using MDA. We could see how complex it is therefore, these development steps are far from being systematic. We have elaborated a new methodology to help the development of Web applications rapidly and effectively. Our approach is based on UML and XML technologies supporting data management task in small- and medium-sized projects. We have added some important remarks concerning the implementation phase utilizing XML technologies to develop modular, scalable and expandable Web based systems. Ongoing researches can go in several interesting directions in the design and development phase. We are going to study the additional expandability of our UML-based methodology.

Obviously, the current state of our work could only be a part of a more comprehensive Web Information System development framework. Besides the modeling steps described in Section 4, it is also common to include modeling of personalization and presentation (Fraternali, 1999). In order to attain personalization, different participants of the system may need different viewpoints of the system's structure, navigation or presentation. These can be modeled using standard modeling notations which are big steps towards developing highly customizable and flexible systems. On the other hand, presentational aspects could be extended with current technologies like AJAX, OpenLaszlo, etc., only PIM–PSM transformations are required to define the mappings between abstract presentational models and concrete technologies.

Another possible research area is the measurement of models' quality. In order to improve system quality, several metrics should be defined and measured. There are several code metrics in the literature but only few of them are really useful. When applying model-driven development, even these code metrics might not be appropriate since the high majority of code is generated. As MDA development is based on creating models and transforming them into code, model quality should be measured instead of code quality. Therefore, model metrics are needed which are able to describe different properties of models. To obtain which possible metrics are worth measuring, further research is needed. Later on, statistical models should be applied to achieve higher quality. For the correct measurement we need to identify operational profiles and applicable statistical models to find the relevant factors and methods [22] [21].

Irodalomjegyzék

[1] Mendes, E., Mosley, N., "Web Engineering." Germany : Springer, 2006.

[2] Portal, Web Engineering Community., *http://webengineering.org/*. [Online] http://webengineering.org/.

[3] Gingeri A., Murgesan S., "The Essence of WebEngineering." IEEE Multimedia Vol. 8., 2004.

[4] Gamma, E., Helm, R., Johnson J. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston : Addision Wesley, 1995.

[5] Conallen, J., *Building Web Applications with UML*. 2nd Edition. Boston : Addision Wesley, 2002.

[6] Ceri, S., Faternali, P., Bongio, A., "Web Modeling Language (WebML)." In Proc. of 9th World Wide Web Conference, 2000.

[7] Leune, O. M. F. De Troyer and C. J., "WSDM: a user centered design method for Web sites." Proceedings of the Seventh International World Wide Web Conference, 1998.

[8] Mellor, S. J., Balcer, M. J., *Executable UML*. Indianapolis : Addision Wesley, 2002.

[9] Zhao, W., Kearney, D. and Gioiosa G., "Architectures for Web based Appliactions." AWSA : ismeretlen szerző, 2002.

[10] Hennicker, R., Koch, N., "A UML-based Methodology for Hypermedia Design." UML '2000 (LNCS 1939), 2000., old.: 410-424.

[11] Schmidt, D.C., "Model-Driven Engineering." IEEE Computer 39 (2), 2006., old.: 25-31.

[12] Kleppe, A, Warmer, J., Bast, W., *MDA Explained.* Boston : Addision Wesley, 2003.

[13] Object Management Group., Model Driven Architecture (MDA). [Online] 2001. július. http://www.omg.org/mda/.

[14] Mernik, M., Heering J. and Sloane A. M., "When and how to develop domain-specific languages." ACM Computing Surveys, hely nélk. : ACM, 2005., 37(4). kötet, old.: 316–344.

[15] Object Management Group., "Meta Object Facility 2.0 Core Specification." [Online] 2003. http://www.omg.org/docs/ptc/03-10-04.pdf.

[16] R. Soley et. al., Object Management Architecture Guide. [Online] 1997. http://doc.omg.org/ab/97-05-05.

[17] Object Management Group., "Unified Modeling Language." [Online] november 2007. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.

[18] —. *QVT MOF 2.0 Query/Views/Transformations RFP.* 2002. October.

[19] Jacobson I., Booch G., Rumbaugh J., *The Unified Software Development Process.* s.l. : Addision Wesley, 1999.

[20] Object Management Group., "Object Constraint Language." [Online] május 2006. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.

[21] Arató M., Fazekas G.,Kollár L., "Statistical measurement of software quality." Proc. of the 7th International Conference on Applied Informatics, 2007.

[22] Adamkó A., Arató M., Fazekas G., Juhász I., "Performance Evaluation of Large-scale Data Processing Systems." Proceedings of the 7th International Conference on Applied Informatics, 2007.

[23] Conallen, J., *Building Web Applications with UML.* Boston : Addision Wesley, 1999.

[24] Kruchten, P., *The Rational Unified Process: An Introduction.* USA : Addision Wesley, 1999.

[25] Fowler, M., Dealing with Roles. [Online] 1997. július 20. http://martinfowler.com/apsupp/roles.pdf.

[26] Riehle, D., "Describing and Composing Patterns using Role Diagrams." In Proc. Ubilab Conference, 1996., old.: 137-152.

[27] D Bäumer, D Riehle, W Siberski, M Wulf., "The Role Object Pattern." Proceedings of PLoP, 1997.

[28] Schwabe, D., Rosi, G., "An Object-Oriented Approach to Web-Based Application Design. Theory and Practice." TAPOS Vol. 4, 1998., old.: 207-225.

[29] Koch, N. and Kraus, A., "The expressive Power of UML based Web Engineering." Proceedings of the 2nd International Workshop on Web Oriented Software Technology, 2002.

[30] Reenskaug, T. M. H., *Models - Views – Controllers.* USA : Xerox PARC, 1979.

[31] Sun Microsystems, Inc., Sun ONE Architecture Guide. [Online] 2002. http://www.sun.com/software/sunone/docs/arch.

[32] Adamkó A., Bornemissza Cs., "Combining the Benefits of MVC Design Pattern and UML Based Modeling for Different Software Platforms." : Proceedings of the 7th International Conference on Applied Informatics, 2007.

[33] Gnaho, C., "Web-based Information Systems Development - A User Centered Engineering Approach." Lecture Notes in Computer Science, 2001., old.: 105-118.

[34] Scharl, A. és Gebauer J. and Bauer, C., "Matching Process Requirements with Information Technology to Access the Efficienty of Web Information Systems." Information Techology and Management 2(2), 2001., old.: 192-210.

[35] Holck, J., "4 Perspectives on WebInformation Systems." Proc. of the 36th Hawaii International Conference on System Science, 2003., old.: 265-275.

[36] Koch, N. and Kraus, A., "Towards a Common Metamodel for the Development of Web Applications." Proc. of the 3rd International Conference on WebEngineering (LNCS 2722), 2003., old.: 497-506.

[37] Evans, Eric., *Domain-Driven Design: Tackling Complexity in the Heart of Software.* hely nélk. : Addison Wesley, 2003.

[38] Mernik, M., Heering,J. and Sloane, A. M., "When and how to develop domain-specific languages." ACM Computing Surveys, 2005., old.: 37(4):316–344.

[39] Schwabe, D., Rosi, G., "Web Applications Models are more than Conceptual Models." Proc. of the International Workshop on Conceptual Modeling and the Web, 1999.

[40] Fowler, M., *UML Distilled, Third Edition.* : Addision-Wesley, 2003.

[41] Ceri, S., Fraternali, P. and Matera, M., "Conceptual Modeling of Data-Intensive Web Applications." hely nélk. : IEEE Internet Computing, 2002., 6(4). kötet.

[42] Distante, D., Rossi, G., Canfora, G., and Tilley, S., "A Comprehensive Design Model for Integrating Business Processes in Web Applications." In International Journal of Web Engineering and Technology (IJWET) : Inderscience Publishers, 2007., Vol. 2, Issue 1. kötet, old.: pp 43-72.

[43] Gomez, J., Cachero, C. and Pastor, O., "Extending a Conceptual Modeling Approach to Web Application Design." Proceedings of The 12th International Conference on Advanced Information Systems Engineering : Sprinter Verlag, 2000., LNCS 1789. kötet.

[44] Koch, N., Zhang, G., Escalona, M., j., "Model Transformations from Requirements to Web System Design." New York : ACM Press, 2006.

[45] Langham, M., Ziegeler, C., *Building XML Applications*. USA : Sams Publishing, 2002.

[46] Object Management Group., XML Metadata Interchange (XMI). *XML Metadata Interchange (XMI)*. [Online] Object Management Group, 2003. http://www.omg.org/.

[47] Schmid, H.A., Donnerhak, O., "OOHDMDA - An MDA Approach for OOHDM." In Proceedings of the 5th International Conference on Web Engineering : LNCS, 2005., 3579. kötet, old.: 569-574.

[48] L. Baresi, F. Garzotto, P. Paolini., "Meta-modeling Techniques meets Web Application Design Tools." Proc. of FASE 2002 : Springer Verlag, 2002., LNCS 2306. kötet, old.: 294-307.

[49] T. Berners-Lee, J. Hendler, O. Lassila., "The Semantic Web." *Scientific American*. 2001.

[50] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose., *Eclipse Modeling Framework*. Boston : Addison-Wesley, 2003.

[51] S. Ceri, P. Fraternali, M. Brambilla, A. Bongio, S. Comai, M. Matera., *Designing Data-Intensive Web Applications*. USA : Morgan Kaufmann, 2002.

[52] Rossi, G., Gordillo, S., and Distante, D., "Improving Web Applications Evolution by Separating Design Concerns." : IEEE Software Technology and Engineering Practice, 2005.

[53] Meliá, S., Gómez, J., Koch, N., "Improving Web Design Methods with Architecture Modeling." : 6th International Conference on E-Commerce and Web Technologies, 2005.

[54] World Wide Web Consortium., XML Schema. [Online] W3C, 2004. október. http://www.w3.org/TR/xmlschema-0/.

[55] —. Cascading Style Sheets. [Online] W3C, 2006. http://www.w3.org/Style/CSS/.

[56] —. Extensible Markup Language (XML) 1.1 (Second Edition). [Online] W3C, 2006. augusztus. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[57] —. XForms 1.1. [Online] W3C, 2007. november. http://www.w3.org/TR/xforms11/.

[58] —. XSL Transformations (XSLT) Version 2.0. [Online] W3C, 2007. január. http://www.w3.org/TR/xslt20/.

[59] —. XHTML™ 1.1 - Module-based XHTML - Second Edition. [Online] W3C, 2007. február. http://www.w3.org/TR/xhtml1.

[60] —. Web Services Description Language (WSDL). [Online] W3C, 2007. június. http://www.w3.org/TR/wsdl20-primer.

[61] —. XQuery 1.0: An XML Query Language. [Online] W3C, 2007. január. http://www.w3.org/TR/xquery/.

Függelék

Az UML metamodell kiterjesztése

A metamodell a modellezési elemek, a köztük fennálló kapcsolatok és jól formáltsági szabályok precíz definíciója. A metamodell határozza meg a modellek létrehozására használható nyelvet, és a modell a metamodell egy példánya. Az UML nyelv esetében (amely az OMG metamodell hierarchiájának második szintjén található) ez azt jelenti, hogy ha a metamodellben módosítunk, akkor a diagramjaink már az újonnan bevezetett szakterületi ismereteinket is meg tudja jeleníteni. Azonban a metamodell módosítása maga után vonja, hogy a támogató eszközök már nem fogják tudni helyesen kezelni az új nyelvet. Ennek érdekében a metamodellben leírt változásokat célszerű átalakítani az UML nyelv kiterjesztési mechanizmusa által támogatott UML profillá.

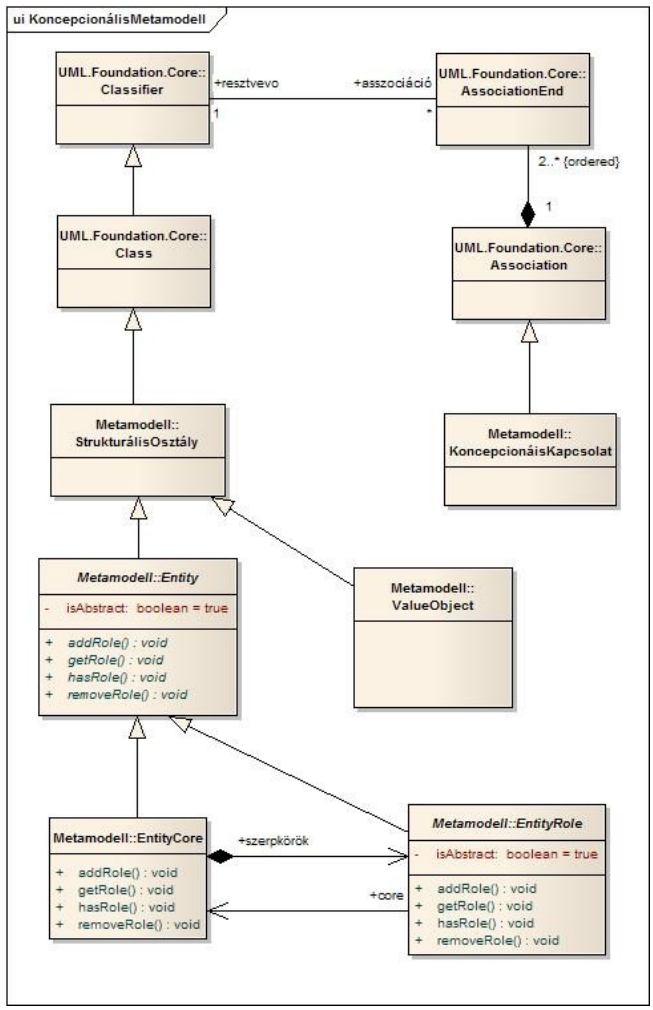
A mi esetünkben ez a kiterjesztési mechanizmus teszi lehetővé, hogy a koncepcionális és a navigációs modellek esetén megfogalmazott szakterület specifikus koncepcióinkat meg tudjuk jeleníteni. Az UML profilunk ennek megfelelően sztereotípiákat, kulcsszavas értékeket és megszorításokat fog tartalmazni.

Koncepcionális metamodell

A koncepcionális modellezés során a tartomány specifikus nyelvünkből készített CIM modellt szeretnénk PIM modellé alakítani. A bevezetett definíciók és összefüggések leírásához az UML metamodellt kell pontosítanunk, hogy megfeleljenek a koncepcionális modellezés fázisában használt elemeknek.

Ennek érdekében először bevezetjük a Szerkezeti osztályt, amelyet az UML nyelv Class osztályából származtatunk. Ez az új osztály fogja az alapot szolgáltatni a tartomány specifikus fogalmak és összefüggések modellezéséhez. Ebből fogjuk származtatni az entitásokat és az érték objektumokat. A szerepkörök modellezéséhez pedig tovább finomítjuk a modellt, hogy bármely entitás esetén megvalósítható legyen a szerepkörök leírása. A megvalósításához a Role Object tervezési minta

lapján járunk el. Azaz az entitásokat absztrakt osztállyal dogjuk modellezni, amely már tartalmazni fogja azon jellemzőket (most ezalatt értsük a metódusokat), amelyek szükségesek ennek eléréshez. Ebből az osztályból minden esetben származtatunk majd egy központi osztályt, amely az általános viselkedés leírásáért felelős, és ehhez kapcsolódhatnak majd a szerepköröket megvalósító osztályok kompozícióval. A következő ábra a metamodell kiterjesztésünket mutatja be.

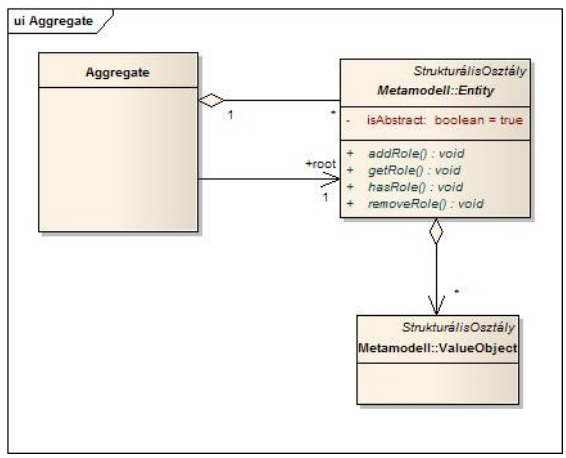


45. ábra: Koncepcionális metamodell

Annak érdekében, hogy az általunk bevezetett fogalmak helyesen működjenek, pás megszorítást kell tennünk. Egyrészt a strukturális osztályok csak az általunk bevezetett *KoncepcionálisKapcsolat*-on keresztül kapcsolódhatnak, annak érdekében, hogy biztosítsuk a saját osztályaink között a kapcsolatot. Ezt a következő OCL kifejezéssel írhatjuk le:

```
conext  KoncepcionálisKapcsolat
inv:    self.connection->size() = 2
inv:    self.connection.participiant->
                              forAll(oclIsKindOf(KoncepcionálisKapcsolat))
```

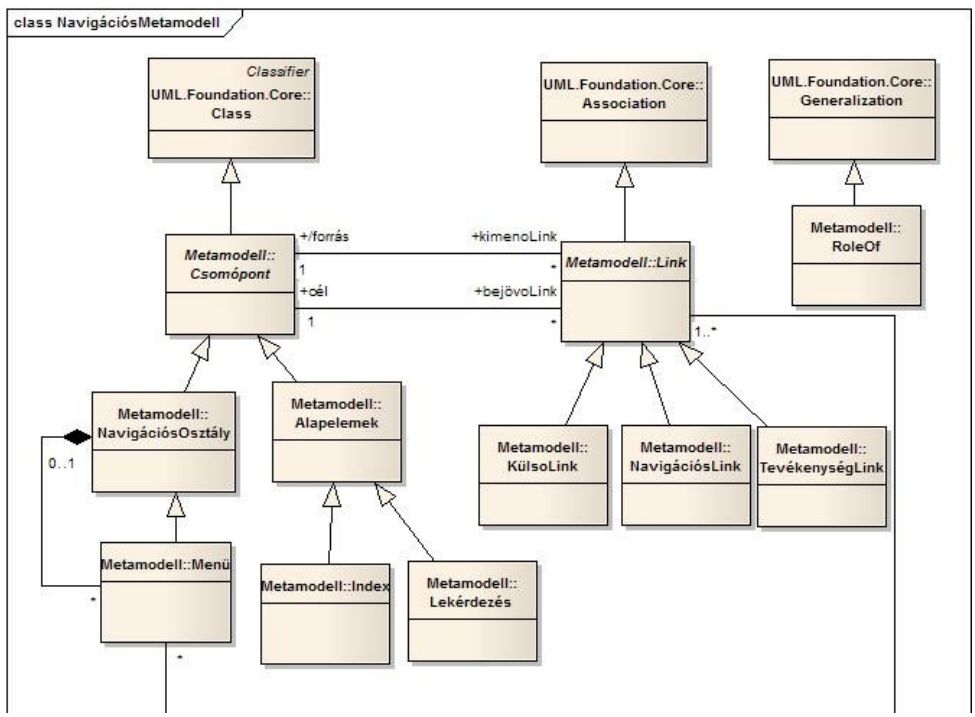
Természetesen a modellekben használhatjuk a tartomány alapú tervezés kiegészítő technikáit, mint például az aggregátorokat.



46. ábra: Aggregátor

Navigációs metamodell

A navigációs metamodell célja, hogy a navigációs tervezés során kialakulható összefüggéseket írja le. Az alapvető elemek a navigációs csomópont és a link. A metamodellben ezeket az UML Class és Asszociáció osztályokból fogjuk származtatni. Hasonlóan, mint a koncepcionális szintnél az entitás, itt a navigációs csomópontnak megfelelő osztály lesz absztrakt, ami azt jelenti, hogy csak a lentebb található specializált osztályok elemei példányosíthatóak. A navigációs alapelemek kialakításához a már korábban említett UWE [29] módszer által bevezetett elemeket és szemantikát használjuk fel.



47. ábra: Navigációs metamodell

A link osztály szintén absztrakt, amely a csomópontok összeköttetését írja. Egy forrás csomópontot köt össze a célcsomóponttal, ahogyan azt a ábrán két asszociáció is mutatja. Ennél a pontnál további megfontolásokat is figyelembe vehetünk, amelyek befolyásolhatják a linkek működését. Ilyen lehet például egy attribútum felvétele,

amely azt mondhatja, hogy a linket automatikusan követni kell, vagy éppen felvehető még egy plusz feltétel, ami a link alkalmazhatóságát befolyásolhatja. A linkek irányítottak, ezért egy kétirányú kapcsolat esetén azt fel kell bontani két irányított linkre.

A csomópont tovább bomlik két alosztályra, a navigációs osztályra és az alapelemekre. Az alapelemek határozzák meg, hogy az egyes csomóponthalmazokat hogyan tudjuk elérni. Ez lehet index, vagy lekérdezés, amelyek eredményeképp egy csomópont halmazt kapunk és ezek segítségével tudjuk folytatni egy adott csomópont felé a navigációt. A tartomány specifikus nyelvünkben kifejezett tárolók (*repository*) segítségével tudjuk majd az adott entitásokat elérni. További alapelemeket a navigáció tervezésénél vizsgáltunk. A navigációs osztály pedig nem más, mint egy koncepcionális osztálynak a származtatása. Ezt a kapcsolatot formálisan egy asszociációval fejezhetjük ki az adott koncepcionális osztály és a navigációs osztály között.

A menük specializált navigációs osztályok, amelyek egy navigációs osztály kifelé irányuló linkjeinek a meghatározását biztosítják. A menük kompozíció segítségével kapcsolódnak a navigációs osztályhoz, amint ezt az ábrán is jelöltük.

A navigációs linkek is tovább osztályozhatók. A külső linkek az alkalmazáson kívülre mutató hivatkozások. A navigációs link az alkalmazáson belül a navigációs osztályok közötti haladást teszik lehetővé, míg a tevékenység linkek az egyes helyeken elérhető feladatok végrehajtásához szükséges kifejezőeszközök. Ez a mi szövegkörnyezetünkben a szolgáltatásokhoz történő kapcsolódás.

Ábrajegyzék

Publikációs lista

I. Nemzetközi konferenciák referált kiadványában megjelent dolgozatok

1. Adamkó A.: *Design concepts for data intensive Web applications*, Proceedings of the 6th International Conference on Applied Informatics, Eger, Hungary, 2004., Volume II, pp. 9

2. Adamkó A.: *New methods in Web application modeling with UML and XML,* IEEE EuroCon 2005, Belgrade, Serbia, 2005., Proceedings, **IEEE** Catalog Number: 05EX1255C, ISBN: 1 4244 0050

3. Adamkó A.: *Rapid Web Application Development and Modeling, based on XML and UML technologies*, Proceedings of the 7th International Conference on Applied Informatics, Eger, Hungary, 2007.

4. Adamkó A., Arató M., Fazekas G., Juhász I.: *Performance Evaluation of Large-scale Data Processing Systems*, Proceedings of the 7th International Conference on Applied Informatics, Eger, Hungary, 2007.

5. Adamkó A., Bornemissza Cs.: *Combining the Benefits of MVC Design Pattern and UML Based Modeling for Different Software Platforms*, Proceedings of the 7th International Conference on Applied Informatics, Eger, Hungary, 2007.

6. Adamkó A., Kollár L.: *MDA-based Development of Data-Driven Web Applications*, Web Information Systems and Technologies (**WEBIST** 2008), Funchal, Portugal, 2008.

II. Nemzetközi konferenciák kiadványaiban megjelent dolgozatok

7. Adamkó A.: *Web Information Systems Engineering: problems and solutions*, CSCS 2004, The Fourth Conference of PhD Students in Computer Science Szeged, Hungary, July1 - 4, 2004., Volume of extended abstracts, page 18. Full paper submitted

8. Adamkó A., Bornemissza Cs.: *Planning and Developing Dynamic Web Sites in different platforms,* CSCS 2004, The Fourth Conference of PhD Students in Computer Science Szeged, Hungary, July1 - 4, 2004., Volume of extended abstracts, page 19. Full paper submitted

III. Nemzetközi folyóirat cikkek

9. *UML-based Modeling of data-oriented Web Applications*, Journal of Universal Computer Science, **Vol. 12** / **Issue 9** page 1104-1117, 2006

IV. Hazai konferenciák kiadványaiban megjelent dolgozatok

10. Adamkó A.: *E-business támogató CA rendszerek*, Informatika a felsőoktatásban 2002, Debrecen,Hungary, 2002. aug. 28-30., 7 oldal, http://www.date.hu/if2002, cd kiadvány

11. Adamkó A.: *Elektronikus információs és nyilvántartási rendszer a doktori iskolák fiatal kutatói részére*, Networkshop 2004, Győr, Hungary, 2004. április 5-7., http://nws.iif.hu, cd kiadvány

12. Adamkó A.: *Doktori Iskolák egy Információs Rendszere*, AgriaMédia 2004, Eger, Hungary, 2004. október 18-19., cd kiadvány

13. Adamkó A.: *Webalkalmazások modellezése új tervezési stratégiákkal,* DOSZ – Tavaszi Szél 2005, Debrecen,Hungary, 2005. május 5-8., konferencia kiadvány, 4-7

14. Adamkó A.: *Uml alapú adatcentrikus Webalkalmazások modellezése,* Informatika a felsőoktatásban 2005, Debrecen, Hungary, 2005. aug. 24-26., cd kiadvány

V. Lektorált oktatási segédanyag

15. Adamkó Attila: *Operációs rendszerek 1. Gyakorlati segédanyag*, mobiDIÁK könyvtár, 2004

    Frissítése és bővítése: 2008. tavasz

VI. Konferencia előadások:

magyar nyelven:

1. *E-business támogató CA rendszerek*, Informatika a felsőoktatásban 2002, 2002. augusztus 29.

2. *Elektronikus információs és nyilvántartási rendszer a doktori iskolák fiatal kutatói részére*, Networkshop 2004, 2004. április 6.

3. *Doktori Iskolák egy Információs Rendszere*, AgriaMédia 2004, Eger, 2004. október 19.

4. *Webalkalmazások modellezése új tervezési stratégiákkal*, Tavaszi Szél 2005, Debrecen, 2005. május 7.

5. *Uml alapú adatcentrikus Webalkalmazások modellezése*, Informatika a felsőoktatásban 2005, Debrecen, 2005. augusztus 25.

angol nyelven:

6. *Design concepts for data-intensive Web applications*, 6th International Conference on Applied Informatics, Eger, Hungary, 2004. január 27.

7. *Web Information Systems Engineering: problems and solutions*, CSCS 2004, The Fourth Conference of PhD Students in Computer Science, Szeged, 2004. július 2.

8. *Planning and Developing Dynamic Web Sites in different platforms*, CSCS 2004, The Fourth Conference of PhD Students in Computer Science, Szeged, 2004. július 2.

9. *New methods in Web application modeling with UML and XML*, IEEE EuroCon 2005, Belgrad, Serbia, 2005. november 22.

10. *Performance evaluation of large-scale data processing systems*, XXVI. Seminar on Stability Problems for Stochastic Models, Sovata, Romania, 2006. augusztus 28.

11. *Rapid Web Application Development and Modeling, based on XML and UML technologies*, 7th International Conference on Applied Informatics, Eger, Hungary, 2007. január 30.

12. *Performance Evaluation of Large-scale Data Processing Systems*, 7th International Conference on Applied Informatics, Eger, Hungary, 2007. január 31.

VII. Szakmai előadások

1. *Webalkalmazások modellezése*, Tanszéki szeminárium, Debrecen, 2005. április 19.
2. *Webes Információs Rendszerek fejlesztése*, V. Gyires Béla napok, Debrecen,
   2005. november 18.

VIII. Szoftver termék

A Debreceni Egyetem Matematika és Számítástudományok Doktori Iskolájának Webes Információs rendszere, 2004-2005, ~12.000 sor program kód + dokumentáció

Web alapú Információs Rendszerek modellezése

Értekezés a doktori (Ph.D.) fokozat megszerzése érdekében
az informatika tudományágban

Írta: Adamkó Attila okleveles programtervező matematikus

Készült a Debreceni Egyetem Matematika és Számítástudományok
doktori iskolája
(informatika programja) keretében

Témavezető: Dr. Fazekas Gábor

A doktori szigorlati bizottság:

elnök:	Dr. ……
tagok:	Dr. ……
	Dr. ……

A doktori szigorlat időpontja: 200… …….

Az értekezés bírálói:

	Dr. ……
	Dr. ……

A bírálóbizottság:

elnök:	Dr. ……
tagok:	Dr. ……
	Dr. ……
	Dr. ……
	Dr. ……

Az értekezés védésének időpontja: 200… . …… … .