SZAKDOLGOZAT

Pálfi Attila
Tar Dániel

Debrecen
2010

VÁLLALATIRÁNYÍTÁSI RENDSZER ASP.NET ALAPOKON

Témavezető:
Dr. Juhász István
egyetemi adjunktus
Juhász József Csaba
Development Leader

Készítette:
Pálfi Attila
Tar Dániel
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

1. Bevezetés

Szakdolgozatunkban egy kis- és középvállalatok által használható vállalatirányítási rendszer tervezési- és egyes részeinek implementálási folyamatát szeretnénk bemutatni.

Miért is vágtunk neki ennek a feladatnak? A piacon megtalálható, ilyen célú megoldások általában nagyvállalatok számára készülnek, összetettek és bevezetésük hatalmas költséget emészt fel, hosszú ideig tart, a legtöbb esetben az infrastruktúrát is fejleszteni kell, és a legfontosabb, a befektetett összeg nem minden esetben térül meg. Mivel ezek a termékek általános célúak, rengeteg olyan funkciót tartalmaznak, amelyeket nem használnak ki, és csak a dolgozók munkáját nehezítik. További probléma, hogy az átlagfelhasználók számára ezen programok felhasználói felülete nem intuitív, és nehezen használható. Gyakran csak angol nyelven érhetőek el, és nem felelnek meg a hatályos magyar jogszabályoknak, megnehezítve ezzel az egyszerű irodai alkalmazott, és a HR-es munkáját is, hiszen az esetleges változások miatt a szoftverfejlesztő cég segítségére van szükség.

Ezeket a problémákat figyelembe véve terveztünk meg egy olyan rendszert, ami átlátható felhasználói felülettel rendelkezik, és nem bonyolítja túl a funkciók átláthatatlan tengere. A rendszer a .NET-keretrendszer részét alkotó ASP.NET-en alapszik, MySQL-t alkalmaz adattárolásra, és NHibernate segítségével kommunikál az adatbázissal.

1.1. Vállalatirányítási rendszerek

A vállalatirányítási rendszerek (angolul enterprise resource planning, rövidítve ERP) olyan információs rendszerek, melyek a vállalat kommunikációs folyamatainak menedzselését, az üzleti tevékenységekkel kapcsolatos információk kezelését, valamint a humán erőforrással kapcsolatos feladatok adminisztrálását végzi.

Miért van szükség vállalatirányítási rendszerekre? Egy vállalat életében megszámlálhatatlanul sok üzleti folyamat, tevékenység és ezeket támogató rendszer szükséges, hogy az üzlet hatékonyan működjön. A folyamatok egy része manuális, jó része automatizált, de ilyen vagy olyan módon dokumentálásra kerül. Elég például a raktárkezelő programokra, számlázó és egyéb alkalmazásokra gondolni, amelyek egymásról mit sem tudva, függetlenül működnek, annak ellenére, hogy akár ugyanazon folyamat, különböző részeit vezérlik. Az ilyen „szigetszerűen” működő alkalmazások nem tudnak hatékonyan együttműködni, az inkonzisztens adatok pedig komoly anyagi és egyéb problémákat is okozhatnak. Célszerű tehát, hogy az összefüggő folyamatokat támogató szoftverek egymással tudjanak kommunikálni, a többszö-

rös adatbevitel elkerülésével növelve a pontosságot, és a felesleges munka megszüntetésével javítva vállalat működésének hatékonyságát. Valóéletbeli példa: egy vevői rendelés teljesítésekor, nemcsak a rendelés státusza és a vevő adatai módosulnak, hanem a készlet is csökken a kiszállított mennyiség függvényében, sőt termelő vállalat esetében a termelést befolyásoló alapanyagok rendelkezésre állása is módosulhat. Ha más-más szoftverek tárolják ezeket az információkat, akkor vagy egyenként kell mindegyikbe felvezetni a változásokat, vagy olyan köztes szoftvereket kell alkalmazni, melyek automatikusan, vagy félautomatikusan elvégzik ezeket a munkákat. A vállalatirányítási rendszernél nincs erre szükség, a különböző területeken dolgozó munkatársak különösebb erőfeszítés és plusz munka nélkül összehangolhatják tevékenységüket és döntéseiket.

Az információ az üzlet szempontjából komoly anyagi értékkel bír, abban az esetben is amennyiben téves, csak az ellenkező előjellel. Sok vállalt szenved az „igazság több verziójától", azaz a helytelen és inkonzisztens információk rengetegétől. Például ha a számlázási osztály és a vevőszolgálat két különböző, nem összehangolt adatbázist használ, akkor a vevő kérdésére helytelen választ is adhatnak. Ez a vevői elégedettségre károsan hat, rontva a vállalat megítélését.

1.2. A .NET-keretrendszer, az ASP.NET és a JavaScript

A rendszer megvalósításához a Microsoft által készített .NET-keretrendszert, és az ezen alapuló ASP.NET-et választottuk. A keretrendszer számos előnyös tulajdonsággal rendelkezik más programozási nyelvekkel és keretrendszerekkel szemben. Számos területre kiterjedő, az alapcsomag részét képező osztálykönyvtárral rendelkezik, így a mai alkalmazások fejlesztéshez szükséges leggyakrabban használt funkciókhoz nincs szükség külső (*third party*) könyvtárak csatolására. A .NET-keretrendszer teljesen Unicode-kompatibilis, ami elengedhetetlen az olyan nemzetközi körökben használt alkalmazásoknál, mint a vállalatirányítási rendszerek. A másik jelöltünk, a PHP épp emiatt esett ki, hiszen a jelenlegi (a dolgozat írásának idején még fejlesztés alatt álló 6-os előtti) verziói esetén külső könyvtárak telepítésére, és a konfigurációs fájlok módosítására van szükség, vagy kerülőmegoldások alkalmazására. Az ASP.NET további fontos jellemzője, hogy segítségével igen gyorsan lehet webalkalmazásokat fejleszteni; ezt az is segíti, hogy a szerveroldali kódot bármelyik .NET-nyelven (C#, Visual Basic, IronPython stb.) meg lehet írni. A Java-világ erényeként emlegetett platformfüggetlenség és eszközök ingyenessége ma már .NET-körökben is elérhető, hiszen a Mono-projekt keretében

fejlesztett, nyílt forráskódú XSP-kiszolgáló legújabb változata már képes az ASP.NET legutolsó, 3.5-ös verzióját használó projekteket is futtatni, valamint a frissen kiadott 4.0-ás implementálása is folyamatban van.

Az rendszer másik alapköve egy olyan eszköz, amit az ASP.NET kezdetektől fogva támogatott: ez a mai web egyik legdinamikusabban fejlődő technológiája, a JavaScript. A JavaScript egy kliensoldali programozási nyelv, ami lehetővé teszi a statikus HTML-lapok elemeinek módosítását, például a lap egyes vagy egész részének módosítását. Az elmúlt években hatalmas fejlődésen ment keresztül, és a böngészők legújabb verziói már azon versenyeznek, hogy melyik tudja gyorsabban futtatni a JavaScriptben megírt kódot. Népszerűsége is megnőtt a könnyű programozhatósága miatt, valamint hogy számos egyszerűen használható, nyílt forráskódú könyvtár jelent meg az utóbbi időben. Csak a néhány legnépszerűbb: jQuery, a MooTools, a YUI, az Ext, a Dojo, és még sorolhatnánk hosszan tovább. Mi az egyik legnépszerűbbet, a jQueryt választottuk, amelyet olyan széles körben látogatott oldalakon is használnak, mint a Twitter, az Amazon, az addons.mozilla.org, valamint a Microsoft és a jQuery fejlesztők közös munkájának köszönhetően az ASP.NET és a Visual Studio szerves része.

1.3. Vékony kliens-technológia

A vékony kliens-technológia alapötlete az, hogy az alkalmazásokat egy központi gépen futtatjuk, a felhasználók felé történő megjelenítésért pedig kis teljesítményű, „buta” számítógépek felelősek. Az üzleti logika tehát szerveroldalon történik megvalósításra, bár az illúzió megvan a felhasználók részéről, hogy asztali alkalmazást futtatnak. A vékonykliens architektúra nagy szerver- és hálózati terhelést jelent, hiszen minden számítás a szerveren történik, így számításkritikus feladatok, (pl.: CAD, illetve multimédiás alkalmazások) ellátására nem alkalmas. Olyan helyeken célszerű alkalmazni, ahol sok felhasználó dolgozik, kis teljesítményt igénylő programokkal. Jól alkalmazható, ha egy központosított rendszert kell elosztottá tenni. Tipikusan jó példa erre az irodai alkalmazások, illetve vállalatirányítási rendszerek.

Ezen esetekben a technológia számos előnye megmutatkozik:

- **Üzleti logika a szerveroldalon:** Az alkalmazás által használt adatok biztonságban vannak az adatbázisszerveren, amely gondoskodik a redundáns tárolásról, az adatok konzisztenciájáról. A program frissítése, módosítása esetén elég egy helyen elvégezni ezt a feladatot.

- **Alacsony költség:** A kliensek csak a megjelenítésért felelősek, ezért minimális méretű vassal is kiválóan ellátják a feladatukat, alacsony energiafogyasztásúak lehetnek, és mivel nem kell fejleszteni őket, akár több évre is tervezhetünk velük. (Akár merevlemez nélküli terminálokat is lehet alkalmazni.) Mivel webalapú technológiát használnak, csak a hálózati kapcsolatot kell tudni kezelnie, valamint egy webböngészővel kell rendelkeznie az operációs rendszernek, erre a célra pedig nagyon jó ingyenes megoldások is léteznek.
- **Skálázhatóság:** Amennyiben az alkalmazás több felhasználót kell hogy kiszolgáljon, vagy magasabb követelményeknek kell hogy megfeleljen, a szerveroldal skálázásával a teljesítményproblémák megoldhatók. Mivel webalapú a rendszer, a szervernek nem feltétlenül kell zárt hálózaton lennie, a kliensek távolról, akár interneten keresztül is kapcsolódhatnak hozzá.
- **Könnyű karbantarthatóság:** Mivel a kliens egy buta számítógép, amely semmilyen extra feladatot nem lát el a megjelenítésen kívül, extra program nincs telepítve rá, ezért meghibásodása esetén bármilyen számítógépet azonnal munkába tudunk helyette állítani, ezzel is csökkentve a technikai személyzet számát. Elegendő szakembert fenntartani a szerver számára.
- **Biztonság:** Zárt belső hálózat esetén az adatok teljes biztonságban vannak. Amennyiben a szervert interneten keresztül érik el a kliensek, a megfelelő biztonsági protokollok beiktatásával ez a mód is biztonságosan használható (pl. autentikációhoz biztonsági SMS küldése)

1.4. MySQL és a Connector/Net

Adattároláshoz a világ legnépszerűbb nyílt forráskódú relációs adatbázis-kezelő rendszerét, a MySQL-t választottuk. Olyan gyakran látogatott oldalak is használják adattárolásra, mint a Facebook, YouTube, Flickr vagy a Wikipédia, azaz rendkívül jól skálázható. Emellett még több okunk is volt a választásra: a MySQL üzembe állítása sokkal egyszerűbb más rendszereknél, teljesen ingyenes, korlátozások nélkül (ellentétben az Oracle adatbázis-kezelő rendszerével, melynek ingyenes változata esetén korlátozva van a rendszer által használható memória és tárhely). Másik fontos szempont az volt, hogy hivatalos .NET-es osztálykönyvtárral rendelkezik Connector/Net néven, melyet a MySQL aktuális fejlesztőcége (a dolgozat írásának idején ez az Oracle) készít. Harmadik szempont pedig, hogy nincs a Windows operációs rendszerhez kötve (ellentétben az SQL Serverrel), így ha az adatbázisrendszer számára

külön gépet jelölünk ki, azon futhat valamilyen ingyenes operációs rendszer is, pl. Linux vagy BSD. A fejlesztés során a 6.2.2-es változatot használtuk a MySQL 5.1.41-es verziójával együtt.

1.5. Az ORM-technológia és az NHibernate

Az objektumorientált gondolkodásmód célja a valós világ modellezése, középpontjában az objektumok állnak, és ezek egymáshoz való kapcsolatai. Az objektumnak van adat- és viselkedésmodellje, ami egymástól nem választható el. Azonban az objektumok a memóriában élnek, az alkalmazás futásának befejezése után megsemmisülnek. Az objektumok adatait ilyenkor kimentjük relációs adatbázisba, ami egy másik világ, majd amikor ismét szükség van az objektumra, akkor összeszedegetjük a morzsákként szétszórt adatokat és újragyártjuk az objektumot. (A relációs adatbázisok nagyon népszerűek, de sok kérdést felvetnek, pl. hogy az adatbázisban tárolt objektum viselkedését a tárolt eljárások vagy a program határozza-e meg, valamint hogy hol él az objektum.) Ez azért nem jó, mert koncepcionális szinten elválik az adat- és viselkedésmodell, továbbá megnehezíti a gondolkodást, a két paradigma eltérése, így megsokszorozza az emberi hiba előfordulását. Lehetőséget ad inkonzisztens állapot létrejöttéhez, adatvesztésre, egyéb adathibákra. Az ORM-rendszer segít leképezni az objektumokat az adatbázisba, így segít megmaradni az OO gondolkodási szintjén.

Sokkal hatékonyabban lehet fejleszteni, hogy ha nem kell mindkét paradigmában gondolkodni, valamint a relációs adatbázisok esetén nem léteznek az olyan fogalmak, mint az öröklődés, a polimorfizmus, a generikusok stb. Az objektum kimentésénél (azaz az objektum adattestének SQL-ekre való leképezése esetén) a séma megegyezik az esetek 90%-ban, így célszerű, ha ezt egy jól kiforrott rendszer végzi.

További előny hogy meghagyja a rendszert egy magasabb, adatbázis-független szinten, így a rendszer hordozható többféle adatbáziskezelő-rendszer között, nem kell SQL-lel foglalkozni.

A NHibernate egy ORM-technológia, feladata a .NET-osztályok leképezése adatbázistáblákká, valamint a .NET típusainak SQL-adattípusoknak való megfeleltetése. Az NHibernate SQL-parancsokat generál az objektumokkal kapcsolatos műveleteknek, és leképezéseknek megfelelően, megkímélve a programozót ezek „kézi” megvalósításától. Természetesen ennek az egyszerűségnek is van ára, a natív SQL-lekérdezésekhez képest jelentős teljesítménycsökkenés figyelhető meg egyes esetekben.

Az NHibernate használata nem ment fel minket teljesen az adatbázissal való közvetlen kommunikáció alól, hiszen nem előnyös ha egy lista bejárásakor, minden esetben az adatbázishoz fordul. Sok objektum összekapcsolása esetén pedig néhány esetben túlságosan is erőforrás-igényes, és összetett SQL-lekérdezéseket gyárt. Érdemes ilyenkor gyorsítótárat használni, illetve natív SQL-lekérdezéseket írni. A fejlesztés során az NHibernate 2.0-ás változatát használtuk.

1.5.1. Kapcsolatok kezelése az NHibernate-ben

Az UML módszertana 4 különböző viszonyt definiál az osztályok között:

- öröklődés
- asszociáció
- aggregáció
- kompozíció

Ezek közül az OO-nyelvek csak az öröklődés fogalmát ismerik, a másik három ismeretlen a számukra. Szerencsére az Nhibernate ismeri a kapcsolatok fogalmát, ám ami nem túl szerencsés, explicit módon csak az asszociációt. Ahhoz, hogy a többi kapcsolattípust is implementáljuk, olyan speciális asszociációként kell definiálnunk, melyek kétirányúak, és a szülő törlésekor a gyermekeket is töröljük.

1.6. Fogalomtár

árrés

A beszerzési ár és az eladási ár közötti különbség.

beszerzési ár

Az áru vételárát és a beszerzés során felmerülő további költségeket (pl. szállítási költség) együttesen tartalmazó ár.

kompozíció

Az aggregáció egy szigorúbb fajtája, egyes irodalmak erős aggregációként hivatkozzák. A kompozíció azt mondja, hogy a két objektum együtt alkot egészet, külön-külön nem létezhetnek, nincs értelmük. Tehát az így kialakult struktúrát csak egyben lehet törölni, azaz bármelyik résztvevő törlésével törlődik a másik is. Ez két dolgot jelent:

- A két objektumot egyszerre hozhatjuk csak létre, és kapcsoljuk őket össze. (Gyakorlatilag a "szülő" létrejöttekor létrejön a gyerek is, és mindkettő asszociációt hoz létre a másikra)
- Bármelyik törlésekor törli a másikat.

objektum

Az objektum olyan konkrét programozási eszköz, amely mindig egy adott osztályhoz kapcsolódóan jön létre. A létrehozés folyamata a példányosítás. Egy osztálynak akárhány példánya lehet, melyeknek van:

- címe
- állapotai
- öntudata

objektumperzisztencia

Az objektumperzisztencia egy objektumorientált adattárolási módszer, melynek lényege, hogy az objektumhierarchiát az objektumok számára transzparens módon mentsük ki (azaz ne vegyék észre, hogy kimentettük őket) relációs adatbázisba, és töltsük vissza onnan.

objektumidentitás

Az OO kimondja, hogy minden objektumnak van identitása, azaz egyedi. Mikor mondjuk azt, hogy két objektum ugyanaz? Ha az attribútumaik megegyeznek? A .NET objektumidentitás-fogalma nem teljesen egyezik meg a relációs adatbázisban használatos helyettesítő kulcs (*surrogate key*) fogalmával.

osztály

Az osztály az absztrakt adattípus implementációs eszköze (megvalósítja a bezárást vagy információelrejtést), maga is absztrakt eszköz, vannak attribútumai, melyek segítségével tetszőlegesen bonyolult adatmodellt lehet kezelni, valamint metódusai, amelyek a funkcionális modell megvalósítói.

öröklődés

Az öröklődés az újrafelhasználhatóság fontos eszköze, amely osztályok közötti aszimmetrikus viszont definiál. Mindig van egy ősosztály, és ehhez kapcsolódóan vele öröklődési viszonyban lévő alosztály, mely definiálása pillanatától átveszi a bezárás által meg-

engedett láthatósági szintű ősosztálybeli attribútumokat és metódusokat. Ezek mellett új attribútumokat és metódusokat definiálhat, az átvett eszközöket átnevezheti, a metódusokat újraimplementálhatja, megváltoztathatja a láthatóságot, azonos néven saját eszközöket definiálhat.

perzisztens objektum állapotai

Egy perzisztens osztály példányai három különböző állapotban lehetnek, amelyek a környezet perzisztenciájának megsértése nélkül lettek definiálva. Az NHibernate ISession objektum a perzisztens környezet:

- **tranziens**: a példány nem, és még soha nem volt kapcsolatban egyetlen perzisztens környezettel sem. Nincs perzisztens azonossága (elsődleges kulcs-értéke)
- **perzisztens**: a példány éppen kapcsolatban van egy perzisztens környezettel. Van perzisztens azonossága, és talán egy neki megfelelő sor az adatbázisban.
- **detached**: a példány már hozzá volt rendelve egy perzisztens környezethez, de azt a környezetet lezárták, vagy a példány már szerializálva lett egy másik folyamat számára. Van perzisztens azonossága, és talán egy neki megfelelő sor az adatbázisban.

kapcsolat

Az osztályok közt fennálló viszonyok.

kapcsolat számossága (multiplicitás)

A kapcsolat megvalósulásánál részt vevő egyedek számát szabja meg mindkét oldalon.

ORM-rendszer

Az Object-relational mapping egy programozási technológia amely konverziót biztosít az egymásra ortogonális relációs adatbázis és objektumorientált paradigma között. Segítségével az objektumok egyszerűen képezhetők le relációs adatbázisba.

integrált vállalatirányítási rendszer (ERP)

A vállalatok legfontosabb feladata a vállalkozás működtetéséhez szükséges technikai, műszaki és humánerőforrások folyamatos újratervezése. Ezt a feladatot ellátó egységes adatbázisrendszerrel működő informatikai rendszer az ERP rendszer.

likviditás

A likviditás a vállalat fizetőképességét mutatja meg, azaz hogy egy adott pillanatban, a fizetési kötelezettségeknek milyen mértékben tud eleget tenni. A likviditás a következő képlettel számítható:

$$\frac{\text{Pénzeszközök + Likvid értékpapírok +Tárgyeszközök}}{\text{Rövidlejáratú kötelezettségek}}$$

kifizetés

A vállalattal vagy partnerrel szemben fennálló fizetési kötelezettség kiegyenlített része.

bevételezés

A raktári készletbe történő bevezetés.

kivételezés

A raktári készletből történő kivezetés.

árumozgatás

A vállalat két raktára közti termékforgalom.

selejtezés

Selejtezésre kerülnek mindazon eszközök, készletek, amelyek rendeltetésszerű használatra alkalmatlanok vagy ha szavatossági idejük lejárt. A selejtet kivezetjük a készletből.

szállítólevél

A szállított áru feletti rendelkezési jogot megtestesítő/igazoló *okmány*/*bizonylat.*

Bizonylat

A gazdasági műveleteket, ill. megtörténtüket hitelt érdemlően igazoló okmány.

munkabér

A tényleges munkavégzésért járó díjazás, valamint a munkaviszonyra tekintettel a munkáltató teljesítménye alapján nyújtott díjazás, továbbá a munkavégzés nélküli időszakokra nyújtott bér.

alkalmazott

A munkáltató érdekkörében, a munkáltató által biztosított eszközökkel munkát végez, (munkaerejét hasznosítják), ezért munkabért kap.

beszállító

Olyan külső felek, akiktől a szervezet termékeket vagy szolgáltatásokat vásárol vagy akik ezen termékekkel, és szolgáltatásokkal szerződéses alapon ellátják a szervezetet.

megrendelő

A végső fogyasztó, ügyfél, megbízó, kedvezményezett vagy másik fél, aki termékeket vagy szolgáltatásokat vesz igénybe a szervezettől.

beszerzői megrendelés

A vállalat számára szükséges anyagok, áruk, külső forrásokból történő biztosítása.

számla

Az áru szállításakor, vagy szolgáltatás teljesítésekor a teljesítésről szóló elszámolást és az érte fizetendő összeget tartalmazó kereskedelmi okirat (bizonylat), amely szerint történik meg az ellenérték kifizetése.

leltár

A leltár a mérleg alapja. A vállalkozás saját eszközeit és forrásait tartalmazza. A bérelt, kölcsönvett, eladott és még elszállítatlan eszközöket, a vállalkozásnál tárolt idegen tulajdonú eszközöket nem szabad a saját leltárban szerepeltetni.

raktár

Olyan komplex létesítményként értelmezendő, amely az áruk minőségét és mennyiségét veszteség nélkül képes az áruk szükség szerinti be-, ki- és áttárolását.

2. A rendszerről általánosságban

2.1. A Bizonylat-koncepció

A fejlesztés legelső lépése egy olyan általánosított dokumentummodell megtervezése volt, amely segítségével minden egyes vállalati folyamatot le tudtunk írni néhány olyan adat kivételével, melyet nem lehet általánosítani. Amennyiben egy konkrét folyamatot leíró dokumentumnak további adatokat kell tárolnia, a későbbiekben ebből az általánosított dokumentummodellből származtatjuk. A modellnek több ok miatt kell általánosítottnak lennie. Első, hogy a folyamatok közötti információcserét megkönnyítse. Ilyen esetekben ugyanis anélkül is hozzáférhetünk a bennük tárolt adatokhoz, hogy ismernénk a dokumentum konkrét típusát.

Második, hogy a program fejlesztése és modulokkal való bővítése során nincs szükség újabb és újabb adatmodellek létrehozására, hanem a meglévőt felhasználva és adott esetben specializálva lehet leírni a folyamatot.

Ezt a koncepciót szem előtt tartva megvizsgáltuk, hogy a vállalati folyamatok adatai közül melyek azok, amelyek sok esetben szerepelnek. Ennek eredményeként született meg a rendszerben használt `Bizonylat` objektum. Voltak olyan attribútumok, amelyek maguktól adódtak. Ilyen a dokumentum létrejöttének dátuma és ideje, a dokumentumot rögzítő alkalmazott, a folyamat teljesítésének dátuma és ideje, valamint hogy jóvá van-e hagyva a dokumentum, és hogy ki hagyta jóvá. Voltak olyan attribútumok, melyeket kisebb gondolkodás után rá tudtunk húzni egy közös sémára, ilyenek voltak a folyamatok résztvevői, a forrás- és a célpartner. Egyes attribútumok nem jelentek meg mindegyik típusnál, azonban mivel nem kötelező megadni, így nem jelentenek problémát.

2.2. A Bizonylat-koncepció implementálása

Az általánosított `Bizonylat` objektum a következő attribútumokkal rendelkezik:

- `LetrehozasIdeje: DateTime`
  A dokumentum rendszerbe való rögzítésének az ideje.
- `FizetesHatarideje: DateTime`
  A kifizetések határidejét tartalmazó attribútum. Tájékoztató jellegű, a listázásoknál használható fel. Amennyiben olyan bizonylatról van szó, amihez nem tartoznak kifizetés tételek (pl. selejtezésről szóló bizonylat), ott nem használja fel a rendszer ezt az attribútumot.
- `TeljesitesIdeje: DateTime`
  Egy újabb tájékoztató jellegű időpont, amit a rendszer használ fel statisztikák készítéséhez. Azt jelzi, hogy az adott folyamatot mikorra kell teljesíteni, vagy amennyiben a `Teljesitve` attribútumnak már `True` értéke van, akkor a konkrét teljesítés idejét tartalmazza.
- `Elfogadva: Bool`
  Ahhoz hogy egy bizonylatot teljesíteni lehessen, vagy bármilyen kifizetést hozzá lehessen rendelni, először egy arra jogosult személynek jóvá kell hagynia. Az efféle jogosultságokat a dolgozat 4.1-es fejezetében tárgyaljuk.

A bizonylat objektum a következő kapcsolatokkal rendelkezik:

- `ForrasKliens`

  A kapcsolat típusa: aggregáció, számossága 1:1

  Az a kliens, aki a dokumentálandó folyamat során valamilyen szolgáltatást nyújt a `CelKliens` felé. Ez lehet kifizetés, megrendelés, árumozgatás, selejtezés stb., leszármazott osztály típusának megfelelően.

- `CelKliens`

  A kapcsolat típusa: aggregáció, számossága 1:1

  Azt a klienst jelenti, aki a dokumentálandó folyamat során valamilyen szolgáltatást vesz igénybe a `ForrasKliens`től. Ez lehet kifizetés, megrendelés, árumozgatás, selejtezés stb., leszármazott osztály típusának megfelelően.

- `Alkalmazott`

  A kapcsolat típusa: aggregáció, számossága 1:1

  Az az alkalmazott, aki rögzíti a rendszerbe a bizonylatot.

- `BizonylatTetel`

  A kapcsolat típusa: aggregáció, számossága 1:N

  Egy `BizonylatTetel` egy termékre vonatkozik, tartalmazza az eladási egységárat (kivételezési és bevételezési bizonylat esetén ez 0), valamint a mennyiséget.

- `Kifizetes`

  A kapcsolat típusa: aggregáció, számossága 1:N

  A bizonylathoz tartozó kifizetés, ami a bizonylat típusától függően lehet a Forrás illetve a Célpartner által teljesítve.

  - `Szamla`

    A kapcsolat típusa:aggregáció, számossága 1:1

    A bizonylathoz egy számla tartozhat, ez is csak akkor generálódhat, ha a bizonylathoz tartozó összes kifizetés összege, nagyobb vagy egyenlő mint a bizonylattételekben szereplő termékek értékének szummája. Mivel számlát csak egyszer lehet nyomtatni, a számla generálásakor csak akkor jön létre a számla objektum ha a nyomtatás sikeresen végrehajtódott.

  - `ElfogadoAlkalmazott`

    A kapcsolat típusa: aggregáció, számossága 1:1

    Olyan speciális jogkörrel rendelkező alkalmazott aki jogosult elfogadni az adott bizonylattípust.

A bizonylat objektum a következő metódusokkal rendelkezik:

- **AddBizonylatTétel**(BizonylatTétel): void
  A Bizonylat és a paraméterül kapott BizonylatTétel objektum között definiál kapcsolatot. (A Kifizetési bizonylat esetén nem használatos)
- **AddKifizetés**(Kifizetés): void
  A Bizonylat és a paraméterül kapott Kifizetés objektum között definiál kapcsolatot.
- **CreateSzámla**(): void
  Számlát generál a bizonylatból, ha a bizonylathoz tartozó összes kifizetés összege, nagyobb vagy egyenlő mint a bizonylattételekben szereplő termékek értékének szummája.
- **CloneBizonylat**(Bizonylat, bool): void
  A paraméterül kapott bizonylat (vagy leszármazottja) adatainak lemásolását végzi el, részletek a Bizonylat osztály konstruktorának leírásánál találhatóak meg.
- **SetTeljesítve**(): void
  Beállítja, hogy az adott bizonylat teljesítve van.
- **SetElfogadva**(Alkalmazott): void
  Beállítja, hogy az adott bizonylatot el lett fogadva, ezt csak a megfelelő jogosultsággal rendelkező alkalmazott végezheti.

Mivel az alkalmazásban az adatbázissal való kommunikációhoz az NHibernate-et használjuk, ezért szükség van egy mapping XML-fájl létrehozására (`Bizonylat.hbm.xml`), ami leírja az NHibernate számára a használandó osztály attribútumait.

```
<class name="Bizonylat">
  <id name="Id">
    <generator class="guid" />
  </id>
  <property name="ForrasKliens" />
  <property name="CelKliens" />
  <property name="Alkalmazott" />
  <set name="BizonylatTetelek" table="BizonylatTetelek" cascade="all-
delete-orphan">
    <key column="Id"/>
    <one-to-many class="VIR.BizonylatTetel, VIR"/>
  </set>
  <set name="Kifizetesek" table="Kifizetesek" cascade="all-delete-
orphan">
    <key column="Id"/>
    <one-to-many class="VIR.Kifizetes, VIR"/>
  </set>
  <property name="Szamla" />
  <property name="Elfogadva" />
```

```
    <property name="ElfogadoAlkalmazott" />
    <property name="LetrehozasIdeje" />
    <property name="FizetesHatarIdeje" />
    <property name="Teljesitve" />
    <property name="TeljesitesIdeje" />

    ...

</class>
```

A fenti példából ki lettek vágva a `Bizonylat` leszármazottjai, melyek a `Bizonylat`ot tartalmazó `<class>` elem `<subclass>` gyermekeiként jelennek meg a mapping XML-ben (további magyarázat a 2.3.1-es fejezetben található). A fenti kód magyarázata:

- Egy osztály implementációját egy `<class>` elem segítségével adjuk meg. A `name` attribútum tartalmazza a konkrét osztály nevét.
- Minden NHibernate-tel használt osztálynak rendelkeznie kell valamilyen elsődleges kulcsként szolgáló azonosítóval. Ez lehet egy növekedő egész szám (`Int32`, `Int64`), valamelyik az NHibernate által támogatott további típusokból (pl. az általunk használt `Guid`), de saját generátorosztályt is használhatunk a célra. Az `<id>` gyermekelemmel definiáljuk, és itt adhatjuk meg a generátort is (GUID-hez a `<generator class="guid" />` sort kell elhelyezni az `<id>`-n belül).
- Az egyszerű tulajdonságokat a `<property>` gyermekelemmel definiáljuk. Amennyiben nincsenek speciális követelmények, egyedül a `name` attribútumot kell megadni az adott tulajdonság nevével. (Az NHibernate lehetőséget biztosít egyedi oszlopnevek hozzárendelésére is, valamint explicit típusmegnevezésre.)
- A mapping XML-ek egyik legösszetetteb része a kapcsolatok leírása. Az NHibernate nem támogatja natívan az aggregációk és kompozíciók definiálását, azonban a leírónyelv eszközeivel meg lehet őket valósítani. Ez a `<set>` elemben történik. Ahol a mapping XML magyarázata során először feltűnik egy adott kapcsolattípus, ott leírjuk a kapcsolattípus magyarázatát is.

Ha közelebbről megnézzük a `Bizonylat` osztály implementálását, láthatjuk, hogy semmi bonyolult dolog nem történik, még a kapcsolatok kiépítésekor sem. Mindenről gondoskodik az NHibernate. A tulajdonságok definiálása azon a néven történik, amelyen a mapping XML-ben megadtuk őket:

```
public virtual DateTime LetrehozasIdeje { get; set; }
```

Kapcsolatok esetén egy kicsit bonyolultabb a helyzet. Mivel az aggregációt explicit módon nem ismeri az NHibernate, implementálása, annak definíciója alapján történik. Aggregáció olyan rész egész viszonyt kifejező kapcsolattípus, mely azt mondja ki, hogy a gyerek nem létezhet a szülő nélkül, fordítva viszont nem igaz. Ez két dolgot jelent:

1. A gyermek objektum létrejöttekor azonnal kapcsolódnia kell a szülőhöz. Ezt a gyermek konstruktorában tehetjük meg.

```
public BizonylatTetel(Bizonylat szulo, Termek termek,
      double mennyiseg,double egysegar)
{
  EgysegAr = egysegar;
  Mennyiseg = mennyiseg;

  this.Termek = termek;
  this.SzuloBizonylat = szulo;
  szulo.AddBizonylatTetel(this);

  Termek = termek;
}
```

2. A szülő törlésekor az összes gyereknek törlődni kell. A szülő osztály mapping XML-jébe a kapcsolatot reprezentáló `ISet<T>` tulajdonságnak adni kell egy `cascade="all-delete-orphan"` értéket. Ez arról rendelkezik, hogy a szülő törlésekor, törli a szülő nélküli entitásokat.

```
<set name="BizonylatTetelek" table="BizonylatTetelek"
    cascade="all-delete-orphan">
  <key column="Id"/>
  <one-to-many class="VIR.BizonylatTetel, VIR"/>
</set>
```

A másik megoldás ha megadjuk `on-delete` attribútumnak, hogy kaszkádolt törlés van, azaz törölje az összes gyermeket.

```
<set name="Alkalmazottak" inverse="true" cascade="save-update">
  <key name="FelhasznaloiCsoport" on-delete="cascade"/>
  <one-to-many class="Alkalmazott"/>
</set>
```

Az osztály implementációjánál így jelenik meg az attribútum:

```
public virtual ISet<BizonylatTetel> BizonylatTetelek { get; set; }
```

A `Bizonylat` két publikus, valamint egy nem publikus konstruktorral rendelkezik. A nem publikus konstruktor létrehozásának célja az volt, hogy kiemeljük egy közös metódusba

azokat az inicializálási feladatokat, melyeket minden egyes bizonylat létrehozásánál el kell végezni:

```
protected Bizonylat()
{
  this.LetrehozasIdeje = System.DateTime.Now;
  Teljesitve = false;
  Elfogadva = false;

  BizonylatTetelek = new HashSet<BizonylatTetel>();
  Kifizetesek = new HashSet<Kifizetes>();
}
```

A konstruktor első három sorában található értékadások egyértelműek: automatikusan beállítódik a létrehozás ideje a konstruktor futtatásának időpontjára, és mivel a bizonylat létrejöttekor még nincs elfogadva és teljesítve, ezek az értékek `False`-ra lesznek állítva. A másik két utasítás az `ISet<T>` generikusokat megvalósító tulajdonságok inicializálására szolgál, esetünkben `HashSet<T>`-et használunk.

A másik két konstruktor a bizonylat létrehozásának kétféle körülményét reprezentálja. Az egyik, amikor teljesen új bizonylatot hozunk létre:

```
public Bizonylat(Alkalmazott alkalmazott, Kliens cel_kliens,
      Kliens forras_kliens, DateTime FizetesHatarIdeje)
  :this()
{
  this.FizetesHatarIdeje = FizetesHatarIdeje;

  this.Alkalmazott = alkalmazott;
  alkalmazott.Bizonylatok.Add(this);

  this.ForrasKliens = forras_kliens;
  forras_kliens.ForrasKliensBizonylatok.Add(this);

  this.CelKliens = cel_kliens;
  cel_kliens.CelKliensBizonylatok.Add(this);
}
```

A paraméterül kapott fizetési határidő egyszerű értékadás, emellett a kapcsolatok beállítása történik: mind a szülőoldalon, mind a másik oldalon feljegyezzük az új kapcsolatot a forrás- és célkliensre, valamint a bizonylatot létrehozó alkalmazottra vonatkozóan.

A másik publikus konstruktor arra az esetre van, ha egy bizonylatot egy másik bizonylat alapján hozunk létre:

```
public Bizonylat(Alkalmazott alkalmazott, Bizonylat b)
  :this()
{
```

```
    CloneBizonylat(b);

    this.Alkalmazott = alkalmazott;
    alkalmazott.Bizonylatok.Add(this);
}
```

A recept egyszerű: meghívjuk a `CloneBizonylat` metódust a paraméterül kapott `Bizonylat` objektummal paraméterként, valamint beállítjuk a bizonylatot létrehozó alkalmazottat.

Megjegyzés: az aggregációt az életciklus-módszerrel is meg lehet oldani. A perzisztens osztályok megvalósíthatják az `ILifecycle` interfészt, ami olyan metódusokat specifikál, amelyek segítségével megvalósíthatjuk azokat a feladatokat, amiket az objektum létrehozásakor, törlésekor, illetve módosításakor el kell végeznünk. (A relációs adatbázis triggereihez hasonlatosak)

```
public interface ILifecycle
{
    LifecycleVeto OnSave(ISession s);
    LifecycleVeto OnUpdate(ISession s);
    LifecycleVeto OnDelete(ISession s);
    void OnLoad(ISession s, object id);
}
```

- `OnUpdate` - Az objektum módosításakor hívódik meg
- `OnSave` - Az objektum mentésekor hívódik meg
- `OnDelete` - Az objektum törlésekor hívódik meg
- `OnLoad` - Az objektum betöltésekor hívódik meg

`Az OnSave()`, `OnDelete()` és `OnUpdate()` metódusokkal a kaszkádolt törlések és egyéb műveletek valósíthatók meg, azaz az objektum törlésével az aggregáltak, és kompozíciók törlése.

`OnLoad()` metódus az ideiglenes (transient) tulajdonságok beállítására szolgál, amikor az objektum az adatbázisból betöltésre kerül.

Az `OnLoad()`, `OnSave()` és `OnUpdate()` metódusok arra is használhatók hogy az aktuális `ISession` objektumra eltároljanak egy referenciát, későbbi használat céljából.

Az `OnUpdate()` nem hívódik meg mindíg amikor az objektum perzisztens állapota frissül. Csak akkor hívódik meg ha egy ideiglenes objektumot (transient) az `ISession.Update()` metódusnak adódik át.

Ha `OnSave()`, `OnUpdate()` or `OnDelete()` `LifecycleVeto.Veto` értékkel tér vissza, a művelet hatásai érvénytelenítődnek. Ha `CallbackException` kivétel dobódik, a művelet hatásai érvénytelenítődnek, a kivétel az alkalmazásnak továbbadódik.

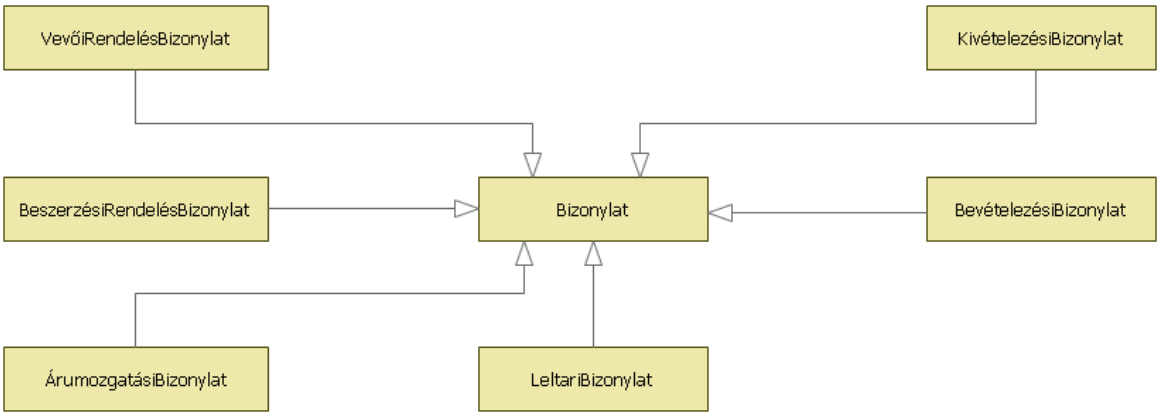
Az `OnSave()` az után hívódik meg hogy azonosító kapcsolódott az objektumhoz,kivéve ha idegen kulcsot (native key) használtunk.

2.3. A `Bizonylat` kiegészítő osztályai

Amint a Bizonylat osztály implementációjánál látni lehetett, számos kiegészítő osztály tartozik hozzá, melyek különböző adatokat tárolnak. Ezek az osztályok kerülnek alább bemutatásra.

2.3.1. A `Bizonylat` leszármazottai

A rendszer magjának legfontosabb részét a `Bizonylat` osztály leszármazottjai adják. A rendszer magjában található leszármazottak az alábbi ábrán láthatóak:



Az adott bizonylat tárgyalása abban a fejezetben történik, amely tárgyalt modulhoz kapcsolódik. A leszármazott osztályok reprezentálása NHibernate-ben a következő módszer szerint történik: A két osztály közt fennálló öröklődési viszonyt is a mapping XML-ben tudjuk megadni, úgy, hogy a leszármazott osztályt `<subclass>` elemként definiáljuk az ősosztály definícióján belül. Attribútum segítségével megadhatjuk, hogy a leszármazott osztály esetében csak az új attribútumokat tárolja külön táblában, vagy az összeset, azaz az ősosztályét is. Egy leszármazott definiálása a következőképpen néz ki:

```
<class name="Bizonylat">
  ...
```

```
  <subclass name="BeszerzoiRendelesBizonylat">
    <property name="BevetelezesiBizonylat" />
  </subclass>

</class>
```

2.3.2. Kifizetes

A `Kifizetes` osztály tárolja az adott Bizonylathoz tartozó pénzmozgásokat. A név kicsit csaló, hiszen nem csak a vállalat által mások felé történt tranzakciókat fedi le, hanem azokat is, melyek a vállalat felé történnek. Legfőbb attribútumai a következők:

- `Datum`: a fizetés (vagy tranzakció) végrehajtásának ideje (típusa `DateTime`)
- `Osszeg`: a kifizetés során átmozgatott pénzösszeg (típusa `Double`, értéke mindig pozitív a pénzmozgás irányától függetlenül)
- `Penznem`: annak a pénznemnek a hárombetűs azonosítója, amelyben a pénz érkezett, vagy elküldésre került (típusa: `String`, pl. HUF, EUR, USD)
- `Arfolyam`: a megadott pénznem árfolyama a rendszerben alapértelmezettként használt árfolyamhoz képest, a kifizetés rögzítésekor (típusa: `Double`)
- `KifizetesiMod`: a kifizetés módja, pl. bankkártyás fizetés vagy készpénz (típusa: `KifizetesiMod`, ami egy enum típus)
- `Azonosito`: a kifizetéshez kapcsolódó azonosító, például a banki tranzakció száma (típusa: `String`)
- `Alkalmazott`: a kifizetést a rendszerbe rögzítő alkalmazott (típusa: `Alkalmazott`, leírását lásd a 2.3.2-es fejezetben)
- `Típus`: a pénzmozgás típusa, azaz hogy ki- vagy befizetés történt (típus: `FiztesTipus`, ami egy enum típus)

2.3.3. Szamla

A `Szamla` osztály objektumai a vállalat által kiadott, valamint a vállalathoz beérkezett számlák olyan kiegészítő adatainak tárolására szolgál, melyeket nem tudunk magában a bizonylatban tárolni. Mivel a számlán szereplő adatok nagy része megegyezik azon bizonylatban tárolt adatokkal, melyből a számlát készítjük, így csak a következő adatokat tároljuk benne:

- `Azonosito`: a számla egyedi azonosítója (típusa: `String`)
- `KiadasDatuma`: a számla elkészítésének dátuma (típusa: `DateTime`)
- `SzuloBizonylat`: az a bizonylat, amihez a számla tartozik (típusa: `Bizonylat`)

Amennyiben egy `Bizonylat`-objektum már rendelkezik hozzá tartozó számlával, az azt jelenti, hogy a hivatalos számlát már kiadtuk az illetékes személyeknek. Ekkor már csak számlamásolatot tud nyomtatni a rendszer.

2.3.4. `Kliens`, `Partner`, `Alkalmazott`

A vállalattal kapcsolatban álló személyek közös, absztrakt ősosztálya a `Kliens`. Ez tartalmaz minden olyan attribútumot, ami közös a vállalat alkalmazottai és a vállalat partnerei esetén:

- `Nev`: a kliens teljes neve (típusa: `String`)
- `Telefonszam`: a kliens telefonszáma (típusa: `String`)
- `Email`: a kliens e-mail címe (típusa: `String`)
- `Adoszam`: a kliens adószáma (típusa: `String`)
- `Szamlaszam`: a kliens bankszámlaszáma (típusa: `String`)

A kliens kapcsolatai:

- `Cim`: a klienshez felvett lakcím vagy telephely
- `ForrasKliensBizonylatok`: azon bizonylatok halmaza, melyeknél a kliens forráskliensként szerepel.
- `CelKliensBizonylatok`: azon bizonylatok halmaza, melyeknél a kliens célkliensként szerepel.

A kliens egyik leszármazottja a `Partner`. A neve alapján kiderülhet, hogy a vállalat egy partnerét reprezentálja. A `Kliens` attribútumait és kapcsolatait a következőkkel egészíti ki:

- `Tipus`: a partner típusa, tájékoztató jellegű információ (típusa: `String`)
- `SzallitasiCim`: a partnerhez kapcsolódó szállítási cím.

A `Kliens` másik leszármazottja az `Alkalmazott`, aki a vállalat dolgozója, valamint egyben a rendszer felhasználója is. A személyes adatokon felül így a rendszerben való azonosításhoz szükséges információk is csatolva vannak az osztályhoz:

- `SzulDatum`: az alkalmazott születési dátuma (típusa: `DateTime`)
- `SzigSzam`: az alkalmazott személyigazolványának száma (típusa: `String`)
- `TajSzam`: az alkalmazott TAJ-száma (típusa: `String`)
- `FelhasznaloNev`: az az azonosító, amit a felhasználó ad meg a rendszerbe való bejelentkezéskor (típusa: `String`)

- `Jelszo`: a rendszerbe való bejelentkezéshez használt jelszó SHA-1 hashe (típusa: `String`)
- `Aktiv`: ez az attribútum adja meg, hogy a felhasználó aktív-e, azaz használhatja-e a rendszert (típusa: `Bool`)

Az Alkalmazott osztály kapcsolatai

- `FelhasznaloiCsoport`: az a felhasználói csoport, melynek tagja az alkalmazott. Kötelező kapcsolat, egy felhasználó egy, és csakis egy csoportba kell, hogy tartozzon.
- `BerezesiSzabaly`: az alkalmazott bérezésekor alkalmazott szabály. Nem kötelező megadni, ekkor a bérezés során az alkalmazott felhasználói csoportjának bérezési szabálya kerül alkalmazásra.
- `Munkanapok`: az alkalmazotthoz kapcsolódó `Munkanap`-obejktumok, melyek azt írják le, hogy az alkalmazott mikor végzett tevékenységet a vállalatnál.

Az Alkalmazott osztály egyetlen érdekes metódusa a `GetBerezesiSzabaly`, aminek célja, hogy visszaadja az alkalmazotthoz tartozó aktuális bérezési szabályt. Amennyiben az Alkalmazott rendelkezik saját bérezési szabállyal, azt adja vissza, egyébként az alkalmazott felhasználói csoportjához tartozó bérezési szabályt.

2.3.5. `Cim`

Adattároló osztály, egy cég, személy vagy raktár címét tárolja. Attribútumai mind String típusúak: `Orszag`, `Varos`, `Iranyitoszam`, `Utca`, `Hazszam`.

2.3.6. `BizonylatTetel`

A `Bizonylat`okhoz a termékeket a `BizonylatTetel` objektumain keresztül kapcsoljuk hozzá. Ezen burkolóosztály létrehozására több ok miatt volt szükség. Mivel a termék ára gyakran, akár napról napra változhat, ezért a `Bizonylat` létrejöttekor rögzíteni kell, hogy a terméknek mennyi volt az ára. Továbbá el kell tárolnunk a termék mennyiségét is. Mivel a termék törzse nem változik, így nincs arra szükség, hogy a termék törzséről ujjlenyomatot készítsünk, csupán a változó adatokat kell eltárolnunk – ilyen az ár. Attribútumai az `EladasiAr`, az `EgyegAr` és a `Mennyiseg` (mindkettő `Double` típusú), kapcsolata a `Termek`kel van, ami asszociáció.

Az objektumok közti asszociáció implementálásához a szülő osztályban attribútumként deklarálni kell a kapcsolatot, a számosságnak megfelelően. A kapcsolat megfelelő oldalán álló

osztály N számossággal vesz részt a kapcsolatban, akkor az `ISet<T>` generikus interfészt megvalósító osztályt használunk (ez a `System.Collections.Generic` névtérben található, a .NET keretrendszer 4.0-ás változatának újdonsága), 1-es számosság esetén a megfelelő típust kell megadni. A kapcsolatot tulajdonságokon keresztül lehet elérni. A mapping XML-be a kapcsolathoz a számosságnak megfelelően kerül be a `many-to-one`, a `one-to-many` vagy a `one-to-one` gyermekelem. Az alábbi példában a `BizonylatTetel` asszociációja látható, ami one-to-one-típusú, így elég egy attribútumként deklarálni:

```
<class name="BizonylatTetel">
  <id name="Id">
    <generator class="guid" />
  </id>
  <property name="EgysegAr" />
  <property name="EladasiAr" />
  <property name="Mennyiseg" />
  <property name="Termek" />
</class>
```

2.3.7. Termek

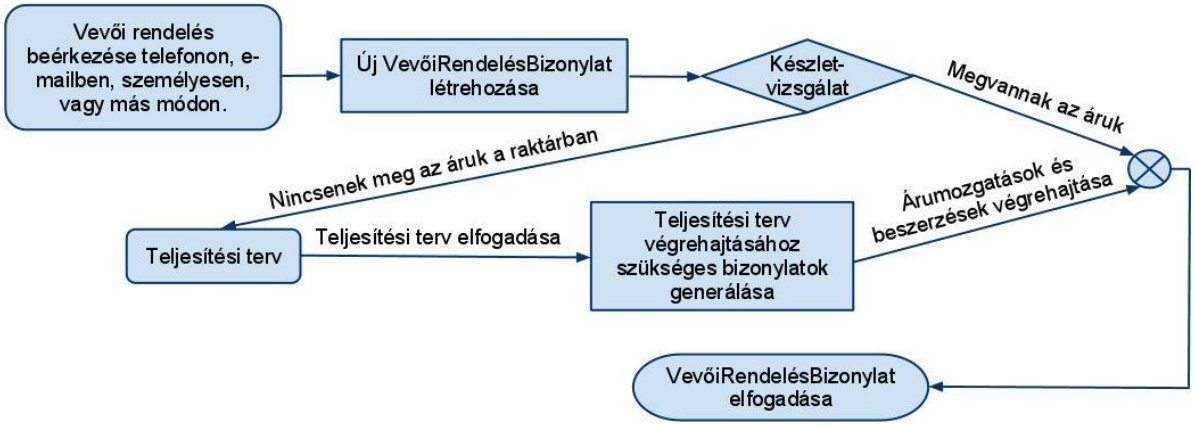
Adattároló osztály, egy termék információit tárolja. Attribútumai: `Nev` (`String`), `EgysegAr` (`Double`), `Mertekegyseg` (`String`), `Vonalkod` (`String`), `Gyorskod` (`String`), `Termekkod` (`String`), `AfaKulcs` (Int32), `Keszletminimum` (Int32), `Vamtarifaszam` (`String`)

3. A rendszer moduljai

3.1. Vevői és beszerzői megrendelések

Ez a rendszer egyik legfontosabb modulja, a rendelések nyilvántartásáért felelős. Két fő bizonylattípust kezel: a cég felé irányuló vevői megrendeléseket, és a cég által külső cégek felé irányuló beszerzői megrendeléseket.

3.1.1. Vevői rendelések



A vevői rendelések feldolgozása a rendszerben az alábbi folyamat szerint történik:

A vevők által személyesen, e-mailben, telefonon vagy más módon eljuttatott megrendeléseket egy megfelelő jogosultságokkal rendelkező személy rögzíti a rendszerbe. A rögzítés alkalmával egy új `VevoiRendelesBizonylat` jön létre, ami tartalmazza a megrendelt termékeket, azok mennyiségét és az árukat.

A frissen létrejött `VevoiRendelesBizonylat` még csupán a rendelési szándékot rögzíti. Ahhoz, hogy tényleges árukiadás történjen, a `VevoiRendelesBizonylatot` el kell fogadni. Az elfogadáshoz egyrészt megfelelő jogosultság kell (ami nem egyezik meg a bizonylat létrehozásához szükséges jogosultságokkal), másrészt lennie kell egy olyan raktárnak, ahogy egyben ki tudjuk elégíteni a rendelési igényt.

Amennyiben nincs ilyen, a rendszer egy árumozgás- és beszerzési rendelési javaslatokat tartalmazó teljesítési tervet készít. Amennyiben a hiányzó mennyiségek más raktárakban megtalálhatóak, felajánlja, hogy árumozgatások segítségével azok a megfelelő helyre, esetünkben a raktárba, ahonnan ki akarjuk adni a termékeket, legyenek szállítva. Emellett azon beszállítókat is felajánlja, akiktől már korábban rendeltük az adott terméket. Ebből a listából lehet be-

szerzői rendeléseket generálni. A teljesítési terv elfogadása esetén a rendszer legenerálja a megfelelő bizonylatokat.

Ha minden termék rendelkezésre áll, a VevoiRendelesBizonylatot is el lehet fogadni. Ekkor az adott raktárnál a termék mennyiségéből lefoglalásra kerül a kívánt mennyiség.

Abban az esetben, ha a megrendelő személyesen átveszi a termékeket, vagy a szállító érte jön, KiveteliBizonylatot generálunk, továbbá adott esetben szállítólevelet, így jelezve, hogy a termék fizikailag is ki lett adva a raktárból. Teljesítésének feltétele a hozzá kapcsolódó KiveteliBizonylat teljesítettsége.

A Bizonylat osztályból származó VevoiRendelesBizonylat a következő attribútumokkal és kapcsolatokkal egészíti ki az ősét:

- Új kapcsolatként jelenik meg a szállítási cím (aggregáció, 1:n számosságú), és egy KiveteliBizonylat (ez is aggregáció, 1:1 számosságú). A kivételezési bizonylat a VevoiRendelesBizonylatból generálható automatikusan a CreateKivetelezésiBizonylat metódus használatával, amennyiben a bizonylatot már engedélyezték.
- Az osztály Teljesitve attribútuma akkor állítható True-ra, ha a bizonylathoz tartozó kifizetések összege nagyobb vagy egyenlő, mint a bizonylatban szereplő végösszeg. (Felüldefiniáljuk a szülő SetTeljesitve metódusát.)
- VevoiRendelesBizonylatnál a forráskliens minden esetben a cég, ahol a rendszer fut, a célkliens a megrendelő.

A VevoiRendelesBizonylat mapping XML-je a jövetkező (a Bizonylatot tartalmazó <class>-on belül):

```
<subclass name="VevoiRendelesBizonylat">
  <property name="Cim" />
  <property name="SzallitolevelAzonosito" />
  <property name="Partner" />
  <property name=”KivetelezesiBizonylat” />
</subclass>
```

A mapping XML-ben nem található újdonság a Bizonylathoz képest, csupán a fentebb leírt tulajdonságokkal egészíti ki az ősosztályt.

3.1.2. Beszerzői rendelések

Beszerzési igény felmerülése esetén (pl. vevői rendelés teljesítéséhez szükséges, vagy készletminimum tartásához) a megfelelő jogosultságokkal rendelkező alkalmazott egy `BeszerzoiRendelesBizonylat`ot hoz létre. Teljesítésének feltétele a hozzá kapcsolódó `BevételezésiBizonylat` létrejötte.

A `Bizonylat` osztályból származó `BeszerzoiRendelesBizonylat` a következő attribútumokkal és kapcsolatokkal egészíti ki az ősét:

- Új kapcsolatként jelenik meg egy `BeveteliBizonylat`, mely automatikusan generálható a `BeszerzoiRendelesBizonylat`ból a `CreateBevetelezesiBizonylat` metódus segítségével, amennyiben a bizonylatot már engedélyezték. Az így létrejött bizonylat a beszerzési rendeléssel azonos információkat és termékeket tartalmaz.
- Az osztály `Teljesitve` attribútuma akkor állítható `True`-ra, ha a bizonylathoz tartozó kifizetések összege nagyobb vagy egyenlő, mint a bizonylatban szereplő végösszeg. (Felüldefiniáljuk a szülő `SetTeljesitve` metódusát.)
- `BeszerzoiRendelesBizonylat`nál a célkliens minden esetben a cég, ahol a rendszer fut, a forráskliens a beszállító.

A BeszerzoiRendelesBizonylat mapping XML-je csupán a bizonylathoz kapcsolódó bevételezési bizonylattal egészíti ki az ősosztályt:

```
<subclass name="BeszerzoiRendelesBizonylat">
  <property name="BevetelezesiBizonylat" />
</subclass>
```

3.2. Raktár

A raktármodul kezeli a raktárak árumozgását, és a készletek nyilvántartását. Ide tartozik a bevételezés, kivételezés, árumozgatás és a selejtezés.

A vállalatnál a `Raktar`-objektumok segítségével reprezentáljuk a termékek tárolására alkalmas – akár virtuális, akár fizikailag létező – helyeket. Ehhez az osztályhoz ki- és bevételi bizonylatok kapcsolódnak, az ezekben szereplő mennyiségek megfelelő előjellel vett szummája adja az aktuális készletet. Annak érdekében, hogy ne kelljen minden egyes lekérdezésnél ezt az összeadást végrehajtani, az objektum rendelkezik egy `RaktarCache` attribútummal, ami kivételezés és bevételezés esetén automatikusan frissül.

3.2.1. Bevételezés

Amennyiben áru érezik a céghez, bevételezési bizonylat kerül létrehozásra. Abban az esetben, ha ez egy beszerzői megrendelés vagy árumozgatás esetén történik, akkor a bizonylatot az alapján generáljuk, majd kapcsoljuk hozzá. Ez a bizonylattípus automatikusan teljesítésre kerül. A bizonylatok, melyekből generáltunk (`BeszerzoiRendelesBizonylat`, `ArumozgatasiBizonylat`), teljesítettnek lesznek jelölve.

Bevételezési bizonylatok tárolására a `Bizonylat` osztályból származtatott `BevetelezesiBizonylat` osztály objektumai szolgálnak. Új tulajdonságként a `SzallitoiLevélAzonosito` és a `Raktár` jelenik meg, valamint amennyiben `BeszerzoiRendelesBizonylat`ból vagy `ArumozgatasiBizonylat`ból generáltuk az adott bevételezési bizonylatot, a nekik megfelelő tulajdonság az adott bizonylatra hivatkozik. Ha a bevételezési bizonylat önállóan készült el, akkor mindkét hivatkozás értéke `null`. Az osztály override `SetTeljesitve()` metódusának meghívása esetén a bizonylathoz tartozó `Raktár` gyorsítótárazott értékei módosítva lesznek a bizonylat tételei alapján, valamint ha van kapcsolódó bizonylat, akkor azok is teljesítettnek lesznek jelölve. A bizonylattípushoz nem tartozik kifizetés, valamint a `CelKliens` minden esetben a rendszert használó cég.

Az osztály mapping XML-je a következő:

```
<subclass name="BevetelezesiBizonylat">
  <property name="SzallitolevelAzonosito" />
  <property name="Raktar" />
  <property name="BeszerzoiMegrendelesBizonylat" />
  <property name="ArumozgatasiBizonylat" />
</subclass>
```

3.2.2. Kivételezés, selejtezés

Az áru fizikai kiadása esetén kivételezési bizonylatot készítünk. Ha valamilyen folyamat bizonylata esetén történik fizikai árumozgás, a `KivetelezesiBizonylat` a szülőbizonylatból kerül generálásra. Létrejöttéhez szükséges, hogy a kivételezésre szánt termékből a raktárban megfelelő lefoglalatlan mennyiség legyen.

A raktárban a szállítás, vagy árumozgatás közben megsérült, lejárt szavatosságú termékek leselejteződnek, és a raktárból kivételezésre kerülnek. Ugyanez a helyzet a cég birtokában lévő eszközökkel is. Ezt egy selejtezési bizonylat reprezentálja, ami egy olyan speciális kivételezési bizonylat, melynél a `Selejtezes` attribútum értéke `True`. Amennyiben engedélyezik ezt a bizonylattípust, automatikusan generálódik egy

Kivételezési bizonylatok tárolására a `Bizonylat` osztályból származtatott `KivetelezesiBizonylat` osztály objektumai szolgálnak. Új tulajdonságként a `SzallitoiLevelAzonosito` és a `Raktar` jelenik meg, valamint a logikai értéket felvevő `Selejtezés`, ami azt mondja meg, hogy egyszerű kivételezési bizonylatról van szó (`False`) vagy selejtezési bizonylatról (`True`). Emellett amennyiben a bizonylatot egy `VevoiRendelesBizonylat`ból vagy `ArumozgatasiBizonylat`ból generáltuk, a nekik megfelelő tulajdonság az adott bizonylatra hivatkozik. Ha a bevételezési bizonylat önállóan készült el, akkor mindkét hivatkozás értéke `null`. Az osztály override `SetTeljesítve()` metódusának meghívása esetén a bizonylathoz tartozó `Raktar` gyorsítótárazott értékei módosítva lesznek a bizonylat tételei alapján, valamint ha van kapcsolódó `VevoiRendelesBizonylat`, akkor az is teljesítettnek lesznek jelölve. (`ArumozgatasiBizonylat` esetén azért nem, mert az árumozgatási folyamatban a kivételezés csak az első lépés) A bizonylattípushoz nem tartozik kifizetés, valamint a `CelKliens` minden esetben a rendszert használó cég.

Az osztály mapping XML-je a következő:

```
<subclass name="KivetelezesiBizonylat">
  <property name="Raktar" />
  <property name="VevoiRendelesBizonylat" />
  <property name="ArumozgatasiBizonylat" />
  <property name="Selejtezes" />
</subclass>
```

3.2.3. Árumozgatás

Az árumozgatás a cég tulajdonában lévő termékek raktárak közötti szállítását jelenti. Az árumozgatás folyama során egy kivételi és egy bevételi bizonylat generálódik. Akkor kerül teljesítésre, ha mindkét bizonylat teljesítésre kerül.

A raktárközi árumozgatásokat a `Bizonylat` osztályból származtatott `ArumozgatasiBizonylat` osztály objektumaiban tároljuk. Ennek a bizonylattípusnak az a speciális tulajdonsága hogy a forráskliens és a célkliens is a rendszert használó cég. Új tulajdonságként jelenik meg a `ForrasRaktár` és a `CelRaktár`, a két hely, amely között az árumozgás történik, valamint amennyiben már generáltunk `KiveteliBizonylat`ot és `BeveteliBizonylat`ot hozzá, akkor az ezekre való hivatkozás is megtalálható. A bizonylat automatikusan teljesítve lesz, ha a hozzátartozó bevételezési és kivételezési bizonylat is létrejött.

Az osztály mapping XML-je:

```
<subclass name="ArumozgatasiBizonylat">
  <property name="BevetelezesiBizonylat" />
  <property name="KivetelezesiBizonylat" />
  <property name="ForrasRaktar" />
  <property name="CelRaktar" />
</subclass>
```

3.2.4. Leltárazás

Minden raktárkezelő program alapvető funkciója a leltárazás, melyet több célból is szoktak végezni. Az egyik az esetleges személyi változások esetén a hanyag vagy hűtlen kezelések felderítése, illetve a hivatalos szervek ellenőrzései esetén a készlettel el lehessen számolni. Elengedhetetlen még a kimutatások, elszámolások (évvégi mérleg) esetén.

A leltárazásokhoz kapcsolódó információk tárolását a `Bizonylat` osztály egy újabb leszármazottja, a `LeltariBizonylat` osztály végzi. A leltárazás úgy történik, hogy a megfelelő termékekből `BizonylatTetel`eket hozunk létre, és ezeket bekötjük a leltári bizonylathoz az `AddBizonylatTetel` metódus segítségével. A leltár végeztével (ebben az esetben ez azt jelenti, hogy a `Teljesitve` tulajdonságot `True`-ra állítjuk) a `BizonylatTetel`ekben szereplő mennyiségeket termékenként összeadja, és összehasonlítja a `RaktarCache`-ben szereplő mennyiségekkel. Az esetleges eltérésekre azonnal fény derül, így felelősségre lehet vonni a felelősöket. A programban szereplő és a valós készletmennyiségek szinkronizálásához lehetőség van ki- illetve bevételezési bizonylatok generálására.

3.3. Pénzügy (számlázás, kifizetések)

Ez a modul a vállalati pénzmozgások nyilvántartását, valamint a pénztárak és egyéb pénzügyek kezelését végzi. Mivel a számviteli és pénzügyi szabályok rendkívül összetettek és feltérképezésük, megtervezésük és implementálásuk nagyon nagy szakértelmet kíván ebben a témakörben, továbbá számos dolgot törvény szabályoz, (pl hivatalos számlát is csak az APEH által jóváhagyott program bocsáthat ki), ezért csak felszínesen érintettük a témakört.

3.3.1. Számlakezelés

A teljesített vevői rendelésről számla készíthető. (A teljesítés feltételeit a 3.1.1-es fejezetben írtuk le részletesen.) A bizonylatokhoz (néhány kivételt, pl. a `BerezesiBizonylat`ot leszámítva) kapcsolódhat számla, melyet magukból a bizonylatokból generálunk.

Lehetőség van arra, hogy a beszállítói számlák adatait is rögzítsük a rendszerben (kiállítás dátuma, számlaazonosító), ebben az esetben nem a bizonylatból generáljuk, hanem kézzel adjuk meg az adatokat. Hivatalos számlát csak egy darabot lehet nyomtatni, azt is három példányban (egyedi azonosítóval rendelkeznek; van egy vevői, egy saját és egy törzspéldány), viszont másolatból akárhányat.

`Szamla` objektum tárolja a már kinyomtatott példányok számát, amennyiben ez még 0 akkor készül a hivatalos számla, minden más esetben másolatot nyomtat.

A számla nyomtatásának feltétele az, hogy a bizonylathoz kapcsolódó kifizetések összege, nagyobb vagy egyenlő legyen mint a bizonylathoz kapcsolódó bizonylattételekben szereplő összegeknek a szummája.

A kinyomtatott számlán a következő adatoknak kell szerepelnie:

- eladó neve, címe, számlaszáma (logója, egyéb elérhetősége)
- vevő neve címe, számlaszáma,
- teljesítés dátuma,
- számlakiadás dátuma,
- fizetés határideje
- tétel(ek) nevei (besorolási száma), mennyisége, mennyiségi egység,
- áfa mértéke,
- áfa nélküli összeg
- számla végösszege áfa nélkül,
    - összes áfa,
    - számla végösszege

Mivel ezen adatok a bizonylatban is megtalálhatók, a számla pedig nem létezik bizonylat nélkül (hiszen aggregáltja), ezért a számla objektum nem tárolja ezeket külön.

3.3.2. Pénztárkezelés

A program alkalmas a pénztári feladatkör ellátására: termékek eladása esetén `KiveteliBizonylat`ot, és `Kifizetes`t is generál, illetve ha szükséges, számlát is készít.

A rendszer kezeli a vonalkódbeolvasást, illetve a gyorskódot elsősorban azokhoz a termékekhez, melyeknek nincsen vonalkódja. Lehetőség van termék stornójára is. A nap végén lehetőség van a pénztár zárására.

3.3.3. Bérszámfejtés

Egy vállalat alkalmazottainak bérét több módszer szerint határozhatják meg a munkaadók. Van, aki fix bért kap, azonban a kapott összeget meghatározhatja a ledolgozott órák száma, vagy egy teljesen egyedi szabály is. A rendszer lehetőséget biztosít ezen bérezési szabályok megadására egy egyszerű, matematikai műveleteket és feltételeket tartalmazó formula segítségével. Ilyen szabályokat minden felhasználócsoporthoz rendelhetünk, ami a csoport összes alkalmazottjára vonatkozik, valamint amennyiben szükséges, alkalmazottanként is meghatározható a szabály.

A szabályokat egy XML-dokumentum írja le. Ezek a szabályok sztringként tárolódnak a `BerezesiSzabaly` osztály példányainak `Szabaly` attribútumában. A `BerezesiSzabaly` osztály egyszerű adattároló osztály. Alább látható mintaként egy olyan bérezési szabály, ahol az alkalmazott 120000 Ft alapbért kap, valamint ha több, mint 50 órát dolgozik, akkor megkapja jutalomként az alapbér egytizedét.

```
<BérezésiSzabály>
  <Konstans név="alapbér" érték="120000" />
  <Konstans név="jutalom" érték="alapbér/10" />
  <Szabály>alapbér+(dolgozott_órák>50?jutalom:0)</Szabály>
</BérezésiSzabály>
```

Az XML felépítése a következő:

- A gyökérelem a `BérezésiSzabály`.
- `Konstans` elem (A `BérezésiSzabály` gyermekeleme): ezen elem segítségével fix értékeket definiálhatunk. Ez azért hasznos, mert nem a formulába kell belenyúlni a módosításnál, hanem csak ezt az értéket kell átírni. A `név` attribútumban tároljuk azt az azonosítót, amivel majd a szabályban vagy a későbbi érték elemben hivatkozni akarunk az elemre. Az érték attribútum egy konstans értéket (pl. `5000`), egy C#-ban érvényes kifejezést (`4000*2+300`, vagy `Math.Pow(2+3, 3) + 5000`), vagy egy olyan C#-ban érvényes kifejezést, amiben egyes értékek helyén egy korábban már definiált konstans tartalmazzák (pl. `jutalom/10`) tartalmaz.
- `Szabály` elem: A bérezés kiszámításához használandó szabályt tartalmazza. Tartalma egy olyan C#-ban érvényes kifejezés, ami tartalmazhat korábban definiált konstansokat, valamint a rendszer által definiált konstansokat. A képlet feldolgozásakor először az azonosítók helyettesítődnek be a megfelelő értékre, majd a kifejezés értékelődik ki (implementációs részletek alább).

A rendszer által nyújtott konstansok, melyek használhatóak a szabályokban:

- `dolgozott_napok`: az adott időszakban munkával töltött napok száma
- `dolgozott_órák`: az adott időszakban munkával töltött órák száma
- `hétvégi_dolgozott_napok`: az adott időszakban munkával töltött hétvégi napok száma
- `hétvégi_dolgozott_órák`: az adott időszakban munkával töltött órák száma hétvégi munkanapokon
- `hétvégi_napok`: a hétvégi napok száma
- `minimálbér`: az éppen aktuális minimálbér összege
- `munkanapok`: az időszak munkanapjainak száma
- `túlórák`: a túlórák száma

A bérezési szabály kiértékelését a .NET Framework CodeDom-technológiája, a reflexió és a Microsoft saját C#-elemzője segítségével végzi el a rendszer. Ez a megoldás azért előnyös, mert nincs szükség saját elemzőmotor írására, valamint fejlettebb .NET-es matematikai metódusokat is fel lehet használni a megadáskor (pl. `Math.Pow`). A szabály elemzése során először feldolgozásra kerül az XML-fájl: ekkor behelyettesítődnek a szabályba a konstans értékek, valamint a rendszer által szolgáltatott konstansok. Ezután biztonsági ellenőrzést végzünk – ez azért fontos, mert mivel egy C#-elemzőnek adjuk át a kódot, esetlegesen fennállhat a veszélye annak, hogy valamilyen támadókódot tartalmazhat a kifejezés. Ezért azon szabályokat, melyek tartalmaznak bizonyos kifejezéseket, pl. az utasítást záró pontosvesszőt (`;`), vagy bizonyos C# kulcsszavakat (`return`, `public`, `void`), nem adjuk át a kifejezést a feldolgozónak. Az elemzés a következő kód segítségével történik:

```
CSharpCodeProvider cs = new CSharpCodeProvider();
CompilerParameters cp = new CompilerParameters();
cp.GenerateExecutable = false;
cp.GenerateInMemory = true;
cp.OutputAssembly = "BerezesiSzabalyKiertekelo";

string source = "using System;" +
  "namespace BerezesiSzabalyKiertekelo {" +
    "class Kiertekelo{" +
      "public static double Kiertekel() {" +
        "return " + expr + ";" +
      "}" +
    "}" +
  "}";

CompilerResults cr = cs.CompileAssemblyFromSource(cp, source);
```

```
if (cr.Errors.Count > 0)
{
  throw new ArgumentException();
}
else
{
  MethodInfo mi = cr.CompiledAssembly
    .GetType("BerezesiSzabalyKiertekelo.Kiertekelo")
    .GetMethod("Kiertekel");
  return (double)mi.Invoke(null, null);
}
```

Az első öt sorban a C#-fordító paramétereinek inicializálása történik meg. Megadjuk, hogy ne készüljön EXE-fájl (`GenerateExecutable = false`), a fordítás a memóriában történjen (`GenerateInMemory = true`), valamint a kimeneti szerelvény (assembly) nevét. A következő sorokban megadjuk a lefordítandó forrás szövegét, behelyettesítve a return után a kiértékelendő kifejezéssel, melyben már ekkor be vannak helyettesítve a konstansok értékei, így egy olyan kifejezést kell elvégeznie, amely ha helyesen van megadva, biztosan lefut. Az utolsó előtti lépésben lefordítjuk a szerelvényt, és amennyiben nem történt hiba, meghívjuk reflexió segítségével a memóriában lévő szerelvény `BerezesiSzabalyKiertekelo` névterében található `Kiertekelo` osztályának `Kiertekel` metódusát.

Alkalmazott kifizetésekor a kiválasztott időszakra (ami általában egy hónap, de ez nincs megkötve) egy új bérezési bizonylatot hozunk létre. A bérezési bizonylat a `Bizonylat` osztály leszármazottjában, a `BerezesiBizonylat` osztály objektumaiban tárolódik. Ilyen bizonylat létrehozható az alkalmazott éppen aktuális bérezési szabálya alapján (azaz ha az alkalmazott rendelkezik bérezési szabállyal, akkor azt felhasználva, ha nem, akkor a csoportjának szabályával), vagy egyedi, egyszer használandó szabály megadásával. `BerezesiBizonylat` esetén a forráskliens mindig a cég, a célkliens pedig az alkalmazott. Plusz attribútumként eltároljuk a bérezett időszakot (`Idoszak`), a kiszámított összeget (`Osszeg: Double`), valamint a bér kiszámításához használt aktuális bérezési szabály kibővített változatát. Ez azért szükséges, hogy később vissza lehessen keresni, mi alapján számolódott az összeg. A kibővített XML annyiban különbözik az eredetitől, hogy tartalmazza a rendszer által kiszámított konstansokat (pl. a dolgozott órák számát) is `<Konstans>` elemek formájában, amennyiben azok valahol használva voltak a szabályban.

Az összeg kiszámításakor egy adott időszakba azok a munkanapok tartoznak, melyek vagy teljes egészükben egy időszakba tartozó napon hajtódtak végre, vagy átnyúltak a megadott időszakba (tehát azok a munkanapok már nem, melyek az időszakon túlra nyúlnak). A

munkanapok közül azok nem lesznek figyelembe véve, amelyekhez már tartozik `BerezesiBizonylat`. A bizonylat létrehozásakor azon `Munkanap`-objektumoknál, melyek fel lettek használva az összeg kiszámolásához, kapcsolatot hozunk létre az éppen létrehozott `BerezesiBizonylat`-objektumhoz.

A `Munkanap` az alkalmazott munkaóráit tartalmazó adattároló osztály. Egy munkanapnál feljegyzett munkaórák között lehet szünet is, hiszen az osztály több időszakot is el tud tárolni egy munkanapra vonatkozóan. Kapcsolatai a következőek:

- `Alkalmazott`: az az alkalmazott, akihez a munkanap tartozik
- `Idoszakok`: egy aggregációs kapcsolat, a munkanaphoz tartozó időszakok
- `BerezesiBizonylat`: amennyiben a munkanap már tartozik valamilyen bérezési bizonylathoz, akkor ez a kapcsolat arra a bérezési bizonylatra mutat.

Az `Idoszak` egy egyszerű adattároló osztály, amely két tulajdonsággal rendelkezik: a `Kezdet` és a `Veg`, mindkettő `DateTime` típusú.

A bérezési bizonylatok akkor számítanak teljesítettnek, ha a bizonylathoz a bérrel megegyező kifizetés kapcsolódik az alkalmazott felé.

3.4. Kimutatások

Az üzleti világban az információnak, pénz értéke van, hiszen a megfelelő döntések meghozatalához, gyorsan, és megbízhatóan kell tudni információhoz jutni. Az is nagyon fontos hogy a vállalat hiába rendelkezik nagy mennyiségű adatvagyonnal, ha ebből nem tudja megfelelő pontossággal a hasznos információkat kinyerni, a vezetőség nem tud megfelelő döntéseket hozni. Életbevágó tehát, hogy az adattengerből üzleti információkat lehessen kihasználni. A kimutatáskészítő modul ezt a célt szolgálja. Kimutatásokat készíthetünk minden tevékenységgel kapcsolatban, amit a rendszer végez.

3.4.1. Beszerzések, eladások, megrendelések

- **Beszerzések vizsgálata:** A nyereséges kereskedelem egyik alapfeltétele, hogy a termékeket midig a legkedvezőbb áron szerezzük be. Ehhez szükséges a megfelelő beszállító kiválasztása, amiben segítséget nyújthat a korábbi megrendeléseink elemzése. A program képes egy adott időszakra, termékre, vagy beszállítóra vonatkozóan beszerzéseket megjeleníteni táblázatos formában illetve diagramon. Segítségével könnyen elemezhetők a be-

szerzési árak, könnyebben **kiválasztható a megfelelő beszállító.**

- **Vásárlások vizsgálata:** A másik fontos tényezője a kereskedelemnek a vásárlási szokások elemzése, a vásárlói igények feltérképezése. A program képes táblázatszerűen vagy grafikonon szemléltetni a vásárlásokat termékre, időszakra, illetve akár egy pénztárra vonatkozóan. A kimutatásnál további információk is megjeleníthetők, mint pl. az eladási ár, mennyiség stb.
- **Teljesítetlen megrendelések:** Az idő pénz, tartja a mondás is, megrendelések nem teljesítése, a határidők be nem tartása, komoly pénzbe kerülhet, akár beszerzői, akár vevői rendelésről van szó. Fontos tehát, hogy mindig pontos képet kapjunk a teljesítetlen megrendelésekről. Táblázatos vagy grafikus formában tájékozódhatunk a megrendelések állapotáról és teljesítési határidejükről.

3.4.2. Raktár

- **Selejtezések:** Az esetleges visszaélések, illetve emberi mulasztások felderítése céljából célszerű elemezni a selejtezéseket is. Ez könnyen és pontosan adott időszakra, termékre illetve dolgozóra vonatkozóan lekérdezhető.
- **Készletvizsgálat:** Lehetőség van a programban raktárakban található mennyiségek lekérdezésére, a későbbi megrendelések készítése során hasznos információ lehet. Lehetőség van az eredményt termékre, illetve mennyiségre, készletminimumra vonatkozóan szűkíteni.
- **Árumozgások:** A raktárba történő összes be-és kivételezést a Be-és KiveteliBizonylatok tárolják. Kimutatás készíthető egy adott időszakra vonatkozó összes árumozgásról, termékre raktárra, illetve mennyiségre vonatkozóan.

3.4.3. Pénzügy

- **Alkalmazott bérek, beszállítói kifizetések, vevői kifizetések:** A rendszer képes kimutatásokat készíteni az összes tárolt pénzmozgásra vonatkozóan, ezek eredményét partnerre, időszakra vonatkozóan, táblázatos, illetve grafikus formában. Lehetőség van több feltételből rugalmasan összeállítani a lekérdezést, például egy adott beszállítóra vonatkozóan, adott időszakon belüli teljesített, és teljesítetlen kifizetések arányát.
- **Likviditásszámítás:** Az adott időpillanatra vonatkozóan kiszámolja a likviditást. Beállít-

ható kritikus érték, amely alá csökkenve a rendszer jelez, hogy alacsony a likviditás.

3.4.4. HR

A vállalat dolgozóinak munkaóráit, valamint hiányzásait lehet nyomon követni. Egy konkrét példa:

A vállalat vezetője kíváncsi egy adott alkalmazott munkaóráira, az elmúlt egy hónapra vonatkozóan. A rendszer lekéri az adott dolgozót reprezentáló objektumot, amelyhez aggregálnak munkanapok. Ezek után feloldja ezen kapcsolattípust, azaz eléri az összes hozzá kapcsolódó munkanap típusú gyermek-objektumot. Ezen objektumok mindegyikéhez további, -intervallum típusú- objektumok tartoznak, amelyek a ledolgozott időszakokat reprezentálják. Ezek segítségével kiválasztjuk azt a napot, melyek a megfelelő időintervallumba esnek. (Mint látható összetett lekérés, ami sok osztályt érint, a natív SQL-hez képest jelentős overhead figyelhető meg. Az optimalizáció egyik lehetséges módja, hogy natív SQL-t írunk)

4. Egyéb implementációs kérdések

4.1. Felhasználó- és jogosultságkezelés

Egy olyan rendszernél, ahol a felhasználók száma a több százat is elérheti, elengedhetetlen, hogy megfelelő felhasználó- és rugalmasan alakítható jogosultságkezelés legyen. Ezen terület kezelésében két nagyon fontos probléma merül fel:

Az első, hogy meg lehessen adni a felhasználói fiókra, a felhaszálónévre, valamint a jelszóra vonatkozó megkötéseket. Számos helyen például előírás, hogy a jelszót le kell cserélni egy adott idő eltelte után, a rendszer védelme érdekében. Ugyanide tartozik a felhasználói fiókok inaktívvá tétele, amennyiben annak tulajdonosa nem jelentkezik be egy adott időszak eltelte után. A programnál ezek a paraméterek rendszerszinten megadhatóak.

A másik fontos probléma, hogy rugalmas legyen a felhasználókezelés és a jogosultságadás. Ezt a problémát a rendszer úgy kezeli le, hogy bevezeti a felhasználócsoport fogalmát. Az egyes műveletekhez tartozó jogosultságokat egy ilyen csoporthoz rendeljük hozzá, és a felhasználók – az alkalmazottak – kötelezően beletartoznak egybe, így meghatározva, hogy mit tehetnek a rendszerben. Szinte minden különböző művelethez tartozik külön jogosultság, azaz különböző van például egy vevői rendelés-bizonylat létrehozásához, és egy másik a vevői rendelés-bizonylat elfogadásához.

Egyetlen kiegészítő osztály ehhez a témakörhöz a `FelhasznaloiCsoport`. Egyszerű adattároló osztály, attribútumai a `Nev` és a `Jogosultsagok`. Az utóbbi egy egyszerű String-tömbben tárolja a jogosultságok azonosítóit.

4.2. Internationalization

A vállatirányítási rendszerek általában a világ minden táján használatosak, nem csak angolul beszélő felhasználók által. Fontos, hogy lehetőség legyen a szoftver felületének egy, az ügyfél által kívánt nyelven való megjelenítésére. A szoftver többnyelvűvé tételére az *internationalization*, vagy röviden *i18n* kifejezést használják.

Számos módszer létezik a probléma megoldására. A *Framework Class Library* részeként a .NET is rendelkezik saját *i18n*-megoldással. A módszer meglehetősen jó, de nem hibátlan: a nyelvspecifikus szövegeket ugyanis ún. *satellite assembly*kben tárolja. Ez azért nem jó, mert míg egy szövegfájllal, egy XML-dokumentummal, vagy a gettext esetén használt .po-formátumú fájllal bármelyik fordító elboldogul, és azonnal tesztelni tudja az eredményt, addig

satellite assemblyk létrehozásához és frissítéséhez szükség van a fejlesztők közreműködésére. Másrészt új nyelvek hozzáadása esetén a projekt újrabuildelésére van szükség, mert az elérhető nyelveket már az előtt meg kell határozni.

A fenti problémákat megoldja a linuxos világból ismert *gettext* .NET-es implementációja. A nyelvek számát a fejlesztő közreműködése nélkül maga a fordító is bővítheti. A fordítást Portable Object-formátumban (.po) végezheti. Mivel ez az implementáció a .NET belső fordítómechanizmusát használja, szükség van a .po-fájlok DLL-lé való konvertálására, de ez egy egyszerű parancssori művelettel elvégezhető, és itt sincs szükség fejlesztői közreműködésre. Ezen előnyök ellenére a gettext legnagyobb hátránya, hogy a lefordított sztringek azonosítójaként nem egy szabadon megadott azonosítót használ, hanem minden esetben magát a sztringet, így nincs lehetőség arra, hogy megjelöljük, ha egy adott szöveg más kontextusban van, vagy hogy egy időbélyeg-formázásnak beszédes, a fordítót segítő nevet adjunk; ez az oka, hogy nem ezt a megoldást választottuk.

Mivel az elvárásainknak (a nyelvek számát a program újrabuildelése nélkül is lehessen bővíteni, különböző kontextusban lévő azonos szövegek megkülönböztetése, a fordítók ne legyenek teljes egészében a fejlesztőkre utalva) egyik feni módszer sem felelt meg, ezért úgy döntöttünk, hogy saját megoldást fejlesztünk ki. A rendszer XML-fájlokból olvassa be a fordított szövegeket. Egy-egy ilyen fordítást tartalmazó fájl felépítése a következő:

```
<Localization langid=”hu” langorig=”Magyar”
      langeng=”Hungarian” translator=”Core”>
  <Translation id=”DateFormat”>yyyy. mmm. dd.</Translation>
</Localization>
```

Az XML-fájlok felépítése a következő:

- A gyökérelem a `Localization`. Attribútumai a `langid`, ami a nyelv ISO-kódját tartalmazza, a `langorig`, ami a nyelv nevét tartalmazza az adott nyelven, a `langeng` a nyelv nevét angolul, a `translator` pedig a fordító adatait.
- A `Localization` elem gyermekelemei egy adott szöveg fordítását tartalmazó `Translation` elemek. Az `id` attribútum az a sztring, amire a programban hivatkozunk, mikor ennek az adott szövegnek a fordítását keressük. Az elem tartalma (*innerText*) maga a fordítás.

A programban a `VIR.I18n` névtérben található a singleton, nyelvi funkciókat tartalmazó `Localization` osztály. Az osztály inicializációja során feltérképezi az elérhető nyelvek listáját, majd betölti a paraméterül kapott nyelv fordításait. A `Localization` két lényeges metódussal rendelkezik, az egyik a `GetLanguageList()`, ami egy `List<Language>`-objektumban visszaadja az elérhető nyelvek listáját. (A `Language` egy egyszerű adattároló osztály, ami tartalmazza a nyelv XML-ben tárolt adatait) A másik a `Translate`, ami elérhető a *gettext*ből ismert alulvonás (_) néven is. A `Translate` függvénynek két paramétert kell megadni, az egyik az azonosító, amit keresünk az XML-fájlban, a másik pedig az alapértelmezett szöveg, ami akkor jelenik meg, ha nem létezik fordítás az adott nyelven. Amennyiben a `Translate` függvénynek egy paramétert adunk meg, akkor az azonosító és az alapértelmezett szöveg megegyezik (*gettext*-szerű működés).

4.3. Zárolás és konzisztencia

Az objektumokkal való munka során a legkisebb munkaegység a tranzakció. A tranzakció atomi, több utasítás tartozik egy oszthatatlan egységbe. Továbbá a tranzakciók biztosítják, hogy több felhasználó konkurens módon dolgozhasson ugyanazon az objektumon, az integritás, és a konzisztencia veszélyeztetése nélkül.

A tranzakcióknak nem szabad látni, illetve befolyásolni a velük konkurensen futó tranzakciókat, ezt izolációnak nevezzük. Ennek a fajta viselkedésnek az implementálására különféle technikák léteznek.

4.3.1. Zárolás

A zárolási mechanizmus segítségével konkurens hozzáférés valósítható meg ugyanahhoz az adathoz. Ha egy tranzakció zárat kér egy objektumra, más tranzakció nem olvashatja/módosíthatja az adott objektumot a zár fajtájának megfelelően. A zár lehet pillanatnyi, amíg az objektumot a másik tranzakció betölti, illetve tranzakció szintű, amely a zárat kérő tranzakció befejezéséig tart. Mivel a rendszer egy webalapú alkalmazás, minden kérésnél új tranzakciót indítunk. Ez maga után vonja azt, hogy a zár csak addig él, amíg a kérés lefut. Például valaki lekérdez egy bizonylatot, módosítani akarja, de mivel a betöltést végző kérés végével a tranzakció véget ér, a zár is oldódik. Ha e közben valaki más módosítja az objektumot, akkor inkonzisztens állapot léphet életbe, hiszen az ő módosításai elvesznek. Ezért a zárolást alkalmazás szinten kell megvalósítani, úgy hogy a konkurens hozzáférést a lehető legoptimálisabban valósítsuk meg. Ezt az adott osztályhoz tartozó repository osztályban való-

sítjuk meg, valamint az `ObjectLock` nevű osztály segítségével. Ez reprezentálja az objektumokra esetlegesen fennálló zárat. Egy attribútuma van, amely a zárolandó objektum `Id`-jét tartalmazza. Amikor egy objektumot a repository `GetById()` metódusával lekérünk, létrehozunk egy `ObjectLock` objektumot, így képzünk rá írási zárat. A logikai tranzakció bejefeztével a zárat oldjuk. (Ez a metódus hívódik meg a repository `Update` metódusában is).

A repository az objektum frissítése előtt ellenőrzi, hogy nem áll e zárolás alatt, amennyiben a zár fennáll `ObjectLockedException`t dob, ellenkező esetben pedig elvégzi az objektum módosítását.

Mivel a felhasználói felületek nagyrészt a lista-form koncepcióra épülnek, ezért a felületeken a listák a `GetByType()` metódus segítségével érik el a megfelelő objektumokat (ebben az esetben nem képződik zár), a szerkesztő `Form`ok pedig a listából kiválasztott objektumot ismét lekérik a `GetById()` metódus segítségével (ebben az esetben zár képződik).

```
public Bizonylat GetById(Guid bizonylatId){
  using (ISession session = NHibernateHelper.OpenSession()){
    IQuery q = session.CreateQuery(
      "FROM ObjectLock Where ObjectId="+bizonylatId.ToString());
    //ha nincs zár visszaadjuk az objektumot
    if (q.List().Count == 0){
      ObjectLock oj = new ObjectLock(bizonylatId);
      session.Save(oj);
      return session.Get<Bizonylat>(bizonylatId);
    }
    else throw new ObjectLockException();
  }
}
public IList GetByType(string tipus){
  string hql = "FROM "+tipus;
  IList ered = null;
  using (ISession session = _sessionFactory.OpenSession()){
    IQuery q = session.CreateQuery(hql);
    ered = q.List();
    return ered;
  }
}
public void Unlock(Guid Id){
  IList ered = null;
  using (ISession session = NHibernateHelper.OpenSession()){
    IQuery q = session.CreateQuery(
      "FROM ObjectLock Where ObjectId=" + Id.ToString());
    IList ered = q.List();
    session.Delete(ered[0]);
  }
}
```

5. Továbbfejlesztési lehetőségek

Mivel célunk nem egy komplett rendszer, hanem csak egy alapfunkciókat tartalmazó program megtervezése volt, ezért számos továbbfejlesztési lehetőséggel rendelkezik. Alább kigyűjtöttünk párat a teljesség igénye nélkül, válogatva a jelenleg elérhető vállalatirányítási rendszerek funkciókínálatából, és a tervezés során felmerülő egyéb szempontok alapján:

- Az üzleti folyamatok mélyebb szintű modellezése: ajánlatkérés, szerződések, ellátási láncok nyilvántartása.
- Raktárak esetén készletoptimalizáció
- Business intelligence alkalmazása a döntési folyamatok előkészületénél
- Adatbányászati eszközök alkalmazása üzleti adatok előállításához (pl. vásárlási trendek felderítése)
- Teljesítmény és munkaminőség alapján történő bérezés lehetősége
- Gyártási folyamatok modellezése
- Minőségbiztosítás
- Automatikus és teljes körű biztonsági mentések készítése
- Adatimportálás külső programokból
- Integrálás irodai alkalmazásokkal (Word-, Excel-, és OpenOffice-formátumok támogatása)

6. Összefoglalás

A dolgozat készítése során a hangsúlyt nem az implementációra, hanem a tervezésre helyeztük, hiszen egy rosszul megtervezett rendszer beszűkíti a továbbfejlesztési lehetőségeket, és megnehezíti a karbantartást. Ezeket szem előtt tartva alakítottuk ki a Bizonylat-koncepciót, amely a rendszer architektúrájának alappillére. A koncepció absztrakt volta miatt rendkívül egyszerűvé teszi a további funkciókkal való bővítést.

A tervezés során rengeteg problémával találtuk szembe magunkat. Ezek közül számos olyan volt, ami nem tartozik szervesen az informatika tárgykörébe, így ezeket felületesen érintettük, egyrészt terjedelmi okok miatt – csak a számvitel témakörébe tartozó ismeretekkel teljes könyveket lehetne megtölteni –, másrészt nem egy teljeskörű rendszer megalkotása volt a célunk, hanem csak az alapkövek letétele. Emiatt implementálásra csak a rendszer magja került: a Bizonylat-koncepció, a felhasználó- és nyelvkezelés. A további bővítésekhez nincs szükség a mag módosítására.

Mivel az alkalmazott technológiáknak vannak hátrányaik is, ezért hogy a dolgozat szemlélete objektív legyen, bemutattuk a felhasznált megoldások hátrányait és az elvetettek előnyeit is. Azokban az esetekben, ahol több lehetőség közül választhattunk, kifejtettük, hogy egy-egy technológiát miért nem alkalmaznánk egy éles rendszerben.

Reméljük, hogy a dolgozat megfelelő képet nyújt egy vállalatirányítási rendszer megalkotásának nehézségeiről, és a probléma összetettségéről. Igyekeztünk kiemelni az alkalmazott technológiákat, azok előnyeit, és hogy milyen problémákra jelentettek megoldást.

7. Irodalomjegyzék

7.1. Könyvek, cikkek, tanulmányok

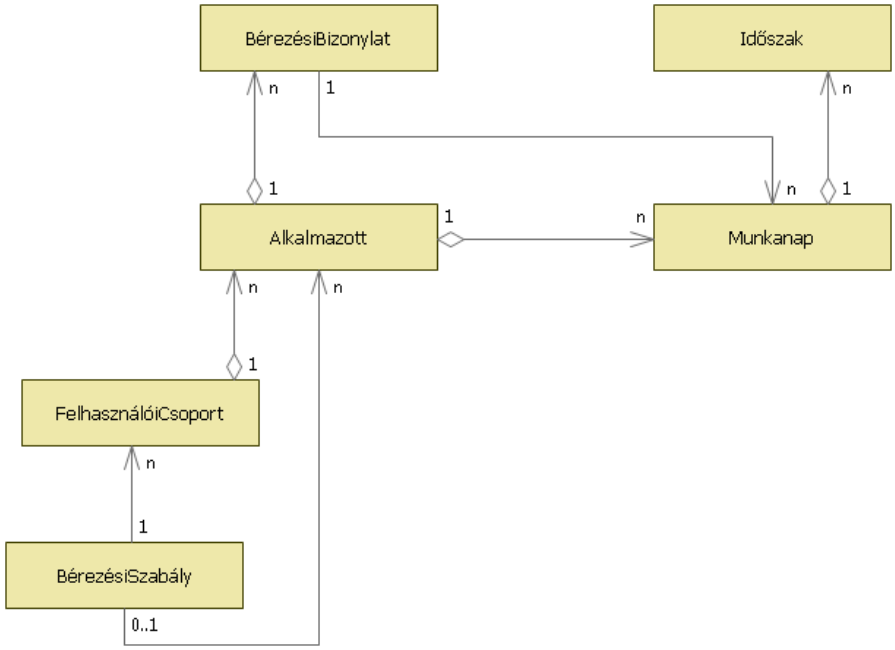
- Dr. Szikora Béla (szerk.): Vállalatirányítási rendszerek (BME-jegyzet)
- Jeffrey Richter: *CLR via C#*, Wintellect (2010)
- Pierre Henri Kuaté, Tobin Harris, Christian Bauer, and Gavin King: *NHibernate In Action*, Manning (2009)
- Tóth Elek: *Rendszerfejlesztés* (esettanulmány)

7.2. Internetes adatforrások

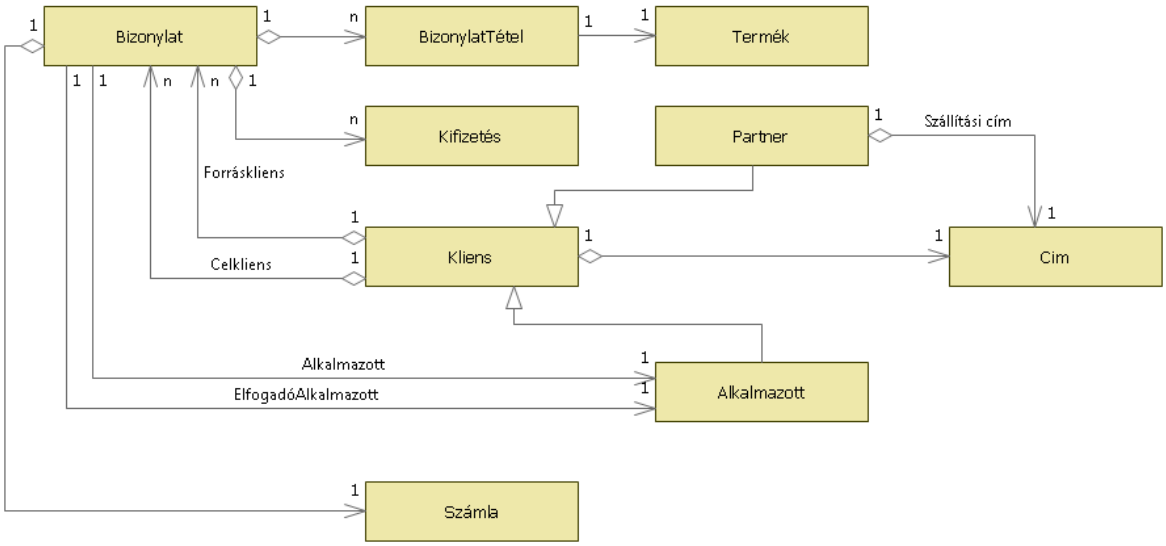
- About MySQL – (http://www.mysql.com/about/)
- GNU 'gettext' utilities – C# (http://www.gnu.org/software/hello/manual/gettext/C_0023.html)
- NHibernate Reference Manual 2.1.0 (http://nhforge.org/doc/nh/en/index.html)
- Wikipédia: Object-relational mapping (http://en.wikipedia.org/wiki/Object-relational_mapping)

A. Függelék

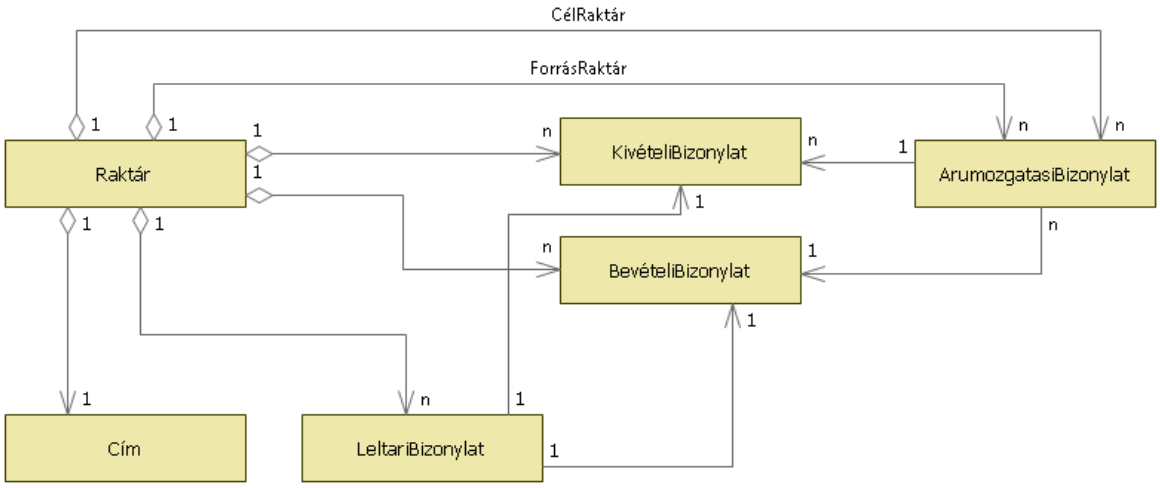
A.1. A bérezéssel kapcsolatos osztályok kapcsolatai



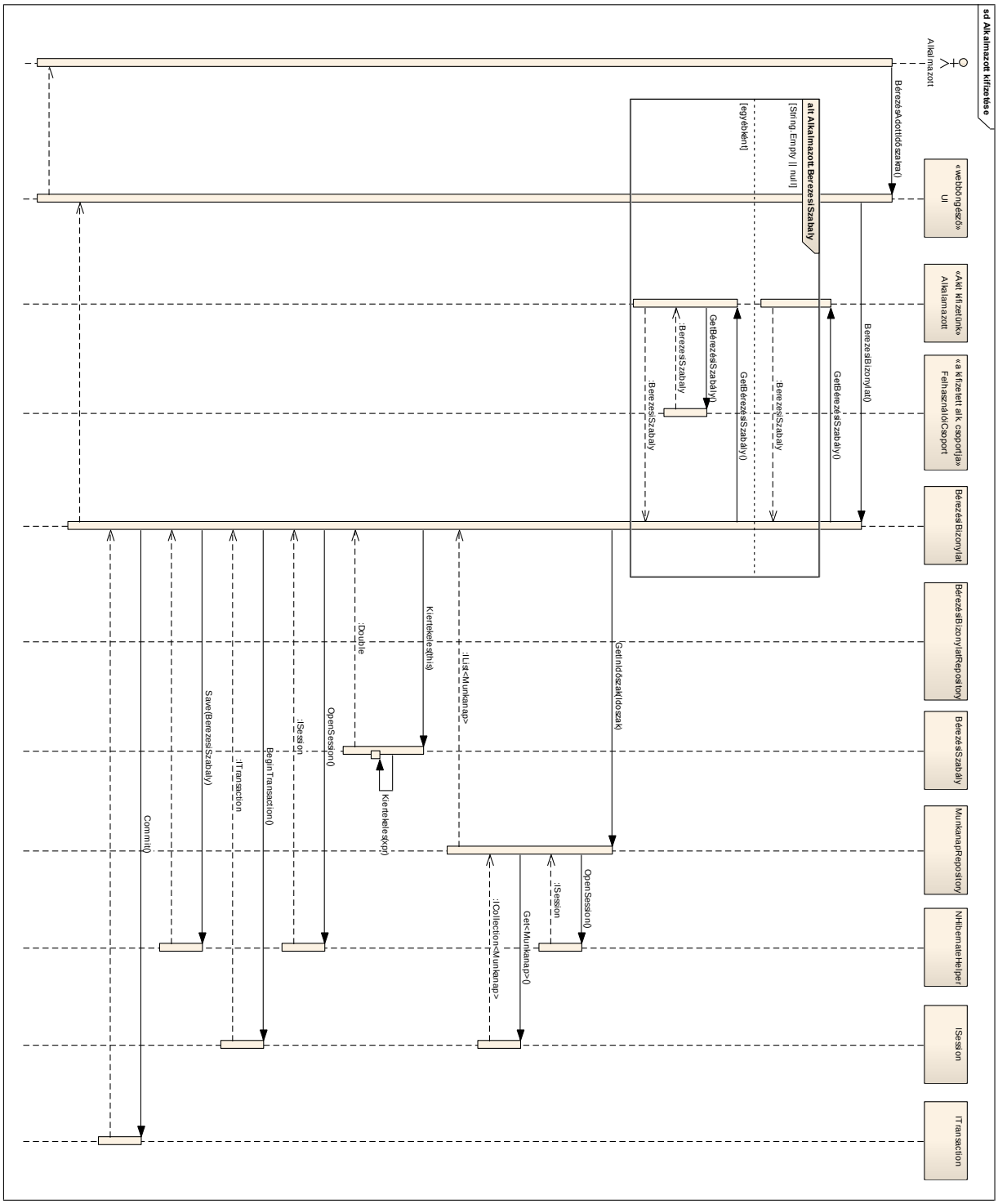
A.2. A `Bizonylat`tal kapcsolatos osztályok kapcsolatai



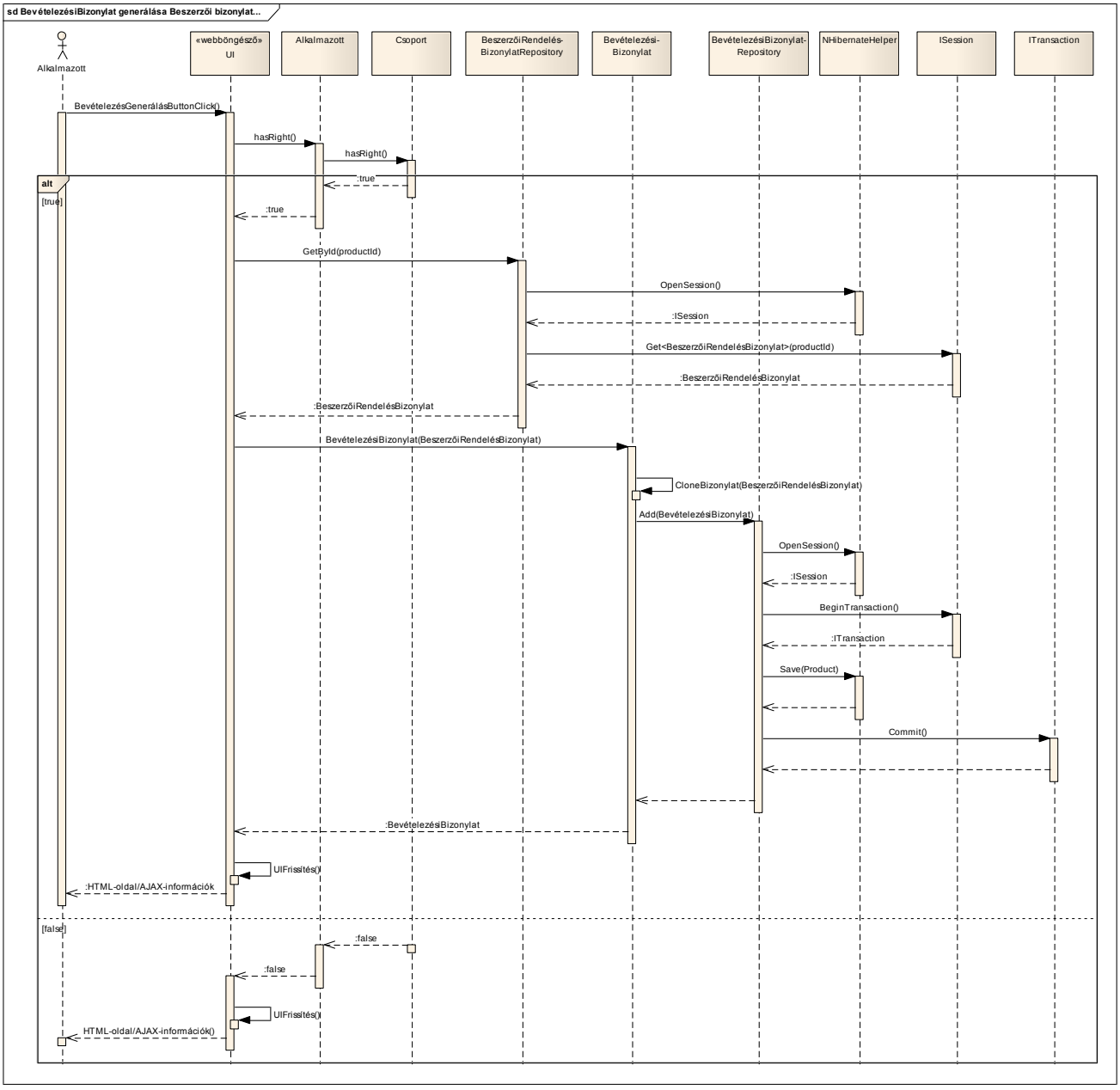
A.3. A raktárrakkal kapcsolatos osztályok kapcsolatai



A.4. Bérezési bizonylat létrehozása



A.5. Bevételezési bizonylat generálása beszerzői rendelésből



A.6. Zárolási probléma