Debreceni Egyetem, Informatikai Kar, Információ Technológia Tanszék

Állatmenhely webalkalmazás tervezése UML segítségével

Témavezető:

Pánovics János

egyetemi tanársegéd

Készítette:

Deák Zoltán

programtervező matematikus

Debrecen
2009

TARTALOM

ELŐSZÓ

Diplomamunkám témája az UML gyakorlati alkalmazásának bemutatása. De mi is az az UML? „Az UML egy általános célú vizuális modellező nyelv, amely arra használható, hogy specifikáljuk, szemléltessük, megtervezzük és dokumentáljuk egy szoftverrendszer architektúráját.” 1997-ben jelent meg az 1.0 verzió, azóta az objektumorientált irányzat széles körben felhasznált alapvető eszköze lett, ami annak is köszönhető, hogy egy szabványos megoldás.

Az UML gyakorlati alkalmazását egy saját rendszer tervezésén keresztül szeretném bemutatni. Fontos szempontnak tartottam a tervezni kívánt alkalmazás kiválasztásakor, hogy azon az UML eszközeinek minél szélesebb skáláját lehessen bemutatni. Végül a választásom egy webes rendszer tervezésére esett. Azért választottam webes megoldást, mert manapság a webes alkalmazások egyre nagyobb teret nyernek, és szeretném bemutatni, hogy az UML ezekhez is kiváló partner.

A tervezendő webes alkalmazás azt célozza meg, hogy állatmenhelyeknek nyújtson segítséget. Nem csak azért választottam ezt a témát, mert nagy állatbarát vagyok, hanem azért is, mert ilyen alkalmazással még nem találkoztam, és szerettem volna valami újat alkotni.

Tehát a diplomamunkám ennek az alkalmazásnak a megtervezéséről szól az UML eszközeinek segítségével. De még mielőtt belevágnék a tényleges tervezésbe, először bemutatom a szoftverfolyamatot és annak modelljeit. Majd foglalkozok kicsit az UML történetével és összefoglalom diagramtípusait.

Ezután térek át az alkalmazás megtervezésesre. Egy rövid áttekintés után elmélyülök a használati eset diagramok világában, majd megismerkedem a tervezés szempontjából néhány igen fontos kiegészítő dokumentummal. Majd következik egy újabb nagy falat, az osztálydiagramok. Ezt követően az objektumdiagramokra térek át. Később bemutatom az interakciós diagramokat, az időben lezajló változás diagramjait és végül az implementációs diagramokat.

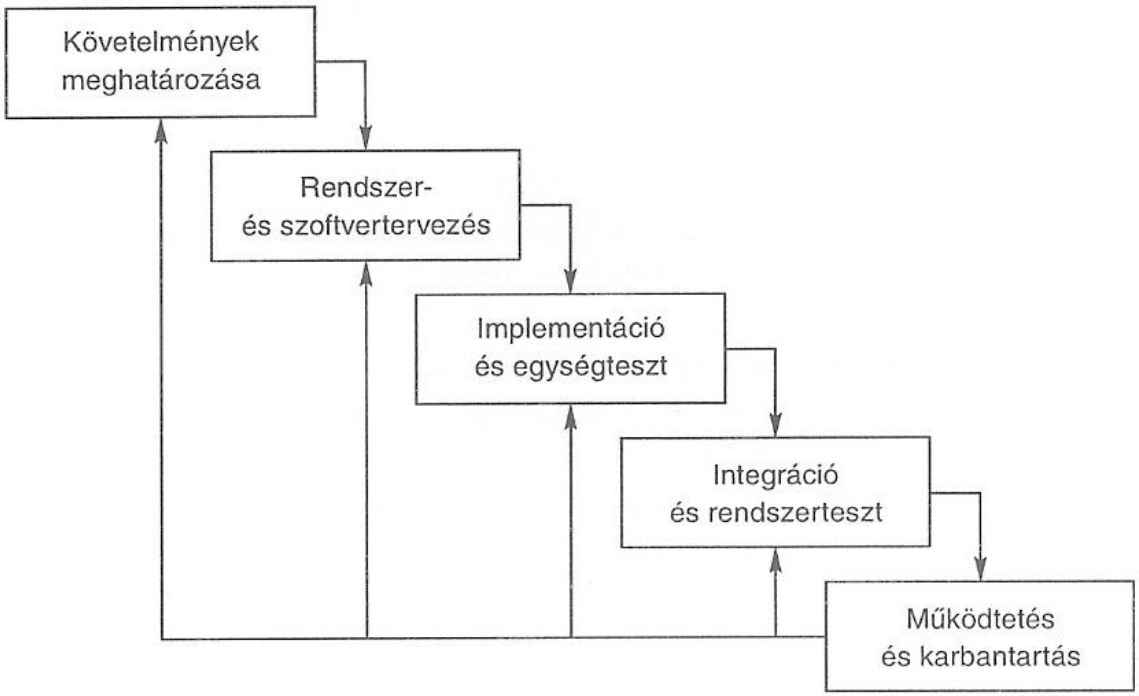
A SZOFTVERFOLYAMAT

A szoftverfolyamat olyan tevékenységek és kapcsolódó eredmények sorozata, amelyek egy szoftverrendszer előállítását célozzák meg.

A SZOFTVERFOLYAMAT MODELLJEI

VÍZESÉS MODELL

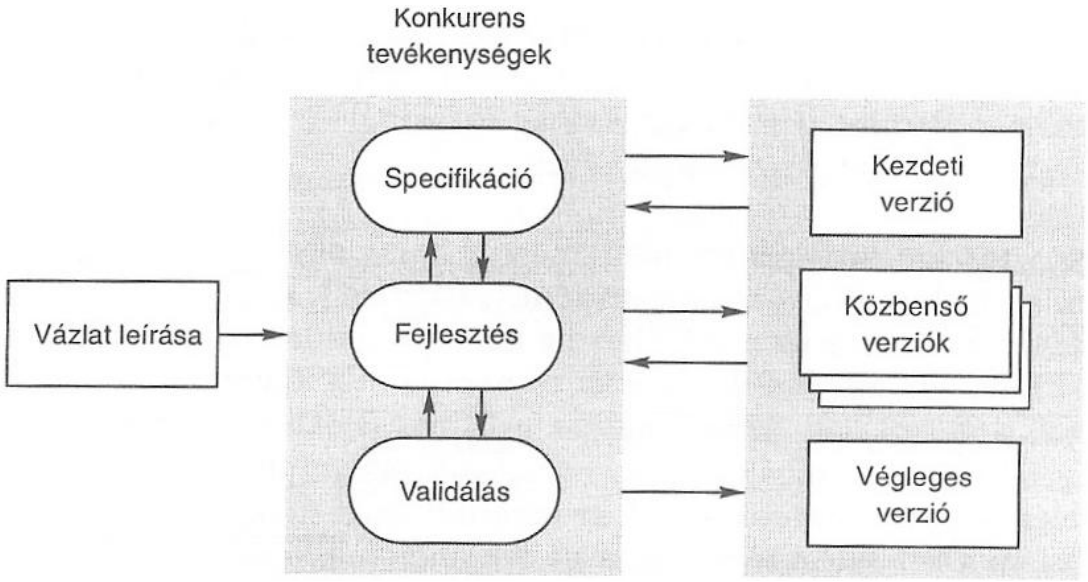
A folyamat alapvető tevékenységeit különálló fázisként tekinti. Ezek a fázisok lépcsősen kapcsolódnak egymáshoz. A fázisok eredményeit úgy kell elképzelni, mint jóváhagyott, aláírt dokumentumokat. Egy munkafázisnak be kell fejeződni, mielőtt a következő elkezdődhet. Tehát nincs átfedés, nincs párhuzamosság, emiatt könnyen menedzselhető. Viszont nehéz a változó megrendelői igényekhez igazodni, mert a projekt nehezen változtatható részegységekből áll. Ez a modell akkor hasznos, ha a követelmények jól ismertek és csak nagyon kis változások lehetségesek a fejlesztés során. Sajnos csak kevés üzleti rendszernek vannak stabil követelményei. A modellt kis méretű projekteknél ma is használják.



- Követelmények elemzése és meghatározása: a rendszer szolgáltatásai, megszorításai, céljai a felhasználóval történő konzultáció alapján alakulnak ki. Ezek szolgáltatják a rendszerspecifikációt.
- Rendszer- és szoftvertervezés: A rendszertervezés szakaszában választódnak szét a hardver- és szoftverkövetelmények. Itt kell kialakítani a rendszer átfogó architektúráját.
- Implementáció és egységteszt: a szoftverterv programok illetve programegységek halmazaként realizálódik. Az egységteszt azt ellenőrzi, hogy minden programegység megfelel-e a specifikációjának.
- Integráció és rendszerteszt: a különálló programegységek integrálása és teljes rendszerként történő tesztelése
- Működtetés és karbantartás: általában ez a leghosszabb fázis. A rendszer telepítése és használatba vétele után a fellépő hibák kijavítása, a rendszeregységek implementációjának továbbfejlesztése, valamint a szolgáltatások továbbfejlesztése a felmerülő új igényeknek megfelelően.

EVOLÚCIÓS FEJLESZTÉS

Az evolúciós fejlesztés során először kifejlesztünk egy kezdeti verziót. Ezt véleményeztetjük a felhasználókkal, így nagyon gyorsan kapunk egy visszacsatolást. Majd a kezdeti verziót finomítjuk sok-sok verzión keresztül a felhasználókkal való folyamatos együttműködéssel, míg el nem érjük a kívánt eredményt. A rendszer egyes elemeinek fejlesztése párhuzamosan történik.



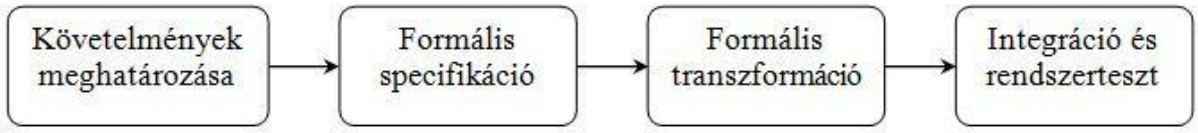
Két típusa:

- Feltáró fejlesztés: A fejlesztést a rendszer jól ismert részeivel kezdjük, majd az ügyfél által kért tulajdonságok folyamatos hozzáadásával alakítjuk ki a végleges rendszert.
- Eldobható prototípus készítése: Itt az alapötlet az, hogy először a rendszer azon részére kell koncentrálni, amelyek kevésbé érthetőek. Így a prototípusokon keresztül tökéletesítjük a követelményeknek való megfelelést.

A modell előnye tehát, hogy akkor is alkalmazható, ha kezdetben az összes követelmény nem ismert és az első prototípus viszonylag hamar elkészül, így gyors visszacsatolást kaphatunk a felhasználóktól. A modell így nagy rendszereknél is használható. Hátránya, hogy a fejlesztés nem átlátható és folyamatos változtatások oda vezetnek, hogy a rendszer rosszul strukturált lesz.

FORMÁLIS RENDSZERFEJLESZTÉS

Hasonlít a vízesésmodellhez, de itt a futtatható programot transzformálással alakítjuk ki formális matematikai eszközök segítségével.

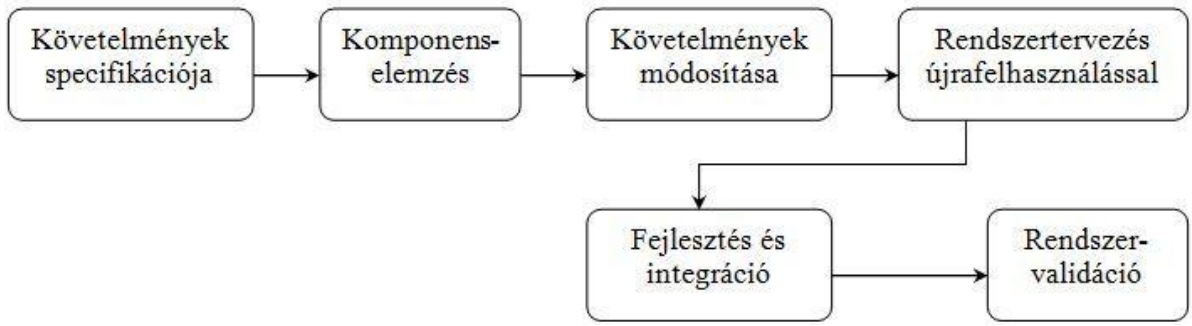


Tehát ennél a modellnél a szoftverkövetelmények specifikációit matematikai jelölésekkel leírható formális specifikációvá kell alakítani. Valamint a tervezés, az implementáció és az egységteszt fejlesztési folyamatokat egy transzformációs folyamat helyettesíti, ahol a formális definíció a transzformációk során finomodik, míg végül programmá válik.

Ismert példa az IBM Cleanroom-folyamata. Olyan rendszereknél alkalmazták, ahol a biztonság, megbízhatóság, védelem nagy szerepet játszik. Probléma, hogy kezdetben az összes követelmény általában nem ismert és, hogy a rendszerkövetelmények matematikai eszközökkel való felírása általában nagyon nehézkes.

ÚJRAFELHASZNÁLÁS-ORIENTÁLT RENDSZERFEJLESZTÉS

A modell alapötlete az újrafelhasználás. Ha már létezik olyan komponens, amely hasonlít a kívánthoz akkor használjuk azt, alakítsuk át és építsük be a rendszerbe, ezáltal csökkenthetjük a fejlesztési időt. Ilyen modell a komponensorientált szoftverfejlesztés.



- Komponenselemzés: a követelményspecifikációban szerepeltek alapján meg kell vizsgálni, hogy mely komponensek állnak rendelkezésre, és a komponensek mely funkcióikban felelnek meg a követelményeknek. Legtöbbször nincs pontos illeszkedés, a felhasznált komponens a funkcióknak csak egy részét nyújtja.
- Követelmények módosítása: elemezni kell a követelményeket a komponensek információit felhasználva. Kísérletet kell tenni a követelmények átalakítására az elérhető komponenseknek megfelelően. Ha ez nem sikerül, akkor vissza kell térni a komponenselemzési fázisba és alternatív megoldást keresni. (Másik komponens vagy saját fejlesztés)

- Rendszertervezés újrafelhasználással: a rendszer szerkezetének kialakítása annak figyelembevételével, hogy milyen komponenseket akarnak újrafelhasználni és együttműködtetni.
- Fejlesztés és integráció: A nem megvásárolható komponenseket ki kell fejleszteni és a felhasznált komponensekkel egy rendszerbe kell integrálni.

A modell segítségével lecsökkenthetjük a kifejlesztendő komponensek számát, ezáltal csökken a költség, az idő és a kockázat is. Viszont mivel olyan komponenseket használunk, amelyek teljes mértékben nem felelnek meg a követelményeknek ezért gyakran kompromisszumokat kell kötnünk és nem tudunk megfelelni a megrendelő minden elvárásának.

INKREMENTÁLIS FEJLESZTÉS

Az inkrementális fejlesztés a vízesés és evolúciós modell hibrid modellje. Támogatja a folyamat iterációt, amely lényege, hogy a specifikációt a szoftverrel együtt fejlesztjük.

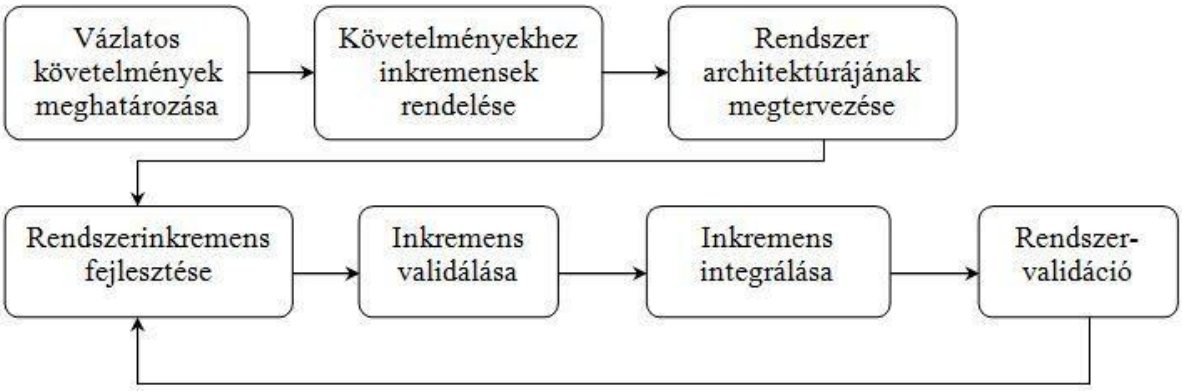
A vízesésmodell:

- hátránya: véglegesítenünk kell az egyes fázisokat mielőtt a következő fázisba belekezdünk így nem elég rugalmas a változtatásokra
- előnye: a fázisok elkülönítése miatt egyszerűen menedzselhető

Az evolúciós modell:

- előnye: elhalaszthatjuk a követelményekkel és a tervezésekkel kapcsolatos döntéseket
- hátránya: gyengén strukturált és nehezen karbantartható rendszerekhez vezethet

Az inkrementális fejlesztési megközelítés a két módszer előnyeit igyekszik kombinálni.



A megrendelő első lépésben a rendszer szolgáltatásait nagy körvonalakban határozza meg, és megadja a szolgáltatások fontossági sorrendjét. A szolgáltatásokat inkremensekben helyezik el, a nagyobb prioritású szolgáltatásokat előbb kell biztosítani a megrendelő felé. Az inkremenseken belül az előállítandó szolgáltatások követelményeit már részletesen kell definiálni. Az inkrementst úgy kell elkészíteni, hogy azt a megrendelő rögtön fel is tudja használni. Ezáltal rögtön tapasztalatot szerezhet a működéssel kapcsolatban, ami segít a többi inkremens követelményeinek meghatározásában és az aktuális inkremens későbbi verzióinak a pontosításában. Az új inkrementst mindig integrálják az előzőekkel, így a rendszerfunkciók köre fokozatosan bővül. Nincs szükség arra, hogy minden inkremensnél ugyanazt a fejlesztési modellt használják, mindegyik inkremensnél a legmegfelelőbb modell alkalmazására kerülhet sor.

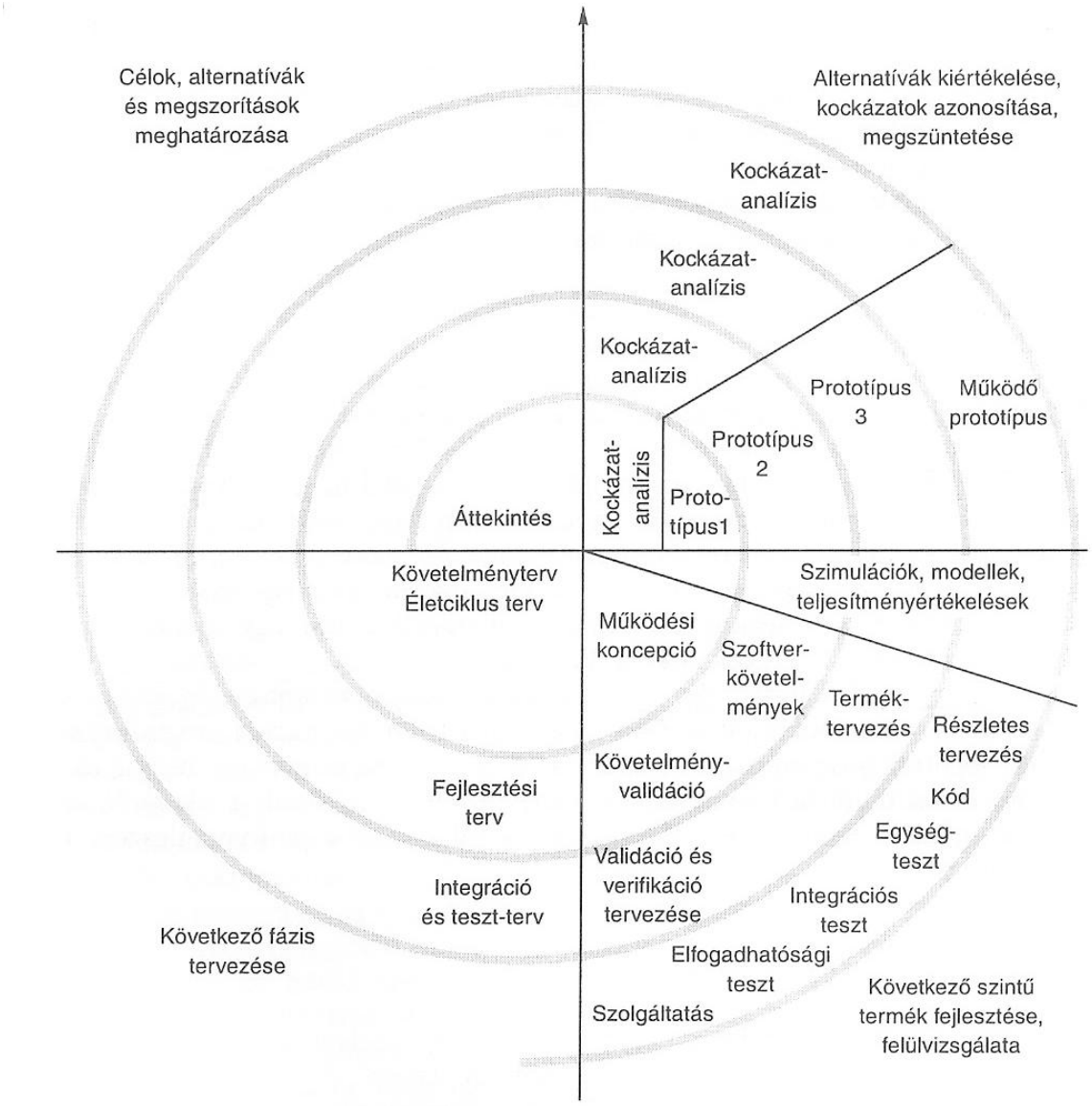
A modell előnye hogy, gyors visszacsatolást biztosít, hisz a megrendelő az első inkremens elkészülésével már működő programhoz jut. Ezáltal korán tapasztalatot szerez a rendszer működéséről, ez nagyban segíti a további inkremensek követelményeinek pontosítását. Mivel először a legfontosabb szolgáltatások készülnek el, amibe folyamatosan integráljuk bele az új szolgáltatásokat, ezért a rendszer magját teszteljük a legtöbbször. És mivel az egyes inkremensek külön is működőképesek így kisebb az esélye, hogy a teljes rendszer kudarcba fullad.

Hátrány, hogy nehéz a megfelelő inkremensek kialakítása, hisz alapvető elvárás az velük szemben, hogy, kis méretűek legyenek és hogy, valamilyen rendszerfunkciót szolgáltassanak.

Az inkrementális fejlesztés legújabb változata az extrém programozás, amely nagyon kis funkcionalitással rendelkező inkremensek fejlesztésén alapul.

SPIRÁLIS FEJLESZTÉS

A szoftverfolyamatot nem tevékenységek és a közöttük található esetleges visszalépések sorozataként tekinti, hanem inkább spirálként reprezentálja. A spirálban minden egyes kör a szoftverfolyamat egy-egy fázisát reprezentálja.



Mindemellett a spirált 4 szektorral oszthatjuk fel:

- Célok kijelölése: egy adott projektfázis által kitűzött célok meghatározása

- Kockázat becslése és csökkentése: minden egyes felismert projektkockázati tényező esetén részletes elemzésre kerül sor. Ha például fennáll a kockázata annak, hogy a követelmények nem megfelelők, akkor prototípust lehet fejleszteni.
- Fejlesztés és validálás: a kockázatok mérlegelése után egy megfelelő fejlesztési modellt választunk.
- Tervezés: A projektet áttekintjük, és eldöntjük, hogy elkezdhető-e a következő fázis. Ha igen, akkor felvázoljuk a projekt következő fázisát.

AZ UML

Az UML (Unified Modeling Language) egy grafikus nyelv szoftver rendszerek, üzleti rendszerek, és más nem szoftver rendszerek elemeinek specifikálásához, vizualizálásához, létrehozásához, és dokumentálásához.

TÖRTÉNETE

Az első szoftverfejlesztési módszerek a hetvenes évek elején jelentek meg. Tíz évvel később pedig már objektumorientált módszerekről is beszélhetünk. A folyamatos fejlődés következményeként a kilencvenes évek kezdetén már körülbelül egy tucat különböző megoldás versengett egymással. Végül a legfőbb irányzatok vezető alakjai egyesítették erejüket melynek következményeként 1996 az OMG kiadta az UML 0.9 verziót. Onnantól kezdve folyamatosan jelentek meg az 1.x verziók. A jelenleg is érvényes 2.0 verzió 2005-ben jelent meg.

DIAGRAMTÍPUSOK

- Strukturális diagramok: A strukturális diagramok a modellezett rendszer elemeire vonatkoznak. Altípusai:
    - Osztálydiagramok: A rendszerben használt osztályokat mutatja azok attribútumaival együtt. Az osztálydiagram tartalmazza továbbá az osztály szintű kapcsolatokat.
    - Komponensdiagramok: A komponensdiagram a rendszer fizikai komponenseit és az azok közötti függőségeket mutatja.

- Telepítési diagramok: A telepítési diagramok a rendszerimplementációhoz használt hardvert, a hardverre telepített szoftverkomponenseket és azok viszonyát hivatottak reprezentálni.
- Objektumdiagramok: A modellezett rendszer egy adott időpillanatbeli állapotát mutatják az objektumdiagramok. Az objektumdiagram pillanatfelvétel a rendszer állapotáról. Osztályok példányait és kapcsolatait jeleníti meg.
- Csomagdiagramok: A csomagdiagram azt mutatja, hogy hogyan szerveződnek a szoftverelemek csomagokba illetve hogyan viszonyulnak ezek a csomagok egymáshoz.

- Viselkedési diagramok: A viselkedési diagramok azt írják le, hogy minek kell történnie a modellezett rendszerben:
  - Aktivitásdiagramok: Az aktivitásdiagramok modellezik.
  - Állapotgép diagramok: Az állapotgép diagramok a rendszer lehetséges állapotait és az azok közötti átmeneteket mutatják állapotgépes ábrázolással.
  - Use case diagramok: A use case diagramok fogalmazzák meg a rendszer használati eseteit.
  - Interakciós diagramok: Az interakciós diagramok fogalmazzák meg a rendszerelemek közötti kommunikációt. Altípusai:
    - Kommunikációs diagramok
    - Interakciós áttekintő diagramok
    - Szekvenciadiagramok
    - Időzítődiagramok

ÁTTEKINTÉS

Az Állatmenhely webalkalmazás lehetővé teszi az állat menhelyek hatékony együttműködését, ezáltal megnöveli az állatok gazdára találásának esélyeit. Mivel webalkalmazásról van szó, ezért nem szükséges az egyes menhelyeken semmiféle telepítés, csupán egy böngésző kell a program eléréséhez.

Az alkalmazásban különböző állat menhelyek kerülnek felvételre. Az alkalmazás képes az egyes menhelyeken lévő állatok adatainak rögzítésére, az azokról készült képek tárolására. Képes az örökbeadásokat kezelni, tárolja az örökbefogadók, örökbefogadások adatait. Nyomon követhetők vele a menhelyen elpusztult állatok. Mindezt persze úgy teszi lehetővé, hogy az adatokhoz csak az arra jogosultak férhetnek hozzá. Az alkalmazásnak van egy nyilvános, tehát bejelentkezés nélkül elérhető része, ahol az érdeklődők megtekinthetik az éppen gazdára váró állatokat.

Az alkalmazással tehát három különböző felhasználó típus kerül kapcsolatba: látogató, ügyintéző, adminisztrátor. Az egyes csoportokkal kapcsolatos elvárások:

Látogató: Bejelentkezés nélkül képesnek kell lennie az örökbe fogadható állatok keresésére (pl. fajta, szín, állapot, vagy település szerint). Az egyes állatok adatlapjának böngészésére. Az állatok képeinek megtekintésére. Az állat menhelyének adatainak lekérésére.

Ügyintéző: Csak bejelentkezés után érheti el a neki szánt funkciókat. Egy ügyintéző egy menhelyhez tartozik és csak saját menhelyéhez tartozó állatok adatait szerkesztheti, más menhelyhez tartozó állatok adatait legfeljebb csak megtekintheti. Képesnek kell lennie már felvett állatok keresésére (különböző szempontok alapján). Az állatok adatainak megtekintésére, szerkesztésére, törlésére. Képek feltöltésére, szerkesztésére, törlésére, alapértelmezett kép beállítására. Képesnek kell lennie örökbefogadók keresésére (különböző szempontok alapján). Örökbefogadók adatai adatainak megtekintésére, szerkesztésére, törlésére. Képesnek kell lennie örökbefogadás regisztrálására, törlésére, korábbi örökbefogadások megtekintésére. Képesnek kell lennie elpusztulások regisztrálására, törlésére, korábbi elpusztulások megtekintésére.

Adminisztrátor: Csak bejelentkezés után érheti el a neki szánt funkciókat. Képes menhelyek böngészésére, felvitelére, szerkesztésére, törlésére. Képes ügyintézők böngészésére, felvitelére, szerkesztésére, törlésére. Képes új adminisztrátor felvitelére. De más adminisztrátorok adatait nem szerkesztheti, törölheti.

HASZNÁLATI ESET DIAGRAM

A használati eset diagram segítségével összegyűjthetjük és áttekinthetjük az alkalmazással szemben támasztott legfontosabb követelményeket. Segítségével rajzolhatjuk meg a rendszer határait, hogy mely funkciók kerüljenek be az alkalmazásba és melyek ne. A használati eseteket úgy kell összeállítani, hogy segítségükkel teljesíteni tudjuk az áttekintésben megfogalmazott összes felhasználói célt.

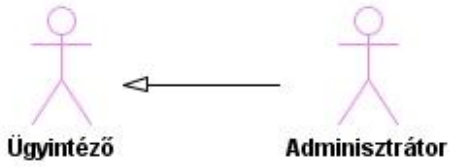
AKTOROK

Első lépésben meghatározzuk, hogy mik azok a rendszer határain kívül eső elemek, amik közvetlenül kapcsolatba kerülnek, kommunikálnak a leendő szoftver rendszerrel. Ezen elemeket aktoroknak nevezzük. Az aktor egy szerep, amit az elem játszik a rendszerrel folytatott interakcióban.

Ezek alapján az Állatmenhely alkalmazásban a látogató, az ügyintéző és az adminisztrátor a lehetséges aktor. A diagramon az aktorok jele a pálcikaember.

Érdemes megvizsgálni, hogy az egyes aktoroknak léteznek-e említésre méltó alváltozatai. A különböző változatok az alkalmazáshoz eltérő módon kapcsolódhatnak. Az UML az általánosítás viszonyt háromszögben végződő vonallal jelöli, ahol a háromszög az általánosabb elem felé mutat.

Ha a próbaalkalmazásunkban az adminisztrátor csak egy speciális ügyintéző lenne, vagyis mind azon tevékenységeket is elláthatná, amit az ügyintéző ellát, akkor közöttük általánosítás viszonyt értelmeznénk, az ügyintéző az adminisztrátor általánosítása lenne. Ezt a következőképpen kellene jelölnünk a diagramon:



De mivel esetünkben, az adminisztrátor teljesen más tevékenységeket lát el, mint az ügyintéző, nálunk általánosítás az aktorok között nem lesz.

HASZNÁLATI ESETEK

Az aktorok felderítése után már rendelkezésünkre is áll egy kezdetleges használati eset diagram. A következő lépésben minden egyes aktort sorra véve meghatározzuk, hogy mely funkciókon keresztül kapcsolódnak a rendszerhez. A rendszer ezen kapcsolódási pontjait használati eseteknek nevezzük. Ezeket a diagramon ellipszis alakzattal jelöljük.

Az aktorok és funkciók között húzott vonallal jelöljük, hogy az egyes aktorok milyen használati esetekkel kapcsolódnak a rendszerhez.

Vegyük sorra, hogy az Állatmenhely alkalmazásban az egyes aktorok hogyan kapcsolódhatnak a rendszerhez.

Látogató:

- Állatok keresése
- Állat adatainak megtekintése
- Menhely adatainak megtekintése

Ügyintéző:

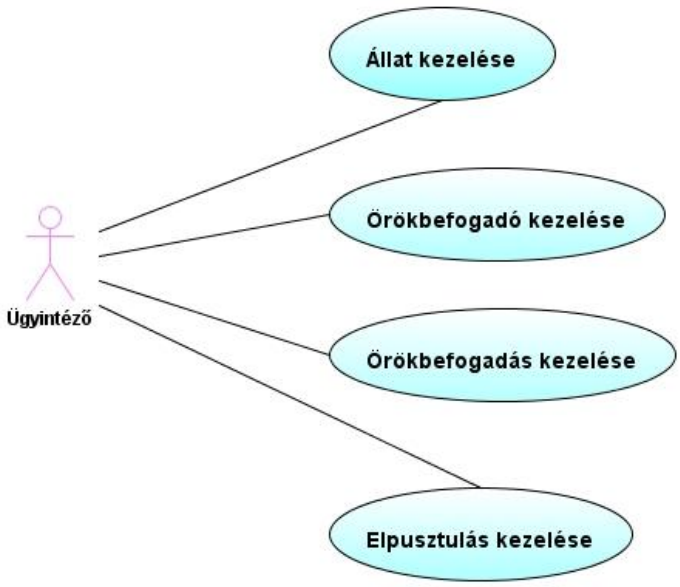
- Állatok kezelése:
    - Új állat felvitele
    - Állat szerkesztése
    - Állat törlése
    - Képek kezelése:
        - Új kép felvitele
        - Kép szerkesztése

        - Kép törlése
- Örökbefogadók kezelése:
    - Új örökbefogadó felvitele
    - Örökbefogadó szerkesztése
    - Örökbefogadó törlése
- Örökbefogadás kezelése:
    - Új örökbefogadás felvitele
    - Örökbefogadás szerkesztése
    - Örökbefogadás törlése
- Elpusztulás kezelése:
    - Új elpusztulás felvitele
    - Elpusztulás szerkesztése
    - Elpusztulás törlése

Adminisztrátor:

- Menhelyek kezelése:
    - Új menhely felvitele
    - Menhely szerkesztése
    - Menhely törlése
- Felhasználók kezelése:
    - Új felhasználó felvitele
    - Felhasználó szerkesztése
    - Felhasználó törlése

Ezek alapján az ügyintéző használati este diagramja a következőképpen nézne ki, ha egyelőre eltekintünk a részletektől:



KITERJESZTÉS, RÉSZFUNKCIÓ, ÁLTALÁNOSÍTÁS

Ahogy az aktorok között, úgy a használati estek között is lehetnek kapcsolatok. Beszélünk kiterjesztésről, részfunkcióról és általánosításról.

KITERJESZTÉS

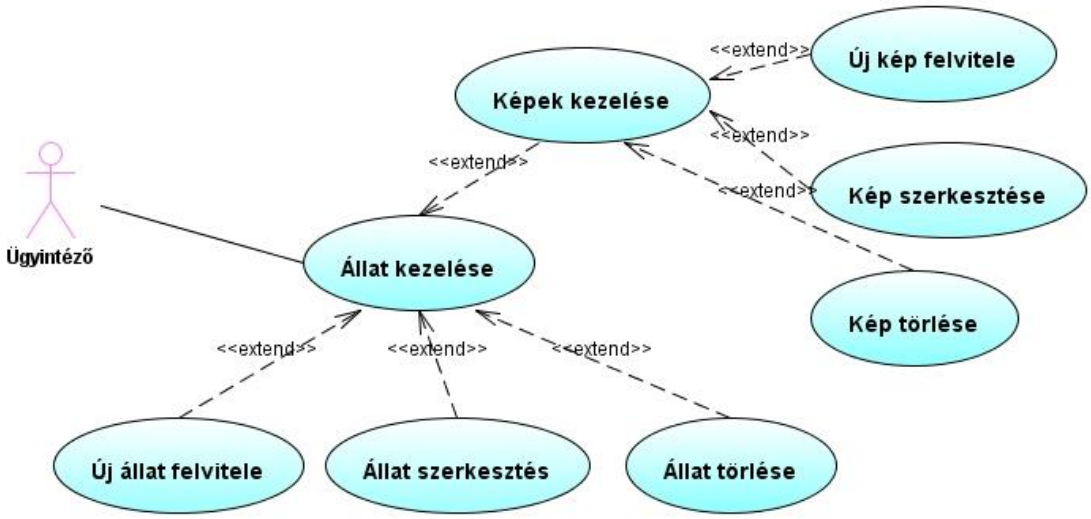
Az egyes használati eseteket kiterjeszthetjük, ezzel az alapeset működéséhez bizonyos többletet adunk. Azokat a használati eseteket jelöljük kiterjesztéssel, amelyek az alapeseten belül opcionálisak, azaz nem feltétlenül, vagy csak bizonyos feltételek esetén hajtódnak végre. Az UML diagramon a kiterjesztésből az alapesetig <<extend>> sztereotípiával jelölt szaggatott vonalú nyilat húzunk.

Ha a példaalkalmazásunkra gondolunk, akkor tudjuk, hogy az ügyintéző mikor egy állatot kezel, az a következőket jelentheti:

- Új állat felvitele
- Állat szerkesztése
- Állat törlése
- Képek kezelése:

- Új kép felvitele
- Kép szerkesztése
- Kép törlése

Kiterjesztés segítségével, az Állat kezelése használati eset részleteit is modellezni tudjuk:



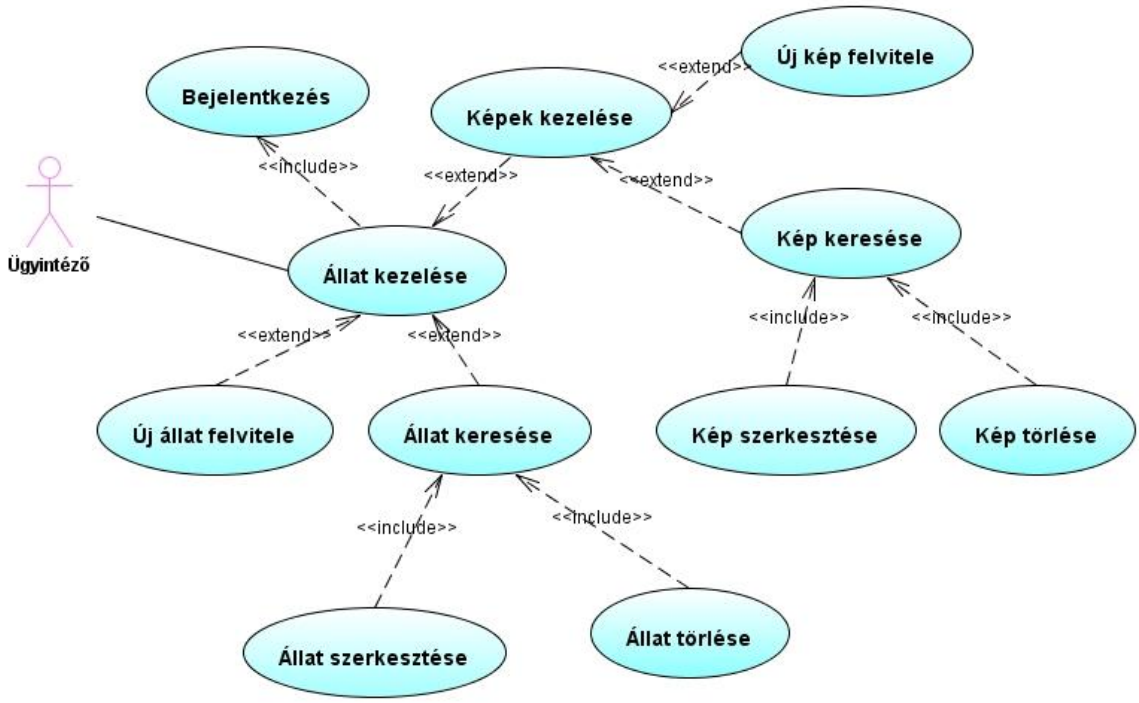
Amint látjuk az Ügyintéző aktor az Állat kezelése használati esettel áll közvetlen kapcsolatban. Azonban ez a használati eset négy másikhoz is kapcsolódik, amelyek kiterjesztik azt: Új állat felvitele, Állat szerkesztése, Állat törlése, Képek kezelése. Vagyis amikor az ügyintéző az állatokat akarja kezelni, ez jelentheti azt, hogy új állatot visz fel, már meglévő állat adatait módosítja, esetleg egy már felvitt állatot töröl. De az is lehet, hogy egy állathoz tartozó képeket kívánja kezelni. Ezek opcionálisak. Azt is láthatjuk, hogy a Képek kezelése használat esetnek is vannak kiterjesztései. Tehát a képek kezelése jelentheti egy új kép felvitelét, egy már meglévő szerkesztését, esetleg törlését.

RÉSZFUNKCIÓ

Egy használati esetet részfunkciókkal is kiegészíthetünk. Ezen funkciók az alapeset olyan részei, amelyeket valamilyen oknál fogva hangsúlyozni szeretnénk. A részfunkciókat kiemelésre is használhatjuk, ugyanis részfunkcióként ábrázoljuk a több használati eset működésében megismétlődő közös részeket. A kiterjesztésekhez hasonlóan a részfunkciók is a modellünk részletezésének eszközei. Az UML diagramon a részfunkciós kapcsolatot úgy

ábrázoljuk, hogy az alapesetből a lényegesnek tartott részfunkcióig <<include>> sztereotípiával jelölt szaggatott vonalú nyilat húzunk.

Az előző diagramunkat ezek alapján ki tudjuk egészíteni. Ugyanis előzőleg „elfelejtkeztünk" néhány igen fontos dologról. Az ügyintéző a rendszer legfontosabb funkcióit, így az állatok kezelését is, csak bejelentkezés után érheti el. És azt sem vettük figyelembe, hogy mielőtt az ügyintéző egy már meglévő állatot kíván szerkeszteni vagy törölni előbb még meg kell találni azt a rendszerben. A kiegészített diagram a következőképpen néz ki:



Amint látjuk megjelent a Bejelentkezés használat eset, ami az Állat kezelése részfunkciója. Tehát ahhoz, hogy az Ügyintéző az állatokat kezelhesse, előbb be kell jelentkeznie. Valamint megjelent az Állatok keresése használat eset, az Új állat felvitele és Állat szerkesztése részfunkciójaként. Tehát mielőtt egy már meglévő állatot szerkesztenénk vagy törölnénk előbb azt meg kell keresnünk. Ugyanezen oknál fogva jelent meg a Kép keresése részfunkció is.

ÁLTALÁNOSÍTÁS

Az aktorokhoz hasonlóan a használati eseteknél is értelmezzük az öröklődési viszonyt. A leszármazott használati eset örökli a szülő funkció viselkedését és kapcsolatait és természetesen ezeket ki is egészítheti, esetleg felüldefiniálhatja. Pontosításként jelölhetjük az általános megvalósítási lehetőségeit, illetve általánosként ábrázolhatjuk több használati eset hasonló jellegét. Az UML az általánosítás-pontosítás viszonyt a pontosított elemtől az általánosig húzott, háromszögben végződő vonallal jelöli.

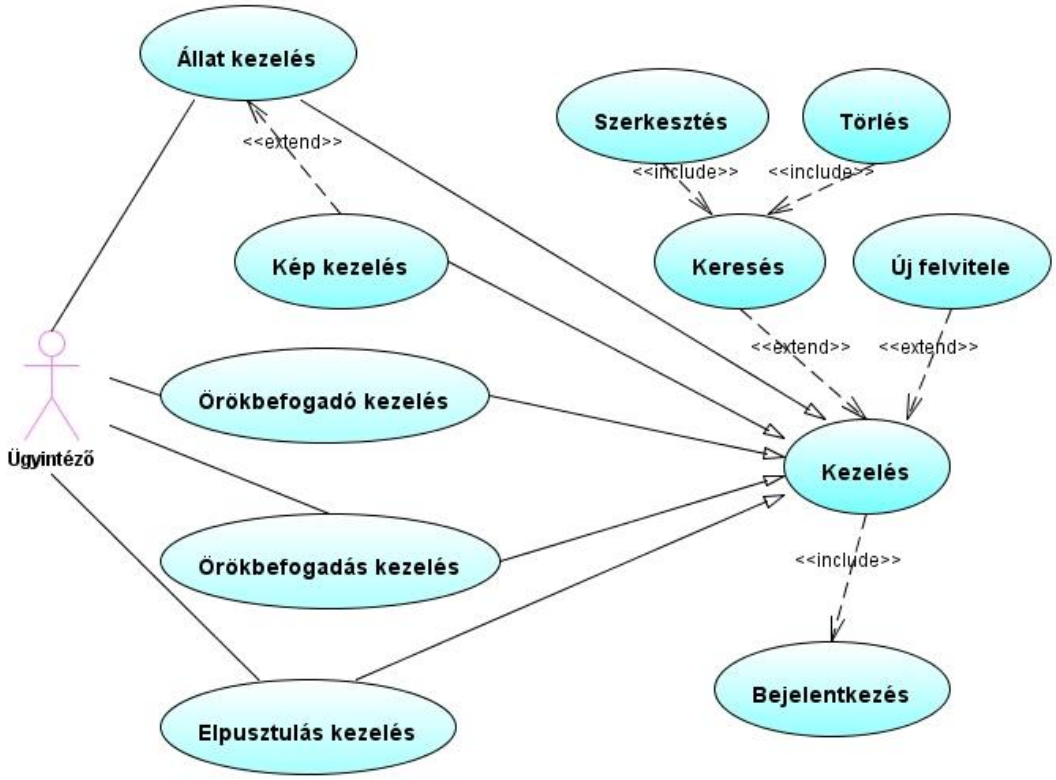
Ha az ügyintéző használati eseteit vizsgáljuk, akkor megfigyelhetjük, hogy a tevékenységek öt csoportba sorolhatók:

- Állatok kezelése
    - Képek kezelése
- Örökbefogadók kezelése
- Örökbefogadások kezelése
- Elpusztulás kezelése

És minden csoporton belül a következő tevékenységek szerepelnek:

- Új létrehozása
- Szerkesztés
- Törlés

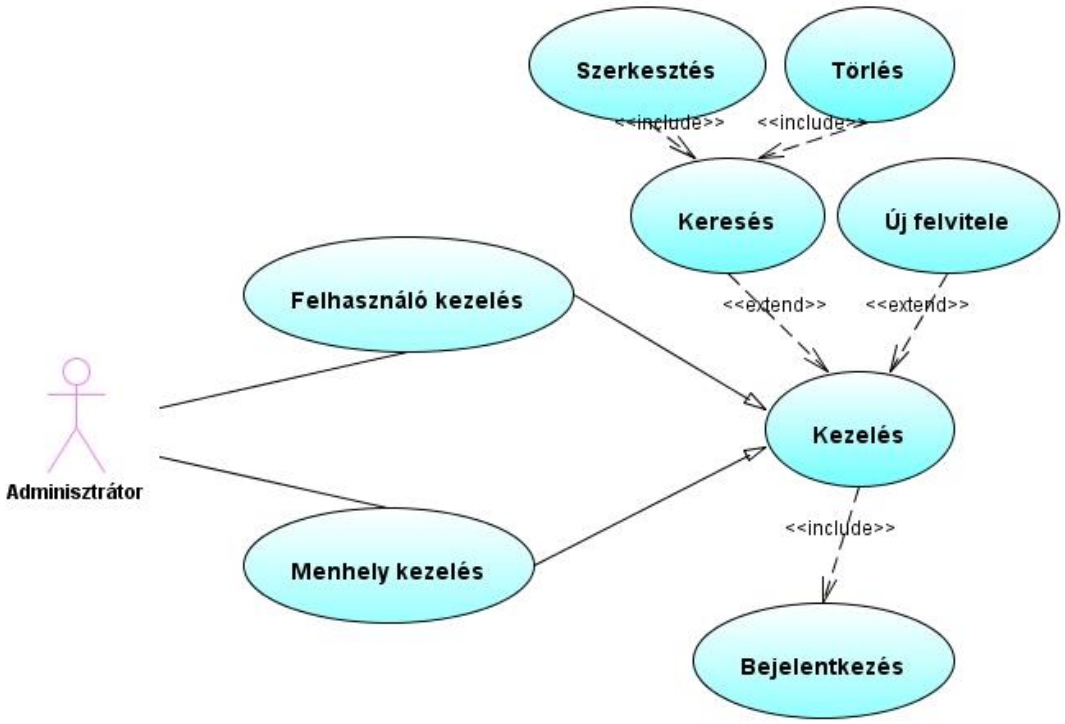
Logikusnak tűnik egy új használati eset létrehozása, ami az egyes kezelés használati esetek általánosítása lesz. Ezek alapján a diagram:



Láthatjuk, hogy az Ügyintéző az Állat kezelés, Örökbefogadó kezelés, Örökbefogadás kezelés és Elpusztulás kezelés használati eseteken keresztül kapcsolódik a rendszerhez. Az Állat kezelésnek van egy kiterjesztése, a Kép kezelés. Vagyis az állat kezelése során az ügyintézőnek lehetősége van az ahhoz tartozó képek kezelésére is. Mindezen használati esetek a Kezelés pontosításai. Ugyanis ezen használati esetek ugyanolyan részfunkciókkal és kiterjesztésekkel rendelkeznek. Minden kezelés részfunkciója a Bejelentkezése, vagyis ahhoz, hogy az ügyintéző bármit kezelni tudjon, előbb még be kell jelentkeznie. A Kezelésnek két kiterjesztése van: Keresés és Új felvitele. A Keresés a Szerkesztés és Törlés részfunkciója, vagyis ahhoz, hogy töröljünk vagy szerkesszünk valamit, előbb meg kell keresnünk. Ezek után láthatjuk, hogy a kezelés nem jelent mást, mint új felvitelét, szerkesztést vagy törlést.

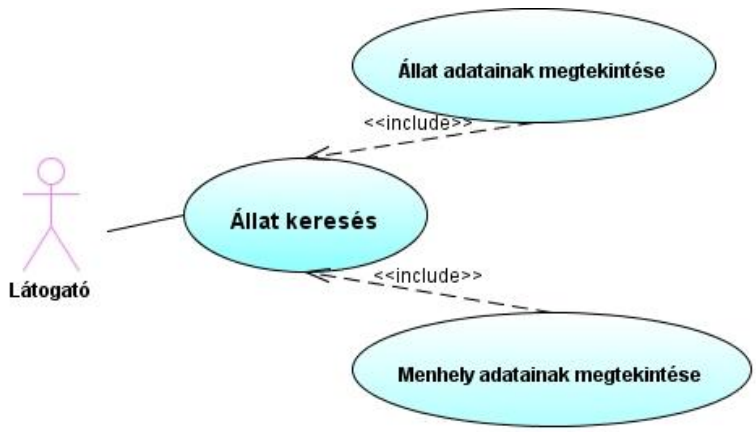
Ezen általánosítással nem csak érthetőbbé és átláthatóbbá vált a diagramunk, de még a méretét is nagymértékben lecsökkentettük.

Most tekintsük meg az adminisztrátor használati eset diagramját:



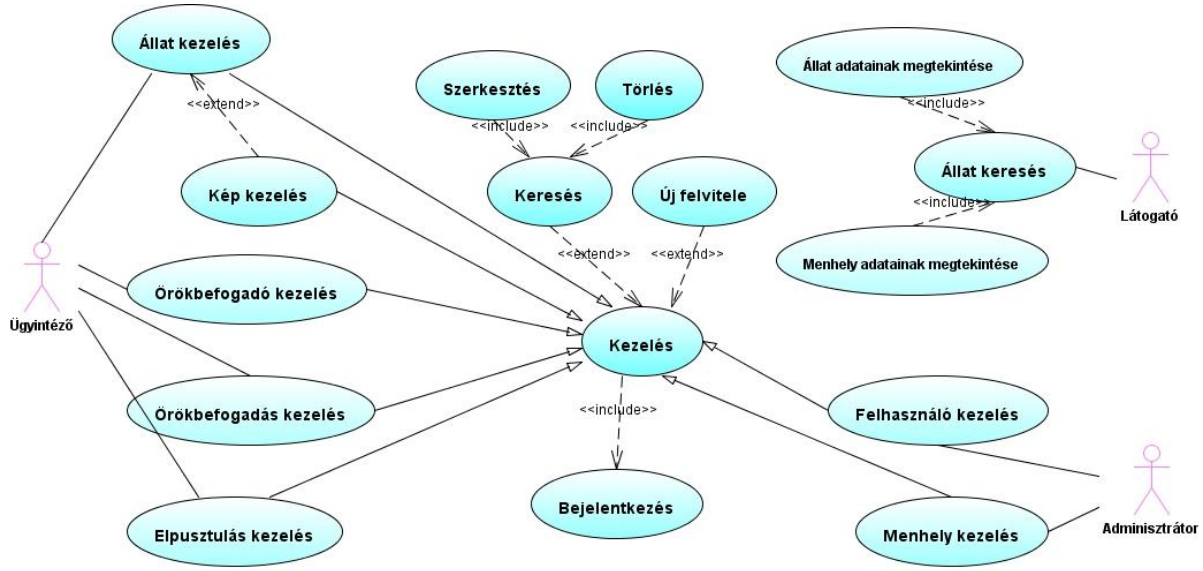
Láthatjuk, hogy az Adminisztrátor a Felhasználó kezelés és Menhely kezelés használati eseteken keresztül kapcsolódik a rendszerhez. Ezek a Kezelés pontosításai. Ugyanis ezen használati esetek ugyanolyan részfunkciókkal és kiterjesztésekkel rendelkeznek. Minden kezelés részfunkciója a Bejelentkezése, vagyis ahhoz, hogy az adminisztrátor bármit kezelni tudjon, előbb még be kell jelentkeznie. A Kezelésnek két kiterjesztése van: Keresés és Új felvitele. A Keresés a Szerkesztés és Törlés részfunkciója, vagyis ahhoz, hogy töröljünk vagy szerkesszünk valamit, előbb meg kell keresnünk. Ezek után láthatjuk, hogy a kezelés nem jelent mást, mint új felvitelét, szerkesztést vagy törlést.

A látogató használat eset diagramja:



A Látogatónak nem kell bejelentkezni a neki szánt funkciók eléréséhez. A látogató miután megkeresett egy állatot megtekintheti az ahhoz tartozó adatokat, képeket és megtekintheti annak a menhelynek az adatlapját ahol az adott állat található.

Végül tekintsük meg összes aktort és használati esetet tartalmazó végleges diagramot:



A KÖVETELMÉNYELEMZÉS SZÖVEGES DOKUMENTUMAI

A követelmények pontosítása érdekében érdemes szöveges dokumentumokat is készítenünk, melyekkel konkrét formába önthetjük a felhasználók igényeit. A kiegészítésekkel részletesebb képet alkothatunk a kifejlesztendő alkalmazásról. Ilyen szöveges dokumentumok az alkalmazási példák, a forgatókönyvek és a működési leírások, felhasználói felületek. Ezen dokumentumok nem részei az UML-nek, de talán érdemes egy kicsit foglalkozni velük, mivel nagyon hasznosak lehetnek a megrendelőkkel történő egyeztetések során.

ALKALMAZÁSI PÉLDÁK

A követelményeket kiegészíthetjük olyan alkalmazási példákkal, amelyekkel szemléltethetjük, hogy a használat esetek milyen módon alkalmazhatók az egyes felhasználói célok megvalósítására. Segítségükkel a használat estek logikai kapcsolatait, időbeli egymásutániságukat ábrázolhatjuk és rögzíthetjük az érvényes sorrendeket.

Örökbefogadás alkalmazási példája:

- A látogató ellátogat az Állatmenhely weboldalára és keres magának egy állatot, ami szeretne örökbe fogadni
- Megtekinti az állat menhelyének adatait, így most már tudja hova kell érte mennie
- A személy megjelenik a menhely telephelyén és jelzi az ügyintézőnek, hogy melyik állatot szeretné elvinni
- Ha a személy most fogad először örökbe, vagyis nincs még rögzítve a rendszerben:
    - Az ügyintéző felviszi az örökbefogadó adatait és rögzíti új örökbefogadóként
- Az ügyintéző rögzíti az új örökbefogadást
- Az örökbefogadó elviszi az állatot
- Az állat lekerül a gazdira váró állatok listájáról

FORGATÓKÖNYVEK

A követelményeket úgy is pontosíthatjuk, ha az egyes használati esetekre fordítjuk a figyelmet. A részletezés egy egyszerű módja, ha minden használati esethez megadunk egy vagy több forgatókönyvet, amelyben felsoroljuk, hogy a funkció milyen, az aktor és az alkalmazás között lezajló párbeszédet igényel.

Az ügyintéző felvisz egy új képet egy állathoz:

- Az ügyintéző bejelentkezik
- Kiválasztja az állat keresése funkciót, megadja a keresési feltételeket és keres
- Megjelenik a keresés eredménye és az ügyintéző kiválasztja a keresett állatnál a képek kezelése funkciót
- Megjelenik az eddig felvitt képek listája és az ügyintéző kiválasztja az új kép felvitelét

- Feltölti a képet és megadja a szükséges adatokat

MŰKÖDÉSI LEÍRÁSOK

A forgatókönyvek nem alkalmasak olyan funkciók ábrázolására, amelyek viszonylag kevés interakcióra, ugyanakkor több és összetettebb tevékenységsorozatra épülnek. Ezek leírására a működési leírás a legalkalmasabb dokumentum. A működési leírások különösen akkor hasznosak, ha a használati este valami olyan folyamatot indít be, amely az aktortól részben függetlenül működik.

FELHASZNÁLÓI FELÜLETEK

A forgatókönyvek egyes elemei a felhasználóval vagy külső rendszerrel történő információátadást írnak le. A használati estek pontosításának következő lépéseként célszerű megtervezni az információátadás eszközeit, a felhasználói felületeket.

Látogatói felülettervek:

- Állat keresése felületterv:

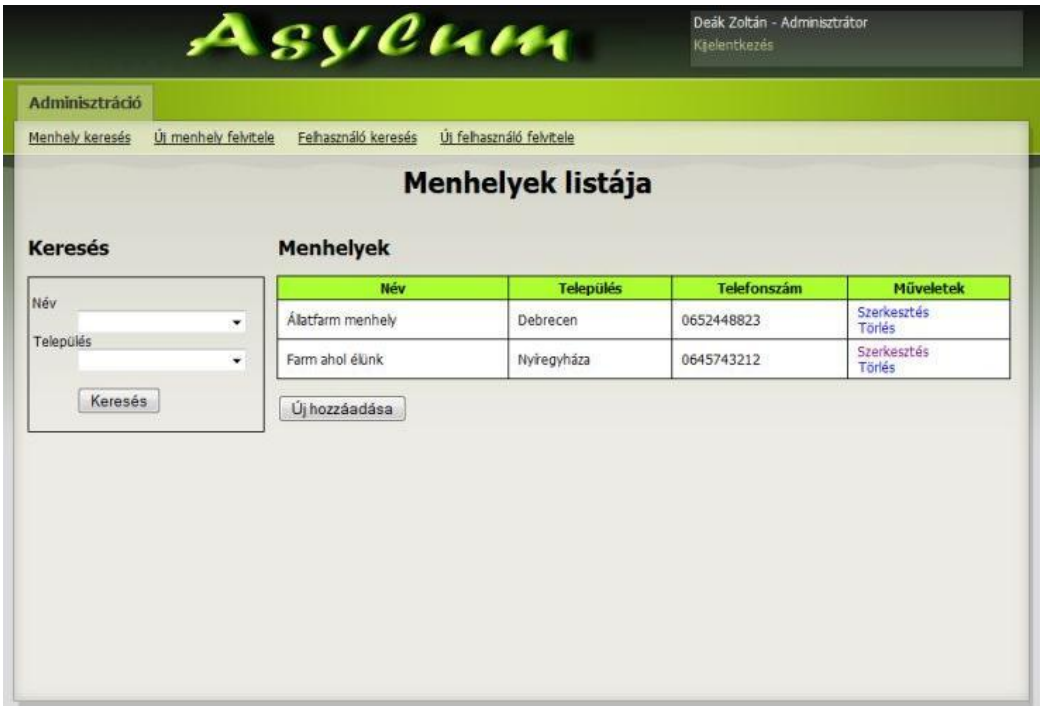
Ügyintézői felülettervek:

- Állatok keresése felületterv:

- Állat szerkesztése felületterv:

Adminisztrátori felülettervek:

- Menhelyek keresése:



- Új felhasználó felvitele:

OSZTÁLYDIAGRAMOK

A használat eset diagramok segítségével az alkalmazással szemben támasztott követelményeket tisztázhatjuk. Az alkalmazást kívülről látható képét jelenítik meg grafikus módon. Tehát a felhasználó szemszögéből ábrázolják a rendszert. A használati esetek köré szervezzük a követelményeket pontosító többi dokumentumot.

Az alkalmazás belső szerkezetének ábrázolása szempontjából hasonló szerepűek az osztálydiagramoknak, melyek így nem a felhasználó, hanem a fejlesztő szemszögéből alapvető jelentőségűek. Osztálydiagramokkal ábrázoljuk az alkalmazás belső felépítését, az architektúrát és azok köré szervezünk több diagramtípust, melyekkel a nem strukturális viszonyokat szemléltetjük.

Az osztálydiagram a legismertebb objektum-orientált modellezési technika. Az OO módszertanok legalapvetőbb eszköze, a rendszer objektumait leíró osztályokat és a közöttük levő kapcsolatokat ábrázolja.

MNV ARCHITEKTÚRA

Az MNV, vagyis Modell-Nézet-Vezérlő (Model-View-Controller – MVC) egy szoftvermérnöki munkában használt szerkezeti minta. Összetett, sok adatot a felhasználó elé táró számítógépes alkalmazásokban gyakori fejlesztői kívánalom az adathoz (modell) és a felhasználói felülethez (nézet) tartozó dolgok szétválasztása, hogy a felhasználói felület ne befolyásolja az adatkezelést, és az adatok átszervezhetők legyenek a felhasználói felület változtatása nélkül. A modell-nézet-vezérlő ezt úgy éri el, hogy elkülöníti az adatok elérését és az üzleti logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.

Az Állatmenhely alkalmazást az MNV architektúra alapján tervezzük meg. Kezdetben csak a modell réteggel foglalkozunk, de miután megismerkedtünk osztálydiagramokkal kicsit részletesebben rátérünk a többi rétegre is.

OSZTÁLY

Az alkalmazás belső szerkezete a valóságnak a szakterület által lefedett részletén alapszik. Tehát az alkalmazás alapszerkezetét úgy írhatjuk le, hogy megkeressük a valóság ezen részének építőelemeit és felderítjük a köztük lévő viszonyokat.

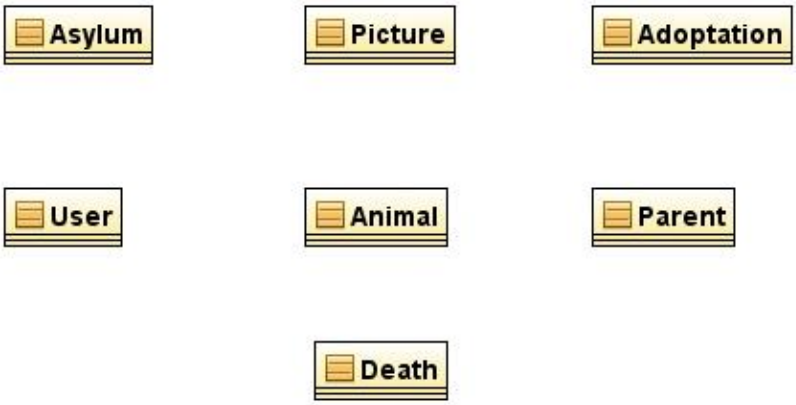
Az objektum-orientált világban objektumoknak vagy példányoknak nevezzük a valóság alapelemeit. Az objektumok a valós világ elemihez hasonlóan jellemzőkkel bírnak. Tulajdonságaik vannak, ismert az állapotuk, leírható a viselkedésük. Azonban az objektumokat általában nem egyenként és külön definiáljuk, hanem az azonos jellegzetességű példányoknak a típusát adjuk meg, amit osztálynak nevezünk. Az osztály az azonos jellegzetességű, más szóval egy típusba tartozó objektumokat csoportosítja, azaz egy absztrakt objektumot határoz meg.

Az UML jelöléseivel az osztályt egy téglalappal jelöljük.

Ha az Állatmenhely alkalmazás modell rétegét tekintjük, akkor a következő osztályokat tudjuk elkülöníteni:

- Menhely (Asylum) osztály:
    - Ez az osztály a felelős a menhelyek adatainak tárolásáért
- Felhasználó (User) osztály:
    - Ez az osztály a felelős a felhasználók adatainak tárolásáért
- Állat (Animal) osztály:
    - Ez az osztály a felelős az álatok adatainak tárolásáért
- Kép (Picture) osztály:
    - Ez az osztály a felelős az állatokhoz tartozó képek adatainak tárolásáért
- Elhalálozás (Death) osztály:
    - Ez az osztály a felelős az állat elpusztulásával kapcsolatos adatainak tárolásáért
- Örökbefogadó (Parent) osztály:
    - Ez az osztály a felelős az örökbefogadók adatainak tárolásáért
- Örökbefogadás (Adoptation) osztály:
    - Ez az osztály a felelős az örökbefogadásokkal kapcsolatos adatok tárolásáért

Ezek alapján a következő kezdetleges osztálydiagramot tudjuk felrajzolni:



ATTRIBÚTUMOK

Az objektumok meghatározott tulajdonságokkal, jellemző sajátosságokkal, úgynevezett attribútumokkal rendelkeznek. Tehát egy attribútum az objektum egy adott szempontból vett tulajdonságát írja le.

Az UML jelöléseivel az osztály attribútumait a névrész alatt, attól egy vonallal elválasztva adhatjuk meg a következő formában:

név : típus = kezdeti érték

Ezt a következőképpen értelmezhetjük:

- az attribútum név elnevezi a szempontot
- egy objektum konkrét attribútum értéke megadja, hogy az objektum milyen az adott szempontból
- az attribútum típusa meghatározza a lehetséges értékeket
- az opcionális kezdeti érték az objektum készítésekor beállítandó értékeket jelenti

Az Állatmenhely alkalmazás modell rétegének osztályainak attribútumai:

- Menhely (Asylum) osztály:
    - name: név
    - address: cím
    - phone: telefonszám

    - email: email cím
- Felhasználó (User) osztály:
    - username: felhasználói név
    - password: jelszó
    - name: név
    - identifier: személyi azonosító szám
    - address: cím
    - phone: telefonszám
    - type: felhasználó típusa (ügyintéző vagy adminisztrátor)
- Állat (Animal) osztály:
    - identifier: állat azonosító
    - name: név
    - kind: faj
    - breed: fajta
    - color: szín
    - birthYear: születési év
    - state: állapot
    - arrivalDate: érkezés dátuma
    - location: elhelyezés
    - note: megjegyzés
- Kép (Picture) osztály:
    - default: alapértelmezett kép-e
    - path: elérési út
    - note: megjegyzés
- Elhalálozás (Death) osztály:
    - date: dátum
    - note: megjegyés
- Örökbefogadó (Parent) osztály:
    - identifier: személyi azonosító szám
    - name: név
    - address: cím
    - phone: telefonszám

- Örökbefogadás (Adoptation) osztály:
    - date: dátum
    - note: megjegyzés

ATTRIBÚTUM SZÁMOSSÁGA

Attribútumok esetén megadható a számosság is:

- lehet egy számérték (például 5)
- lehet egy a..b formában megadott értéktartomány (pl. 1..10)
- lehet ezek vesszővel elválasztott sorozata (pl. 1,2,3, 10..15)

A * a végtelent jelöli. Ezt szögletes zárójelek között tehetjük meg a típus után. Attribútumok esetén alapértelmezettnek az UML az 1 számosságot tekinti. A szögletes-zárójelek megegyeznek bizonyos programozási nyelvek tömb jelölésével, itt azonban mindössze arra utalnak, hogy az adott attribútumhoz több érték is tartozhat.

A leggyakrabban használt számosságok:

- 1: pontosan 1
- *: tetszőleges, 0 vagy több
- 0..1: opcionális, 0 vagy 1
- n: pontosan megadott számérték
- 0..n: legfeljebb egy adott számérték

LÁTHATÓSÁG

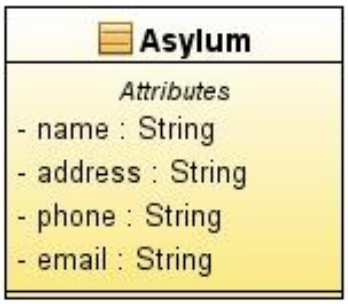
Az objektumorientált világban a bezárás technikája segítségével lehetőségünk van arra, hogy az objektumnak meghatározzuk a külvilág számára látható részét és elrejtsük annak saját attribútumait és műveleteit.

Az UML a láthatóság három szintjét definiálja:

- + (public): publikus, mindenki által látható modellelem
- # (protected): védett elemek, amelyek az objektum határain kívül nem láthatóak
- - (private): privát, vagy saját elemek

A láthatóság jelét a modellelemek elnevezése elé írjuk.

Ezek alapján a Menhely osztályt a következőképpen ábrázolhatnánk az osztálydiagramon:



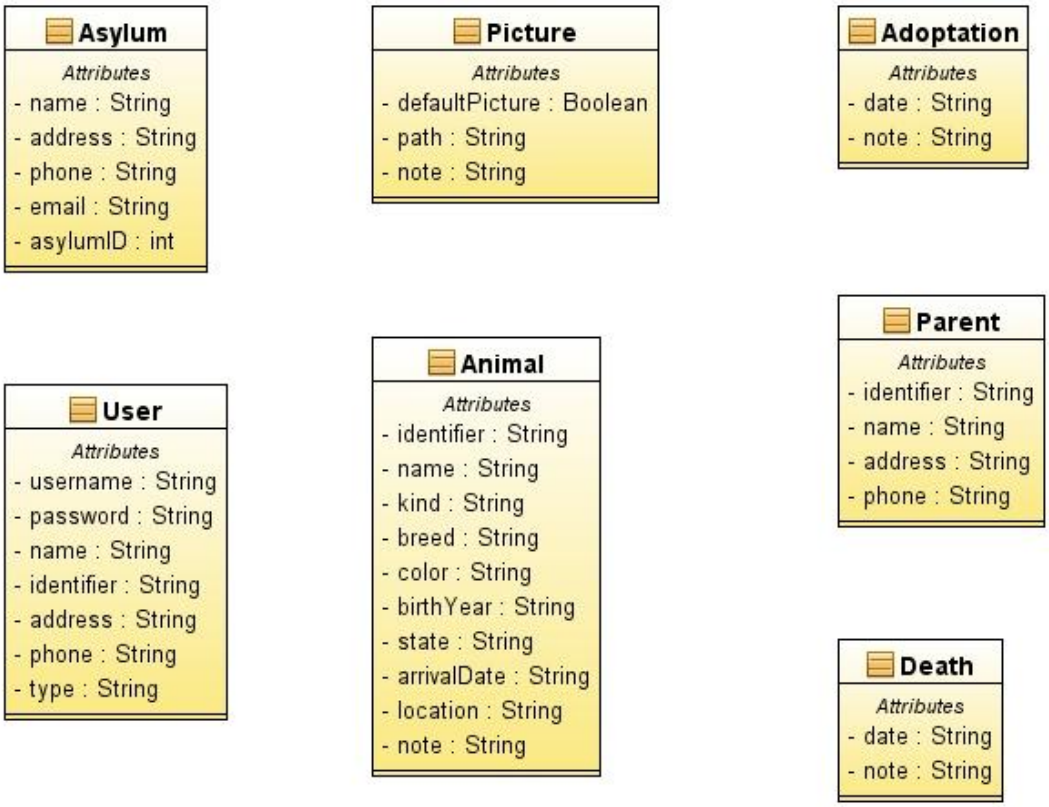
Az osztály neve alatt találhatók az attribútumok. A jelöléseket a következőképpen értelmezhetjük:

attribútum neve	láthatóság	típus	multiplicitás
name	private	String	1
address	private	String	1
phone	private	String	1
email	private	String	1

Ha megengednénk, hogy egy menhelyhez több telefonszámot is meg lehessen adni, akkor ezt a következőképpen jelölnénk:

- phone : String[0..*]

Most nézzük meg az alkalmazásunk modell rétegének osztálydiagramját, ami már az attribútumokat is tartalmazza:



MŰVELETEK

Az objektumokat úgy is jellemezhetjük, hogy megadjuk azok időbeli változását, más néven viselkedését. Az összetett változást műveletekre tördelve írjuk le. A művelet egyszerűen egy olyan eljárás, amelyet az objektum végre tud hajtani, vagy olyan kérdés, amelyre az tud válaszolni.

A műveletet a nevével és paramétereinek típusával azonosítjuk, amelyet együttesen a művelet szignatúrájának nevezzük. Mivel az azonosítás a paraméterek típusával együtt történik, ezért több műveletnek is lehet azonos a neve, ha a paramétereik különböznek, ez nevezzük túlterhelésnek.

A művelet megvalósítása a metódus. Egy osztály minden objektuma azonos műveletekkel rendelkezik. A metódusok segítségével végzünk műveleteket a tulajdonságokon.

Az UML a műveleteket az attribútumok alatt sorolja fel, azoktól egy újabb vonallal elválasztva.

Egy műveletet a következőképpen adunk meg:

név (paraméterek) : típus

A név a művelet neve, az opcionálisan megadható típus a visszatérési érték típusa.

A paramétereket vesszővel elválasztva a következő formában soroljuk fel:

jelleg név: típus = alapértelmezett érték

- a jelleg az in, out vagy inout kulcsszó egyike, amelyek közül az alapértelmezett az in.
    - in: érték szerinti paraméterátadás
    - out: eredmény szerinti paraméterátadás
    - inout: érték-eredmény szerinti paraméterátadás
- a név a paraméter neve
- a típus a paraméter által felvehető értékeket határozza meg
- az alapértelmezett érték pedig opcionális

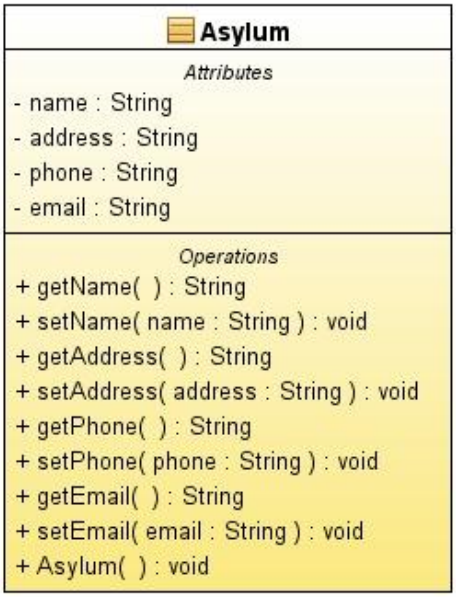
A műveletek speciális fajtái:

- Beállító: valamely attribútum értékét állítja be.
    - Prefixe: set
    - Pl. setName() – Beállítja az objektum Name attribútumának értékét
- Lekérdező: valamely attribútum értékét kérdezi le.
    - Prefixe: get illetve is
    - Pl. getName() – Lekérdezi az objektum Name attribútumának értékét
- Konstrukció: Adott osztály objektumának létrehozása. Az osztályoknak általában megadható a konstrukciós művelete. Ez a konstruktor, amely legtöbbször az attribútumait és kapcsolatait állítja be egy kezdeti értékre.
- Destrukció: Adott osztály objektumának lebontása. A destrukciós művelet a destruktor, amelyben az objektum megszüntetésekor végrehajtandó tevékenységeket adhatjuk meg.

Most térjünk vissza kicsit a próbaalkalmazásunkhoz. Az előzőekben felsoroltuk a modell réteg osztályait és azok attribútumait. Most a műveletek következnek. Mivel minden

attribútumot privát láthatósággal hoztunk létre ezért szükségesnek tűnik, hozzájuk publikus lekérdező és beállító műveleteket definiálni, hogy valamilyen módon hozzáférjünk az attribútumokhoz, az objektumon kívülről is. Tehát miden egyes attribútumhoz létre kell hoznunk egy publikus gettes és setter metódust. Valamint minden osztályba szükségünk lesz egy konstruktorra.

Most nézzük meg a Menhely osztály hogyan nézne ki a diagramon, kiegészítve a műveletekkel:



Az attribútumok alatt találhatók a műveletek. A jelöléseket a következőképpen értelmezhetjük:

művelet neve	láthatóság	paraméterek	visszatérési típus	művelet típusa
getName	public		String	lekérdező
setName	public	name: String		beállító
getAddress	public		String	lekérdező
setAddress	public	address: String		beállító
getPhone	public		String	lekérdező
setPhone	public	phone: String		beállító
getEmail	public		String	lekérdező
setEmail	public	email: String		beállító

Asylum	public			konstruktor

ASSZOCIÁCIÓ

Az osztályoknak általában együtt kell működniük más osztályokkal, hogy szolgáltatásaikat biztosítani tudják. Tehát az osztályoknak egymással kapcsolatot kell létesíteni. Az asszociáció az osztályok közötti kapcsolat leírása, absztrakciója. Az asszociáció tehát osztályok közötti viszony, míg a kapcsolatot objektumok között értelmezzük.

Az UML diagramon két osztály közötti kapcsolatot a két osztályt összekötő vonal reprezentálja.

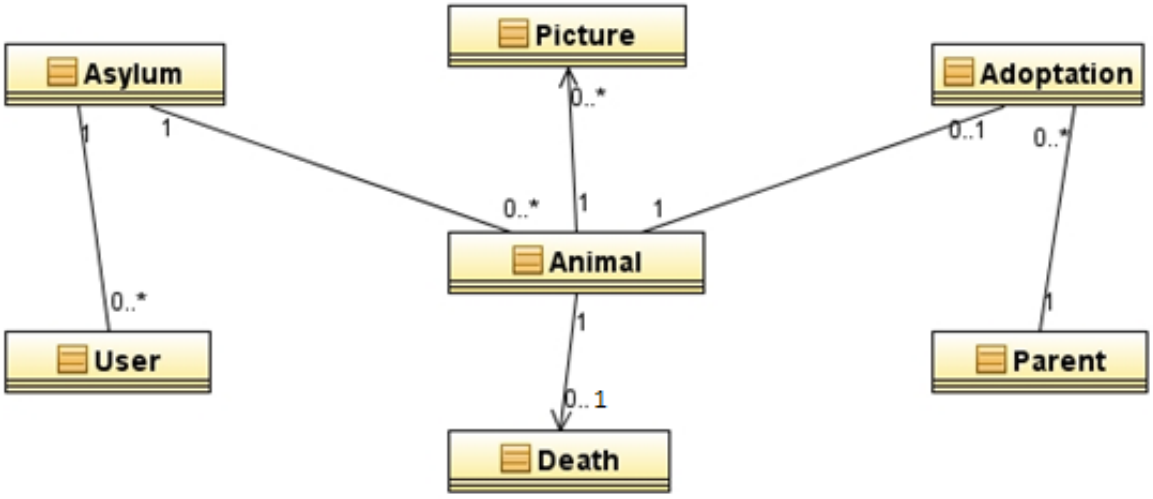
Az asszociációhoz név rendelhető: ezt az osztályokat összekötő vonal fölé, középre helyezve írjuk.

Megadhatjuk az osztályoknak az asszociációban játszott szerepét: minden kapcsolathoz két szerep rendelhető, amelyek az asszociáció két végén lévő osztályoknak az adott asszociációban betöltött szerepére vonatkoznak.

A kapcsolat fokának megadásával jelölhetjük, hogy hány objektum vehet részt az asszociációban: a multiplicitás kifejezi, hogy az egyik osztály egy objektumához a másik osztályból hány osztály tartozik, vagyis kifejezi, hogy az osztályok objektumai milyen számosságban kapcsolódnak egymáshoz.

A navigálhatóság iránya, az asszociáció bejárhatóságának iránya: a kapcsolatok mentén kommunikáció zajlik, ami lehet egyirányú vagy kétirányú. A kommunikáció irányának jelölésére az osztályokat összekötő vonalra nyilat helyezünk. A navigáció azért fontos, mert megadja, hogy az asszociációval összekötött osztályok közül melyik kezdeményezi a kommunikációt, melyik osztály objektumainak kell ismernie a másik osztály objektumait.

Ezek alapján az osztálydiagramunk:



Nézzük sorba az asszociációkat:

Asylum - User	Az asszociációban a User osztály multiplicitása 0..* ugyanis azt szeretnénk, hogy egy menhelyhez tetszőleges számú felhasználó tartozhasson. Az Asylum osztályé 1, ugyanis azt szeretnénk, ha egy felhasználó pontosan egy menhelyhez tartozna. A kapcsolat kétirányú mivel azt szeretnénk, hogy egy menhely objektum tudná, mely felhasználók tartoznak hozzá, és ha egy felhasználó objektum is tudná, melyik menhelyhez tartozik.
Asylum - Animal	Az asszociációban az Animal osztály multiplicitása 0..* ugyanis azt szeretnénk, hogy egy menhelyhez tetszőleges számú állat tartozhasson. Az Asylum osztályé 1, ugyanis azt szeretnénk, ha egy állat pontosan egy menhelyhez tartozna. A kapcsolat kétirányú mivel azt szeretnénk, hogy egy menhely objektum tudná, mely állatok tartoznak hozzá, és ha egy állat objektum is tudná, melyik menhelyhez tartozik.
Animal – Picture	Az asszociációban a Picture osztály multiplicitása 0..* ugyanis azt szeretnénk, hogy egy állathoz tetszőleges számú kép tartozhasson. Az Animal osztályé 1, ugyanis azt szeretnénk, ha egy kép pontosan egy állathoz tartozna.

	A kapcsolat egyirányú mivel csak azt tartjuk fontosnak, hogy egy állat objektum tudja, mely képek tartoznak hozzá.
Animal - Death	Az asszociációban a Death osztály multiplicitása 0..1 ugyanis egy állat, vagy elpusztul, vagy nem, tehát vagy tartozik hozzá Death objektum vagy nem. Az Animal osztályé 1, ugyanis azt szeretnénk, ha egy elhalálozás pontosan egy állathoz tartozna. A kapcsolat egyirányú mivel csak azt tartjuk fontosnak, hogy egy állat objektum tudja, tartozik-e hozzá Death objektum.
Animal – Adoptation	Az asszociációban az Adoptation osztály multiplicitása 0..1 ugyanis egy állatot, vagy örökbe fogadták, vagy nem, tehát vagy tartozik hozzá Adoptation objektum vagy nem. Az Animal osztályé 1, ugyanis azt szeretnénk, ha egy örökbefogadás pontosan egy állathoz tartozna. A kapcsolat kétirányú mivel azt szeretnénk, hogy egy állat objektum tudná, tartozik-e hozzá Adoptation objektum, és ha egy örökbefogadás objektum is tudná, melyik állathoz tartozik.
Parent - Adoptation	Az asszociációban az Adoptation osztály multiplicitása 0..* ugyanis azt szeretnénk, hogy egy örökbefogadóhoz tetszőleges számú örökbefogadás tartozhasson. Az Parent osztályé 1, ugyanis azt szeretnénk, ha egy örökbefogadás pontosan egy örökbefogadóhoz tartozna. A kapcsolat kétirányú mivel azt szeretnénk, hogy egy örökbefogadó objektum tudná, mely örökbefogadások tartoznak hozzá, és ha egy örökbefogadás objektum is tudná, melyik örökbefogadóhoz tartozik.

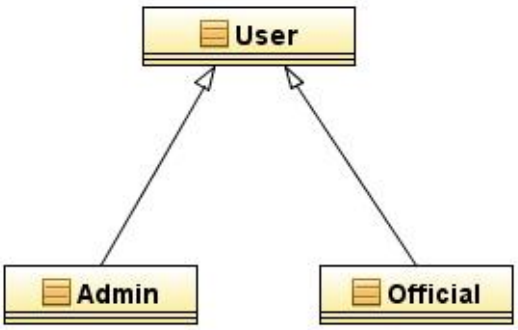
ÖRÖKLŐDÉS

Az öröklődés az OO szemlélet egyik legfontosabb eleme. Az osztályok között értelmezett öröklődési viszonyban az utód osztály sajátjaként kezeli a nála magasabb szinten lévő osztály attribútumait és műveleteit. Jele egy üres háromszögben végződő nyíl, a háromszög csúcsa az ős osztálynál található.

Öröklődési viszonyt kétféleképpen definiálhatunk a modellünkben:

- Általánosítás: A különböző objektumok sokszor tartalmaznak közös jellemzőket. Az általánosítás az a folyamat, amikor ezeket a jellemzőket kiemeljük egy ős osztályba. Az általánosítás eredményeképpen létrejön egy általános/közös sajátosságokat tartalmazó ős vagy szülő osztály, amelyhez tartozik egy vagy több speciális tulajdonságokkal rendelkező al vagy gyerek osztály.
- Specializáció: A specializáció az a folyamat, amikor meglévő osztályokból származtatott osztályokat képzünk finomítással. A finomítás célja az osztályok specifikációjának pontosítása, az objektumok egyedi jellegének megerősítése az egyedi jellegre utaló jellemzők definiálásával.

Ha vesszük az alkalmazásunkat, akkor tudjuk azt, hogy van egy User osztályunk, amely a rendszer felhasználóinak adatainak tárolására szolgál. Tudjuk azt is, hogy kétféle felhasználónk van, az ügyintéző és az adminisztrátor. Ezek alapján megtehetnénk, hogy a User osztálynak létrehozunk két alosztályt, egyik az ügyintéző, a másik az adminisztrátor adatainak tárolására. Ez a következőképpen nézne ki:



SPECIÁLIS FOGALMAK, ASSZOCIÁCIÓS VISZONYOK

Az osztályok között értelmezett viszonyokat tovább finomíthatjuk. Az UML, a már tárgyalt alapvető elemek (attribútumok, műveletek, asszociáció, öröklődés) mellett számos fogalmat, elemet és jelölést ajánl az osztálymodell pontosabb leírására.

AGGREGÁCIÓ ÉS KOMPOZÍCIÓ

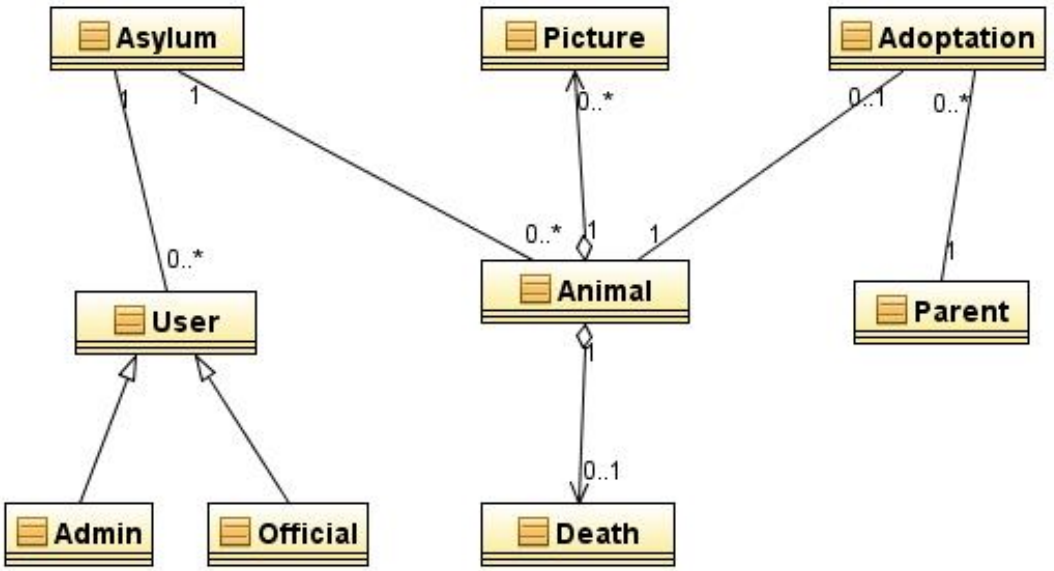
Lehetséges olyan este, amikor egy objektum egy vagy több másik objektumot tartalmaz. Az UML lehetőséget ad összetett objektumok definiálására és kezelésére, aminek eredményeként az osztályok között ún. rész-egész viszony jön létre. Ennek kétféle formája létezik:

- Aggregáció: a rész-egész viszony gyengébb formája. A tároló objektum és az azt felépítő részobjektumok elkülönülnek. Aggregációról csak akkor beszélünk, ha a részoldal nem értelmes az egész nélkül. Abban az esetben, amikor az osztály a másik oldal nélkül is értelmes, akkor egyszerű asszociációs viszony van.
- Kompozíció: a rész-egész viszony erősebb változata. A tároló objektum a részobjektumokat is fizikailag tartalmazza. Ez azt jelenti, hogy együtt keletkeznek és szűnnek meg, vagyis az egész megszűnésével a rész is megszűnik. Az egész oldal számossága csak egy lehet.

Az aggregációnak és kompozíciónak az UML diagramon más jelölése van, mint az egyszerű asszociációnak. Az aggregációt egy rombusz, a kompozíciót egy sötétített rombusz szimbolizálja, a rombusz a tartalmazó osztály felöli oldalon helyezkedik el.

Ennek tudatában kicsit át kell alakítanunk az osztálydiagramunkat. Ugyanis az Animal osztály és Picture osztály között nem egyszerű asszociáció van. Ez a kapcsolat sokkal inkább aggregációként értelmezhető. A kép objektumok az állat objektum részeit képezik.

Ugyanígy az Animal osztály és Death osztály között is inkább aggregációs kapcsolat van:

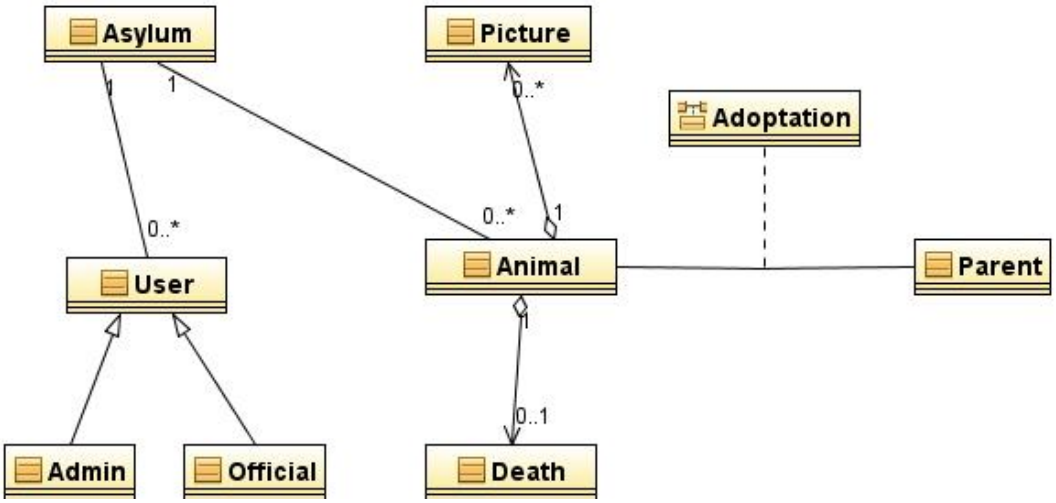


ASSZOCIÁCIÓS OSZTÁLY

Asszociációs osztályt általában akkor alkalmazunk, amikor két osztály elemei között több-több jellegű leképezést akarunk megvalósítani, de az egymáshoz rendelt párokhoz még további információkat is hozzá akarunk rendelni, melyek igazából egyik osztályhoz sem tartoznak igazán. Ezeket az információkat egy külön osztályba specifikáljuk. Az így keletkezett osztályt asszociációs osztálynak nevezzük.

Az UML-ben az asszociációs osztályt és a kapcsolatot szaggatott vonallal kötjük össze.

Az alkalmazásunkban az Adoptation osztály lényegében csak az Animal és Parent osztályok összekapcsolására szolgál és némi információt tárol az örökbefogadásról. Tehát akkor ez nem más, mint egy asszociációs osztály:

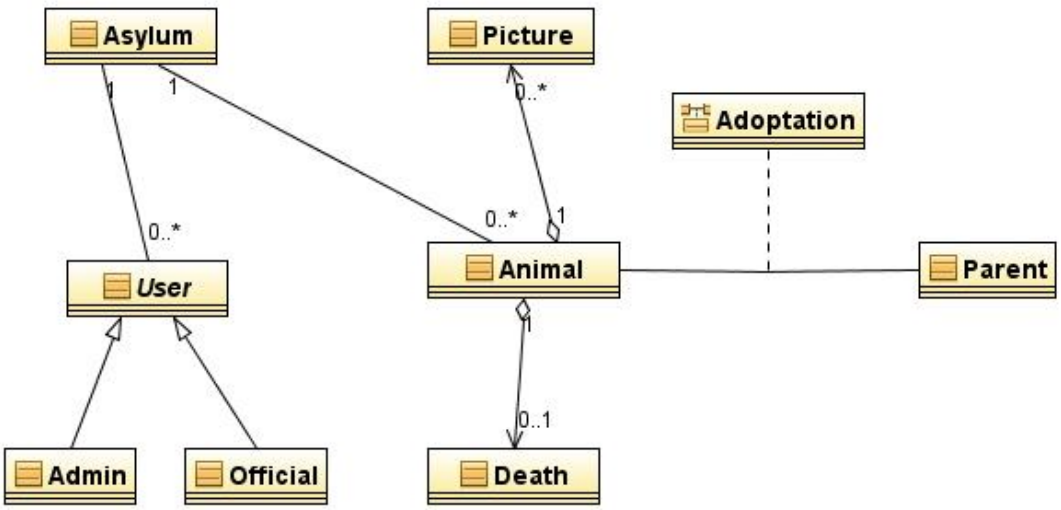


ABSZTRAKT OSZTÁLYOK

Az absztrakt osztályok speciális osztályok, amelyeknek nem létezhetnek objektumai. Arra használjuk őket, hogy további osztályokat származtassunk belőlük, amelyek öröklik az absztrakt osztály attribútumait, műveleteit és asszociációit.

Absztrakt osztályok specifikálásakor az osztály nevét döntött betűkkel kell írni.

Mivel a User osztálynak nem lehetnek példányai mivel a rendszerben egyszerű felhasználók nem léteznek csak ügyintézők vagy adminisztrátorok, ezért ez egy absztrakt osztály lesz:



Ezzel az alkalmazásunk modell rétegének osztálydiagramja elkészült. A továbbiakban még tisztázunk néhány jelölést és áttérünk a vezérlés rétegre.

OSZTÁLY-ATTRIBÚTUMOK, OSZTÁLY-MŰVELETEK

Az osztály szintű attribútumok, az osztály minden objektumában ugyanazt az értéket veszik fel. Az osztály szintű műveletek minden objektumra azonos módon lejátszódó műveletek.

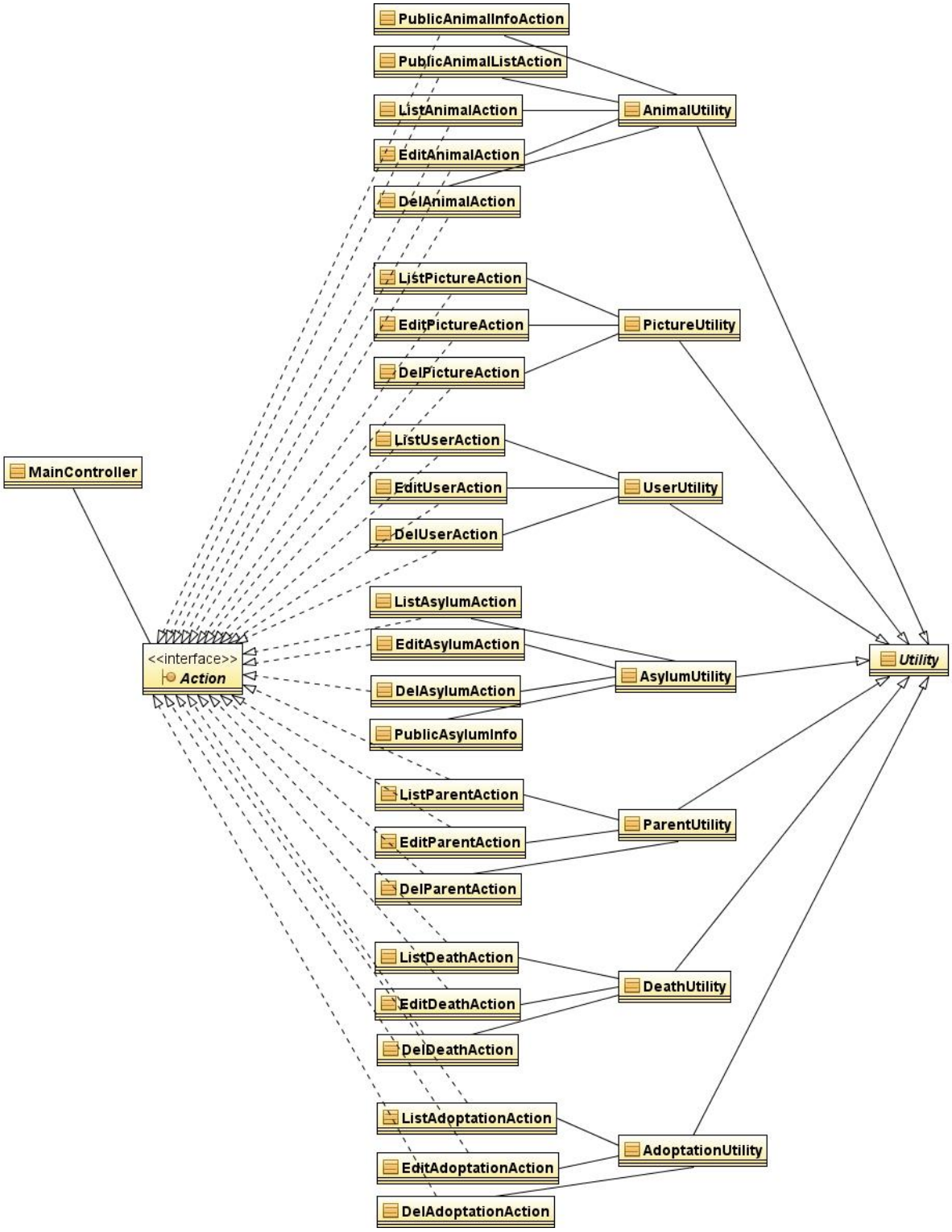
Az osztály szintű attribútumokat és műveleteket az UML aláhúzással jelöli.

INTERFÉSZ

Interfésznek nevezzük azt a speciális absztrakt osztályt, amely kizárólag látható absztrakt műveleteket tartalmaz. Az interfész így attribútumokkal nem rendelkezhet, mindössze bizonyos műveletek létezését jelentheti ki.

Az UML az interfészt külön, osztály jellegű, de mégsem osztály modellelemnek tekinti, melyet az <<interface>> sztereotípiával jelöl.

Az Állatmenhely alkalmazás vezérlési rétegének osztálydiagramja:



A működése:

Az alkalmazás talán legfontosabb osztálya a MainController. Minden kérés, ami a böngészőtől érkezik, ebbe az osztályba fut be. Ez az osztály felelős a kérések értelmezéséért

és elosztásáért, valamint annak vezérléséért, hogy mindenki csak a neki szánt funkciókat érhesse el. Létrehozásakor felépít egy hashtáblát ami a funkciók kódjait tartalmazza kulcsként és az azokat megvalósító Action objektumokat értékként. Vagyis amikor beérkezik egy kérés az MainController megvizsgálja a kérést, ebből kideríti az elérni kívánt funkció kódját. Ezután megvizsgálja, hogy a kérőnek ehhez van-e joga. Ha igen a hashtáblából kikéri a megfelelő Action objektumot és továbbítja neki a kérést végrehajtásra.

Amint az osztálydiagramon is látjuk a MainControler tehát az Action interfésszel áll kapcsolatban. Az Action interfésznek egyetlen művelete van, ezt hívja meg a MainController amikor továbbít egy kérést végrehajtásra. Láthatjuk, hogy nagyon sok osztályunk van, amelyek megvalósítják az Action interfészt. Tehát ezek az osztályok felelnek a kérés végrehajtásáért. A kérés végrehatása során Utility objektumokkal dolgoznak és kapcsolatba lépnek a nézet réteggel.

Most nézzük az Utility absztrakt osztályt. Ez az osztály és leszármazottai felelősek az adatbázis kapcsolatért, az adatbázis műveletek elvégzéséért, ezek lépnek kapcsolatba a modell réteggel. Tehát minden egyes Utility osztály rendelkezik művelettel adatbázisban tárolt objektumok elérésére, keresésére, mentésére és törlésére. Értelemszerűen mondjuk az AnimalUtility Animal objektumok eléréséért, kereséséért, mentéséért, törléséért felel.

A nézet rétegből nem tudok osztálydiagrammal szolgálni, mivel ebben a rétegben nem osztályokkal dolgozunk, hanem HTML kódot tartalmazó fájlokkal.

OBJEKTUMDIAGRAM

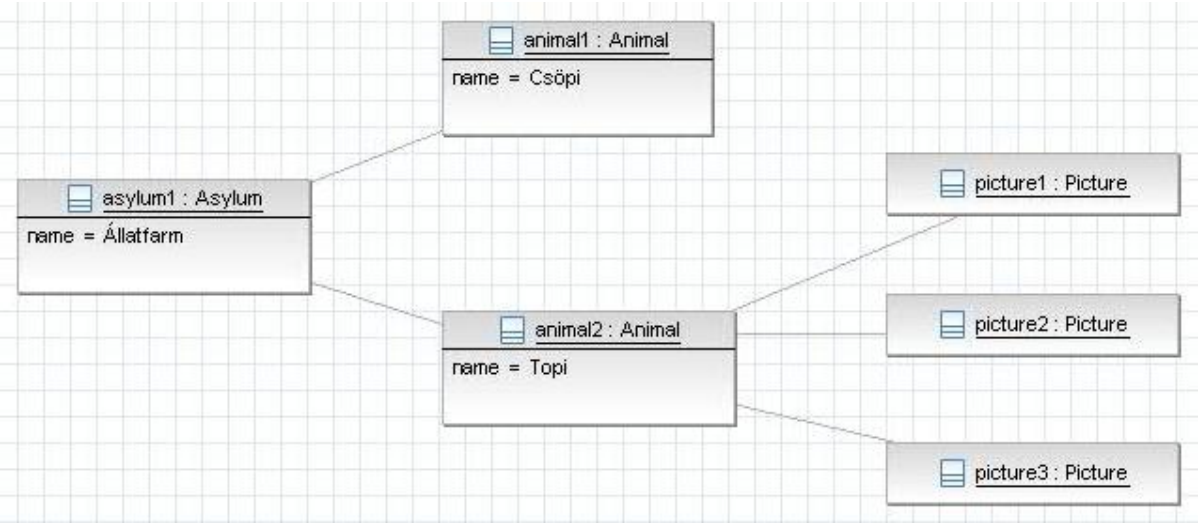
Az objektumdiagram arra szolgál, hogy egy adott időpillanat konkrét objektumait és a közöttük lévő kapcsolatokat ábrázoljuk. Segítségével nem csak egy konkrét szituációt, hanem egy jellemző helyzetet is felvázolhatunk.

Az UML jelöléseivel egy objektumnak megadhatjuk az azonosító nevét, egy kettőspontot követően az osztályt, vagy mindkettőt. Azt, hogy objektumról van szó, az azonosító név, illetve az osztály aláhúzásával jelöljük.

Egy objektumot úgy jellemzünk, hogy megadjuk annak attribútum értékeit. Nem szükséges az összes attribútumot felsorolni, elegendő csak azokat, melyek lényegesek a helyzet jellemzésének szempontjából.

Az értéket az attribútum név és egy egyenlőségjel után adhatjuk meg. Az UML az objektumok közötti kapcsolatokat az asszociációkhoz hasonlóan, vonallal jelöli.

Most pedig nézzünk meg egy példát a példaalkalmazásunkból:



A diagram bal oldalán láthatunk egy Asylum objektumot, az azonosítója asylum1. Egy attribútumát tüntettük csak fel az ábrán, ez a name attribútum, melynek értéke Állatfarm. Ehhez az Asylum objektumhoz két Animal objektum is tartozik. Az egyik az animal1, a másik az animal2. Ezen objektumok esetén is csak a name attribútum értékét adtuk meg. És végül azt láthatjuk, hogy az animal2 objektumunkhoz három Picture objektum is tartozik. Egyiknél sem adtunk meg attribútumot.

INTERAKCIÓS DIAGRAMOK

Az interakciós diagramokon az együttműködő objektumokat a közöttük lefolytatott kommunikáció alapján jellemezzük. Segítségükkel leírhatjuk, hogy adott objektumok adott körülmények között hogyan működnek együtt. Egy interakciós diagram általában egy

használati eset részleteit határozza meg. Szabványosan az interakciók két diagramtípussal ábrázolhatók. Szekvencia-diagramokkal a folyamat időbeliségét emelhetjük ki. Együttműködési diagramok alkalmazásával az együttműködő elemek szerkezeti felépítését hangsúlyozzuk.

PÉLDAOBJEKTUMOK

A diagramok alapelemei a példaobjektumok. Interakcióként legtöbbször egy jellemző helyzetet, például egy művelet végrehajtása során az együttműködő objektumok közötti üzenetváltásokat ábrázoljuk. Előfordulhat olyan eset, amikor egy üzenet, több, azonos osztályba tartozó objektumnak szól, ekkor azt multiobjektumként ábrázoljuk.

ÜZENETEK

Az objektumok az együttműködést interakciókkal valósítják meg. Ez a gyakorlatban üzenetváltást jelent, tehát üzenetek küldését és fogadását.

Egy üzenetküldést a következő alapelemekkel adhatunk meg:

- vezérlő információk:
    - feltétel, amely az üzenet küldésének feltételét adja meg. Jele az üzenet neve elé, szögletes zárójelek közé írt logikai kifejezés
    - az interakciós jelző jelzi, hogy az üzenetet fogadó objektumnak az üzenet többször is elküldésre kerül, erre a * utal.
- visszatérési érték vagy értékek
- az üzenetek elnevezése
- az üzenetek argumentumai

Az üzenet küldése tehát egy művelet hívásának a szintaktikáját követi, de azt nem feltétlenül metódushívással valósítjuk meg.

Az üzenet irányát a küldőtől a fogadó felé mutató nyíl jelöli. A nyílhegy az üzenetátadás jellegére utal:

- Két vonallal megrajzolt nyílhegy: egyszerű, nem függvényhívás-jellegű üzenet

- Teljes nyílhegy: szinkron, függvényhívás/visszatérés jellegű üzenet. Az üzenetet küldő megvárja, míg a fogadó befejezi az üzenet feldolgozását.
- Egyetlen vonallal rajzolt fél-nyílhegy: aszinkron üzen. Az üzenetküldő nem várja meg az üzenet végrehajtását, hanem folytatja működését.

Az üzenet feldolgozásából történő visszatérést szaggatott vonallal ábrázoljuk.

SZEKVENCIA DIAGRAM

A szekvencia diagramon az üzenetek időbeli viszonyait emelhetjük ki. Ábrázolhatjuk konkurens folyamatok működését, objektumok létrehozását, lebontását, az öndelegációt, illetve a különböző feltételek melletti elágazást.

A szekvencia diagramok esetén a példaobjektumok egy-egy függőlege vonal, az életvonal tetején helyezkednek el, amelyek az objektumok élettartamára utalnak. Az objektumokat a diagram vízszintes tengelye mentén soroljuk fel, a függőleges tengely pedig fentről lefelé az idő múlását ábrázolja.

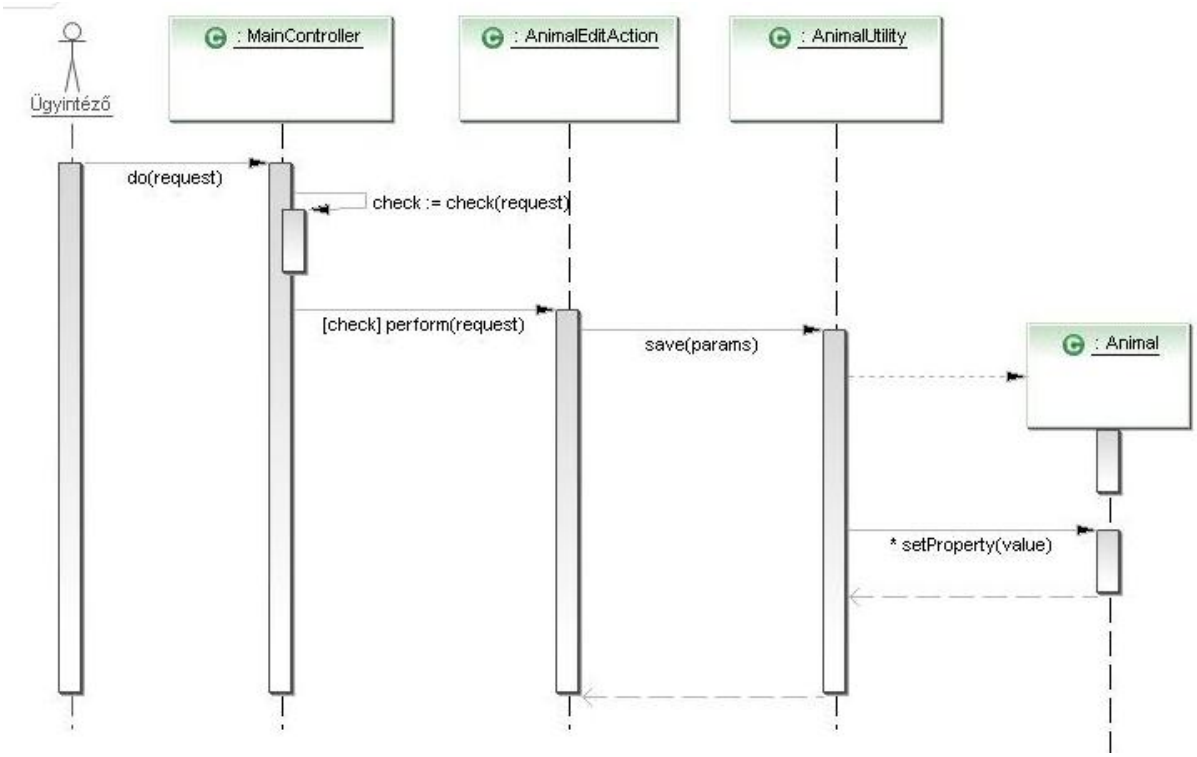
Az üzenetküldést a küldőtől a fogadó objektum életvonaláig vezető, nyílban végződő vonal jelzi, melyen az üzenetváltás elemeit címkeként adjuk meg.

A diagramon az elágazást az üzenet vonalának és az életvonal leágazásával jelöljük. A feltételt szögletes zárójelek között adjuk meg, ezt az üzenet őrszemének nevezzük.

Öndelegációnak nevezzük, ha az objektum a saját műveletét használja fel, azaz önmagának küld üzenetet.

Különleges üzenet az új objektumot létrehozó konstrukció, mely nyílhegye a példaobjektumban végződik. Az objektum lebontását, destrukcióját az életvonalat lezáró X jelzi.

Most nézzük egy példát a próbaalkalmazásunkból. Egy új állat felvitelének a következőképpen néz ki a szekvencia diagramja:



Láthatjuk, hogy a diagram négy próbaobjektum együttműködését mutatja be. A folyamat elindításáért egy ügyintéző aktor felelős.

Az ügyintéző megadja az adatokat az állat felvitele űrlapon, majd elküldi annak tartalmát. Ennek hatására a MainController objektum do() metódusa hívódik meg, amely paraméterként megkapja a kérés részleteit. Először ellenőrzi a kérés jogosságát a check() metódusával. Ez egy öndelegáció, hisz az objektum önmagának küld üzenetet. Az ellenőrzés eredménye egy logikai érték, amit check változóban tárol el. Ezután egy feltételes hívás következik. A feltétel a check változó éréke. Ha igaz, vagyis jogos a kérés, csak akkor hívja meg az AnimalEditAction perform() metódusát és adja át paraméterként a kérés részleteit.

Ez az objektum meghívja az AnimalUtility objektum save() metódusát, paraméterként átadja neki az elmentendő állat adatait.

Az AnimalUtility objektum létrehoz egy Animal objektumot és sorba meghívja annak setter metódusait a megfelelő paraméterekkel. Majd menti az adatbázisba. A vezérlést pedig visszaadja az AnimalEditAction objektumnak, ami a megfelelő oldalra irányít.

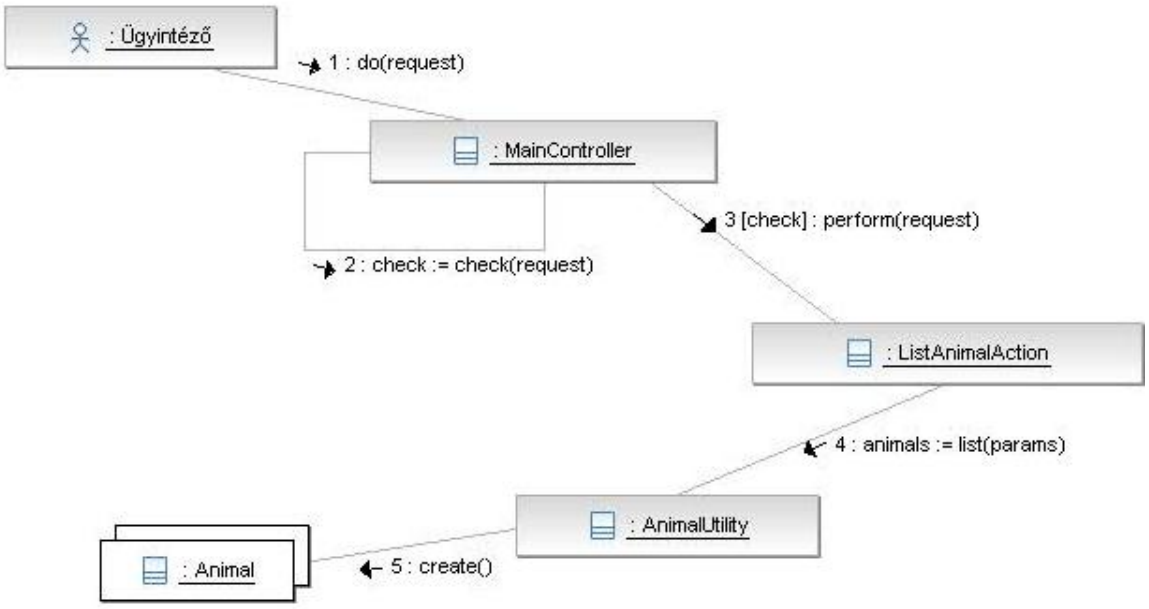
EGYÜTTMŰKÖDÉSI DIAGRAM

Az együttműködési diagramok az objektumok szerveződésére, kapcsolódási módjaira helyezik a hangsúlyt. Ez az UML másik interakciós diagramtípusa.

Az együttműködési diagramokon a példaobjektumok ikonokként jelennek meg, az üzenetek pedig a kapcsolatok közelében elhelyezett nyílként.

A sorrendiséget az üzenetek elé írt számozás határozza meg. A számozás lehet folytonos, vagy hierarchikus. Az aszinkron módon küldött üzeneteket nem számokkal, hanem betűkkel jelöljük.

Most nézzük egy példát a próbaalkalmazásunkból. Az állat keresésének a következőképpen néz ki az együttműködési diagramja:



Az ügyintéző megadja a keresés feltételeit az állat keresés űrlapon, majd elküldi annak tartalmát. Ennek hatására a MainController objektum do() metódusa hívódik meg, amely paraméterként megkapja a kérés részleteit. Először ellenőrzi a kérés jogosságát a check() metódusával. Ez egy öndelegáció, hisz az objektum önmagának küld üzenetet. Az ellenőrzés eredménye egy logikai érték, amit check változóban tárol el. Ha jogos a kérés, akkor meghívja az ListAnimalAction perform() metódusát és átadja paraméterként a kérés részleteit.

A ListAnimalAction objektum meghívja az AnimalUtility objekum list() metódusát átadva neki a keresi feltételeket.

Az AnimalUtility egy adatbázis lekérdezés után létrehozza a megfelelő Animal objektumokat és visszaadja azokat a ListAnimalActionnek. Az átadja ezeket a megfelelő lapnak és átirányít.

IDŐBEN LEZAJLÓ VÁLTOZÁS DIAGRAMJAI

Az UML két diagramtípust kínál, amivel az időben lezajló változást ábrázolni tudjuk. Az egyik az aktivitás diagram, ami a változást alapvetően aktív szempontból írja le, a másik az állapot átmenet diagram, ami a változást alapvetően passzív oldalról közelíti meg.

AKTIVITÁS DIAGRAM

Aktivitás diagramok segítségével az alkalmazás dinamikáját, időben lezajló változását aktív oldalról, a végrehajtandó tevékenységek sorrendiségének meghatározásával ábrázoljuk. Nagy előnye, hogy a tevékenységeknél nem csak egyszerű szekvenciális sorrendet, hanem egymás melletti, párhuzamos, konkurens végrehajtást is meghatározhatunk.

AKTIVITÁS

A diagram alapeleme az aktivitás, vagy más néven tevékenység, ami egyszerűen egy olyan feladat, amit meg kell csinálni. Az aktivitás az alkalmazásban általában egy osztály valamely műveletének hívásaként jelenik. Jele a diagramon az ívelt oldalú téglalap.

SORRENDISÉG

Az időbeli sorrendet a nyílhegyben végződő vonallal, az átmenettel jelöljük. A nyíl rendre az aktivitás befejezése utáni következő végrehajtandó tevékenységre mutat.

SZINKRONIZÁCIÓ

Lehetnek olyan tevékenységcsoportok, amelyek között nem lényeges a sorrend, azok egymás mellett párhuzamosan is végrehajthatók. Ilyenkor egy szinkronizációs vonal segítségével a vezérlés több szálra bontható. Ekkor nem feltétlenül párhuzamos végrehatást írunk elő, csupán jelezzük, hogy a tevékenység csoportok logikailag függetlenek egymástól. Ha nem

szükséges tényleges párhuzamosság, akkor a szálak tevékenységcsoportjai tetszőleges sorrendben végrehajthatók.

A szinkronizációs vonal jelölése egy vastag, vízszintes vonal.

A szinkronizációt ugyancsak szinkronizációs vonallal jelöljük. Ez az a pontot, ahol a konkurensen végrehajtható szálak befejeződését megvárjuk. A szinkronizációval jelezzük, hogy a következő tevékenység végrehajtásához szükséges az előző aktivitások sikeres végrehajtása.

Az alternatív tevékenységcsoportok közötti választás rombusszal jelöljük. A döntésből kivezető éleket szögletes zárójelek között megadott feltételekkel címkézzük.

Tehát a szinkronizációs vonal „és”, a döntés „vagy” jellegű elágazást jelöl.

ÁLLAPOT TÍPUSOK

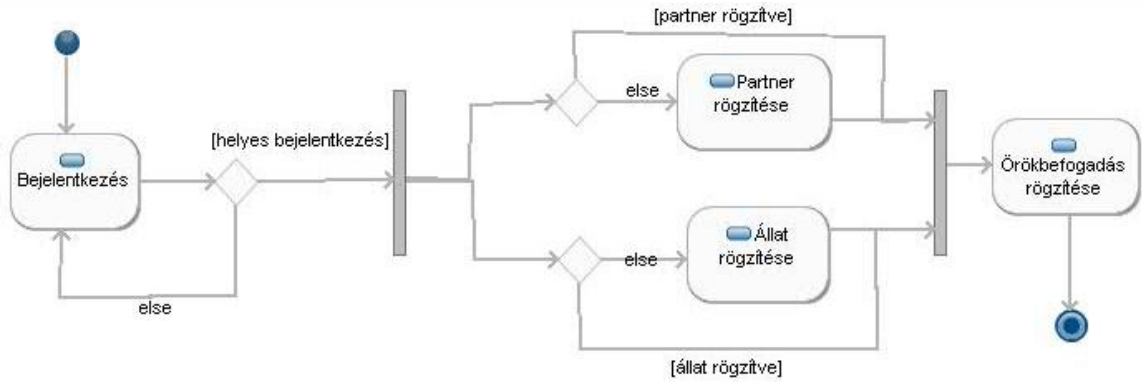
Az aktivitás diagramon a végrehajtandó műveleteket állapotokként vehetjük fel. Az UML az állapotokat két fő típusra bontja:

- Akció-állapot: akciókat helyezhetünk el benne. Akcióknak az atomi műveleteket nevezzük. Úgy tekintjük, hogy egy atomi művelet végrehajtása nem igényel időt, így az egyetlen időpillanatban zajlik le és ezért félbe sem szakítható.
- Aktivitás-állapot: aktivitások szerepelhetnek benne. Az aktivitások vagy más néven tevékenységek olyan műveletek, amelyek végrehajtása időt vesz igénybe ezért félbe is szakíthatóak. Az aktivitás végső soron atomi akciókból felépített összetett tevékenységet jelöl.

A diagramon két különleges, úgynevezett pszeudó-állapot is szerepelhet:

- Start-állapot: az összetett tevékenység kezdetét határozza meg. Jele fekete kör.
- Stop-állapot: az összetett tevékenység befejeződését jelöli. Jele körön belüli fekete kör.

A példaalkalmazásunk örökbefogadás rögzítése funkciójának aktivitás diagramja:



A kezdőállapotból a Bejelentkezés aktivitásba jutunk. Ennek során a felhasználó megadja felhasználói nevét és jelszavát. Ezek után amint láthatjuk egy döntés következik. A kérdés az, hogy helyesek-e a felhasználó által megadott adatok. Ha igen akkor tovább engedjük a felhasználót, ha nem akkor újra a Bejelentkezéshez jut.

Ha sikeres volt a bejelentkezés, akkor tudjuk csak rögzíteni az örökbefogadást. Ennek azonban két előfeltétele is van. A partnernek és az állatnak már rögzítve kell lennie a rendszerben. Ha ezek közül valamelyik még nem teljesült, akkor pótolni kell.

Ezért ezen a ponton a vezérlés két szálra bomlik. Ezt egy szinkronizációs vonallal jelöltük. Az egyik szál a partner létezését vizsgálja. Vagyis látjuk, hogy egy újabb döntés következik. Ha a partner még nincs rögzítve, akkor Partner rögzítése aktivitáshoz jutunk. Itt a felhasználó megadja a partner adatait és elmenti.

A másik szál ehhez nagyon hasonló tevékenységet végez. Megvizsgálja, hogy létezik-e már az állat, ha nem akkor rögzítjük: Állat rögzítése aktivitás.

A két szál ezek után egyesül, ezt ismét egy szinkronizációs vonallal jelöljük. A két száll tetszőleges sorrendbe hajtódhatott végre, vagyis nem számít, hogy előbb az állatot visszük-e fel vagy a partnert. De ezen a ponton tudjuk, hogy a szükséges állat és partner már rögzített. Ezért semmi akadálya annak, hogy elvégezzük a legutolsó lépést, rögzítsük az örökbefogadás.

A szinkronizáció után tehát az Örökbefogadás rögzítése aktivitás következik. Innen már csak a végállapotba vezet nyíl, tehát készen vagyunk.

IMPLEMENTÁCIÓS DIAGRAMOK

Az UML két fajta implementációs diagramot definiál, a komponens- és az alkalmazási diagramot. Segítségükkel rögzíthetjük a modellünkben ábrázolt logikai szerkezet megvalósításával kapcsolatos döntéseket, például a fizikai elrendezést.

KOMPONENS DIAGRAMOK

Komponensdiagramok segítségével az alkalmazás fizikai szoftver-alkotóelemeit és az azok közötti viszonyokat ábrázoljuk. Alkalmasak a kész alkalmazás fizikai szerkezetének vázolására. Segítségükkel a fejlesztés során a használt fájlok közötti viszonyok is szemléltethetjük.

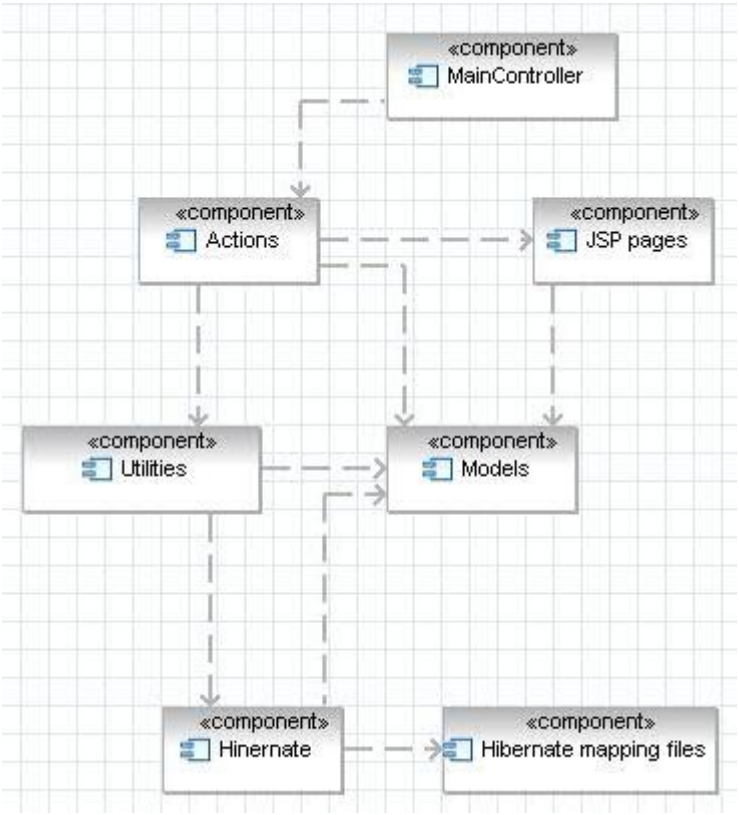
Fejlesztés alatt álló vagy a már kész alkalmazás fizikai alkotóelemeit nevezzük komponenseknek. Például:

- forrásállományok
- kódállományok
- programkönyvtárak
- futtatható állományok
- dokumentumok
- adatfájlok

A komponens egyszerűen valami olyasmi, ami például egy másolás paranccsal másolható, tehát többnyire egy állományként jelenik meg.

A komponens jele egy téglalap, mely bal oldalán két téglalap alakú címkét veszünk fel

Az alkalmazásunk egy lehetséges komponens diagramja:



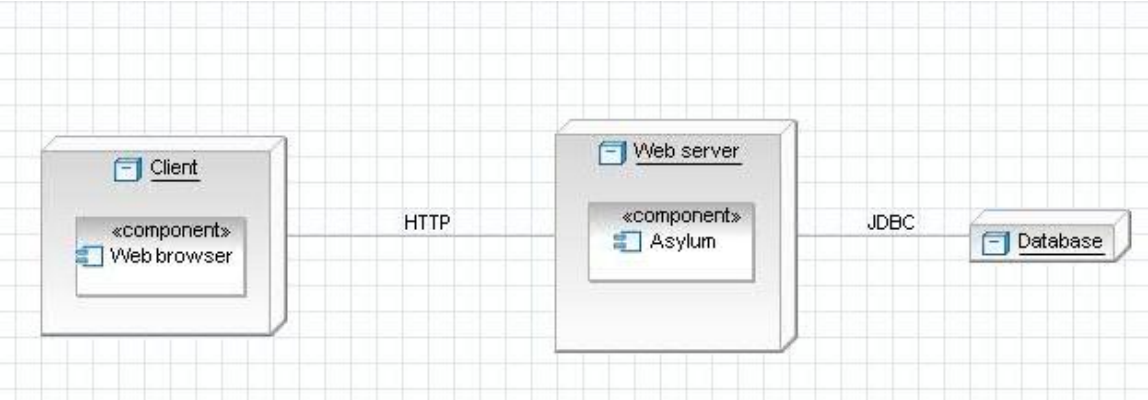
Az implementációt ebben az esetben Javaban képzeltük el. A MainController komponensünk egy szervlet. Ez közvetlenül az Action osztályokkal áll kapcsolatban. Az Action osztályokból érjük el a JSP oldalakat és Utility osztályokat. Az Utility osztály az adatbázis műveletek megvalósításáért felelősek. Ebben az esetben az objektumok perzisztenciáját Hibernate segítségével oldottuk meg. A Hibernate a modell és mapping fájlokkal dolgozik. És természetesen az Utility osztályoknak és JSP lapoknak is szükségük van a modell osztályokra.

ALKALMAZÁSI DIAGRAMOK

Alkalmazási diagramok segítségével az alkalmazással kapcsolatban álló hardver elemeket, számítógépeket és egyéb egységeket és az azok között lévő kapcsolatokat ábrázolhatjuk.

Az alkalmazási diagram alapeleme a csomópont, amelyet az UML a kocka alakzattal ábrázol. A csomópont egy számítógépes egységet, a legtöbb esetben egy hardver elemet jelent.

Az alkalmazásunk alkalmazási diagramja:



A kliens egy böngésző segítségével kapcsolódik az alkalmazásunkhoz, ami egy web szerverre van telepítve. A web szerver pedig egy adatbázissal áll kapcsolatban.

ÖSSZEGZÉS

A diplomamunkám célja az volt, hogy az UML eszközeinek minél szélesebb skáláját mutassam be egy saját rendszer tervezésén keresztül. Azt hiszem ezt a célkitűzést sikerült teljesíteni. Az UML nyolc különböző diagramtípusát mutattam be. Részletes leírást adtam a használati eset diagramról és osztálydiagramokról, foglalkoztam az objektum, a szekvencia, az együttműködési, az aktivitás, a komponens és az alkalmazási diagramokkal.

Természetesen teljes mértékben elégedett mégsem lehetek. Négy diagramtípusról szót sem ejtettem és a bemutatott diagramtípusoknál is könnyen találunk olyan elemet, amelynek bemutatása elmaradt. De hát ez nem meglepő. Az UML olyan hatalmas eszköztárat kínál, amelynek teljes bemutatása jóval meghaladja egy diplomamunka kereteit.

Viszont úgy érzem, az elkészült diagramok igen hasznosak, felhasználásukkal nem lenne bonyolult a tervezett alkalmazás implementálása. Tehát az UML webes alkalmazások tervezéséhez is kiváló partner.

IRODALOMJEGYZÉK

- Juhász István - A rendszerfejlesztés technológiája (órai jegyzet)
- Maksimchuk, Robert A. –Naiburg, Eric J. (2006): UML földi halandóknak. Kiskapu Kiadó, Budapest
- Störrle, Harald (2007): UML 2. Panem Könyvkiadó, Budapest
- Dr. Sziray József – Kovács Katalin (2005): Az UML nyelv használata. Universitas-Győr Kht, Győr
- Vég Csaba (1999): Alkalmazásfejlesztés a Unified Modelling Language szabványos jelöléseivel. Logos Kiadó, Debrecen
- http://hu.wikipedia.org/wiki/Unified_Modeling_Language
- http://vinci.org/uml/index.html
- http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/index.htm

KÖSZÖNETNYILVÁNÍTÁS

Szeretném megköszönni témavezetőmnek, Pánovics Jánosnak a diplomamunka elkészítése során mutatott türelmet, bizalmat, segítőkészséget.

Köszönöm családomnak, akik megteremtették a lehetőséget, hogy idáig eljuthattam.

Végül köszönöm páromnak, Dusa Ágnesnek a lelki támogatást és édesanyjának a folyamatos unszolást.