UML alapok

Egységes modellező nyelv

Az UML kialakulása és helye a szoftverfejlesztésben

Ebben a fejezetben áttekintjük, milyen igények vezettek egy, a tervezést támogató grafikus leírónyelv megalkotásához a szoftveripar számára.

A műszaki tervezés (legyen ez építőipar, gépipar vagy elektronikai ipar) nagyon korán létrehozta a közös grafikus nyelvét a különböző műszaki rajzok formájában. Ehhez hasonló módon a szoftverek kialakulása magával hozta a tervezés igényét, de nem tudta átvenni a már korábban bevált műszaki terv technikákat sajátos igényei miatt. Míg a műszaki tervezés alapvetően statikus, a szerkezetet mutatja meg, a szoftverek esetében mind a felépítés ábrázolása, mind a viselkedés, a szerkezeti elemek kölcsönhatása megfelelő grafikus nyelvet igényel.

Több próbálkozás volt ennek az igénynek a kielégítésére, de végül 2000-ben létrejött az egységes modellező nyelv, az UML[1]. Azóta is folyamatosan fejlesztik, a specifikációt több más tervezési leíró nyelvvel együtt az OMG[2] gondozza.

A nyelv célja a szoftverfejlesztés általános leírásának megteremtése, a tervek egységesítése, és a korábban megszokott szöveges leírások helyett grafikus elemek segítségével egy sokkal jobban áttekinthető tervezés bevezetése.

Fontos hangsúlyozni a nyelv jelleget. Míg a műszaki tervezésben a jelölések egységesek, azaz a vonalak, alakzatok, egyéb elemek jelentése rögzített, az UML lehetőséget biztosít (természetesen a fix értelmezéseken belül) az igény szerinti részletezésnek megfelelő átfogóbb vagy éppen a részletekbe menő kifejezésre. Ezért az UML nincs távol és semmiképpen nincs ellentétben a szöveges leírásokat alkalmazó tervezéstől. A grafikus elemek biztosította lehetőségek ezeket részben ugyan feleslegessé teszik, bár nem zárják ki teljesen.

Eredményesen akkor tudjuk alkalmazni, ha mindig a megfelelő részletességet biztosító nézetet használjuk és nem félünk a szöveges kiegészítésektől.

Az UML ma már nemzetközi standard is, több, országokon átívelő projektben, de akár az országos közigazgatási szoftverrendszerek esetében is előírják alkalmazását. Mint nyelv, erre is jellemző, hogy még az aktuális tudásszintünk által korlátozottan is jól használható, segítségével egyértelműen tudjuk kifejezni magunkat a szoftveripar bármely területén és a szoftverfejlesztés bármely fázisában.

[1] Unified Modeling Language
[2] Object Management Group

Nagy előnye az UML-nek, hogy nem csak a szoftver felépítése, a komponensek, entitások[3] kapcsolatai, számossága modellezhető, de az események által indított folyamatok, tevékenységek, vagy akár a kívánt funkcionalitások is leírhatók, mindig a megfelelő eszközkészletet alkalmazva.

Az UML folyamatos fejlesztése során nagy hatással volt számos párhuzamosan fejlesztett, specializált leíró technikára. Bár ezek nem részei a szigorúan vett UML specifikációnak, így nem tekinthetjük ezeket az UML kiterjesztéseinek, a köztudatban viszont sokszor ez összemosódik. Nem utolsósorban azért, mert hajlamosak vagyunk a hasonló grafikai elemekkel operáló összes technikát UML-nek tekinteni. Ezt a félreértést erősítik azok a szerkesztőeszközök, amelyek több párhuzamos leíró technikát kínálnak fel, egységes grafikus megjelenítés mellett.

Hangsúlyozni kell, hogy a látványos grafikai megoldások ellenére az UML nem illusztrációs eszköz, tehát nem lehet az a cél, hogy a hagyományos szöveges leírásokban az ábrákat szolgáltassák az UML eszközeivel kialakított diagramok. Maga az UML kell, hogy adja a szoftver tervezés leírását, és a szöveges kiegészítések csupán a magyarázatok, részletezések szerepét játsszák mellette.

Az UML-t nagyon sok, különböző szakterület képviselői használhatják munkájuk során. Egy részük a modell elkészítését végzi, más részük felhasználja a kész modellt, mert annak alapján építi fel a szoftvert. Az első csoportba tartoznak a rendszer tervezői, a rendszerelemzők, de az üzleti elemzők is jól tudják hasznosítani. A funkciók tesztelésének megtervezéséhez például a tesztelők számára is értékes az úgynevezett használati eset diagram. Elsődleges felhasználóként az implementálók (programozók, kódolók) jöhetnek számításba, de a projektben érdekelt szereplők, a menedzserek, ügyvezetők és nem utolsó sorban a dokumentátorok számára is nélkülözhetetlen.

Ellenőrző kérdések:

Mi az UML nyelv kifejlesztésének fő célkitűzése?

Melyek az UML alkalmazásának előnyei?

Ki vagy kik felelősek az UML nyelv fenntartásáért, folyamatos fejlesztéséért?

Hogyan használhatják a tesztelők az UML nyújtotta ábrázolást a munkájuk során?

Kik használhatják az UML-t a szoftvertervezés, -fejlesztés során?

[3] entitás vagy egyed: azon objektum, amelynek a tulajdonságait összegyűjtjük és tároljuk

UML felépítése - elemek, relációk, diagramok

Hogyan épül fel egy grafikus leírónyelv, milyen nyelvi elemekből áll össze? Ezzel a kérdéskörrel foglalkozunk ebben a fejezetben.

Az UML nyelv kialakítását ugyanazok kezdték meg, akik az objektumorientált elveket is bevezették a szoftverfejlesztésbe (gyakran használt rövidítése az OO betűszó), így nem meglepő módon az UML kifejezések sokszor emlékeztetnek az objektumorientált elvek szerinti elnevezésekre. Le kell azonban szögezni, hogy az UML nem csak az objektumorientált elvek mentén tervezett szoftverek leírására alkalmas, hanem bármely szoftver esetében alkalmazható.

Felépítését tekintve az UML egyszerű. Ha fenntartjuk a nyelv analógiát, elemekből, azaz szavakból áll (ezek itt grafikus jelölések, ám értelmezésükben sokat számít a kontextus, a szövegkörnyezet), ezek viszonyait a relációk, a kapcsolatok mutatják, mint a nyelvtan a természetes nyelvek esetében, és a mondatoknak, az egyes részleteket leíró kötött szerkezeteknek a diagramok felelnek meg.

Nem törekedtek arra, hogy minden elem egyedi jelölést kapjon, nem akartak piktogramokat létrehozni. Ezért a grafikus jelölések alapból egyszerűek, csak kivételes esetben különböztetik meg ezeket a jelöléseket az elemek felső sarkaiban elhelyezett ikonokkal. Mégis egyértelmű a jelölés, mert adott kontextusban csak az egyik, vagy másik alkalmazható. Ha további részletezésre van szükség, akkor az összes elem sztereotipizálható, azaz a szakterületnek[4] megfelelő finomabb jelentésük is megadható.

A statikus UML elemek közé tartoznak az objektumorientált értelemben vett osztályok, amelyek tervezési szinten a szoftver adatszerkezetét vázolják fel. Használhatjuk ugyanakkor az osztályok konkrét előfordulásait, az objektumokat is, ha célunk a futó szoftver pillanatnyi állapotának bemutatása, és a szintén objektumorientált értelemben vett interfészeket is. Ezek realizálhatóak, azaz megvalósíthatóak, és ezen keresztül kötelező viselkedést, funkcionalitást adnak különböző osztályoknak. A lényeg a tervezés során az, hogy a tipikus objektumorientált kapcsolatokat, az öröklődést és megvalósítást, valamint az adatcsoportok közötti számossági kapcsolatokat[5] ezek segítségével rögzíteni tudjuk.

Nem csak a statikus felépítés modellezhető azonban, a viselkedési elemek segítségével a folyamatábrákon ábrázolható tevékenységek (aktivitások és akciók), vagy az állapotok és állapotgépek is modellezhetőek.

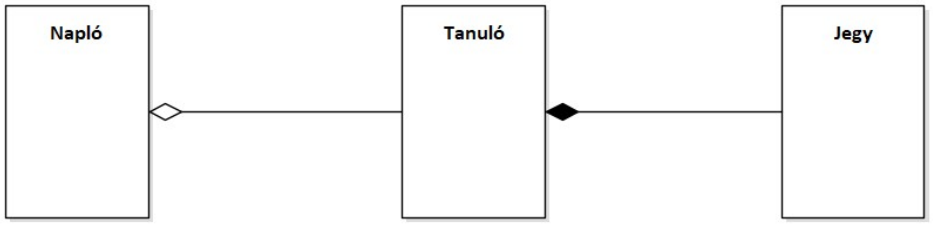
Továbbá egy telepítés is leírható UML elemek segítségével, ábrázolhatóak a hardverek és a rajtuk futó szoftverek, valamint kapcsolataik. Sokkal könnyebb egy telepítést megvalósítani

[4] Az UML használata során magyarul szakterületnek nevezzük azt a feladatot, amelyre a szoftver készül és speciális elnevezésekre van szükség. Így beszélhetünk banki szakterületről, vagy a közigazgatás egyes területeiről. Külön szakterület lehet például a gépkocsi nyilvántartás, vagy egy iskolai támogató szoftver, vagy a népességnyilvántartás.

[5] A számossági kapcsolat, vagy angolul multiplicity azt írja le, hogy egy adatcsoport egy konkrét előfordulásához, adott másik adatcsoportból hány konkrét előfordulás tartozhat. Gondoljunk a könyvtárban tárolt könyvekre, egyetlen konkrét könyvtár számos könyvet tartalmazhat.

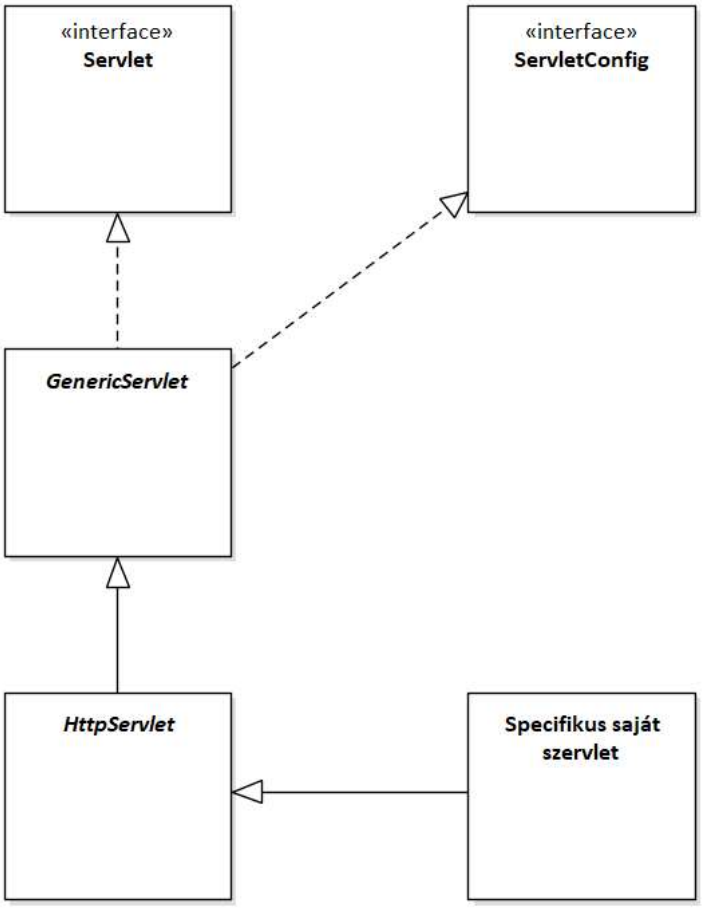
akkor, ha nem csak szöveges leírást biztosítanak hozzá, hanem jól áttekinthető felépítési ábrákat is.

Az elemek önmagukban azonban nem elegendőek egy szoftver megtervezésére, szükség van a már többször előkerült kapcsolatokra is. Ezek a kapcsolatok, az UML relációi, képezik a nyelvtant és segítségükkel az összes elemviszony ábrázolható. Itt sem törekedtek egyéni megjelenítésre a különböző relációk esetében (bár a főbb relációk ránézésre is beazonosíthatóak), a pontos jelentést mindig a kontextus adja meg[6].



Az ábra három, tartalmazás relációval összekapcsolt osztályt tüntet fel, az osztályok egy elektronikus osztálynapló alkalmazás elemei. A Napló és a Tanuló között úgynevezett gyenge tartalmazást jelző kapcsolat van, vagy szakszóval aggregáció, míg a Tanuló és a Jegy osztály között erős tartalmazást jelöltünk, szakszóval kompozíciót.

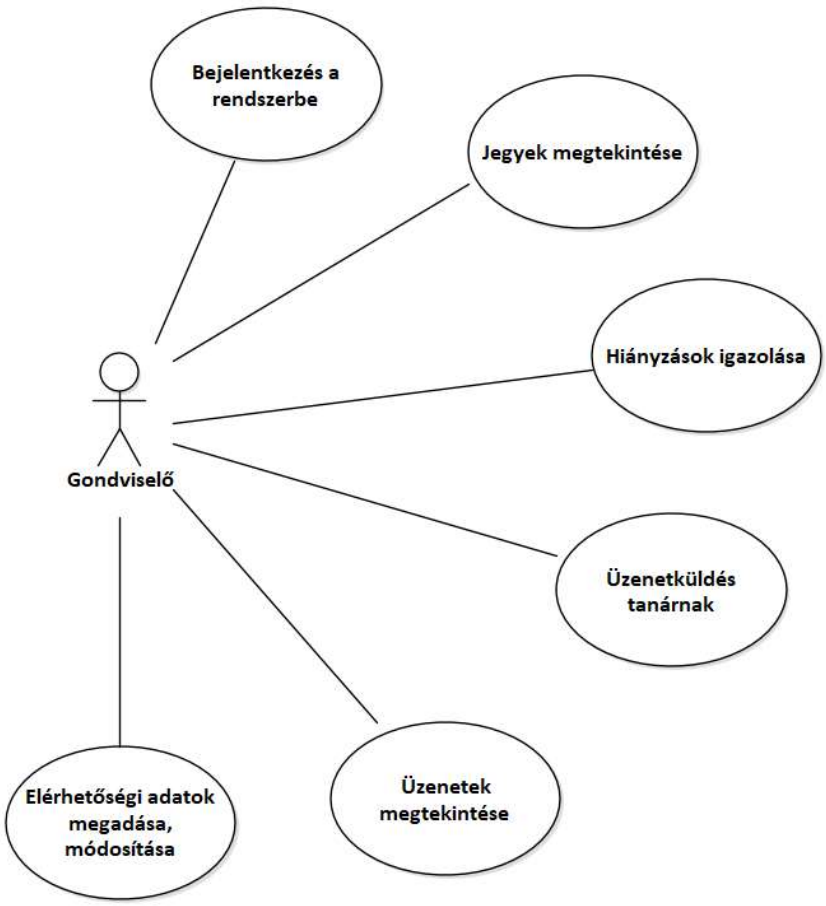
[6] Állapotátmenetek esetén a használt grafikus elem, egy folytonos vonalú nyíl, ugyanaz, mint a folyamatábrák esetén a vezérlés jelölése. Azonban a kettő nem keveredik, folyamatábrán az állapotátmenet nem értelmezett, ahogy állapotgép esetén sem értelmezhető a vezérlés irányát mutató nyíl.



Az ábrán látható osztályok és a közöttük feltüntetett relációk a Java programnyelvben létrehozható szervleteknek, a webes rendszerekben HTTP kéréseket fogadó objektumok osztályának a hierarchikus felépítését mutatják be. Öröklődés látható a HttpServlet és a GenericServlet osztályok között, valamint implementációt jelzünk a GenericServlet és két interfésze között. Az adott feladatra általunk létrehozott szervlet osztálya minden ős és implementált tulajdonsággal rendelkezik.

Az UML talán legfontosabb "találmánya" a diagram. Ezek, mint a nyelv mondatai, egy-egy összefüggő állítást tartalmaznak, ahol a diagramban szereplő elemek és relációk együtt fejeznek ki valamilyen szoftver koncepciót. Leírhatják például a szoftver egészének vagy egy részének adatszerkezetét, benne az adatelemek összefüggéseivel. A folyamatábra, az UML aktivitás diagramja a szoftver futását, a döntési pontokat és az egyes tevékenységek sorrendiségét foglalja össze. Vannak olyan helyzetek is, amikor a szoftver adatelemei, azaz objektumai adott funkciók hatására állapotukat megváltoztatják és ez is ábrázolható az UML állapotgép diagramja segítségével.

Létrehoztak olyan diagramot is (mint például a használati eset diagram), amely a funkcionalitás leírására alkalmas, és kellően részletezve a programozó ennek alapján közvetlenül meg tudja írni az adott funkcionalitás kódját.



A használati esetek az egyes felhasználókhoz kapcsolható rendszerfunkciókat írják le grafikusan. A funkciók további kifejtése, részletezése a használati eset forgatókönyveiben történik.

A UML specifikáció 2.0 verziója 13 különböző nézetet, diagramot határozott meg. Ezek mindegyike természetesen nem szükséges adott szoftver megtervezéséhez, és vannak olyan diagramok, amelyek ma már túlhaladottak, ritkán kerülnek elő. Feladatunk nem más, mint pontos jelentésüket ismerve eligazodni közöttük, hogy mindig a célnak legjobban megfelelő diagramot használjuk.

A csaknem minden alkalmazás esetén használt diagramokat tekintjük a fontosabb diagramoknak, de mivel minden diagram a tervezés alatt álló szoftver adott nézetét jeleníti meg, sorrendiséget nem érdemes felállítani. Az egyes nézetek támpontokat adhatnak egymásnak, így a diagramok párhuzamos kialakítása, fejlesztése előnyös lehet a tervezés során.

Az UML platform és programnyelv független leírónyelv, azaz a szoftver tervezése során nem kell (egyes iskolák szerint nem is szabad) figyelembe venni a programnyelvi jellegzetességeket. Így tökéletesen alkalmas szervíz orientált alkalmazások (SOA) megtervezésére is, ahol az egyes szolgáltatások és azok együttműködése megtervezhető az implementáció (megvalósítás) részletezése nélkül.

Ellenőrző kérdések:

Hogyan épül fel az UML nyelv?

Mi a különbség az öröklődés és az implementáció grafikus jelölése között?

Mik lehetnek konkrét elemek az UML ábrázolásokban? Mondj példákat!

Mi a szerepük a relációknak az elemek között? Mondj példákat különböző relációkra!

Melyik diagram segítségével tudjuk leírni a funkcionalitást?

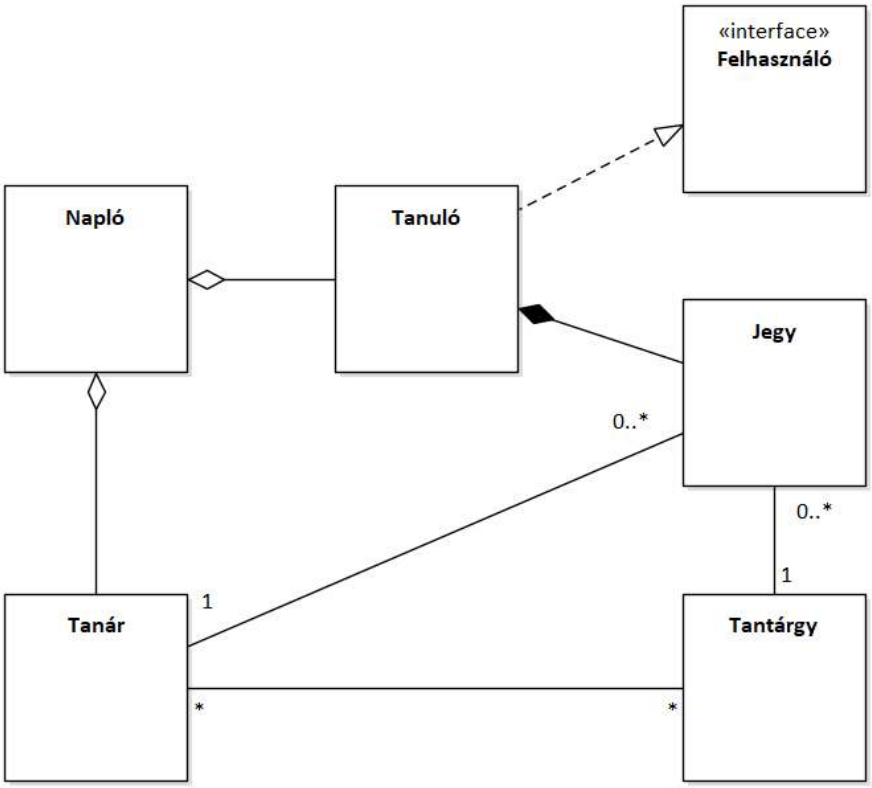
Szakterületi modell és az osztálydiagram

A szoftverek adatszerkezetének precíz leírása mindig elsődleges kérdés. Az UML keretein belül ennek eszköze az osztálydiagram, ezt vizsgáljuk meg ebben a fejezetben.

Ez a diagramok egyik legfontosabbja, az alkalmazás adatszerkezetét írja le. Ez objektumorientált tervezés esetén osztályokat jelent, ahol az egyes osztályok kapcsolódó adattagjai egy-egy entitást írnak le adatszinten. Egy példán keresztül könnyű belátni, hogy egy iskolai elektronikus osztálynapló esetén a kapott osztályzatnak vannak adatelemei, mint például az osztályzat értéke, dátuma, a tantárgy, amiből kapta a diák, a tanár, aki adta, vagy éppen az osztályzat típusa (felelet, dolgozat, órai munka, stb.)

Ez a diagram viszonylag közel áll a programozáshoz, az is valószínűsíthető, hogy nem az elemzők dolgozzák ki majd teljes részletességében. Ez nincs ellentétben az UML alkalmazási javaslatával, a dinamizmusa megengedi, hogy a szoftver egy-egy nézetét folyamatosan fejlesszük a tervezés előrehaladásával párhuzamosan.

Ez a diagram tehát az alkalmazás osztályait írja le, elsősorban az entitásokat, az osztályok által megvalósított interfészeket (specifikált viselkedéseket). Mint minden diagram (nézet) esetében, itt is vannak specifikus relációk, ezek a számosság (multiplicitás), az öröklődés, az interfész realizáció (megvalósítás) kapcsolatait jelölik. Ezek segítségével az adatszerkezetre vonatkozó összes kapcsolat megadható, olyan mélységben, hogy az osztálydiagram alapján az alkalmazáshoz tartozó adatbázis séma, vagy sémák megvalósíthatóak külön adatbázis tervezés nélkül.



Az ábrán látható osztályok, az interfész egy elektronikus osztálynapló alkalmazás elemei. A Napló és a Tanuló között aggregációt, míg a Tanuló és a Jegy osztály között kompozíciót jelöltünk.

A Jegy osztály kapcsolatban van a Tantárgy és a Tanár osztályokkal is, közöttük asszociációt jelöltünk, amelyet számossággal, szakszóval multiplicitással egészítünk ki.

A Tanár és a Napló közötti aggregáció az osztályban tanító tanárok naplóban is feltüntetett listájára utal.

A Tanár és a Tantárgy között az asszociáció sok-sok kapcsolatot jelöl, amely tervezés során tökéletesen elfogadható. Persze tisztában vagyunk vele, hogy a később kialakítandó adatbázisban a Tanár és Tantárgy táblák közé kapcsolótáblát[7] kell létrehozni.

A Tanuló osztály viselkedését részben az osztályban létrehozott operációk, részben a Felhasználó interfészben megfogalmazott viselkedések, az implementált operációk írják le.

Ez a diagram a tervezés során minden adatigényes alkalmazásnál előkerül. Tipikus esetben szoftvereink több, mint 95 százaléka adatkezelő alkalmazás, így használata általánosnak tekinthető a tervezés során. Nem véletlenül kapta magyar nyelven a szakterületi modell nevet, mert a tervezés alatt álló alkalmazás által használt specifikus osztályokat, adatcsoportokat írja le, így szoftverenként egyedi a kialakítása. A programnyelv által biztosított standard osztályokat nem szoktuk feltüntetni, hiszen azok leírása, kapcsolatai az API[8] dokumentációban már rendelkezésre állnak.

Ellenőrző kérdések:

Hogyan kapcsolódik egymáshoz az alkalmazás adatszerkezete és az osztálydiagram?

Mit értünk számosság alatt az osztálydiagram relációi esetében?

Mit értünk entitás alatt?

Asszociáció, kompozíció, aggregáció. Mi ezek jelentése, hogyan írható le a viszony?

Miért van fontos szerepe az osztálydiagramnak az adatintenzív alkalmazások esetében?

[7] A relációs adatbázisok nem tudják kezelni az egyes táblák között létrejövő sok-sok kapcsolatot (tekintsünk például egy könyv-író kapcsolatot, ahol egy könyvnek több írója is lehet, és egy író több könyvet is írhat). Ezért az adatbázis kialakítása során az úgynevezett kapcsolótáblát alkalmazzuk, amely példánk esetében megmutatja, hogy melyik író melyik könyvnek a szerzője.

[8] Application Programming Interface, az egyes speciális programnyelvi területek átfogó neve. Így például beszélhetünk adatbázis API-ról, amely az adatkapcsolatokhoz szükséges osztályokat és egyéb elemeket tartalmazza, vagy a grafikus felhasználói felület megjelenítését biztosító API-ról.

Komponens diagram

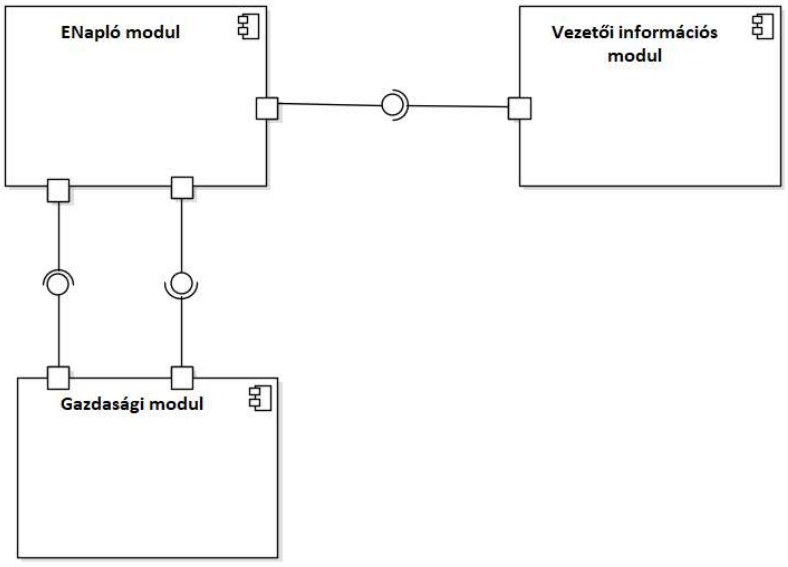
Mi van akkor, ha nagyobb léptékekben gondolkodunk és először a tervezendő szoftver átfogó képét akarjuk megalkotni? Ez a fejezet az UML komponens diagramjának bemutatásával választ ad a kérdésünkre.

A korszerű szoftverek ritkán állnak önmagukban. A tervbe vett fejlesztés sokszor egy már meglévő moduláris rendszer továbbfejlesztése új komponensek, modulok hozzáadásával. A monolitikus rendszerek kora már lejárt, a fejlesztések több okból is a moduláris, önálló komponensekből álló szoftvereket részesítik előnyben.

Több szempontból is előnyös ez a fejlesztési mód. Az egyes komponensek egymástól függetlenül, párhuzamosan fejleszthetők, külön tesztelhetők, és a már korában kialakított, bevált komponensek újrahasznosíthatóak. A komponensek független adatbázis sémákkal dolgozhatnak, a felhasználók független vizuális felületen keresztül kommunikálhatnak a szoftverrel.

Kialakításához viszont fontos, hogy a tervezés korai fázisában meg tudjuk határozni a komponenseket, azok funkcióit és a komponensek egymás közötti kommunikációjához szükséges interfészeket. Ezek sokfélék lehetnek, a különböző webszolgáltatásoktól kezdve az üzenetsorokon át, a speciális technikákig bármi alkalmazható.

Ezek precíz leírására alkalmas az UML komponens diagramja. A kommunikációt jelző assembly relációban a gömbfej az implementálandó interfészt, a félkör a kommunikációhoz szükséges implementált interfész ismeretét jelenti.



A specifikáció szerint az elektronikus osztálynapló alkalmazásunk egy nagyobb rendszer modulja csupán, a már meglévő Gazdasági modullal és a Vezetői információs modullal képez teljes rendszert. A komponenseket majd később kifejtjük, de már ezen az ábrán is fel tudjuk tüntetni az interfészek segítségével a kommunikációs irányokat. Itt a Gazdasági modul és az

ENapló modul kétirányú kommunikációt folytat, míg a Vezetői információs modul csupán az ENapló modulhoz intézhet kéréseket. A kommunikációs pontok leggyakrabban webszolgáltatások, ezeket jelölhetjük a komponens elemhez illesztett port, azaz kapu elemekkel.

A statikus komponens diagram minden olyan tervezésnél, vagy utólagos dokumentációnál jól használható, ahol a zárt egységet képező komponensek egymással folytatott kommunikációját akarjuk kiemelni. A diagram segítségével a komponensek egymáshoz intézett kéréseit, utasításait szeretnénk nevesíteni, a paraméterként átadott és visszatérési értékként kapott adatok meghatározása mellett.

Ellenőrző kérdések:

Miért előnyösebbek a moduláris szoftverek a monolitikusnál?

Hogyan írhatjuk le a komponensek egymás közötti kommunikációját?

Mi az, hogy kommunikációs port a komponenseken?

Mi a kommunikáció iránya a fenti ábrán a Vezetői információs modul és az ENapló modul között?

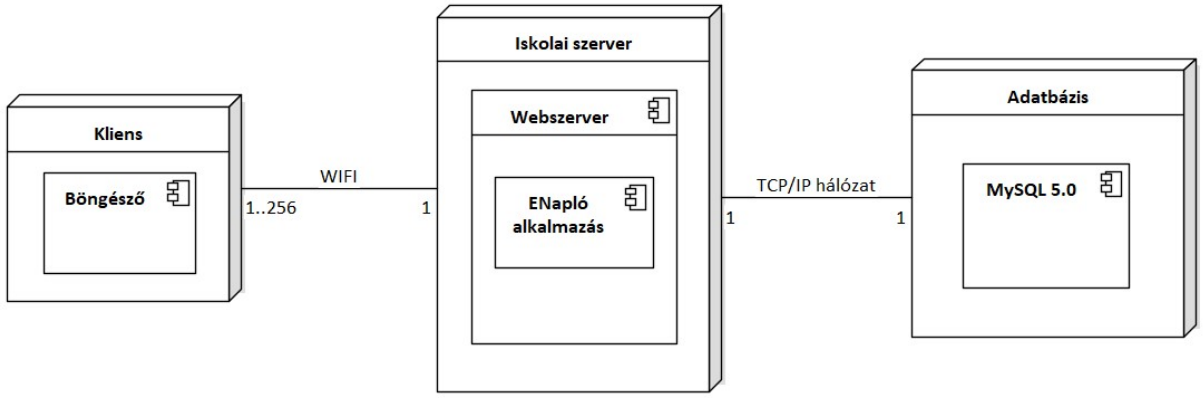
Mai szoftverekben mi a leggyakoribb kommunikációs pont?

Telepítési diagram

Az elkészült szoftvert telepítenünk kell, és az ritkán olyan egyszerű, hogy egy letöltött, vagy máshogy megszerzett fájlt elindítunk a gépünkön. Tervezni kell, és ennek UML eszköze a fejezetben bemutatásra kerülő telepítési diagram.

A modern, komplex rendszerek esetén a sikeres telepítés sok információt igényel. Tudnunk kell, hogy mely hardver elemeken lesznek elhelyezve az adatbázis sémák, ezek milyen protokollok segítségével érhetők el, melyik szoftver komponens biztosítja a webes hozzáférést és melyek a karbantartáshoz, adminisztrációhoz szükséges GUI[9] felületeket. Különösen azok a rendszerek esetében lényegesek ezek az információk, amelyeket folyamatosan fejlesztenek, mint például a közigazgatásban használt szoftverrendszereket. Nem csak a telepítés igényli ezeket az információkat, a fenntartás során is fontos dokumentáció a folyamatosan kiegészített, naprakész telepítési elrendezés.

Az UML ehhez is biztosít speciális, a célnak megfelelő jelölésrendszert. Ebben az egyes hardver komponensek a csomópontok és a csomópontokat összekötő relációk jelzik a kapcsolatokat közöttük. A kapcsolatok tulajdonságai, attribútumai megadják az ismertségi viszonyokat, és az egyes csomópontokon elhelyezett komponensek, modulok a telepítendő szoftver komponenseket, és a komponensek közötti adatáramlási irányokat is megadhatják.



Az ábrán látható diagram tájékoztat arról, hogy a jelen telepítés esetében az iskolai szerver egy webszervert futtat, az alkalmazásunk a webszerver konténerben fut, és egy külön gépen telepített MySQL adatbázissal van kapcsolatban. A szervert az iskolaszerte telepített kliens gépek böngészőn keresztül tudják elérni. Az egyes elemekhez részletes leírások is megadhatók a komponensek pontos specifikációjával.

Bár ez a diagram nem feltétlenül szükséges egy szoftver telepítése során, a szöveges leírást sokan tökéletesen megfelelően tartják, de több olyan eset is ismert, ahol az UML telepítési diagramja segített egyértelművé tenni a telepítési leírást. Ez a dokumentáció a karbantartás, adminisztráció során is sok esetben bizonyult nélkülözhetetlennek.

[9] Graphic User Interface, grafikus felhasználói felület. Az ablakozó rendszerek, kliensek általános neve, szemben a webes felületekkel.

Ellenőrző kérdések:

Hogyan ábrázoljuk telepítési diagramon a hardver és a szoftver elemeket?

A telepítés megtervezése mellett mikor célszerű használni telepítési diagramot?

Az ábra szerint hány szerverként szolgáló gépet kell beszereznünk?

Az ábra szerint mi a kiszolgálható kliensek maximális száma?

Az ábra szerint az ENapló alkalmazás milyen környezetben fut?

További statikus diagramok

Az UML, mint magas szinten karbantartott és támogatott specifikáció, igyekszik minden helyzetre és igényre a megfelelő elemeket, relációkat és diagramokat biztosítani. Így a szerkezetet leíró statikus diagramok között vannak kevésbé népszerű és ezért ritkábban használt nézetek is. A teljesség kedvéért ezeket ismerhetjük meg ebben a fejezetben.

Az UML osztálydiagramja az alkalmazás adattároló osztályait és azok statikus kapcsolatait írja le, de az adatok aktuális értékeiről nem áll rendelkezésre információ. Felmerült az igény, hogy speciális esetekben, a futó alkalmazásban létrehozott objektumokat, azok üzenetváltásait és az adatok konkrét értékét is legyünk képesek feltüntetni. Az igényhez létrehozták az UML objektumdiagramját. Ez azonban erősen programozás közeli információkat tartalmaz, így tehát nem tipikus tervezési, inkább csak értelmezési diagram. Ennek megfelelően jelentősége általában csekély.

Az UML 2.0 verziójában létrehoztak egy kompozit diagramot is, vagy más néven összetett szerkezeti diagramot. Ez a diagram specifikusabb lehet, mint a statikus osztálydiagram, mert annak futás közben mutatott elemeit is megadja. Ehhez feltüntethetjük az osztályok belső szerkezetét is, a várható viselkedéssel (funkcionalitással) és a futás közben létrejövő kapcsolatokkal együtt. Feltehetően a komplexitása miatt nem lett nagyon népszerű, egyszerűbben áttekinthető specifikus diagramokkal kiváltható az alkalmazása.

A statikus diagramok közé soroljuk az UML csomagdiagramját is. Létrehozását az indokolta, hogy az objektumorientált szoftverekben igen nagyszámú osztályt hozunk létre, de a korábbi fejlesztések eredményeképpen és a programnyelv által alapból biztosítva is több tízezer az osztályok száma. Ekkora szám esetén elkerülhetetlen a név duplikáció, ami persze beláthatatlan problémákat okozhat. Az UML ezt a problémát a csomag fogalom bevezetésével és ennek grafikus ábrázolásával igyekezett kivédeni. Azonban a programnyelvek maguk is bevezették a csomag, vagy általánosabban fogalmazva a névtér fogalmát. Ezzel logikailag csoportosítani lehet az osztályokat, így jelentősége a tervezésben csekély. Ennek megfelelően ez a diagram is inkább csak áttekintésre jó, ha a használt tervezőeszköz támogatja a diagramon feltüntetett csomagok lépcsőzetes kibontását.

Használati eset diagram

Ebben a fejezetben az UML talán legnépszerűbb, sokak által elsőként megismert diagramjáról, a használati eset diagramról lesz szó.

Az UML nem csak a tervezés alatt lévő szoftverek statikus, adatszerkezeti, vagy komponens szintű leírására alkalmas, hanem a funkciók összegyűjtése, részletezése, valamint kapcsolódásaik feltárása is történhet grafikus eszközök segítségével. Ennek a törekvésnek a legfontosabb diagramja a használati eset diagram. Rögtön hozzátehetjük, hogy az agilis szoftverfejlesztési módszertanok által előnyben részesített felhasználói történetek is tervezésbe illeszthetők a használati esetek grafikus jelöléseivel. Így elmondhatjuk, hogy a szoftverfejlesztés funkcionális leírására módszertantól függetlenül alkalmas ez a megközelítés.

A diagram felépítése egyszerű és nagyon közel áll a funkcionális specifikációhoz. Aktorokat, azaz szereplőket határoz meg, és a szereplőkhöz rendeli hozzá a tevékenységeket, amit az adott szereplő a rendszeren vagy a rendszerrel végezhet. A szereplők nem a konkrét felhasználók, hanem azonos jogosultságú felhasználó csoportok, tulajdonképpen szerepek, amelyeket a szereplők felvehetnek.

A funkciók (tevékenységek) a használati esetek, amelyeket lehetőleg egymástól független módon kell meghatározni. Ebben az értelemben az aktor kiterjeszthető. Adott szoftverrendszernek maga a rendszer is aktora lehet – gondoljunk például olyan időzített tevékenységekre, mint egy adatbázis mentés, vagy automatikus email figyelmeztetés vagy értesítés kiküldése a regisztrált felhasználóknak.

Több komponensű rendszerek esetében adott komponens is tekinthető a specifikáció szerint aktornak egy másik komponenssel szemben, hiszen ennek a komponensnek a funkcióit, mint rendszer felhasználó veszi igénybe.

A használati eseteket nem könnyű meghatározni, a fő probléma a funkció határaival van. Mi tekinthető tehát egyetlen önálló használati esetnek? Jó iránymutató lehet a tervezés alatt álló szoftver, szintén tervezés alatt álló menüszerkezete. Amit egyetlen menüpontnak tervezünk megvalósítani, az többnyire egyetlen használati eset is lesz. Így például a gyakran használt mentés és mentés másképp funkciók két használati esetnek számítanak még akkor is, ha sok közös elemük van. A varázslószerű működés, amikor a rendszer lépésenként végig vezet egy bonyolultabb feladaton, használati esetek láncolását jelenti egy komplex funkció megvalósítása során.

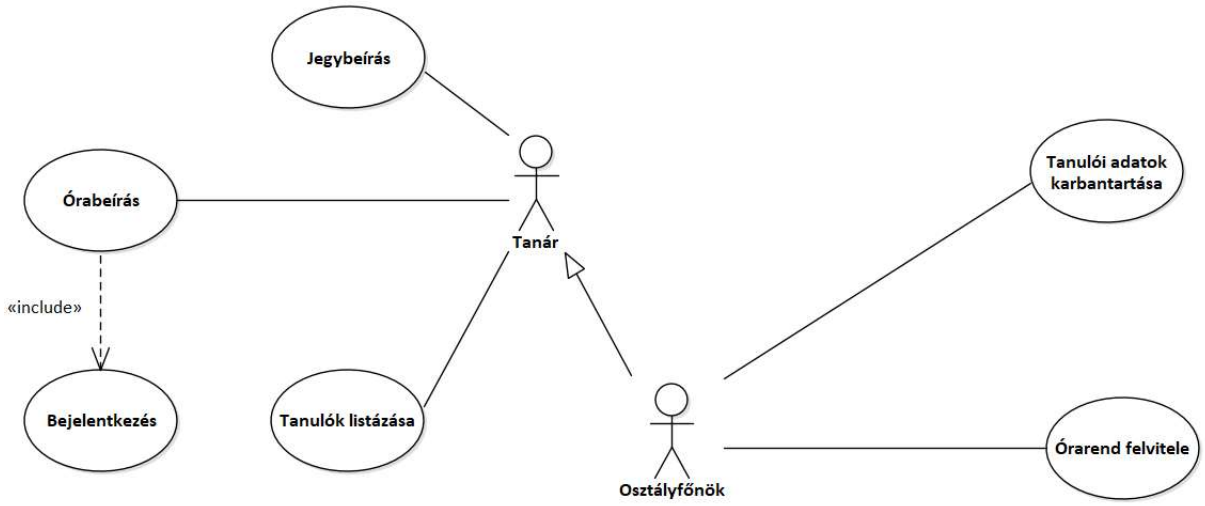
Az UML diagramok lényegét az elemek között felvehető relációk, kapcsolatok adják, ezek határozzák meg a nézeteket. Itt sincs másként, az aktor és a használati esetek között a tipikus reláció a használat (use) kapcsolat, amely pontosan az elnevezés szerint értelmezendő, az aktor használja az adott funkciót. Az aktorok között öröklődési kapcsolatokat fedezhetünk fel, ami konkrétan az aktor által elérhető funkciók öröklését jelenti. Így egy általános felhasználó és egy adminisztrátor felhasználó esetében az admin a gyerekelem a kapcsolatban, mert az öröklődés értelmében a gyerekelem mindent tud, amit az őse, de vannak plusz funkciói is.

A használati esetek között is felismerhetünk kapcsolatokat. Két használati eset között a befoglaló kapcsolat (a szakirodalomban *include* néven szerepel) azt jelzi, hogy az egyik használati eset a futása során a másikat teljes egészében magába foglalja, azaz lefuttatja, feltétel nélkül. Mégis külön használati esetként szerepel a befoglalt eset, több okból is. Egyrészt azért, mert más használati esetek is magukba foglalják, vagy másrészt azért, mert a befoglalt funkcionalitás bizonyos fokú önállóságot élvez, külön fejleszthető.

Egy másik gyakori kapcsolat a kiterjesztés, a szakirodalomban az extend kapcsolat. Jelentése az, hogy az egyik használati eset bizonyos feltételek teljesülése esetén (ez lehet például felhasználói igény) a működését kiterjeszti egy másik használati eset bevonásával. Példaként szolgálhat egy adatmódosítás, ami igény szerint egy értesítés kiküldésével együtt is történhet, on-line módosítás esetén. Az include kapcsolattól eltérően itt mindig van feltétel.

Az UML használati eset diagramja különleges abból a szempontból, hogy míg a többi diagram elsősorban a grafikus ábrázolást hangsúlyozza, használati eset nem létezhet a funkció forgatókönyv-szerű leírása nélkül. Azaz itt a szöveges leírás nem opcionális, hanem kötelező elem. Minden használati esethez legalább egy forgatókönyv tartozik, de lehetnek alternatív, feltétel függő forgatókönyvek és kivételes működést kezelő forgatókönyvek is.

Az agilis fejlesztési módszertanok felhasználói történetei is hasonló módon kezelhetők, itt is lényegében a forgatókönyvek alakulnak ki a felhasználó és a tervező, fejlesztő közötti megbeszélések során. Így a kétféle megközelítés UML szempontból azonosan kezelhető. Amennyiben szükségesnek tartjuk a specifikus jelölést, az UML elemek sztereotipizálhatók, tehát megadható, hogy az azonos jellegű grafikai jelölés használati esetet vagy felhasználói történetet jelent adott diagram esetében.



Az ábrát kiegészíthetjük még azzal, hogy a használati eset diagramok mindig a felhasználók által elvárt rendszerfunkciókat írják le. Az aktorok a felhasználók, az eltérő szerepű felhasználók különböző aktorok. A "use" reláció alapértelmezett ebben a kontextusban, így külön nem is jelöljük a típusát vagy az irányítottságát. Az aktorok között specializáció, azaz öröklődés lehetséges, ez itt annyit jelent, hogy az osztályfőnök aktor minden olyan tevékenységet el tud végezni, amely a tanár aktor tevékenysége. Jelölhetünk egyéb relációkat

is a diagramon, esetünkben az "include" kapcsolat jelzi, hogy a tanár órabeírás funkciója nem futhat le jelszavas bejelentkezés nélkül, azt mintegy magába foglalja.

Ellenőrző kérdések:

Mit ábrázol a használati eset diagram?

Hogyan ábrázolhatók az agilis fejlesztési módszertanok által bevezetett felhasználói történetek (user story)?

Mit értünk a használati esetek <<include>> kapcsolatán?

Mit értünk varázslószerű működés alatt? Hogyan ábrázolható ez használati eset diagramon?

Hogyan értelmezhető az aktorok közötti öröklődés (generalization) kapcsolat?

Aktivitás diagram - önmagában és szcenárió diagramként

A használati eset diagram ugyan talán a legnépszerűbb UML diagram, de a használati esetek részletes leírásához, a jobb megértés érdekében nélkülözhetetlen az aktivitási diagram is. Ismerkedjünk meg tehát vele ebben a fejezetben.

Az UML egyik erőssége, hogy a korábban már létezett tervezési nézeteket kicsit korszerűsítve, a fejlődésnek megfelelő átalakításokkal magába foglalta. Így került be az UML eszköztárába a programozás objektumorientált paradigmájának kialakításakor bevezetett osztály-, interfész- és objektumfogalom, illetve azok grafikai jelölése, valamint a tipikus relációik ábrázolására kitalált rendszer. Hasonlóképpen a programozás, a szoftverek működésének leírására használt folyamatábra is bekerült az eszköztárba, aktivitás diagram néven.

Elsődleges felhasználása itt is a vezérlés leírása. Az induló és leállító csomópontok vagy események között - a kibontásnak megfelelően - kisebb-nagyobb mértékben részletezett aktivitás és akció elemeket ábrázolunk. Ezeket tipikus esetben a döntések kimeneteit és a vezérlés irányát mutató folyamat relációk, kapcsolatok kötik össze. Ezek a relációk mindig irányítottak.

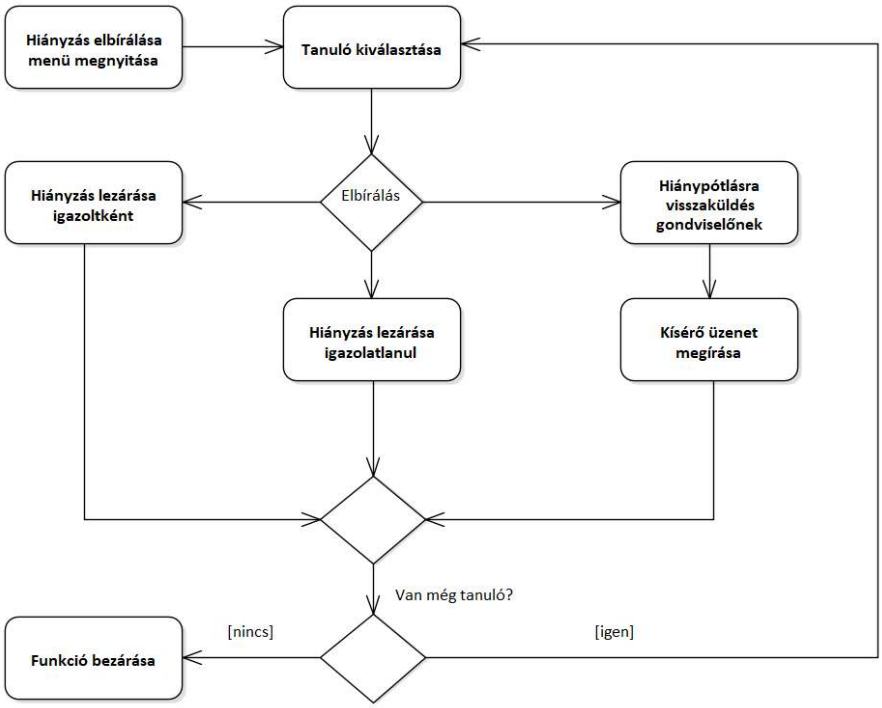
A tetszőleges mértékű részletezés eszköze az akció és az aktivitás elemek megkülönböztetése. Bár mind a kettő grafikailag ugyanúgy néz ki, az aktivitás általában több akcióelemből áll, míg az akció elemet adott összefüggésben, kontextusban atomi tevékenységnek[10] tekintjük. Így lehetőségünk nyílik arra, hogy a tervezés folyamán nagyobb tevékenység blokkokkal indítsunk, ezzel mintegy felvázolva a szoftver működését, átfogó vezérlését, majd később ezeket a szükséges mértékben részletezzük, kifejtsük.

Az akciók és az aktivitások tetszés szerint osztályozhatók sztereotipizálással. Egyes tervező eszközök esetében a sztereotípiák grafikusan, ikonok formájában is megjeleníthetőek a jobb áttekinthetőség érdekében.

A grafikus tervezés kiegészíthető az elemekhez rendelt szöveges információkkal, forgatókönyvekkel is, ha nem akarunk túl részletes grafikai tervezést folytatni.

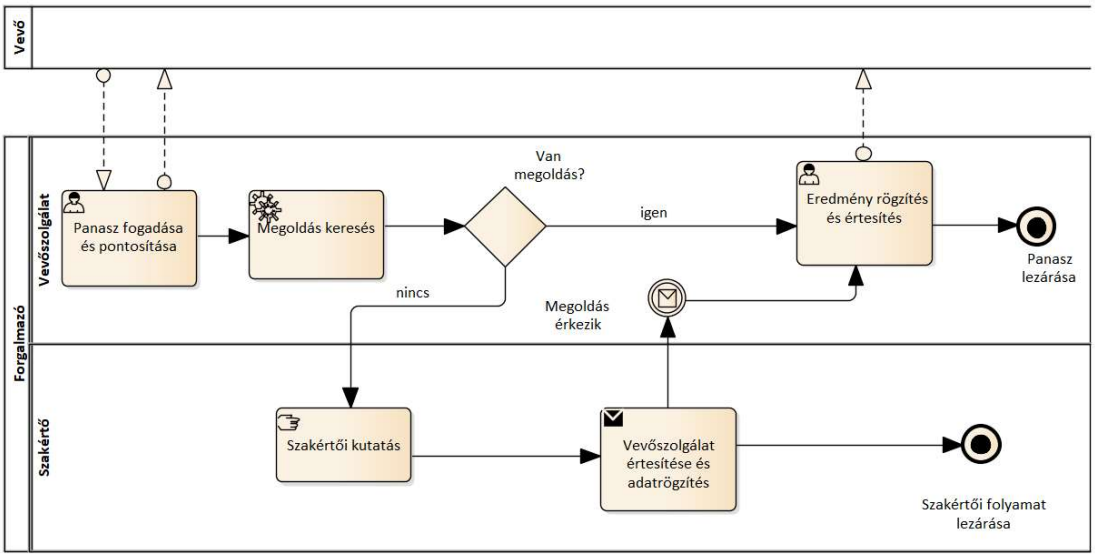
Az aktivitás diagram jelentősége azonban nagymértékben megváltozott az újabb szoftvertervezési módszerek bevezetésével. Ma már alig van arra igény, hogy egy szoftver működését, vezérlését teljes egészében megtervezzük, inkább az egyes funkciók részletes tervezésére helyeződött át a hangsúly. Mivel azonban a funkciók, azaz a használati esetek forgatókönyvei időnként túl bonyolultak, több döntés is történik a futás során, az aktivitás diagram újabb felhasználása a használati esetek forgatókönyv (azaz szcenárió) diagramja lett.

[10] Atomi folyamat az a tevékenység, amelynek lépései vagy mind együtt végrehajtódnak, vagy egyik sem hajtódik végre. Szoftvereknél ennek meghatározása igen jelentős, mert még kivételes eseménynél is gondoskodni kell az atomi folyamat megszakadása esetén a kiindulási állapot helyreállításáról.



Az ábra az ENapló alkalmazás egyik funkcióját, az Osztályfőnök hiányzás elbírálás tevékenységét (azaz használati esetét) részletezi szcenárió diagramként alkalmazott aktivitás diagram segítségével. Lássuk be, hogy jóval könnyebb így áttekinteni a döntés folyamatát, mint szöveges leírás alapján.

Azokat az eseteket, amikor viszont az üzleti folyamatok leírása elengedhetetlen, mert a fejlesztendő szoftver alapját képezi, az üzleti folyamat modellezés (BPM[11]) eszközkészletével tudjuk megtervezni. Bár külső megjelenése és ábrázolástechnikája szinte azonos az UML-nél megszokottal, ez mégsem az UML specifikáció része, azaz szigorúan véve nem UML. Tekintsük meg az alábbi ábrát, amely egy panasz fogadásának és kezelésének az üzleti folyamatát mutatja be BPM jelölésekkel:



[11] Business Process Modeling, üzleti folyamat modellezés

Ellenőrző kérdések:

Mi az összefüggés a folyamatábra és az aktivitás diagram között?

A nyilak milyen relációkat jelölnek az egyes aktivitások/akciók között?

Mit értünk döntések alatt egy aktivitás diagramon? Minek alapján történnek döntések a szoftverekben?

Mit értünk atomi folyamat alatt?

Mit értünk szcenário diagram alatt?

Állapotgép és tipikus alkalmazásai

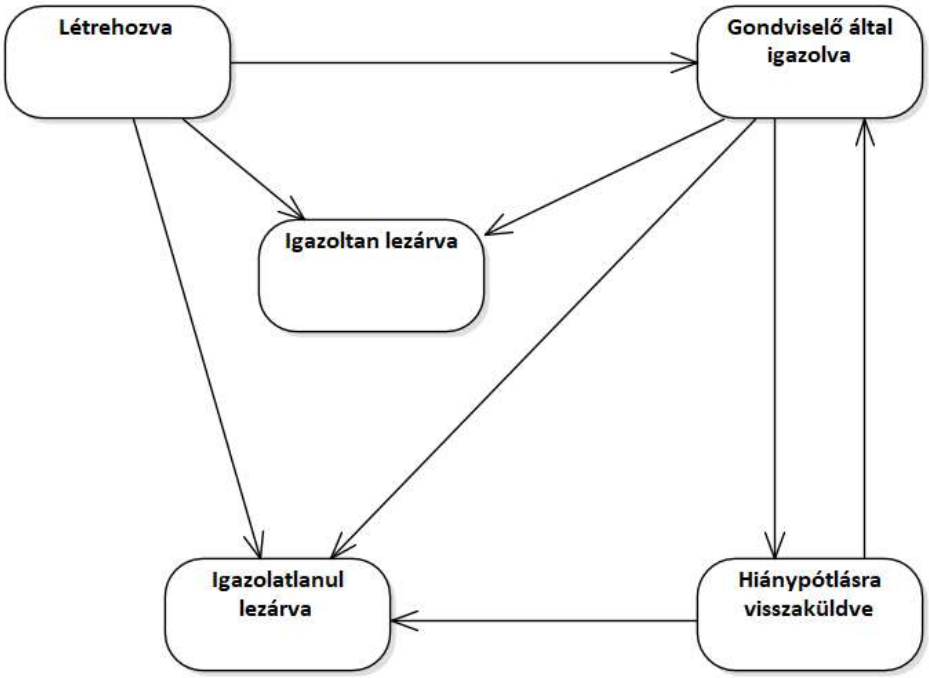
Az UML egyik erőssége, hogy egyes nézetei szokatlan megközelítéssel írják le a szoftver működését, ezzel új és roppant hasznos szempontok kerülnek elő. Az egyik ilyen megközelítést, az állapotátmenetek vizsgálatát és dokumentálását nézzük át ebben a fejezetben.

Szoftvereink egy részében, a tipikus adatkezelő alkalmazásokban gyakran előfordul, hogy a folyamatok lényegét egy bizonyos adatcsoport, azaz objektum, állapotának változásai adják. Gondoljunk egy pénzügyi tranzakcióra, mint objektumra, vagy egy pályázatra, egy hitelkérelemre, vagy akár egy panaszbejelentésre. Ezek mind objektumok a korszerű szoftverekben. Közös jellegzetességük, hogy bizonyos tevékenységek hatására változtatják az állapotukat. Példaként a felvett panasz, kivizsgált panasz, lezárt panasz lehetnek a lehetséges állapotai egy-egy panasz objektumnak.

Ezen szoftverek esetében így tehát az állapotok követése és rögzítése magában az objektumban nélkülözhetetlen az egyszerű működés megvalósítása érdekében. Gondoljuk meg, hogy mennyivel egyszerűbb mentések és beolvasások, keresések esetén, ha a követett objektum tudja magáról az állapotát, és nem valamiféle külső adatbázis táblát kell kezelni az információ rögzítése vagy kinyerése érdekében.

Fontos azt is megemlíteni, hogy az állapotok esetében többnyire nem lehetségesek a tetszés szerinti átmenetek. Minden állapot csak előre meghatározott állapotba mehet át (ez persze több is lehet), így ezeket a szabályokat be kell építenünk a szoftverbe is. Jellemző ezekre az állapotokra, hogy csak akkor érdemes meghatározni ezeket, ha az egyes objektumok mérhető időt töltenek adott állapotban. Egymást követő állapotokat, amikor az egyes átmeneteket nem követi mentés vagy keresés, megjelenítés, nem szoktunk megkülönböztetni.

Az állapotok ábrázolására viszont kiválóan használható az UML állapotgép diagramja. A diagram fő elemei az előre meghatározott állapotok, és az átmenetet reprezentáló kapcsolat, az átmenet reláció. Az ábrát kiegészíthetjük még induló- és végcsomóponttal is, mert az állapotgép mindig indul valahonnan (az állapottal rendelkező objektum létrehozása), és ha a folyamat lefutott, az objektum végső állapotba kerül, amely lehet egy archiválás is.



Az ENapló alkalmazásban a hiányzás objektumok jól körülhatárolható állapotokkal rendelkeznek. A szoftver működését követhetjük ezek állapotváltozásaival, így érdemes ezt a nézőpontot kidolgozni UML diagram formájában.

Az állapotátmenetek a nyilak mentén történnek, és minden átmenet mögött egy funkció, egy használati eset áll. A létrehozva – igazolatlanul lezárva átmenet például a rendszer aktor automatizmusa hatására jön létre, mert amennyiben senki nem törődik a hiányzással, a rendszer 10 munkanap eltelte után lezárja. Hasonló automatikus tevékenység eredménye a hiánypótlásra visszaküldött és az igazolatlanul lezárva állapotok közötti átmenet is.

Ez a nézet egészen más szempontból láttatja az alkalmazást, így sokban hozzájárul a sikeres tervezéshez. A mi történik az objektummal, akkor ha… szituáció elemzése akár új aktor (a rendszer aktor) felvételét is eredményezheti.

Ellenőrző kérdések:

Mit értünk az objektum állapotának változásán?

Mit értünk az állapotátmenet szabályain?

Mit jelez az állapotátmenet reláció? Mi történik a háttérben?

Mivel járulhat hozzá a szoftver tervezéshez az állapotátmenet diagram kidolgozása?

Milyen állapotokat határozna meg egy postai rendszerben a küldemény objektumokra?

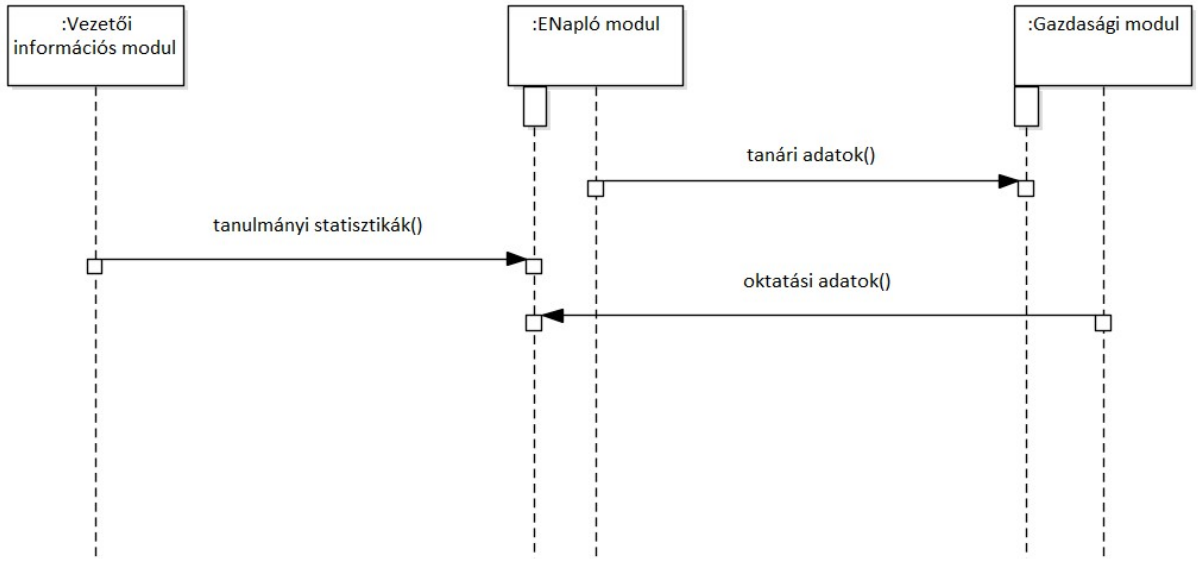
Szekvencia diagram

A szekvencia diagram az UML 2.0 verziójával került be az eszköztárba. A specifikáció gondozói egy olyan nézetet akartak létrehozni, amely segítségével megtervezhető a szoftver futásának vezérlése objektum szinten, azaz feltüntethetők az objektumok egymásnak küldött üzenetei és ezek időbeli sorrendje is. Ismerkedjünk meg ezzel a nézettel ebben a fejezetben.

Ehhez azonban már ajánlatos ismerni az osztályokat, és nem csak adatszinten, hanem operáció, azaz műveleti szinten. Látni kell az osztályok objektumai által biztosított meghívható funkciókat, paramétereikkel és a visszatérési értékekkel együtt. Ennél a diagram típusnál már fellép az előidejűség kérdése, hiszen csak a megtervezett, kidolgozott objektumok üzeneteit használhatjuk fel. Azt is mondhatjuk, hogy ez a nézet egy kölcsönhatás diagram.

Problémát okozhat azonban a modern szoftverrendszerek komplexitása. Ahhoz, hogy egy, a felületen bevitt adatokat ellenőrző, validáló, a mentésüket lebonyolító folyamatot részleteiben ábrázoljunk, sok objektumot és sok üzenetet (azaz funkció hívást) kell a diagramon feltüntetni. Itt lép be az UML egyik komoly problémája. A grafikus felületeken ugyanis nem lehet a láthatóság lerontása nélkül túl sok elemet feltüntetni. Míg a műszaki tervezésben megszokott a méteres oldalhosszúságú tervek használata, és a papíralapú tervezést felváltó szoftverek is ennek megfelelően készültek el, az UML alapú szoftverfejlesztésben ez nem terjedt el. Így ragaszkodnunk kell a sok kisebb, tipikusan A4-es lapokból álló, részletenként történő tervezéshez, ami a szekvencia diagramok esetében nem elegendő az áttekinthető ábrázoláshoz.

A másik akadályozó tényező az objektumok bonyolultsága. Sok funkcióval rendelkeznek, és így ezek együttes ábrázolása szintén problémát okoz. Mellette a szoftverfejlesztés jelenlegi helyzetében és a várható irányzatoknak megfelelően, a főleg programozók által használható szekvencia diagram feleslegesnek bizonyult a tervezésben. Ugyanis a funkciók megtervezése, használati eset alapú leírása elegendő a megvalósításukhoz, nincs reális igény ezek grafikus, szekvencia diagram alapú megtervezéséhez.



Nem veszítette el azonban jelentőségét a szekvencia diagram a komponensek esetében. A korszerű rendszerek egyik legfontosabb tulajdonsága a modularitás, ezért a modulok, komponensek kommunikációjának gondos megtervezése létfontosságú. Ezek tulajdonképpen üzenetek az UML logikája értelmében a komponensek között. Ennek megfelelően a szekvencia diagram továbbra is kiváló és nélkülözhetetlen az üzenetek irányainak és paraméterezésének, valamint a visszatérési értékeknek ábrázolására, azzal a kiegészítéssel, hogy tervezési szinten a komponenseket tekintjük majd a szekvencia diagram objektumainak.

Ellenőrző kérdések:

Mit ábrázol egy szekvencia diagram?

Mit értünk előidejűség alatt a szekvencia diagrammal kapcsolatban?

Mit jelent a kettőspont az elemek elnevezése előtt?

Mit értünk üzenet alatt egy szekvencia diagramon?

Hol használhatók eredményesen a szekvencia diagramok?

Kommunikációs diagram - az agilis módszertanok robusztusság diagramja

Az UML egyik nézete újraéledt az agilis módszertanok elterjedésével. Ennek hátterével, és a nézet újrahasznosításával ismerkedünk meg ebben a fejezetben.

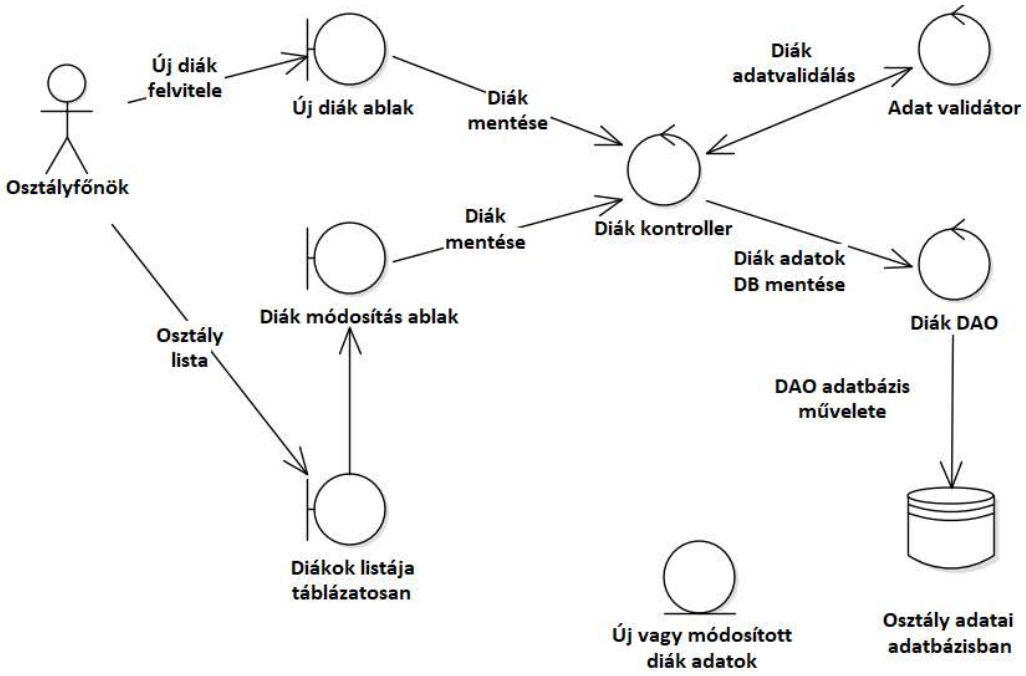
A kommunikációs diagram (korábban kollaborációs diagram) már az UML 1.0 verziójától létező diagramtípus, szerepe hasonló, mint a szekvencia diagramnak. A szoftver futása közben létrejövő objektumokkal, a küldött üzenetekkel foglalkozik. Jelentős különbség a szekvencia diagrammal összevetve az, hogy nem foglalkozik az üzenetek időbeli sorrendjével, az ábrázolásról ez nem olvasható le. Ha mégis szükség lenne rá, akkor ezt csak az üzeneteket képviselő relációk számozásával lehetne megadni.

Kimondhatjuk, hogy a szekvencia diagram kidolgozásával és felvételével az UML eszközkészletébe, a jelentősége megszűnt. Mégis érdemes tárgyalni, mert egyes agilis módszertanok felfedezték maguknak és eljárásukat a használati eset diagramok mellett a kommunikációs diagramok használatára alapozzák. Ez utóbbit azonban átnevezték, és robusztusság (angolul robustness) diagram néven vonult be az eszköztárukba.

Még egy UML ábrázolási lehetőségről érdemes beszélni a robusztusság diagram esetében, ami az objektumok általános tipizálására vonatkozik. Az objektumorientáltság kidolgozása során, az objektumokra előírt kohézió[12] értelmében kialakult, hogy az OO rendszerek MVC architektúrája[13] értelmében minden objektumra megadható egy tevékenység típus. A modell objektumok az adat entitásokkal foglalkoznak, azok futásidejű tárolását, mentését és adatbázis olvasását végzik, a nézet objektumok a megjelenítésre szakosodnak, ezek hozzák létre a felületeket, és a kontroller objektumok, mint az üzleti logika megtestesítői az adatok manipulálását, a felület és az adatbázis közötti átadását végzik.

Ezekre a sztereotipizált objektumokra külön grafikus elemeket dolgoztak ki, ezek használatával áttekinthetőbb lehet az ábra.

[12] Kohézió alatt azt értjük, hogy egy-egy objektum mindig adott célra fókuszál, jól körülhatárolható működési területe van.
[13] Modell-View-Controller architektúra, azaz modell, mint adat, view mint nézet és controller, mint a felület feltöltését, illetve kiolvasását és adat validálást végző architekturális elemek.



Az ábrán az MVC felépítés összes rétege, azoknak együttműködése került ábrázolásra UML alapokon. A felületelemek, mint sztereotipizált objektumok jelennek meg, és ez igaz az adatokat reprezentáló entitásokra – az ábrán a diák objektumok tárolják a diák adatokat.

A kontrollerek azok az elemek, amelyek az üzleti logikát valósítják meg, ez lehet adat összerendezés, de adatellenőrzés is a validálást végző kontrollerben.

Az elemek közötti relációk irányított folyamatokat reprezentálnak, feltüntetik, hogyan adják át egymásnak az MVC rétegek az adatokat, objektumokat.

Ellenőrző kérdések:

Hogyan fogalmazhatjuk meg a kommunikációs diagram szerepét? Mit kíván ábrázolni?

Hol van ma jelentősége a kommunikációs diagramnak?

Hogyan jelenik meg a kohézió és a kapcsolás elve a fenti diagramon?

Mit jelent az MVC betűszó?

Mit reprezentálnak az irányított relációk az elemek között?

További viselkedési diagramok

Mivel az UML specifikáció fenntartói konzorciuma törekszik a legszélesebb körű felhasználásra, így minden számbavehető nézetre kidolgoztak specifikus diagramokat, jelöléseket. Ugyanakkor a visszafelé kompatibilitás érdekében azok a diagramok is az eszközkészletben maradtak, amelyeket az újabb diagramok és a szoftver tervezés új irányelvei többé-kevésbé feleslegessé tettek, vagy jelentőségük és így gyakoriságuk lecsökkent. Ez a fejezet áttekintést igyekszik adni ezekről a nézetekről, diagramokról.

Ezek egyike az időzítés diagram, amely ugyan jól használható specifikus esetekben, bár ritkán kerül elő az általános szoftverfejlesztés során. Lényegében a szekvencia és az állapot diagramok fúziója, az egyes objektumok állapotát az idő függvényében ábrázolja. A szekvencia diagramtól örökölte az objektumok közötti üzenetek feltüntetését. Itt elsősorban az állapotot módosítani képes üzenetekről van szó. Ebből adódik, hogy komolyabb jelentősége csak időfüggő folyamatok esetén van, amelyek lehetnek persze üzleti folyamatok is.

Egy másik szintén ritkán használt diagram a kölcsönhatás áttekintő diagram, amely magasabb szintű folyamatleírásokat képes kezelni. Jelentőségét erősen csökkentette az üzleti folyamatok modellezésének előretörése, amely pont ennek a magas szintű vezérlésnek az ábrázolását, megtervezését tűzte ki a fő irányvonalnak.