Értékünk az ember

Humánerőforrás-fejlesztési Operatív Program

Dr. Sziray József – Kovács Katalin

AZ UML NYELV HASZNÁLATA

SZÉCHENYI ISTVÁN EGYETEM
GYŐR

Magyarország célba ér
NEMZETI FEJLESZTÉSI TERV

Készült a HEFOP 3.3.1-P.-2004-09-0102/1.0 pályázat támogatásával.

Szerzők: Kovács Katalin
egyetemi tanársegéd
(3. fejezet, 4. fejezet, 7. fejezet, 9.1. alfejezet)

dr. Sziray József
egyetemi docens
(Bevezetés, 1. fejezet, 2. fejezet, 4.4. alfejezet, 5. fejezet, 6. fejezet, 7.7. alfejezet, 8. fejezet, 9. fejezet, 10. fejezet)

Lektor: dr. Kallós Gábor
egyetemi docens

A dokumentum használata

Mozgás a dokumentumban

A dokumentumban való mozgáshoz a Windows és az Adobe Reader megszokott elemeit és módszereit használhatjuk.

Minden lap tetején és alján egy navigációs sor található, itt a megfelelő hivatkozásra kattintva ugorhatunk a használati útmutatóra, a tartalomjegyzékre, valamint a tárgymutatóra. A ◀ és a ▶ nyilakkal az előző és a következő oldalra léphetünk át, míg a Vissza mező az utoljára megnézett oldalra visz vissza bennünket.

Pozícionálás a könyvjelzőablak segítségével

A bal oldali könyvjelző ablakban tartalomjegyzékfa található, amelynek bejegyzéseire kattintva az adott fejezet/alfejezet első oldalára jutunk. Az aktuális pozíciónkat a tartalomjegyzékfában kiemelt bejegyzés mutatja.

A tartalomjegyzék használata

Ugrás megadott helyre a tartalomjegyzék segítségével

Kattintsunk a tartalomjegyzék megfelelő pontjára, ezzel az adott fejezet első oldalára jutunk.

Keresés a szövegben

A dokumentumban való kereséshez használjuk megszokott módon a Szerkesztés menü Keresés parancsát. Az Adobe Reader az adott pozíciótól kezdve keres a szövegben.

Tartalomjegyzék

Bevezetés

A szoftvertechnológia területén az elmúlt tíz évben döntő mértékben meg-növekedett az objektumorientált fejlesztés súlya és jelentősége. Ebben a folyamatban fontos szerepük volt azoknak a tervezési módszereknek, programozási nyelveknek, ill. számítógépes segédeszközöknek, amelyek a tíz év során jöttek létre és terjedtek el. Ide sorolható az a kiemelkedő eredmény is, amely az UML nyelv megalkotásában nyilvánult meg.

Az UML (Unified Modeling Language, jelentése magyarul: Egységes modellező nyelv) három kiváló szakember, Grady Booch, Ivar Jacobson és James Rumbaugh egyesített munkájának a terméke. A szerzők a nyelv legelső, 1.0-s változatát 1997-ben adták közre az Egyesült Államokban. Azóta az UML az objektumorientált irányzat egyik modern, széles körben felhasznált alapvető eszköze lett. Elterjedését nagymértékben előmozdította, hogy grafikus elemeken alapszik, ami igen szemléletes és jól áttekinthető alkalmazhatóságra vezetett.

Az UML jelentőségét lényegesen növelte az a tény, hogy sikerült szabványosítani. Megalkotói szerint „az UML egy általános célú vizuális modellező nyelv, amely arra használható, hogy specifikáljuk, szemléltessük, megtervezzük és dokumentáljuk egy szoftver rendszer architektúráját”. Az iménti meghatározással összhangban a nyelv grafikus formában teszi lehetővé a szoftver specifikálását, ill. működésének modellezését, aminek alapján a konkrét implementálást el lehet végezni. A kiinduláskor előállított ún. tervezési diagramok lényegesen különböznek az implementáláskor szöveges formában előállított programozási forráskódtól. A két megjelenési forma ugyanannak a szoftvernek a definiálására szolgál, viszont ez a funkció egymástól eltérő absztrakciós szinteken jut kifejezésre.

A tervezési folyamatban az UML-t a kiindulási fázisban használjuk, azzal a céllal, hogy minél biztonságosabban, áttekinthetőbben és megbízhatóbban készíthessük el a teljes szoftver tervét, az objektumorientált fejlesztési elvhez kapcsolódóan. Az így elkészült tervezési diagramok alapján válik lehetővé a forráskód megírása és a futtatható szoftver elkészítése. Ebben a folyamatban természetes igény az, hogy az UML-fázis és a kódolási fázis teljes összhangban legyenek egymással. Mint ismeretes, az ilyen jellegű ekvivalencia igazolásának folyamatát szoftververifikálásnak nevezzük.

Mindezek után megállapítható, hogy az UML nem más, mint egy tervezési nyelv, ami egy szoftver rendszer minél megalapozottabb kidolgozásának, elkészítésének folyamatát szolgálja. Ebben a tankönyvben bevezető áttekintést adunk az UML-ről, annak grafikus komponenseiről, az egyes komponenseknek az objektumorientált fejlesztésben történő felhasználásáról, valamint a nyelvnek a tervezési folyamatban betöltött modellezési szerepéről. Mindezzel a szoftvertervezés átgondolt és konzisztens végrehajtásának megalapozását kívánjuk elérni.

A könyvben nem szenteltünk teret egyik meglevő programozási nyelv felhasználásának ismertetésére sem. Ehelyett olyan általános elveket tárgyalunk, amelyek az implementációs nyelvek bármelyikénél alkalmazhatók. Az általános elvek megvalósítására azonban néhol bemutatunk egy-egy példát, elsősorban a Java nyelvre támaszkodva.

A könyv tíz fejezetből áll, amelyek tartalma a következő:

Az 1. fejezet az objektumorientált szoftverek legfőbb jellegzetességeit foglalja össze, a klasszikus procedurális szoftverekkel történő összehasonlításban.

A 2. fejezet bevezetést ad az UML nyelvről, majd itt kerül sor annak ismertetésére, hogy a grafikus tervezés és modellezés milyen kapcsolatban áll a forrásnyelvi megvalósítással. A további fejezetrészekben egy olyan fejlesztési modellt mutatunk be, ami szervesen illeszkedik az UML-alapú tervezéshez. Ebben a modellben a fejlesztési folyamat négy egymást követő fázisra van felbontva.

Az UML több különböző modellezési, folyamatleírási lehetőséget biztosít. Mindegyik modellezési módhoz külön grafikus megjelenítési forma, diagramfajta tartozik. A nyelvi elemeket, ill. használatukat a könyv 3. fejezetétől kezdve a 9. fejezetig terjedően mutatjuk be. Az egyes diagramfajták, bemutatásuk sorrendjében a következők:

- Use case diagram (használati eset diagramja): A szoftver rendszer kívülről látható működését írja le, a felhasználók és a rendszerfunkciók egymással való kapcsolatának ábrázolásával. (3. fejezet.)
- Osztálydiagram: Az egyes osztályok belső felépítésének és az osztályok közötti statikus információs kapcsolatoknak a leírására szolgál. (4. fejezet.)
- Interakciós diagramok: Az egyes objektumok közötti együttműködést mutatják be, a köztük fennálló üzenetküldésekkel és azok hatásának

feltüntetésével együtt. Két fajtájuk létezik: a szekvenciadiagram (eseménykövetési diagram), ill. az együttműködési diagram. (5. fejezet.)

- Csomagdiagram: Az egymással szorosabb funkcionális kapcsolatban álló osztályok csoportba sorolására, valamint az egyes csoportok közötti információs függőség leírására szolgál. (6. fejezet.)
- Állapotdiagram: Egy adott objektum különböző állapotait ábrázolja, azoknak a vezérlési feltételeknek a megadásával, amelyek az állapotváltozásokat eredményezik. (7. fejezet.)
- Aktivitási diagram: A vezérlési struktúrákat, valamint az egyes tevékenységek egymást követő, ill. egyidejű, párhuzamos lefolyását mutatja be. (8. fejezet.)
- Komponensdiagram, valamint a telepítési diagram: Az előbbi a szoftverkomponensek hierarchikus elrendezésének és a közöttük létesült kapcsolatoknak a leírására szolgál. Az utóbbi pedig azt ábrázolja, hogy egy szoftverrendszer komponensei milyen hardveregységeken helyezkednek el, és milyen kommunikációs kapcsolatok állnak fenn az egyes összetevők között. (9. fejezet.)

A tankönyv 10. fejezete azokkal a problémákkal foglakozik, amelyek egy adott szoftvertervhez tartozó különböző UML-diagramok egymással való összhangjára, konzisztenciájára vonatkoznak. Ugyanitt lesz még szó a diagramok teljességének feltételeiről is.

A fentiekhez még annyit fűzünk, hogy ezt a tankönyvet azoknak az egyetemi és főiskolai hallgatóknak ajánljuk, akik olyan informatikai képzésben részesülnek, amibe beletartozik a szoftverfejlesztés is.

Budapest, Győr, 2006. május 15.

A szerzők

1. Objektumorientált szoftverek

1.1. A procedurális szoftverek jellemzői

A klasszikus, hagyományos ún. *procedurális* elvű szoftverek már fél évszázada vannak használatban, számos magas szintű programozási nyelv jött létre és terjedt el ebben a körben. Például: Algol, PL/1, Cobol, Fortran, Pascal, vagy a C. Ennek a területnek a modernebb ágát képviselik az ún. *strukturált* elvű szoftverek, amelyek jellemző nyelvei a Pascal és a C.

A procedurális programoknál a kialakított adatstruktúra szolgál arra, hogy a programegységek, programmodulok információt cseréljenek egymással. Ezt a működési elvet az 1.1. ábrán szemléltetjük, ahol három modul, M1, M2 és M3 vesz részt a feladat megoldásában. A modulok rendelkeznek saját, ún. *lokális adatokkal,* ezen kívül egymás között paraméterekkel is tudnak adatot váltani.

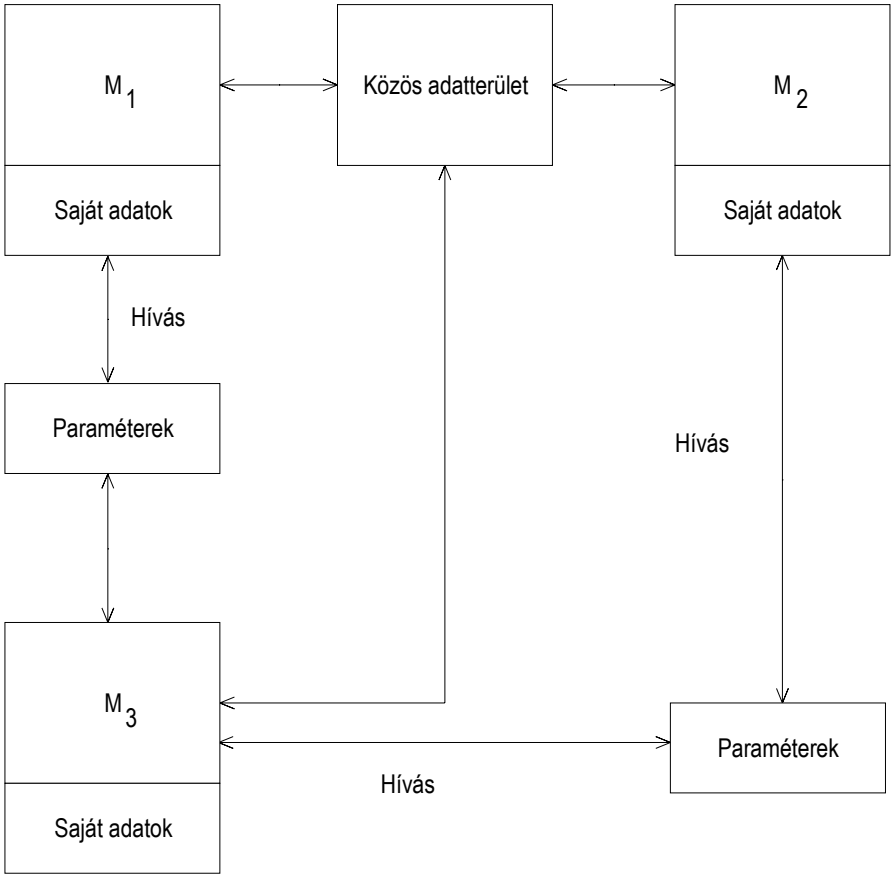
A procedurális programozási nyelveken megírt modulok, a belső vezérlési feltételektől függően, egy-egy jól elkülönült utasítássorozat folyamatos végrehajtásán keresztül fejtik ki működésüket. Ebben a megközelítésben önálló modul lehet például egy Fortran-szubrutin, egy Pascal-procedúra, vagy egy C-függvény. A modulok által használt adathalmazok felépítésére nézve jellemző az azonos hosszúságú mezők ismétlődése, ill. a fix hosszúságú rekordok használata.

Egy programrendszer működési folyamatait a különböző funkciókat megvalósító modulcsoportok együttese valósítja meg. A procedurális szoftvereknél a folyamatok és az adatok szét vannak választva. Ebből adódóan a funkciók megosztásán kívül még külön meg kell tervezni az adatstruktúrát is, mégpedig úgy, hogy az összhangban legyen a funkciók együttesével. Mint ismeretes, a megtervezett adatstruktúra összetétele, az adatok elrendezése igen nagy mértékben befolyásolja a számítási folyamatok hatékonyságát.

A legújabb alkalmazások már egyedi, speciális felépítésű adatbázist igényelnek, amelynek szervesen kell igazodnia a feldolgozás jellegéhez. Ilyen alkalmazások, ill. adatbázisok a következők:

- *Számítógépes tervezőrendszerek (CAD: Computer-aided Design):* A mérnöki tervezési adatokat tartalmazzák, beleértve a tervezett komponenseket, a köztük levő kapcsolatokat, valamint a különböző tervezési verziókat.

- *Számítógépes szoftvertechnológia (Computer-aided Software Engineering, CASE):* A szoftverfejlesztők számára szolgáló adatokat foglalja magában, mint például a forráskódot, a modulok közötti függőségeket, a felhasznált változók definícióját, a fejlesztési folyamat menetét.
- *Multimédia:* Képi, grafikus, térképészeti, hang, zenei, videó típusú információt kódoló adatokat tartalmaz.
- *Irodaautomatizálási rendszerek:* Dokumentumok tárolására, nyilvántartására, kezelésére, keresésére, ill. munkafolyamatok adminisztrálására szolgáló adatok találhatók bennük.
- *Szakértői rendszerek:* Nemcsak az adatokat, hanem az emberi szaktudást reprezentáló összefüggéseket, szabályokat is hordozzák.



1.1. ábra. Procedurális modulok együttműködése

1.2. Az objektumorientált szoftverek jellemzői

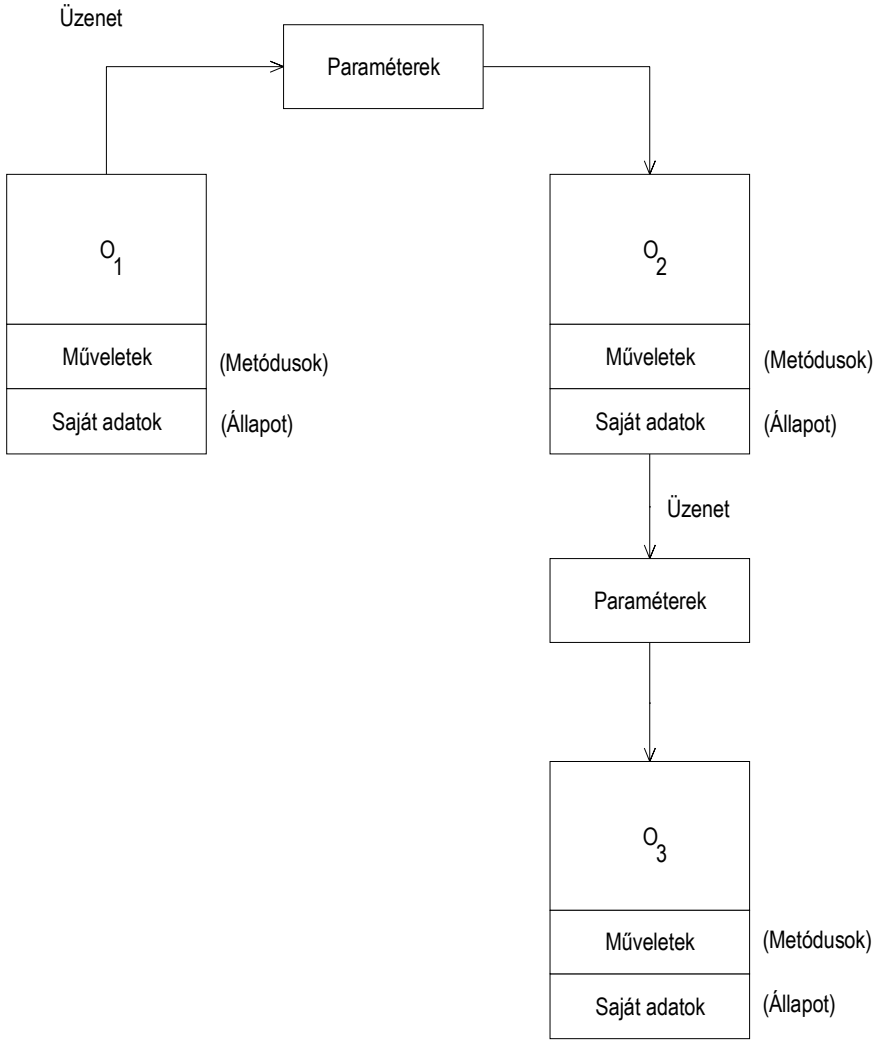
Az *objektumorientált szoftverek,* röviden *OO-szoftverek* tervezési filozófiája, tervezési elve lényegesen eltér a procedurális elvű szoftverekétől. Az OO-stratégia az információ elrejtésén alapul. Egy szoftver rendszert úgy kell tekintenünk, mint egymással kölcsönhatásban álló objektumokat, amelyeknek saját, privát állapotuk van, ahelyett, hogy egy közös globális állapoton osztoznának. Ebben a megközelítésben a programkód és a hozzá tartozó adatok ugyanabban a szoftveregységben találhatók. Ezek az egységek maguk az objektumok.

Az OO-technológia alapvető koncepciója az, hogy a feladatok végrehajtása objektumok közötti *üzenetek küldése* következtében történik meg. Az ilyen jellegű működés megköveteli, hogy az objektumokhoz definiálva legyenek azok az üzenetek, amelyekre azok reagálnak.
Az OO-szoftverek fontosabb jellemző vonásai a következők:

- Az objektumok független egységek, amelyek önállóan működnek. Mindegyik objektum saját adatokkal rendelkezik, amelyek mindenkori aktuális értékei az objektum *állapotát* fejezik ki.
- Egy objektum működése a benne meglevő önállóan kezelt, elkülönített utasításcsoportok, az ún. *műveletek* (angolul *operations*), vagy más néven *metódusok (methods)* végrehajtása révén valósul meg.
- Az objektumok azáltal kommunikálnak egymással, hogy egymás műveleteit, metódusait hívják meg, ahelyett, hogy változóértékeket osztanának meg egymás között. Egy művelet hívása üzenetküldéssel történik meg. Az üzenetek ebben a szisztémában is hordozhatnak vezérlő paramétereket.
- Az objektumok által képviselt programok akár szekvenciálisan, akár párhuzamosan hajthatók végre. Párhuzamos végrehajtásnál az objektumok műveletei egyidejű, konkurrens módon mennek végbe. Ez egyprocesszoros gépen nem jelent szó szerinti egyidejűséget, csak a végrehajtás szervezésében, ütemezésében jelentkezik.

Az objektumok közötti együttműködés vázlatos sémáját az 1.2. ábrán láthatjuk. Itt az O1, O2 és O3 objektumok szerepelnek együtt.

Az OO-környezetben értelmezett üzenet nem jelent olyan fizikai jelküldést, mint ami a számítógép-hálózatok csomópontjai (hosztjai) között megy végbe. Esetünkben egy üzenet nem más, mint objektumok között továbbított, valamilyen konkrét feladatmegoldásra vonatkozó kérés.



1.2. ábra. Objektumok együttműködése

Mivel egy objektum kizárólag üzeneteken keresztül tart fenn kapcsolatot a külső környezetével, ezért lehetőség van arra, hogy módosítsuk az objektum változóit és metódusait, anélkül, hogy ez befolyásolná más objektumok működését. Ez a lehetőség, amellyel élve egy objektumot úgy tudunk módosítani, hogy az nem hat ki a rendszer többi részére, az egyik legfontosabb előnye az OO fejlesztési elvnek, szemben a procedurális fejlesztési elvvel. Az OO-szoftverek létrehozásához számos jól bevált programozási nyelv áll rendelkezésre. Ilyenek például: Smalltalk, C++, Java, Perl, PHP.

2. Egy objektumorientált fejlesztési modell

2.1. Az UML nyelv megjelenése

Az objektumorientált elemzési és tervezési módszerek az 1980-as évek végén, ill. az 1990-es évek elején jelentek meg. Ennek a fejlődésnek egyik fontos későbbi állomása volt az *UML nyelv* kidolgozása. Az elnevezés az angol *Unified Modeling Language* (magyarul: Egységes modellező nyelv) kezdőbetűiből származik. A nyelv három kiváló szoftverfejlesztő, Grady Booch, James Rumbaugh és Ivar Jacobson együttes munkája révén jött létre, az Egyesült Államokban, 1997-ben. Ez volt az UML 1.0-s verziója. Azóta az UML az objektumorientált irányzat egyik modern, széles körben felhasznált alapvető eszköze lett. Elterjedését többek között annak köszönheti, hogy grafikus elemeken alapszik, ami igen szemléletes és jól áttekinthető alkalmazhatóságra vezetett. Az 1.0 után a szerzők elkészültek egy újabb változattal, a 2.0-val, amit 2004-ben bocsátottak ki.

A nyelv elterjedését nagymértékben előmozdította az, hogy sor került a szabványosítására. A szabványosítási folyamatot az OO-fejlesztés területén kulcsszerepet játszó OMG elnevezésű (Object Management Group) konzorcium hajtotta végre. Az OMG konzorcium a Hewlett-Packard, az IBM, valamint a Microsoft cégekből tevődik ki, mindegyikük az informatikai fejlesztések óriása az USA-ban.

Megalkotói szerint „az UML egy általános célú vizuális modellező nyelv, amely arra használható, hogy specifikáljuk, szemléltessük, megtervezzük és dokumentáljuk egy szoftver rendszer architektúráját”. Az iménti meghatározással összhangban a nyelv grafikus formában teszi lehetővé a szoftver specifikálását, ill. működésének modellezését, aminek alapján a konkrét implementálást el lehet végezni. A kiinduláskor előállított ún. *tervezési diagramok* lényegesen különböznek az implementáláskor szöveges formában előállított programozási forráskódtól. A két megjelenési forma ugyanannak a szoftvernek a definiálására szolgál, viszont ez a funkció egymástól eltérő absztrakciós szinteken jut kifejezésre.

A tervezési folyamatban az UML-t a kiindulási fázisban használjuk, azzal a céllal, hogy minél biztonságosabban, áttekinthetőbben és megbízhatóbban készíthessük el a teljes szoftver tervét, az objektumorientált fejlesztési elvhez kapcsolódóan. Az így elkészült tervezési diagramok alapján

válik lehetővé a forráskód megírása és a futtatható szoftver elkészítése, ami a fejlesztési folyamat végső célja. Ebben a folyamatban természetes igény az, hogy az UML-fázis és a kódolási fázis teljes összhangban legyen egymással. Mint ismeretes, az ilyen jellegű ekvivalencia igazolásának folyamatát szoftververifikálásnak nevezzük.

Az UML arra szolgál, hogy vizuálisan specifikáljuk, megjelenítsük, ill. dokumentáljuk egy szoftverfejlesztés fázisainak eredményét. A nyelv igen hasznos a különböző tervezési alternatívák leírására, valamint az eredmények dokumentálására. Az UML-diagramok egyaránt alkalmasak a megvalósítandó objektumorientált rendszer statikus és dinamikus megjelenítésére.

A statikus képet adnak: az osztálydiagram, a csomagdiagram, a telepítési diagram, továbbá a komponensdiagram. Ezek az objektumorientált terv alkotó elemei között meglevő állandó kapcsolatokat dokumentálják.

A dinamikus képet a use case diagram, az aktivitási diagram, az interakciós diagramok, valamint az állapotdiagram szolgáltatják. Az ide sorolt diagramok a működésben, vagyis a programfutás közben megnyilvánuló változásokat, „mozgásokat" tükrözik.

Az UML az OO-világban elterjedt Ada, Smalltalk, C++, vagy a Java nyelvek bármelyikéhez felhasználható a megírandó programok tervezésében. Abban a tekintetben sem korlátoz bennünket, hogy milyen jellegű szoftver elkészítéséhez használjuk. Egyaránt alkalmazható valós idejű, webes, hálózati, vagy akár adatfeldolgozó rendszerekhez is. Jóllehet ez egy grafikus nyelv, de mégis ugyanúgy rendelkezik szintaktikai szabályokkal, mint a karakterekre épülő nyelvek.

Önmagában véve az UML csak egy modellezési nyelv, semmilyen módszer, tervezési megfontolás nem képezi integráns részét. Az idevonatkozó tervezési módszerek ugyanakkor arra valók, hogy a nyelv felhasználására adjanak irányelveket, megoldásokat, jól beváló „recepteket". Az ilyen jellegű módszerek harmonikusan összefüggő együttesét *módszertannak* vagy *metodológiának* nevezzük. Egyik ilyen fontos és széles körben elterjedt fejlesztési módszertan az, amelyet maguk a nyelvalkotók, Booch, Rumbaugh és Jacobson dolgoztak ki és alkalmaztak számos UML-projektben. A módszertan elnevezése: *RUP (Rational Unified Process),* ami a tervezési lépések szigorú rendbe foglalására irányul.

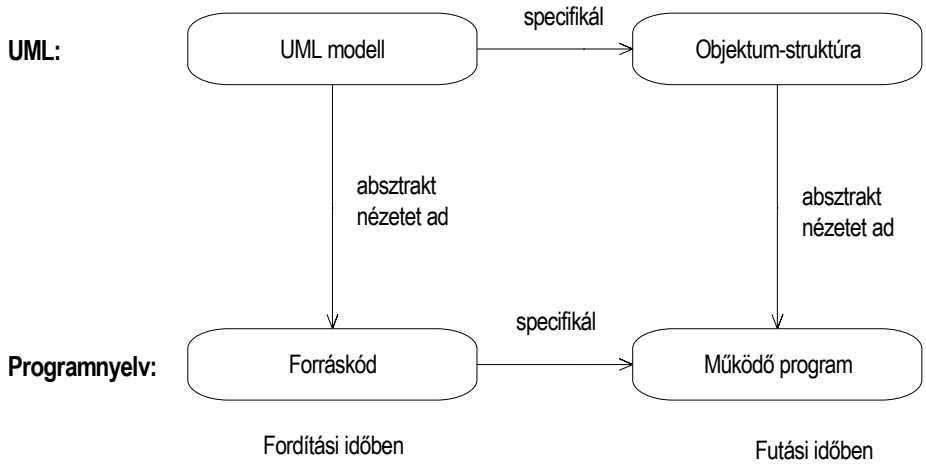
Ennél a pontnál érdemes megemlíteni a következőket: A RUP elnevezésben a hangsúly a „Unified Process"-en van, ami „Egységes folyamat"-ot jelent. A „Rational" előtag annak a cégnek a nevéből származik, amelyben a három alkotó dogozott a módszertan közös megalkotásakor. Ez a cég a

Rational, Inc. volt az USA-ban. A szoftvertechnológiában az UML-lel elért egyértelmű áttörés, és a cég által kifejlesztett Rational Rose elnevezésű CASE-eszköz piaci sikere után, 2003-ban az IBM felvásárolta a Rational, Inc.-et. Emiatt újabban már csak a Unified Process (UP) elnevezés kezdi jelölni a fejlesztési módszertant. A Unified Process jelenleg széles körben terjed a világon, a fejlesztési területek csaknem teljes spektrumát fedi le már.

2.2. A tervezési modell és a programkód viszonya

Egy szoftver rendszer terve és a tervet realizáló forráskód között, legalábbis elvben, szoros kapcsolatnak és konzisztenciának, összhangnak kell fennállnia. A terv, a kód és a kész rendszer közötti kapcsolatokat a 2.1. ábrán mutatjuk be.

Az ábrán különbséget teszünk a fordítási időben és a futási időben megjelenő reprezentációk között. A forráskód, ami például Java, vagy C++ nyelven íródott, olyan dokumentum, ami egyértelműen definiálja a program működését. Azt a működést, amit a lefordítás, betöltés és elindítás előz meg. Az UML diagram is egy dokumentum, ami egy rendszert specifikál, de ebből a megjelenési formából még nem lehetséges közvetlenül előállítani a forráskódot, mert nem létezik közöttük egy-egy értelmű megfelelés.



2.1. ábra. A modell és a kód közötti kapcsolatok

A tervezési diagramok és a forráskód egyaránt arra szolgálnak, hogy egy elvárt rendszert specifikáljanak, de abban különböznek egymástól, hogy ezt különböző absztrakciós szinteken fejezik ki. A forráskód a futtatható kód összes sajátságát kifejezi, hordozza, míg a tervezési modellből jórészt hiányzik a működésnek az a részletessége, ami a kódban megtalálható.

Az ábrán a vízszintes nyilak a közvetlen specifikációs kapcsolatot mutatják, míg a függőleges nyilak az absztrakciós kapcsolatot.

Ebben a felfogásban az UML tervezési modell a forráskódban levő információ absztrakciója. Mint ismeretes, az OO-programok működését maguk az objektumok valósítják meg. Ezek a futás közben létrejönnek, megszűnnek, miközben adatokat dolgoznak fel és kommunikálnak egymással. Az erre vonatkozó modellösszetevő az ábrán az *objektumstruktúra*. Az objektumstruktúra annak az absztrakciója, hogy valójában mi történik akkor, amikor a program futásban, működésben van.

A fentiekből két következtetés vonható le:

- Először is az, hogy az UML nyelven megadott diagramok nem egyszerűen csak képek, hanem konkrét jelentésük van a tekintetben, hogy mit specifikálnak a rendszer működési sajátságaira vonatkozóan. Ebben a megközelítésben a diagramok olyanok, mint a programok, csak azoknál absztraktabb, elvontabb formában.
- Másodszor, az UML nyelv jelölésrendszere jól követi azt a szemléletmódot, amit a programozók alkalmaznak a kódírás folyamán. Ennek megfelelően, az UML és az OO-nyelvek ugyanazokkal a szemantikus alapokkal rendelkeznek, ami lehetővé teszi, hogy az UML-tervből konzisztens programfejlesztést lehessen megvalósítani.

Az OO tervezési nyelvek és programozási nyelvek közös vonása, hogy egyaránt a szoftverműködés meghatározására szolgálnak. Az ún. *objektummodell* az a közös számítási modell, amit az UML és az OO programozási nyelvek osztanak meg egymással. A kétféle nyelv között szoros kapcsolat van, ahol is a nyelvek különböző absztrakciós szinteken fejeznek ki működési vonásokat a programokról.

Az objektummodell nem egy specifikus modell az UML-ben, hanem inkább általános gondolkodási mód az OO-programok struktúrájáról. Olyan koncepciók kerete, amelyek arra használhatók, hogy megmagyarázzák az összes tervezési és programozási tevékenységet. Amint a neve is kifejezi, az objektummodell alapja azoknak az objektumoknak az együttese, amelyek a programot alkotják, amelyek a működését megvalósítják.

Ezekben testesülnek meg a számítógépi program alapvető vonásai, nevezetesen a bennük levő *adatok* és azok *feldolgozási folyamata*.

Egy adott objektum csak egy kis részét valósítja meg a teljes rendszer funkcionalitásának. A rendszer általános működése a különböző objektumok közötti interakció révén valósul meg. Összegezve, az objektummodell a szoftverfejlesztésben azt fejezi ki, hogy úgy tekintjük az OO-programot mint egymással kommunikáló és együttműködő objektumok összességét.

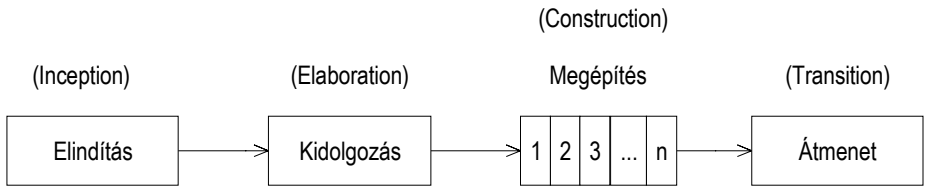
Az objektummodell az alapja az UML tervezési modellnek. Az UML-ben megvalósuló tervezési folyamat eredménye az egymással összeköttetésben levő és egymással kommunikáló *objektumok dinamikus hálózata*. Ezen belül a *statikus modellek* az objektumok között létező összeköttetéseket, azok topológiáját írják le. A *dinamikus modellek* az objektumok között küldött üzeneteket írják le, valamint az üzenetek hatását az objektumokra. Mindezek az OO-programok futási, végrehajtási sajátságait is kifejezik. Mint látni fogjuk, a statikus és dinamikus modellt az UML ezekre a célokra szolgáló diagramjai valósítják meg.

2.3. A fejlesztési folyamat általános menete

2.3.1. A fejlesztési fázisok

Az objektumorientált szoftverek fejlesztéséhez elterjedten alkalmazzák azt a négy fázisra, négy fejlesztési szakaszra bontott modellt, amelyet a 2.2 ábrán mutatunk be. A modell Booch, Rumbaugh és Jacobson munkájának az eredménye, az általuk kidolgozott RUP módszertan is ebből indul ki, és erre is épül. Az egyes fázisok, az eredeti angol elnevezésükkel együtt a következők:

- *Elindítás (Inception).*
- *Kidolgozás (Elaboration).*
- *Megépítés (Construction).*
- *Átmenet (Transition).*



2.2. ábra. Az objektumorientált fejlesztés fázisai

Az ábrán látható teljes folyamat igen fontos jellegzetessége, hogy egyrészt *iteratív,* másrészt pedig egyúttal *inkrementális* is. Mindez azt jelenti, hogy a szoftvert nem egyetlen kész adagban, egyszerre bocsátják ki a projekt végén, hanem e helyett egymás után következő külön fejlesztési darabokban, fejlesztési változatokban. Ekkor az egyes darabok, változatok fejlesztésének menete megismétlődik. Ezáltal a végső termék folytonos, több lépéses kidolgozású részek egymás után megépülő változataiként jön létre. Ezek a motívumok elsősorban a *megépítési* fázisban érvényesülnek, ahol számos ismétlődő, iteratív lépés után áll elő a szoftver. Egy-egy ismétlés, iteráció itt magában foglalja az összes szokásos életciklus-tevékenységet, ideértve az elemzést, tervezést, megvalósítást, valamint a tesztelést. A két motívum (iteráció és inkrementáció) egyébként a folyamat másik három fázisában is érvényre jut.

A végleges változat akkor jön létre, amikor már nincs szükség újabb ismétléses fejlesztés elvégzésére. Az *inkrementalitás* ebben a folyamatban úgy érvényesül, hogy mindegyik iterációban változik, vagyis módosul, bővül, esetleg szűkül a termék. (A folyamat hasonlít például ahhoz, ahogyan Lev Tolsztoj a „Háború és béke" című regényét írta meg. A regényt az író nyolc alkalommal fogalmazta újra, átdolgozva azt elejétől végéig, amíg a maga számára is elfogadhatónak nem találta. Mindez tizenkét évet igényelt.)

Az *elindítási* fázisban az egész projekt üzleti értelmét határozzuk meg, valamint hogy mire terjedjen ki a projekt, mit foglaljon magában. Mindehhez meg kell szerezni a projekt finanszírozójának a jóváhagyását, akinek el kell döntenie, hogy mekkora költséget tud vállalni.

A *kidolgozási* fázisban már részletesebben meghatározzuk a követelményeket, magas hierarchikus szintű analízist és tervezést végzünk, azzal a céllal, hogy definiáljuk a szoftverarchitektúra alapjait, továbbá tervet készítsünk a *megépítés* fázisához.

Az *átmeneti* fázisra olyan teendők maradnak, mint például a bétatesztelés, a teljesítmény hangolása, vagy a felhasználók kiképzése.

Még egyszer hangsúlyozzuk, hogy ebben a teljes folyamatban mindegyik fázis *iteratív* és *inkrementális* jellegű, több lépésre, ismétlésre és változtatásra, módosításra alapozva. Mindez különböző mértékben érvényesül az egyes fázisokban, mégpedig leginkább a *megépítés* során.

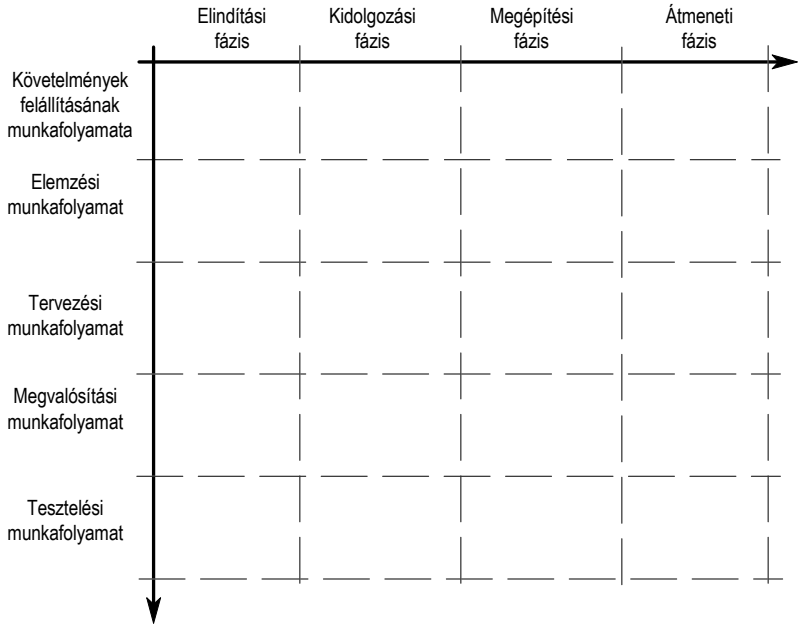
2.3.2. A RUP-modell két dimenziója

Mint ismeretes, a szoftverfejlesztési életciklust leíró klasszikus vízesés-modell öt alapvető fázisból áll:

1. Követelmények felállításának fázisa.
2. Elemzési fázis.
3. Tervezési fázis.
4. Megvalósítási fázis.
5. Tesztelési fázis.

Ez a modell egydimenziós, és egy fentről lefelé haladó tengelyen ábrázolható. Az egyes fázisok között, a menet közben felmerülő módosítási igények következtében, visszacsatolás jöhet létre.

A RUP-modell magában foglalja a vízesésmodellt is, ezáltal kétdimenzióssá válik. A két dimenzió oly módon jön létre, hogy a RUP négy fejlesztési fázisa a vízesésmodell öt fázisával van kombinálva. Ez úgy értendő, hogy minden egyes fázis a vízesés-modell öt fázisa alatt képviselt *munkafolyamatra (work flow)* van felbontva (2.3. ábra). Másként fogalmazva: Mindegyik RUP fejlesztési fázisban öt egymást követő lépésben valósul meg a tervezési eredmény.



2.3. ábra. A kétdimenziós RUP-modell

Az egyes fejlesztési szakaszok résztevékenységekre való bontása a teljes munkafeladat kisebb egységekre való szétválasztását teszi lehetővé. Mindehhez kapcsolódik az iteratív és inkrementatív végrehajtási folyamat, ami a szoftverépítés egészét áttekinthetőbbé, hatékonyabbá és biztonságosabbá teszi.

A RUP-modellről megállapítható, hogy igen jól szolgálja a nagybonyolultságú szoftverek létrehozását, mivel a feladat kisebb részekre való bontását és a részek összehangolt, lépésenkénti kidolgozását teszi lehetővé. Amit még hozzátehetünk: A szoftvertechnológia jelenlegi fejlettségi szintjén, az objektumorientált szférában ez a modell alkalmazható a legelőnyösebben a többi más modellhez képest.

2.3.3. Az egyes fázisok áttekintése

Elindítás

Többféle megvalósítása lehetséges, a projekt méretétől függően. Egy kisebb projekt elindítható akár egy rövidebb megbeszéléssel is. Például ezzel a felvetéssel kezdve: „Tegyük ki a szolgáltatásaink listáját a webre.” Egy ilyen jellegű indításra elegendő lehet egy-két nap is. Ezzel szemben egy nagy projektnél elképzelhető, hogy egy részletes kivitelezhetőségi, megvalósíthatósági tanulmány elkészítésére van szükség. Ez a tanulmány akár több hónapon keresztül is készülhet, több ember munkája révén.

Az elindításnál igen fontos a projekt üzleti oldalának az alapos elemzése. Ekkor meghatározandó, hogy egyrészt mekkora költségvonzata van a projektnek, másrészt pedig hogy mekkora bevétel, ill. haszon várható belőle. A projekt méretének, költségeinek megállapításában döntő tényező a ráfordítandó emberhónap, valamint az átfutási idő. Mindehhez járul még a felhasználandó hardver és szoftver eszközök költsége is. Az így meghatározott költségeket, valamint az alapvető fejlesztési célokat részletesen egyeztetni kell a finanszírozóval. A finanszírozó egyaránt lehet az a cég, ahol a fejlesztés megvalósul, vagy pedig egy külső megbízó, amely megrendeli a fejlesztést a maga számára.

Ebben a fázisban döntést lehet hozni arról is, hogy szükséges lesz-e megvizsgálni a projekt addigi menetét a következő, a kidolgozási fázisban, azzal a céllal, hogy módosítsanak a méretén, ill. a kitűzött célokon.

Kidolgozás

Az elindult projekt első teendői között át kell gondolni a követelményeket, jól meg kell érteni a problémát. Olyan kérdésekre kell választ adni, mint:

Mi a cél? Milyen jellegű szoftverre lesz szükség? Milyen legyen a specifikációja, mit kell tudnia? Hogyan legyen az megvalósítva, hogyan fogjuk felépíteni? Milyen szoftvertechnológiát alkalmazzunk?

Ebben a fázisban szükség van a *kockázatok* előzetes átgondolására, elemzésére. A lehetséges kockázatok ugyanis olyan veszélyeket rejtenek, amelyek nagymértékben félrevihetik a projektet, főleg időben és költségekben, ami lehetőleg elkerülendő.

Martin Fowler, az „UML Distilled" című kiváló szakkönyvében (USA, 1997) a kockázatok négy kategóriáját különbözteti meg:

- *Követelmények kockázata:* A nagy veszély itt abban van, hogy olyan szoftvert tűzünk ki célul, amely az elkészülte után nem fog megfelelni a felhasználónak, mert annak más lett volna a jó. A kidolgozás fázisában erre igen nagy gondot kell fordítani.
- *Technológiai kockázatok: Az itt felmerülő lehetséges kérdések például a következők:*
  - Milyen OO fejlesztési tapasztalattal rendelkeznek a fejlesztők?
  - Megoldható-e a feladat egy webkeresővel, ami egy adatbázishoz kapcsolódik? Jó lesz-e ehhez a C++ nyelv? Vagy legyen a Java?
- *Szaktudási kockázat:* Itt a fő kérdés az, hogy megvan-e a kellő létszám és szaktudás a kitűzött feladat megoldásához?
- *Vállalatpolitikai kockázat:* Azt kell itt elemezni, hogy a fejlesztő cég, vállalat gazdaságpolitikájába beleillik-e projekt. Vannak-e komoly erők a cégnél, akik támogatják, vagy pedig ellenzik azt.

Fowler szerint lehetnek még más egyéb kockázatok is, de azok nem mindig jelentkeznek. Ezzel szemben a fenti négy csaknem minden esetben megvan.

A kidolgozási fázis végső soron arra irányul, hogy a rendszer *architektúráját* hozzuk létre. Ez az architektúra az alábbiakból tevődik össze:

- A követelmények részletes felsorolása.
- A feladat kidolgozásának modellje, ami már a szoftvert alkotó legfontosabb OO-osztályokat is tartalmazza, felsorolva.
- A megvalósítás szoftvertechnológiája, annak részei, és illeszkedésük egymáshoz. Például, egy webes fejlesztés kliens-szerver kapcsolódása, az interfészek megvalósítási módja, a felhasznált programozási nyelvek, a fájlformátumok meghatározása stb.
- Az igénybe veendő hardver elemei, ill. a felhasználandó operációs rendszer.

Természetesen ez a fázis is egy iteratív folyamat, az architektúra is változásokon eshet át. A változások szempontjából nem mindegy, hogy melyik programozási nyelvet használjuk. Például a Java elbírja a jelentős architekturális változtatásokat is. Ezt úgy kell érteni, hogy viszonylag kisebb plusz ráfordítással lehet követni a módosításokat. Ezzel szemben a C++ nyelv már nehezen viseli el az átalakításokat, használata minél stabilabb architektúrát tesz célszerűvé.

Fowler megfigyelései szerint a kidolgozási fázis a teljes projekt mintegy 20%-át viszi el időráfordításban. Ennél többet nem érdemes vele eltölteni. A véget érést a következő momentumok jelzik:

- A fejlesztők már megnyugodtak afelől, hogy jól meg tudják becsülni az emberhónapos ráfordítást, a főbb részfunkciók megvalósítására szétbontva.
- Mindegyik lényeges kockázat meg lett határozva, azonosítva, továbbá az is, hogy miként kell kezelni, leküzdeni ezeket a kockázatokat.

Megépítés

Ez a fázis tipikusan iterációk sorozatából áll, ahol mindegyik iterációs lépés valójában egy mini projekt. A szoftver főbb részfunkcióira egyenként el kell végezni az alábbi tevékenységeket:

- elemzés,
- tervezés,
- kódolás,
- tesztelés,
- integrálás.

Az ismétlések sora akkor zárul le, amikor egy demóverzió adható át a felhasználónak. Ezzel egyidejűleg sor kerülhet a rendszer szintű tesztelés végrehajtására.

A tesztelést és az integrálást nem szabad a fejlesztés végére hagyni. Minél később tesztelünk, annál nagyobb a veszélye annak, hogy sok hibát kell javítani a programokban, igen nagy ráfordítással. A késői tesztelés másik veszélye az, hogy ha a tesztek alapján módosítást kell végrehajtani a programon, az egy előrehaladott fejlesztésnél már költségesebb, mintha korábban oldottuk volna meg a módosításokat. Egyébként általánosan is igaz minden fejlesztésre, hogy minél később derül ki valamilyen tévedés vagy hiba, annál nehezebb a változtatást vagy javítást végrehajtani.

Fowler azt az elvet vallja, hogy a jó tesztelés megköveteli, hogy nagyjából ugyanannyi tesztkódot írjunk, mint amennyi programkódot magában a termékben. Ez az ökölszabály összhangban van azzal az általánosan elterjedt tapasztalattal, miszerint a tesztelés a fejlesztési folyamatban mintegy 50%-os arányban részesül.

Ennek a munkának a kódírással együtt kellene folynia. Mihelyt készen van a kód, elkészítendő a tesztprogramja is. Már a kódolás közben érdemes azon gondolkodni, hogy miként, milyen módon is fogjuk tesztelni az adott programunkat.

A részegységek, modulok tesztelését Fowler szerint maga a fejlesztő végezze, csakúgy, mint az integrációs tesztelést is. Ezzel szemben a teljes rendszer tesztelése már egy független, különálló team feladata kell legyen. Olyan teamé, amely kizárólag tesztelésre van szakosodva. Az ilyen munkában részt vevők számára az a siker, ha minél több hibát tudnak felfedezni, ami a szoftverminőség szempontjából mindenképpen kívánatos hozzáállás.

Mindehhez még annyit fűzünk, hogy az iteratív szakaszoknál is be kell tartani a részfeladatok teljesítésére megszabott határidőt. Ha ez nem sikerülne valamilyen oknál fogva, akkor újra kell egyeztetni a projekt menetét, a finanszírozót is bevonva.

Átmenet

Ez az utolsó fázis. Az a lényeges jellemzője, hogy itt már nem történik valódi, érdemi fejlesztés. A szoftver funkcionalitásán nem változtatunk már, és hibakeresés sem folyik ebben a fázisban. Ezek a tevékenységek lezárultak az előző fejlesztési lépcsőkben. Itt azokat a végső simításokat, javításokat, módosításokat hajtjuk végre, amelyek a szoftver optimálisabb működését eredményezik. Ez az optimalizálás a hatékonyabb, gyorsabb működés elősegítését szolgálja, másrészt pedig a felhasználói kényelem növelését.

Az átmeneti szakasz tehát arra irányul, hogy a nyers végtermékből egy optimalizált, „kicsiszolt” végtermék váljék. Tipikus példa lehet erre az az időszak, ami a béta kibocsátási verzió és a végső kibocsátási termék megjelenése között telik el.

2.4. A fejlesztés végigvitele

2.4.1. Kommunikációs problémák

A szoftverfejlesztésben az a legnagyobb kihívás, hogy egy jól működő rendszert hozzunk létre, olyat, ami kielégíti a felhasználói követelményeket, éspedig ésszerű költségekkel megvalósítva. Ez egy valódi mérnöki feladat.

A feladat megoldását tovább nehezíti az, hogy a fejlesztők szóhasználata, zsargonja sok esetben lényegesen különbözik a felhasználók zsargonjától. Egy kórházi rendszer fejlesztésénél például meg kell küzdeni az orvosi kifejezésekkel, amik ráadásul még csak nem is magyarul vagy angolul vannak. Ugyanilyen gondokkal járhat egy vasúti vezérléseket végző beágyazott szoftver elkészítése is, amihez a vasutasok szakkifejezéseit kell megérteni.

A feladat pontos megértése és a félreértésektől mentes kommunikáció kulcsfontosságú a kifejlesztendő szoftver elfogadhatósága, minősége szempontjából. Ehhez annak a célnak az elérése kapcsolódik, hogy minél jobb, tökéletesebb specifikáció készüljön el, hiszen a specifikáción múlik az egész fejlesztés végeredménye. A tévedések csökkentésére az UML-ben egy alapvető fontosságú segédeszköz áll rendelkezésre: Ez a segédeszköz a *use case,* magyarul *használati eset.* (A tankönyvben általában megmaradunk az angol elnevezés használata mellett, mivel ez az elterjedtebb, és emellett könnyebb a kiejtése is.)

Egy-egy use case nem más, mint a rendszer kívülről látható viselkedésének a leírása, definiálása. Ebben a megjelenítésben fel van tüntetve a felhasználó, valamint a számítógépes rendszernek azok az összetevői, amelyekkel a felhasználó együttműködésben, interakcióban van. Ez a kép felfogható úgy is, mint egy pillanatfelvétel a működés közben. Egy szoftvernek számos use case-e létezhet, attól függően, hogy milyen feladatokat, funkciókat lát el. A használati esetek összessége egy olyan külső képet ad a rendszerről, ami lehetővé teszi a teljes működés áttekintését a felhasználó oldaláról. Ugyanakkor mindez fontos támasz a fejlesztő számára is, aki ezáltal biztosabban megérti, hogy mit kíván a felhasználó.

A use case-ek központi szerepet játszanak az UML nyelv alkalmazása során. Nemcsak a megértést segítik elő, hanem hasznos eszköznek bizonyulnak a projekt-tervezésben is, mivel ezekre lehet alapozni az iteratív fejlesztést. Ez azt jelenti, hogy a felhasználóhoz való visszacsatolásban és az ismétlődő módosításokban is fontos szerepük van, lehet rájuk támasz-

kodni. Mindemellett nem csak a felszínen történő kommunikációhoz adnak segítséget, hanem a belső, mélyebb fejlesztési folyamatokhoz is, amivel a tankönyv későbbi részeiben foglalkozunk.

A használati esetek nem hordoznak információt a megvalósítás módjáról, a rendszer struktúrájáról. A use case-eken alapuló megjelenítés végül is arra alkalmas, hogy a fejlesztőn kívül egyaránt el tudjon rajta igazodni a megrendelő, a felhasználó, a tesztelő, és egyáltalán mindazok, akik nem rendelkeznek ismeretekkel a szoftver konkrét felépítéséről, implementációjáról.

2.4.2. A kidolgozási fázis kockázatai

Ebben az alpontban részletesebben áttekintjük azokat a kockázatokat, amelyek a kidolgozási fázisban jelentkezhetnek. (Ezekről szó esett már korábban, az 1.2.2. alpontban.)

Követelmények kockázata

Ennek a kockázatnak a csökkentésében sokat tud segíteni az UML használata. Itt elsősorban a use case-ekre tudunk hagyatkozni, amelyek a teljes fejlesztési folyamat támaszai is egyben.

A use case-ek különböző terjedelműek, különböző bonyolultságúak lehetnek. A fő dolog az, hogy mindegyik egy funkciót jelöl ki, amit a felhasználó is át tud látni, meg tud érteni, ami fontos is a számára. Ez a fajta segédeszköz hatékonyabbá és eredményesebbé teszi a finanszírozó és a fejlesztő közötti kommunikációt.

A kidolgozási fázisban meg kell találni az összes olyan use case-t, amire a rendszer épülni fog. A gyakorlatban az nem lehetséges, hogy mindegyiket elő tudjuk itt állítani, de a legfontosabbakat, és az összes fontosat viszont meg kell találni. Ehhez arra van szükség, hogy interjúkat ütemezzünk be a felhasználóval, azzal a céllal, hogy use case-eket gyűjtsünk, amikhez szöveges kiegészítéseket is készítünk. Ezek nem kell, hogy túl részletesek legyenek, néhány bekezdésből álló rövid leírás elegendő hozzájuk. A szöveg a lényeget tartalmazó legyen, olyan, ami alkalmas arra, hogy a felhasználó egyértelműen meg tudja érteni, és láthassa belőle az alapelképzelést.

Az UML kidolgozói által javasolt módszertan nem azt igényli, hogy a teljes rendszert egy ütemben alakítsuk ki. Fontos, hogy meglegyenek a kulcsfontosságú szoftverosztályaink és use case-eink. A rendszert ezekre fogjuk alapozni, és további kibővítéseket tudunk majd hozzájuk építeni. A bővítések újabb use case-ek létrehozásával járnak (inkrementális fejlesz-

tés), és ezek mind részei lesznek az iteratív folyamatnak is. A rendszer tehát lépésenként bővül, esetleges visszalépésekkel, javításokkal, módosításokkal együtt.

A use case-eken kívül itt jól felhasználhatók az UML más eszközei, nevezetesen az *osztálydiagramok (class diagram),* az *aktivitási diagramok (activity diagram),* a *szekvenciadiagramok (sequence diagram),* valamint az *állapotdiagramok (state diagram).* Ezek mindegyike a tervezési modell létrehozását szolgálja, miközben szoros kapcsolatban állnak a use case-ekkel és egymással is. (Ezekkel a könyv későbbi fejezeteiben részletesen fogunk foglalkozni.)

A diagramokat egy külön erre a célra létrehozott modelltervező team dolgozza ki. A team javasolt létszáma 2-4 fő. Annak érdekében, hogy ne tudjanak elveszni a részletekben, szoros határidőt kell nekik szabni.

Technológiai kockázatok

A technológiai kockázatok csökkentése érdekében ún. *prototípust* érdemes készíteni. A prototípus ebben az esetben egy megvalósított kísérleti program, amely egy részfeladat megoldásának kipróbálására alkalmas. Ezzel azt lehet vizsgálni, elemezni, hogy egy kisebb szoftverrészlet milyen működést mutat.

Szemléltető példaként vegyünk egy olyan fejlesztést, amiben a C++ nyelvet szándékozzuk használni, egy relációs adatbázishoz kapcsolódva. A prototípus elkészítése a következő lépésekből állhat:

- Vesszük a C++ kompájlert és a hozzá tartozó fejlesztési segédeszközöket *(tool-okat).*
- A meglevő szoftvermodell (tervezési modell) valamelyik kiválasztott egyszerűbb részét leprogramozzuk. Eközben azt is nézzük, hogy miként tudunk boldogulni a felhasznált segédeszközökkel, azok hogyan funkcionálnak.
- Építsük fel az adatbázist, és kössük azt össze a C++ kóddal.
- Próbáljunk ki több különböző segédeszközt. Tapasztaljuk ki, hogy melyikkel a kényelmesebb, hatékonyabb fejleszteni. Eközben szokjunk hozzá a tool-ok használatához, gyakoroljuk a lehetséges fogásokat.

Természetesen azt is szem előtt kell tartani, hogy a legnagyobb technológiai kockázat nem magában egy szoftverkomponensben rejlik, hanem abban, hogy az egyes komponensek hogyan illeszkednek egymáshoz, miként működnek együtt. (Ez a tény persze igaz egy fejlesztő team tagjaira is, és a team egészére nézve is.) Az előző példára vonatkoztatva: Lehet jól ismerni

a C++ nyelvet, lehet járatosnak lenni a relációs adatbázis kezelésében, de ettől még sok nehézséggel járhat a megírt program és az adatbázis összeillesztése, együttes működtetése. Ezért is célszerű a fejlesztésben arra törekedni, hogy a komponensek minél előbb össze legyenek illesztve, hogy minél előbb kipróbálhassuk az együttes működésüket. Mint tudjuk, ennek az az értelme, hogy a korábbi változtatások mindig kisebb veszteséggel járnak, mint a későbbiek.

Igen erősen szemmel kell tartani azokat a fejlesztési részterületeket, amelyeket később nehéz lesz megváltoztatni. Ugyanakkor törekedni kell a tervezés során arra is, hogy az egyes részelemeket minél könnyebben lehessen megváltoztatni.

Ehhez ilyen kérdéseket érdemes feltenni magunk számára:

- Mi lesz akkor, ha egy technológiai eszköz nem válik be?
- Mi lesz akkor, ha két komponens nem illik össze?
- Mi a valószínűsége annak, hogy valami félresiklik a fejlesztésben, valami baj lép fel? Mi legyen a megoldás, ha ilyesmi történik?

Ebben a folyamatban jól fel tudjuk használni a különböző UML-diagramokat. Például, az osztálydiagramok és az *együttműködési diagramok (interaction diagram)* igen hasznosak lehetnek abban, hogy megmutatják, hogyan kommunikálnak egymással a komponensek.

Szaktudási kockázatok

Sokan értékelik utólag úgy az OO-projektjüket, hogy az volt a gond, hogy kevés kiképzést kaptak a résztvevők. Sok helyen sajnálják a pénzt a kiképzésre, és utólag azért fizetnek többet, mert a projekt jóval tovább tartott ahhoz képest, hogy nagyobb felkészültséggel vágtak volna bele.

A kiképzés akkor ér valamit, ha az oktató a fejlesztésben járatos szakember, aki el tudja mondani, hogy mire kell ügyelni, milyen hibákat kell elkerülni, és hogyan lehet bizonyos fontos feladatokat megoldani. (Az a jó, ha olyan tanítja, aki csinálja is.)

Fowler véleménye szerint az objektumorientált fejlesztést nem lehet tanfolyamokon, előadásokon megtanulni. A legjobb megoldás az, ha a tudást egy olyan személytől sajátítjuk el, aki úgy oktat, hogy közben maga is dolgozik a projektben, mégpedig huzamosabb ideig. Ezalatt megmutatja, miként kell csinálni egyes dolgokat, tippeket, tanácsokat ad, figyeli, hogyan oldanak meg problémákat a résztvevők, és közben szükség szerint kisebb tanfolyamokat is tart.

Egy ilyen projektoktató személy *(mentor)* legalább az elején maga is tagja a fejlesztő teamnek, és részt vesz a specifikáció kidolgozásában. Az idő előrehaladtával egyre inkább csak ellenőrzi, követi a fejlesztést, miközben maga egyre kevesebbet dolgozik a fejlesztésen. Legjobb az, ha a vége felé már feleslegessé is tud válni.

Érdemes a projektekhez ilyen külső szakértőket befogadni, és addig támaszkodni rájuk, amíg szükséges. Persze ez komoly költségnövelő tényező, de a minőség érdekében sokszor megéri ezzel a lehetőséggel élni.

Az is megoldás lehet, ha csak arra alkalmaznak egy külső szakértőt, hogy időszakosan vizsgáljon át egy projektet (pl. havonta), és adjon véleményt róla, valamint tanácsokat a továbbfolytatásra. Ugyancsak elterjedt választás még az is, amikor a szakértő maga nem vesz részt a projektben semmilyen formában, viszont azt ellenőrzi, hogy minden az előírt fejlesztési technológia szerint történik-e. Ezt *projekt-minőségbiztosításnak* nevezik. Több formája van: a folyamatos követéstől kezdve az időszakos ellenőrzésig.

A szaktudási kockázatok csökkentésének egy másik természetes módja az önképzés. Fowler szerint érdemes kéthavonta elolvasni egy jó fejlesztési szakkönyvet, és ezt állandóan folytatni, annak érdekében, hogy valaki a szakmában tudjon maradni. A menedzsmentnek erre is pénzt kell áldozni, és a munkatársak számára időt biztosítani az önképzéshez.

Vállalatpolitikai kockázatok

Ez elsősorban gazdaságpolitikai, piaci jellegű terület, ahol a jelentkező kockázatok döntő mértékben nem műszaki természetűek. Ezen a téren minden a vállalti menedzsment üzletpolitikájától, üzleti stratégiájától függ. Az idevonatkozó problémák és megoldási módjuk kívül esnek ennek a tankönyvnek a keretein.

2.4.3. Az alaparchitektúra összetétele

A kidolgozási fázis eredménye abban áll, hogy létrehozzuk a szoftver *alapархitektúráját*. Ez az architektúra a következő elemekből tevődik össze:

- A use case-ek halmaza, ami kifejezi a szoftverre vonatkozó követelményeket.
- A tervezési modell, amely a szoftver működésének alapjait tükrözi, és a legfontosabb osztályok is leszármaztathatók belőle.
- A technológiai platform, amely megadja, hogy milyen szoftvertechnológiai úton, milyen eszközökkel történik a megvalósítása a szoftvernek. Ide tartoznak a számítógépes eszközök, az operációs rendszer, a prog-

ramozási nyelvek, fejlesztési környezet, tesztelési környezet, felhasználandó tool-ok stb.

2.4.4. A megépítési fázis teendői

A kidolgozást követő *megépítési* fázisban iterációk, megoldási ismétlések sorozatára van szükség, amelyekben use case-ek vesznek részt. Ez egy tervkészítési folyamat, ami arra irányul, hogy mindegyik iterációhoz egy use case-t rendelünk. A tervkészítésben itt meg kell becsülnünk, hogy egy-egy use case-hez tartozó iterációs szakasz mennyi ideig fog tartani, hány emberhétből fog állni.

A becslésnek a következő tevékenységekre kell kiterjedni:

- elemzés,
- programtervezés,
- programkódolás,
- modultesztelés,
- modulok integrálása teszteléssel,
- dokumentálás.

A szoftverrendszer megépítése ebben a fázisban iterációk sorozatán keresztül megy végbe, ahol mindegyik iteráció önmagában egy külön mini projekt. Ezek a projektek egyben inkrementálisak is, ami abban nyilvánul meg, hogy egy iteráció az őt megelőző iteráció use case-ére épül rá, azt bővíti, inkrementálja.

A programkódolás önmagában véve is iteratív folyamat. Az egyszer már megírt kódot, annak érdekében, hogy egyre jobb legyen a program, szükséges lehet újra írni, módosítani. A kód módosítása esetenként komoly problémákkal járhat. Itt a következőkről van szó:

- Adva van egy program, amelyhez a fejlesztés alatt új funkciókat kell adni, mert ez vetődött fel. Ha egy eredetileg jól megtervezett és jól megírt kódhoz utólag teszünk újabb funkciót, akkor az eredeti kód „felborulhat”, az egész elveszítheti kiindulási rendezettségét. A programban nem volt benne az új funkció, nem volt beleolvasztva, ezért az új hozzáírás révén nem tud szervesen, összehangoltan illeszkedni a régi programhoz. Ez azért lehet így, mert az esetek többségében az új funkciót egyszerűen csak hozzáírják a régi programhoz, annak a tetejére vagy az aljára ültetik. Ez így nem igazán jó megoldás. A program túl bonyolult lesz, és nehézkesen fog működni.

- Elvileg egy teljesen jónak tűnő megoldás lenne az, ha a programot újraterveznénk, és ezután az új terv alapján újra megírnánk az egész kódot. Ez így rendben is volna, viszont túl nagy ráfordítással járna. Az új programban új problémák keletkeznének és új hibák lennének. Ugyanakkor azonban az is lehetséges, hogy az újratervezés hosszabb távon meg fogja érni, mert a végül összeállított kód egy kiegyensúlyozott, jól összehangolt működést fog eredményezni. A jól tervezett, áttekinthető programot tesztelni is könnyebb.

Mivel két irányban lehet választani, ebben a helyzetben egy mérnöki-gazdasági döntés elé nézünk, ahol az optimális kompromisszumot kell megtalálnunk. Az egyik irány az, hogy újratervezzük, és úgy írjuk meg, a másik irány az, hogy a meglevőt írjuk át. A döntésben természetesen a kitűzött határidőt is szem előtt kell tartani. A szorító határidő természetesen az újratervezés ellen szól.

2.4.5. Az átrendezés elve

A meglevő kód átírására is dolgoztak ki olyan irányelveket, amelyek rendezett, áttekinthető módosításokat tudnak eredményezni. Az itt alkalmazott megoldást *átrendezésnek* (az angolban *refactoring*) nevezzük.

Az átrendezésben olyan technikákat, megoldásokat alkalmaznak, amelyekkel nem változtatják meg a program funkcionalitását, hanem csak a belső struktúráját. A struktúraváltoztatással azt érjük el, hogy könnyebb lesz áttekinteni a programot, ezáltal könnyebb lesz tovább dolgozni vele.

Az átrendezési lépések apróbb változtatásokból állnak. Például: egy objektum metódusának átnevezése, egy adatmező átmozgatása egyik osztályból a másikba, vagy két hasonló metódust egy közös osztályban összevonva megjeleníteni. Jóllehet ezek önmagukban véve kis lépések, mégis azzal járnak, hogy nagyon sokat tudnak javítani a programon. Mindezek után már biztonságosabban és simábban lehet a programhoz hozzáilleszteni az új funkciókat.

Az átrendezés végrehajtását a következő elvek, fogások tudják megkönnyíteni:

- Szigorúan szét kell választani az átrendezést és a funkcióbővítést, vagyis a kettőt nem szabad ugyanabban az időben, együtt végezni. A két tevékenységet egymással váltogatva érdemes kivitelezni: átrendezés – funkcióbővítés, átrendezés – funkcióbővítés, …

- Legyen készenlétben az átrendezés előtt a program tesztkészlete. A teszteket minél gyakrabban futtassuk le a módosítások során. Ezáltal hamar kiderül, hogy nem lett-e elrontva valami a belenyúlásokkal.
- Jól körülhatárolt, jól megfogható kis módosításokat hajtsunk végre, és mindegyik után rögtön teszteljünk is.

Mikor szükséges, ill. célszerű az átrendezés?

- Ha új funkciót adunk a programhoz, és nehézséget okoz a régi kódhoz igazodni. Ekkor a régi kódot át kell rendezni.
- Ha nehézséget okoz a régi kód megértése. Az átrendezés elvégzésével és a tesztelés végrehajtásával mélyebb betekintést nyerünk a kódba, ami a program további felhasználása és fejlesztése szempontjából lesz hasznos.

Előfordulhat, hogy olyan kódot kell átrendezni, amit nem mi írtunk, hanem valaki más. Ilyenkor, ha még elérhető az eredeti szerző, akkor érdemes vele konzultálni arról, hogy miként is gondolkodott a kód megírásakor. A legjobb az, ha vele együtt, közösen nézzük át a kódot, és azután látunk neki az átrendezésnek.

3. Use case modellezés

A valós üzleti folyamatok végrehajtását támogató szoftverrendszerek a folyamatok komplexitásából adódóan nagy bonyolultságú rendszerek. Fejlesztésüknél a megrendelő oldaláról, a felhasználói oldalról kell kiindulni. A felhasználó az, aki részletesen ismeri az elvégzendő feladatokat, ismeri a szervezet működését, azt a környezetet, amelyikben az alkalmazás működni fog. Ezen információk és ismeretek birtokában képes megfogalmazni azokat a célokat, amiket a szoftverrendszernek teljesíteni kell. Másrészt ő az, aki az alkalmazást működteti, használja.

Azokat a folyamatokat, amelyeket a különböző szervezetek valamilyen szoftveralkalmazással kívánnak támogatni, a RUP terminológia üzleti folyamatként definiálja. A számítógépes támogatást igénylő üzleti folyamatok leírására, az üzleti folyamatokat érintő elemek feltárására a RUP módszertan *üzleti modellezés (Business Modeling)* szakaszában kerül sor.

Az üzleti modellezés a rendszerfejlesztési folyamatnak az a szakasza, amelyik még nem a szoftverfejlesztésre koncentrál. Az üzleti modellezés során felállított modellek célja, hogy a felhasználók és a fejlesztők számára egységes kép alakuljon ki a szervezetről, a szervezetben zajló folyamatokról, arról a környezetről, amelyikben a szoftverrendszer működni fog. Ennek a munkaszakasznak az eredményeként elkészül az a részletes üzleti dokumentum, amelyik összefoglalja a felhasználó igényeit, elképzeléseit, pontosan leírja a szervezetben lejátszódó folyamatokat, amelyek számítógépes támogatást igényelnek. A rendszerfejlesztés során ugyanis nemcsak szoftverrendszert tervezünk, a fejlesztés teljes folyamatába beletartozik az a szervezet, amely számára a rendszert fejlesztjük, és azoknak az üzleti folyamatoknak a vizsgálata, amelyek számítógépes támogatását akarjuk megvalósítani. Az üzleti dokumentum részletezi a fejlesztés szempontjából releváns üzleti folyamatokat, a folyamatok más folyamatokkal való kapcsolódását, pontosan leírja a folyamatok során elvégzendő tevékenységeket, meghatározza a folyamatok bemeneteit és kimeneteit, ismerteti a folyamatokban érintett szereplők (üzleti aktorok, üzleti entitások stb.) körét.

A rendszerfejlesztési folyamat során a tényleges szoftverfejlesztési munka a követelményspecifikáció feladataival kezdődik. Ebben a munkaszakaszban az *MDA (Model Driven Architecture – Modell-vezérelt Architektúra)* szabvány által definiált, a fejlesztendő rendszer struktúráját leíró modellek közül a *PIM (Platform Independent Model – Platformfüggetlen Modell)* modell kialakításához kezdünk hozzá.

A tankönyvben a szoftverfejlesztési lépések közül a követelményspecifikáció során végrehajtandó tevékenységeket konkrét gyakorlati példákon keresztül szemléltetjük. A példa teljes mélységű kidolgozására a tankönyv keretein belül nincs lehetőség, azonban a feladatok kidolgozásakor próbáltunk arra koncentrálni, hogy a követelményspecifikáció munkafolyamat, a munkafolyamat során végrehajtandó tevékenységek, a végrehajtás menetének logikája egyértelmű és könnyen követhető legyen. A use case modellezés fejezetben a mintapéldához kapcsolódó szövegrészek kezdetét és végét egy vízszintes vonal jelzi, a leírásokat, magyarázó szövegeket a normál szövegtől való megkülönböztetés érdekében dőlt betűvel írjuk.

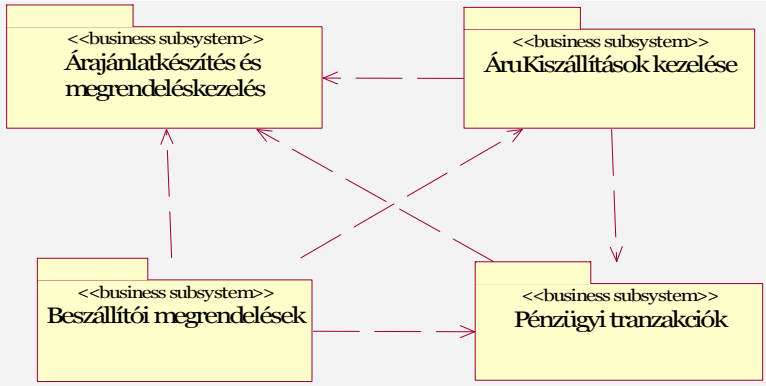
Mintapélda[1]

A SWEProducts cég különböző termékek, áruk forgalmazásával foglalkozik. A cég a termékek értékesítése mellett további szolgáltatásokkal is rendelkezésre áll ügyfelei részére. Árajánlatot készít, vásárlás esetén biztosítja a termékek kiszállítását, átvállalva a kiszállítás költségeit. A cég ügyfélcentrikus szellemben működik, amit mi sem bizonyít jobban, mint az Interneten is elérhető szolgáltatása, az Internetes árajánlat készítésének lehetősége. Az Internetes szolgáltatás kiterjesztésével olyan ügyfeleket céloznak meg, akik pontosan meg tudják adni a termékek megvásárlásához szükséges adatokat. Interneten keresztül történő árajánlat szolgáltatás igénybe vétele esetén pontosan ismerni kell a termékek paramétereit (pl. típus), és pontosan meg kell tudni adni a kiválasztott termékből kívánt mennyiséget (darabszámot).

Az árajánlatokkal, a megrendelések kezesével kapcsolatos feladatokat a cég ügyfélszolgálati munkatársai látják el.

A cégnek az árajánlatok összeállítása, az értékesítés mellett el kell látni a termékbeszerzéssel és a kiszállítással kapcsolatos feladatokat.

[1] A mintapélda kidolgozásakor nem határoztuk meg, hogy a cég konkrétan milyen területen folytatja tevékenységét. Azoknak viszont, akik a gyakorlat példáján könnyebben értik meg az elméleti ismereteket, javasoljuk, gondoljanak egy olyan cégre, amelyik különféle alkatrészek forgalmazásával foglalkozik.

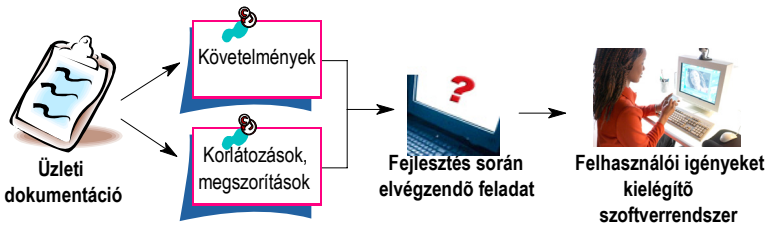


Üzleti folyamatokat leíró csomagok
és a közöttük fennálló függőségi kapcsolatok – részlet

Az üzleti modellezés során összegyűjtött információk rögzítésére is használhatjuk az UML modelleket. A fenti ábrán UML csomagokat láthatunk, amelyek a cég által végzett üzleti folyamatokat modellezik. A *csomag* az az eszköz az UML-ben, amibe a logikailag összetartozó modellelemeket foghatjuk össze, különíthetjük el. A folyamatok közötti kapcsolatot függőségi viszonyt[2] (dependency) reprezentáló szaggatott nyilak jelölik.

3.1. A követelményelemzés szerepe a fejlesztésben

A szoftverfejlesztési folyamat első lépéseként azt kell meghatározni, mi a szoftverfejlesztés célja. A 3.1. ábra azt hivatott szemléltetni, hogy a szoftverfejlesztési tevékenység célja nem más, mint a követelményeknek, a korlátozásoknak, a megszorításoknak (anyagi, erőforrásbeli, emberi stb.) megfelelő szoftverrendszer megtervezése, megvalósítása, üzembe helyezése.



3.1. ábra. A szoftverfejlesztés összetevői

[2] A függőségi viszonyt a 4.3.7. alpont ismerteti.

A kész szoftvertermék kifejlesztéséhez szükséges feladatok végrehajtásához, pontos specifikálásához a felhasználó igényeit és az üzleti folyamatok működését összefoglaló üzleti dokumentumból kell kiindulni. A részletes üzleti dokumentum alapján kell meghatározni:

- A megvalósítandó szoftverfunkciók halmazát, céljait és indokait;
- A tervezett rendszer működésére vonatkozó előre látható korlátozásokat, amelyeket például a megbízó pénzügyi háttere, pillanatnyi elképzelései támasztanak;
- Azokat a feltételeket, korlátozásokat, amelyeknek meg kell felelnie a fejlesztendő rendszernek ahhoz, hogy késznek mondhassuk és átadhassuk.

A fejlesztés kezdetén továbbá egyeztetni kell a szakkifejezések használatát a félreértések elkerülése érdekében.

3.1.1. Felhasználói követelmények

A felhasználónak a szoftverrendszerrel szembeni igényei, elvárásai, azaz a felhasználói célok ún. felhasználói követelmények formájában fogalmazódnak meg. A *felhasználói követelmények* a tervezett szoftverrendszer külső környezetét, viselkedését a felhasználó fogalmaival írják le. A felhasználói követelmények leírásának különböző módszerei ismertek. A leírást készíthetjük természetes nyelven, intuitív diagramok rajzolásával, használhatók elterjedt ábrázolási technikák pl. folyamatábrák, készíthetünk áttekintő, a megértést segítő táblázatokat.

A felhasználók, mint megrendelők általában a szoftverrendszernek csak az alkalmazás felhasználói oldali képét (felületét), nézetét tudják jól leírni. Meg tudják adni, le tudják írni a szervezetnek, ill. a szervezet működésének azt a területét/területeit (alrendszer), azt a szervezeti környezetet, amelyikben majd a fejlesztendő szoftveralkalmazás működni fog. Megadják a rendszertől elvárt szolgáltatásokat (szoftverfunkciók, amiket a szoftver végrehajt), és azokat a feltételeket (megszorításokat), amelyeket a rendszer fejlesztése és majdani működése során be kell tartani. Vannak elképzeléseik az alkalmazás használatáról, az alkalmazás bemeneteiről és a tőle elvárt kimenetekről (pl. listák, bizonylatok formája). A fejlesztő feladata lesz, hogy a rendszer viselkedését ezen információk alapján pontosan definiálja, modellezze.

A rendszer modellezéséhez, leírásához a fejlesztőnek több feladatot kell megoldania. A felhasználói követelményeket (ún. magas szintű célok)

kategorizálni kell[3], közöttük prioritási sorrendet kell kialakítani, majd a felállított fontossági sorrend alapján szükséges mélységben részletesen ki kell dolgozni azokat. Csak a pontosan ismert és részletezett felhasználói célok ismeretében van lehetőség a szoftverrendszertől elvárt konkrét szolgáltatások meghatározására. A szoftverfejlesztésnek ezen szakaszát követelményelemzésnek (követelményspecifikációnak) nevezzük.

Az alkalmazástól elvárt szolgáltatások, szoftverfunkciók a követelményspecifikációban *funkcionális* és *nem funkcionális követelmények* formájában realizálódnak.

3.1.2. Funkcionális követelmények

A *funkcionális követelmények* a szoftverrendszer működésére, a tervezett rendszer szolgáltatásaira, a tényleges funkcionalitásra vonatkozó leírások, amik a szoftverrendszert kívülről nézve, a felhasználó szemszögéből írják le. Legtöbb esetben nem a felhasználó valódi céljait (magas szintű felhasználói célok – felhasználói követelmények) fogalmazzák meg, hanem pontosan azokat a szolgáltatásokat, amiket a felhasználó igénybe kíván venni, azokat a tevékenységeket (funkciók), amiket a felhasználó a rendszerrel el akar végezni, végre kíván hajtani.

A funkcionális követelmények továbbá leírják, hogy a rendszert ért hatásokra, eseményekre a szoftveralkalmazásnak hogyan kell reagálni, a megfogalmazott feltételek, a megadott megszorítások függvényében a rendszernek milyen alternatív végrehajtást kell biztosítani, a bekövetkezett külső események hatására milyen más funkciókat kell aktivizálni. A fejlesztésnek ebben a szakaszában a szoftverrendszer belső struktúráját, belső működését ún. „fekete doboz”-nak tekintjük. Csak arra koncentrálunk, hogy meghúzzuk a fejlesztendő rendszer határát, pontosan definiáljuk, hogy a fejlesztendő szoftverrendszernek milyen funkciókat kell megvalósítani, ill. mely funkciókat nem kell támogatni.

A felhasználói célokat a rendszer a követelményspecifikációban definiált szoftverfunkciók megvalósításával fogja teljesíteni. A szoftverfunkciók leírását, a rájuk vonatkozó funkcionális követelményeket UML modellezés esetén *use case*-ek formájában modellezzük, írjuk le. A RUP modelljének filozófiáját követve a részletes kidolgozási fázis végére minden felhaszná-

[3] A követelmények kategorizálásnak és minősítésének számos hatékony módszere létezik. A szakirodalom a Software Engineering Institute (SEI) és az ISO által kidolgozott módszereket javasolja alkalmazni. Az egységesített módszertan a követelmények csoportosítását a FURPS+ modell alapján végzi.

lói célhoz tartozni kell a funkcionális követelmények egy halmazának, de legalább egy use case-nek.

A fejlesztés során vannak szituációk, amikor a felhasználói célok és a funkcionális követelmények nem határolhatók el tisztán, egzakt módon. Habár az UML nem tartalmaz eszközt a felhasználói célok és a funkcionális követelmények konzekvens megkülönböztetésére, kezelésére, a két fogalom egyedi értelmezésére vonatkozóan, a fejlesztés hatékony menete érdekében viszont, azokat szükséges tudatosan külön kezelni. A felhasználói célok és a funkcionális követelmények elhatárolásának érdekében az alábbi pontokat érdemes figyelembe venni:

- elsőként a felhasználói célokat definiáljuk, majd sorra kell venni azokat a use case-eket, amelyek azokat teljesítik,
- ahol a funkcionális követelmények és a felhasználói célok nem azonos értelmezést kapnak, azokat tudatosan elkülönítve kezeljük,
- a felhasználói célok az alternatív utak átgondolására adnak lehetőséget, amellyel a felhasználói célok teljesíthetők,
- a funkcionális követelményeket elemzési, tervezési célokra használjuk,
- a részletes kidolgozási fázis végére minden felhasználói célhoz tartozni kell a funkcionális követelmények (use case-ek) egy bizonyos készletének.

A működésnek azonban vannak korlátai, feltételei, megszorítások, amelyeket a tervezés során szintén figyelembe kell venni. A működést befolyásoló elemek halmazát *nem funkcionális követelményeknek* nevezzük.

3.1.3. Nem funkcionális követelmények

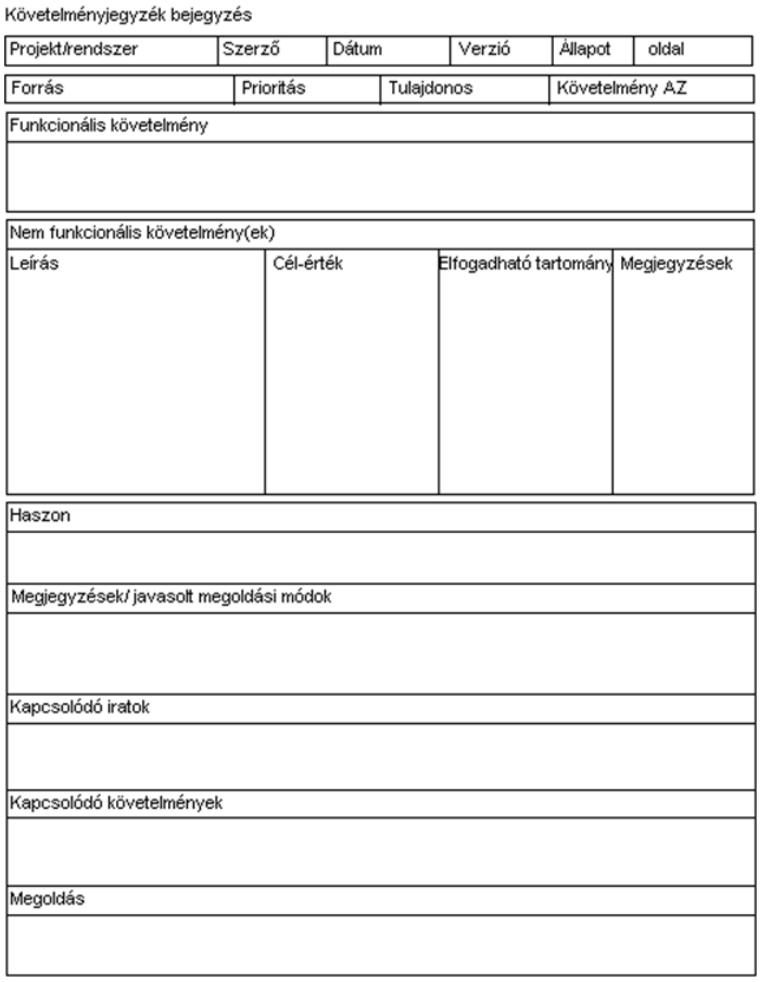
A felhasználói célok összegyűjtése során találkozunk olyan megfogalmazásokkal, amelyek nem konkrétan a szoftverrendszer működésére, a szoftverrendszer által megvalósítandó szolgáltatásokra irányulnak, hanem a szoftverrendszer működését befolyásolják. Ilyenek a működés környezetére vonatkozó kényszerek, feltételek. Ezeket a követelményeket nem funkcionális követelményeknek nevezzük.

A nem funkcionális követelmények vonatkozhatnak a rendszer egészére, vagy egy részére, vagy akár egy konkrét funkcionális követelményre. A nem funkcionális követelmények halmazába tartoznak olyan megkötések, feltételek, megszorítások, mint például:

- a szoftverrendszer hardver- és szoftverkörnyezete,
- a rendszer válaszideje,

- a rendszer megbízhatósága,
- biztonsági szempontból a mentések gyakorisága,
- rendszerösszeomlással kapcsolatos követelmények,
- a szoftverrendszer fejlesztésére vonatkozóan technológiai, technikai és egyéb előírások (a fejlesztési folyamat modelljeinek leírására az UML nyelvet kell használni) stb.

A funkcionális és nem funkcionális követelmények meghatározáshoz és dokumentálásához használhatjuk a *„Követelményjegyzék-bejegyzés" formalapot.* A 3.2. ábrában bemutatott minta-formalap segít a követelmények ponto-


Követelményjegyzék bejegyzés

Projekt/rendszer	Szerző	Dátum	Verzió	Állapot	oldal
Forrás	Prioritás	Tulajdonos	Követelmény AZ		
Funkcionális követelmény					
Nem funkcionális követelmény(ek)					
Leírás	Cél-érték	Elfogadható tartomány	Megjegyzések		
Haszon					
Megjegyzések/ javasolt megoldási módok					
Kapcsolódó iratok					
Kapcsolódó követelmények					
Megoldás					

3.2. ábra. Követelményjegyzék-bejegyzés formalap

sabb, precízebb, ezáltal ellenőrizhetőbb követelményekké való formálásában. A fejlesztési munka korai szakaszában nem biztos, hogy a formalap minden részét ki tudjuk tölteni, a fejlesztés elején lesznek hiányosságok. A fejlesztés előrehaladtával, egyre több információ birtokában a követelményekhez készített formalapot pontosítjuk, szükség esetén módosítjuk egészen addig a pontig, amikor a felhasználók és elemzők egyetértésre nem jutnak abban, hogy a formalap már teljes leírását adja a leendő rendszernek.

„Követelményjegyzék bejegyzés" formalap kitöltését 3.1. táblázatban összefoglalt kitöltési útmutató segíti. A segédlet részletes leírást, magyarázatot ad a formalapon szereplő minden mezőre.

3.1. táblázat. Kitöltési útmutató

Követelmény AZ	egyedi azonosító
Forrás	a követelmény forrása, lehet személy, dokumentum stb.
Prioritás	a követelmény prioritása, a felhasználó szerint, pl. magas/alacsony, vagy kötelező/ javasolt/ választható
Tulajdonos	felhasználó vagy felhasználói szervezet, aki a követelménnyel kapcsolatos egyezkedésért felelős
Funkcionális követelmény	az igényelt lehetőség vagy szolgáltatás leírása
Nem funkcionális követelmény	leírás, lehetőség szerint cél értékkel, elfogadható tartománnyal (minimum, maximum), minősítő megjegyzéssel
Haszon:	a követelmény kielégítéséből származó várható hasznok leírása
Megjegyzések/ javasolt megoldási módok	lehetséges megoldások leírása, általános megjegyzések
Kapcsolódó iratok	hivatkozás kapcsolódó dokumentumokra, mint például felhasználói dokumentumok, projektalapító okirat, adatfolyamábra stb.
Kapcsolódó követelmények	ha különböző követelmények hatnak egymásra, vagy kizárják egymást, akkor a hivatkozást fel kell jegyezni mindkét oldalon, hogy esetleges változtatás esetén fel lehessen mérni a hatást a mások oldalon
Megoldás	a követelmény megoldási módjának feljegyzése, például egy konkrét funkcióleírásra való hivatkozással. Ha egy követelményt nem fogunk kielégíteni, akkor itt kell felírni az okait.

3.1.4. Szakterületi követelmények

A követelmények csoportosításakor, strukturálásakor találkozunk olyan követelményekkel, amelyek nem közvetlenül a felhasználói igényekből származtatva a tervezett szoftverrendszer működésére (funkcionalitására), ill. a működés környezetére vonatkoznak, hanem a leendő rendszer által kiszolgált szakterület követelményeiből adódnak. Ezek a követelmények a tervezett rendszer szakterületén alkalmazott előírásokat, megszorításokat (pl. számítási előírások, az adott szakterület működésére vonatkozó jogszabályok stb.) foglalják össze. A követelményeknek ezt a csoportját *szakterületi követelményeknek (Domain Requirements)* hívjuk.

A szakterületi követelményeket összefoglaló dokumentum tartalmazhat olyan, a szakterület sajátosságaiból adódó előírásokat, amelyek:

- újabb funkcionális és nem funkcionális követelményeket generálnak, vagy
- a már megfogalmazott követelményekre korlátozásokat írnak elő, de előfordulhat az is, hogy
- pontosan meghatározzák egy szoftverfunkció konkrét végrehajtási módját.

A szakterületi követelmények – az alkalmazási terület sajátosságait erősen magukon hordozva – sok esetben hiányosak. A hiányosság fő oka, hogy a szakterület képviselői számára a szakterületen alkalmazott szabályok, előírások olyannyira nyilvánvalóak és egyértelműek (implicitás), hogy fel sem merül a szándék azoknak a fejlesztés során történő megemlítésére, így azok pontos értelmezésére, magyarázatára sem kerül sor. Mindehhez hozzátársul az is, hogy a szakmaterület által használt szaknyelv (szakzsargon) a fejlesztők számára nehezen érthető. Az implicitás és a két szakma (fejlesztő–felhasználó) szaknyelvének különbözősége sok esetben végzetes félreértésekhez vezethet, aminek az eredményeként olyan termék kerülhet kifejlesztésre, ami nem az eredeti céloknak megfelelően fog működni. A fentiek elkerülése érdekében a szakterületre vonatkozó sajátosságokat, előírásokat, szabályokat a fejlesztési munka során megfelelő hangsúllyal kell kezelni, a szakmaspecifikus kifejezéseket pedig – erre a célra rendszeresített – külön fogalomszótárban (értelmező szótár) célszerű dokumentálni.

A követelményeknek funkcionális, nem funkcionális és szakterületi szempontok alapján történő elkülönítése abban játszik nagy szerepet, hogy átgondoljuk

- a tervezett rendszer működését, a működési körülményeket (előírások, szakterületi szabályok stb.)
- milyen funkcionalitást, milyen felületet biztosítson a leendő szoftver a felhasználó felé, milyen belső funkciókat kell teljesítenie ahhoz, hogy a felhasználó igényeit kielégítse, és működés közben milyen előírásokat, szabályokat kell alkalmaznia, betartania.

Felhasználói célok és követelmények leírása

A SWEProducts olyan szoftverrendszer kifejlesztésére adott megbízást, amelyik biztosítja az árajánlatok készítését, a megrendelésekkel (az értékesítéssel) kapcsolatos feladatok ellátását.

Az alábbiakban egy olyan dokumentum részletet mutatunk be, amely összefoglalja, hogy a megrendelő milyen instrukciókat adott az árajánlat kezelési és a megrendelés (értékesítés) szolgáltatás megvalósítására vonatkozóan. A megrendelő által felsorakoztatott igények adják a felhasználói követelmények halmazát.

Az árajánlat kezelési szolgáltatásra vonatkozó felhasználói követelmények:

- Az ügyfelek kérhetnek árajánlatot a cég ügyfélszolgálatánál, amiket a cég munkatársai állítanak össze, de igénybe vehetik az Internetes árajánlat szolgáltatást is.
- A cég megkülönbözteti a vásárlókat. Vannak az „átlagos” vásárlók, és a nagyvásárlók. A cég a nagyvásárlókat kiemelt ügyfélként tartja nyilván:
  - A nagyvásárlóknak a megrendelések pénzügyi teljesítésére 30 napos átfutási idő áll rendelkezésre. Az „átlagosként” nyilvántartott vásárlók a helyszínen fizetnek, készpénzben vagy bankkártyás fizetéssel teljesítenek.
  - A nagyvásárlók a vásárlási volumen függvényében kedvezményt (törzsvásárlói kedvezmény) kapnak. A kedvezmény mértékének meghatározása és jóváhagyása a Kereskedelmi menedzser feladatkörébe tartozik. Abban az esetben viszont, ha az ügyfél három korábbi megrendelését pénzügyileg nem teljesítette, a törzsvásárlói kedvezményét elveszíti.

- Az árajánlatok, az abban feltüntetett árak, kedvezmények az árajánlat készítésének idejétől számított hat hónapig érvényesek.
- Az ügyfeleknek lehetőségük van az interneten készített árajánlatok kinyomtatására.

A megrendelés (értékesítés) szolgáltatásra vonatkozó felhasználói követelmények:

- A fejlesztendő szoftver képes az ügyfélrendelések (vásárlások) kezelésére.
- Vásárolni csak a cég ügyfélszolgálatánál lehet. Interneten csak árajánlat készítésére van lehetőség.
- Amennyiben egy ügyfél rendelkezik érvényes árajánlattal, a rendszerben kell lenni olyan funkciónak, amelyik az árajánlatot vásárlási/megrendelési tranzakcióvá alakítja, konvertálja. Erre a megoldásra azért van szükség, hogy az árajánlatban szereplő adatokat a cég munkatársainak vásárlás esetén ne kelljen újból felvenni, rögzíteni.

A követelmények szerepe meghatározó a majdan működő rendszer minősége szempontjából. Emiatt rendkívül fontos, hogy kiemelt figyelmet fordítsunk a szoftverfejlesztési folyamat korai fázisában a követelmények teljes körű feltárására. Ügyelni kell azonban arra, hogy a követelményeket a fejlesztés kezdetén definiált célok alapján csak a megfelelő mélységig részletezzük ki.

A gyakorlat azt mutatja, hogy a fejlesztés során a megrendelőnek, a felhasználónak újabb elképzelései, igényei merülhetnek fel. A korábban specifikált követelmények így a fejlesztés folyamán változhatnak, módosulhatnak. A modern rendszerfejlesztési módszertanok szerint (így a RUP szerint is) a rendszert fel kell készíteni a követelmények változásának követésére.

A követelményspecifikáció során az igények változásából adódó módosításokat, ill. a felmerült új igényeket első lépésben elemezni kell, majd meg kell vizsgálni, hogy a felmerült változtatások és az új igények milyen hatással lesznek a már felállított követelményrendszerre. A vizsgálat eredményének kiértékelése után lehet csak dönteni a változások megvalósíthatóságáról.

3.2. A use case modell kialakítása

A RUP módszertan szerint a követelményspecifikáció munkaszakasz egyik legfontosabb feladata a fejlesztendő rendszer működését leíró use case modell elkészítése. A use case modellben a rendszer működésére vonatkozó funkcionális követelményeket use case-ek formájában írjuk le, ill. use case-ekhez kapcsolva gyűjtjük össze. A use case modellezés alapjául szolgáló use case-eknek kulcsszerepe van a rendszerfejlesztési folyamat valamennyi fázisában. Végigkísérik a teljes fejlesztési folyamatot. A use case modellt használjuk, finomítjuk tovább az elemzés, a tervezés, az implementáció és végül a tesztelés során. Továbbá a modellben felállított követelményhalmaz az alapja a fejlesztés teljes életciklusában folytatott folyamatos ellenőrzési és kiértékelési feladatoknak.

A követelményspecifikáció végére előálló use case modell a rendszer tervezett funkcionális működését, a rendszer viselkedését írja le a rendszert kívülről, a felhasználó szemszögéből nézve. A modell három építőelemet tartalmaz:

- *use case-ek* (szoftverfunkciók – szolgáltatások), amiket a fejlesztendő szoftverrendszernek meg kell valósítani,
- *aktorok,* akik/amik a rendszer határán kívül vannak, a rendszerrel kapcsolatba kerülnek, hogy a rendszerrel feladatokat (szoftverfunkciók) hajtsanak végre,
- a *kapcsolat* az aktorok és use case-ek közötti viszonyrendszert definiálja.

A témával foglalkozó irodalom a use case modell kialakításakor első lépésben a use case-ek feltárását, egy use case listát javasol elkészíteni. A gyakorlat, a szakmai tapasztalatok azonban azt mutatják, hogy sok esetben elég nehéz a use case-ek listájának meghatározása. Első lépésben könnyebb megtalálni, azono-sítani a rendszerrel kapcsolatba kerülő szereplőket, aktorokat. Ha már ismerjük az aktorokat, azokhoz egyszerűbb hozzátársítani az általuk kezdeményezett, ill. végrehajtott use case-eket (funkciókat). A tankönyv a gyakorlatot követve elsőként az aktorokra vonatkozó ismereteket tekinti át, majd a use case-eket részletezi.

3.2.1. Aktorok

A felhasználóval folytatott beszélgetések, a felhasználói célokat összefoglaló dokumentumok alapján körvonalazódik, hogy mik, vagy kik az érde-

keltek[4] a rendszer határán kívül, amik/akik közvetlenül kapcsolatba kerülnek, kommunikálnak a leendő szoftverrendszerrel. Az érdekelteket a use case modellben aktornak vagy szereplőnek nevezzük. A szereplő elnevezés helyett gyakran találkozhatunk a szerepkör kifejezéssel is. A rendszer felhasználói ugyanis meghatározott feladatkört (szerepet, jogosultságot) betöltve lépnek kapcsolatba a rendszerrel, egy konkrét szerepkört betöltve használhatják a szoftverrendszert és annak szolgáltatásait.

Az *aktor* egy szerep, amit az érdekelt játszik/végrehajt a rendszerrel folytatott interakcióban. A rendszer szereplője, aktora valaki vagy valami a rendszer határán kívül, aki/ami kapcsolatba kerül a rendszerrel, azzal feladatot hajt végre, vagy a rendszernek egy adott szolgáltatását veszi igénybe. Az aktorok nem kizárólag személyek, lehetnek tárgyak, gépek-berendezések, üzleti egységek, vagy a rendszerrel kapcsolatot létesítő valamely külső rendszerek, rendszerkomponensek.

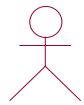
Az aktorok sajátosságai

- Egy felhasználó többfajta szerepet is játszhat/végezhet, többféle szerepkörben lehet; egy szerepkört több felhasználó is betölthet (a rendszerben sohasem lehet két azonos szerepkört betöltő aktor).
- Az aktoroknak a rendszerrel kapcsolatban igényeik vannak, feladatok végrehajtását kezdeményezik, vagy a rendszer által nyújtott funkciók, szolgáltatások megvalósításában vesznek részt.

A feladatok végrehajtását kezdeményező szereplőket *kezdeményező szereplő*nek, a funkció (use case) megvalósításában részt vevőket *résztvevő szereplő*nek hívjuk. Egy use case-t mindig csak egy aktor kezdeményezhet, egy use case megvalósításában viszont több aktor is részt vehet.

- Az UML modellben az aktor nem objektum, hanem egy osztályszerű modellelem, amit az UML classifier minősítéssel azonosít, <<actor>> sztereotípiával[5] ír le;
- Az aktor grafikus szimbóluma egy pálcikaember (lásd 3.3. ábra).

[4] A követelményspecifikáció során azonosított érdekeltek köre (pl. a cég ügyfélszolgálati munkatársai, mint a szoftverrendszer használói) nem feltétlenül, sőt sok esetben abszolút nem egyezik meg az üzleti modellezés során felállított érdekeltek (pl. a cég vezetése, amely tendert írt ki a szoftverrendszer fejlesztésére) listájával.
[5] Sztereotípia: A modellelemek magas szintű tipizálása. Lásd részletesen a 4.3.5. alpontot.



3.3. ábra. Az aktor UML-szimbóluma

Az aktorok megtalálásának módja

A felhasználói célokat összefoglaló dokumentumokból kikeressük a főneveket. A kereséskor a releváns szereplők meghatározása érdekében célszerű arra koncentrálni, hogy:

- Ki/mi használja a rendszert?
- Kik működtetik a rendszert, kik felelnek a rendszer karbantartási és adminisztrációs feladatainak végrehajtásáért?
- Kinek a hatáskörébe tartozik a biztonságkezelés, rendszervédelem?
- Létezik a rendszerben folyamatfigyelés, monitoring folyamat (monitoring process), amelyik hiba esetén újraindítja a rendszert?
- Kommunikál-e az új rendszer más rendszerekkel? Stb.

A rendszer szereplőinek specifikálásra vonatkozó előírások

- a rendszerrel közvetlenül kapcsolatba kerülő, a rendszert használó érdekelteket (személyek, dolgok, más rendszerek, rendszerkomponensek) a feladatkörre, szerepkörre utaló névvel kell ellátni, azonosítani,
- az aktorok neve egy tetszőleges karaktersor,
- az aktor nevének megválasztásakor ügyeljünk arra, hogy az aktor neve azonosítsa a use case-t kezdeményező, vagy a use case megvalósításában részt vevő szereplőt,
- az aktor nevét a szimbólum alá írjuk,
- a specifikációban röviden meg kell adni, hogy az aktor mit vár el a tervezett szoftverrendszertől, mi a felelőssége.

Az aktorok azonosítása

A feladat megfogalmazásából azonnal látható, hogy a rendszert potenciálisan az ügyfelek és a cég munkatársai fogják használni. A rendszernek biztosítani kell, hogy az ügyfelek interneten állíthassanak össze árajánlatokat. A munkatársak az árajánlat-készítés mellett az ügyfélnyilvántartással, a megrendelésekkel (értékesítéssel) kapcsolatos feladatokat látják el. A kiemelt nagyvásárlók részére a törzsvásárlói kedvezmények kezelése a cég menedzserének felelősségi körébe tartozik. A számlák egy külön Számlázási modulban készülnek. Ez azt jelenti, hogy meg kell oldani az új rendszernek a Számlázási modullal való együttműködését, kommunikációját. A korábban meghatározott felhasználói követelmények, és a fenti logikai levezetés alapján a rendszerben négy aktort definiálhatunk: a Kereskedő, az Ügyfél, a Kereskedelmi menedzser és a Számlázó rendszer aktorokat.

A rendszer aktorainak, és azok elvárásainak ismeretében már relatíve könnyebb meghatározni a use case-eket.

3.2.2. Use case-ek

A use case-ek a fejlesztendő szoftverrendszertől a felhasználó által elvárt, megkövetelt viselkedés(ek)t, a rendszer által nyújtott szolgáltatásokat írják le. A use case fogalmának pontos definiálására az irodalomban számos terminológia létezik:

- Feladatok, funkciók kisebb vagy nagyobb egységeinek specifikálására szolgáló grafikus ábrázolási technika – a use case-ek valójában a rendszer funkcionalitását, viselkedését fejezik ki a rendszert kívülről szemlélve.
- A rendszerrel kapcsolatban feltárt use case-ek megadják a fejlesztendő rendszer külső képét.
- A rendszer kívülről látható funkciói, ún. kapcsolódási pontok a szoftverrendszert használók és a szoftverrendszer között.
- Tevékenységek specifikációja.

A *use case* a felhasználó és a számítógépes rendszer közötti interakció definiálására szolgáló modellelem. Tipikusan a szoftver és a felhasználó (aktor) között lezajló kommunikáció, üzenetváltás lépéseit írja le a felhasználó szemszögéből. Egy use case pontosan azt határozza meg, hogy a felhasználó (aktor) MIT akar a szoftverrel végrehajtani, milyen célt kíván megvalósítani, ugyanakkor nem tér ki a megvalósítás, a HOGYAN részleteire.

A use case-ek definiálásához hozzátartozik a use case nevének megadása, valamint egy leírás elkészítése, amely tartalmazza a use case-ben definiált működés végrehajtása során elvégzett lépéseket, megvalósított műveleteket. Ezt a leírást általában forgatókönyvnek nevezzük. A rendszerfejlesztés során ezt a leírást bővítjük, strukturáljuk, ill. elkészítjük a use case-t strukturáltan leíró aktivitási diagramot.

A követelményspecifikáció munkaszakaszban definiált use case-eket a szakirodalom *fekete doboz use case*-eknek *(black-box use case)* is nevezi. A fekete doboz jelző azt hangsúlyozza, hogy a fejlesztésnek ebben a szakaszában nem térünk ki a rendszer, a rendszerkomponensek belső működésére. A cél csak a rendszer viselkedésének specifikálása a rendszert külső szemmel nézve. A követelményspecifikáció során folytatott use case modellezésnek fontos célja a rendszer határainak meghúzása, azaz annak definiálása, hogy a leendő rendszer milyen funkciókat tud végrehajtani, és melyeket nem. A 3.2. táblázat az ÁrajánlatotKészít_Weben use case példáján keresztül demonstrálja a rendszer szokásos külső viselkedésének leírását és a belső működés specifikálását.

3.2. táblázat. Példa egy use case esetén szokásos külső viselkedés leírásra és egy hozzá készített belső viselkedés leírásra

Black-box módszer a rendszer (külső) viselkedésének leírására	**Belső működés specifikálása**
ÁrajánlatotKészít_Weben use case: Végrehajtásakor az Ügyfél megadja az Árajánlat összeállításához szükséges adatokat, majd a rendszer elkészíti az árajánlatot.	ÁrajánlatotKészít_Weben use case: A rendszer ellenőrzi a megadott adatokat, és ha az adatok helyesek, akkor a rendszer az adatbázisba, az árajánlat táblába beszúr egy új sort, rekordot.

A use case-ekre vonatkozó jellemzők, sajátosságok

- A fejlesztendő rendszer szempontjából megkülönböztetünk architektirálisan fontos, egyéb és rendszeridegen use case-eket:
  - Az *architekturálisan fontos use case-ek* az alkalmazás architektú-rájának meghatározása szempontjából kritikusak, szignifikánsak. Ezek a use case-ek valósítják meg a rendszer fő szolgáltatásait. Biztosítják a rendszer alapműködését, a felhasználó által definiált fő feladatokat.
  - Az alap use case-ek mellett vannak ún. *egyéb use case-ek,* amelyek kevésbé kritikusak a teljes rendszer működése szempontjából. A fejlesztés során a munkát célszerű a nagyobb prioritással rendelkező, architektúrálisan fontos use case-ek kidolgozásával kezdeni, majd később foglalkozni az ún. kevésbé kritikus, egyéb funkciókkal. Az architekturálisan fontos és egyéb use case-ek együttes halmaza adja meg a szoftverrendszer határát.
  - A rendszer határának pontos meghúzása a követelményelemzés során történik meg. Ilyenkor előfordul, hogy a korábban definiált use case-ek közül néhány kikerül a rendszer végső követelményhalmazából. Ezekkel a use case-ekkel a fejlesztés további szakaszaiban nem szabad foglalkozni. A fejlesztendő rendszer szempontjából ezeket a use case-eket *rendszeridegen use case-ek*nek hívjuk.
- Egy use case lehet „kicsi vagy nagy”:
  A use case-ek-hez készített forgatókönyvek (lásd később) a rendszer funkcionalitását különböző megközelítésben és különböző szinteken fejezik ki. Mivel egy use case absztrakciós szintjének mélységét mindig az határozza meg, hogy azt milyen céllal, és kinek készítjük, így bizonyos értelemben más-más részletezettséggel és pontosító információkkal készítünk use case-eket a felhasználókkal való kommunikációra, mást a fejlesztők közötti kommunikáció leírására. A RUP módszertan logikáját követve a use case-ek feltárása a részletes kidolgozási fázisban kezdődik, de azok részletes kifejtésére nem mindig ebben a fejlesztési szakaszban kerül sor. A use case-ek finomítására, a fejlesztés későbbi szakaszaiban történő teljes mélységű kifejtetésére a RUP által követett iteratív fejlesztési paradigma ad lehetőséget.
- Egy use case konkrét célt teljesít, valósít meg az aktor számára:
  Ebben az értelemben a use case-eket kétfajta megközelítésben értelmezzük. Beszélhetünk rendszer interakciókról (a rendszer interakciók megnevezése szinonim a korábban használt funkcionális követelmé-

nyek elnevezéssel), és megkülönböztetünk felhasználói célokat. A use case-ek ilyen fajta megkülönböztetésére az ad lehetőséget, hogy a definiált use case-ek milyen alapossággal írják le a követelményeket, a megvalósítandó célokat. A 3.3. táblázatban olvasható példarészlet egy szövegszerkesztő szoftver fejlesztésekor felmerülő potenciális use case-eket tartalmazza, elkülönítve a felhasználói célokat a rendszer interakcióktól (funkcionális követelmények).

3.3. táblázat. Rendszer interakciók és felhasználói célok [12]

Felhasználói célok – user goals	**Rendszer interakciók – system interaction**
– „ensure consistent formatting for a document”; – „make one document’s format the same as another”;	– „define a style”; „change a style”; – „move a style from one document to another”;
A fenti kettőség nem minden esetben határolható el egzakt módon: *„index the document”;*	

- A use case-eket minden esetben az aktorok kezdeményezik:
  Az informatikai alkalmazásokat a felhasználók (a rendszer aktorai) működtetik, használják feladatok végrehajtásához. A fejlesztés során a felhasználók szoftverrel szembeni igényeit use case-ek formájában írjuk le. Az elérendő célok szempontjából releváns use case-ek megtalálására a gyakorlatban számos módszer létezik. Megoldással szolgálnak:
  – az adott területre jellemző felhasználóval való folytatott közös beszélgetések, interjúk,
  – kérdőívek használata csoportos felmérés esetén,
  – brainstorming (ötletbörze) módszer alkalmazása. Használata elsősorban új fejlesztések esetén hasznos, vagy nehezen megfogható, leírható problémák megoldásakor.
  – vitakurzusok a korábbi beszélgetések során definiált dolgok (feladatok, funkciók) megvitatására, tisztázására,
  – egyszerű, ún. favágó módszer: a célokat megfogalmazó dokumentumokból kigyűjtjük az igéket.

A use case-ek specifikálásra vonatkozó szabályok

- A követelményspecifikáció során meghatározott funkcionális követelményeket leíró use case-eket a funkciójellegre utaló névvel kell ellátni, azonosítani.
- A use case-t azonosító név egy tetszőleges karaktersor.
- A use case neve kettős szerepet tölt be. Azonosítja a diszkrét feladatot, amit a rendszernek teljesíteni kell, másrészt a megnevezés az adott use case-t meg is különbözteti a többi use case-től.
- A use case-eket (diszkrét feladat, funkció) az UML modellező nyelv szabályai szerint grafikusan egy ellipszis szimbólum jelöli (lásd 3.4. ábra).
- A use case (funkció) nevét az ellipszis alá írva adjuk meg.

3.4. ábra. A use case szimbóluma UML-ben

- Minden use case-hez tartozni kell egy use case leírásnak.

A fejlesztés során minden use case-hez készül egy részletes leírás. A részletes leírásban a felhasználó szemszögéből rögzítjük a felhasználó és a rendszer között zajló üzenetváltás (párbeszéd) lépéseit. Egy use case azonban nemcsak a szokásos (normál) működéssel hajtódhat végre. A use case szokásos működését a use case normál lefutásának, más szóval alapfolyamatnak hívjuk. A működésnek lehetnek különleges, alternatív esetei (pl. hibás működés), ezek az alfolyamatok. A fejlesztés során minden use case esetén fel kell tárni az összes alternatív lefutási menetet. A use case-ek normál és alternatív működését külön forgatókönyvekben rögzítjük.

A *forgatókönyv*[6], más szóval *szcenárió* a use case egy konkrét végrehajtása, lefutása, a use case egy példánya (instanciája). A use case-ben definiált működés lépésenkénti, ún. step-by-step végrehajtását tartalmazza a fel-

[6] A forgatókönyv nem az UML és nem a RUP tartozéka. Már a hagyományos és korai OO-fejlesztésekben is alkalmaztak forgatókönyveket a feladatok végrehajtási lépéseinek leírásához, használatuk azonban informális jellegű volt. A forgatókönyvszerű leírások nagyban segítették a feladatok megértését, lehetőséget adtak a bonyolultabb problémák átgondolására, majd azok logikai strukturálására.

használó szemszögéből. Egy use case-hez az alternatív működések miatt több forgatókönyv készülhet, de egy biztosan. A forgatókönyv készítésekor a rendszer és a felhasználó között zajló üzenetváltásokat a felhasználó szerepében célszerű megfogalmazni, hiszen a felhasználó fogja az alkalmazást használni. Az üzenetváltások leírásakor a MIT-re koncentráljunk, azt írjuk le, hogy a use case működéskor MI történik, milyen tevékenységek zajlanak, és ne térjünk ki a HOGYAN részleteire, a megvalósítás módjára. A forgatókönyvben elsőként a use case-ben definiált működés normál végrehajtását írjuk le, de el kell készíteni az alternatív esetekhez tartozó forgatókönyveket is.

A forgatókönyv készítésére nincsenek szigorú formai előírások, megkötések. A forgatókönyvben a use case működését, vagyis az egymás után zajló tevékenységeket szöveges formában, mondatszerűen fogalmazzuk meg. A forgatókönyv tartalma (egy lehetséges alternatíva):

- a feladat rövid értelmezése,
- az alternatív útvonalak meghatározása,
- a végrehajtásban résztvevő szereplők meghatározása,
- a közös feladatok kiválasztása.

A forgatókönyv készítésekor érdemes megvizsgálni a következőket:

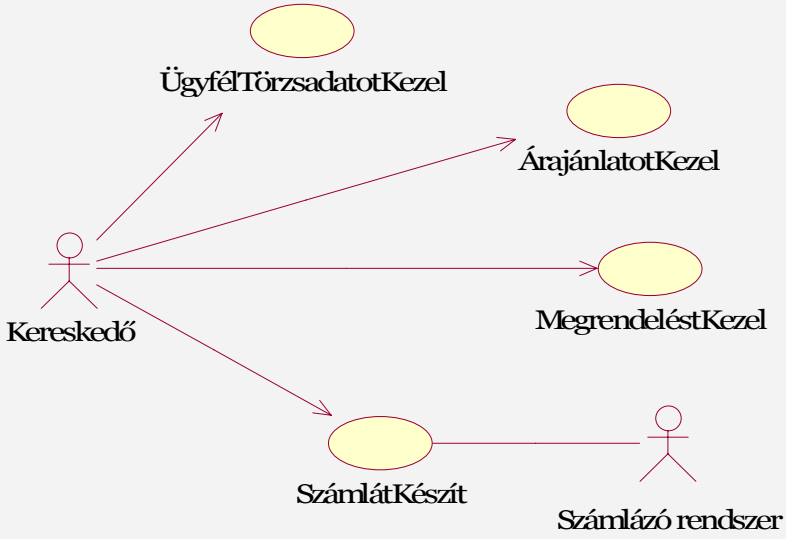
- Hogyan kezdődik a use case?
- Hogyan ér véget a use case?
- Milyen kölcsönhatások történnek az aktor és a use case között?
- Milyen adatok cserélődnek a use case és az aktor között?
- Milyen ismétlődő viselkedést hajt végre a use case?

A követelményspecifikáció munkaszakaszban a use case-ekhez készített forgatókönyvek összessége tisztán és érthetően modellezi a szoftver működését. A use case-ek teljes részletezettségű kifejtésére az elemzés/tervezés fázisban kerül sor, ahol már a megvalósítás (use case realization) miként jére fókuszálunk, a HOGYAN kérdésre helyezzük a hangsúlyt. Ebben a szakaszban térünk át az implementációs részletekre.

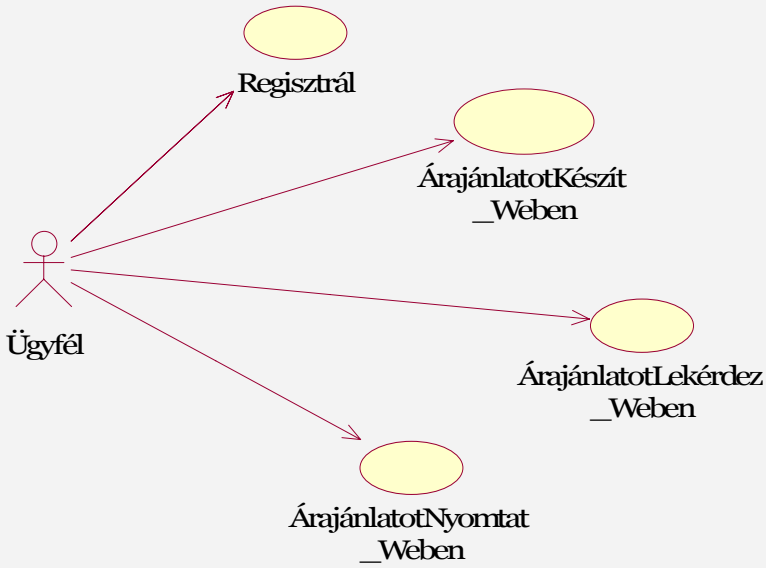
Use case-ek definiálása

A felhasználói követelmények és a rendszer szereplőinek ismeretében határozzuk meg a funkcionális követelményeket leíró use case-eket. Vegyük sorra a rendszerben definiált aktorokat, és határozzuk meg, hogy milyen use case-ek végrehajtását kezdeményezik, ill. mely funkciók végrehajtásában vesznek részt, működnek közre.

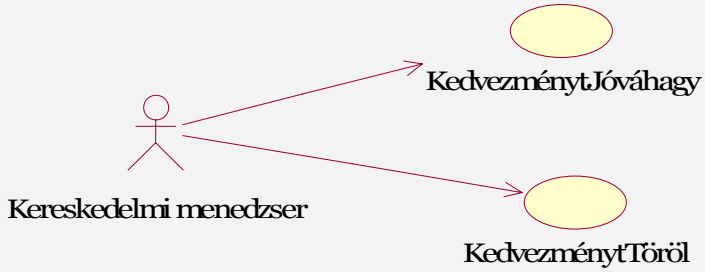
- A Kereskedő végzi az ügyfelek nyilvántartását, árajánlatot készít, számlát állít ki, a megrendelésekkel (értékesítés) kapcsolatos feladatok felelőse. A számlakészítés a Számlázó rendszerben történik. A számlakészítéshez szükséges adatokat a Számlázó rendszer az új rendszerből veszi, ezért meg kell oldani az új rendszer és a Számlázási modul kommunikációját is.



- Az Ügyfél interneten árajánlatot tud összeállítani. Az Ügyfélnek, hogy igénybe vegye a cég webes árajánlat készítési szolgáltatását, regisztrálni kell magát a rendszerbe. A regisztráció során ki kell tölteni egy Regisztrációs lapot. A regisztráció elfogadásával az Ügyfél bekerül a cég ügyfélnyilvántartásába. Az Ügyfél megtekintheti a korábban készített árajánlatokat (a rendszer az árajánlatokat csak a készítés időpontjától számított hat hónapig tárolja).

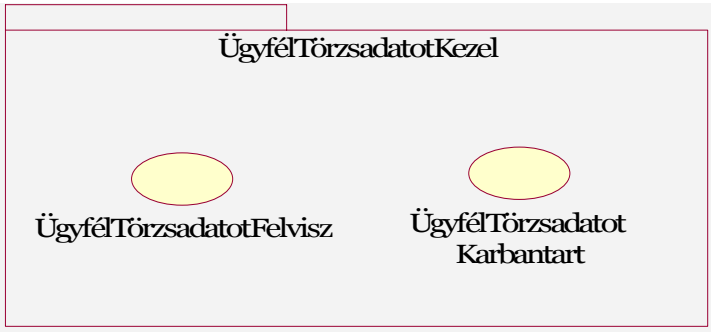


- A Kereskedelmi menedzser kezeli a cég kiemelt ügyfeleit, a nagyvásárlókat. A nagyvásárlóknak adott kedvezmények mértékének (meghatározása) jóváhagyása, törlése tartozik hatáskörébe.

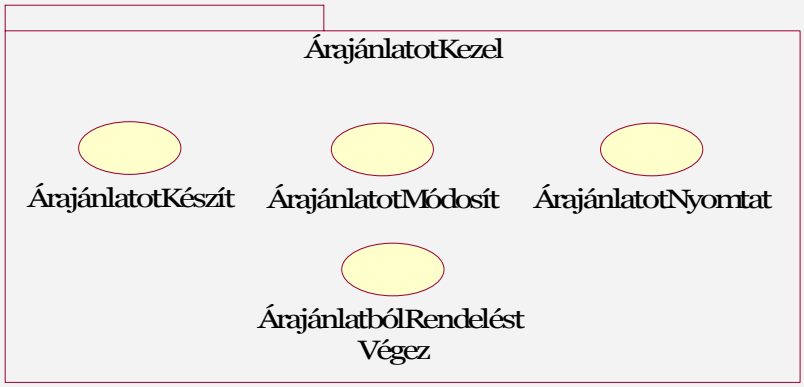


Use case-ek finomítása

A rendszer funkcionalitásának teljes mélységű feltárása, megértése érdekében a Kereskedő aktor által kezdeményezett use case-ek közül az ÜgyfélTörzsadatotKezel use case-t tovább kell részletezni. A finomítás eredményeként egy alsóbb szinten az ÜgyfélTörzsadatotFelvisz, ÜgyfélTörzsadatotKarbantart use case-eket definiáltuk.



Az ÜgyfélTörzsadatotKezel use case esetén végrehajtott finomításhoz hasonlóan az ÁrajánlatotKezel use case-t is tovább kell részletezni. A leképezés eredményeként az ÁrajánlatotKészít, az ÁrajánlatotMódosít, az ÁrajánlatotNyomtat és az ÁrajánlatbólRendeléstVégez use case-ek specifikálhatók.



Az ÜgyfélTörzsadatotKezel és az ÁrajánlatotKezel use case-eket a finomításkor csomag elemként specifikáltuk.

Felhasználói követelmények megvalósítása use case-ekkel

A követelményspecifikáció első szakaszában meghatároztuk a felhasználói követelményeket. A use case modell finomítása során a felhasználói követelményeket egy vagy több use case valósítja meg. Tekintsünk feladatunkból egy konkrét példát!

Az árajánlat kezelési szolgáltatásra vonatkozó felhasználói követelményeket az alábbi módon definiáltuk (lásd 41. oldal):

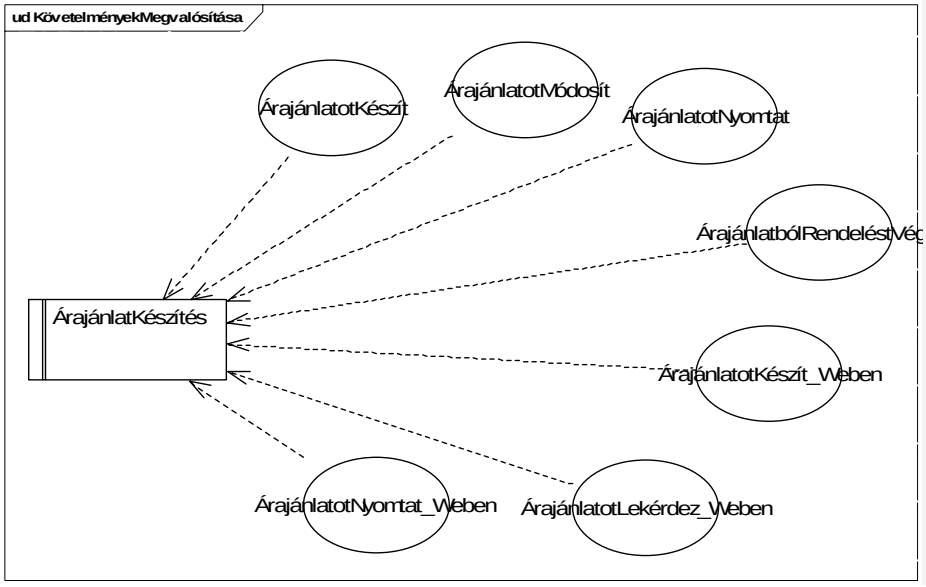
- Az ügyfelek kérhetnek árajánlatot a cég ügyfélszolgálatánál, de igénybe vehetik az internetes árajánlat szolgáltatást is.

Az árajánlat szolgáltatásra vonatkozó felhasználói követelményeket a modellben:

- az ÁrajánlatotKészít,
- az ÁrajánlatotMódosít,
- az ÁrajánlatotNyomtat,
- az ÁrajánlatbólRendeléstVégez,
- az ÁrajánlatotKészít_Weben,
- az ÁrajánlatotLekérdez_Weben és
- az ÁrajánlatotNyomtat_Weben use case-ek valósítják meg, implementálják.

A felhasználói követelmények, és az azokat megvalósító use case-ek közötti kapcsolatot a függőséggel (dependency) modellezzük[7].

Követelmények megvalósítása Enterprise Architect-ben



[7] A felhasználói követelmények, és az azokat megvalósító funkcionális követelményeket leíró use case-ek közötti kapcsolatot leíró diagram az Enterprise Architect 5.0 (EA) CASE fejlesztőeszközzel készült. Az EA lehetőséget nyújt a követelmények modellezésére. Az eszköz a követelményeket a diagramban is jól látható négyzetes szimbólummal jelöli.

Részletes leírás készítése use case-ekhez

A use case lista összeállítása után minden use case-hez készíteni kell egy részletes leírást. Az ÁrajánlatotKészít_Weben use case-hez készített részletes leírásban négy forgatókönyvet kell részletezni:

- A use case normál lefutása szokásos működés esetén: a use case sikeresen véget ér, ha az Ügyfél elfogadja az Árajánlatra vonatkozó feltételeket.

A szokásostól eltérő működéskor a lehetséges lefutások, alternatív útvonalak:

- Az Ügyfél a felhasználói név és a jelszó megadásakor a MÉGSEM gombra kattint.
- Az Ügyfél a felhasználói név és a jelszó megadásakor az OK gombot választja, azonban a rendszer a megadott adatok ellenőrzésekor hibát talál. A megadott adatok vagy azért hibásak, mert az ügyfél azokat rosszul adta meg, vagy azért, mert nem regisztrált felhasználó. Ilyen esetben a use case véget ér.
- Az Ügyfél a rendszer által küldött Árajánlat elfogadása üzenet megerősítésekor a MÉGSEM gombot választja.

Forgatókönyv a normál működés leírására

- Az Ügyfél a cég honlapján aktiválja az „Árajánlat-készítés” funkciót.
- A rendszer megjeleníti a Login dialógusablakot, ahol kéri az Ügyfél felhasználói nevét, és jelszavát.
- Az Ügyfél megadja a felhasználói nevét, és jelszavát.
- A rendszer ellenőrzi (validálja) a megadott adatokat. Hibás adatok esetén újra kéri az adatokat.
- A rendszer megjeleníti az „Árajánlat-készítés” felületet.
- Az Ügyfél megadja a kért adatokat.
- A rendszer validálja a megadott adatok helyességét, ellenőrzi az adatok konzisztenciáját. Ha az Ügyfélnek nem sikerült érvényesen kitölteni a lapot, a rendszer mindaddig visszatér a laphoz, amíg azt az Ügyfél helyesen ki nem tölti.
- A rendszer megerősítést kér az Ügyféltől az Árajánlat elfogadására.
- A rendszer elmenti az Árajánlat adatait, és nyugtázó üzenetet küld a képernyőn keresztül az Ügyfélnek az Árajánlat készítésének sikerességéről.

A forgatókönyv készítésekor ellenőriztük, hogy:

- A use case akkor kezdődik, amikor a cég honlapján az Ügyfél aktiválja az „Árajánlat-készítés” funkciót.
- A use case sikeresen véget ér, ha az Ügyfél elfogadja az Árajánlatra vonatkozó feltételeket, a rendszer elmenti az adatbázisba az Árajánlat adatait.
- Az Ügyfél és a rendszer között kölcsönhatások történnek akkor, amikor például az Ügyfél a felhasználó név és a jelszó megadása után megnyomja az OK gombot. Helyes felhasználó név és jelszó megadása esetén a rendszer megjeleníti az „Árajánlat-készítés” felületet.
- Adatok cserélődnek a szoftverrendszer és az Ügyfél között akkor, amikor például az Ügyfél megadja a felhasználói nevét és jelszavát, vagy akkor, amikor az Árajánlat készítésekor az Ügyfél megadja az adatokat.
- A szoftverrendszer mindaddig kéri a felhasználó nevet és jelszót, valamint az Árajánlat készítési lapon szereplő adatokat, amíg azok konzisztencia és validitási szempontból nem helyesek.

A forgatókönyv a use case egy konkrét végrehajtásának lépéseit mutatja be. A forgatókönyvben meghatározott lépéseket az UML-ben tevékenységeknek, aktivitásoknak nevezzük. A forgatókönyv tehát a tevékenységek sorozatát írja le szöveges formában. A forgatókönyvben meghatározott tevékenységek menetének grafikus szemléltetésére az UML diagramok közül az *aktivitási (tevékenységi)* diagramot használjuk. Egy use case-hez annyi aktivitási diagram[8] készül, amennyi a forgatókönyvek száma. Gyakran azonban az alternatív végrehajtásokhoz tartozó forgatókönyveket – amennyiben nem bonyolítja az aktivitási diagramot – egy közös aktivitási diagrammal írunk le.

A forgatókönyveknek aktivitási diagramokkal való szemléltetése azért célszerű, mert grafikusan áttekinthetőbbé teszi, hogy mi történik, milyen tevékenységek zajlanak le a use case végrehajtásakor.

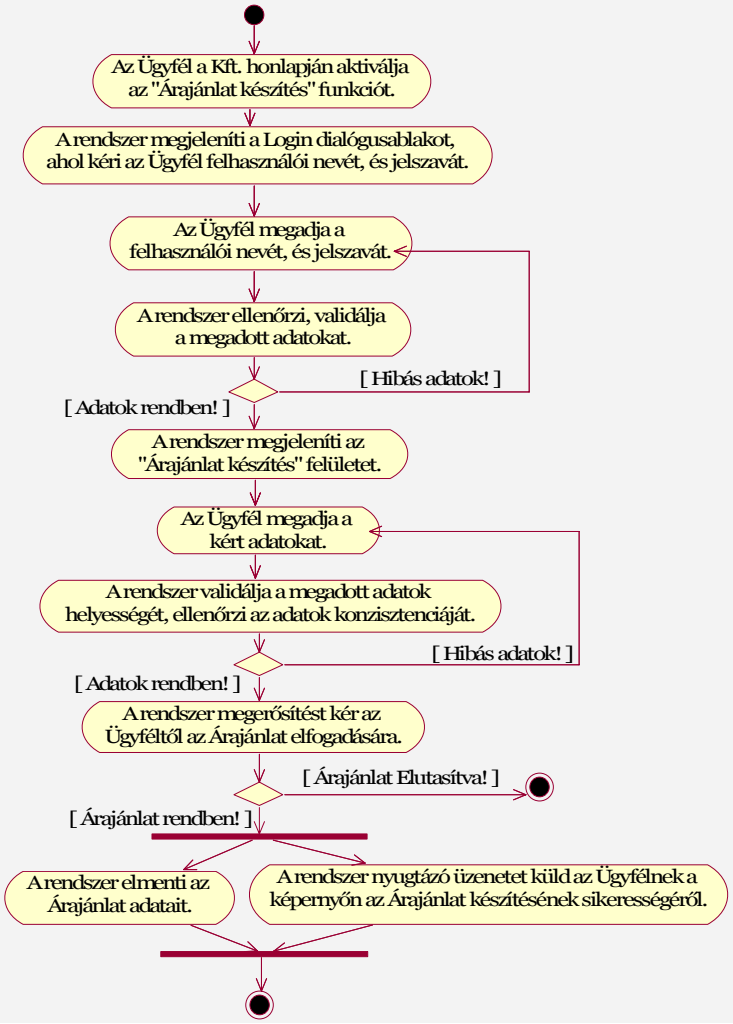
Az aktivitási diagram egy kezdő- és egy vagy több végpontot tartalmaz. A use case végrehajtási menetében az elágazási/döntési (őrfeltétel) pontoktól függően több végpont is lehetséges. A gyakorlatban a diagram készíté-

[8] Az aktivitási diagramokat a 8. fejezet ismerteti.

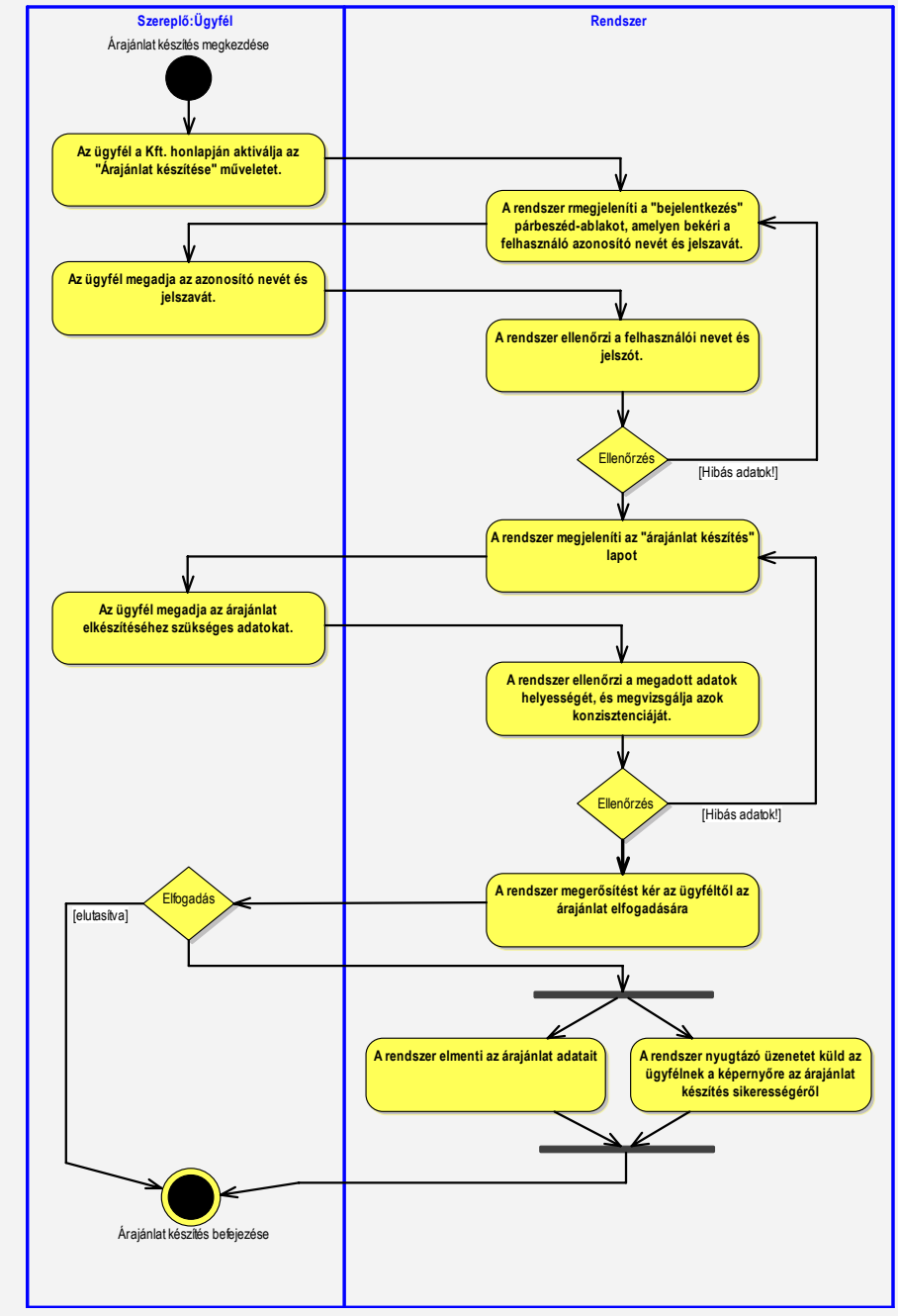
sekor csak egy végpontot célszerű feltüntetni. A kezdő és végpontok mellett a diagram felsorakoztatja az aktivitásokat (tevékenységeket), grafikusan kiemeli az elágazásokat jelölő döntési pontokat, a szinkronizációs vonallal elkülöníti a párhuzamosan, az azonos időben zajló tevékenységeket.

Aktivitási diagram készítése use case-hez

Az ÁrajánlatotKészít_Weben use case-ben definiált működés normál végrehajtását leíró forgatókönyv alapján készítsük el az UML aktivitási diagramot!



Aktivitási diagram úszópályákkal (swimlane-ekkel) kiegészítve



3.2.3. Kapcsolatok

A use case modell harmadik építőeleme a kapcsolat. *Kapcsolat* alatt klasszikus értelemben az aktorok és a use case-ek közötti kapcsolatokat értjük. Az UML-ben a rendszer szereplői és a use case-ek között definiált kapcsolatok modellezésére a use case diagram szolgál. A kapcsolatot a diagramban az aktorokat és a use case-eket összekötő vonal szimbolizálja. A vonal lehet irányított.

Aktorok és use case-ek között értelmezett kapcsolatok típusa

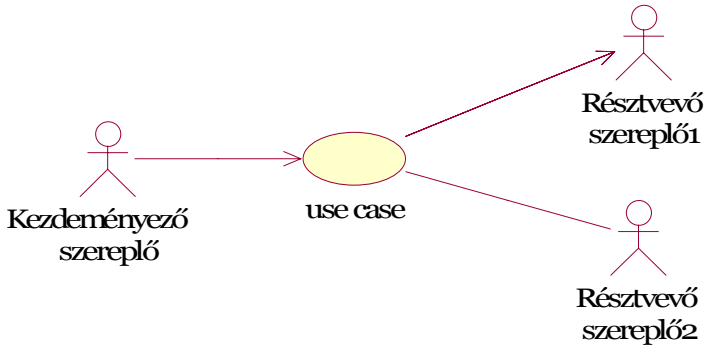
A korábbi alfejezetekben tárgyaltuk, hogy egy use case-t aktor kezdeményezhet, aktivizálhat. A rendszer szereplői és a use case-ek közötti kapcsolatot egy nyíl jelöli, mint ahogy azt a 3.5. ábra use case diagramja illusztrálja. Az irányítás nem a kommunikáció irányát mutatja, hanem azt, hogy melyik aktor kezdeményezi az adott use case végrehajtását.

Az aktor és a számítógépes rendszer között alapértelmezésben kommunikációs jellegű kapcsolat van, a kommunikatív jelleget a kapcsolatot szimbolizáló nyíl felett <<communicate>> sztereotípiával jelölhetjük. A gyakorlatban azonban a sztereotípiát nem szokás külön kiírni a kapcsolat típusának klasszikus volta miatt.

3.5. ábra. Kapcsolat ábrázolása kezdeményezés esetén

Egy feladat (use case) végrehajtásában több aktor is közreműködhet (lásd 3.6. ábra). A use case megvalósításában részt vevő szereplőket és a use case-t egyszerű (irányítás nélküli) vonal köti össze.

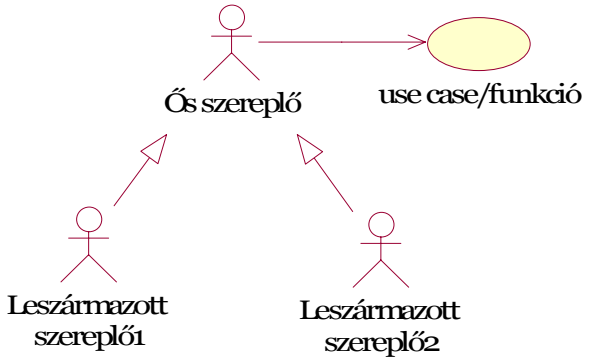
A use case diagramban az aktorok és a use case-ek között definiált klasszikus társítási kapcsolatok mellett léteznek ún. speciális kapcsolatok. Értelmezünk két aktor közötti kapcsolatot. Definiálhatunk use case-ek közötti speciális viszonyokat is.



3.6. ábra. Kapcsolat szemléltetése közreműködés esetén

Speciális kapcsolat két aktor között

Két aktor öröklődési viszonyban állhat egymással. Öröklődési kapcsolat akkor jöhet létre, ha egy use case végrehajtásakor több szereplő is betölti ugyanazt a szerepet. Az öröklődési viszonyban két aktortípust különböztetünk meg, az egyik a leszármazott, a másik az ős szereplő. A leszármazott minden use case-zel kapcsolatban van, amivel az ős szereplő kezdeményez kapcsolatot. Az ős szereplőnél definiált minden use case végrehajtását kezdeményezheti, de annál akár többet is. Az aktorok között értelmezett öröklődési viszony UML-ben történő grafikus leírását a 3.7. ábra szemlélteti.



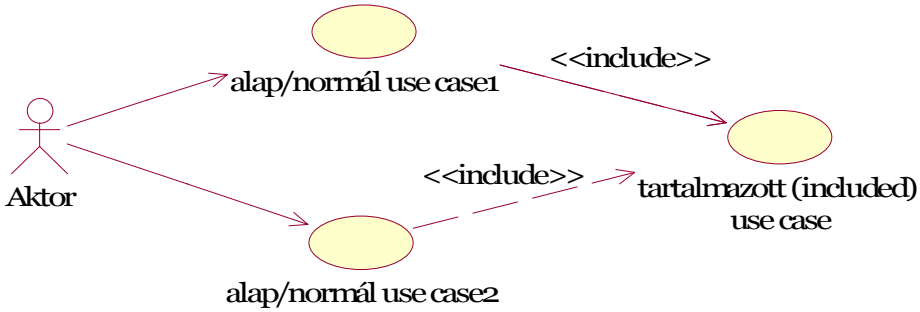
3.7. ábra. Öröklődési viszony aktorok között

Speciális kapcsolatok use case-ek között

A use case-ek között három speciális viszonyt értelmezünk. Megkülönböztetünk tartalmazást, kiterjesztést és öröklődést.

Tartalmazás – include viszony

- A szereplő által kezdeményezett (alap vagy normál) use case-ek végrehajtásában vannak olyan részek, lépések, amelyek mindegyik use case végrehajtásakor bekövetkeznek és azonos módon játszódnak le. Érdemes az azonos viselkedéseket egy külön use case-be kiemelni. A kiemelt viselkedést tartalmazott vagy rész use case-nek hívjuk. A tartalmazott elnevezés utal arra, hogy a tartalmazott use case az alap use case-nek szerves része. A tartalmazott use case végrehajtása feltétel nélküli, vagyis az alap use case végrehajtáskor mindig bekövetkezik, lejátszódik.
- Az UML-ben az alap use case-t és a tartalmazott use case-t szaggatott nyíl köti össze.
- A szaggatott nyíl az alap use case-től a tartalmazott felé mutat.
- A kapcsolat tartalmazás jellegét a szaggatott nyílon elhelyezett, francia zárójelek közé írt <<*include*>> *sztereotípiával* jelöljük. A leendő rendszer fejlesztése során definiált use case-ek között értelmezett tartalmazás viszonyt a 3.8. ábra use case diagramja illusztrálja.

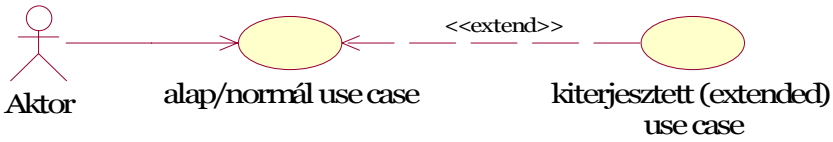


3.8. ábra. Include kapcsolat use case-ek között

Kiterjesztés – extend viszony

- A modellben lehetnek use case-ek, amelyek végrehajtási menetében bizonyos feltételek bekövetkezésekor a vezérlés egy másik use case-nek adódik át. Ilyenkor a normál use case-nek egy bővített változata játszódik le. Mivel a normál use case viselkedésében a feltétel csak bizonyos esetekben következik be, ezért a normál use case-t bővítő viselkedést érdemes külön use case-ben leírni.
- A feltételt a követelmények specifikálásakor kell megadni.

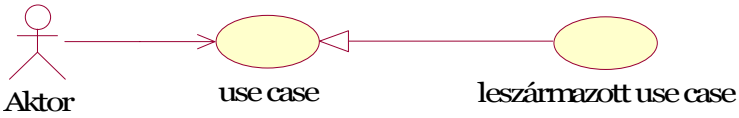
- Az UML-ben az alap use case-t és a kiterjesztett use case-t szaggatott nyíl köti össze.
- A szaggatott nyíl a kiterjesztett use case-től az alap use case felé mutat.
- Az extend viszonyt a szaggatott nyílon elhelyezett, francia zárójelek közé írt <<*extend*>> *sztereotípiával* adjuk meg. A normál use case végrehajtása során a normál use case végrehajtási menetében definiált feltétel teljesülésekor bekövetkező vezérlésátadás UML szerinti grafikus leírását a 3.9. ábra use case diagramja szemléleti.



3.9. ábra. Az extend kapcsolat

Öröklődés – generalizáció

- Az öröklődési viszony esetén a leszármazott use case örökli a normál use case viselkedését, kapcsolatait. A leszármazott az eredeti/normál use case viselkedéséhez hozzáadhat újabb viselkedéseket, sőt az eredeti use case viselkedését felülbírálhatja, felülírhatja.
- Az öröklődést az UML-ben egy telt fejű nyíl jelöl. A nyíl a leszármazott use case-től az általánosított normál (ős) use case felé mutat. Use case-ek között értelmezett öröklődési viszony grafikus megjelenítését a 3.10. ábra use case diagramja illusztrálja.



3.10. ábra. Öröklődési viszony use case-ek között

A speciális kapcsolatokkal a rendszerünkben zajló folyamatokat tudjuk még pontosabban modellezni. Az <<include>> minősítést azoknál a funkcióknál használjuk, amelyek végrehajtási menetében közös tevékenységek azonosíthatók. Az <<include>> sztereotípia megadásával a közös viselkedést a rendszerben csak egyszer kell modellezni. A gyakorlatban az <<include>> sztereotípiát általában hibaesemények kezelésének, a fel-

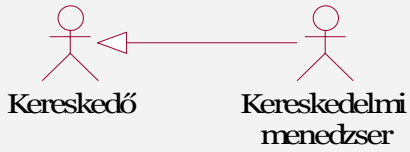
használói visszalépéseknek, megszakításoknak és egyéb különleges részte-vékenység sorozatoknak a leírására használjuk.

A use case-ek között értelmezett <<include>> és <<extend>> kap-csolatok modellezésekor az általánosítás irányának jelölése ellentétes. A nyíl az általánosabb viselkedésű modellelemtől a specializáltabb felé mutat. Az ábrázolásnak ez a módja logikus, hiszen <<extend>> viszony esetén a bővített use case az általánosabb, <<include>> esetén pedig a normál use.

Use case modell készítése, strukturálása

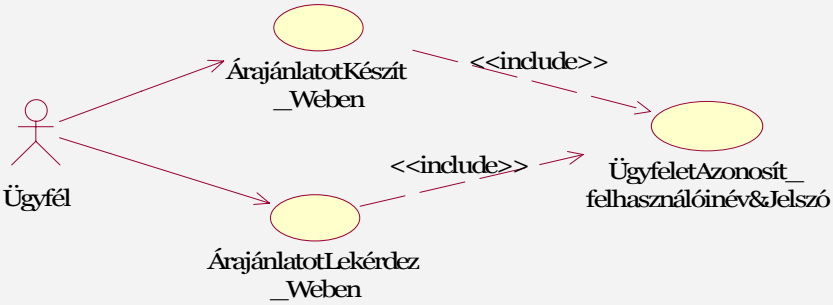
A use case modell strukturálása során a modellben azonosított speciális kapcsolatok az alábbiak:

- A Kereskedelmi menedzser rendelkezik mindazokkal a jogokkal, ami-vel a Kereskedő munkatárs. Ez a modellben azt jelenti, hogy a Keres-kedelmi menedzser a Kereskedő összes kapcsolatát örökli. A Keresked-dő által elindított összes use case-t kezdeményezheti, sőt további fel-adatokat (KedvezménytJóváhagy, ill. KedvezménytTöröl use case-ek) is végrehajthat.

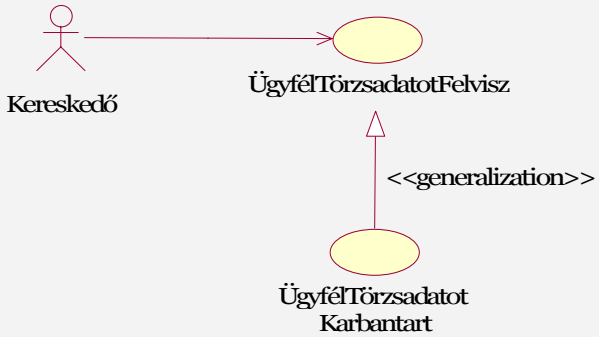


- A megrendelések (MegrendeléstKezel use case) összeállításakor a rendszer nagyvásárlók esetén ellenőrzi, hogy az Ügyfél három korábbi megrendelését pénzügyileg teljesítette-e. Abban az esetben, ha pénz-ügyi teljesítésekre vonatkozó feltétel nem teljesül, (Extention Point) a rendszer nem engedi rögzíteni az újabb megrendelést. A vezérlés a Kedvezmény zárolása use case-nek adódik át, aminek végrehajtása so-rán figyelmeztetés megy a Kereskedelmi menedzser-nek, aki törli az Ügyfélnek adott törzsvásárlói kedvezményt (KedvezménytTöröl use case). A KedvezménytZárol use case normál esetben (ha a feltételek: nagyvásárló+pénzügyi teljesülnek) nem játszódik le, csak akkor, ha a feltételek (az Ügyfél nagyvásárló és az Ügyfél három korábbi megren-delését pénzügyileg nem teljesítette) teljesülnek.

- Interneten az ügyfelek árajánlatot csak akkor készíthetnek, vagy a meglevő ajánlataikat csak akkor kérdezhetik le, ha regisztrált ügyfelek a rendszerben. Ez azt jelenti, hogy minden alkalommal, új árajánlat felvitelekor, lekérdezésekor azonosítani kell magukat a regisztráció (Regisztrál use case) során megadott helyes felhasználói névvel és jelszóval. Az ügyfél-azonosítás az ÁrajánlatotKészít_Weben, az ÁrajánlatotLekérdez_Weben use case-ek végrehajtásakor azonos módon játszódik le, ezért ezt érdemes kiemelni egy külön use case-be (ÜgyfeletAzonosít_felhasználó név&Jelszó use case).



- Az ÜgyfélTörzsadatotKarbantart use case örökli az ÜgyfélTörzsadatotFelvisz use case viselkedését. Az ÜgyfélTörzsadatotFelvisz use case végrehajtása új ügyfelet rögzít a rendszerben. Az ÜgyfélTörzsadatotKarbantart use case végrehajtása az ügyféladatok rögzítése mellett még biztosítja az ügyféladatok lekérdezését és módosítását. Módosítás esetén a korábban rögzített és módosított ügyféladatok felülíródnak.

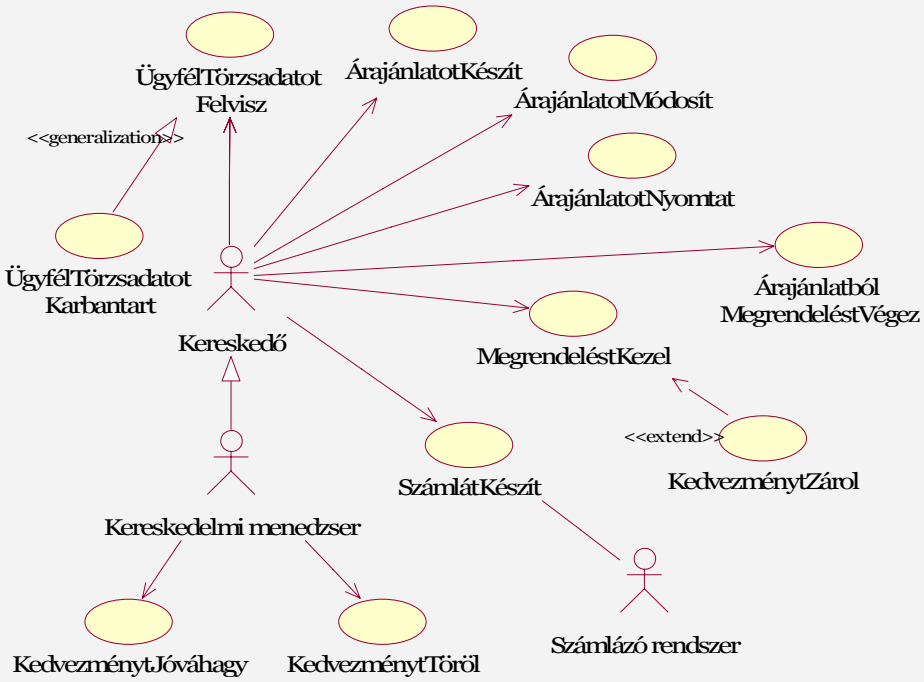


3.2.4. Use case diagram

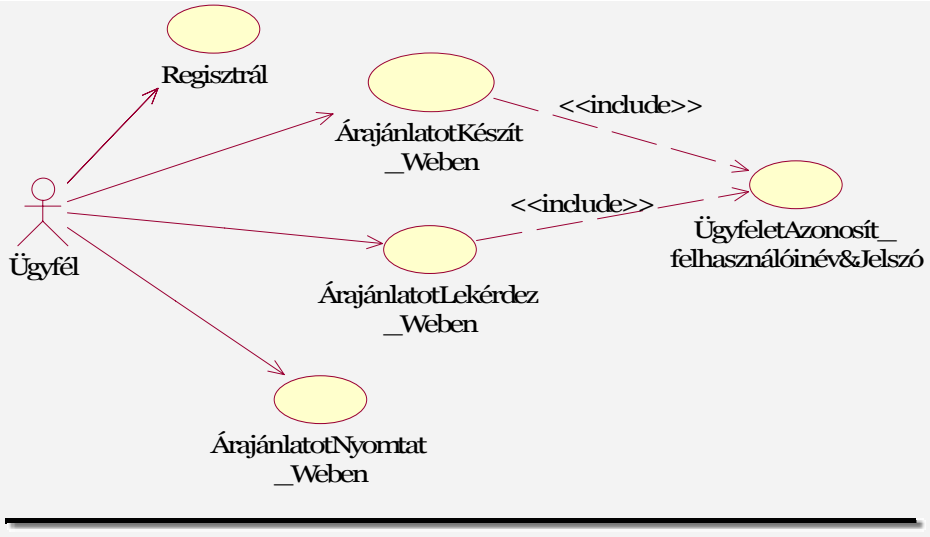
A követelményspecifikációban feltárt use case-eket, szereplőket és a közöttük értelmezett kapcsolatokat *use case diagramban* ábrázoljuk. A diagram segíti a rendszer viselkedésének megértését és grafikus modellezését. A fejlesztendő rendszerrel kapcsolatban elkészített use case diagramok alkalmasak a tervezett rendszerrel szemben támasztott követelmények (use case-ek halmaza) meghatározására, leírására. A diagram szemléletes, könnyen áttekinthető, logikája könnyen érthető, emiatt nemcsak a fejlesztők által végzett követelményelemzést könnyíti meg, de hatékony eszköz a felhasználókkal történő kommunikáció megkönnyítésére.

Use case modell

Az ügyféladatok, az árajánlatok és a megrendelések (vásárlások) kezelését modellező use case diagram részletet az alábbi ábra szemlélteti.



Az Ügyfél által kezdeményezett use case-eket összefoglaló use case diagramot a következő ábra foglalja össze.



3.3. A use case modell dokumentálása

A use case modellezés során fontos a különböző modellelemek megfelelő dokumentálása. Minden elemhez készíteni kell szöveges specifikációt, leírást, szükség esetén a még jobb megértés érdekében részletesebb magyarázó kiegészítést kell adni.

A use case modell dokumentálásának menete:

- Aktorok azonosítása.
- Minden aktor esetén meg kell határozni, hogy mit vár el a rendszertől.
- Use case-ek feltárása, use case lista összeállítása.
- Minden use case-hez részletes leírás készítése:
  - Alternatív forgatókönyvek készítése a use case működésének leírására.
  - Aktivitási/tevékenységi diagram készítése.
- Kapcsolatok meghatározása:
  - Kapcsolatok aktor és use case között.
  - Speciális kapcsolatok azonosítása.
- A rendszer funkcionalitásának, viselkedésének grafikus modellezése use case diagramok készítésével.
- A fejlesztés során menetközben módosult funkciók dokumentálása, az elfogadott módosítások átvezetése a követelménydokumentumba.

3.4. Use case realizáció – a use case-ek megvalósítása

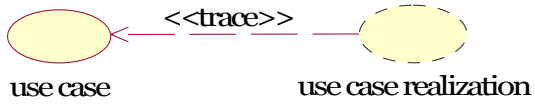
A követelménymodellben a use case-ek viselkedésének vizsgálatakor csak a felhasználó és a szoftverrendszer közötti interakciók pontos leírására koncentráltunk. Az új rendszert kívülről vizsgáltuk, elemeztük, a MIT-re, a rendszer által megkövetelt szolgáltatások megfogalmazására helyeztük a hangsúlyt, nem tértünk ki a megvalósítás részleteire, a HOGYAN-ra.

A use case modellben definiált use case-eket az elemzési, majd a tervezési modellben tovább elemezzük, részletezzük. Ezekben a fejlesztési munkaszaka-szokban a rendszer belsejét térképezzük fel. Azt vizsgáljuk, hogyan lesznek a rendszertől elvárt funkciók, use case-ek megvalósítva, majd implementálva. Ennek a munkának egy fontos lépése, hogy a use case modellben definiált minden use case-hez el kell készíteni annak megvalósítását specifikáló use case realizációt (use case realization).

Minden use case-hez definiálhatunk az elemzési modellben egy use case realizációt. A *use case realizáció* a use case egy lefutásának részletes leírása a rendszeren belül. A use case realizáció a modellben egy olyan use case lesz, aminek a sztereotípiája <<use case realization>>. A modellben a use case realizációt szaggatott vonalú ellipszis szimbolizálja (lásd 3.11. ábra).

3.11. ábra. A Use Case Realization UML szimbóluma

A követelménymodellben definiált use case és az elemzési modellben ennek megvalósítására szolgáló use case realizáció között függőségi viszonyt értelmezünk. Az asszociációnál erősebb megszorítású függőségi viszonnyal azt fejezzük ki, hogy a use case megvalósítása a követelménymodellben meghatározott use case viselkedésétől függ. A követelménymodellben meghatározott use case viselkedésében bekövetkezett változás kihat a megvalósításra, módosítja az implementációt. A két elem között értelmezett függőségi kapcsolat felett a <<trace>> sztereotípia megadásával jelöljük, hogy az irányítatlan végén szereplő modellelem a másik modellelem megvalósítása (lásd 3.12. ábra).



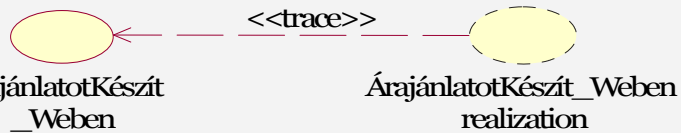
3.12. ábra. Use case megvalósítása függőségi viszonnyal

A use case realizáció modellezésére az UML diagramok közül az eseménykövetési (szekvencia) diagramot használjuk. A diagram tartalmazza a működéshez szükséges objektumokat és az objektumok közötti üzenetváltásokat, azok időbeli sorrendjében. A use case-ben leírt működést a rendszerben definiált objektumok valósítják meg, mely objektumok egymással üzenetváltásokkal kommunikálnak. A use case funkciója az üzeneteken keresztül teljesül. Az üzenetek eljárás vagy függvény jellegűek lehetnek, ez utóbbiak visszatérési értékét is megadhatjuk. Az üzenetek paraméterezhetők.

Az elemzési szakaszban a use case egy részletes lefutását leíró eseménykövetési diagramot a tervezési modellben tovább részletezzük. Habár a különböző munkaszakaszokban elkészített eseménykövetési diagramok mindegyike az eredeti use case-ben leírt működést írja le, azonban a tervezés során készített modellek tovább részletezik az elemzési modellben leírt működést a kiválasztott implementációs környezetben megvalósítható objektumok, ill. osztályok definiálásával és a köztük megvalósuló üzenetváltások leírásával. Az elemzési modellben nagyvonalakban definiált szoftverarchitektúra végleges formája a tervezési fázisban alakul ki. A tervezési szakaszban pontosan definiálni kell a leendő rendszer felépítését, egyértelműen elhatárolva egymástól a rétegeket (üzleti logika réteg, alkalmazási réteg, adatbázis réteg stb.), és a rétegekbe tartozó objektumokat.

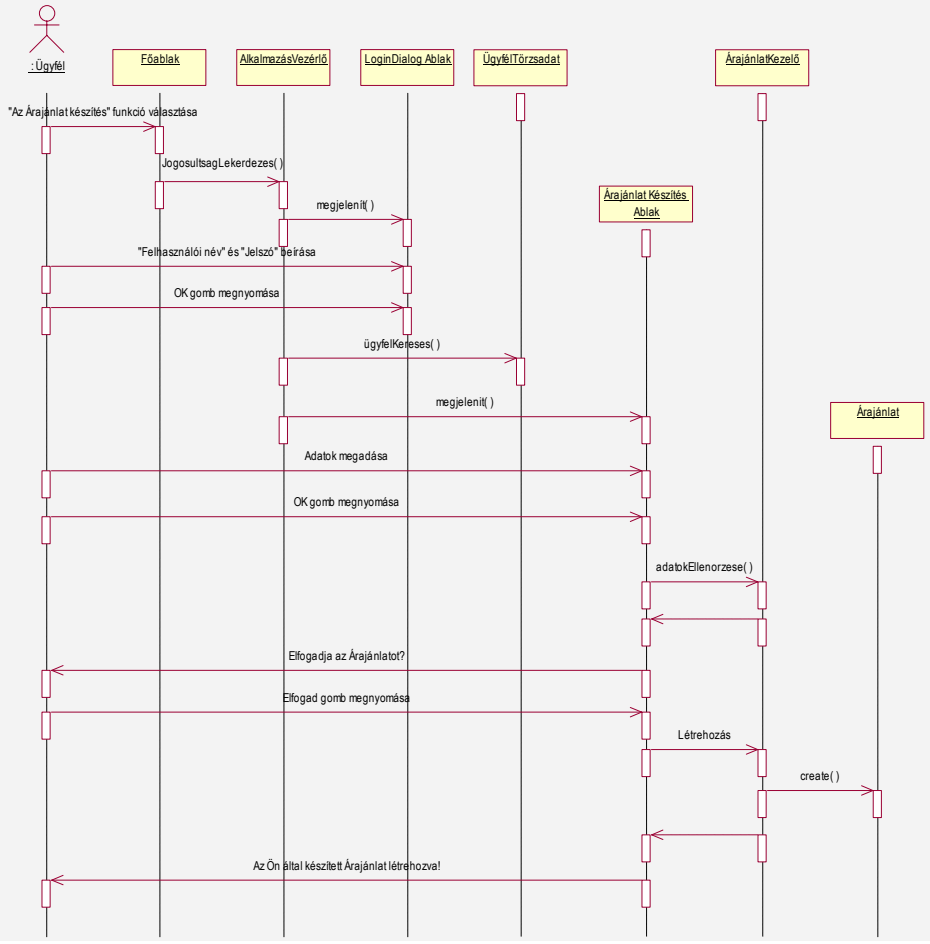
Use case relaizáció készítése

Az ÁrajánlatotKészít_Weben use case lefutását a rendszeren belül az ÁrajánlatotKészít_Weben use case realization valósítja meg.



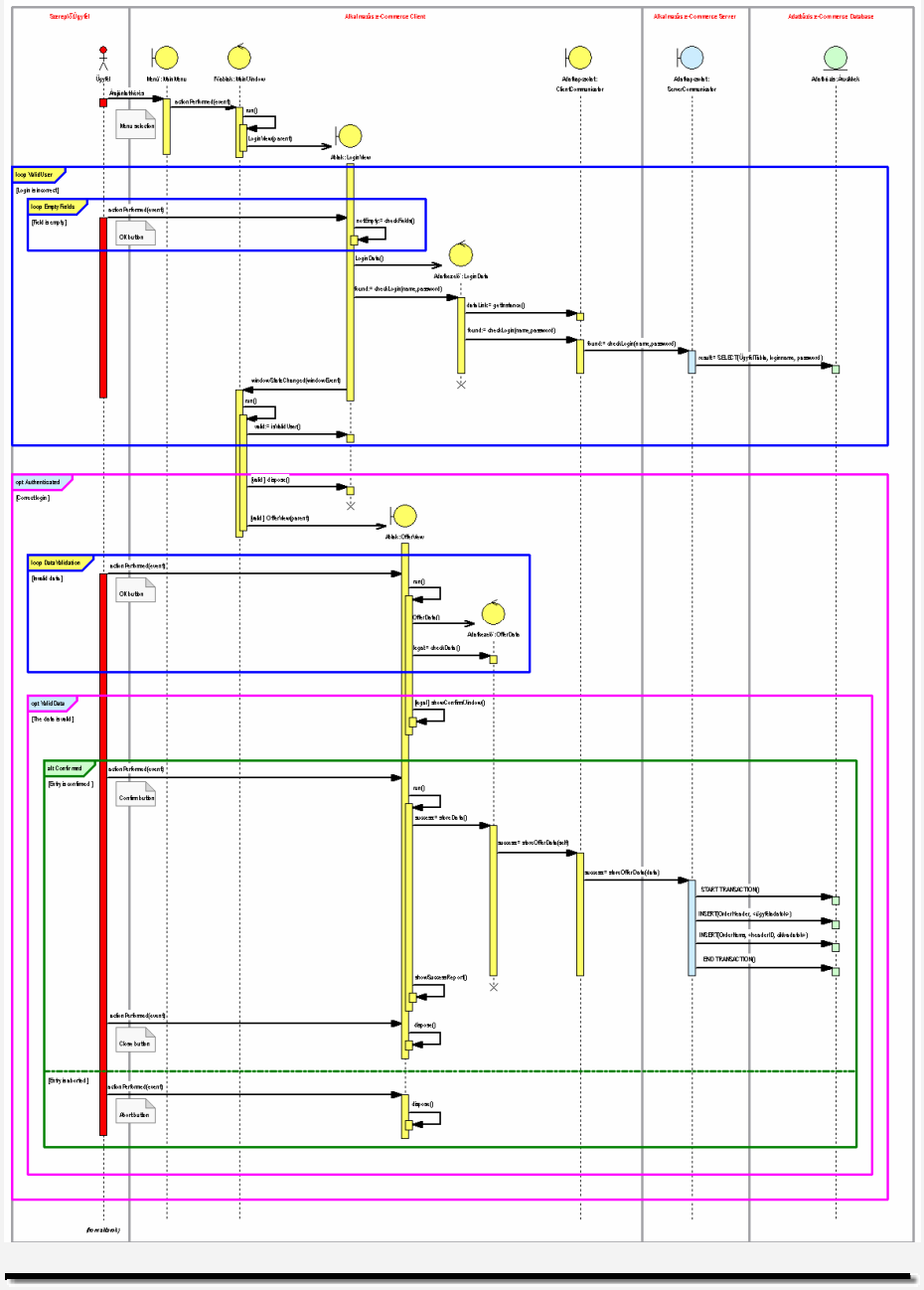
Eseménykövetési diagram készítése az elemzési munkaszakaszban

Az ÁrajánlatotKészít_Weben use case realizációt az eseménykövetési diagrammal írjuk le. A diagram az UML 1.5-ös szabvány alapján készült. A diagramon ábrázoljuk a működésben megjelenő objektumokat és az objektumok közötti üzenetváltásokat, azok időbeli sorrendjében. Az elemzési szakaszban a use case realizációt leíró eseménykövetési diagram még nem olyan részletes, mint a tervezési modellben készített eseménykövetési diagram. Az eseménykövetési diagram (szekvenciadiagram) leírása az 5.2. alpontban található.



Eseménykövetési diagram készítése a tervezési szakaszban

A diagram készítésekor az UML 2.0 szabvány leírását követtük.



A szoftverfejlesztési folyamatban a követelmények összegyűjtése, elemzése és változásaik nyomon követése fontos feladat. Az ellenőrzési munka során azt kell megvizsgálni, hogy a követelmények, a követelményeket leíró modellek (use case modellek), dokumentációk (pl. fogalomszótár) stb. minden szempontból megfelelnek-e a felhasználó elvárásainak. Az ellenőrzési feladatok végrehajtását review-k (értékelő áttekintés) készítése vagy prototípusok készítése segíthetik.

A követelményspecifikációban az ellenőrzési munkának ki kell terjedni a teljes követelményhalmazra. A rendszert leíró követelményhalmaznak:

- Minden szempontból teljesnek kell lenni, azaz tartalmazni kell minden szolgáltatás pontos leírását, az ehhez szükséges információkat.
- Konzisztensnek kell lenni, vagyis a rendszer leírásban, a szolgáltatások definícióiban, a felállított modellekben nem lehetnek ellentmondások.
- Hibamentesnek kell lenni.

A rendszerre vonatkozó követelményhalmaz mellett ki kell térni a követelmények egyedi vizsgálatára. Érdemes számba venni, hogy a vizsgálat tárgyát képező követelmény érthető-e, jól definiált, tesztelhető-e, ismert-e a forrása, megvalósítása esetleg sért-e valamilyen szakterületi követelményt.

A követelmények ellenőrzése, az ellenőrzési munka eredményeként a változtatások végrehajtása, a javítások elvégzése elengedhetetlenül szükséges a további fejlesztési feladatok felhasználói igényeknek megfelelő végrehajtásához.

4. Osztályok, objektumok, osztálydiagram

Az osztálydiagram a legismertebb objektumorientált (OO) modellezési technika. Az OO-módszertanok legalapvetőbb eszköze, a rendszer objektumait leíró osztályokat és a közöttük levő kapcsolatokat modellezi.

A fejlesztés különböző munkaszakaszaiban készülnek osztálydiagramok. Ez nem újabb és újabb modellek megalkotását jelenti, a fejlesztés teljes ideje alatt egy alapmodellt vizsgálunk. Ezt az egy modellt fokozatosan bővítjük a megfelelő részletekkel a fejlesztés menetében előrehaladva. A modell aktuális állapotát, érettségét az egyes szakaszokban készített osztálydiagramokkal ábrázoljuk.

4.1. Osztálydiagramok használata a fejlesztés különböző szakaszaiban

Az osztálydiagramokat a fejlesztés különböző szakaszaiban más és más céllal definiáljuk és használjuk, ennek megfelelően azok értelmezése is eltérő.

4.1.1. Osztálydiagramok az üzleti modellezésben

Az *üzleti modellezés* során készített ún. *szakterületi osztálymodell* a szakterület valós elemeit ábrázolja, amelyek meghatározó szerepet játszanak a vizsgálat tárgyát képező szervezet, szakterület működésében, a probléma megértésében. A modell célja az elemek fogalmi szintre emelése, a közöttük értelmezett kapcsolatok és szerepeik meghatározása. A szakterületi osztálymodell elemei az üzleti aktorok és üzleti entitás jellegű objektumok. A szakterületi osztálymodell elkészítésének alapja az *üzleti folyamatmodell (Business Process Model),* a folyamatmodellezés során meghatározott üzleti use case-ek, és a use case-ek megvalósításában részt vevő üzleti szereplők/aktorok (szerepkörök) és entitások. A szakterületi osztálymodellezés során azonosított üzleti entitásobjektumokat, az objektumok közötti kapcsolatokat az *üzleti objektummodellben (Business Object Model)* foglaljuk össze.

Az üzleti modellek (szakterületi osztálymodell, üzleti objektum modell) és a követelményspecifikációban felállított use case modell az alapja, bemenete az elemzés/tervezés[9] során fokozatosan bővített osztálymodelleknek.

[9] A RUP az elemzést és a tervezést nem önálló munkafolyamatként definiálja, azokat a kétdimenziós modellben egy munkaszakaszként értelmezi.

4.1.2. Osztálydiagramok az elemzési munkaszakaszban

Az elemzés kezdeti szakaszában a *szekvenciadiagramok* segítségével feltárjuk a use case-ek működéséhez (use case realization) szükséges objektumokat, meghatározzuk az objektumok között zajló üzenetváltásokat. Az objektumok kapcsolati viszonyait, a kapcsolatokban játszott szerepeket az *együttműködési (collaboration) diagramban* ábrázoljuk. Az együttműködési diagram, amelyet a gyakorlatban objektumdiagramnak is neveznek, csak a use case működésében azonosított objektumokat, a kapcsolatokat, szerepeket és az objektumok kapcsolódásának számosságát képes leírni. A szekvenciadiagramok és az együttműködési diagramok segítségével feltárt objektumokat leíró osztályok, adataik (attribútumok), műveleteik és kapcsolataik közvetlen forrásként, bemenetként szolgálnak a fejlesztendő szoftveralkalmazás *elemzési osztály-modelljének* elkészítéséhez.

Az elemzési osztálymodell az elemzés során feltárt új elemek mellett tartalmazza az üzleti objektum modellben specifikált azon elemeket is, amelyek a tervezett szoftverrendszer működéséhez szükségesek[10]. Az üzleti modellezés során felállított üzleti objektum modell nem más, mint az elemzési osztály-modell egy speciális, más megközelítésben értelmezett formája. Az üzleti objektummodell a szakterület objektumai alapján általánosított kategóriákat, és azok kapcsolatait ábrázolja. Az itt azonosított objektumokat részletezzük, újabb jellemzőkkel bővítjük az elemzési osztálymodellben a szekvencia és az együttműködési diagramok segítségével.

4.1.3. Osztálydiagramok a tervezési szakaszban

Az elemzési osztálymodellt az elemzés/tervezés során továbbbővítjük, fejlesztjük. Részletezzük az osztályokat, pontosan deklaráljuk az attribútumokat, műveleteket, értelmezzük a kapcsolatokat, azonosítjuk a speciális asszociációs viszonyokat. A fenti műveletek eredményeként áll elő a szoftveralkalmazás részletes osztálymodellje, amely a kódolás alapja. A szakirodalom ezt a modellt **tervezési osztálymodellként** azonosítja.

4.1.4. Az osztálydiagramok jelölésrendszere, perspektívák

A fejlesztés különböző szakaszaiban (üzleti modellezés, elemzés, tervezés) készített osztálydiagramok jelölésrendszerükben számos hasonlóságot mutatnak. Az osztályok és az osztályok között értelmezett kapcsolatok ábrázolására az UML nyelv szimbólumait használjuk. Az osztálydiagramok

[10] Az üzleti modellben definiált üzleti modellelemek nem mindegyike szerepel az elemzési modellben, csak azok, amelyek a működést meghatározzák.

céljukban, tartalmukban különböznek, az eltérő munkaszakaszokban felállított osztálymodellekkel mást akarunk leírni, mást hangsúlyozunk. A fejlesztés kezdetén főként arra van szükség, hogy csak a probléma megértéséhez szükséges lényegi elemeket és azok kapcsolatait modellezzük, nem törődve azok megvalósításával.

Az elemzési szakasz végén – amikor a probléma már világos, és minden követelményt részletesen elemeztünk és definiáltunk – hasznos készíteni egy összefoglaló osztálydiagramot, amelyik részletesebb. Ez a modell már szoftverkomponenseket, interfész specifikációkat tartalmaz, de nincs elkötelezve egyik programnyelvi környezetnek sem.

A tervezési fázis végén rendelkezésre kell állni annak az osztálymodellnek, amelyik már pontosan egy bizonyos programfejlesztési környezethez kötődik (például a kódolás a Java programfejlesztői környezetben történik).

A fenti gondolatmenetre alapozva számos módszertan az osztálydiagramok három alapvető perspektíváját különbözteti meg, amelyekhez az osztálydiagramok modellelemeinek három különböző értelmezése kapcsolódik. Bár ez nem része az UML definíciójának, de hasznos abban az értelemben, hogy mindig csak az aktuális probléma megoldására engedi a fejlesztésben résztvevő szakembereket koncentrálni.

- *Konceptuális (lényegi – conceptual) perspektíva:* Ezek az osztálymodellek segítenek a probléma megértésben, a kommunikációban. A konceptuális modell csak a lényegi elemekre, az elemek közötti kapcsolatok leírására koncentrál. A diagramban ábrázolhatjuk a kapcsolat fokát, megadhatjuk az asszociáció nevét, a különféle asszociációs viszonyokat, mint öröklődés, aggregáció, kompozíció. A lényegi elemek megtalálásnak egyik hatékony módszere, amikor a felhasználóval folyatott megbeszéléseket rögzítő dokumentumokban megkeressük a főneveket. A modell készítésekor nem vesszük figyelembe azt a szoftvert, amelyik majd ezeket az elemeket implementálja.
- *Specifikációs (specification) perspektíva:* Átmenet a konceptuális és az implementációs megközelítés között. Az osztálymodellek ezen típusa már a fejlesztendő szoftverre koncentrál. A use case-ek fizikai megvalósításának módját írja le, a megvalósítás vázát adja. A modellelemekhez felelősségeket határoz meg, de ezt nem kód szinten teszi. A specifikációs modell az alkalmazás struktúrájára, felépítésére összpontosít. A fejlesztendő szoftver funkcionalitását tekintve bonyolult, elsőként célszerű

azt logikus részekre bontani. A specifikációs aspektusban készített modell a funkcionális felosztás eredményeként keletkezett szoftverkomponenseket, azok közötti kapcsolatokat (interfész), a kapcsolódás módját hangsúlyozza. Emiatt ebben a megközelítésben nem igazán osztályokról, hanem típusokról beszélünk. (egy Order-nek lesz felelőssége, hogy megmondja melyik Customer-ért felelős – az Order és a Customer közötti kapcsolatra koncentrál).

- *Implementációs (implementation) perspektíva:* Ezek az osztálymodellek már valódi, a későbbi rendszerben megvalósítandó osztályokat írnak le konkrét programnyelvi környezetben.

Az üzleti modellezők főként a valós elemeket leíró konceptuális modellt készítenek. Az elemzők, tervezők szoftverkomponensekre koncentrálnak, a fejlesztésnek ebben a szakaszában a kidolgozás részletezettségétől függően a specifikációs vagy az implementációs perspektíva érvényesül.

4.2. Objektum, osztály

Az *objektumok* a valós világ elemeit reprezentálják, modellezik. Például ezek a valós elemek a fizikai világban megjelenő konkrét egyedek, diszkrét entitások, amik lehetnek személyek, fizikai berendezések, elvont fogalmak (pl. kamatláb) stb. Az objektumok a valós elemekhez hasonló jellemzőkkel bírnak. Tulajdonságaik vannak, ismert az állapotuk, leírható a viselkedésük.

Az objektum állapota

Az objektumok életük folyamán a környezetükben levő más objektumokkal kapcsolatba kerülhetnek, egymásra hatással lehetnek. Az objektumok az őket ért hatásokra adott módon reagálnak, aminek eredményeként különböző állapotokba kerülhetnek. A pillanatnyi állapotot az adott pillanatban felvett tulajdonságértékek és az objektumnak a környezetében levő más objektumokkal megvalósított kapcsolatai határozzák meg.

Mit takar az objektumok viselkedése?

Az objektumokat különböző hatások érhetik, másrészt az objektumok maguk is hatással lehetnek más objektumokra. Az objektum viselkedése nem más, mint az általa végrehajtott tevékenységsorozat, reakció az általa előidézett vagy az őt ért hatásra. Az objektumok egymásra üzenetek formájában fejtik ki hatásukat, viselkedésük egyik kifejezési módja a kommunikáció. Az objektumok önálló, diszkrét entitások, ezért a környezetükkel

folytatott kommunikáció csak bizonyos módszerek által alkotott felületen keresztül engedhető meg (egységbezárás[11] fogalma), ezáltal a rendszer áttekinthetőbb és könnyebben módosítható lesz.

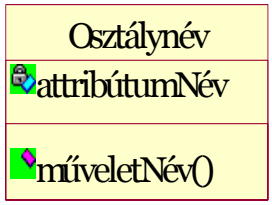
4.2.1. Az osztály

Az objektumok információt tárolnak (tulajdonságaikat leíró attribútumok formájában) és kérésre feladatokat ún. metódusokat hajtanak végre. Az objektum tehát adatok és metódusok összessége.
Az objektumokat a modellben az osztály modellelem írja le. Az *osztályok* az azonos tulajdonságokkal (attribútumok), viselkedéssel (metódusok) és kapcsolatokkal rendelkező objektumok csoportjának, halmazának leírására szolgálnak. Az objektum az osztály konkrét megjelenési formája, egyedi példánya (instancia). Az osztály tehát attribútumok és metódusok (függvények, eljárások) gyűjteménye, a közös tulajdonságokkal és viselkedéssel rendelkező objektumokat írják le.

A fejlesztési munka végső célja a rendszert felépítő osztályok, és az osztályok közötti kapcsolatok feltárása, majd az osztályok forráskódhoz rendelése és kódolása.

Az osztályt az UML-ben egy három részre osztott téglalap modellezi (lásd 4.1. ábra):

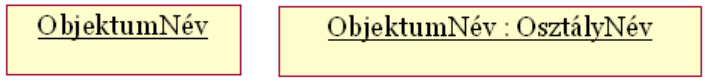
- az osztály megnevezését a felső részben adjuk meg,
- a középső részben találhatók az attribútumok (tulajdonságok),
- az alsó rész a műveleteket tartalmazza.



4.1. ábra. UML osztályszimbólum

[11] Egy tetszőleges objektum tulajdonságadatai csak az adott objektum metódusaiban láthatók, kívülről más osztály metódusaiból csak azok az elemek érhetők el, amelyeket az osztály engedélyez.

Az UML-ben a három részre osztott téglalap alapértelmezésben osztályt szimbolizál. Ha az osztály egy konkrét példányát, vagyis pontosan egy objektumot akarunk modellezni, annak jelölésére az osztályhoz hasonló téglalap szimbólumot kell használni[12]. A 4.2. ábra jól illusztrálja, hogy az objektumot az osztálytól az különbözteti meg, hogy az objektum neve alá van húzva. Lehetőség van csak az objektum nevét megadni az ObjektumNév szintaxissal, vagy a téglalapban az ObjektumNév:OsztályNév szintaxissal jelölhetjük azt is, hogy az objektum pontosan melyik osztályhoz tartozik, melyik osztály példánya.



4.2. ábra. Objektumok jelölése

Az osztálynév megválasztása

Az osztály elnevezésekor arra kell ügyelni, hogy a választott név tisztán, egyedien jellemezze az osztályt. A név adja az osztály identitását. A választott név lehetőleg nagybetűvel kezdődjön, célszerűen egyes számú főnév legyen. Ha az osztálynév több szóból áll, célszerű a szavakat nagybetűvel kezdeni, vagy aláhúzást használni a szavak között.

4.2.2. Attribútumok

Az objektumok meghatározott tulajdonságokkal, jellemző sajátosságokkal ún. *attribútumokkal (attribute)* rendelkeznek. Az attribútum-érték az attribútum egy konkrét előfordulása, egy adott pillanatban felvett állapot, az objektumpéldányban tárolt elemi adat. Az attribútum elnevezésekor a választott név mindig utaljon az általa hordozott információtartalomra. Az attribútumokhoz az attribútum nevének megadásán túl további információk rendelhetők. Az attribútumok specifikációjának általános formája:

[láthatóság] név [: típus] [= kezdeti érték] [{jelleg}]

A láthatóság kifejezi, hogy az objektum különböző tulajdonságai, metódusai mennyire fedhetők fel a külvilág számára. Az UML az osztályjellemzőkhöz (attribútum, művelet) három láthatósági szintet javasol:

[12] Az UML-ben az aktivitási, a szekvencia és az együttműködési diagramok mindegyike tartalmaz objektumokat.

- a + public: a jellemző interfészen keresztül tetszőlegesen minden osztály által elérhető, nincs szükség metódusra az eléréséhez,
- a − private: csak az osztály látja, csak saját objektumon belül látszik.
- a # protected: a jellemző csak saját objektumból és az utódokból látszik, csak az adott osztályon érhető el, a gyermek (leszármazott) és a barát (friend[13]) osztályok látják.

A típus egyszerű adattípusokat jelent. A kezdeti érték az objektum készítésekor beállítandó értéket jelenti. A jellemzők speciális tulajdonságokat mutatnak (pl. frozen).

Az attribútumok meghatározásakor figyelmesen kell eljárni, mert egyes adatok az elemezés/tervezés későbbi szakaszaiban maguk is objektumoknak minősülhetnek (pl. lakcím vagy dátum). Az ilyen attribútumok számára a modellben külön osztályt kell definiálni.

4.2.3. Műveletek

A *művelet (operation)* az osztály által nyújtott szolgáltatás. Feladat, tevékenység, amit az osztály végre tud hajtani. Az osztályok által nyújtott szolgáltatásokat elsősorban eseménykövetési diagramok üzenetei alapján azonosítjuk.

A művelet megvalósítása a metódus. Egy osztály minden konkrét objektuma azonos műveletekkel rendelkezik. A metódusok segítségével végzünk műveleteket a tulajdonságokon, ill. módosíthatjuk azokat.
A műveleteket:

- a művelet jellegének, nevének,
- a paraméterek nevének és típusának,
- a visszatérési értékeknek és típusuknak, valamint
- az alapértelmezett értékeknek a megadásával jellemezzük.

A műveleteknek két nagy csoportját különíthetjük el. A módosító műveletek azok a műveletek, amik megváltoztatják az adott objektum állapotát, ezért végrehajtásukra mindig figyelni kell. A lekérdező műveletek csak paraméterértéket kérnek más objektumoktól, nem változtatják meg a lekérdezés pillanatában megfigyelt állapotot. Az utóbbiak további jellemzője, hogy tetszőleges sorrendben hajthatók végre. A műveletek ezen két

[13] Egy osztálynak barátja (friend) egy olyan metódus vagy akár teljes osztály, amely hozzáférhet az adott osztály minden – ill. deklarációtól függően néhány – mezőjéhez és metódusához.

csoportját az UML szintaxis nem különbözteti meg, ezért ezt célszerű valamilyen módon a nevükben jelezni.

A műveletek specifikációjának általános formája:

[láthatóság] név [(paraméter lista)] [: visszatérési érték típusa] [{jelleg}]

4.2.4. Asszociáció

Az osztályok szolgáltatásaikat legtöbbször csak más osztályokkal együttműködve tudják biztosítani. Ez a megszorító jellegű állítás azt jelenti, hogy az osztályoknak egymással kapcsolatot kell létesíteni, egymással kapcsolatban kell lenni.

Az *asszociáció (association)* az osztályok objektumai közötti kapcsolat leírása, absztrakciója. Az asszociációt, mint viszonyt megtestesítő fogalmat az osztályok közötti viszonyokra értjük. A kapcsolat fogalmat az objektumok közötti viszonyra értelmezzük.
Az osztályok közötti asszociációs viszonyokat számos paraméterrel jellemezhetjük:

- Két osztály vagy osztályok közötti kapcsolatot a *két osztályt összekötő vonal* reprezentálja.
- Az *asszociációhoz név rendelhető:* a nevet 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 levő osztályoknak az adott asszociációban betöltött szerepére vonatkoznak. A szerepek definiálásával párhuzamosan általában megadjuk a kapcsolat (asszociáció) irányát. A szerepeknek az osztályokhoz rendelése az attribútumokhoz hasonló funkciót töltenek be.
- A *kapcsolat fokának* megadásával jelölhetjük, hány objektum vehet részt az asszociációban *(multiplicitás)*[14]: a multiplicitás kifejezi, hogy az egyik osztály egy objektumához a másik osztályból hány objektum tartozik, vagyis kifejezi, hogy az osztályok objektumai milyen számosságban kapcsolódnak egymáshoz. A multiplicitás jelölése az UML-ben:
  1: az adott osztály egy objektumához a másik osztályból pontosan egy objektum kapcsolódik
  0..1: 0 vagy 1 objektum kapcsolódik, opcionális kapcsolat

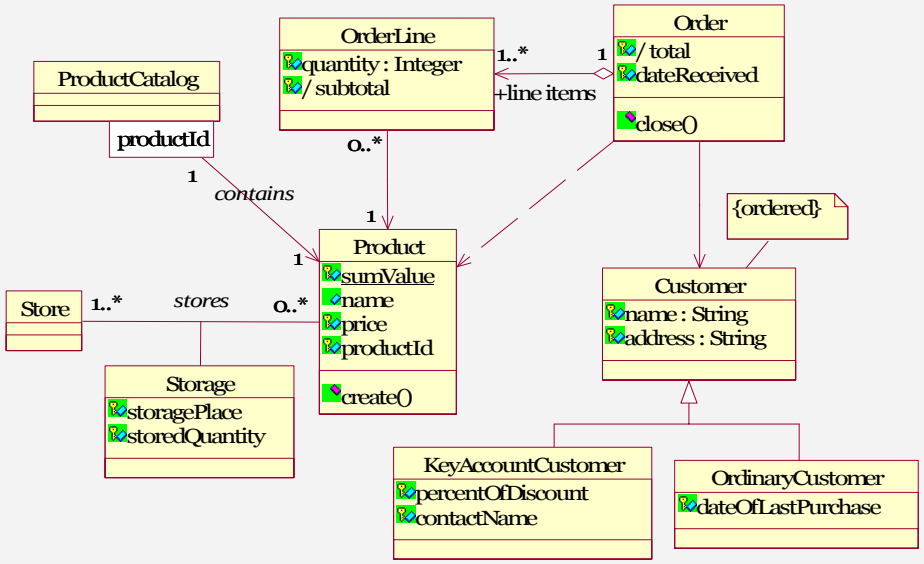
[14] A multiplicitáshoz kapcsoló másik fontos fogalom a kardinalitás, a számosság fogalma. A kardinalitás megadja az adott osztályban specifikálható előfordulások minimális, illetve maximális számát.

0..*: opcionális többes kapcsolat
1..*: 1 vagy többes kapcsolódás, kötelező többes kapcsolat
22..44: egy objektumhoz [22,44] zárt intervallumnak megfelelő számú objektum kapcsolódhat
9: ebben az esetben pontosan 9 objektum kapcsolódik a megjelölt osztály egy objektumához
2 és 13 értékeken kívül bármilyen számosság lehetséges: a számosság akár *szövegesen* is megadható.

- A *navigálhatóság iránya,* az asszociáció bejárhatóságának iránya: a kapcsolatok mentén kommunikáció zajlik, amely lehet egy 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.
  - *Megszorítás*[15] *(constraint):* korlátozás, ami az egyes modellelemek között definiált asszociációs viszony finomítására szolgál. A megszorítások a modellben kapcsos zárójelek {} között szerepelnek. Klasszikus példa a megszorításra, amikor azt akarjuk hangsúlyozni, hogy az asszociációban az egyik objektummal kapcsolatban levő objektumok rendezve legyenek elérhetőek, láthatóak. Az elemek rendezettségére előírt megszorítást az UML modellben {ordered} formában adjuk meg.

[15] Megszorítások megfogalmazására az UML specifikáció részét képező OCL (Object Constraint Language) nyelv ad lehetőséget. Az OCL segítségével további szemantikai értelmezéssel lehet finomítani a modellt.

Osztálydiagram – részlet



A kapcsolatok implementálása

Az üzleti modellezés során az osztályok közötti kapcsolati viszonyok elég egyértelműek. Egyik osztály vagy objektum kapcsolatban van a másik osztállyal vagy objektummal. Az osztályok tulajdonságokkal jellemezhetők, meg tudjuk fogalmazni, hogy a két modellelem között milyen fokú kapcsolat van, de nem térünk ki, és nem is kell kitérni a kapcsolat megvalósításának, az implementálásnak a kérdéseire. A kapcsolatokat csak fogalmi szinten értelmezzük.

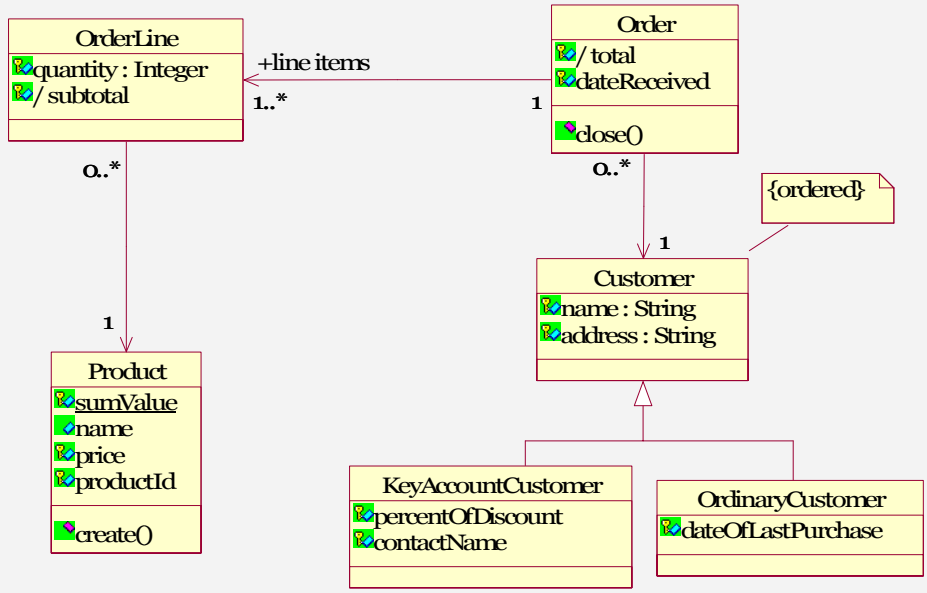
Az elemzés/tervezési szakaszban az osztályok specifikációját tovább finomítjuk. Véglegesítjük az attribútumok listáját, a specifikációt pontosítjuk az attribútumokra megadható jellemzőkkel (láthatóság, adattípus, kezdőérték stb.) kiegészítve. Az eseménykövetési diagramok segítségével az objektumok üzeneteiből meghatározzuk az osztályok által nyújtandó szolgáltatásokat, műveleteket. Az eseménykövetési diagramban az objektumok közötti üzenetváltások világosan mutatják az osztályok közötti asszociáció szükségességét. Ugyanis ha két objektum között kapcsolati viszony van, az azt is jelenti, hogy az osztályaik között asszociációnak kell lenni. A fejlesztésnek ebben a szakaszában az asszociáció már nem csak elvi/fogalmi

szinten kerül megfogalmazásra, mint az üzleti modellezéskor, hanem az asszociációval két osztály közötti felelősségeket határozunk meg.

Az objektumok közötti üzenetváltások azonban még mindig nem jelentenek biztos információt a kapcsolat implementálására. A kapcsolatok kódolására az implementációs szakaszban kerül sor.

A konceptuális modell

A konceptuális modell csak a lényegi elemekre, az elemek közötti kapcsolatok leírására koncentrál. A diagramban ábrázolhatjuk a kapcsolat fokát, megadhatjuk az asszociáció nevét és különféle asszociációs viszonyokat ábrázolhatunk.



A specifikációs modell

A specifikációs szintű modell csak felelősségeket definiál, de nem kód szinten: egy Order-nek lesz olyan felelőssége, hogy megmondja melyik Customer-hez tartozik. A specifikációs modell készítésekor azt vizsgáljuk meg, hogy az egyik objektum a másik objektummal kapcsolatban a rendszer által megvalósított szolgáltatás végrehajtása érdekében mit csinál, és mit ad vissza. Esetünkben tehát a specifikációs modell azt fejezi ki, hogy

egy Order-nek az a felelőssége, hogy megmondja pontosan melyik egyedi azonosítójú Customer tartozik hozzá.

```
class Order
{
  public Customer customer();
  public Enumeration orderLines(); //Az Order osztálytól
            // lekérdezhetők hozzátartozó orderLine-ok.
}
```

Az implementációs modell

Implementációs modell esetén már a felelősségek pontos megvalósítását is definiáljuk. Az implementációs osztálydiagramban azt írjuk le a példában, hogy minden Order típusú objektum tartalmaz egy pointert, amelyik pontosan egy adott Customer objektumra mutat.

Az Order osztály definíciója

```
class Order  // Az Order osztály definíciója
{
public:
  Order();
  virtual ~Order();

  long lPrice;
  long lNumber;
  Customer* m_pTheCustomerOfTheOrder;
  Customer* getTheCustomerOfTheOrder();
};
```

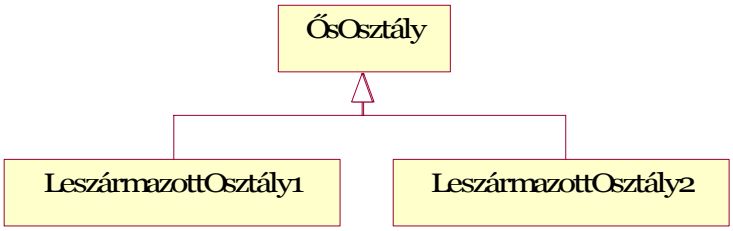
A getTheCustomerOfTheOrder() függvény megvalósítása:

```
Customer*  Order::getTheCustomerOfTheOrder()
{
  return m_pTheCustomerOfTheOrder;
};
```

4.2.5. Az öröklődés

Az *öröklődés* az OO-szemlélet egyik legfontosabb elve. Az öröklődés a modellelemek (use case-ek, osztályok) között értelmezett sajátos viszony. Lehetővé teszi, hogy a modellelem sajátjaként kezelje a nála általánosabb szinten definiált modellelem jellemzőit (a modellelemek jellemzőihez beállított láthatóság alapján). Az osztályok között értelmezett öröklődési viszonyban az utód osztály (leszármazott) sajátjaként kezeli a nála általáno-

sabb/magasabb szinten levő osztály (ős osztály) attribútumait és műveleteit. Mint ahogy azt a 4.3. ábra is illusztrálja, az UML-ben az öröklődés jele egy üres háromszögben végződő nyíl, a háromszög csúcsa az ős osztálynál található.



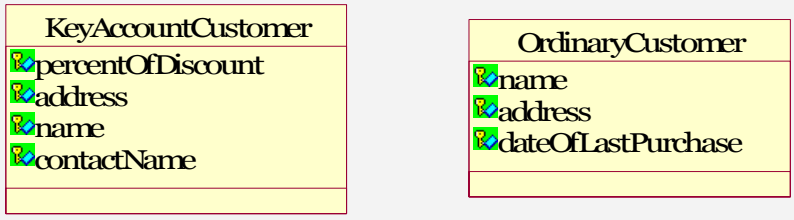
4.3. ábra. Öröklődési viszony

Öröklődési viszonyt kétféleképpen definiálhatunk a modellünkben:

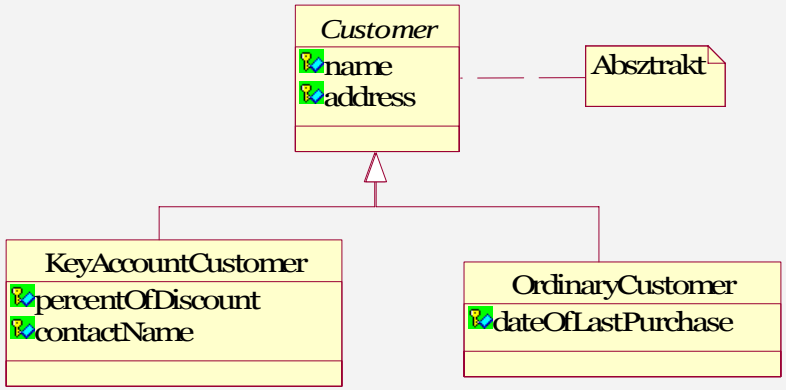
- *Általánosítás (generalizáció – generalization):* A különböző objektumok sokszor tartalmaznak közös jellemzőket (adatok, műveletek). Az általánosítás az a folyamat, amikor ezeket a közös jellemzőket kiemeljük egy ős osztályba (alulról felfelé). Az általánosítás eredményeként létrejön egy általános/közös sajátosságokat tartalmazó ős vagy szülő osztály, amelyhez tartozik egy vagy több speciális tulajdonságokkal rendelkező al vagy gyerek osztály. Az ős osztály általában absztrakt osztály, olyan osztály, aminek nincsenek példányai. Az ős osztály szerepe ugyanis csak a közös sajátosságok egy helyen történő tárolása, segítségével jelentősen egyszerűsödik a modellünk felépítése. A generalizáció használatának célja a kód újrafelhasználási fokának emelése a rendszerben.
- *Specializáció (pontosítás – specialization):* A specializáció az a folyamat, amikor meglevő osztályokból származtatott osztályokat képezünk finomítással (fentről lefelé). A finomítás célja az osztályok specifikációjának pontosítása, az objektumok egyedi jellegének megerősítése az egyedi jellegre utaló jellemzők definiálásával. A modellben a származtatott osztályok keresése során feltételezzük, hogy az osztályok általánosak (gyűjtőfogalmat jelenítenek meg). A specializáció során azt vizsgáljuk, hogy újabb attribútumok és műveletek hozzáadásával milyen, a feladat szempontjából értelmes osztályok határozhatók meg. Az attribútumokat és az asszociációkat mindig a legáltalánosabb osztályhoz rendeljük. Ebben az esetben az ős osztály nem feltétlenül absztrakt.

Öröklődési viszony – Általánosítás

A rendszernek külön kell kezelni a nagyvásárlókat a többi ügyféltől. Vannak olyan jellemzők (name és address attribútumok), amelyek mindkét ügyféltípusnál azonosak, azokat érdemes egy ős osztályba kiemelni.

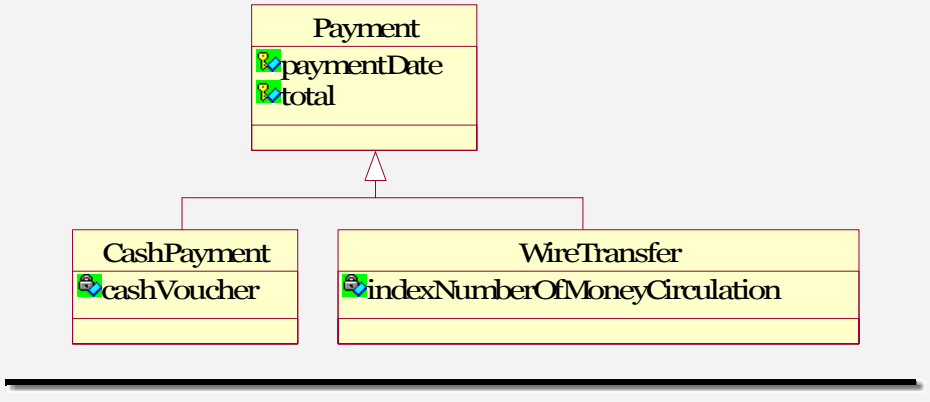


Az általánosítás eredményeként létrehozhatjuk a Customer absztrakt osztályt, amelynek nem lehetnek példányai. A Customer osztály csak a közös jellemzőket tárolja, egyszerűsíti a modellt. A modellünk így kiemeli azt a tényt, hogy a különböző típusú ügyfeleink (KeyAccountCustomer, OrdinaryCustomer) azonos tulajdonságai a név (name) és cím (address) tulajdonságok. Az általánosítás azért teszi hatékonyabbá a rendszert, mert a kiemelt tulajdonságokat és műveleteket csak egyszer kell leírni. Esetünkben ez azt jelenti, hogy a name és az address tagváltozókat csak egyszer kellett definiálni, ill. ha később ezeket kezelő függvényeket definiálunk, akkor ezt is csak a Customer osztályban kell egyszer megtenni.

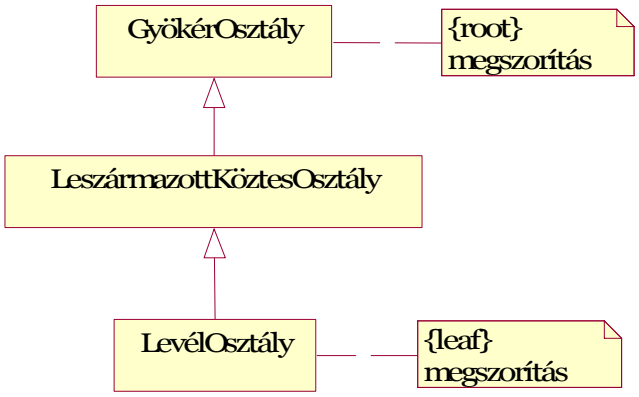


Öröklődési viszony – Specializáció

A fizetésekkel kapcsolatos információkat – mint a kifizetés dátuma (paymentDate), vagy a végösszeg (total) – a Payment osztályban definiáltuk. A nagyvásárlókra és a magánszemélyekre más fizetési feltételek vonatkoznak. A nagyvásárlók átutalással teljesítenek, a többi ügyfél a helyszínen fizet, készpénzben vagy bankkártyával. Ha a Payment osztályhoz még hozzárendeljük a kiadási pénztárbizonylat (cashVoucher) attribútumot, akkor a CashPayment osztályt kapjuk. A pénzforgalmi jelzőszám (indexNumberOfMoneyCirculation) az átutalásos fizetésekhez (WireTransfer osztály) kapcsolódó információ.



Az osztályok közötti öröklődési viszonyban *öröklődési hierarchia,* ún. *öröklődési lánc* is kialakulhat. Öröklődési lánc leírására a 4.4. ábra diagramja mutat példát. A hierarchiában azokat az osztályokat, amik nem örökölnek másoktól sajátosságokat – vagyis nincsenek szüleik – *gyökérnek (root)* nevezzük. Ezt a modellben a {*root*} *megszorítással* jelöljük, amit az osztály neve alá vagy megjegyzésbe írunk. Azokat az osztályokat, amik az öröklődési lánc legalsó szintjén helyezkednek el – vagyis nem örökítenek tovább jellemzőket – *levél (leaf)* osztályoknak nevezzük. Ezt a tényt a modellben is rögzíthetjük megszorításként, ilyenkor a {*leaf*} *megszorítás* az osztály neve alá vagy megjegyzésbe kerül.



4.4. ábra. Öröklődési lánc

Az öröklési viszony lehet:

- egyszeres: egy osztálynak csak egy őse lehet.
- többszörös: a kapcsolatban egy osztálynak több szülője is lehet.
- többszintű: egy osztálynak legalább egy őse és több leszármazottja van. Ez az osztály az öröklődési lánc köztes szintjén helyezkedik el.

4.3. Speciális fogalmak, asszociációs viszonyok

Az osztályok leírását, az osztályok között értelmezett viszonyokat tovább finomíthatjuk. Az UML a fent említett, az osztályokat leíró alapvető elemek (attribútum, műveletspecifikáció, asszociáció, öröklődés) mellett még számos fogalmat, elemet és jelölést ajánl az osztálymodell pontosabb leírására.

4.3.1. Aggregáció és kompozíció

Többször találkozunk olyan esettel, amikor egy objektum több másik objektumból épül fel, vagyis 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. A rész-egész viszonynak kétféle formája létezik (összetett objektum kétféleképpen keletkezhet):

- *Aggregáció (aggregation):* a rész-egész viszony gyengébb formája. A tároló objektum (az egész osztály) és az azt felépítő részobjektumok (részelemek) elkülönülnek. Aggregációról csak akkor beszélünk, ha a rész-

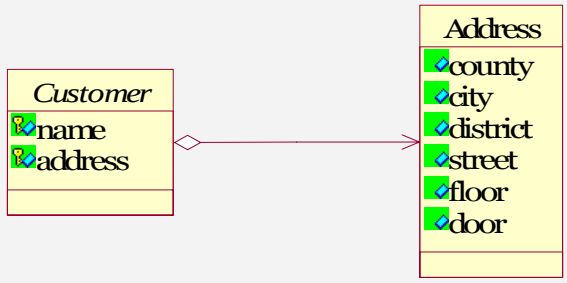
oldal nem értelmes az egész nélkül. Abban az esetben, amikor az osztály a másik oldal nélkül is értelmes, akkor a két modellelem között egyszerű asszociációs viszony van.

- *Kompozíció (composition):* 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 megszűntével a rész is megszűnik. Az egész oldal számossága csak 1 lehet.

Az UML az aggregációt és a kompozíciót más szimbólummal jelöli. Az aggregációt egy rombusz, a kompozíciót egy sötétített rombusz szimbolizálja, a rombuszok a tartalmazó osztály felöli oldalon helyezkednek el.

Aggregáció

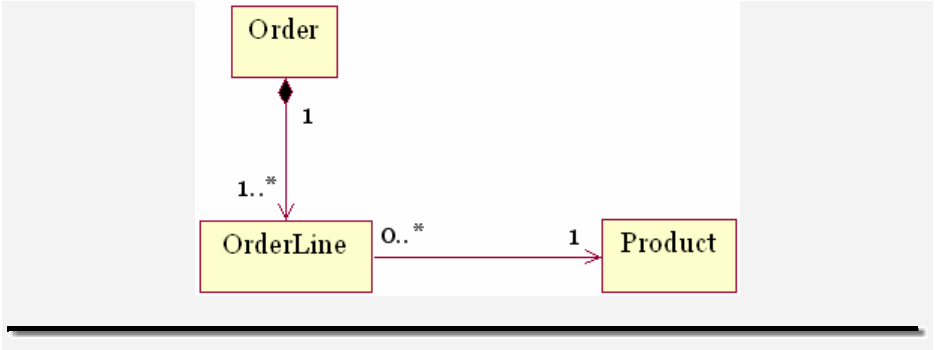
Rendszerünkben az ügyfelek adatainak tárolásánál el kell tárolnunk az ügyfelek címét. Ha a címet a megye, a város, a kerület, az utca, a házszám, az emelet és az ajtó jellemzők megadásával akarjuk azonosítani, akkor a címet (address) érdemes külön egy cím (Address) osztályban definiálni. Az eljárás eredményeként a Customer és az Address osztályok között aggregációs viszonyt értelmezünk.



Kompozíció

A rendszernek kezelni kell az egyes vásárlásokhoz/megrendelésekhez (Order) tartozó termékeket. Az adott Order-hez tartozó Product-okat az OrderLine osztályban tároljuk.

A rendelés (Order) rendelési tételekből (OrderLine) áll, a rendelési tétel csak a rendeléssel együtt értelmes. Az Order és az OrderLine között az aggregáció erősebb változatát, a kompozíciót értelmezzük.



4.3.2. Minősített asszociáció

Minősített asszociációról akkor beszélünk, ha két osztály között asszociációs viszony áll fenn és az egyik osztály egy tulajdonságot, attribútumot (*minősítő* – qualifier) használ fel kulcs gyanánt az asszociáció másik oldalán levő objektumok elérésére, azonosítására. Azt az osztályt, amelyik a másik osztály objektumait akarja a minősítő attribútum felhasználásával elérni, célosztálynak, a kapcsolatban részt vevő másik osztályelemet forrásosztálynak nevezzük. A rendszer működése során a meghatározott minősítő attribútumok (kulcsértékek) alapján kérdezhetők le az egyes elemek. Implementációs szinten a minősített asszociáció rendszerint a célosztály objektumaiból képzett indexelhető tömb, vagy valamilyen más, kulcs szerinti keresésre alkalmas összetett adatstruktúra (pl. hashtábla) megjelenését eredményezi a forrásosztályban.

Minősített asszociáció

A cég termékeit (Product) három különböző kategóriában kínálja ügyfelei részére. Az egyes kategóriákba eső ugyanazon termékek minőségben és árban térnek el egymástól. Minden kategóriához egyedi termékkatalógus (ProductCatalog) készül. A modellben a ProductCatalog tartalmaz (contains) kapcsolatban áll a Product osztály objektumaival, amit az osztályok közötti asszociáció megnevezése szemléltet. A ProductCatalog és a Product osztály elemei között a multiplicitás 1..*, azaz a ProductCatalog egy objektumához a Product-ból egy vagy több elem tartozik.

Ebben a kapcsolatban a Product osztály productID attribútuma kulcsjellemzőként, ún. minősítőként szolgál a ProductCatalog számára. A productID a ProductCatalog osztállyal tartalmaz kapcsolatban levő Product-ok közül pontosan meghatároz egy objektumot. A minősített asszociáció eredményeként a két osztály között értelmezett multiplicitás is megváltozik, a minősítő attribútumot tartalmazó osztálynál a többes kapcsolat egyre változik, amit úgy értelmezünk, hogy a minősítő attribútum egyértelműen meghatározza, hogy mely objektum esetén jön létre az asszociációként jelölt kapcsolat.

4.3.3. Asszociációs osztály

Ha a modellben két osztály közötti kapcsolat jellemzésére, ill. annak létrejöttéhez önálló tulajdonságok, esetleg metódusok tartoznak, akkor érdemes ezeket egy külön osztályba specifikálni. Az így keletkezett osztályt *asszociációs osztálynak (association class)* nevezzük. Ilyen megoldásra akkor van szükség, ha két osztály elemei között több-több jellegű leképezést akarunk megvalósítani, de az egymáshoz rendelt párokhoz, ill. az asszociációval jelzett kapcsolat létrejöttéhez még további információkat is hozzá akarunk rendelni, amelyek igazából egyik osztályhoz sem tartoznak kizárólagos módon.

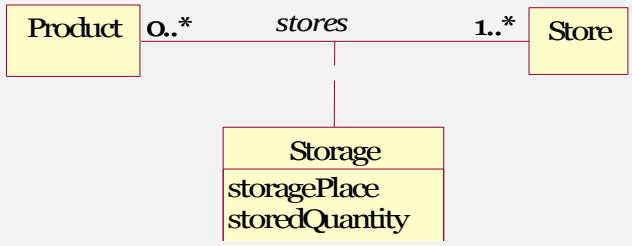
Az UML-ben az asszociációs osztályt és a kapcsolatot szaggatott vonal köti össze. Az osztályoknak ezt a speciális változatát főként az elemzési fázisban alkalmazzuk. Látványosan érzékeltethetjük vele az osztályok közötti kapcsolat fontosságát, jellegét. A tervezési és az implementációs modellben ezek az osztályok normál osztályokká szerveződnek.

Asszociációs osztály

A cégnek két raktára (Store) van. Egy raktárban többféle terméket tárolnak (az asszociáció megnevezése: stores), egy adott termékből mindkét raktárban lehet készlet. Minden terméknek pontosan meg van határozva adott raktáron belüli tárolási helye. A rendelések összeállításakor a rendszernek pontosan meg kell tudni adni:

- a termék raktáron belüli tárolási helyét (storagePlace), vagyis a termék melyik raktárban hol található.
- Továbbá ismerni kell a raktári készletet (storedQuantity), azaz adott termékből az adott raktárban hány darab található.

A storagePlace attribútumot a Product osztálynál nem célszerű tárolni, mert nem tudja megmondani, hogy a termék melyik raktárban van. Ha a storedQuantity attribútumot a Store osztályban definiáljuk, ez a raktárban levő összes termék darabszámát adja vissza. Ha a Product osztályban tároljuk a storedQuantity-t, akkor az adott termék összkészletét kapjuk meg. A storagePlace és a storedQuantity attribútumok olyan információk, amelyek se nem a Store, se nem a Product osztályhoz nem rendelhetők kizárólagosan. Ezek az információk a kapcsolatra jellemző sajátosságok, amelyeket egy külön osztályban, az ún. Storage asszociációs osztályban érdemes definiálni.



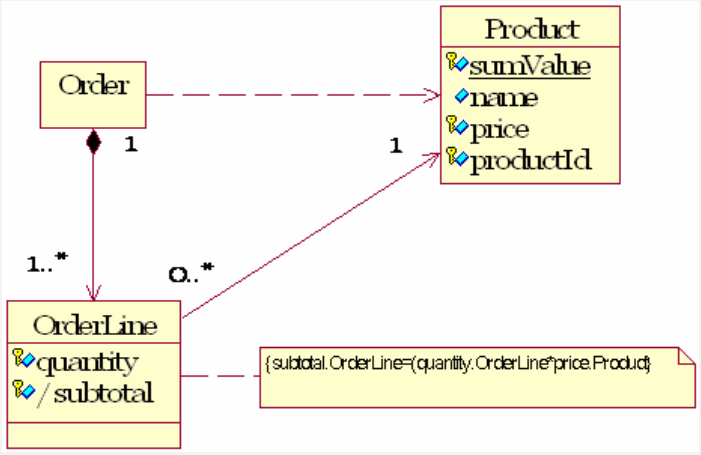
4.3.4. Származtatott elemek

Az UML-ben lehetőség van olyan elemek megadására, amelyek más, már a modellben definiált elemekből származnak, azokból számítódnak. A származtatásra leggyakrabban az attribútumoknál és az asszociációknál lehet szükség. A számított tulajdonságokat a tulajdonság neve előtt megje-

lenő per (/) jellel jelöljük. A számításhoz használt kifejezést kényszerként (megszorítás) kapcsos zárójelek {} között megadva írhatjuk le.

Származtatott elemek

Minden egyes megrendelés (Order) esetén a rendszernek meg kell tudni mondani az adott vásárláshoz tartozó végösszeget (total) és a rendelési tételenkénti részösszegeket (subtotal). A részösszeget a megvásárolt termékek darabszáma (quantity attribútum az OrderLine osztályban) és a termék ára (price attribútum a Product osztályban) szorzataként kapjuk meg. A végösszeg (total) önmaga is származtatott attribútum, a rendeléshez tartozó részösszegek szummája.



4.3.5. Sztereotípia

Az UML-ben minden modellelemhez, így az osztályokhoz is rendelhetünk sztereotípiát. A sztereotípia alkalmazása tehát nem korlátozódik csak az osztályokra, használatuk általános az UML modellezésben.

A *sztereotípia* a modellelemek tipizálása, minősítésre szolgál. Ha modellelemekhez sztereotípiát akarunk rendelni, akkor a sztereotípia nevét karaktersorának francia zárójelek közé tételével vagy szimbólummal, vagy a kettő egyidejű alkalmazásával adjuk meg.

A sztereotípia felhasználására vegyük példaként az elemzési munkaszakaszban az elemzési osztályok, ill. a belőlük példányosított objektumok minősítésére használt három típust.

Elemzési osztályok sztereotípiái

Az elemzési modellben az osztályok minősítésére a <<boundary>>, a <<control>> és az <<entity>> sztereotípiákat használhatjuk. A sztereotípusok meghatározását az interakciós diagramok készítésekor, elemzésekor lehet megtenni.

- A <<*boundary*>> sztereotípiával megjelölt osztályokat határ osztályoknak is nevezik. A határ szó az osztálynak a rendszerben betöltött feladatára, szerepére utal. Ezek az osztályok ugyanis a tervezett szoftverrendszer és a rendszer környezete közötti kommunikációt biztosítják, a rendszer és a környezete közötti kommunikációs kapcsolat megvalósításáért felelősek. A határ osztályok valósítják meg a leendő rendszerünk interfészeit a külvilág felé. Interaktív alkalmazás fejlesztése esetén például a határ osztályok objektumai reprezentálják a felhasználói felület elemeit. Biztonsági, riasztó (érzékelő funkcióval rendelkező) rendszerek fejlesztésekor <<boundary>> sztereotípussal látjuk el a rendszer külső környezetéből érkező jelek feldolgozására alkalmas osztályokat.
- A <<*control*>> sztereotípiával minősített vagy vezérlő névvel ellátott osztályok felelősek elsősorban a tervezett rendszer viselkedésének definiálásáért, koordinálják és vezérlik annak működését. A vezérlő objektumok jellemzően sok műveletet tartalmaznak, feladatuk más objektumok együttműködésének vezérlése. A <<control>> sztereotípiát például tranzakció-kezelést, erőforrás-kiosztást és hibakezelési feladatokat végrehajtó objektumok minősítésére használjuk.
- Az <<*entity*>> sztereotípiát entitás jellegű osztályok minősítésére használjuk. Az entitás osztályok jellemzően a rendszerben tárolt adatok tárolásáért felelősek. A tervezési modellben ezek az adatbázis tervezésekor fognak elsősorban fontos szerepet játszani.

Az objektumok általában mindhárom fenti jellegből tartalmaznak valamennyit, tiszta „profilú” objektum csak ritkán létezik. A megfelelő sztereotípus kiválasztásánál a meghatározó jelleget kell figyelembe venni.

Az elemzési osztályoknál a sztereotípusok használata megkönnyíti a diagramok összeállítását, értelmezését, elősegíti a szoftverrendszer rétegei-

nek (felület/prezentációs réteg, alkalmazáslogika, adattárolási logika) szétválasztását.

4.3.6. Absztrakt osztályok

Az *absztrakt osztályok (abstract class)* speciális osztályok, amelyeknek nem hozhatunk létre példányait. Absztrakt osztályok specifikálásakor az osztály nevét döntött betűkkel kell írni. Az absztrakt osztályból mindig származtatunk további osztályokat, amelyek öröklik az absztrakt osztály attribútumait, műveleteit és asszociációit.

4.3.7. Függőség

A *függőség (dependency)* az UML-ben két elem egymásra hatását fejezi ki. Két elem függőségi viszonya azt jelenti, hogy az egyik elem definíciójának (specifikációjának) változása a másik elem megváltozását okozhatja, eredményezi. A kapcsolatban az előbbi a független, az utóbbi a függő elem. A függési kapcsolatot irányított, szaggatott vonallal modellezzük. A nyíl iránya egyben meghatározza a függőség irányát. A nyíl a függő elemtől indul, és a felé az elem felé mutat, amitől az elem függ.

Függőségi kapcsolatot gyakran adatcsere jellegű kapcsolatok leírására használjuk. Ilyen kapcsolat lehet például a felhasználói felület egy ablaka, mely adatokat kér be a felhasználótól (átmenetileg létező objektum) és az adatokat egy szakterületi, perzisztens módon eltároló objektum között. Az ablak esetében ahhoz, hogy teljesíteni tudja funkcióját, szüksége van a tároló objektumra.

Függőségi viszonyt az UML bármely eleme között értelmezhetünk akár use case-ek, objektumok, komponensek, csomagok-elemek között. Általános tervezési elv, hogy célszerű a fejlesztés során a modelleket úgy felépíteni, hogy bennük a függőségek száma lehetőleg minimális legyen.

4.3.8. Osztályattribútum, osztályművelet

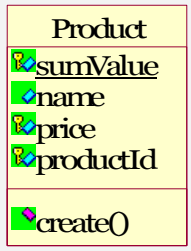
Az osztályok attribútumait és műveleteit különleges szereppel és jelleggel ruházhatjuk fel, ún. scope specifikációt rendelhetünk hozzájuk:

- A *classifier típusú attribútumok (class-scope attibute)* az osztály minden objektumában, vagyis instanciájában ugyanazt az értéket veszik fel.
- A *classifier típusú műveletek (class-scope operation)* minden objektumra, vagyis instanciára azonos módon lejátszódó műveletek.

A classifier típusú attribútumokat és műveleteket az UML aláhúzással jelöli.

Osztályattribútum, osztályművelet

A modellben osztályattribútum lesz a termékeket reprezentáló Product osztály sumValue attribútuma, ha ez az érték az összes termék értékét tárolja. A Product osztályban definiált objektumpéldányokat létrehozó create() művelet a Product minden objektumára azonos módon játszódik le, ezért ezt a műveletet a modellben osztályműveletként specifikáljuk.



4.3.9. Template (paraméterezett, minta) osztály

Komplex rendszerek fejlesztése esetén célszerű *template* (mintaként használt) osztályokat létrehozni, amelyek az osztályokat valamilyen logika szerint csoportba fogva paraméterként általános definíciókat tartalmaznak.

Ha a modellben ún. konténer (tároló) osztályokat definiálunk, akkor template osztályokkal adhatjuk meg bizonyos objektumok tárolásának és elérésének mintáját. Lista, ill. tömb osztályok esetén vannak olyan osztályok, amelyek működése független az osztály tagváltozóinak (az objektumok állapotát leíró adatszerkezet) típusától. Például a lista osztály implementációja független attól, hogy a lista int vagy double típust tárol. A típustól függetlenül lehet az új elem beszúrását vagy egy meglévő elem törlését implementálni.

Paraméterezett osztály

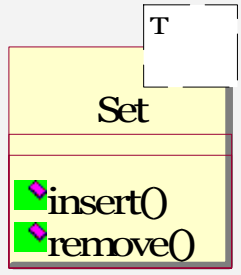
Set paraméterezett osztály definíciója

```
class Set <T> // Set paraméterezett osztály definíciója
{
  void insert {T newElement};
  void remove {T newElement};
}
```

Employee elemeket tartalmazó „Set” definiálása

```
Set <Employee> employeeSet;  // Emloyee elemeket
tartalmazó „Set” definiálása
```

Paraméterezett osztály UML-ben



4.4. A CRC-kártyák használata

Az 1980-as évek végén az amerikai Tektronix elektronikai cég kutatólaboratóriuma volt az objektumorientált technológia egyik legelismertebb központja. A Smalltalk programozási nyelv alkalmazásában értek el kitűnő eredményeket, az OO-fejlesztések terén nagyon sok alapvető elvet ott munkáltak ki először a világon. Közéjük tartozott két munkatárs, Ward Cunningham és Kent Beck.

Ők ketten dolgozták ki az ún. *CRC-kártyák (CRC cards)* használatának elvét. Az elnevezés eredete a következő: *CRC: Class – Responsibility – Collaboration* (Osztály – Felelősség – Együttműködés).

Cunningham és Beck nem diagramok alapján alakították ki fejlesztési modelljüket, hanem táblázatos beosztású lapokon (kártyákon) írták le az egyes osztályokat. Ebben a leírásban nem metódusokat és attribútumokat adtak meg, hanem az osztályokhoz rendelhető *felelősségeket.*

A felelősség itt valójában annak a magas szintű leírása, hogy mi a fő célja az illető osztálynak. Ebben a megközelítésben nem koncentrálunk arra, hogy milyen adataink vannak, hogy milyen folyamataink vannak. Ehelyett arra összpontosítunk, hogy minél pontosabban, minél egyértelműbben meghatározzuk egy osztály működésének a célját, mégpedig egy rövid, tömör leírás formájában. A leírásnak el kell férnie egy adott lapon, kártyán. Egy ilyen CRC-kártya tartalmát mutatjuk be a 4.5. ábrán.

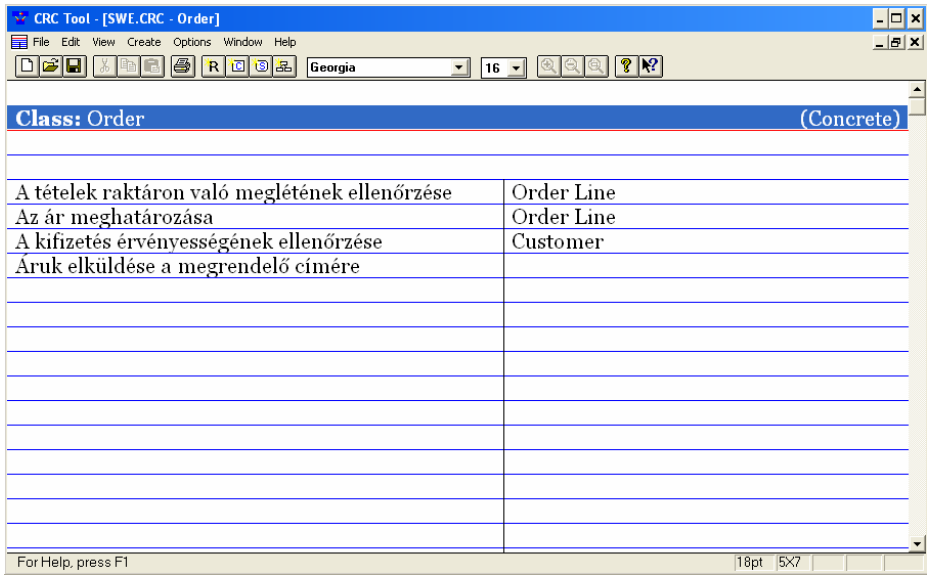
<Osztály neve>

Order	
A tételek raktáron való meglétének ellenőrzése	Order Line
Az ár meghatározása	Order Line
A kifizetés érvényességének ellenőrzése	Customer
Áruk elküldése a megrendelő címére	

<Felelősség> <Együttműködés>

4.5. ábra. CRC-kártya alkalmazása

A kártyák elkészítésére speciális szoftver eszközök is rendelkezésre állnak. Ezt demonstrálja a 4.6. ábra, ugyanarra a példára.



4.6. ábra. CRC-kártya készítése CRC-eszközzel[16]

[16] A CRC-eszköz az Ixtlan Software terméke.

A CRC-ben a második C az *együttműködésre* utal. Az együttműködő fél egy másik osztály, amely egy adott felelősséghez kötődik. Egy felelősséghez tehát azt kell megadni, hogy melyik osztály az, amelyre, vagy melyik osztályok azok, amelyekre szükség van ahhoz, hogy az adott felelősséget teljesíteni lehessen. Ezekkel az osztályokkal kell tehát dolgozni az adott felelősség esetében. Mint látható, ez a megoldás tulajdonképpen magas szinten tünteti fel az osztályok közötti kapcsolatokat.

A CRC-kártyák igazi értéke abban áll, hogy a fejlesztők közötti hasznos és élénk megbeszélést segítik elő. Akkor a leginkább előrevivők, amikor egy use case-en megyünk keresztül, és azt nézzük meg, hogy a use case-t hogyan fogják az egyes osztályok megvalósítani, leképezni. A fejlesztők azt nézik a kártyákon, hogy az osztályok hogyan működnek együtt a use case-en belül. Ebben a folyamatban azt alakítják ki, hogy mik lesznek az elvárható felelősségek, amiket végül is rá fognak írni a kártyákra.

A felelősségekről való vita és döntés azért fontos, mert elviszi a fejlesztőket attól, hogy az osztályok belső világával foglalkozzanak, azok részletes működésével, adataival, és e helyett inkább az osztályok magasabb szintű működésére tudjanak koncentrálni. Ez a tervezési szemlélet azt segíti elő, hogy a jól kiérlelt, helyes döntések magasabb szinten szülessenek, és a döntések részletes megvalósításával csak később kelljen foglalkozni. Nincs rosszabb annál ugyanis, mint ha egy felső szinten hozott rossz döntést kell az alsóbb szinteken végigvinni, kivitelezni.

Fowler gyakori hibának észleli a fejlesztők körében azt, hogy hosszú listákat állítanak össze az alsó szintű felelősségekből. Szerinte ez a lényeglátás hiányát tükrözi. A felelősségeknek rá kell férniük egyetlen kártyára. Ugyanakkor egy kártyán nem érdemes három-négynél több felelősséget feltüntetni. Ha ennél többre volna szükség, akkor érdemes megfontolni azt, hogy egy osztályt több részre, több osztályra bontsunk, ami a felelősségek tisztább szétosztását segítené elő.

Mikor használjunk CRC-kártyákat?

A CRC-kártyákat elsősorban az objektumorientált analízis munkafolyamataiban használják. Ilyenkor a fejlesztő team mindegyik osztályra elkészíti a megfelelő kártyát.

Ezt a megközelítést a későbbiek folyamán kibővítették. Először is, egy CRC-kártya gyakran tartalmazza az osztály metódusait és attribútumait, azáltal, hogy pontosítva adja meg egy „felelősség” leírását. Másodszor pedig, a kártyák használata helyett ma már számítógépes segédeszközöket

vesznek igénybe, ahol webes környezetben kommunikálnak egymással a team tagjai, és ezáltal jön létre egy-egy osztály végső CRC-reprezentációja. Ezt a folyamatot CASE-eszközök is segítik, mint például a *System Architect,* amely külön komponenseket tartalmaz a „kártyák" számítógépes előállításához.

A CRC-kártyák hasznossága abban áll, hogy a team-tagok közötti interakció során kiderülhet, hogy egy osztályban hibás vagy hiányzó mezők vannak, akár metódusok, akár attribútumok tekintetében. Az osztályok közötti viszony is világosabbá válik a kártyahasználat során. Igen jónak bizonyult az a megoldás, amikor a kártyákat szétosztják a fejlesztőcsoport tagjai között, akik ezután kimunkálják a saját osztályuk felelősségeit. Így például valaki azt mondhatja, hogy „én vagyok a *Date (Dátum) osztály,* és az én felelősségem az, hogy új dátum-objektumokat hozzak létre". Egy másik team-tag erre felvetheti, hogy neki további funkciókra van szüksége a Date osztályból. Ilyen újabb funkció lehet például az, hogy a dátumok szokásos formátuma konvertálódjék át egészszámmá úgy, hogy a konverzió az 1900. január 1-jétől kezdve eltelt napok számát adja meg egy dátumra. Ezáltal két dátum között eltelt napok száma egyszerű kivonással határozható meg, vagyis csak a két megfelelő integer különbségét kell képezni. (Az ilyen számítások elsősorban a pénzügyi, banki rendszerekben szükségesek, például a különböző időtartamokra eső kamatok vagy az inflációs értékek megállapításánál.)

Mindezek után megállapíthatjuk, hogy a CRC-kártyák felelősségeinek egyenkénti kimunkálása nagymértékben elősegíti annak igazolását, hogy a végül létrehozott osztálydiagram teljes és korrekt lesz.

Fowler szerint mindenképpen érdemes a használatukat kipróbálni a team-munkában, és meggyőződni arról, hogyan válik be, hogyan fogadja be a team. Különösen akkor érdemes használatba venni, ha a team túlságosan belebonyolódik a részletekbe, még túl korán a fejlesztési folyamatban. Vagy pedig akkor, amikor nem sikerült világosan definiálni az egyes osztályok funkcióját.

A CRC modellezés eredményeit fel tudjuk használni az osztálydiagramok elkészítésénél. Mindez annak tudható be, hogy ezek az UML-diagramok az osztályok működésével és a köztük levő kapcsolatokkal vannak összefüggésben, csakúgy, mint a CRC-kártyák. Ilyenkor feltétlenül szükség van arra, hogy az osztálydiagramok mindegyik osztályához meg legyen adva azok felelősségi kör a CRC-kártyán.

5. Interakciós diagramok

5.1. Az objektumok közötti együttműködés

Egy objektumorientált rendszerben a számítási műveleteket az egymással információs kapcsolatban álló objektumok végzik el. Ezeknek a kapcsolatoknak a tervezés során történő modellezése alapvető fontosságú. Az UML-ben ezt a célt az ún. *interakciós diagramok (interaction diagram)* szolgálják.

Az interakciós diagram alkalmazásának tipikus esete az, amikor egy kiragadott use case viselkedését, működését írjuk le vele. Ez a diagram ilyenkor a use case-ben megnyilvánuló objektumokat mutatja be, a közöttük folyó vezérlési információcserével, vagyis az elküldött üzenetekkel együtt.

Hogy betekintést nyerjünk ebbe a folyamatba, az alábbiakban példaként bemutatjuk, hogy a Java programozási szférában miként kommunikál egymással két objektum.

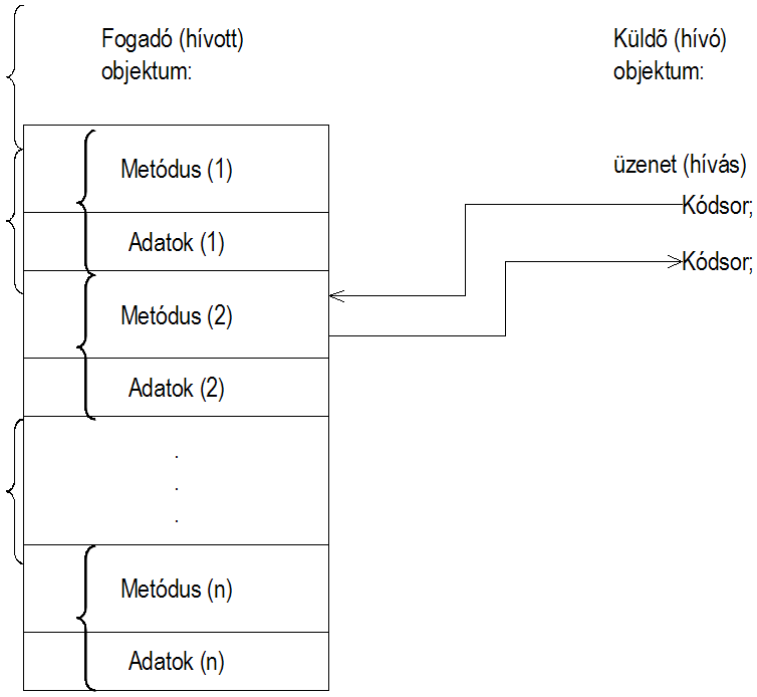
Egy Java-osztály olyan programegység, ami konkrét objektumokban ölt testet, azokban realizálódik a számítógépes végrehajtás során. Egy objektumban metódusok vannak, amelyek önállóan kezelhető, futtatható kódszegmensek, kódrészek.

Egy objektumhoz tartozó metódus a következő formátumú *üzenettel (message)* indítható el az objektumon kívülről, mégpedig egy másik objektumból:

Objektum neve . üzenet (argumentum1, argumentum2, … , argumentumn);

Itt szokás még a *fogadó (hívott)* objektum elnevezés is, ill. üzenet helyett a *szelektor* elnevezés. A szelektor a fogadó objektumon belül annak a metódusnak a kiválasztására szolgál, amelynek az üzenet szól. Tehát a fogadó objektumon belül a kiválasztott (hívott) metódust az üzenet maga választja ki. Az argumentumok pedig nem mások, mint az üzenethez tartozó vezérlő paraméterek. Ha a metódus argumentum nélkül működik, akkor ez a lista üres marad.

Egy üzenet mindig egy kiválasztott metódust indít el az objektumon belül. A metódus lefutása után a vezérlés az üzenetküldő utasítás utánra adódik vissza, a klasszikus hívás-visszatérés mechanizmusnak megfelelően. Megemlítendő még, hogy egy objektumon belüli metódusok is hívhatják egymást. A hívási folyamatot az 5.1. ábrán illusztráljuk.



5.1. ábra. Egy metódus lefuttatása üzenetküldéssel

Példák a hívásra:

localComputers . addComputer („Vax1”);

localComputers . removeComputer („Vax2”);

A két utasítás egy olyan programból van kiemelve, amelyben az első hívás hatására a *localComputers* objektum *addComputer* nevű metódusa az objektum által tárolt listába felveszi a *Vax1* tartalmú szövegkonstansot. A másik hívás hatására ugyanebből a listából törlődik a *Vax2* elem.

Az UML-ben kétféle interakciós diagram létezik:

- *Szekvenciadiagram,* vagy más néven *eseménykövetési diagram.* Az angol elnevezés: *sequence diagram.*
- *Együttműködési diagram (collaboration diagram).*

Az interakciós diagramok valójában az egyes use case-ek végrehajtási példáját, végrehajtási forgatókönyvét (szcenárióját) mutatják be.

A szekvenciadiagramok az objektumok időbeli viselkedésének menetét, folyamatát tükrözik, az objektumok közötti üzenetek időbeli követésé-

vel együtt. Arra használhatók, hogy egy adott use case szcenárió kölcsönhatásait szemléltessék.

Az együttműködési diagramok ezzel szemben az osztálydiagrammal való kapcsolatot jelenítik meg. Az objektumok közötti kölcsönhatást üzenetek halmazaként ábrázolják, ahol az üzeneteket az egyik objektum küldi a másiknak, valamilyen részfunkció teljesítése céljából. Ennek a megjelenítési módnak fontos sajátsága az, hogy az üzenetek csoportosan, bekövetkezésük sorrendjében vannak feltüntetve.

A szekvenciadiagramok és az együttműködési diagramok valójában egymással ekvivalens leírást adnak. Mindkettő azt mutatja, hogy az objektumok közötti kölcsönhatás miként megy végbe üzenetek váltása révén. Egy bizonyos dinamikus nézetet adnak a rendszerről azáltal, hogy grafikusan jelenítik meg az objektumok *futási időben (at run time)* végbemenő működését. A különböző működési folyamatok különböző forgatókönyveket valósítanak meg.

A következőkben külön-külön tárgyaljuk és mutatjuk be a két diagramtípust.

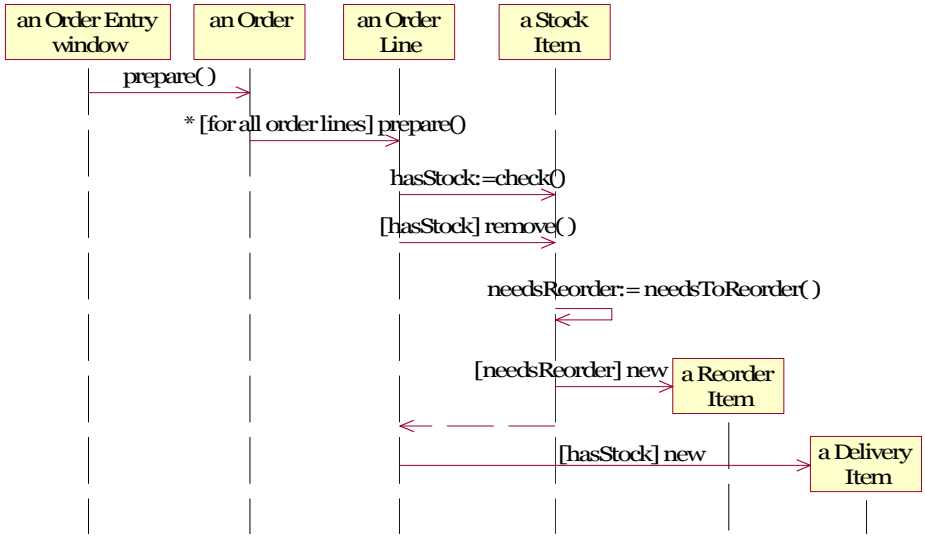
5.2. Szekvenciadiagramok

A szekvenciadiagram használatát egy példán keresztül fogjuk ismertetni. Ehhez egy use case-ből indulunk ki, amely a következő működést valósítja meg:

- Az *Order Entry* (megrendelési belépés) képernyő-ablak egy *„prepare”* (előkészítés) üzenetet küld az *Order*-nek (megrendelés).
- Az *Order* egy *„prepare”* (előkészítés) üzenetet küld mindegyik *Order Line* (megrendelési tétel) számára, éspedig azokra a tételekre vonatkozóan, amelyek az *Order*-ben (megrendelésben) szerepeltek.
- Minden egyes *Order Line* ellenőrzi a neki megfelelő, hozzá tartozó *Stock Item*-et (raktári tételt), azzal a céllal, hogy kiderüljön, megvan-e a kellő mennyiség a raktáron a megrendelt tételből.
  - Ha az ellenőrzés „igaz” eredményt ad vissza, az *Order Line* az aktuális *Stock Item*-et annyi darabbal csökkenti, amennyi az *Order*-ben szerepelt.
  - Ha nincs meg maradéktalanul a megrendelt mennyiség az adott tételből, akkor a *Stock Item* egy új utánpótlási szállítást, feltöltést igényel meg.

A fent leírt folyamat szekvenciadiagramját az 5.2. ábrán láthatjuk. A diagramon egy objektum egy dobozként van ábrázolva, egy függőleges szaggatott vonal tetején. Ezt a függőleges vonalat *életvonalnak (life line)* nevezzük. Az életvonal az objektum működésének lefolyását reprezentálja, az események menetét követve. (Ezért van használatban az *eseménykövetési diagram* elnevezés is.) Az itt látható ábrázolási mód egyébként Jacobsontól származik.

A két objektum közötti üzenetet egy vízszintes nyíl ábrázolja, a két életvonal között húzódva. Az üzenetek abban a sorrendben, szekvenciában követik egymást, fentről lefelé haladva, ahogyan egymás után bekövetkeznek. A nyilakon fel van tüntetve az üzenet neve. Ehhez még hozzá lehet írni az üzenet argumentumait (vezérlési paramétereit), továbbá bizonyos vezérlési információt is.



5.2. ábra. Egy szekvenciadiagram

Egy objektum önmagának is küldhet üzenetet. Az ilyen eseményt egy visszakanyarodó nyíl jelzi, a saját életvonalból kiindulva, és ugyanoda visszatérve. Ezt *öndelegálásnak* (*self-delegation*-nek) is szokás nevezni.

A vezérlési információ megadására még két másik lehetőség is kínálkozik:

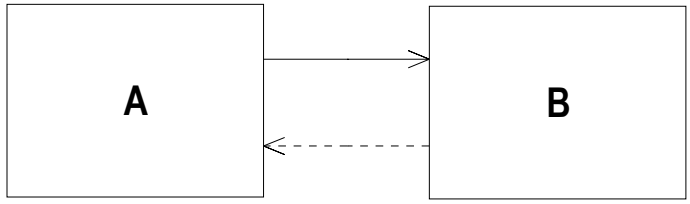
- Meg lehet adni annak a *feltételét (Condition),* hogy mikor, mitől függően lesz elküldve egy üzenet. Itt például: needsToReorder == true esetén az *Reorder Item*) (tétel utánrendelése) objektum létrehozására (new) és

elindítására kerül sor. Ekkor az objektum alatt természetesen az életvonala is megjelenik.

- A másik hasznos vezérlési jelölés az ún. *ismétlési jel (iteration marker),* ami azt mutatja, hogy egy üzenet több alkalommal lesz elküldve. A példában annyiszor, ahány fajta megrendelési tétel szerepelt a listában. Ezt a többszörözést a ** prepare ()* formában leírt üzenet fejezi ki, ami azt is jelenti egyben, hogy ennyi darab *Order Line* (megrendelési tétel) objektum lesz megszólítva, aktivizálva, mivelhogy mindegyik lehetséges tételhez egy külön objektum tartozik.

Az ábrán jól áttekinthető és követhető az objektumok közötti események menete, ami igen előnyös tulajdonsága a szekvenciadiagramoknak. Az OO-programozásban ez különösen fontos. Ott ugyanis sok olyan metódus (művelet) szerepelhet az egyes osztályokban, amelyek egymás utáni működtetése a szekvenciadiagram nélkül nehezen volna követhető, áttekinthető, kézben tartható. A forráskód ebből a szempontból kevésbé használható, bár megvannak benne az objektumok, amelyek egymás metódusait hívják az üzeneteken keresztül, de ebből a sorrendiség nehezen vehető ki. Egy olyan programozónak, akinek még új az OO-technika, különösen sok gondot okozna a kódban való eligazodás.

Az 5.2. ábra tartalmaz egy visszatérési *(Return)* nyilat is. Ez nem jelent semmilyen új, igazi üzenetet, hanem csak egy üzenetet követő visszatérést. Használata nem kötelező, mivel a diagram e nélkül is egyértelmű. Olyan, mint a hívásból való visszatérés. Az 5.3. ábrán **A** hívja **B**-t, ami után **A** visszakapja a vezérlést. A hívási kapcsolat kifejezésére itt elegendő lenne a felső nyíl.

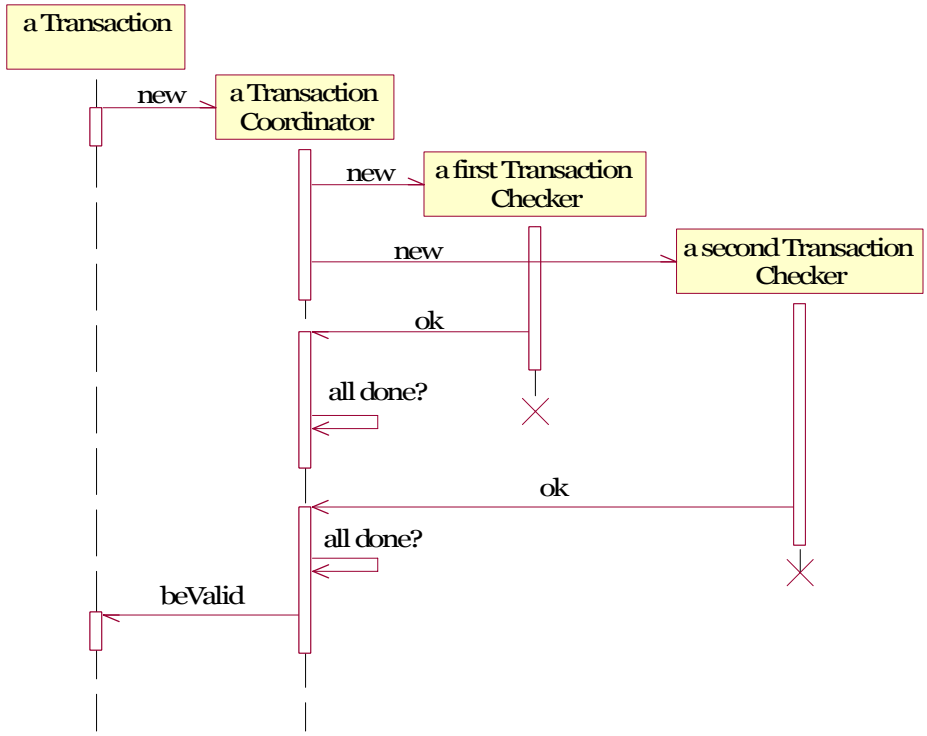


5.3. ábra. Hívási és visszatérési kapcsolat két modul között

A *Return* nyilat akkor érdemes használni, amikor az javítja az áttekinthetőséget a diagramon. A példában csak az illusztráció miatt használtuk, el is lehetett volna hagyni, mert nem segített az áttekinthetőségen.

5.3. Konkurrens folyamatok és az aktiválások

A szekvenciadiagramok alkalmasak arra is, hogy konkurrens (egyidejű) folyamatokat lehessen velük leírni. Az 5.4. ábra olyan objektumokat tartalmaz, amelyek egy banki tranzakciót ellenőriznek.



5.4. ábra. Konkurrens folyamatok és aktiválások

Amikor a *Transaction* (tranzakció) nevű objektum kerül létrehozásra (*new* parancs), akkor az a következő lépésben létrehoz egy *Transaction Coordinator* (tranzakció-koordinátor) nevű objektumot (ismét *new*). A *Coordinator*-nak az a feladata, hogy ellenőrizze a tranzakciót. Ezt úgy végzi el, hogy annyi különböző ellenőrző objektumot hoz létre, ahány egymástól eltérő ellenőrzési feladatot kell a banknál az ottani szabályok szerint ellátni. Ebben a példában kettő ilyen *Transaction Checker* (tranzakció-ellenőrző) objektum szerepel, (*first* és *second*). Mindkettőnek megvan a maga külön megszabott ellenőrzési funkciója.

A két ellenőrzési tevékenység egymástól függetlenül folyik, semmilyen szinkronizálásra, ütemezésre nincsen velük kapcsolatban szükség. Ezt úgy is megfogalmazhatjuk, hogy a két objektum egymással *aszinkron* módban működik. Az aszinkronitás egyszerűbbé teszi a vezérlésüket, mivelhogy bármelyikük indítható először is, meg másodszor is. Ha kettőnél több fajta ellenőrzésre lenne szükség, azok is ugyanígy, egymással párhuzamosan, egymással egyidejűleg tudnának működni. Ezek a folyamatok a futás szervezése, az eredmények feldolgozása szempontjából *egyidejűlegesek,* más szóval *konkurrensek*. Természetesen a gépi végrehajtás szempontjából nem tudnak párhuzamosan futni, hanem vagy egymás után, vagy valamilyen felváltásos szervezésben. (A valódi párhuzamos futás többprocesszoros számítógépet igényelne, de itt csak egy processzort tételeztünk fel.)

Amikor egy *Transaction Checker* befejezi a tevékenységét, akkor erről értesíti a *Transaction Coordinator*-t. A *Coordinator* ekkor megnézi, hogy bejött-e a másik visszajelentkezés is, vagyis hogy mindkét *Checker* elvégezte-e a feladatát. Ha ez még nem történt meg, akkor a *Coordinator* nem csinál semmit, csak várakozik. Ha viszont mind a két visszajelzést megkapta, és mind a két ellenőrzés rendben találta a tranzakciót, akkor a *Coordinator* értesíti a *Transaction* objektumot, hogy minden teljes rendben van.

Az 5.4. ábrán a szekvenciadiagramok több új rajzeleme figyelhető meg:

Az egyik ilyen az ún. *aktiválás,* ami az életvonalon egy lefelé nyúló téglalap. Ez addig tart, amíg az adott objektum egyik metódusa (művelete) aktív, vagyis működésben van: vagy végrehajtódik a benne levő utasításokkal, vagy pedig várakozik arra, hogy megkapja a várt információt egy *Return* után.

Fowler tapasztalatai szerint sokan vannak, akik az aktiválást minden helyen alkalmazzák, feltüntetik. Ő maga úgy véli, hogy ezzel nem sok plusz információt visznek be a végrehajtás menetébe. Ebből kifolyólag csak a konkurrens folyamatok megjelenítésénél él a használatukkal.

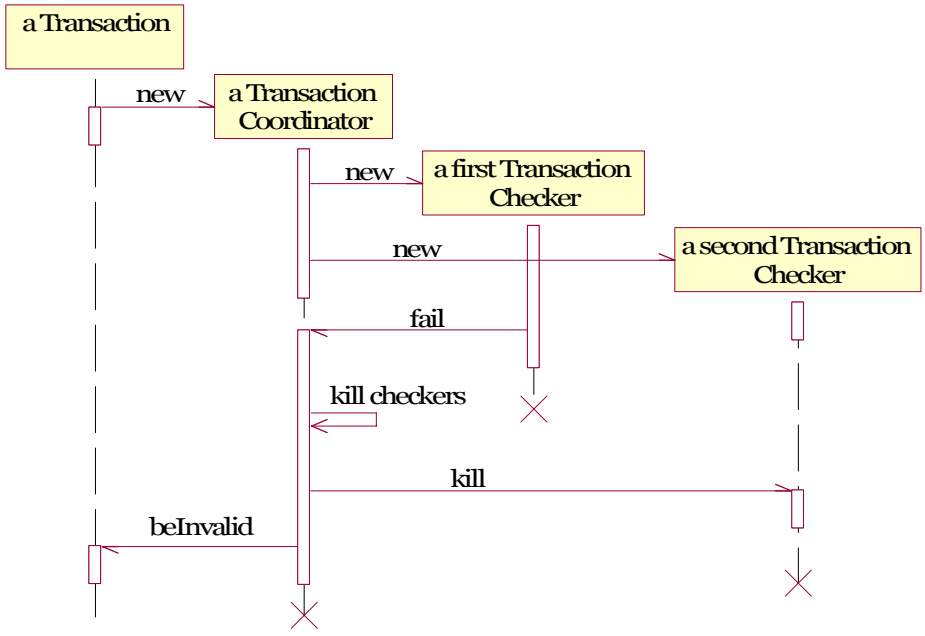
A másik új jelölés a *fél nyílfej,* amely egy *aszinkron* típusú üzenetet fejez ki. Mint láttuk, az aszinkronitás itt azt jelentette, hogy az ilyen jellegű üzenet (hívás) bármikor kiadható, vagyis a hívó objektumot semmi nem korlátozza az üzenet kiadásának időpontját illetően. Ez egyben azt is jelenti, hogy az aszinkron üzenet semmilyen módon nem korlátozza a hívót, az a hívás után szabadon folytathatja tovább saját feldolgozó tevékenységét.

Egy aszinkron üzenet három különböző dolgot tud eredményezni:

- Egy új programozási szálat hoz létre, ami ebben az esetben egy aktiválásnak a kezdetéhez fog kapcsolódni. (Mármint a szálat képviselő aktiválás kezdetéhez.)
- Létrehoz egy új objektumot.
- Kommunikál egy olyan szállal, amely éppen futásban van.

A harmadik új jelölés az objektum *törlése,* amit egy nagyméretű **X** jel reprezentál. Az OO-technikában ezt az utasítást egy objektum *destrukciójának, elpusztításának,* vagy *megszüntetésének* nevezik. Hatására az objektum futása megszakad, miközben maga az objektum is eltűnik a memóriából.
Egy objektum kétféle módon szűnhet meg:

- Vagy saját magát szünteti meg *öndestrukcióval (self-destruction),* amint az 5.4. ábrán is látható.
- Vagy pedig egy másik objektum pusztítja el egy megfelelő üzenettel, amint az 5.5. ábrán látható. Ez az üzenet a *„kill”*.



5.5. ábra. Szekvenciadiagram tranzakció-ellenőrzéssel

Az 5.4. ábra a sikeres tranzakciót írta le, míg az 5.5. ábra a sikertelent. Az utóbbi esetben az első *Checker* hiányosságot észlelt, és ezt követően már ki lehetett lőni, el lehetett pusztítani a második *Checker*-t.

A két ábrán jól látható az *önhívás (ön-delegálás)* menete, ami itt az aktiváláshoz kapcsolódik. Ezt az egymásra helyezett aktiválási mezők még jobban kiemelik, ahhoz képest, mintha csak a puszta életvonalak lennének feltüntetve. Ily módon szemléltetni lehet azt is, hogy mikor kerül sor egy hívásra az önhívást követően, akár a hívó, akár pedig a hívott metódusban. Ezt az egymásra helyezett mezők teszik lehetővé.

Az 5.4. ábra és az 5.5. ábra két *szcenáriót* mutat be a tranzakcióellenőrzés lefolytatására. Nyilvánvaló, hogy ez két különböző use case-t jelent. A két szcenáriót külön rajzoltuk meg. Lehetett volna egyben is megrajzolni őket, de azzal túl komplikálttá, túl áttekinthetetlenné vált volna a kép.

5.4. Együttműködési diagramok

Az interakciós diagramok második fajtája az ún. *együttműködési diagram (collaboration diagram)*. Az objektumok ezeken is téglalapokkal vannak feltüntetve, és a köztük levő üzeneteket ugyancsak nyilakkal jelöljük, hasonlóan a szekvenciadiagramokhoz. A különbség itt abban nyilvánul meg, hogy az üzenetek sorszámokkal vannak ellátva, ami az egymás utáni sorrendiség megadására szolgál. A másik eltérés az, hogy az objektumok tetszőlegesen helyezhetők el a rajzon, a szerint, ahogyan a számozott üzenetek ezt megkívánják.

Mint látni fogjuk, ez az ábrázolási mód nehezebbé teszi a sorrendiség áttekintését, követését ahhoz képest, hogy egymás alá rajzolnánk az üzeneteket, mint a szekvenciadiagramokon. Másfelől azonban a térbeli elrendezés lehetővé teszi, hogy más dolgokat könnyebben lehessen megmutatni. A rajzon való elrendezésből jobban kitűnik, hogy miként vannak összekapcsolódva az objektumok, hogyan vannak statikusan összekötve egymással.

Mindehhez kétféle számozási séma rendelhető. Az egyik az *egyszerű,* 1-től egyesével növekvő számozás (5.6. ábra), a másik pedig egy lebontásos, *decimális osztású (tizedes osztású)* számozás (5.7. ábra). A két ábra ugyanazt a diagramot jeleníti meg, különböző számozással.

Régebben az egyszerű számozási szisztémát használták a legtöbben. Az UML-en belül ma már a tizedes osztású számozás van elterjedve. Ez azért lett így, mert világosan kifejezi, hogy melyik művelet (metódus) me-

lyik műveletet (metódust) hívja közvetlenül. (Ugyanakkor azonban ez megnehezíti a közvetlen sorrendiség követését.)

A választott számozási rendszertől függetlenül itt is megadható az a kiegészítő vezérlési információ, amit a szekvenciadiagramoknál is lehetett használni.

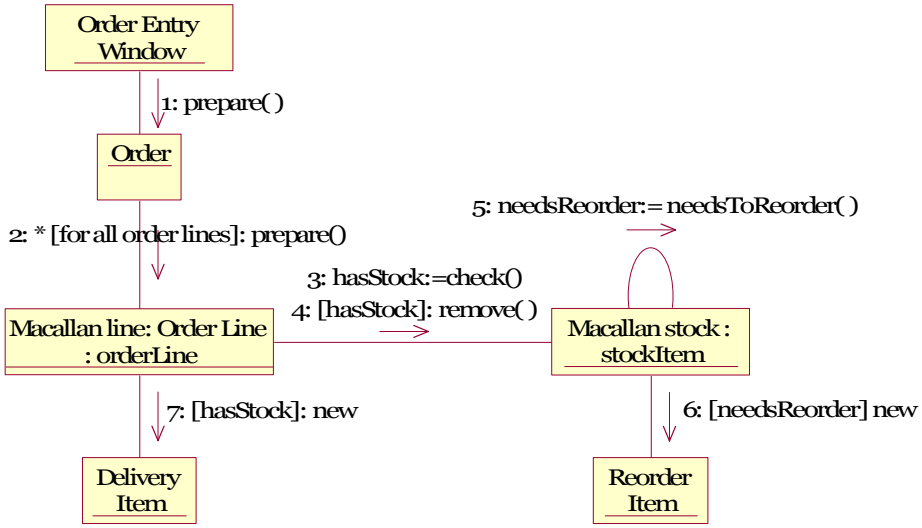
Az 5.6. ábrán és az 5.7. ábrán látható az is, hogy az UML miként nevezi el az objektumokat. Ennek az UML-szabványos alakja a következő:

objectName : ClassName

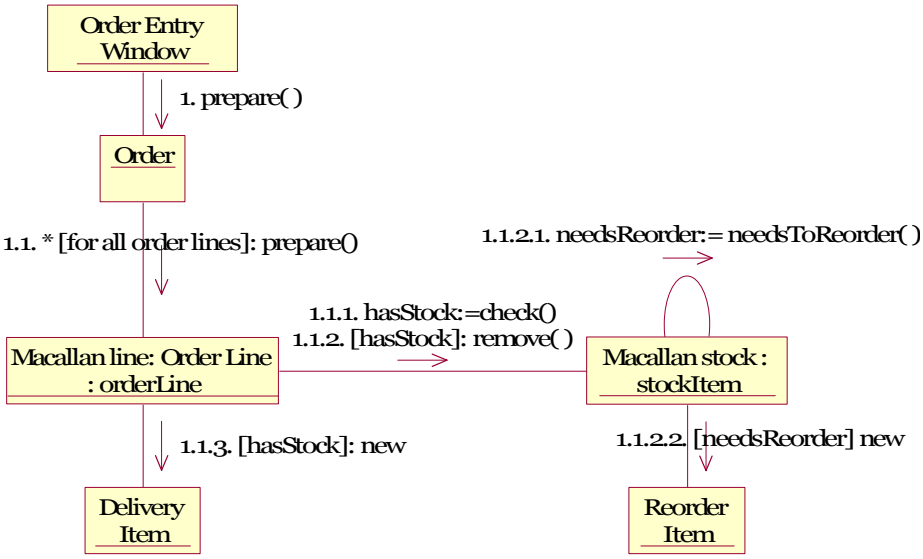
ahol vagy az *objectName* (objektum neve), vagy a *ClassName* (osztály neve) elhagyható. De a kettő közül csak az egyik! Ha az objektum nevét hagytuk el, akkor is ki kell tenni a kettőspontot, hogy lássuk azt, hogy az osztály neve van feltüntetve, nem pedig az objektum neve. Eszerint a

„Macallen Line : Order Line"

az *Order Line* osztály egy példányát jelenti, amit *Macallen Line*-nak neveznek. (A Macallen egy amerikai édességmárka.)



5.6. ábra. Együttműködési diagram egyszerű számozással



5.7. ábra. Együttműködési diagram decimális osztású számozással

5.5. Felhasználási példa könyvtári kölcsönzésre

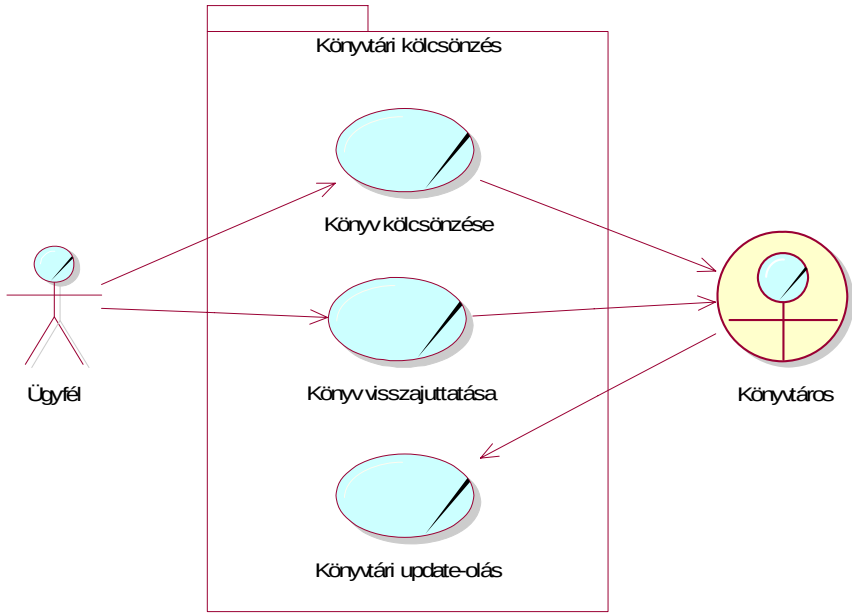
A use case diagramok átfogó képet adnak a szoftver rendszerben szereplő aktorokról, valamint azokról az akciókról, amelyeket a rendszer hajt végre. Az akciók észlelhető eredményt adnak, amelyek az aktorok számára lesznek értékesek. A diagramok egy rendszert a teljes összefüggésében írnak le, azáltal, hogy a funkcionalitást olyan tranzakciókra bontják, amelyek ugyancsak hasznosak az aktorok számára. Emellett azt is mutatják, hogy milyen kölcsönhatás lép fel az aktorok és a tranzakciók között.

A tervezett rendszer működésének leírásakor jól használható, a fejlesztők által is támogatott use case technikát nemcsak a szoftverkövetelmények leírására használhatjuk. A use case-eket eredményesen alkalmazhatjuk a rendszerfejlesztés első fázisában az üzleti folyamatok, az üzleti követelmények leírásakor. A rendszerfejlesztésnek ebben a szakaszában még nem térünk ki a szoftverrendszer működésére, csak azt vizsgáljuk, hogy a fejlesztendő szoftverrendszernek milyen üzleti folyamatok (üzletmenet) számítógépes támogatását kell megvalósítani.

A modellben az üzleti folyamatok leírásakor használt use case-eket business use case-eknek nevezik. A modellben a business sztereotípia fejezi ki, hogy a use case-eket üzleti folyamatok, üzleti követelmények leírására használjuk, amik támogatására egy szoftverrendszert akarunk fejleszteni. A

CASE fejlesztőeszközök többsége ma már szinte kivétel nélkül támogatja az üzleti folyamatok use case-ek és aktorok segítségével történő leírását a business sztereotípia megadásával. Vannak fejlesztőeszközök, amik a business jelleg szemléltetésére másfajta szimbólumot használnak. A business sztereotípiájú modellelemek grafikus megjelenítésekor a modellelemek jobb oldalán elhelyezett ferde vonallal utalnak az üzleti jellegre.

Tekintsük példaként a könyvtári kölcsönzést, amelynek business (üzleti) use case diagramját az 5.8. ábra tünteti fel.

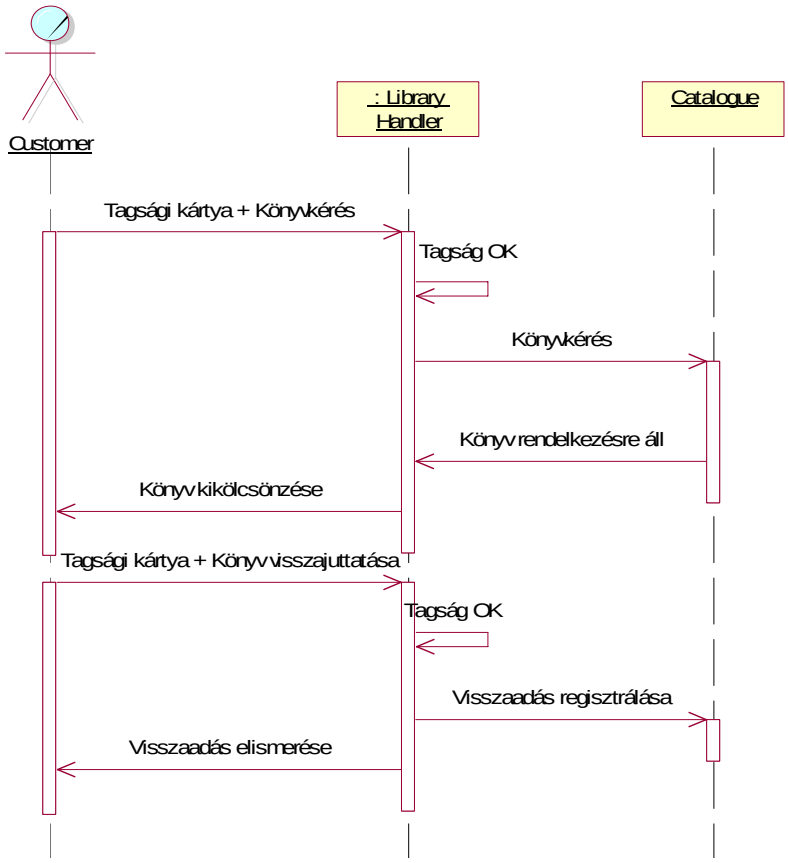


5.8. ábra. Könyvtári kölcsönzés business (üzleti) use case diagramja

A tervezett rendszer lehetővé teszi, hogy kölcsönözni és visszajuttatni tudjuk a könyveket. Ezek az akciók az ügyfeleket és a könyvtárosokat vonják egy körbe. A könyvtárosok ezen kívül még update-olhatják (naprakész állapotba hozhatják) a könyvtári állományt, azáltal, hogy új könyveket vesznek fel a katalógusba, ill. törlik a leselejtezett régieket.

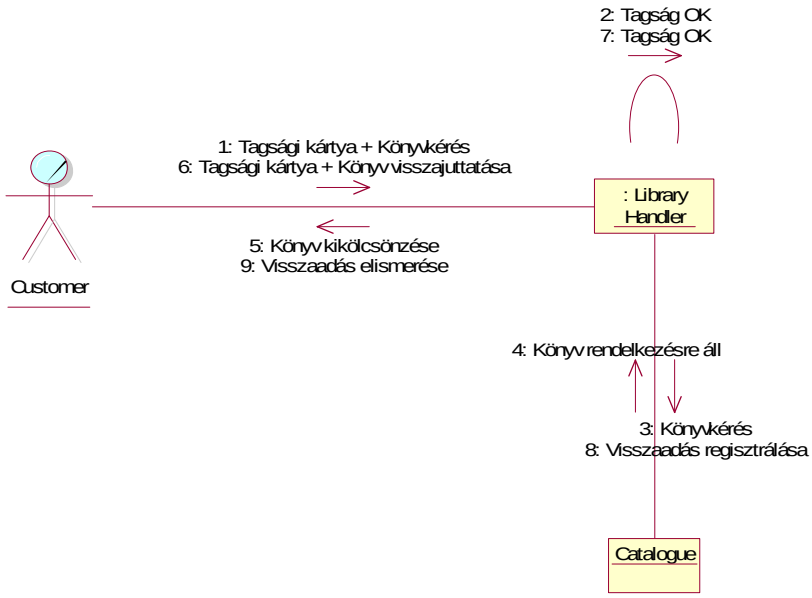
A kölcsönzés és visszajuttatás folyamatainak egy lehetséges forgatókönyve az 5.9. ábra szekvenciadiagramjával valósítható meg. Az ábrán szereplő objektumok nevei: *Customer (Ügyfél), Librarian (Könyvtáros),* valamint *Catalogue (Katalógus).* Kölcsönzéskor az ügyfél a tagsági kártyáját bemutatja a könyvtárosnak, aki ellenőrzi, hogy az nem járt-e le. Ha a kártya érvényes,

akkor a katalógus ellenőrzésére kerül sor, hogy rendelkezésre áll-e a keresett könyv. Ha igen, akkor az ügyfél kikölcsönözheti azt. Hasonló módon megy végbe a könyv visszajuttatása is.



5.9. ábra. A Könyvkölcsönzés business use case-ben definiált működést leíró szekvenciadiagram

Az 5.10. ábra ugyanezt a szcenáriót írja le együttműködési diagram segítségével. Az események időbeli sorrendjét az objektumok közötti kapcsolatokon feltüntetett sorszámozás fejezi ki.



5.10. ábra. A Könyvkölcsönzése business use case-ben definiált működés megvalósításában részt vevő objektumok kommunikációját leíró együttműködési diagram

5.6. A kétféle diagram összehasonlítása

A különböző fejlesztők nem egyformán preferálják a két diagramot. Van, aki az egyiket részesíti előnyben, van, aki a másikat. Sokan a szekvencia-diagramot szeretik, mivel az jobban kifejezi a sorrendiséget. Abban könnyű követni azt a sorrendet, amiben a dolgok végbemennek. Mások azért preferálják az együttműködési diagramokat, mivel ott olyan elrendezésben tudják felrajzolni az objektumokat, amiből látszik, hogy azok hogyan kapcsolódnak egymáshoz statikusan.

Mindkét fajta interakciós diagramnak fontos jellemzője, hogy eléggé egyszerűek. Jól áttekinthetők bennük az üzenetek az objektumok között. Ugyanakkor az ilyen diagramok használatát megnehezíti, ha sok feltételes elágazás van bennük, vagy pedig hurkokat tartalmaznak. Hogyan érdemes a feltételes működést ábrázolni:

- Az egyik megoldás az, amikor külön diagramot rajzolunk fel minden egyes szcenárióra nézve.
- A másik megoldás az, amikor az üzeneteket egészítjük ki azokkal a feltételekkel, amelyek teljesülése esetén megy ki egy adott üzenet.

Az első megoldás előnye a biztonságos áttekinthetőség. Az interakciós diagramokat illetően általában is érdemes törekedni arra, hogy elkerüljük a túl bonyolult ábrázolást, mert azt már nehéz lesz átlátni.

Mikor érdemes használni őket?

Az interakciós diagramokat leginkább akkor érdemes használni, amikor több objektum együttes működését akarjuk látni, egyetlen egy use case-en belül. Egy ilyen diagram jól leírja az objektumok egymás közötti információáramlását, és az ennek hatására történő viselkedésüket.

Ha egyetlen objektum működését akarjuk kiragadottan végigkövetni egy use case-en keresztül, akkor a legcélszerűbb az objektum állapotdiagramját felhasználni (7. fejezet).

Ha az általános működést akarjuk követni több use case-en keresztül, akkor a legcélszerűbb az aktivitási diagramot alkalmazni (8. fejezet).

6. Csomagdiagramok

6.1. A szoftverműködés lebontása

A szoftvertechnológia egyik alapvető kérdése az, hogy milyen módon bontsunk le egy nagy rendszert kisebbekre. Ez azért fontos kérdés, mert ha egy rendszer túlságosan kiterjedtté válik, akkor igen nehéz lesz áttekinteni, megérteni a működését, és ugyancsak nehéz lesz ilyenkor a változtatásokat is elvégezni rajta. A lebontással nyert kisebb rendszereknél ezek a nehézségek is jóval kisebbek lesznek.

A procedurális, és ezen belül a strukturális tervezési módszerek *funkcionális lebontást (functional decomposition)* alkalmaznak. Ebben a megközelítésben először a teljes rendszer egészének a funkcióját veszik figyelembe, és utána ezt bontják szét több alfunkcióra, majd ezt követően az alfunkciókat bontják le további al-alfunkciókra. Az így kezelt funkciók hasonlítanak egy objektumorientált szoftver use case-eire, feltéve, hogy az egyes funkciókat összerakva valóban a teljes rendszer működése fog előállni.

A procedurális szoftvereknél a folyamatok és az adatok szét vannak választva. Ebből adódóan a funkcionális lebontáson kívül még külön meg kell tervezni az adatstruktúrát is, mégpedig úgy, hogy az összhangban legyen a funkciók együttesével.

Az objektumorientált technológia éppen ezen a téren hozta a legnagyobb változást. A folyamatok és az adatok szétválasztása eltűnt, megszűnt, a funkcionális lebontás úgyszintén megszűnt benne. Mindazonáltal az eredeti nagy kérdés továbbra is megmaradt: Hogyan bontsuk szét a teljes nagy rendszer egészét kisebb rendszerekre?

Az egyik megoldási megközelítés az, hogy az osztályokat csoportosítjuk magasabb szintű egységekbe. A legtöbb OO tervezési módszer is ezt az elvet próbálja követni. Az UML-ben is megvan az erre irányuló csoportosítási mechanizmus. Egy-egy ilyen jellegű csoportot *csomagnak (package)* nevezünk.

6.2. Függőségi viszonyok

A csomagba való csoportosítás elve nem csak az osztályokra alkalmazható, hanem bármilyen alkotóelemre. A csoportosítás menetére vonatkozóan nem állnak rendelkezésre egzakt algoritmusok, bevált módszerek. Vagyis: általános recept nincsen. Ehelyett intuitív, heurisztikus megoldásokat lehet alkalmazni.

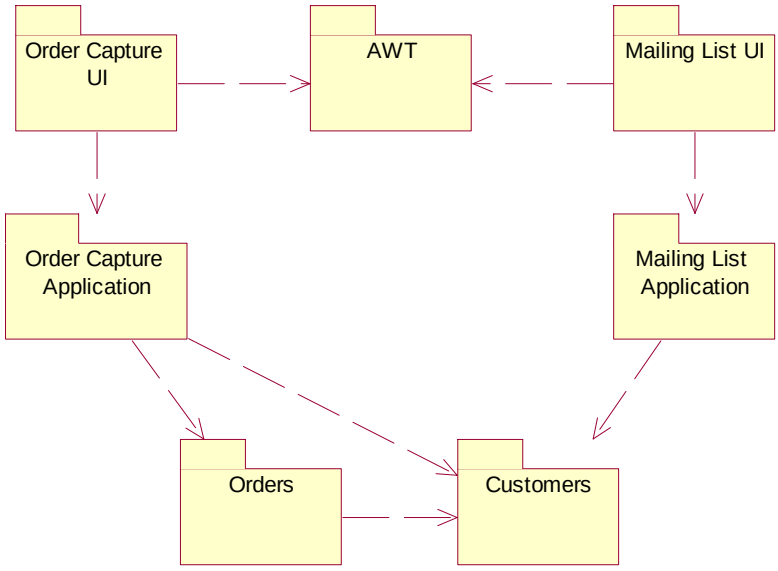
Az UML-ben a leginkább elterjedt rendezési elv az ún. *függőség (dependency)*. A csomagokat ábrázoló diagramokat, amelyek a függőségeket is kifejezik, *csomagdiagramnak (package diagram)* nevezzük. Fontos megjegyezni, hogy az ilyen diagramokban a függőségi kapcsolatok és a csoportosítás az osztályokra vonatkoznak.

Szigorúan véve a csomagok és a köztük levő függőségek egy osztálydiagramhoz tartozó elemek. Mondható tehát, hogy a csomagdiagram az osztálydiagramnak egy bizonyos megjelenési formája. A két külön elnevezés használata azért indokolt, mert a két megjelenési formát is külön célokra érdemes használni.

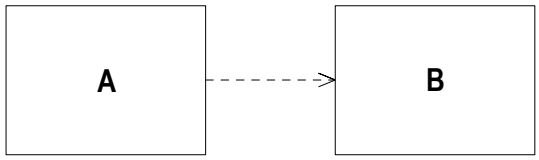
Mindezek után definiáljuk a *függőség* fogalmát. Két tetszőleges elem között akkor létezik *függőség,* ha az egyik elem specifikációjában, megvalósításában történő változtatás változást okozhat a másik elem specifikációjában, megvalósításában. Röviden szólva, az egyik elem megváltoztatása kihat a másik elemre. Osztályok esetében a függőség különböző okokra vezethető vissza. Például: az egyik osztály üzenetet küld a másiknak; az egyik osztály a másik osztályt adatai részeként használja; az egyik osztály a másikat egy saját műveletének (metódusának) paramétereként használja fel. Ha például egy osztálynak megváltoztatjuk az interfészét, akkor bármelyik általa küldött üzenet elveszítheti korábbi érvényességét.

Ideális esetben csak annak kellene befolyásolnia bármelyik más osztály működését, ha egy osztálynak megváltoztatnánk az interfészét. A nagyméretű projekteknél általános cél, hogy minimalizáljuk a függőségeket. Ennek a célnak az a nyilvánvaló értelme, hogy egy változtatás hatása minél kisebb körre terjedjen ki, és ezáltal minél kisebb ráfordítással lehessen a teljes rendszeren átvezetni a változtatásokat.

Példaként tekintsük a 6.1. ábrát. Ezen *domén osztályok* láthatók, amelyek egy üzleti modellt adnak ki. A modell két csomagba van csoportosítva: *Orders* (megrendelések) és *Customers* (ügyfelek. Mind a két csomag egy általánosabb domén csomag részét képezi. Az *Order Capture Application* (megrendelési alkalmazás) függőségben van mind a két domén csomaggal. Az *Order Capture UI* (megrendelési fogadás felhasználói interfésze) függőségben van az *Order Capture Application*-nel, valamint az *AWT*-vel (az AWT egy Java grafikus eszközkészlet). Az ábrán a függőséget a szaggatott vonallal megrajzolt nyilak fejezik ki. A nyíl abba az irányba mutat, amelyik elemtől a függőség ered. A 6.2. ábrán egy ilyen kapcsolat látható, ahol az **A** elem függőségben van **B**-vel, úgy, hogy **A** függ **B**-től.



6.1. ábra. Egy csomagdiagram



6.2. ábra. Függőségi kapcsolat

Két csomag között akkor áll fenn függőség, ha bármilyen függőség létezik az egyik csomag bármelyik osztálya és a másik csomag bármelyik osztálya között. Például, ha a *Mailing List Aplication* (postázási lista alkalmazás) csomag bármelyik osztálya függőségben van a *Customers* (ügyfelek) csomag akármelyik osztályával, akkor függőség fog létezni a két, szóban forgó csomag között is.

A függőségek a csomagokra nézve *nem tranzitívek,* ami itt egy fontos sajátosság. A *tranzitivitás* a függőség láncszerű továbbszármaztatását jelenti, a függőségi kapcsolatokon keresztül. Ez viszont nem feltétlenül teljesül a csomagdiagramoknál, amire egy szemléltető példát ad a 6.3. ábra. Az ábrán **A** függ **B**-től, és **B** függ **C**-től. Ebből a relációból viszont még nem következik egyértelműen, hogy **A** is függ **C**-től. Természetesen, ez nincsen kizárva, csak nem garantálható törvényszerűen a függőségi lánc megléte **A** és **C** között.

6.3. ábra. Nem tranzitív kapcsolatok

Egy példa a *tranzitív* relációmeglétére:

„Géza magasabb, mint Péter”, „Péter magasabb, mint Tamás”.

A fenti kettőből egyértelműen következik, hogy

„Géza magasabb, mint Tamás”, vagyis a „magasabb, mint” reláció tranzitív.

Ezzel szemben a „barátja valakinek” reláció már nem tranzitív. A

„Géza barátja Péternek”, „Péter barátja Tamásnak”

relációkból nem következik, hogy *„Géza barátja Tamásnak”*. Lehet, hogy nem is ismerik egymást. Persze azt nem lehet kizárni, hogy esetleg mégis barátok lennének.

A csomagdiagramoknál pozitív sajátosság, hogy a függőségi viszony nem tranzitív, tehát nem terjed feltétlenül tovább. A 6.1. ábra példáján ez az elv a következő módon tud érvényesülni:

Ha az *Orders* csomagban megváltozik egy osztály, az nem vonja maga után azt, hogy az *Order Capture UI* csomagot meg kellene változtatni. Amit maga után von, az mindössze annyi, hogy csak azt kell megvizsgálni, hogy az *Order Capture Application* csomagnál szükség van-e a változtatásra.

Az *Order Capture UI* csomagot csak akkor kellene megváltoztatni, ha az *Order Capture Application* csomag *interfészében* történt volna változtatás. Végül is, itt az látható, hogy az *Order Capture Application* csomag mintegy védőfalként szerepelt az *Order Capture UI* csomag felé, és ezáltal megóvta azt az *Orders* csomag változásának hatásától, következményeitől.

Ez az a működési elv, amit a klasszikus *réteg szervezési* megoldás is alkalmaz. Ebben a megoldásban az egymással nem határos szoftverrétegek önállóan változtathatók, és nem hat ki rájuk a nem-határos réteg szoftverjében végrehajtott változtatás. Mint ismeretes, tipikusan ily szervezésűek például az operációs rendszerek.

Az egyes OO programozási nyelvek különböző módon segítik elő, teszik lehetővé a változtatási hatások továbbterjedésének gátlását, megakadá-

lyozását. Ez erősen összefügg azzal, hogy egy adott OO-nyelv milyen láthatóságot biztosít egy osztály esetében a többi osztály számára. Vagyis, a többi osztály mit lát belőle, és mit tud befolyásolni benne, másrészt a különböző osztályok milyen mértékben képesek egymás működését látni, ellenőrizni, befolyásolni. Tehát egy programozási nyelvtől függ az, hogy menynyire engedi meg azt, hogy az osztályok egymással össze tudjanak fonódni a működésükben. Ebben a tekintetben a C++ nyelv jóval nagyobb láthatóságot, azaz tranzitivitást enged meg, mint például a Java nyelv.

A csomagok önmagukban véve nem adnak támpontot abban, hogy miként tudjuk csökkenteni a függőségeket a szoftverünkben. Amiben a segítségünkre vannak, az abban áll, hogy látni tudjuk belőlük, hol vannak a függőségek. Ezeket a függőségeket csak úgy tudjuk megszüntetni vagy csökkenteni, ha tisztában vagyunk azzal, hogy hol vannak ilyenek. A csomagdiagramok kitűnő segédeszközök arra, hogy a teljes szoftverrendszer struktúrája felett áttekintésünk legyen, és abba célszerűen be tudjunk avatkozni, azt át tudjuk szervezni, alakítani.

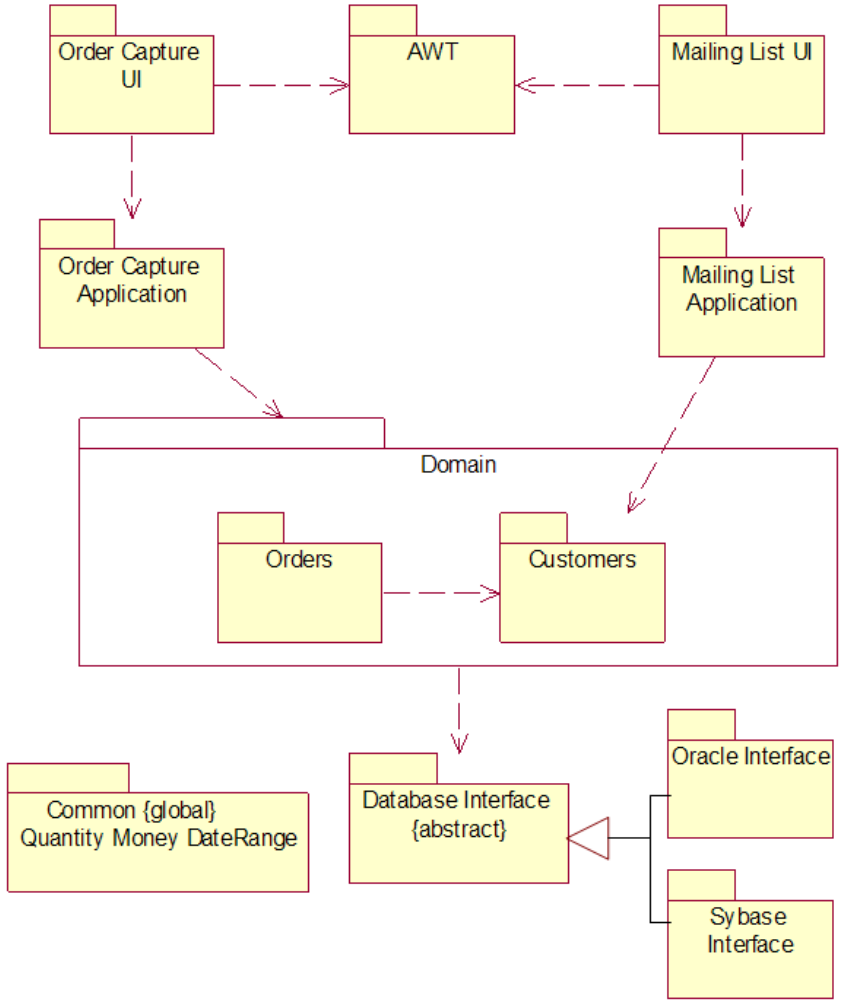
6.3. Kibővítések a csomagdiagramban

A csomagdiagramok további lehetőségeket is tartalmaznak. A 6.4. ábra ilyen kibővítéseket tüntet fel a 6.1. ábrához képest.

Az egyik bővítés az, hogy egy közös *Domain* nevű csomagba lett belerakva az *Orders* és a *Customers* nevű csomag. Ezáltal a függőséget csak erre a magasabb szintű csomagra kell értelmezni, a benne levő elemekre vonatkozó szétbontott függőségek helyett. Az ilyen jellegű módosítás hasznos lehet az elemzés továbbvitelében. Itt az *Order Capture Application* a teljes *Domain* csomagtól van függésben, míg a *Mailing List Application* csak a *Customers* csomagtól van függőben.

A másik bővítés a *Common* (közös) elnevezésű csomag megjelenítése. Ez a csomag a {*global*} megjelölést kapta, ami azt jelenti, hogy a diagram mindegyik másik csomagja függésben van a *Common* csomagtól. Ebben például benne van a *Money* (pénz) nevű osztály, amit mindegyik csomag közösen használ. Mindehhez nem is kellett nyilakat rajzolni.

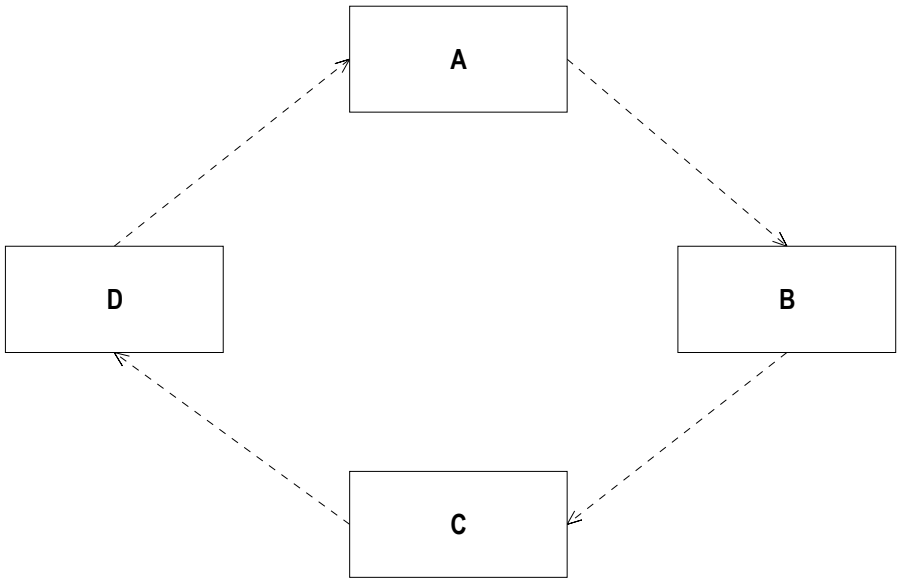
A harmadik bővítés az {*abstract*} megjelölésű *Database Interface* (adatbázis-interfész) csomag, ami egy általános interfész csomag. Ez nem végleges, csak egy átmeneti csomag, aminek a tervezésben van szerepe. Itt két adatbázis-interfész használatának a lehetősége szerepel: az *Oracle Interface* és a *Sybase Interface*. A terv véglegesítése után az absztrakt, általános csomag helyébe a konkrét, specifikus interfészcsomag fog bekerülni.



6.4. ábra. Kibővített csomagdiagram

A függőségi struktúrában előfordulhat, hogy az egymáshoz kapcsolódó függőségek egy zárt kört írnak le. Az ilyen kört *ciklusnak* vagy *huroknak* nevezzük. A jelenséget a 6.5. ábrán szemléltetjük.

Általánosan követendő szabályként lehet kimondani, hogy törekedni kell a ciklusok elkerülésére, vagy megszüntetésére. Ez nem mindig oldható meg, de ha nem lehet is megszüntetni mindegyik ciklust, akkor is minimális számúra kell csökkenteni őket. Az is egy jó elv, hogy ha nem tudunk megszüntetni egy ciklust, akkor próbáljuk meg a ciklusban levő csomagokat egy nagyobb, mindegyiket tartalmazó csomagba beleépíteni.



6.5. ábra. Függőségi ciklus

6.4. Használati szempontok

A csomagdiagramokat érdemes akkor is használni, amikor egy meglevő szoftver átalakítására van szükség. Ilyenkor meghatározzuk a meglevő osztályok között fennálló függőségeket, ezután az osztályokat csomagokba osztjuk be, és elemezzük a csomagok közötti függőségeket. Az elemzés után *átrendezést* (*refactoring*, lásd a 2. fejezetben) hajtunk végre, azzal a céllal, hogy csökkentsük a függőségek számát.

A csomagdiagramok használata feltétlenül szükséges és hasznos a nagy projektek esetében, ahol igen sok osztály szerepel a tervben. De kisebb méretű osztálydiagramok esetén is megérheti, ha támaszkodunk rájuk.

Mindig törekedni kell a függőségek számának minimalizálására, bár erre általánosan alkalmazható algoritmus, módszer egyelőre nem létezik. Csak intuitív, heurisztikus elvekre lehet támaszkodni. A jó megoldás elérésében sok függ az egyéni ügyességtől és tapasztalattól.

A csomagokat igen jól lehet felhasználni a szoftver tesztelése során. Ehhez kétféle megközelítést érdemes alkalmazni:

- Először az egyes osztályokhoz készítsünk, írjunk teszteket.
- A második menetben az egyes csomagokhoz, mint magasabb szintű egységekhez készítsük el és hajtsuk végre a teszteket.

7. Állapotmodellezés

A rendszerben feltárt objektumokat működésük során különböző hatások érik, amely hatásokra adott módon reagálnak, és eközben megváltozhat viselkedésük. Az objektumok tehát a külső hatások eredményeként megváltoztatják állapotukat, valamint hatnak környezetükre, amivel a környezetükben levő más objektumok állapotváltozását idézhetik elő.

A változó működést produkáló objektumok viselkedésének leírása és az állapotuk részletes definiálása fontos feladat. Különösen a tervezés és az implementáció szakaszában van ennek nagy jelentősége, amikor a rendszerünk procedurális részeit akarjuk megtervezni. Lesznek a modellben olyan osztályok, amelyeken a művelet végrehajtása attól függ, hogy az objektum milyen állapotban van.

Az objektumok állapota az attribútumaikban lesz eltárolva. Az állapotmodellezés segít az attribútum-lista végleges meghatározásában. Osztályokat leíró attribútumokat már az elemzés korai szakaszában definiálunk. Az elemzés során felállított lista azonban számos olyan attribútum-jellemzőt is tartalmaz, amelyekre a modellben valójában nincs szükség. Az állapotmodellezés segít a felesleges jellemzők kiszűrésében.

Az objektumok viselkedésének modellezésére az állapotátmeneti diagramok szolgálnak. Az állapotátmeneti diagramok ábrázolására a módszertanok különböző megoldásokat javasolnak. Az egyes megoldások alapvetően szemantikájukban különböznek egymástól. Az UML állapotátmeneti diagramja a David Harel[17] által kidolgozott módszer logikáján alapszik.

7.1. Az állapotátmeneti diagram

Az *állapotátmeneti diagram* az objektumok dinamikus viselkedésének leírására szolgáló eszköz. Segítségével egy-egy objektumot elemezhetünk, arra keresve a választ, hogy az objektum az események hatására hogyan változtatja meg a belső állapotát, valamint az események hatására milyen üzeneteket küld a többi objektum számára. Egy állapotdiagramot mindig egy osztályobjektumhoz készítünk, ill. egy magasabb szintű diagram pontosításaként. A fejlesztés során csak az intenzíven dinamikus (változó) viselkedésű objektumok állapotátmeneteit érdemes modellezni. A gyakorlatban a

[17] A David Harel által kidolgozott állapotleírást az OO-módszertanok közül elsőként az OMT (Object Modeling Technique) módszertan alkalmazta, majd 1994-ben Booch is beépítette módszertanába.

legtöbb osztályhoz nem készül állapotátmeneti diagram, mert azok a rendszerben csak kisegítő ún. utility osztályok.

Az állapotátmenetet modellező diagram szerepét megfogalmazhatjuk tágabb és szűkebb értelemben:

- Tágabb értelemben:
  - a rendszer dinamikus nézetét jellemző technika.
  - az állapotátmeneti diagramokkal leírható egy rendszer viselkedése.
- Szűkebb megközelítésben: egyetlen osztályhoz kapcsolódó fogalom, amely bemutatja az osztály konkrét objektumának élettartama (a rendszerben levő) alatt mutatott viselkedését, vagyis:
  - azokat a lehetséges állapotokat, amelyekbe az objektum kerül, jut, és
  - ahogy az objektum állapota változik (állapotátmenet) az őt érő események hatására.

Az állapotok leírásában az eseménykövetési diagramok is segítenek. Az állapotok feltérképezéséhez ki kell gyűjteni az adott osztály objektumait reprezentáló függőleges vonalakat (életvonal – lifeline), a kapott és küldött üzeneteket jelölő nyilakkal együtt. A kapott üzenetek forrása és a küldött üzenetek célja az állapotmodell szempontjából lényegtelen. Egyetlen eseménykövetési diagramrészletből kell kiindulni, végig kell nézni az objektumhoz érkező, és az objektumot elhagyó üzeneteket. Az állapotok két esemény között írják le az objektumot. Ha két egymást követő állapot között az objektum kívülről kap üzenetet, akkor ezt az eseményt kell megadni az állapotátmenet eseményeként. Amennyiben a két állapot között az objektum küld üzenetet, az üzenet elküldését az átmenet során elvégzendő akcióként kell megadni. Vannak esetek, amikor az eseménykövetési diagram ciklusokat tartalmaz. Ilyenkor a bejárt állapotokat csak egyszer kell létrehozni, és az állapotátmeneteket úgy kell kialakítani, hogy a ciklus végén az objektum újra a ciklus elején lévő állapotba kerüljön.

7.2. Az objektum állapota

Az objektumok a valós dolgokat leíró elemek a rendszerben. A valós dolgokhoz hasonlóan az objektumoknak is van létük, mindegyik valamikor, valaminek a hatására létrejön, egy bizonyos ideig létezik, majd megszűnik. Az objektumok kommunikálnak egymással. A kommunikáció során az objektum detektálja az őt ért eseményeket, illetve reagál az eseményekre. Egy esemény hatására az objektum állapota megváltozhat. Az objektumok

a felvett állapotokat csak véges ideig tartják, ez alatt az idő alatt az objektum az adott állapotban van. Egy objektum pillanatnyi állapotát jellemzőinek (az adott pillanatban felvett tulajdonságértékek, és az objektumnak a környezetében levő más objektumokkal megvalósított kapcsolatai) pillanatnyi értékei határozzák meg.

Az állapot

- Az állapot az objektum életének egy szakaszát jellemzi. Az objektum pillanatnyi állapota alatt az adott időpontban felvett attribútumértékeinek és aktuális kapcsolatainak együttes halmazát értjük, ezek alapján határozhatjuk meg azt.
- Az állapot két esemény közötti időtartamig érvényes, egy nem nulla hosszú időintervallum, ami alatt az objektum valamely esemény bekövetkezését várja, vagy ami alatt az objektum folyamatosan végrehajt egy akciót.
- Egy adott állapotban levő objektum ugyanarra az eseményre mindig ugyanúgy reagál, vagyis ugyanazt az akciósorozatot hajtja végre.
- Az állapotokat lekerekített téglalapok szimbolizálják.
- Az állapotok közötti átmeneteket nyilak jelölik.

Az objektumok viselkedését leíró állapotátmeneti diagram pontosan egy kiinduló állapotot, egy vagy több végállapotot, valamint tetszőleges számú belső állapotot modellez. A kiinduló állapotot egy teli kör szimbolizálja, a belső állapotokat lekerekített téglalapok jelölik. A végállapot/végállapotokat egy üres körben teli kör ábrázolja. Az állapotokat reprezentáló téglalap két részre osztható. A felső részben az állapotra utaló név szerepel, az alsó rekesz az állapothoz tartozó belső állapotátmenetet jelöli. Itt tevékenységek megadására van lehetőség, amelyeket az objektumok különböző üzenetek (események) hatására hajtanak végre, miközben állapotuk nem változik.

7.3. Az állapotátmenet

Az állapotmodellezés témakörben az állapot mellett másik fontos fogalom az állapotátmenet értelmezése. Az átmenet az objektum válasza, reakciója egy külső eseményre. Az objektumokat életük során hatások, ún. események érik. Az objektumok a bekövetkezett eseményeket érzékelik. Az események hatására állapotuk megváltozhat, vagyis az aktuális állapotból másik állapotba kerülhetnek. Az állapotátmenetet tehát egy esemény váltja

ki. Az állapotátmenet általában eljáráshívással jár együtt, vagyis valamilyen tevékenység végrehajtásával. Az állapotátmenetnek két típusáról beszélünk:

- külső állapotátmenet: Az objektum állapotai között értelmezzük, az állapotokat összekötő nyíllal jelöljük. Az aktuális állapotból másik állapotba vezet. Eljáráshívás vagy történik, vagy nem.
- belső állapotátmenet: Az objektum egy adott állapotában értelmezzük. Eljáráshívással jár együtt anélkül, hogy az objektum állapota megváltozna. A belső átmenetet az állapot alsó rekeszében adjuk meg.

Az állapotátmenet vezethet az aktuális állapotba is, azaz az állapotátmenet diagram szintjén az állapotátmenet nem feltétlenül jár együtt állapotváltozással.

Az állapotátmenethez megadhatók pluszjellemzők. Megadásuk opcionális:

- Esemény (trigger vagy event): Az átmenet a kiváltó esemény hatására következik be. Az esemény egyaránt jöhet kívülről vagy belülről.
- Feltétel (guard): Az átmenethez tartozhat feltétel. A feltétel egy logikai kifejezés az objektum attribútumok és üzenet-paraméterek felhasználásával. Az átmenet csak akkor következik be, ha a megadott feltétel igaz. A feltétel megadható az OCL[18] segítségével.
- Akció (action): Az objektum által végzett elemi műveletek, amik az átmenetkor hajtódnak végre.
- Szintaxisa: esemény [feltétel] / akció – Event [condition] / action.

A továbbiakban tekintsük meg az állapotátmenethez hozzárendelhető pluszjellemzőket egyenként, részletesebben!

7.3.1. Esemény

- Esemény vagy üzenet: egy objektumnak egy másik objektumra történő hatása. Az objektum adott állapotából adott esemény hatására másik, új állapotba kerülhet.
- Az állapotátmenetet események idézik elő.
- Az esemény egyetlen időpillanathoz és nem időtartamhoz kapcsolódik. Elemi dolog, nem szakítható meg.

[18] Object Constraint Languge

- Ha egy átmenethez nincs esemény megadva, akkor esemény nélküli állapotváltásról beszélünk. Ekkor az objektum az állapothoz rendelt tevékenységet végrehajtva automatikusan a következő állapotba kerül.
- Nem minden esemény okoz feltétlenül állapotátmenetet.
- A modellben le nem kezelt események az objektum számára elvésznek.

7.3.2. Feltétel

- A feltétel (őrszem – guard/condition) az állapotátmenet feltétele.
- A feltétel egy megadható logikai kifejezés. A feltétel értékének az állapotátmenethez igaznak kell lenni.
- A feltétel maga is az objektum állapotára utal, de ezt nem jelöljük külön névvel.
- A feltételt szögletes zárójelek [feltétel] között az esemény neve után adjuk meg.
- Az objektum egy adott állapotából vagy egy esemény hatására, vagy automatikus átmenettel csak akkor kerül az átmenet által meghatározott állapotba, ha teljesíti az őrszemként megadott feltételt, vagyis ha az átmenet feltételhez van kötve, az állapotváltozás csak az esemény bekövetkezése és a feltétel igaz értékének bekövetkezése esetén történik meg.
- Ha a feltétel egy esemény nélküli átmenethez tartozik, akkor az átmenet a feltétel igazzá válásakor rögtön végrehajtódik.
- A feltétel hamis értéke esetén az objektum állapota nem változik.

7.3.3. Tevékenységek, akciók

Az állapotokhoz megadhatók végrehajtandó tevékenységek. Az objektumok meghatározott ideig vannak egy adott állapotban, eközben különböző tevékenységeket végezhetnek. Az akció az objektum által végzett elemi műveletek halmaza, amiket az átmenetkor kell végrehajtani. A tevékenységek és az akciók a rendszer működtetése során végrehajtandó feladatok, műveletek (eljárások), amelyek az objektum metódusaként implementálhatók.

Az UML igazán nem tesz különbséget a két fogalom között, mégis lehet következtetni, hogy elemi műveletről (akció) vagy tevékenységről van szó (lásd 7.1. táblázat). Például az állapothoz tartozó tevékenységet félbeszakíthatja egy külső esemény (azonban a kimeneti tevékenység ekkor is végrehajtódik), az akció nem szakítható félbe, mert nem rendelkezik időtartammal.

7.1. táblázat. Tevékenységek, akciók

Tevékenységek	Akciók
A tevékenységek végrehajtási idővel rendelkező műveletek: • Elemi műveletekből álló tevékenységek. • Összetettebbek, további lépésekre bonthatók. • Események által megszakítható tevékenységek a végrehajtás folyamatában. • Az elemi műveletekből (akciók) álló tevékenységek elvégzéséhez meghatározott időre van szükség – ezért az állapotokhoz kapcsolódnak.	Az akciók pillanatnyi műveletek: • Az objektum által végzett elemi műveletek. • A funkcióhierarchia legalsó szintjén elhelyezkedő elemi műveletek, ún. atomi műveletek – további lépésekre nem bonthatók. • Nem szakíthatók meg, nem rendelünk hozzájuk időtartamot – ezért az átmenethez kapcsolhatók. • pl. számítási műveletek, objektum manipulálására vonatkozó kérések.

Tevékenységek (activities)

- A tevékenység állapothoz tartozó művelet, aminek végrehajtása időt vesz igénybe, tehát a tevékenységekhez időtartam tartozik.
- A tevékenységek végrehajtása alatt az objektum állapota nem változik.
- A tevékenységekhez kapcsolhatók belépési pontok, kilépési akciók.

A rendszerben történnek események, amelyek nem okoznak közvetlen állapotváltozást, vagyis hatásukra az objektum az adott állapotban marad (belső állapotátmenet). Az események általában valamilyen tevékenységek, akciók végrehajtásával járnak együtt. Az állapotokkal kapcsolatban többféle akció különböztethető meg, egy állapothoz rendelhető:

- bemeneti akció – entry action.
- kimeneti akció – exit action.
- belső akció – internal action.
- belső állapotátmenet – internal state-transition.

Bemeneti akció

- Az *entry/* jelölés után adható meg.
- A bemeneti akció az állapotba való belépéskor végrehajtandó elemi utasítást definiálja.
- Azt az esetet rövidíti, amikor az állapotba vezető minden átmenet esetén azonos tevékenységet akarunk végrehajtani.

Kimeneti akció

- Az *exit/* jelölés után kell megadni.
- A kimeneti akció az állapot elhagyásakor végrehajtandó elemi utasítást definiálja.
- Azt az esetet rövidíti, amikor az állapotból kivezető minden átmenet esetén azonos tevékenységet akarunk végrehajtani.

Belső akció

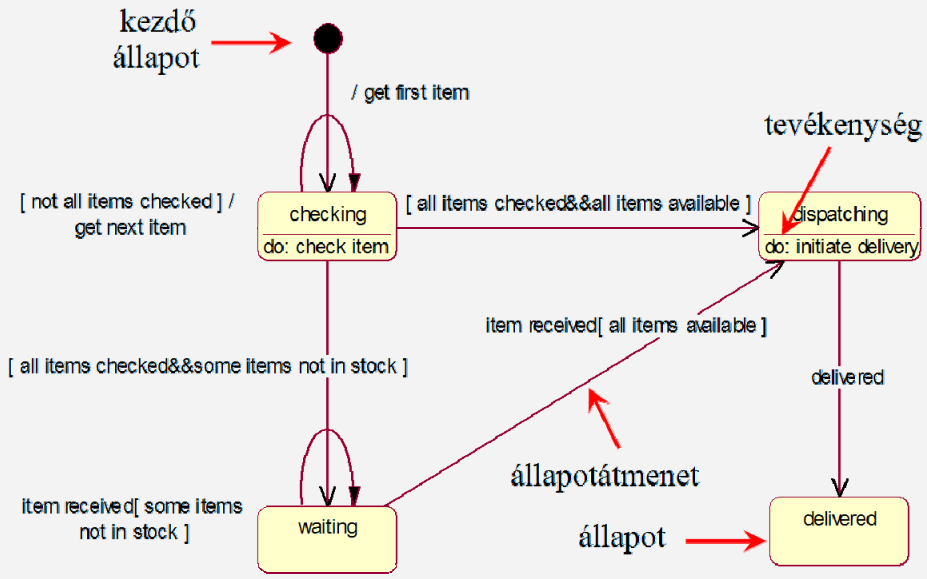
- Az objektum adott állapotban tevékenységeket végezhet. A végrehajtandó tevékenységeket a *do prefix* után kell megadni.
- A tevékenység végrehajtásához időre van szükség, ez alatt az idő alatt az objektum állapota nem változik.

Belső állapotátmenet

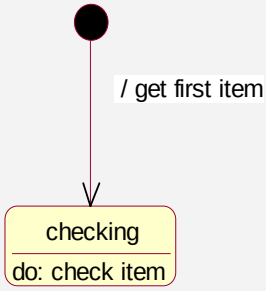
- A belső átmenetek azok, amikor az állapotból kiinduló, az eseménnyel címkézett él ugyanabba az állapotba tér vissza, de ekkor nem hajtódik végre sem kimeneti, sem bemeneti akció, mivel az objektum ugyanabban az állapotban maradt.
- A belső átmenet egy adott esemény hatására kerül végrehajtásra. Az esemény egy akciót vált ki, amit végre kell hajtani anélkül, hogy az objektum állapota megváltozna.
- Belső átmenet megadásakor megadjuk az esemény nevét, paraméterlistát kerek zárójelekben, szögletes zárójelben definiáljuk a feltételt, a feltételt követő perjel után a végrehajtandó akció nevét: *event (arguments) [condition] / action.*

Az Order objektum állapotai

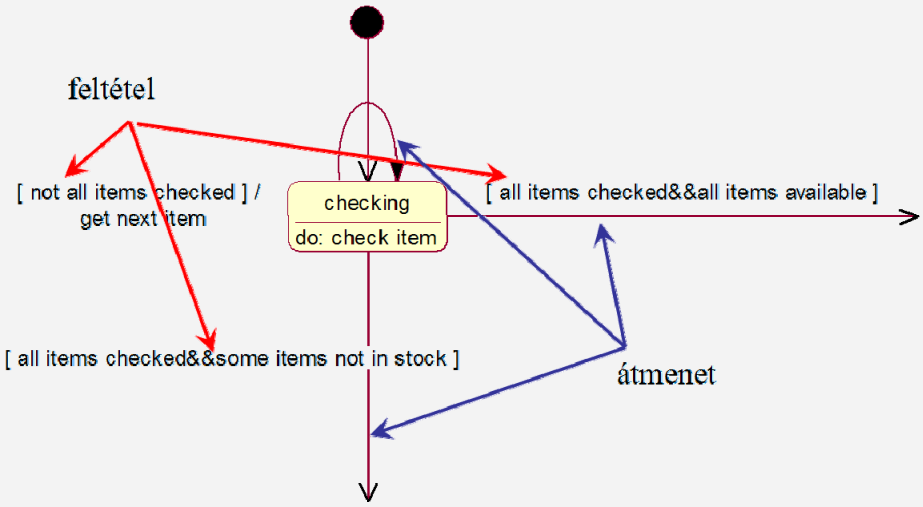
Az Order objektum lehetséges állapotait az alábbi állapotátmenet diagram szemlélteti.



A diagram mutatja a kezdő állapotot (start point): az objektum létrejöttekor automatikusan a kezdő állapotba lép. A kiindulási/kezdő *átmenethez* csak egy jellemző tartozik, a„/ get first item” akció. Ez az átmenetkor végrehajtandó akció, elsőként ezt az *akciót* kell végrehajtani a checking állapotba való belépéshez.

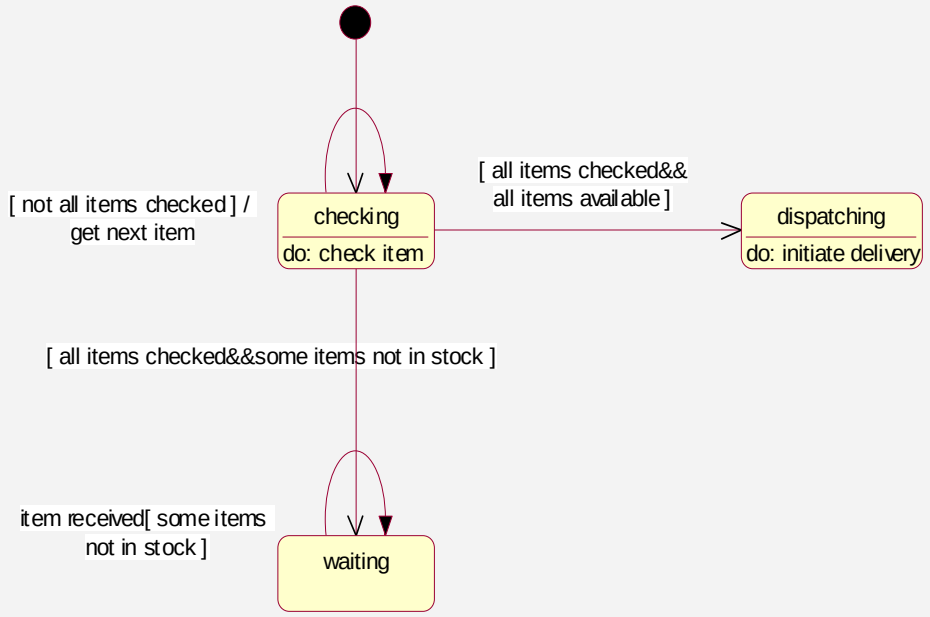


A checking állapotból három átmenet látható. Mindhárom átmenethez feltétel, őrszem tartozik.

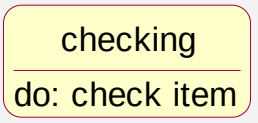


Egy adott állapotból csak egy átmenet következhet. A példában a checking állapotból három átmenet látható:

- Rendeléskor ellenőrizni kell, hogy létező termékről van-e szó: checking állapot (A cég kínálatában mindig újabb és újabb termékek jelennek, bizonyos termékek gyártása pedig megszűnhet). Az ellenőrzés egészen addig tart, amíg a rendeléshez tartozó összes terméket le nem ellenőrizte a rendszer. A modellben ezt a /get next item akció és a checking állapotba való visszatérés mutatja.
- Ha minden terméket leellenőriztünk, de azok közül nem mindegyik van raktáron az adott mennyiségben, akkor az objektum a waiting állapotba kerül.
- Ha minden terméket leellenőriztünk, és azok mindegyike raktáron is van, az objektum a dispatching (a rendelés feladása) állapotba kerül.

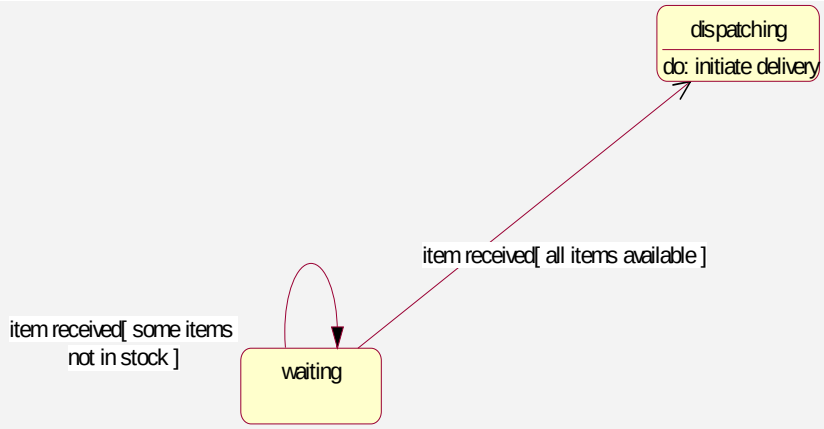


Az Order objektum a checking állapotban „check item” tevékenységet végez.



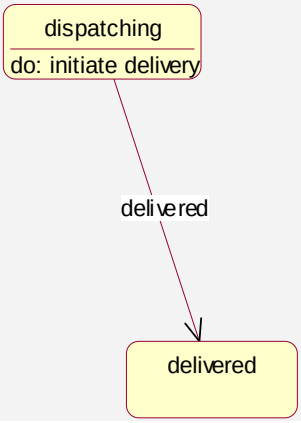
A waiting állapot jellemzése:

- A waiting állapothoz nincsenek megadva végrehajtandó tevékenységek.
- Az adott Order objektum a waiting állapotban tartózkodik egy esemény bekövetkezésére várva.
- A waiting állapotból két kimenő állapotátmenet mindegyikéhez az item received esemény tartozik, ami azt jelenti, hogy az order addig vár, amíg nem észleli az eseményt, vagyis az esemény bekövetkezésére vár.
- Az Order kiértékeli az átmeneten levő feltételeket, majd végrehajtja a megfelelő tranzakciót:
  - vagy visszamegy a waiting állapotba,
  - vagy az order a feltételek kiértékelésének eredményeként a dispatching állapotba kerül.



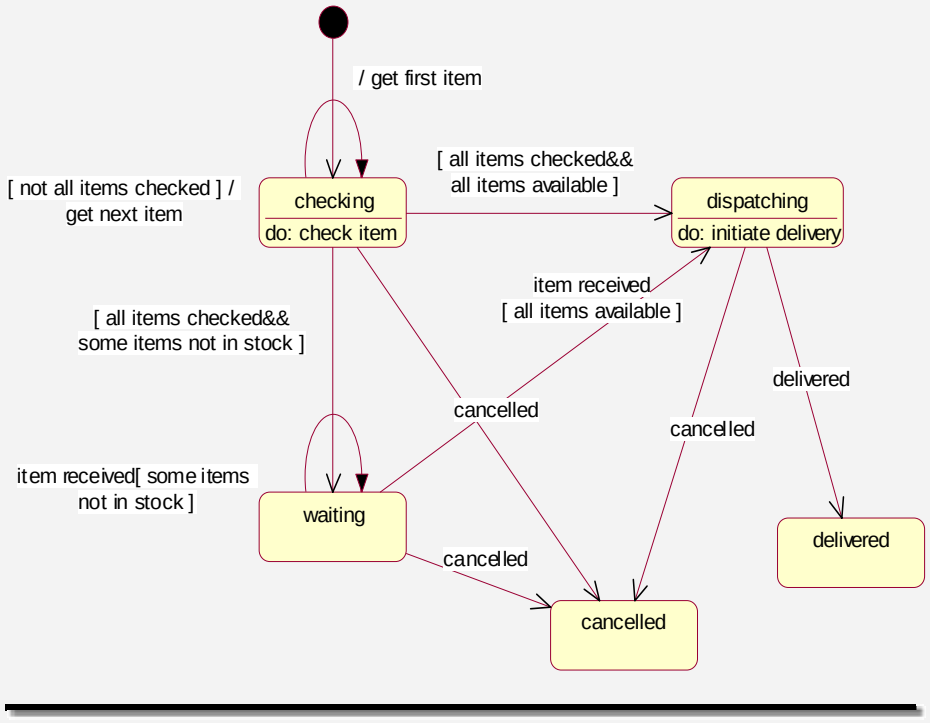
A dispatching állapot

- A dispatching állapotban meg van adva egy végrehajtandó tevékenység, amit a do prefix jelez. Az objektum a dispatching állapotban initiate delivery – szállítás elindítása tevékenységet végez.
- Létezik egy feltétel nélküli átmenet a delivered eseménnyel triggerelve. Ez azt jelzi, hogy az átmenet feltétel nélkül mindig bekövetkezik, amikor a delivered esemény bekövetkezik, azonban az átmenet nem következik be automatikusan, ha a tevékenység befejeződött.
- Ha az initiate delivery tevékenység befejeződött, az adott order egészen addig marad a dispatching állapotban, amíg a delivered esemény be nem következik.



A cancelled átmenet

- A rendszernek lehetőséget kell adni arra, hogy a rendelési folyamat bármely pontjában töröljük a megadott rendelést, mielőtt a termékek szállításra kerülnének.
- A modellben ezt úgy oldjuk meg, hogy mindegyik állapotból: checking, waiting, dispatching biztosítunk egymástól elkülönített átmeneteket a cancelled állapotba.



Az állapotátmeneti diagram sok esetben bonyolulttá válhat. A diagramot finomíthatjuk, aminek eredményeként beszélhetünk:

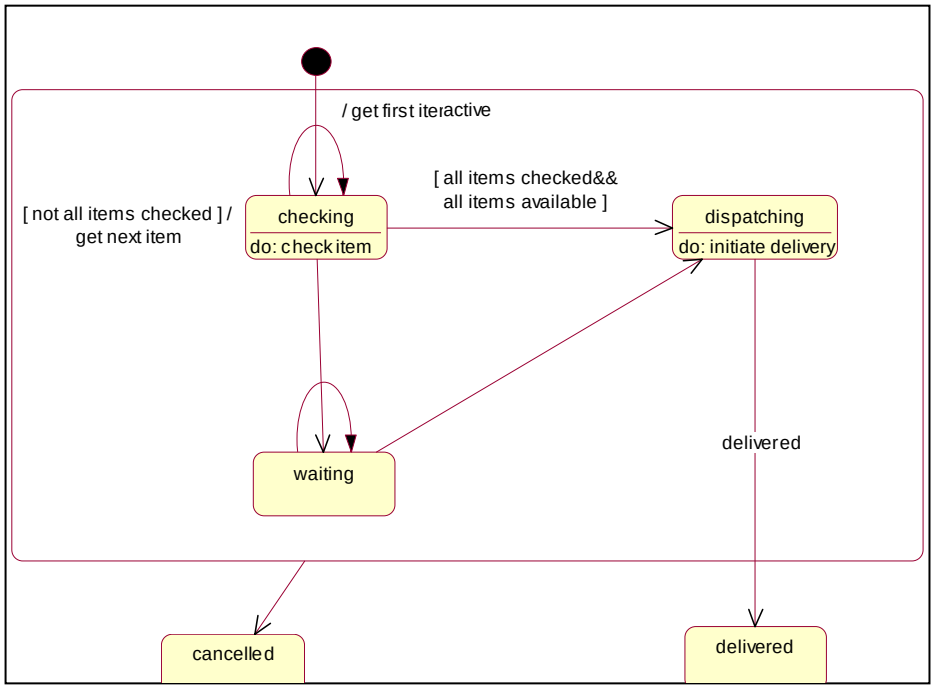
- *állapothierarchia* viszonyról: öröklődési viszony definiálható az állapotok között.
- *konkurrens viselkedésről:* párhuzamos állapotfolyamok (szálak) egyidejű végrehajtását jelenti.
- *emlékező állapotról:* megszakítás esetén a legutolsó aktív állapotba való visszatérést biztosítja.

7.4. Állapothierarchia – Szuperállapot, szubállapotok

Ha vannak a diagramban olyan állapotok, amelyek logikailag egy nagyobb egységbe összefoghatók, akkor ezeket az összetartozó állapotokat érdemes egy ún. szuperállapotba kiemelni, és ebben az egységben kezelni. A logikailag együttkezelt állapotokat alállapotoknak vagy szubállapotoknak nevezzük, a létrejött új állapot a szuperállapot. A szubállapotok és a kiemelés eredményeként létrejött szuperállapot közötti viszony egyfajta öröklődési hierarchiát eredményez. Az alállapotok öröklik a szuperállapot átmeneteit (a példában a *cancelled* átmenetet), ha azt nem definiálják át. Továbbá egy szuperállapotban levő objektum lehet a szubállapotok bármelyikében.

Szuper- és szubállapotok

Az Order objektum a checking, waiting, dispatching állapotok mindegyikéből a cancelled esemény hatására cancelled állapotba kerülhet. Ebben az esetben érdemes a checking, waiting, dispatching állapotokat egy nagyobb egységbe összefogni. Érdemes létrehozni a három állapotnak (szubállapotok) egy szuperállapotát, majd ebből megrajzolni egyetlen átmenetet a



cancelled állapotba. A modellben a szuperállapotnak az active nevet adtuk, jelentésében megkülönböztetve azt a cancelled állapottól. A checking, waiting, dispatching szubállapotok öröklik az active szuperállapot cancelled átmenetét. Az active szuperállapotban levő order objektum a szubállapotok bármelyikében lehet, egyidejűleg azonban csak az egyik szubállapotban.

A szubállapotok egy meghatározott szuperállapoton belül vagy:

- szekvenciálisan követik egymást:
  - az állapotok diszjunkt állapotok,
  - a szuperállapotot egy adott időpontban az alállapotok közül mindig csak egy határozza meg, vagy
- párhuzamos végrehajtásúak – konkurrens szub-állapotok.

7.5. Konkurrens állapotdiagram

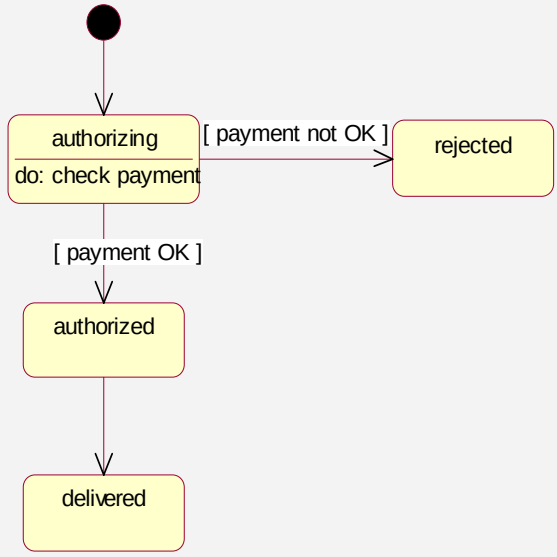
- Olyan műveletsorok (állapotfolyamok) végrehajtását jelenti, amelyek párhuzamosan végezhetők egymással.
- A párhuzamos állapotfolyamok mindegyikének külön állapota van – az egy szuperállapotba zárt, konkurrens állapotgépeknek saját induló és végállapotaik vannak.
- A szuperállapot akkor következik be, amikor a benne definiált összes párhuzamos állapotfolyam befejeződik, vagyis eléri a végállapotot – ha valamelyik szál előbb fejeződik be, az vár.
- A konkurrens állapotmodellezés hasznos, ha egy objektumnak egymástól független viselkedéscsoportjait akarjuk leírni.

A modellben törekedjünk minél kevesebb konkurrens szubállapot kialakítására. Ha túl bonyolult egy objektumhoz tartozó konkurrens állapotdiagram, érdemes az objektumot további objektumokra szétszedni.

Konkurrens állapotdiagram

A megrendelés során ellenőrizni kell a fizetőképességet, amit a modellben a Payment authorizing (fizetésengedélyezés) állapotátmenet diagrammal modellezünk. A diagram értelmezése:

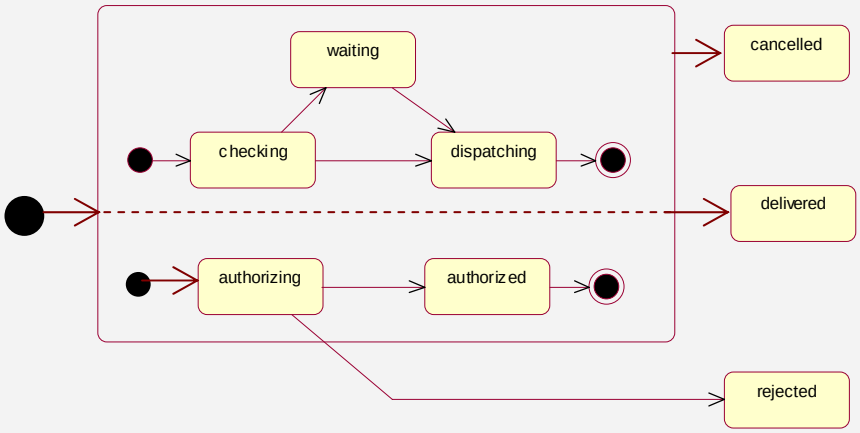
- Az Order objektum az authorzing állapotban check payment tevékenység hajt végre.
- A tevékenység egy signal-lal fejeződik be, ami jelzi, hogy a payment rendben van-e.
- Ha a payment rendben: OK, a feltétel értéke igaz, akkor az adott order objektum addig vár az authorized állapotban, amíg a delivered esemény be nem következik.
- Abban az esetben, ha a feltétel értéke hamis, az objektum a rejected állapotba kerül.



A rendszerben definiált Order állapotai:

- egyrészt a rendelési tételek elérhetőségén alapulnak,
- másrészt léteznek olyan állapotok is, amelyek a fizetés engedélyezésén alapulnak.

Az Order viselkedésének modellezésekor két állapotfolyamot azonosítottunk. Az egyik szál a termékekre vonatkozó adatokat vizsgálja, a másik szál a likviditási vonalat ellenőrzi.

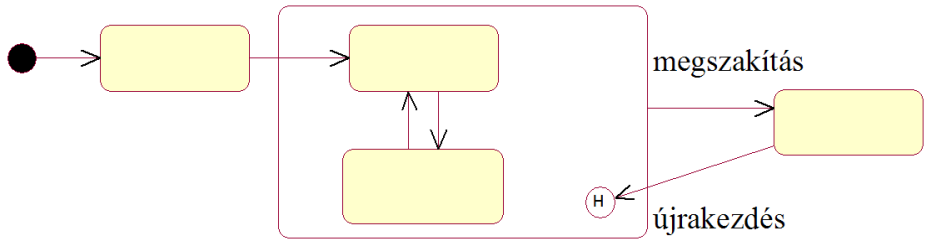


A konkurrens állapotátmenet diagramban:

- ha az authorizing állapot check payment tevékenysége sikeresen befejeződött, az objektum a checking és az authorized állapotba kerül – párhuzamos végrehajtású szálak.
- Az objektum a párhuzamos folyamok bármely állapotából cancelled állapotba kerülhet.
- A párhuzamos szálak mindegyikének befejeződése után a műveletsor kilép a szuperállapotból és a végrehajtás ismét egy ágban fog folytatódni.

7.6. Emlékező állapot

Az *emlékező állapot (history state)* különleges állapot. Emlékszik arra, hogy melyik alállapotból terminált, és képes arra, hogy az állapotba való újabb belépéskor ugyanabba az állapotba kerüljön. Az emlékező állapot a legutolsó aktív állapotkonfigurációt tárolja. A 7.1. ábra emlékező állapotot is tartalmazó állapotátmenet diagramot szemléltet.



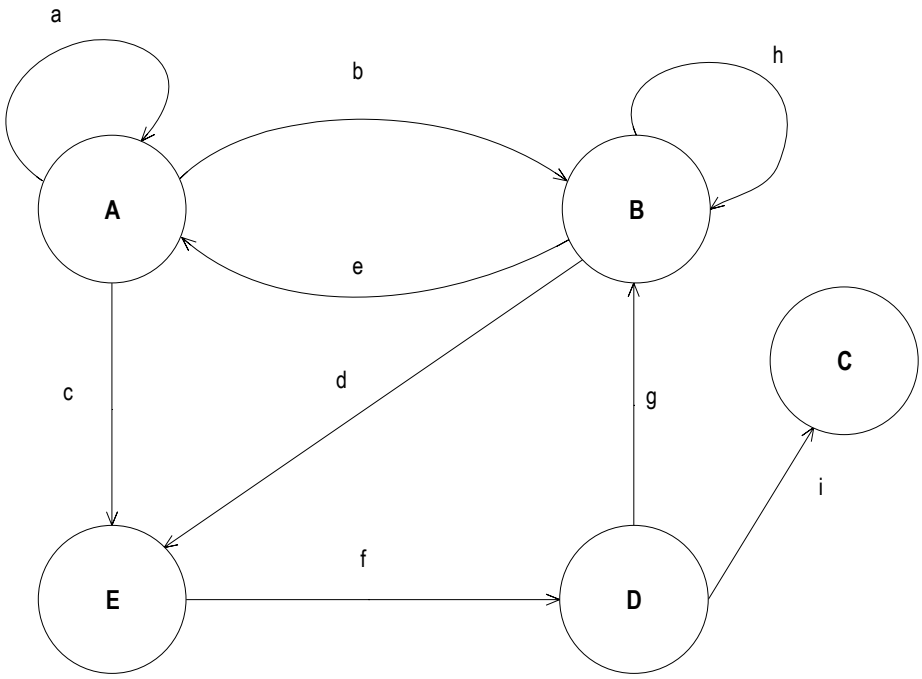
7.1. ábra. Emlékező állapot

Az állapotmodellezés során az objektumok viselkedésének, és ez által a rendszer viselkedésének leírására van lehetőség. Az objektumok életük alatt különböző állapotokba kerülhetnek, és ott különféle feladatokat, tevékenységeket végezhetnek. Az állapotokat, az állapot változást kiváltó eseményeket, azok sajátosságait az állapotátmeneti diagramban foglaljuk össze. Az állapotátmeneti diagram igazán hasznos aszinkron történések leírására. Hatékony eszköz az osztály attribútumainak és műveleteinek meghatározáskor. Az állapotmodellezés nagy segítséget jelent az egyes metódusok kódolásakor is.

7.7. Objektumok tesztelése állapotdiagram alapján

A rendelkezésre álló állapotdiagramokat jól fel tudjuk használni az objektumok működésének vizsgálatakor, vagyis a teszteléskor. A tesztelés célja ilyen esetben az állapotdiagram összes állapotának a bejárása, másrészt pedig több lehetséges kombinációjú állapotbejárás kivitelezése. A különböző állapotbejárások ugyanis különböző működési hibák felfedésére lehetnek alkalmasak.

A 7.2. ábrán egy állapotdiagramot látunk, ami absztrakt ábrázolásban egy irányított gráf, öt állapottal. Az állapotokat nagybetűvel jelöltük, az átmeneteket mutató éleken pedig kisbetűket tüntettünk fel a vezérlő események jelölésére. Például: az **f** esemény hatására az objektum az **E** állapotból a **D** állapotba jut át.



7.2. ábra. Állapotátmeneti diagram

A bejárások által képviselt tesztek közül az alábbiakban mutatunk be hármat. Feltételezve, hogy az **A** állapotból indulunk ki, a következő eseménysorozatokat írjuk elő:

T1: a – b – e – c – f – g – h – d – f – i

Ez egy teljes bejárás, ami az összes állapotot magában foglalja.

T2: a – a – c –f – g – h – h – h – d – f – g – e – c – f – i

Ez egy másik teljes bejárás, ami ismétléseket is tartalmaz.

T3: b – h – e – c – f – g – h – d – f – g – b – a – a

Ez egy nem teljes bejárás, ismétlésekkel.

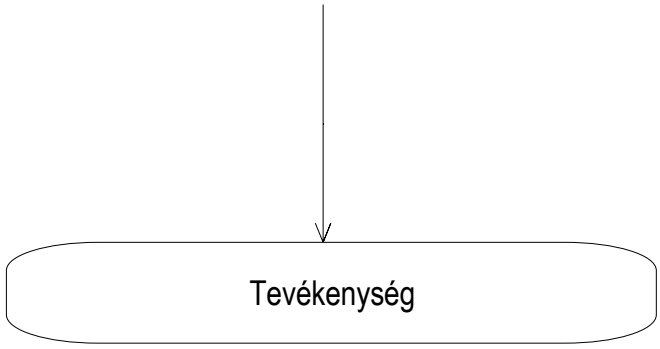
Mindehhez még annyit fűzünk, hogy a tesztsorozatokat nem csak az **A** állapotból kiindulva tervezhetjük meg.

8. Aktivitási diagramok

8.1. Folyamatok, tevékenységek leírása

A mérnöki világban és az informatikában nagyon sokszor van szükség arra, hogy működési folyamatokat írjunk le, tevékenységek menetét mutassuk be. Erre a célra az UML-ben az ún. *aktivitási diagramok* vagy *tevékenységi diagramok* szolgálnak. Angol elnevezésük: *activity diagram.*

Az aktivitási diagramok kialakulásának számos előzménye volt. Eseménydiagramok, folyamatábrák, különböző állapotmodellezési megoldások, valamint az ún. *Petri-hálók* stb., amelyek mindegyike valamilyen formában folyamatok leírására volt alkalmas. Ezek a diagramok különösen akkor használhatók jól, amikor munkafolyamatok megjelenítésére van szükség, vagy amikor egymással párhuzamosan, vagyis egyidejűleg futó folyamatokat kell ábrázolni. Az UML nyelvben definiált aktivitási diagram is ilyen célokra alkalmas. Ebben a központi szerepet játszó szimbólum az *aktivitáshoz (tevékenységhez)* tartozik. Ábrázolása a 8.1. ábrán látható. A szimbólumban a mindenkori konkrét tevékenység megnevezése szerepel.



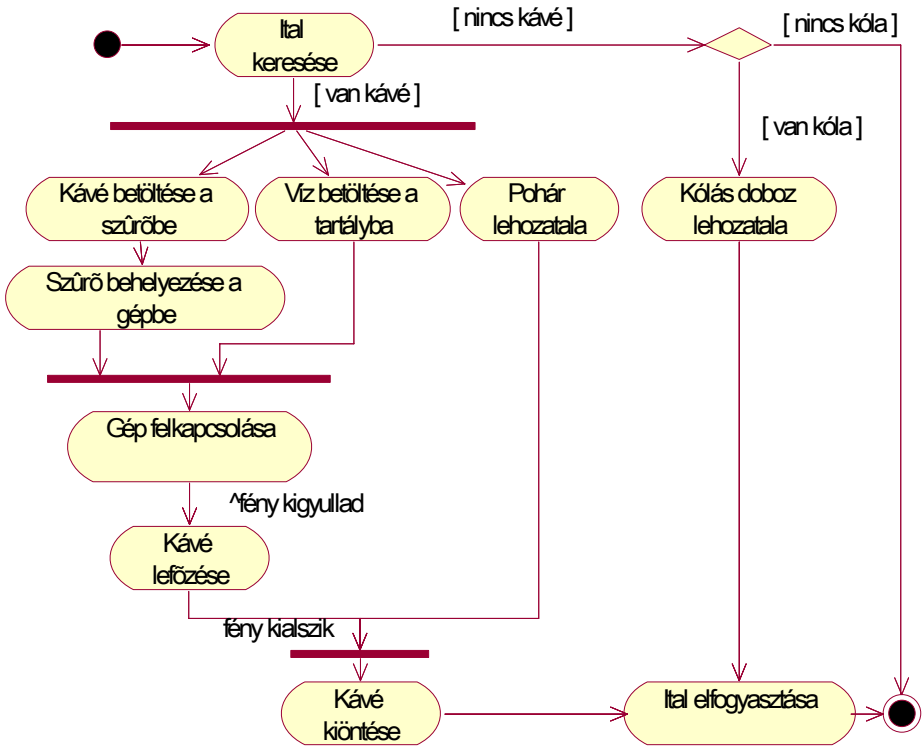
8.1. ábra. A tevékenység szimbóluma

Koncepcionális megközelítésben a tevékenység egy bizonyos feladat, amit el kell végezni, vagy egy személynek, vagy pedig egy számítógépnek.

Specifikációs megközelítésben, vagy implementációs megközelítésben a tevékenység egy művelet vagy metódus egy osztályon belül.

A tevékenységek egymást követik, kapcsolódhatnak egymáshoz, a kapcsolatokat nyíllal ellátott összekötő vonalak jelölik. A 8.2. ábrán egy olyan aktivitási diagramot láthatunk, ami egy italkiadó automata használatát mu-

tatja be. Az automata vagy kávét főz, vagy hideg kólát ad ki, a kiindulási tevékenység *(Ital keresése),* valamint a többi kapcsolódó tevékenység, művelet elvégzése után.



8.2. ábra. Egy aktivitási diagram

Az első tevékenység két irányban ágazhat szét, ami logikai elágazás: A személy italt keres. Ha nem kávét akar, akkor a kóla után néz. Ekkor a folyamatban egy olyan döntési helyzet áll elő, amit egy rombusz szimbolizál: van kóla, vagy nincs kóla. Ha van, akkor a következő tevékenység a doboz kinyerése, majd tartalmának elfogyasztása. Ha nincs, akkor a folyamat véget ér. A végső, kilépési pontot egy telt kör jelzi, ami egy külső körrel van keretezve. A diagram kezdő pontja egy sima telt kör.

A kávé irányában indulva egy fontos szimbólum található, az ún. *szinkronizációs vonal,* ami egy kivastagított vízszintes szakasz. Ez a vonal arra szolgál, hogy megszabja azt az időpontot, amikor a belőle kiágazó események egyáltalán elindulhatnak. Az ábrán három ilyen esemény indulhat el az első vonal után: *Kávé betöltése a szűrőbe, Víz betöltése a tartályba, Po-*

hár lehozatala. Ez a három tevékenység egymással párhuzamosan folyhat le, ahol a sorrendjük tetszőleges lehet. Mindegy, hogy melyikük megy végbe először, másodszor, vagy harmadszor, de át is fedhetik egymást, vagyis egyidejűleg is végbemehetnek.

Az aktivitási diagram tehát lehetőséget ad arra, hogy megválasszuk a tevékenységek végrehajtásának sorrendjét, mármint egy szinkronizált csoporton belül. A diagram egy lényegi sorrendiséget állít fel, amit követni kell, nem pedig egy lebontott, részletesen meghatározott sorrendiséget. Ez fontos különbség a folyamatábra és az aktivitási diagram között. A szoftvertervezésben használt folyamatábrák a szigorúan szekvenciális folyamatok leírására szolgálnak, míg az aktivitási diagram a párhuzamos folyamatokat is tudja kezelni.

Ez a lehetőség jól kihasználható az üzleti folyamatok modellezésénél. Az ilyen folyamatokban igen gyakran alkalmazható a párhuzamos végrehajtás, ami hatékonyabbá teszi a modellezést. Az aktivitási diagram használata megengedi a tervezőnek, hogy éljen a párhuzamosítás lehetőségével, ami javítani tud az üzleti folyamatokon.

Az aktivitási diagramok igen hasznosak lehetnek az ún. *konkurrens programok* esetében, ahol különböző *szálakon* futnak az események. Ezeket a szálakat grafikusan tudjuk ábrázolni, és meg tudjuk adni, hogy mikor kell őket szinkronizálni, egybehangolni. Amikor ugyanis párhuzamos működés áll fenn, mindig szükség van a szinkronizálásra.

Az adott példában: Addig nem lehet felkapcsolni a kávéfőzőt, amíg nem tettük bele a szűrőt, és nem töltöttük bele a vizet a tartályba. Ezért van az, hogy a két tevékenység egy szinkronizációs vonalon találkozik. A vonal itt azt jelenti, hogy csak akkor léphet fel a kijövő esemény a vonal után, ha már mind a két bejövő esemény lezajlott.

Egy újabb szinkronizáció: A kávé meg kell legyen főzve, és a poharak is rendelkezésre kell hogy álljanak, mielőtt kiöntenénk a kávét.

A másik esetben, amikor is a kóla irányában indulunk el, egy döntési elágazás szerepel. Itt az egyik eset az, amikor a *Kólás doboz lehozatala* tevékenység megy végbe, a másik eset pedig az, amikor nincs kóla. Az aktivitási diagramban egymásba ágyazott döntések is meg vannak engedve, tetszőleges számban és mélységben.

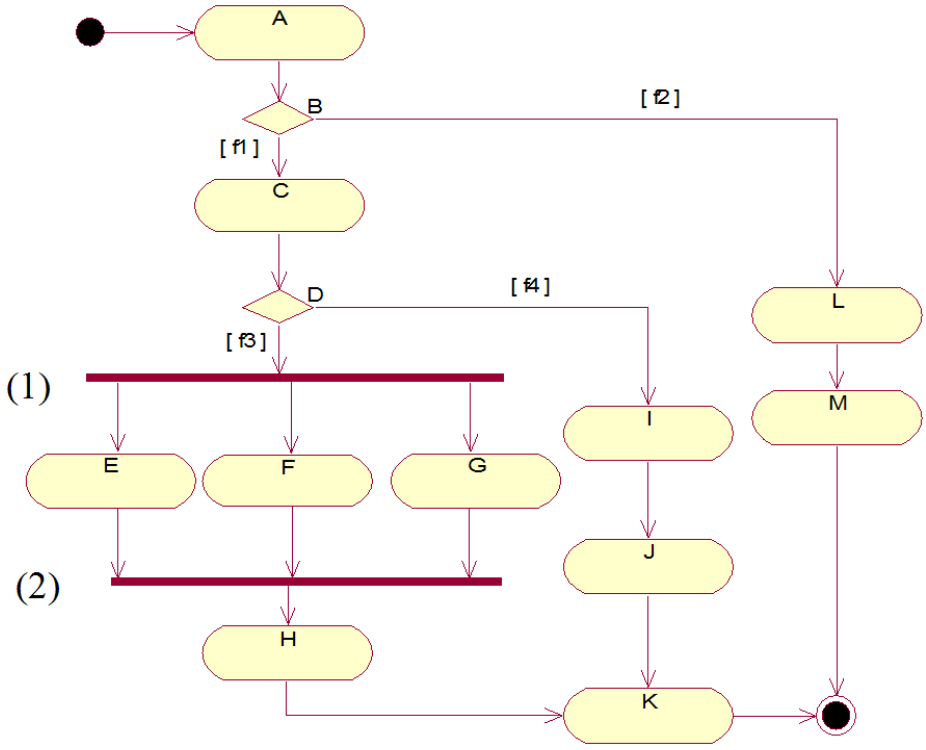
Az *Ital elfogyasztása* tevékenység két bemenő nyíllal rendelkezik, ami azt jelenti, hogy bármelyik nyíl esetén végre lesz hajtva. Ez tehát „VAGY” logikai kapcsolatot fejez ki ebben az esetben. Ugyanilyen értelemben a szinkronizációs vonal pedig „ÉS” kapcsolatot fejez ki. A kime-

nő esemény csak akkor történhet meg, ha mindegyik bemenő esemény megtörtént már.

Implementációs megközelítésben a 8.2. ábra egy metódust ír le, mégpedig a *Person (Személy)* típusra vonatkozóan. Az aktivitási diagramok alkalmasak arra, hogy tetszőleges bonyolultságú metódusokat írjunk le velük. Ugyanakkor jól lehet őket használni use case-ek tevékenységi folyamatainak megjelenítésére is, amivel a következő alpontban foglalkozunk.

8.2. Felbontási hierarchia

Az aktivitási diagramok több lehetséges működési felbontást képviselhetnek, amelyek hierarchikus kapcsolatban vannak egymással. A felbontások fentről lefelé haladva fejtendők ki, az általános működési folyamat (a legfelsőbb szint) leírásától kezdve, és egészen a konkrét megvalósításig (legalsó szint) jutva el.



8.3. ábra. Egy általános aktivitási diagram

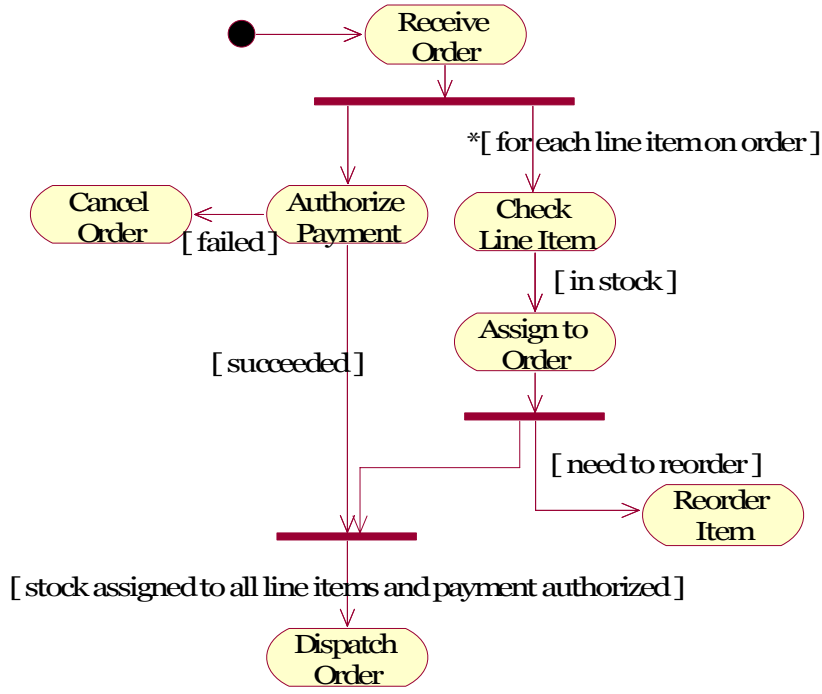
Egy általános aktivitási diagramot tüntet fel a 8.3. ábra, amelyen az egyes döntéseket és tevékenységeket betűkkel különböztettük meg. A hierarchiát követő felfogásban a 8.3. ábra tevékenységei például lehetnek bizonyos banki tranzakciók (betétszámla nyitása, pénz betétele/kivétele, hitel folyósítása stb.), míg finomabb felbontásban mindegyik tevékenység például egy Java-program műveletének felelhet meg.

Az ábrán látható (1) szinkronizációs vonaltól három párhuzamos folyamat indul el, míg a (2) vonal a folyamatok szinkronizált befejezését eredményezi.

8.3. Aktivitási diagramok use case-ekhez

Példaként az *Order* (megrendelés) feldolgozására vonatkozó aktivitási diagramot vesszük. Ez egy áruházra vonatkozik, ahol a megrendelési folyamat a következő lépésekből áll:

Megrendelési lista átvétele, – raktárkészlet ellenőrzése, – szükség esetén raktári utánrendelés, – fizetési feltételek ellenőrzése, – megrendelés (leszállítás) teljesítése vagy elutasítása.

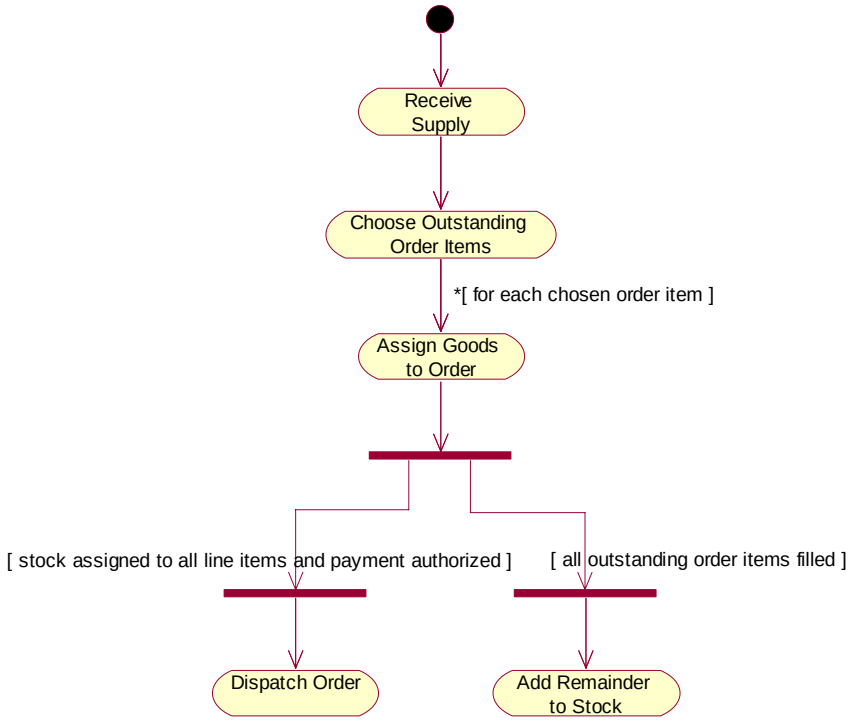


8.4. ábra. Egy megrendelés fogadása

A diagram a 8.4. ábrán látható. Ezen a * jel a megszokott többszörözést jelképezi. Annyiszor értendő a vele megjelölt tétel, ahányszor a helyzet azt megkívánja. Esetünkben ez a megrendelt árutételek darabszáma. Itt azonnal látható, hogy a legfelső szinkronizációs vonalból ezáltal annyi párhuzamos tevékenység jön ki, amennyi különböző fajta árutételt tartalmaz a megrendelés.

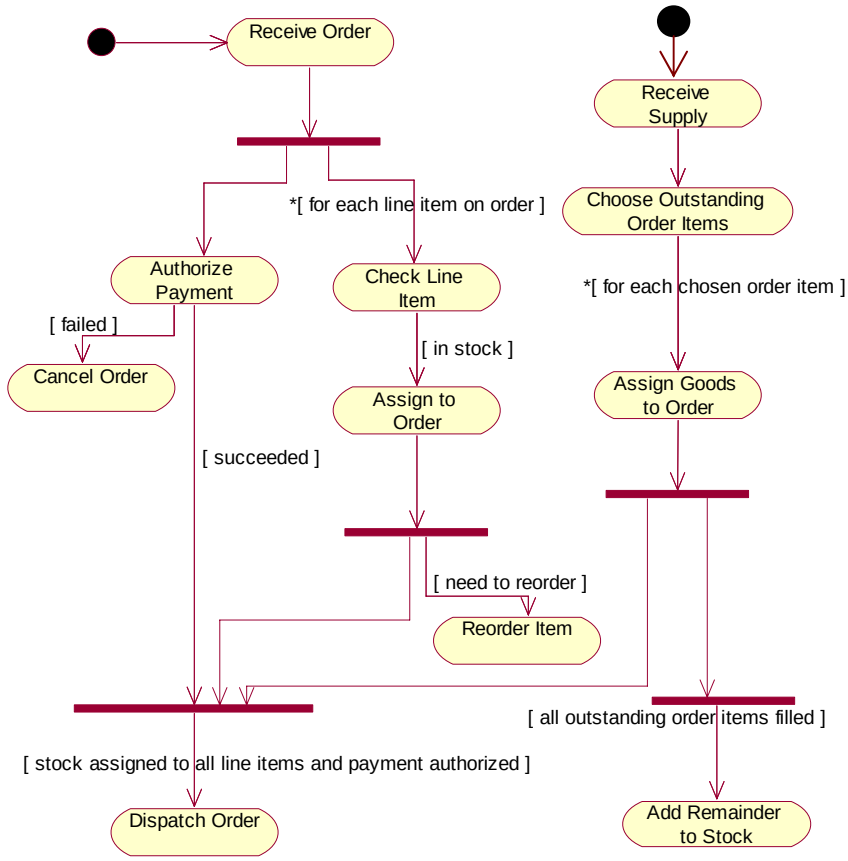
A többszörösen bejövő esemény többszörösen is érvényesülhet egy szinkronizációs vonalnál. A példában a *Dispatch Order* (megrendelés teljesítése) tevékenység előtti vonalon minden olyan alkalommal ellenőrződik a kimenő esemény teljesíthetősége, amikor arra bemenő esemény érkezik. A kimeneti teljesítés itt akkor történik meg, ha már mindegyik megrendelt árutételt a kért példányszámban egyszerre le tudja szállítani az áruház. Az aktivitási diagramon a zárójeles magyarázatok teszik ezt egyértelművé.

Egy aktivitási diagramnak akkor van csak határozott végpontja, ha az elindított események lefutása után ott tud véget érni a folyamat. A 8.4. ábrán nem szerepel ilyen végpont.



8.5. ábra. A beérkező árupótlás fogadása

A diagramon ugyanakkor van egy zsákutca is: ez a *Reorder Item* (árutétel újrarendelése) tevékenység. Ennek végrehajtása után ugyanis semmi több nem történik. Egy másik hiányosság is felfedezhető: Mi legyen akkor, ha a *Check Line Item* (árutétel ellenőrzése) tevékenységnél nem találjuk a raktáron a keresett tételt? Akkor ez a szál is véget ér itt. Ennek viszont az lenne a következménye, hogy a hiányzó tételt nem tudják elküldeni a megrendelőnek. Ahhoz, hogy a megrendelést teljesíteni lehessen, arra van szükség, hogy a raktárkészletet előbb feltöltsék a hiányzó áruval. Ez egy külön use case beiktatását teszi szükségessé, aminek a tevékenységei eddig nem voltak definiálva.



8.6. ábra. A megrendelés és az árupótlás fogadása

Az új use case ellenőrzi a feltöltött készletet a nem teljesített megrendelések szempontjából, és intézkedik a lehetséges teljesítésekről. Az ennek megfelelő aktivitási diagram a 8.5. ábrán látható. Ez azt mutatja be, hogy a megrendelés addig várakozik, amíg az összes hiányzó tétel be nem érkezett a raktárba.

A 8.6. ábrán az előző két diagram egyesítése szerepel. Ezen jól követhetjük, hogy miként befolyásolják az egyik use case tevékenységei a másikéit. Az is látható, hogy nem csak egy kezdőpont van, ami kellőképpen tükrözi az üzleti világra jellemző helyzetet, amikor is több külső eseményre kell reagálni.

Általánosan is megállapítható, hogy igen hasznos dolog az egymással kapcsolatban álló use case-ekhez úgy szerkeszteni aktivitási diagramot, hogy az a use case-ek egyesített működését jelenítse meg.

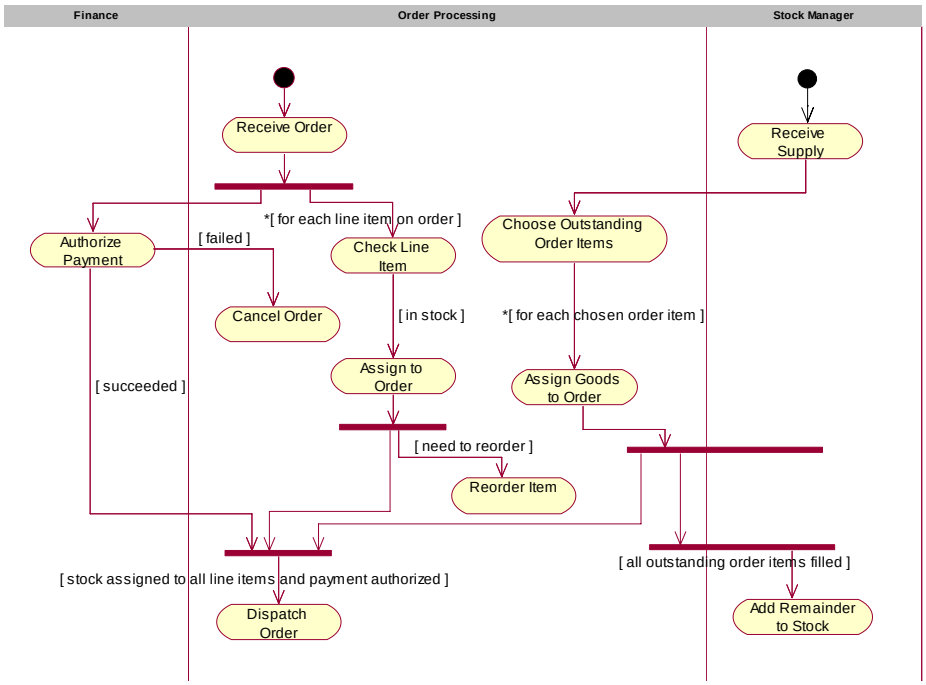
8.4. Úszópályák

Az aktivitási diagramok bemutatják az egymást követő történéseket, de azt nem fejezik ki, hogy a történések kitől erednek. Nem tükrözik, hogy melyik részleg vagy melyik személy felelős az egyes tevékenységekért. Programozási szempontból ez úgy értelmezhető, hogy a diagram nem hordoz információt arra nézve, hogy a tevékenységek melyik osztályhoz tartoznak. Meg lehetne tenni azt, hogy mindegyik tevékenység mellé odaírnánk, hogy melyik osztályhoz vagy személyhez rendelhető, de ez nem nyújtana annyi információt, mint az *interakciós* diagramok Ezek, mint tudjuk, az objektumok közötti kommunikációt mutatják be (ld. az 5. fejezetet).

Ezen az információhiányon próbál segíteni az ún. *úszópályás (swimlane)* elrendezés, mégpedig a következő módon:

Az aktivitási diagramot függőleges zónákba rendezzük, ahol az egyes zónák (ezek az úszópályák) szaggatott vonallal vannak elválasztva egymástól. Mindegyik zóna egy meghatározott programozási osztályhoz van rendelve, vagy mint a 8.7. ábrán levő példában, egy meghatározott áruházi részleghez.

Az úszópályák arra jók, hogy kombinálják az aktivitási diagramok logikai információját az interakciós diagramok hozzárendelési, felelősségi információjával. Túl bonyolult diagramok esetében azonban már nehézséget okozhat a zónákra osztás.



8.7. ábra: Úszópályás aktivitási diagram

Az aktivitási diagramokat többféleképpen szokták használni a tervezők. Van, aki az objektumokból indul ki, és később rendeli hozzájuk a tevékenységeket. Van, aki először megrajzolja az aktivitási diagramot, ezáltal áttekintő képet alkot a működésről, majd ezt követően rendeli a tevékenységeket az objektumokhoz. Általános recept itt nem adható, mind a két megközelítés hasznosnak bizonyulhat. Az a fő, hogy a tervezés eredményeként meglegyen a tevékenységek és az osztályok (objektumok) egymáshoz párosítása.

A feladatok természetéből adódóan az üzleti modellezésben célszerű először a tevékenységeket megtervezni, és csak később megadni azt, hogy az egyes tevékenységeket melyik objektum lássa el.

8.5. Egy tevékenység lebontása

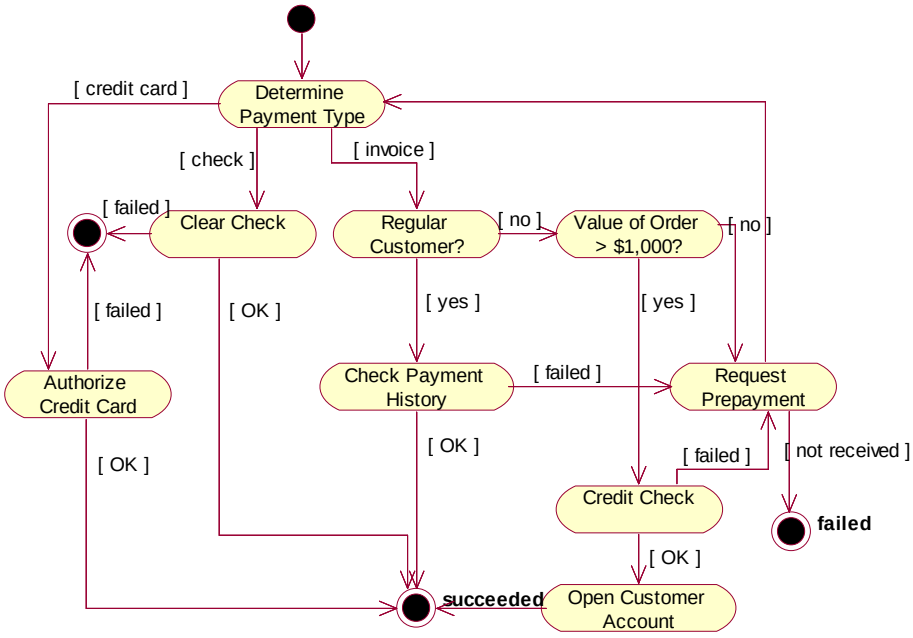
Sokszor érdemes lehet egy adott tevékenységet tovább részletezni, altevékenységekre bontani. Ez egy részletesebb aktivitási diagramot fog eredményezni. Egy ilyen lebontást mutat be a 8.8. ábra, ami a megrendelések fizetési folyamatára vonatkozik. A lebontott diagramnak mindig csak

egyetlen kezdőpontja van. Végpontja, a szükségtől függően tetszőleges számú lehet. Az aktuális példában két végpont van: sikerült a hitelkártya-lehívás, vagy nem sikerült. Ez a két eredmény lehet majd a tevékenységet megvalósító metódus lefutása után visszaküldött információ, amit a hívást leadó objektum kap meg.

A kellően részletezett tevékenységi diagram alkalmas arra is, hogy belőle már közvetlenül lehessen a programkódot írni. (Ekkor már a folyamatábra elkészítése is szükségtelenné válhat.) Ha egy folyamat logikai összefüggései túlságosan bonyolultak lennének, akkor érdemes lehet igazságtáblázatokat is igénybe venni.

8.6. Használati célok

Az aktivitási diagramok nagy előnye az, hogy elősegítik a párhuzamos működés tervezését. Nagyon jó eszköznek bizonyulnak arra, hogy munkafolyamatokat modellezzünk velük, másrészt pedig arra, hogy a többszálú programozást segítsék elő. Ugyanakkor komoly hátrányuk az, hogy nem lehet velük egyértelmű és világos kapcsolatot teremteni a tevékenységek és objektumok között. Emiatt sok fejlesztő nem is él használatukkal.



8.8. ábra. Lebontott aktivitási diagram

Mindet összevéve, a gyakorlati tapasztalatok alapján azonban biztosan kijelenthető, hogy a következő esetekben bizonyulhatnak hasznos fejlesztési segédeszköznek:

- Egy use case elemzésénél. Ilyenkor nem kell gondot fordítani arra, hogy a tevékenységeket objektumokhoz rendeljük, hanem e helyett arra érdemes koncentrálnunk, hogy feltérképezzük, milyen tevékenységekre van szükség, és ezek milyen összefüggésben vannak egymással. Az objektumokat és a bennük levő metódusokat elég lesz később megkonstruálni, amihez elsősorban interakciós diagramokat használjunk fel.
- Több use case-en áthaladó munkafolyamatok vizsgálatánál. Ez különösen hasznos lehet akkor, amikor a use case-ek kölcsönhatásban vannak egymással.

Amikor nem érdemes aktivitási diagramot használni:

- Ha azt akarjuk látni, hogyan működnek együtt az objektumok. Erre az interakciós diagramok a legjobbak.
- Ha azt akarjuk látni, hogyan viselkedik egy objektum a teljes működési idejében. Ilyen célra legalkalmasabb az állapotdiagram használata.

9. Komponensdiagramok és telepítési diagramok

Az UML diagramok közül a fizikai megvalósítás leírására jellegzetesen két diagramot használnak. A tervezett szoftverrendszer struktúráját, a programkomponensek, állományok kapcsolatát a *komponensdiagram (component diagram)* írja le, a rendszer fizikai felépítését a *telepítési diagram (deployment diagram)* rögzíti.

A fizikai megvalósítás modellezésekor a két diagram közül a telepítési diagramnak a használata elterjedtebb, mivel a diagramon nemcsak a rendszer hardver architektúráját írhatjuk le, hanem az egyes hardver elemeken ábrázolhatjuk a rendszer egyes komponenseit is. A telepítési diagram önmagában egy, a rendszer architekturális felépítését összefoglaló technika, ami az egyes szoftverkomponensek által igényelt hardvererőforrásokat írja le.

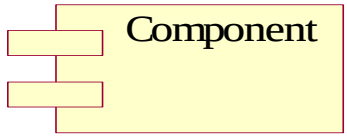
A rendszer architektúrájának modellezésére elsőként a komponensdiagramot mutatjuk be, majd ezután a telepítési diagram szerepét ismertetjük.

9.1. A komponensdiagram

A *komponensdiagram (component diagram)* a komponensekből felépülő szoftverrendszer struktúráját vázolja fel. Segítségével a rendszer szoftver elemeit rendszerezhetjük, csoportosíthatjuk, valamint az egyes komponenseket egymáshoz rendelhetjük, egymásba leképezhetjük. A diagram jól modellezi a szoftverkomponensek egymáshoz való viszonyát, kommunikációját.

A komponensdiagram fő építőeleme a *komponens (component)*. A komponens adott modellelemek (osztályok, csomagok) fizikai egysége. Egy fizikailag bonthatatlan szoftver egység, amit állomány, például forráskód, szerkesztendő vagy futtatható szoftver elem: lefordított tárgykód, bájtkód, futtatható program vagy dinamikus könyvtár modellezésére lehet használni. A fejlesztés menete szerint a tervezési munkaszakaszban meghatározott tervezési osztályokat, alrendszereket implementáljuk, ami nagyobb részben a forráskódok előállítását jelenti. Egy komponensben számos implementációs osztály valósulhat meg.

Az UML a komponens jelölésére egy téglalap szimbólumot használ, amelynek bal oldalát két kisebb téglalap metszi. A komponens szimbólumát a 9.1. ábra illusztrálja.



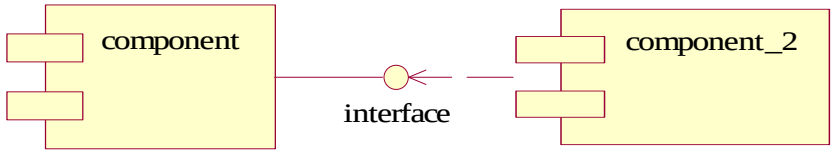
9.1. ábra. A komponens UML szimbóluma

A komponens diagramban lehetőség van a komponens típusát külön jelölni. A különböző komponens típusok megjelenítésére az UML-ben a következő sztereotípusok állnak rendelkezésre:

- végrehajtható program : <<executable>> sztereotípus,
- forráskódot vagy adatot tartalmazó állomány: <<file>> sztereotípus,
- statikus vagy dinamikus programkönyvtár: <<library>> sztereotípus,
- adatbázistábla: <<table>> sztereotípus,
- alapvetően szöveget tartalmazó dokumentum: <<document>> sztereotípus.

A komponens diagramban a komponensek által használt vagy felkínált metódusok egy csoportját az *interfészek (interface)* jelölik (lásd 9.2. ábra). A modellben az interfészeket üres körök szimbolizálják.

A szoftverkomponensek közötti kapcsolatot, kommunikációt a függőségi viszonnyal írjuk le. A függőségi viszonnyal jól tudjuk modellezni, hogy mely komponens melyiknek szolgáltat valamilyen információt, eljárást stb. A modellben a komponensek közötti függőségi kapcsolatot szaggatott nyíllal prezentáljuk. Komponensek közötti kommunikációt a 9.2. ábra komponens diagramja szemlélteti.



9.2. ábra. Komponensdiagram interfésszel

A komponens szintű modell-terv a tervezési munkaszakasz végére áll elő. A tervezés leírásai szerint meghatározott szoftverkomponenseket az implementáció munkaszakaszban fejlesztjük ki. Ahogy már a bevezetőben is elhangzott, a komponens diagram sok esetben a telepítési diagrammal együtt, azzal összehangolva készül. A gyakorlatban több olyan fejlesztési

projekt zajlik, ahol a rendszer hardver architektúrája eleve adott, rögzített a megbízó aktuális infrastrukturális környezete miatt. A rendelkezésre álló hardver infrastruktúra eredményeképpen ilyenkor a komponens modellterv elkészítése a követelményspecifikáció és elemzési munkaszakaszokra tolódhat előre.

9.2. A telepítési diagram összetétele

A *telepítési diagram (deployment diagram)* egy teljes informatikai rendszer szoftver és hardver komponenseit mutatja be, a köztük levő információs kapcsolatok, összefüggések feltüntetésével.

A diagram minden egyes *csomópontja (node)* valamilyen számítási feldolgozási egységet képvisel, egy szoftver vagy egy hardver elemet. A hardver lehet egy kisebb egység, de lehet egy teljes számítógép is.

A 9.3. ábra egy olyan kórházi rendszer hardver-felépítését mutatja be, amely májfunkciók, ill. cukorbetegség vizsgálatára szolgál. A rendszer egy személyi számítógépet (PC-t) és két szervert tartalmaz.

A kórházi rendszer telepítési diagramja a 9.4. ábrán látható. Ezen a diagramon a személyi számítógép egy *UNIX* szerverhez van kötve, a *TCP/IP* hálózati protokoll (kommunikációs interfész) felhasználásával. A csomópontok közötti összeköttetések az információcsere, kommunikáció útvonalait ábrázolják.

A szoftver komponensek azokon a hardver egységeken vannak feltüntetve, ahol végrehajtásra kerülnek, ahol a kódjuk lefut. Fowlernél ezek a szoftver komponensek pontosan megfelelnek a csomagdiagramon ábrázolt csomagoknak. Vagyis: egy komponens az egy csomag.

A komponensek közötti függőség ugyanaz kell legyen, mint a csomagok között meglevő függőség. Ezek a függőségek itt azt mutatják, hogy az egyes komponensek hogyan kommunikálnak egymással. Tehát: itt a függőség egybeesik a kommunikációval.

A példában a *Liver Unit UI* (a májfunkció egység felhasználói interfésze) függőségben van a *Liver Unit Client Facade*-dal (májfunkció egység beteg felé mutatott felülete), mivel az metódusokat hív meg a *Facade*-ban. Bár a kommunikáció itt kétirányú, abban az értelemben, hogy a *Facade* adatokat küld vissza, a *Facade* nincs tudatában annak, hogy kitől kapta a hívást, s így nem áll függőségben a *Liver Unit UI*-jal.

Ha viszont a két helyen előforduló *Health Care Domain* nevű (egészségügyi ellátás domén) nevű komponenst tekintjük, mind a kettő tudja, hogy egy másik *Health Care Domain* komponenst szólít meg. Ezért a köz-

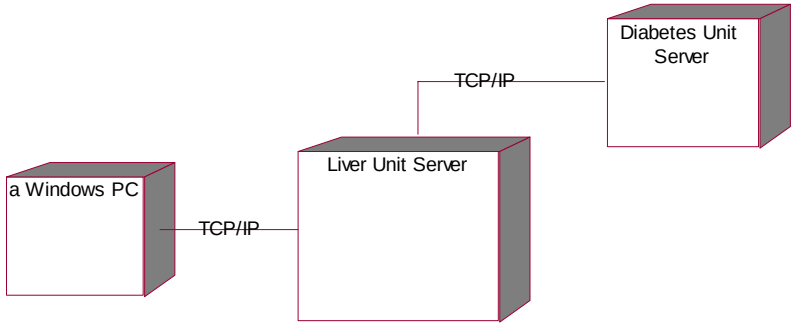
tük levő kommunikáció kétirányú lesz, s így a kommunikációs függőség is kétirányú.

Az interfészeket kis karikák reprezentálják. Egy komponensnek egynél több interfésze is lehet, amiből látható, hogy melyik komponens melyik interfészén keresztül kommunikál. A 9.4. ábrán a *Windows PC*-nek két komponense van: az *UI* és a *Facade*. A *Facade* program a szerveren futó alkalmazást annak interfészén keresztül tudja elérni. A szerveren fut még egy elkülönített konfigurációs komponens, amely két objektumot tartalmaz. A szerver alkalmazási programja kommunikál a saját *Health Care Domain* komponensével, ahol ez a komponens még kommunikálhat több másik *Health Care Domain* komponenssel a hálózatban. (A példában csak egy másik ilyen szerepel.)

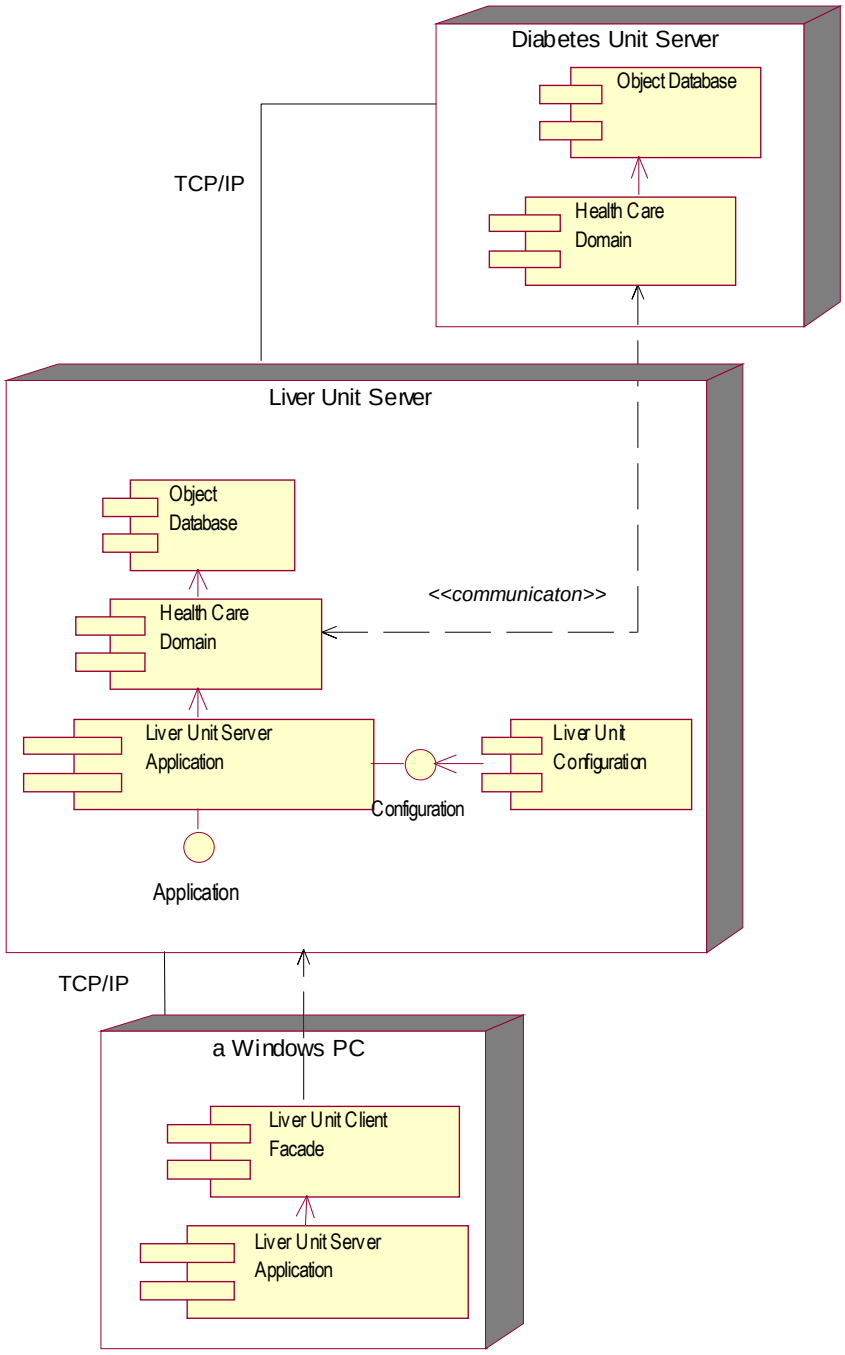
A *Health Care Domain* komponensek használata rejtve van a szerveren futó alkalmazás előtt. Mindegyik *Health Care Domain* komponens egy lokális adatbázissal rendelkezik, az *Object Database*-zel (objektum-adatbázis).

9.3. Felhasználási célok

A gyakorlatban az ilyen diagramokat viszonylag keveset használják. Legalábbis a szabványos formájában nem, csak egyéni skiccek formájában. Várható azonban, hogy nőni fog a telepítési diagramok jelentősége és felhasználási köre, az ún. *elosztott hálózati rendszerek* terjedésével. Ezek a rendszerek egyre bonyolultabb hardver-szoftver kommunikációs megoldásokat hordoznak, amihez a telepítési diagramok jó áttekintést tudnak nyújtani.



9.3. ábra. Egy kórházi rendszer hardver elemei



9.4. ábra. A kórházi rendszer telepítési diagramja

10. Egy UML-modell összefüggősége és teljessége

10.1. A modellezés általános problémái

Az UML-diagramok kifejezőképessége és egyszerű használata azt eredményezte, hogy viszonylag könnyen lehet szoftvermodelleket felépíteni ezekkel az eszközökkel. Ugyanakkor azonban mindez azzal is jár, hogy egy így előállított modell nem lesz összefüggő (konzisztens), vagy nem lesz teljes, vagy más egyéb hibákat is tartalmazhat.

A gyakorlatban megépített modellek néhány tipikus hibáját az alábbiakban soroljuk fel:

1. Az öröklődési kapcsolatokban ciklusok léteznek, vagyis körkörös öröklődés alakul ki.
2. Bizonyos attribútum egyaránt definiálva van az osztályban és az ősosztályban is.
3. Egy művelet (metódus) kétszer van definiálva ugyanabban az osztályban, ugyanazokkal a bemeneti paramétertípusokkal, viszont különböző kimeneti eredménytípusokkal.
4. Egy interfészművelet nincs implementálva egy konkrét alosztályban.
5. Egy művelet (metódus) korlátozottabb láthatósággal van definiálva egy leszármaztatott osztályban, mint egy ősosztályban.
6. Egy interfész úgy van definiálva, mint egy osztály alosztálya.
7. Az állapotdiagram egy állapota nem érhető el, mert csak kivezető átmenetei vannak, amikor az nem kezdőállapot.
8. Egy állapotcsoport elemei zárt ciklust alkotnak, miután nincs belőlük kiágazás. Ezáltal ha valahonnan belépés történik a ciklusba, onnan már nem lehet egy külső állapotot elérni.
9. Egy osztály kardinalitása p-nek van definiálva, míg a hozzá tartozó alosztály kardinalitása q-nak, ahol $q > p$.

Az ilyen jellegű hibák kijavítását minél előbb kell megtenni, annak érdekében, hogy minél kevesebb fejlesztési veszteség keletkezzék az érvénytelen, téves modellek következtében. A hibák feltárása érdekében arra van szükség, hogy a tervezők menet közben rendszeresen átvizsgálják az előállított modellegységeket, külön-külön is önmagukban, másrészt pedig azok egymáshoz való viszonyát és kölcsönhatását is ellenőrizzék.

A minőség javítása és a modell szükséges átalakítása érdekében többek között az alábbiakról érdemes meggyőződni:

1. Hogy az asszociációknál korrekt számosságot (multiplicitást) alkalmaztunk. Előfordulhat például, hogy a megadott felső érték kisebb, mint amennyire szükség lenne egyes szituációkban.
2. Hogy az attribútumok és az asszociációk végénél korrekt minősítő előírást *(qualifier)* alkalmaztunk.
3. Hogy az összes olyan esetet felismertük, amelyben egy asszociáció végét sorrendezni kellett.
4. Hogy az összes öröklődési esetet számba vettük és leírtuk a modellben.
5. Hogy korrekt és informatív típusokat rendeltünk mindegyik attribútumhoz. Például egy *x* koordináta típusát a tartományával jelöljük ki, vagyis mondjuk *1..15-nek* adjuk meg, ahelyett hogy egyszerűen csak *Integernek* deklarálnánk.
6. Hogy megfelelő állapotokat definiáltunk, amelyek pontosan tükrözik a működési módokat és azokat a fázisokat, amelyekbe a rendszer belekerülhet. Bármely két különböző állapot esetében a rendszernek eltérő viselkedést kell mutatnia, máskülönben a két állapot valószínűleg összevonható lesz.

10.2. Az összefüggőség (konzisztencia)

Egy UML-modell *összefüggő (konzisztens),* ha létezik hozzá egy bizonyos lehetséges működőképes implementáció. Egy inkonzisztens specifikáció azzal jár, hogy nem lehet belőle olyan szoftvert fejleszteni, ami működőképes lenne. Ha például egy **C** osztályban úgy definiálnánk három egészszámú attribútumot (x, y és z) a hozzájuk tartozó megkötésekkel, hogy

$x < y$,
$0 \leq y < z$,
$x = y \cdot z + 1$,

akkor nem létezhetne lehetséges megvalósítás a C-re.

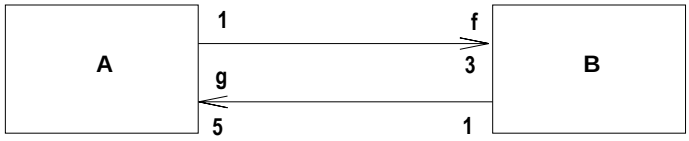
Egy másik már „kifinomultabb” hibát a 10.1. ábrán látható modellben mutatunk be. Az **a.f** halmazok mindegyike 3-as méretű, és **B**-nek egy partícióját alkotja. Az indexelésük **A** elemei által történik, vagyis B mindegyik eleme pontosan egy a.f-ben van benne. Ezért

$$B.size = 3 * (A.size),$$

és hasonlóképpen

$$A.size = 5 * (B.size).$$

A két összefüggés csak akkor teljesülhet egyszerre, ha A is és B is üresek, vagyis egyik osztálynak sem létezik példánya.



10.1. ábra. Egy üres UML-modell

Az ilyen jellegű konzisztenciaproblémák úgy kerülhetők el, ha megköveteljük, hogy a specifikáció tartalmazzon konkrét beállítást az attribútumok és az asszociációk mindegyik kezdőértékére. A másik lehetőség az, hogy jól definiált és dokumentált default-értéket használjunk akkor, ha nem állítunk be konkrét kezdőértéket. A fejlesztőnek ilyenkor mindig ellenőriznie kell, hogy ezek a kezdőértékek kielégítik a modell összes megkötését.

Az első fenti példában az egész változók 0-val való default-inicializálása nem fogja kielégíteni a megkötéseket, de más egyéb beállítás sem elégítené ki, ezért a modell elvetendő vagy módosítandó lesz.

A második példában az A mindegyik új eleméhez három létező B-elemre van szükség ahhoz, hogy az f-et inicializáljuk. Viszont hogy ezek a B-elemek létezzenek, az szükséges, hogy már 15 létező A-elemünk legyen. Ezért lehetetlen az első A-objektumot létrehozni.

Az előbbi példákban adatok közötti inkonzisztencia állt fenn. Ugyanakkor lehetséges az is, hogy egy osztály művelete és adata között lép fel inkonzisztencia. Egy C osztály **op** művelete inkonzisztens, ha az **op** úgy tudja módosítani a C egy adatát, hogy megsérti az adat deklarált típusát, vagy pedig megszegi a C valamelyik megkötését, ami adatra vonatkozik.

Például, ha a C osztály az

```
x : 1..15
```

formában deklarált attribútummal rendelkezik, akkor az

```
incx ( )
post : x = x@pre + 1
```

művelet megsértheti az x típuskorlátozását.

Ha egy műveletet állapotdiagrammal írunk le, akkor mindegyik állapotátmenetnél arra kell ügyelnünk, hogy az csak akkor következhessék be, ha semmilyen megkötést nem sért meg.

Az állapotdiagramokban is el kell kerülni az inkonzisztens akciókat vagy állapotváltozásokat. Ha például két átmenet is bekövetkezhet egy esemény hatására, akkor ezek következményei és célállapotai egymással konzisztensek kell hogy legyenek.

10.3. A teljesség

Egy UML tervezési modellt *teljesnek* nevezünk, ha az általa leírt szoftverrendszer állapotai és viselkedési módjai mindegyik lehetséges működési esetre és feltételre specifikálva lettek. Egy osztály esetében ez a definíció a következő szabályban fogalmazható meg:

A teljesség hiánya és az inkonzisztencia egymással kapcsolatos problémák. Egy művelet inkonzisztens lehet, ha beállít egy *att1* attribútumértéket, de nem állítja be az *att2* attribútumot akkor, amikor ezekre egy megkötés vonatkozik, ami összekapcsolja a két értéket.

Ezen az alapon bizonyos teljességi hiányt fel lehet fedni azáltal, hogy a konzisztenciát ellenőrizzük. Egy másik teljességhiány magukban a megkötésekben is felléphet. Például, ha egy osztálynak van két integer attribútuma, *att1* és *att2,* egy Boole-attribútuma, *att3,* továbbá egy olyan egyedül álló megkötése, hogy

$$att1 > att2 \Rightarrow att3 = true,$$

akkor felmerül a kérdés, hogy mi lesz az att3 értéke abban az esetben, ha

$$att1 \leq att2.$$

A specifikáció teljességére vonatkozó hasznos ellenőrzés az, ha minden egyes $\mathbf{A} \Rightarrow \mathbf{B}$ típusú megkötést tesztelünk abból a szempontból, hogy specifikálva vannak-e azok az esetek, amikor az **A** logikai értéke „false” lesz. Ha ez hiányzik valahol, akkor azt kell ellenőrizni, hogy valóban szükséges-e a specifikálás.

A teljesség hiányára utal az is, ha egy attribútum nem fordul elő egyik megkötésben sem.

A teljesség érvényesülése megköveteli, hogy az állapotdiagram egy adott állapotát tekintve, az abból kiinduló átmenetek mindegyikére teljesüljön a következő kritérium: egy átmenethez tartozó logikai feltételek

együttes VAGY kapcsolata „true” értéket adjon. Ez a kritérium ugyanis kizárja annak lehetőségét, hogy az átmenetet előidéző esemény hatására nem következik be átmenet az új állapotba.

Mindezekhez kiegészítésül még két szabályt adunk meg:

1. Egy osztály művelete (metódusa) teljes, ha annak hatása definiálva van minden egyes lehetséges állapotra, amely kielégíti az osztály összes működési feltételeit.
2. Egy állapotdiagram teljes, ha létezik legalább egy kezdőállapota és legalább egy végállapota, továbbá a működtetés során mindegyik állapot elérhető benne.

10.4. Megállapítások

A konzisztencia és teljesség szempontjából a következő megállapítások tehetők:

- Jelenleg nem áll rendelkezésre olyan módszer, ill. a módszert megvalósító számítógépes eszköz, ami egy szoftverterv UML-diagramjai közötti összhang bizonyítására, ill. cáfolására lenne alkalmas.
- Ugyanígy nincs megoldva még a teljesség bizonyításának vagy cáfolásának feladata sem.
- A szoftvertechnológia jelenlegi fejlettségi állapotában az UML nyelv és a hozzá kapcsolódó diagramok arra szolgálnak, hogy egy szoftvertervet több lehetséges és fontos aspektusból tudjunk megjeleníteni és elemezni, azzal a céllal, hogy minél nagyobb biztonsággal lehessen a szoftvert megvalósítani.
- A szoftvertervhez az UML-leírások kézi, emberi úton történő előállítására van szükség. Ezeknek a leírásoknak az összhangját és teljességét szintén emberi úton történő összevetés és elemzés tudja csak megteremteni.

Az automatizált konzisztencia-ellenőrzés és teljességellenőrzés még számos kutatási és fejlesztési feladat megoldását követeli meg. Ugyanakkor azonban nem várható, hogy átfogó, abszolút és egzakt megoldásokat lehessen kidolgozni ezen a területen. Ez azt jelenti, hogy a tervezésben az emberi közreműködés mindig elkerülhetetlen lesz. A számítógépes szerepvállalás csupán a munkaterhek csökkentésében és a tervek megbízhatóságának fokozásában fog érvényesülni.

Felhasznált irodalom

Szoftvertechnológia témakörrel foglalkozó könyvek

1. Roger S. Pressman: *Software Engineering,* Fifth Edition, McGraw-Hill Book Company, USA, 2001.
2. Ian Sommerville: *Szoftverrendszerek fejlesztése,* Software Engineering, (Sixth Edition), PANEM Könyvkiadó, Budapest, 2002.
3. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli: *Fundamentals of Software Engineering,* Second Edition, Prentice Hall, Pearson Education, Inc., USA, 2003.
4. Ian Sommerville: *Software Engineering,* Seventh Edition, Pearson Education, Inc., USA, 2004.
5. Raffai Mária: *Információ-rendszerek fejlesztése és menedzselése,* Novadat Kiadó, Győr, 2003.

Objektumorientált fejlesztéssel foglalkozó könyvek

6. Kondorosi Károly, László Zoltán, Szirmay-Kalos László: *Objektumorientált szoftverfejlesztés,* Computer Books Kiadói Kft, Budapest, 1997.
7. Richard C. Lee, William M. Tepfenhart: *Practical Object-Oriented Development with UML and Java,* Prentice-Hall, Inc., USA, 2003.
8. Hans-Erik Eriksson, Magnus Penker: *Business Modeling with UML,* John Wiley & Sons, Inc., New York, 2000.
9. Martin Fowler: Analysis Patterns: *Reusable Object Models,* Addison-Wesley, USA, 1996.
10. Martin Fowler, Kendall Scott: *UML Distilled: Applying the Standard Object Modeling Language,* Addison-Wesley-Longman, Inc., USA, 1997.
11. Ivar Jacobson, Grady Booch, James Rumbaugh: *Unified Software Development Process,* Addision-Wesley, USA, 1999.
12. Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language Users Guide,* Addision-Wesley, USA, 1999.
13. Mark Priestley: *Practical Object-Oriented Design with UML,* McGraw-Hill Publishing Company, Great Britain, 2000.
14. Shel Siegel: *Object-Oriented Software Testing,* John Wiley & Sons, Inc., New York, 1996.
15. Balázs Benyó, József Sziray: *Testing Principles for Object-Oriented Software,* Lecture Notes, Széchenyi College, Győr, 2001.

16. Else Lervik, Vegard B. Havdal: *Java the UML Way, Integrating Object-Oriented Design and Programming,* John Wiley & Sons, Ltd., Chichester, England, 2002.
17. Paul C. Jorgensen: *Software Testing,* A Craftsman's Approach, Second Edition, CRC Press LLC, USA, 2002.
18. Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice,* Second Edition, Pearson Education, Inc., USA, 2003.
19. Stephen R. Schach: *Object-Oriented and Classical Software Engineering,* Sixth Edition, McGraw-Hill Companies, Inc., USA, 2005.
20. Craig Larman: *Applying UML and patterns,* Second Edition, Prentice-Hall, Inc., USA, 2002.
21. Kevin Lano: *Advanced Systems Design with Java, UML and MDA,* Elsevier Butterworth-Heinemann, United Kingdom, 2005.