Sike Sándor, Varga László

Szoftvertechnológia és UML

Második, bővített kiadás

Tartalomjegyzék

1. Bevezetés **9**

2. A szoftvertechnológia kialakulása **13**

2.1. Projektmenedzselés 17
2.2. Egyszerű programfejlesztési modell 25
2.3. Programfejlesztés specifikációval 26
2.4. Nagy rendszerek általános fejlesztési modellje 28
2.4.1. Vízesés modell 30
2.4.2. Evolúciós modell 31
2.4.3. Boehm-féle spirális modell 33
2.4.4. A probléma megoldásának előzménye 39
2.4.5. Követelmények leírása 39
2.4.6. Követelmények elemzése és prototípus 45
2.4.7. Programspecifikáció 48
2.4.8. Tervezés 49
2.4.9. Implementáció 52
2.4.10. Verifikáció, validáció 54
2.4.11. Rendszerkövetés és karbantartás (maintenance) 56
2.4.12. Dokumentáció 59
2.5. Programfejlesztés modellalkotással 62
2.6. A programozási technológia fejlődési iránya 65
2.7. Unified Modeling Language (UML) 67
2.7.1. Az UML építőkövei 69

3. Az objektumelvű programozás kialakulása 73
3.1. Absztrakció ... 75
3.2. Adattípus ... 76
3.2.1. Egyszerű adattípus ... 76
3.2.2. Összetett adattípus ... 77
3.2.3. Típusosztály ... 78
3.3. Típusöröklődés ... 80
3.3.1. Típusöröklődés specializációval ... 80
3.3.2. Típusöröklődés újrafelhasználással ... 83

4. Az objektumelvű modellezés alapjai 85
4.1. Nézetrendszer ... 85
4.1.1. Az UML diagramjai ... 87
4.2. Az objektum informális definíciója ... 90
4.3. Az objektumosztály informális definíciója ... 96
4.4. Az osztálydiagram definíciója ... 103
4.5. Az objektumdiagram definíciója ... 104

5. Objektumosztályok közötti kapcsolatok 105
5.1. Társítási reláció, asszociáció ... 105
5.1.1. A multiplicitás jelölése ... 108
5.1.2. Az objektum szerepének jelölése ... 109
5.1.3. Az asszociációhoz kapcsolódó további jelölések ... 113
5.1.4. Példák asszociációra ... 118
5.2. Aggregáció ... 120
5.3. Kompozíció ... 124
5.4. Általánosítás és specializáció ... 126
5.4.1. Többszörös specializáció ... 129
5.4.2. Többszörös általánosítás ... 130
5.5. Osztálydiagramok készítése ... 134
5.5.1. Az osztálydiagramok és az objektumdiagramok kapcsolata ... 146

6. Állapotdiagram 161
6.1. Az állapot informális definíciója 165
6.2. Az esemény informális definíciója 167
6.3. Az állapotdiagram definíciója 169
6.4. Esemény és akció 170
6.5. Az állapotdiagram bonyolultsága 173
6.5.1. Állapotok általánosítása 175
6.5.2. Állapotok aggregációja 176
6.6. Az állapot általános fogalma 186

7. **Szekvenciadiagram 211**
7.1. Osztályszerep 211
7.2. Osztályszerep életvonala 212
7.3. Az osztályszerep aktivációs életvonala 212
7.3.1. Az objektum létrehozása és megsemmisítése . . 212
7.3.2. Az objektum aktivációja 213
7.4. Üzenettípusok 215
7.4.1. Egyszerű üzenet 215
7.4.2. Szinkronizációs üzenet 215
7.4.3. Időhöz kötött várakozás 216
7.4.4. Randevú üzenet 216
7.4.5. Aszinkron üzenet 217
7.4.6. Visszatérési üzenet 217
7.5. A szekvenciadiagram kiegészítései 220
7.5.1. A hierarchia ábrázolása 221
7.5.2. A választások ábrázolása 222
7.5.3. Az iteráció ábrázolása 224
7.5.4. A párhuzamosság ábrázolása 225

8. Funkcionális modell 245
8.1. Együttműködési diagram 245
8.2. Adatfolyam-diagram 249
8.3. A funkcionális modell megalkotása 252
8.4. Aktivációs diagram 254

8.5. Végrehajtási gráf 266

9. Implementációs szempont szerinti diagramok 275
9.1. Komponensdiagram 275
9.2. A rendszer statikus szerkezete 277
9.2.1. Alrendszer 278

10. Az UML további diagramjai 289
10.1. Környezeti diagram 289
10.2. Használati esetek diagramja 290

11. Modellalkotás a programfejlesztésben 295
11.1. Szemléltető példa 299

12. Tervminták, keretek 321
12.1. Tervminta 321
12.1.1. Tervminták megadása 322
12.2. Keret 327

13. A programtermék minőségi mutatói 331
13.1. Programhelyesség 332
13.2. Megbízhatóság, robusztusság 332
13.3. A program bonyolultsága 334
13.4. Teljesítményelvű szoftvertechnológia 336

Tárgymutató 345

Irodalomjegyzék 351

1. Bevezetés

Ez a könyv azoknak az előadásoknak és a hozzájuk kapcsolódó gyakorlatoknak az anyagát tartalmazza, amelyeket a szerzők az Eötvös Loránd Tudományegyetemen tartottak programozó matematikus hallgatók számára a „Programozási technológia" című tantárgy keretében. Az oktatott tárgyat a hallgatók az alapvető matematikai (Bevezető fejezetek a matematikába) és programozási (Bevezetés a programozáshoz) ismeretek elsajátítása után vehetik fel, ez általában a 3. szemeszter során történik meg.

Az anyag tárgyalása során a szerzők igyekeztek elkerülni az elemi matematikai ismereteket meghaladó formalizmusok használatát. A bevezetett fogalmak definícióit, azok tulajdonságait pontokba szedve, informális módon adják meg; és azok megértését, gyakorlatban történő felhasználását számos feladat megoldásának bemutatásával könnyítik meg az olvasó számára.

A számítógéppel ma már egyre nagyobb, egyre bonyolultabb feladatokat oldunk meg. Ezeknek a feladatoknak a megoldására nagy programrendszereket hozunk létre. Nagy programrendszerek létrehozására, legyártására gyártási technológiára van szükség. A témakör tárgyalása során először definiáljuk a szoftvertechnológiával kapcsolatos legfontosabb fogalmakat, majd röviden ismertetjük a programok fejlesztésére szolgáló leggyakoribb modelleket.

A számítógéppel történő problémamegoldást egyre több szoftvereszköz támogatja. Ezek az eszközök egységes rendszert (szoftverfejlesztési környezetet) alkotva támogatják a programozó munkáját, egyre több automatizmust nyújtva számára. Ez a magyarázata annak, hogy

a szoftvertechnológiában egyre nagyobb hangsúlyt kap a probléma megoldására szolgáló módszerek és eszközök kérdése.

Az emberi tevékenységek során kialakult az a gyakorlat, hogy a probléma megoldását először egy, a probléma számára alkalmas rendszerben konstruáljuk meg, azaz létrehozzuk a megoldás absztrakt modelljét, és csak azután kerül sor a megoldás realizálására. Például lerajzoljuk a megépítendő házat, és csak azután fogunk hozzá annak felépítéséhez. Megadunk egy matematikai modellt, például egy egyenletrendszert írunk fel megoldásként valamire, és csak azután foglalkozunk annak kiértékelésével. Ha az absztrakt modell szintaktikailag és szemantikailag egyértelmű, akkor szoftvereszközökkel a konkrét kiértékelés már automatikus úton is megtörténhet.

Egy probléma számítógéppel történő megoldásakor felmerül a kérdés, hogy melyik legyen az a rendszer, amelyben az absztrakt megoldást felírjuk? Objektumelvű megközelítés esetén erre a kérdésre azt a válasz adjuk, hogy az absztrakt modellt annak a valós világnak a terében adjuk meg, amelyben a probléma felmerült. A valós világ objektumait és a rajtuk értelmezett műveleteket használjuk fel a megoldás absztrakt változatának megkonstruálásához.

A következő kérdés az, hogy milyen formalizmust használjunk a modell leírásához? Elvileg lehetne ez a beszélt nyelv is. Ezzel azonban az a probléma, hogy nem biztos, hogy egyértelmű leírást fog eredményezni. Nagy bonyolultságú feladatok esetén az áttekinthetőséggel is probléma lehet. Ugyanakkor célszerű olyan eszközt választani, ami természetes eszköz az emberiség eddigi problémamegoldási gyakorlatában. Ilyen évszázadok során kialakult gyakorlat a megoldás rajzos ábrázolása. A számítógéppel történő problémamegoldás során is természetes eszközként használták a programozók a rajzot, amikor a megoldás adatfolyam-diagramját felrajzolták. Ezután ezt a diagramot ültették át a számítógépre, program formájában.

Az úgynevezett Unifled Modeling Language (UML) szerzőinek a gondolata az volt, hogy szedjük össze azokat a diagramokat, amelyeket a programozási technológiákban eddig is használtunk. Ezeknek hozzuk létre a jól definiált formáit, amelyeket egy egységes rendszer-

ben foglalhatunk össze. Így jött létre a diagramokon alapuló nyelv, az UML. Ennek a nyelvnek a segítségével megadhatjuk a problémák széles körének megoldását, az absztrakt megoldást a valós világ objektumainak terében. Ezt a megoldást azután többé-kevésbé automatikusan áttranszformálhatjuk szoftvereszközökkel egy objektumelvű programozást támogató programozási nyelvre. Természetesen akkor is a technológiai folyamat részét képezheti egy ilyen modell, ha a transzformációs eszközrendszer nem áll rendelkezésünkre. Ezért választottuk mi is a probléma megoldását leíró absztrakt modell nyelveként az UML-t, amelyet a szoftvertechnológia részeként tárgyalunk, a modellalkotással történő programfejlesztés keretében.

Az objektumelvű programozási módszer kialakulásának és alapfogalmainak ismertetése után kerül sor a könyv fő témájának tárgyalására. Ennek lényege a számítógéppel megoldandó probléma modelljének megkonstruálása, amelyet sok példa, feladat tárgyalásán keresztül mutatunk be. A gyakorlati tapasztalat azonban azt mutatja, hogy a megoldás modelljének - különböző nézeteknek megfelelően - más-más kérdésre kell választ adnia. Erre alakultak ki az UML keretében a különböző modellalkotási követelményeknek megfelelő eszközök, amelyek azonban egységes rendszert alkotnak. Bár külön tárgyaljuk az egyes nézetrendszereknek megfelelő problémaköröket, számos olyan feladattal is fog találkozni az olvasó a könyvben, ahol a megoldást több nézetrendszer modelljével együtt mutatjuk be.

A második kiadást kibővítettük az objektumelvű tervezés fázisainak áttekintésével és egy, az objektumelvű tervezést szemléltető példával. Ezenkívül foglalkozunk még a teljesítményelvű szoftvertechnológia alapjaival, a szükséges fogalmakkal, kiegészítésekkel.

A szoftvertechnológia különböző megközelítési módjai, módszerei után érdeklődő olvasó figyelmébe ajánljuk az [1] könyvet. Az objektumelvű szoftverkonstrukció részletes tárgyalását a [2] könyvben találhatja meg az olvasó.

Annak, akit az objektumelvű programozás mélyebb matematikai tárgyalása és vizsgálata érdekel, a [3], [4] könyvre hívjuk fel a figyelmét.

Az UML nyelvvel kezdőként megismerkedni kívánó olvasónak me-

legen ajánljuk az [5] könyvet. Az UML nyelvnek példákon keresztül történő ismertetését találhatjuk meg a [6], [8] könyvekben. Az UML nyelvi szabványának egy olvasmányos definícióját nyújtja a [7] könyv. Végül az UML nyelv különböző területeken történő alkalmazásaira hívjuk fel az olvasó figyelmét az üzleti élet területével foglalkozó [8] könyv és a menedzsment problémakör kérdéseit tárgyaló [9] könyv ajánlásával. A tervminták iránt érdeklődő olvasónak javasoljuk a [10] könyv tanulmányozását. Egy UML alapú teljesítményelvű szoftertechnológiát mutat be a [11] könyv.

2. A szoftvertechnológia kialakulása

A 2000. év küszöbén számos találgatás jelent meg arról, hogy mi lesz az új évszázad fő jellemvonása. Ezek között szinte kivétel nélkül megjelenik az a megállapítás, hogy a 21. század az informatika százada lesz. Valóban, - már napjainkban is - minden gazdaságilag fejlett országban a gazdaság működésében meghatározó jelentősége van az informatikai eszközöknek. A globalizációs elképzelések pedig az informatika számára ma még felmérhetetlen mennyiségű probléma megoldását fogják célul kitűzni. Az informatikai problémák egyre növekvő hányada pedig szoftver-úton oldható meg. Ez adja a szoftvertechnológia jelentőségét a 21. század küszöbén.

Az 1940-es évek közepén készült el Neumann János első elektronikus számítógépe (ENIAC, JONIAC). Az azóta eltelt több mint fél évszázad alatt a számítógép szédületes karriert futott be. Ezt a karriert az alkalmazások által vele szemben támasztott igények motiválták.

Az első alkalmazásokat a sok számítással járó tudományos és műszaki feladatok megoldásának igénye határozta meg. Ez az igény szülte meg az első elektronikus programvezérelt számítógépeket, amelyek még elektroncsöves gépek voltak. Így jellemezhetjük a számítógépek alkalmazását az 50-es években.

A 60-as években már felismerték, hogy a szellemileg munkaigényes, de jelentős mértékben mechanikusan ismétlődő ügyviteli feladatokra is kiválóan alkalmazható a számítógép, ha ellátjuk azt lyukszalagos, lyukkártyás bemenettel és az eredmények kiírásához szükséges nyom-

tatókkal, valamint megfelelő háttértárolókkal. Sőt felismerték a számítógépek jelentőségét a gyorsan változó fizikai folyamatok adatainak gyűjtésében, és az eredmények kiértékelése alapján a folyamatok vezérlésében is. Gondoljunk például a hadiipar és az űrkutatás igényeire a 60-as években!

A békés termelésben a gyártósorok vezérlése, a termelés folyamatához kapcsolódó ügyviteli feladatok, a megrendelések, a megrendelésekhez szükséges raktárkészletek, a szállítások ütemezése stb. a 70-es években már komplex termelésirányítási rendszerek létrehozását igényelte.

A 80-as években már - a nagy kapacitású számítógépekkel kielégíthető alkalmazási igények mellett - megjelentek a személyi számítógépekkel kielégíthető alkalmazási igények. A nagy szolgáltató rendszerek (banki szolgáltatások, biztosítás) mellett előtérbe kerültek a kisvállalkozások, az oktatás, a játék, a szórakozás stb. igényei, az évtized jellemző alkalmazásaiként.

A 90-es évek alkalmazásait már az olvasók legtöbbje saját élménye alapján ismeri. Ez már a globális kommunikációs alkalmazások kora, a multimédia kora, amely ma már a távmunka igényét is magában foglalja.

Az alkalmazások igényének ezt a rohamos fejlődését a hardver követni tudta. Képes volt megoldani a hardveregységek mindenkor szükséges integrációjának problémáját. Képes volt az erőforrások nagyságrendekkel történő növelésére, miközben azok méretét hasonló mértékben csökkenteni tudta. Létre tudta hozni az alkalmazások által igényelt nagy teljesítményű illesztő egységeket és az ember-gép kapcsolat megkívánt eszközeit.

Bár a programok előállítása rendkívüli szellemi munkát igényel, az egyedi alkalmazásokkal a magasan képzett szakemberek képesek voltak megbirkózni. A nagy rendszerek létrehozásánál, amikor már reális időn belül csak több ember együttműködésével lehetett remélni a fejlesztés befejezését, egyre jobban nyilvánvalóvá vált, hogy a létező módszerek alkalmatlanok a feladatok megoldására. Ezekkel a módszerekkel a programok elkészítésének időpontja nem volt kézben tartható,

az elkészült programokban rejtett hibák maradtak, amelyek a használat során zavarokat okoztak, és a programok előállításának költsége előre megbecsülhetetlen módon növekedett. Ezt a jelenséget ismerték fel a szoftverszakemberek, amikor kimondták, hogy a szoftver előállításának gyakorlata krízishelyzetbe került. A probléma megoldásához szükség volt annak a felismerésére, hogy:

- a *program termékké* vált, és mint minden termék esetében
- az *előállításához technológiára* van szükség.

Mit jelent az, hogy a program termék? Azt, hogy:

- van *szolgáltatási funkciója,*
- van *minősége,*
- van *előállítási költsége,*
- van *előállítási határideje.*

A felsoroltak tervezési paraméterek. A szoftvertechnológiának biztosítania kell a tervezési paramétereknek megfelelő termék előállítását.

Természetesen a program előállításának problémái nem az úgynevezett kis programok esetében jelentkeztek. Ezek előállítása során a részleteket az előállító egyetlen személy akár a fejében is képes volt rendszerezve megtartani, és a hibátlan program érdekében felhasználni. A probléma a nagy programok esetében jelentett igazi gondot.

A *szoftvertechnológia tárgya* tehát a *nagy méretű programrendszerek* előállítása.

Melyek a nagy méretű programrendszerek jellemzői?

1. *Nagy bonyolultságú rendszer,* azaz fejben tartva nem kezelhetők a kidolgozás során felhasználandó részletek: az objektumok, azok jellemzői, összefüggései stb.
2. *Csapatmunkában* (teammunkában) készül.

3. *Hosszú élettartamú,* amelynek során számos változatát (verzióját) kell előállítani, azokat követni, karbantartani stb. kell.

Ezért a nagy méretű programrendszer esetében a következő fő feladatokat kell megoldani:

1. A rendszerrel szemben támasztott (funkcionális, szolgáltatási, minőségi stb.) követelményeket előre pontosan meg kell határozni, azokat írásban rögzíteni kell.
2. A program kidolgozásának menetét meg kell tervezni, meg kell határozni az úgynevezett mérföldköveket; azaz ellenőrzési pontokat, határidőket és a hozzájuk tartozó megoldandó feladatokat.
3. Gondoskodni kell a kidolgozáshoz szükséges hardver, szoftver, anyagi és emberi erőforrásokról.
4. Dokumentálni kell a programkészítés fázisainak menetét és eredményeit.
5. Szervezni, irányítani kell a kidolgozásban részt vevő csapat munkáját és az erőforrások felhasználását.
6. Igazolni kell, hogy az elkészült termék megfelel az előre rögzített követelményeknek.
7. Meg kell tervezni és szervezni a rendszer követésének, karbantartásának hosszú évekre elnyúló munkáját.

Ezen feladatok ellátását *projektmenedzselésnek* nevezzük, az a személy, aki ezért felelős, a *projektvezető* vagy projektmenedzser.
Összefoglalva a szoftvertechnológia célkitűzése:

- előírt minőségű programtermék
- előre megállapított határidőre,
- előre meghatározott költségen történő előállítása.

A technológia összetevői:

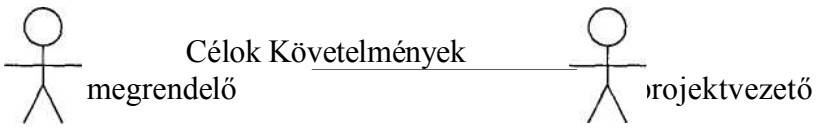
1. Módszerek a programkészítés különböző fázisai számára.
2. Szabványok (kidolgozási, dokumentációs stb.), amelyeket a program kidolgozása során kötelező betartani; és ajánlások, amelyek hozzájárulhatnak a program minőségének javításához.
3. Programeszközök, egységes rendszert alkotó programfejlesztési környezet, amelyek megkönnyítik és biztonságosabbá teszik az emberi munkát.
4. Irányítási módszerek a programkészítés folyamatának vezérlésére, szervezésére.

Először vizsgáljuk meg a projektmenedzselés folyamatát!

2.1. Projektmenedzselés

Szoftverprojektnek nevezzük a szoftvertermék (programrendszer) előállításával kapcsolatos tevékenységeket, az ajánlattevéstől a kész szoftvertermék átadásáig. A továbbiakban szoftverprojekt helyett a projekt fogalmat használjuk.

A *projektvezető* feladata a projekt tevékenységeinek ütemezése, összehangolása, és ő felelős a projekt sikeréért is. A megrendelőtől hozzá kerülnek a szoftverrel szemben támasztott előzetes követelmények, elvárások, amelynek alapján elindíthatja, megszervezheti a projektet (2.1. ábra).



2.1. ábra. Egy projekt kezdete

A projektvezető *felelőssége:*

- A termék nyújtsa a megkívánt szolgáltatásokat!
- A termék minősége feleljen meg az előírt követelményeknek!
- Készüljön el a termék határidőre!
- A projekt összköltsége ne lépje túl a megadott összeget!

A projekt *vezetése* a következőket foglalja magában:

1. Ajánlat készítése.
2. A projekt megvalósításának megtervezése.
3. A projekt megvalósításának költségbecslése.
4. A szükséges erőforrások biztosítása, az erőforrásokkal való gaz dálkodás a megvalósítás során.
5. A munka menetének irányítása, ellenőrzése.
6. Az eredmények bemutatása, átadása.

Ezek közül most az első kettővel foglalkozunk részletesebben.

Ajánlat

Az ajánlatnak a következő kérdésekre kell kitérnie:

- Mi a projekt tárgya?
- Milyen célok elérésére vállalkozik a projekt?
- Mi a téma gazdasági, műszaki háttere?
- Milyen módon kívánja a kitűzött célokat elérni?

- Mi az ajánlattevők szakmai háttere, milyen érveket, garanciá kat tudnak felsorakoztatni a vállalkozás sikere mellett? (Például szakmai referenciák.)
- A projekt tartalmi leírása fő fázisai szerinti bontásban, ráfordí tással, költséggel és határidőkkel együtt.
- Milyen szakmai, esetleg piaci előnyöket jelent a projekt sikere?

A projekt megvalósításának megtervezése

A projektvezetőnek ebben a fázisban a következőket kell elvégeznie:

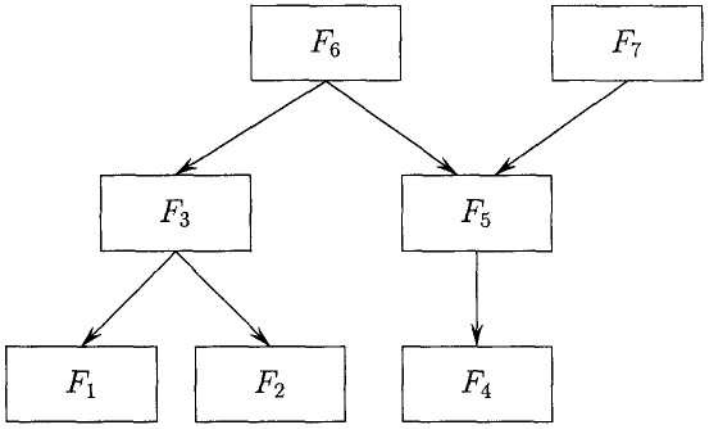
- A projekt részfeladatainak meghatározása.
- Mérföldkövek (ellenőrzési pontok) kijelölése.
- A részfeladatok megoldásához szükséges időtartamok megbecs lése.
- A részfeladatok egymástól való függéseinek meghatározása.

A *részfeladat* egy szoftverfejlesztési fázis, amelynek az eredményessége ellenőrizhető. Például a részjelentés (progress report) vagy a programkód formájában.

A *mérföldkövek* a részfeladatok ellenőrzési pontjai. Egy vagy több olyan részfeladat után helyezzük el, amely részfeladatok eredményes befejezése nélkül nem lehet továbbhaladni.

A *részfeladatok függése* más részfeladatoktól azt jelenti, hogy az adott részfeladat kidolgozása addig nem kezdődhet meg, amíg a szóban forgó részfeladatok nem fejeződtek be. Az eredményt egy *függőségi táblázatban* rögzíthetjük.

Tekintsük példaként a 2.2. ábrán látható, 7 részfeladatból álló programrendszert! A programrendszer fejlesztése alulról felfelé történik, azaz az alsóbb szinten található részfeladatokat kell előbb elkészíteni. A nyilak a felhasználás irányát mutatják, azaz a nyíl kezdőpontjánál található részfeladat függ a végpontnál szereplő részfeladattól.



2.2. ábra. Egy 7 részfeladatból (F_1 — F_7) álló programrendszer, ahol a nyilak a függőségeket ábrázolják

Az egyes részfeladatok kidolgozásának idejét megbecsültük, és a következőket kaptuk:
F1: 10 nap; F_2: 20 nap; F_3: 20 nap; F_4: 4 nap; F_5: 14 nap; F_6: 20 nap; F_7: 14 nap.

A megadott adatok alapján készítsük el a függőségi táblázatot, és határozzuk meg a mérföldköveket!

A függőségi táblázatot a 2.1. táblázat tartalmazza.

Részfeladat neve	Időtartam (nap)	Függőségek
F_1	10	-
F_2	20	-
F_3	20	F_1, F_2
F_4	4	-
F_5	14	F_4
F_6	20	F_3, F_5
F_7	14	F_5

2.1. táblázat. A 2.2. ábrán szereplő programrendszer függőségi táblázata

A függőségi táblázatban az ellenőrizendő pontok, a mérföldkövek a következők lesznek:

M_{12}: F_1 és F_2 helyes működését igazoló dokumentumok bemutatása.

M_4: F_4 helyes működésének dokumentálása.

M_5: F_5 helyességének dokumentálása. M_{35}: F_3 és F_5 helyes működését igazoló dokumentumok bemutatása.

Az eddigiek alapján elkészíthető a projekt *tervdiagramja,* amely tartalmazza a projekt kezdő és befejező időpontját, az egyes részfeladatokat és a mérföldköveket. Ennek érdekében be kell vezetnünk néhány jelölést.

• Legyen a részfeladat jelölése a következő:

<részfeladat neve>
<részfeladat időtartama>

• Egy mérföldkövet jelöljünk a következő módon:

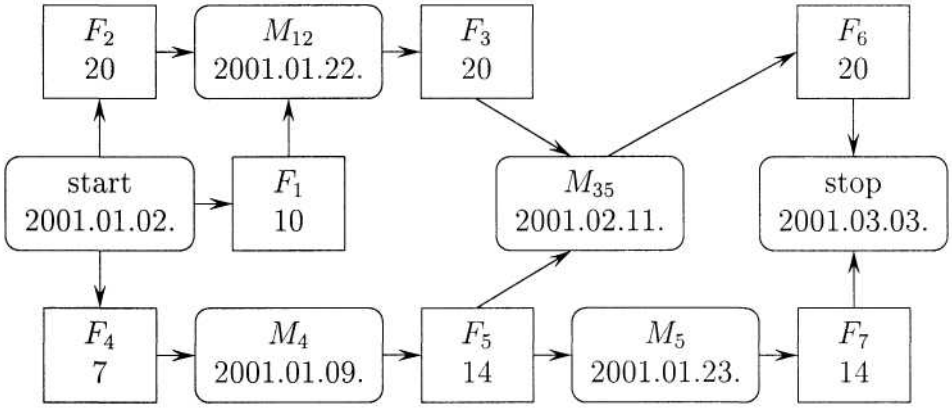
<ellenőrzés azonosítója>
<ellenőrzés dátuma>

• Legyen a projekt indulásának és befejezésének jele rendre:

start
dátum

stop
dátum

Az eddigiek felhasználásával elkészíthetjük a fenti projekt tervdiagramját, és meghatározhatjuk a projekt befejezésének legrövidebb határidejét. Az egyszerűség kedvéért az ellenőrzések időtartamát most



2.3. ábra. A projekt végrehajtásának tervdiagramja

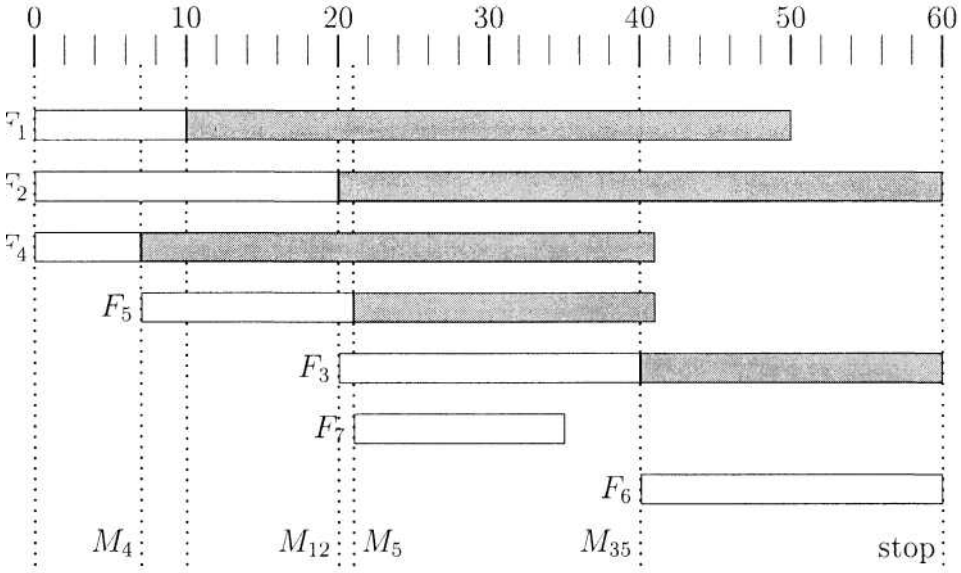
hanyagoljuk el! Tegyük fel, hogy a projekt 2001. január másodikán indul! Így a 2.3. ábrán látható tervdiagramhoz jutunk. Erről leolvasható, hogy a projekt végrehajtásának legrövidebb ideje: 20 + 20 + 20 = 60 nap.

A csúszások lehetőségeinek figyelembevételéhez célszerű feltüntetni azt is, hogy adott feladat elvégzése után minimálisan mennyi időre van szükség a projekt befejezéséhez, feltéve, hogy az őt megelőző és az őt követő feladatoknál nem lesz csúszás.

Ezt mutatja példánk esetében a 2.4. ábrán látható korlátok diagramja, amelyben a fehér téglalapok mutatják a részfeladatok kidolgozásának időtartamát, a szürke téglalapok pedig a projekt befejezéséhez szükséges minimális időtartamot. Így a szürke téglalap hossza a részfeladat befejezése után kidolgozandó részfeladatok kidolgozási idejeinek összege. A fehér téglalap kezdete a feladat lehető legkorábbi kezdési időpontjánál található. Ezt közvetlenül követi a szürke téglalap. A részfeladathoz tartozó szürke téglalap - ha ilyen nincs, akkor a fehér téglalap - vége és a befejezési időpont közötti szakasz hossza adja meg a részfeladat lehetséges csúszásának mértékét.

A diagramról leolvasható, hogy

F_2,F_3,F_6 esetében nem lehet csúszás,



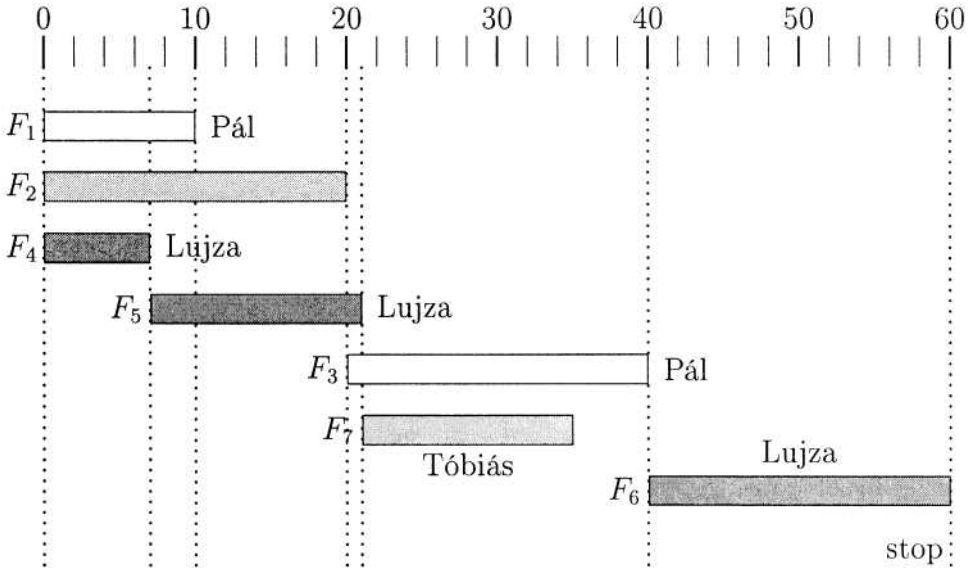
2.4. ábra. A korlátok diagramja

- F_1 kidolgozása 10 napot késhet,
- F_4 esetében 19 nap csúszás lehetséges,
- F_5 csúszása 19 nap lehet,
- F_7 25 napot késhet.

Ennek a diagramnak az alapján becsülhető meg a projekt erőforrás-szükséglete, a kidolgozáshoz szükséges szakemberek száma, a költség stb. is.

Például a szakemberek feladathoz rendeléséhez vegyük alapul a mérföldkövek nélküli korlátok diagramját a fenti példánk esetében! Feltesszük, hogy bármelyik szakember bármelyik részfeladatot megtudja oldani! Ekkor a személyek hozzárendelését az egyes feladatokhoz a 2.5. ábra diagramja mutatja.

A diagram alapján megállapítható, hogy 3 szakemberrel a feladat határidőre kidolgozható. Az is látható, hogy két szakember is elegendő,



2.5. ábra. A személyek hozzárendelésének diagramja

ha az egyikük az F_2,F_3,F_6, a másikuk az F_1, F_4, F_5, F_7 feladatokat dolgozza ki. Kettőnél kevesebben nem tudják határidőre befejezni a projektet.

A projektmenedzselés vizsgálata után rátérünk a programfejlesztés folyamatára. A következőkben áttekintjük a leggyakoribb *programfejlesztési modelleket,* amelyek a következők:

1. Egyszerű programfejlesztési modell.

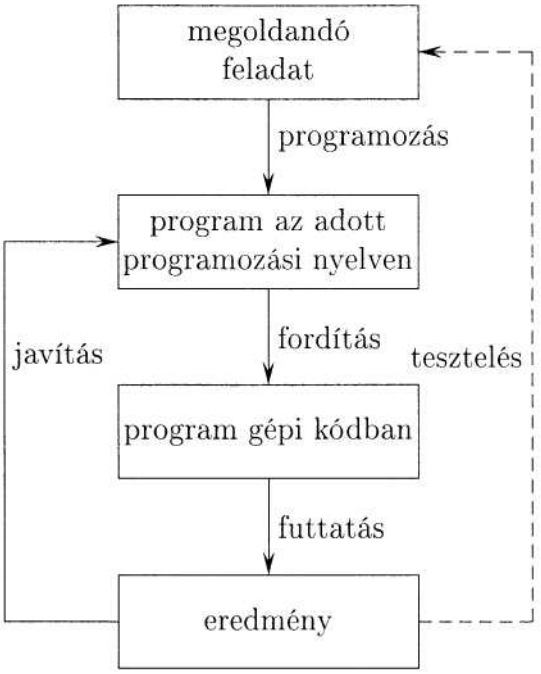
2. Programfejlesztés specifikációval.

3. Nagy rendszerek fejlesztésének általános modellje.

4. Programfejlesztés modellalkotással.

2.2. Egyszerű programfejlesztési modell

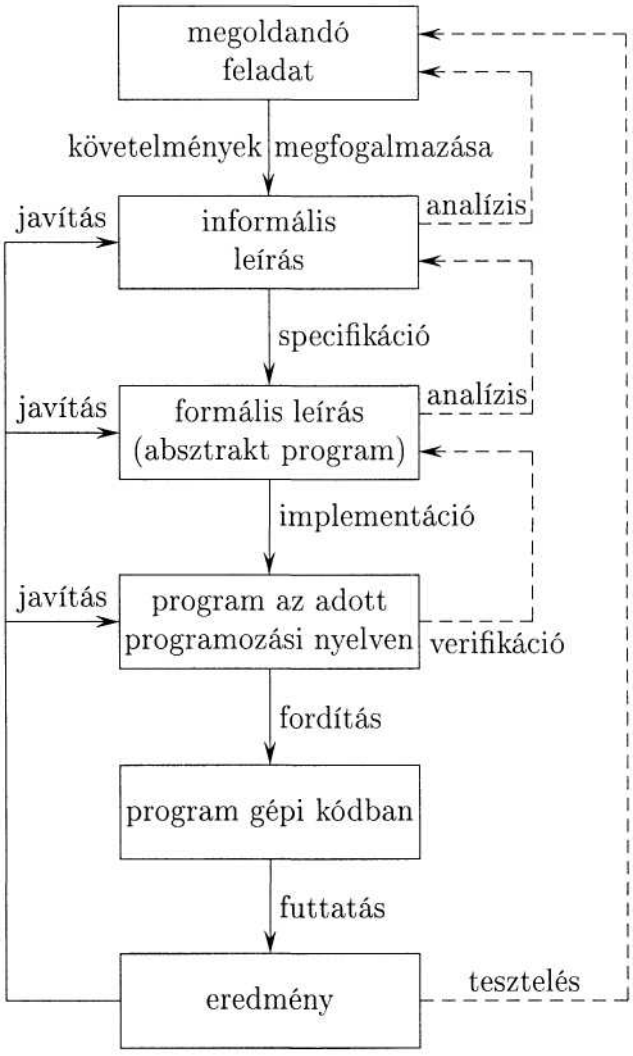
Rendszerint ezt a modellt alkalmazzuk egyszemélyes programfejlesztésnél. Ebben az esetben a megoldandó feladat könnyen áttekinthető, modellezhető, ezért a probléma azonnal megfogalmazható egy adott programozási nyelven. Ezután az eredmény előállításáig szoftvereszközök (fordítóprogram, ...) veszik át a feladatot. A futási eredményeket a feladattal vetjük egybe, és a javításokat közvetlenül a programozási nyelvű leírásban, a programkódban hajtjuk végre (2.6. ábra). Ez tehát a kis programok létrehozásának a modellje.



2.6. ábra. Az egyszerű programfejlesztési modell

2.3. Programfejlesztés specifikációval

A programkészítés folyamatát - bizonyos körülmények esetén - matematikailag is kezelni tudjuk. Ennek a megközelítésnek a terméke a



2.7. ábra. A specifikációra épülő programfejlesztési modell

specifikációra épülő programfejlesztési modell (2.7. ábra).

A feladat megoldását először beszélt nyelven, megfelelően rendszerezve írjuk le. Ennek az eredménye egy informális leírás. Ezt a leírást elemezhetjük, és ennek során összevetjük a megoldandó feladattal annak érdekében, hogy valóban a kívánt megoldást rögzíti-e. Mivel ez a leírás beszélt nyelven készül, ezért jelentése félreértésre is okot adhat, így ez a megoldásnak szintaktikailag és szemantikailag nem egyértelmű leírása.

Az informális leírás alapján készítjük el egy specifikációra alkalmas nyelven a probléma megoldásának első formális, ezért szintaktikailag és szemantikailag egyértelmű leírását. Ennek a leírásnak a helyességét az informális leírással való összevetéssel vizsgálhatjuk, analizálhatjuk. Ez az analízis azonban bizonyos formális bizonyításokat is tartalmazhat. Az informális leírás alapján megfogalmazhatjuk a specifikációs nyelven a megoldás olyan tulajdonságait, amelyek explicit módon a specifikációban nem szerepelnek, de az informális leírásban jelen lehetnek. Ezeket, mint tételeket, matematikailag bizonyíthatjuk a formális rendszerben. Ilyen módon ez az analízis - bizonyos értelemben - verifikációs analízis.

A specifikáció egy absztrakt program leírását szolgáltatja. A konkrét programot, amely egy adott programozási nyelven készül, ezzel vetjük össze. Itt azonban már két szintaktikailag és szemantikailag egyértelmű leírást hasonlítunk össze. A konkrét programnak az absztrakt program szerinti helyességét tehát - elvileg - bizonyítani, verifikálni lehet. A verifikáció formális végrehajtása azonban a gyakorlati esetekben rendszerint nehézségekbe ütközik, és ezért kivitelezhetetlennek bizonyul. Ennek oka, hogy nagyobb méretű feladatok esetén a formális leírás túl bonyolulttá válik, előállítása rengeteg munkát és magas szintű matematikai ismereteket igényel.

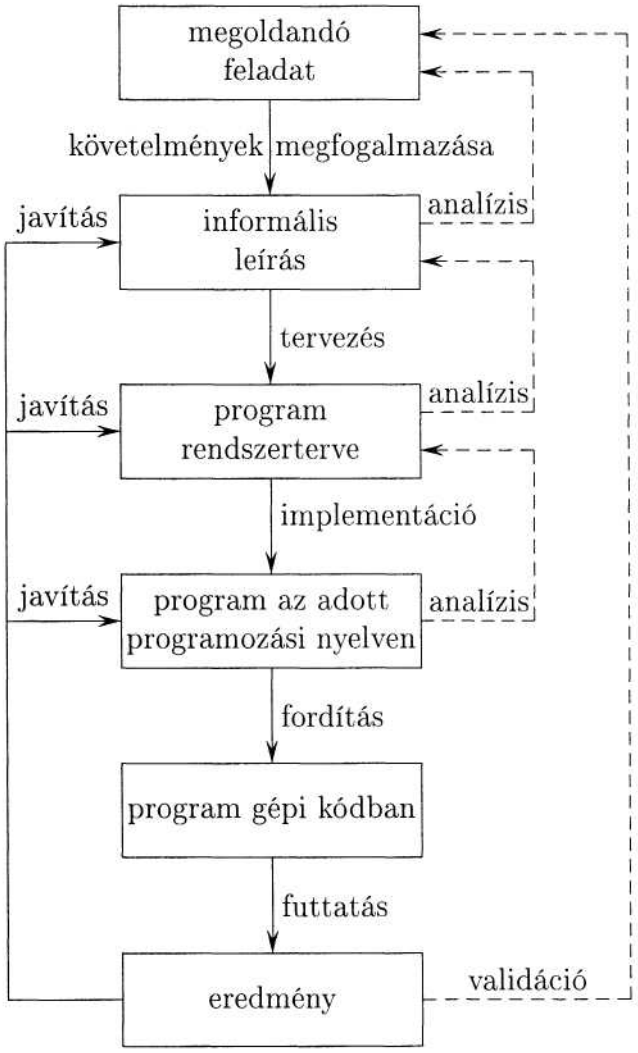
A tesztelés eredménye alapján azután a javítások érinthetik - a nem verifikált programrészek esetében - a programozási nyelven írt változatot vagy az analízis hiányosságai miatt a specifikáció során létrehozott absztrakt programot vagy a beszélt nyelven megfogalmazott informális leírását a probléma megoldásának.

Összefoglalva ennek a fejlesztési módszernek

- az előnyei:
  - Magas szintű, az absztrakt szintaxis és a szemantika szempontjából pontos megoldás a programkészítés korai fázisában.
  - Az absztrakt leírás (specifikáció) formális elemezhetősége.
  - A konkrét leírás (a program) helyességének bizonyíthatósága az absztrakt leírás szerint.
- A hátrányai:
  - A matematikailag kezelhető formális specifikációs módsze rek alkalmazása, elemzése magas szintű matematikai isme reteket kíván meg.
  - A formális specifikáció alkalmazása nagyobb feladatok leí rására nehézkes, és általában áttekinthetetlen.
  - A matematikai bizonyítás rendszerint egyszerű és automatizálható, de a tételek megfogalmazása összetettebb esetekben már nehéz feladat.

2.4. Nagy rendszerek általános fejlesztési modellje

Nagy rendszerek esetén a formális specifikációs módszerek legfeljebb kisebb részfeladatok - például adattípusok - formális leírását, specifikációját teszik lehetővé. Ezért ebben az esetben a specifikációs modell formális leírásának helyébe a programrendszer tervének elkészítése lép. Ez egy jól strukturált, többszintű leírás - esetleg kisebb részfeladatok formális definíciójával -, amely tartalmazza az implementáció számára szükséges ajánlásokat, előírásokat (2.8. ábra).



2.8. ábra. Nagy rendszerek fejlesztésének egy általános modellje

Ehhez a modellhez alakult ki a programkészítés hagyományos fázisainak az a rendszere, amellyel a következőkben kicsit részletesebben foglalkozunk. Megvizsgálunk néhány, a gyakorlatban elterjedt modellt, majd az egyes fázisokat mutatjuk be részletesen.

A programkészítés hagyományos fázisai:

- követelmények leírása,
- specifikáció,
- tervezés (vázlatos tervezés, finom tervezés),
- implementáció, integráció,
- verifikáció, validáció,
- rendszerkövetés, karbantartás.

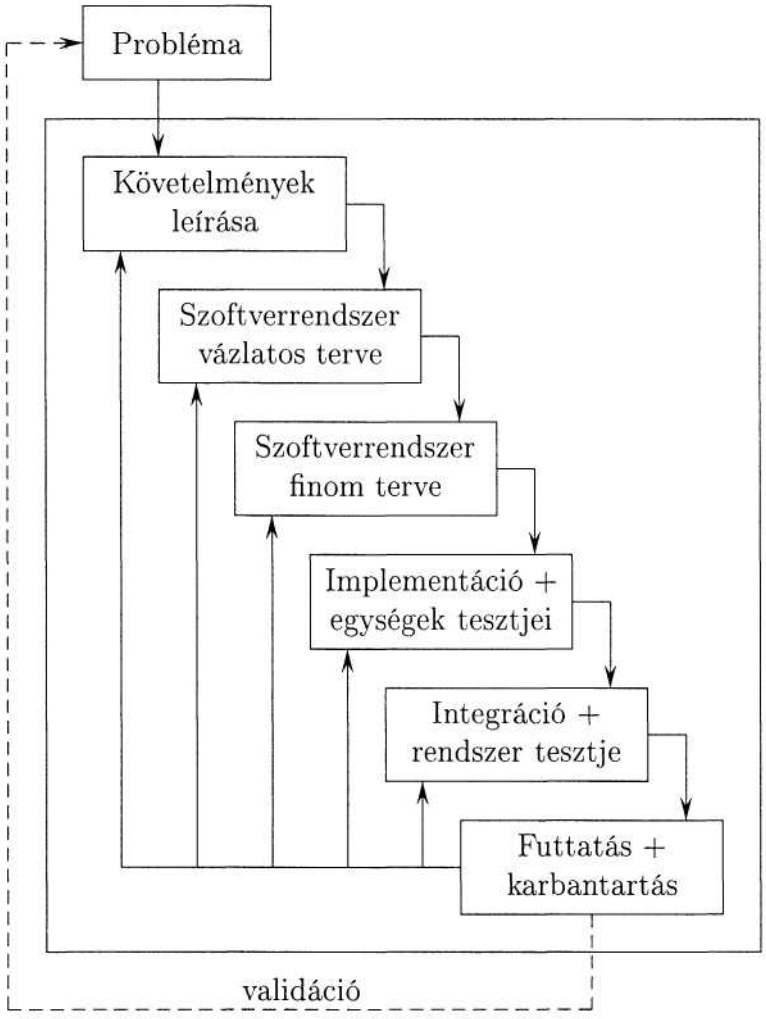
A fázisok közötti kapcsolatok leírására is kialakultak modellek:

1. Vízesés modell.
2. Evolúciós modell.
3. Boehm-féle spirális modell.

2.4.1. Vízesés modell

A modell szerinti kapcsolatokat mutatja a 2.9. ábra. Az ábráról jól leolvasható a fejlesztés menete. Az egyes fázisok egymást követik, a módosítások a futtatási eredmények ismeretében történnek. Egy bizonyos fázisban elvégzett módosítás az összes rákövetkező fázist is érinti. Ezt a modellt a gyakorlat szülte. A vízesés modell hátrányai:

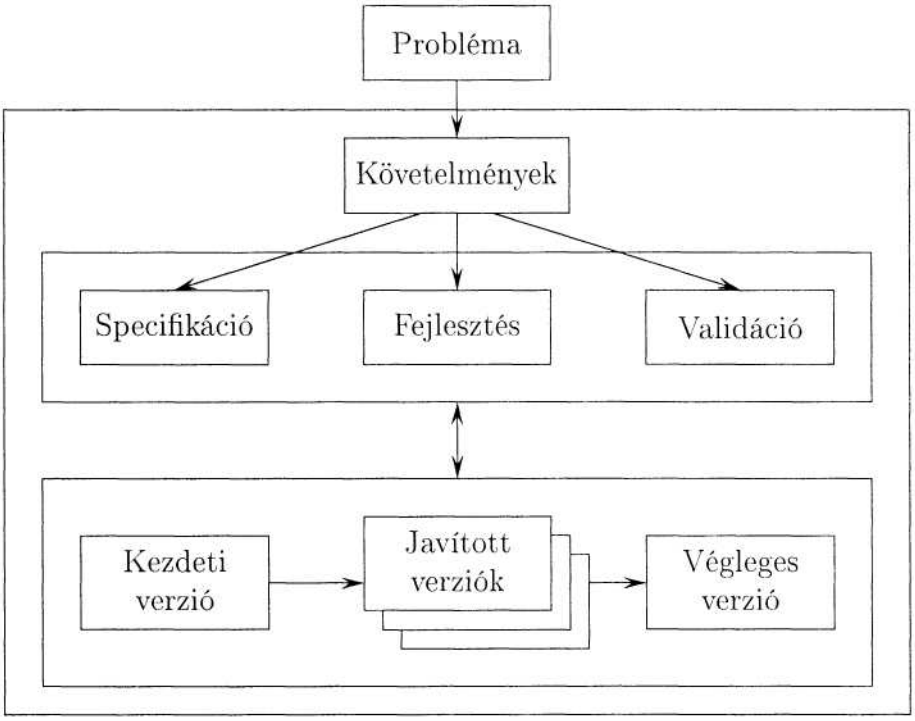
- A projekt munkájának megszervezése rendkívül nehézkes. (Pár huzamos munka, ellenőrzési pontok elhelyezése.)
- Új szolgáltatások utólagos bevezetése szinte minden fázison mó dosítást kíván meg.
- A validáció az egész életciklus megismétlését követelheti meg.



2.9. ábra. A vízesés modell

2.4.2. Evolúciós modell

Az evolúciós modellt szemlélteti a 2.10. ábra. A modell lényeges vonása, hogy a megoldás közelítő verzióinak, prototípusainak sorozatát állítjuk egymás után elő, és így haladunk lépésenként egészen a végleges megoldásig. Ennek során egy verzió elkészítésekor a specifikáció, a



2.10. ábra. Az evolúciós modell

fejlesztés és a validáció párhuzamosan történik.

A programrendszerek fejlesztése során rendszerint nem áll rendelkezésünkre egy teljes követelményleírás. Nagyon sok részlet csak a felhasználó fejében létezik, így tehát nem is tudunk egy teljes és ellentmondásmentes specifikációt készíteni. Ennek alapján vagyunk kénytelenek megkezdeni a fejlesztést. Előállítjuk tehát a megoldás egy első verzióját. Ehhez elkészítjük a felhasználói felületet szimuláló prototípust. Ezt egyeztetjük a felhasználóval, és az elemzés alapján döntünk a következő verzió előállításáról. A verziók sorozatának fejlesztésével közelítünk a végső megoldáshoz.

A modell fő erénye a működés közben tanulmányozható prototípusok sorozatának megléte. Különösen előnyös ez olyan feladatok esetén, amikor nehézséget okoz a részletes specifikáció elkészítése.

A modell hátrányai:

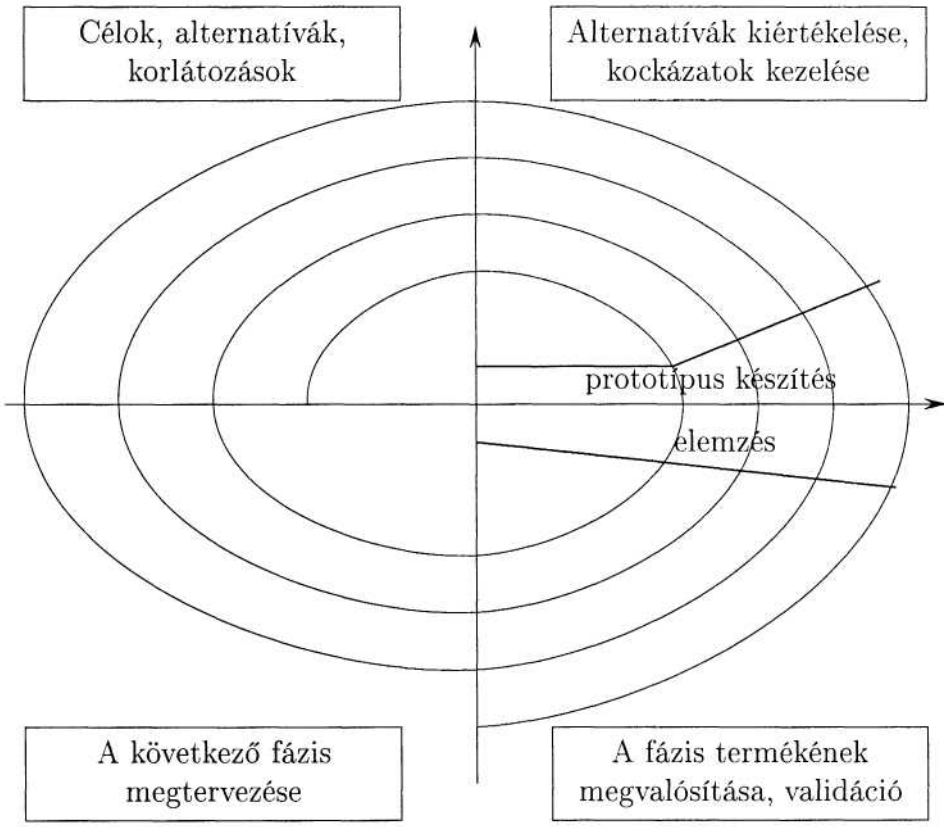
- Elsősorban csak rövidebb idő alatt megvalósítható rendszerek esetén ajánlható.
- A rendszer utólagos módosítása, bővítése nehézkes.
- Nehéz a projekt áttekintése.
- A gyors fejlesztés rendszerint a dokumentáltság rovására megy.

2.4.3. Boehm-féle spirális modell

Boehm 1988-ban javasolta programrendszerek kidolgozására ezt a modellt. A javaslat lényege az a felismerés, hogy a programok rendszerint iterációkon keresztül nyerik el végső formájukat. Javaslata szerint ezek az iterációk spirálisba olvaszthatok össze. Az iteráció a spirális egy fázisával modellezhető, amely négy szakaszra bontható (2.11. ábra). Ezek a következők:

1. Az adott iterációs lépés során elérendő célok és a megoldást kor látozó feltételek definiálása. A megoldás lehetséges útjainak, az alternatíváknak a meghatározása.
2. A különböző megoldások során felmerülő esetleges kockázati té nyezők elemzése, stratégiák kidolgozása a kockázati tényezők hatásának csökkentésére, fellépésük elkerülésére; szükség szerint prototípusok készítése, és azok eredményeinek értékelése.
3. Az előző pontban javasolt elemzés alapján az iterációs lépés fela datának megoldása, a megoldással szemben támasztott követel mények teljesülésének igazolása, validáció.
4. A következő iterációs lépés megtervezése.

A modell leírása során felhasznált néhány fogalom értelmezését adjuk meg a következőkben:



2.11. ábra. Boehm-féle spirális modell vázlata

Cél: az elemzés tárgya.

Korlátozás: ami a megvalósításban határt szab a lehetőségeknek.

Alternatívák: a célok megvalósításának különböző útjai.

Kockázatok: az egyes alternatívák nagy valószínűséggel hibát okozó forrásai.

Kockázat kezelése: stratégia a kockázat hatásának a csökkentésére.

Validáció: a terv előírt célok szerinti teljesülésének ellenőrzése.

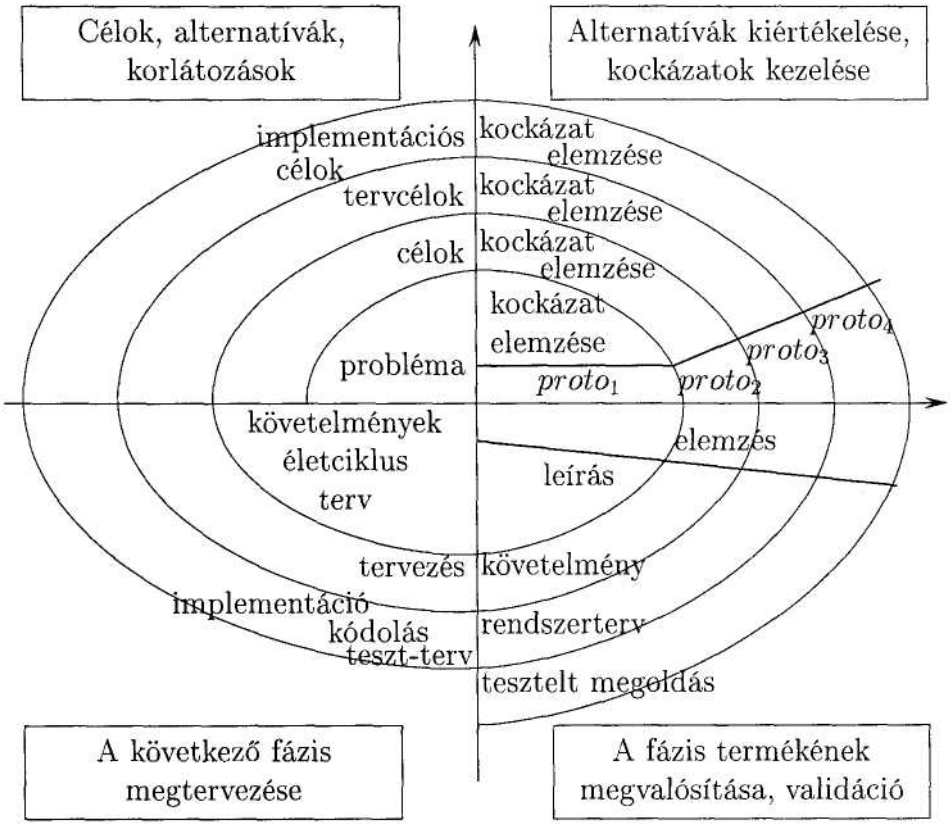
A spirális modell a projektmunka szervezésének áttekinthető, szabványos modellje, amelynek előnyeit a következőkben foglalhatjuk össze:

- A projekt kidolgozásának jó dokumentáltságát eredményezi, mivel mind a négy szakaszban dokumentumok készülnek.
- A projekt strukturáltsága, az iterációs fázisok - a projektvezető elképzelésének megfelelően - szabadon megválaszthatók.
- A fázisokon belüli szakaszok tartalmának (a kockázati tényezők, azok kezelése stb.) meghatározása szintén iterációs lépésenként, a projektvezető elképzelése szerint történhet meg.
- A fejlesztést, az eredményt mindig validáció követi.
- Ciklusonként döntés születhet a projekt egy újabb fázisáról, ha az eredmény alapján az kívánatos.

A spirális modell alkalmazásával kapcsolatban a következő problémákat szokták felvetni:

- A modell alkalmazása általában munkaigényes, bonyolult fel adat. A programkészítés általunk választott minden fázisának fenti módon történő szakaszokra bontása és végrehajtása általá ban költséges és nehezen megoldható.
- A projekt kidolgozásához szükséges szakembereket nem könnyű gazdaságosan foglalkoztatni. A párhuzamos foglalkoztatás szinte csak a 3. szakasz során lehetséges. Ezért a modellel kapcsolatban gyakran elhangzik a gazdaságtalanság jelzője.

Példaként készítsük el a nagy programrendszerek fejlesztésének részleteiben megismert projektjét spirális modell formájában! A spirális fázisai: probléma leírása, követelmények megfogalmazása, terv készítése, megvalósítás (2.12. ábra).



2.12. ábra. Boehm-féle spirális modell

1. A probléma leírásának megoldása, amelynek fázisai:

   - Megfogalmazzuk a felmerült probléma megoldásával kap csolatos célkitűzéseinket. Számba vesszük céljaink elérésé nek járható útjait (mi a tervünk a kidolgozó kiválasztására, a finanszírozásra, a haszonra stb.), tervünkkel kapcsolatos korlátainkat.
   - Elemezzük a felmerült alternatívákat, és kiválasztjuk a leg kedvezőbbet. Leírjuk, hogy milyen problémák merülhetnek fel (kivitelezési, finanszírozási, piaci problémák stb.), azokat

elemezzük, és meghatározzuk azok elkerülésére vonatkozó stratégiánkat. Elkészítjük a probléma leírásának vázlatát, egy prototípust *(proto$_1$)*. Elvégezzük ennek elemzését, és kialakítjuk a végleges elképzelésünket.

- Kidolgozzuk a megoldandó probléma végleges leírását, és elvégezzük a validációt.
- Megfogalmazzuk a követelmények kidolgozására, az életcik lusok megválasztására vonatkozó tervünket.

2. A követelmények megfogalmazásának megoldása, amelynek fá zisai:

   - A probléma elemzése alapján elkészítjük a követelmények kidolgozására vonatkozó célkitűzéseinket. Alternatív meg oldásokat fogalmazunk meg például arról, hogy milyen ré szeknél milyen követelményleírásokat (definíció, specifiká ció) alkalmazzunk. Számba vesszük az erőforrásaink (sze mélyek, eszközök, határidők) által támasztott korlátjainkat.
   - Alternatívák elemzése, a legjobbnak ítélt megoldások kivá lasztása. Kockázatok felvetése, elemzése. Például megvizs gáljuk, hogy melyik rész specifikációjának kidolgozása, me lyik szakemberünk által, milyen következményekkel járhat, és ennek megfelelően döntünk. Elkészítjük a követelmények vázlatos leírását *(proto$_2$)*, azt elemezzük.
   - Elkészítjük a követelmények végleges leírását, és elvégezzük annak validációját.
   - Megfogalmazzuk a terv elkészítésére vonatkozó előírásun kat.

3. A terv elkészítése, amelynek fázisai:

   - Meghatározzuk, hogy a terv készítése során mik a célki tűzéseink. Számba vesszük az alternatívákat: például meg vizsgáljuk, hogy milyen szerkezeti megoldások jöhetnek szá mításba. Meghatározzuk a korlátokat: például megadjuk a

szerkezet kialakítására vonatkozó előírásokat, ajánlásokat, szabványokat.

- Elemezzük a felmerült megoldásokat. Megvizsgáljuk az al ternatívákat a kockázatok szempontjából. Például az elem zés alapján kiválasztjuk a programrendszer legjobbnak ítélt szerkezetét. Kidolgozzuk a rendszer tervének egy megoldá sát *(proto$_3$),* és elemezzük azt.
- Elkészítjük a rendszer végleges tervét, specifikáljuk annak egységeit, és végrehajtjuk a validációt.
- Kidolgozzuk az implementációra vonatkozó tervet. Ebben rögzíthetjük például az implementációs ajánlásokat.

4. A rendszer megvalósítása, amelynek fázisai:

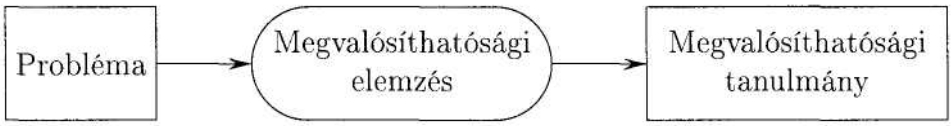
- Rögzítjük az implementációra vonatkozó célkitűzéseinket. Ennek során például implementációs stratégiákat fogalma zunk meg, adatbázisokra vonatkozó alternatívákat adunk meg. Megadjuk az implementációra vonatkozó szabványo kat, ajánlásokat, egyéb korlátozó tényezőket.
- Az alternatívák és a korlátozó tényezők elemzése alapján döntünk a megoldásról, és végrehajtjuk az implementációt *(proto$_4$).*
- A megoldás elemzése alapján elkészítjük a végleges megol dást, és teszteljük azt.

A tesztelés, a validáció külön iterációs lépést jelenthet nagy rendszerek esetén. Példánkban mi most itt megállunk, mivel célunk a spirális modell alkalmazásának szemléltetése volt.

A következőkben a nagy rendszerek fejlesztésének fázisaival foglalkozunk részletesebben.

2.4.4. A probléma megoldásának előzménye

Egy adott probléma megoldása előtt meg kell vizsgálni annak megvalósíthatóságát, illetve annak miként jét. Ez egy elemzés tárgya, amelynek eredménye a *megvalósíthatósági tanulmány* (2.13. ábra).



2.13. ábra. A probléma megvalósíthatóságának elemzése

A megvalósíthatósági tanulmány a következő kérdésekre válaszol:

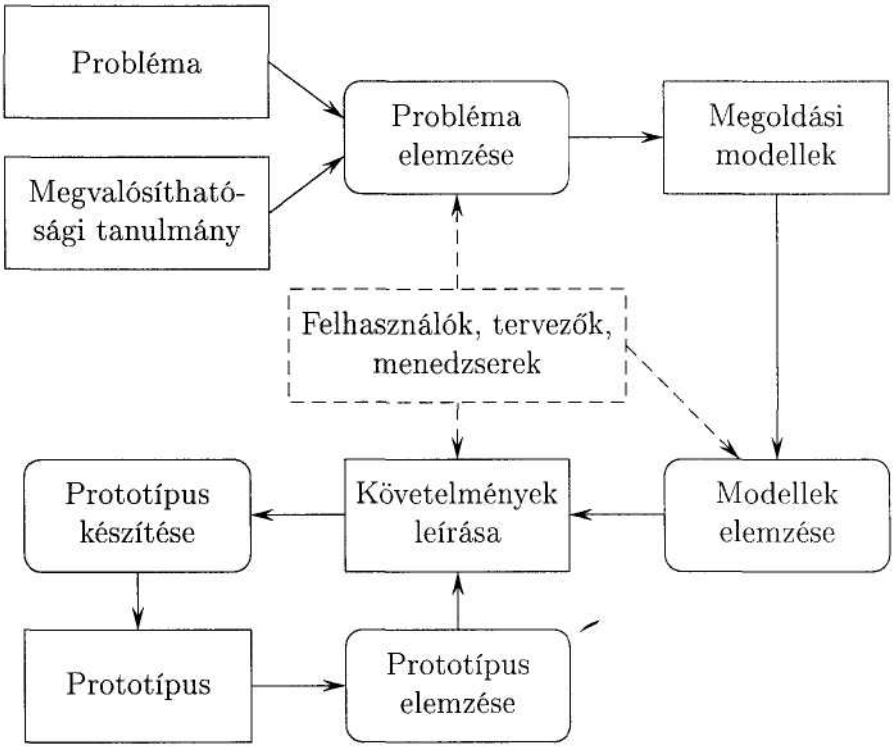
- A probléma milyen erőforrások felhasználásával oldható meg?
  - Hardver-szoftver környezet.
  - —Szakemberigény.
- Milyen költséget jelent a rendszer létrehozása?
- Mikorra várható, hogy kész lesz?
- A rendszer üzemeltetése milyen feladatokat jelent, milyen költséggel jár?

2.4.5. Követelmények leírása

A követelmények leírását rendszerint iteratív módon állítjuk elő, és ebben a folyamatban a prototípust használjuk a leírás finomítására, pontosítására (2.14. ábra).

A követelmények leírásának tartalmaznia kell a válaszokat a következő kérdésekre:

- Mi a probléma?
- Mik a korlátozó tényezők? (Hardver, szoftver stb.)

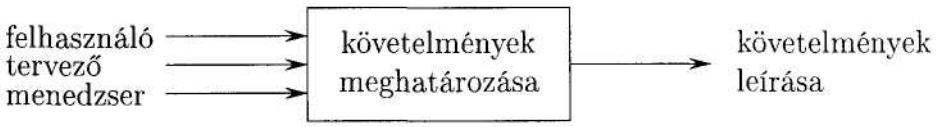


2.14. ábra. Követelményleírás elkészítésének folyamata

• Mi a probléma elfogadható megoldása? (Minőség)

A leírás elkészítését nevezzük a követelmények meghatározásának. Ennek módszere: interjúk, hagyományos megoldásban való részvétel (2.15. ábra).

A követelmények leírásának fajtái:

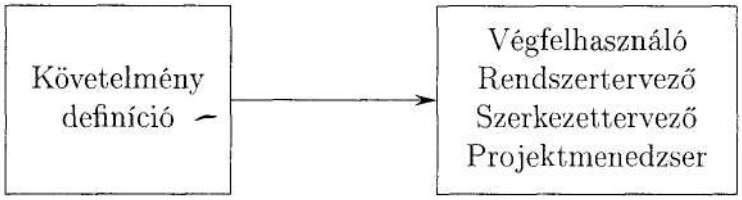


2.15. ábra. Követelmények meghatározása

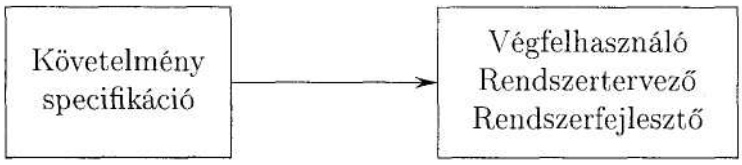
- Funkcionális követelmények leírása. (A rendszer által nyújtandó szolgáltatások leírása.)
- Nem funkcionális követelmények leírása. (Korlátozások az erő forrásokra, környezetre. Az átadási feltételek meghatározása.)

A követelmények leírásának szintjei:

- Követelmény definíció. (Beszélt nyelven leírt állítások, amelyeket diagramok egészítenek ki.)



Követelmény specifikáció. (Szintekre tagolt, szemantikailag egyértelmű, részletes leírás a rendszertervezés számára.)



Program specifikáció. (Szintaktikailag és szemantikailag egyértelmű absztrakt leírás a megvalósítás számára.)

Nagy rendszerek esetén a követelmények leírása szinte sohasem teljes és végleges.

Funkcionális követelmények leírása

A rendszer szolgáltatásainak, leképezéseinek a leírása, amely a következő kérdésekre válaszol:

- Mi a formája a szolgáltatás kezdeményezésének?
- Mik a szolgáltatás bemenő adatai; hogyan, milyen módon kell ezeket megadni?
- Mi a szolgáltatások igénybevételének előfeltétele, mik a korlátozások?
- Mi a szolgáltatás kezdeményezésére a válasz, mik az eredmények?
- Milyen formában, hol jelenik meg a válasz?
- Mi a bemenő adatok és az eredmények közötti reláció?

A funkcionális követelmények leírásának módszerei:

1. *Beszélt nyelven történő leírás.*
   Egy szöveges dokumentumot készítünk, amelyben lehetőleg jól tagoltan írjuk le a követelményeket. Ez könnyen érthető, ugyanakkor ellentmondásos és hiányos lehet.

2. *Formáknak alávetett leírás.*
   Például a függvények specifikációját alapszavakkal vezérelt formában adjuk meg:

   function: számla-felújítása
   input: számla, összeg, tranzakciótípus
   output: készpénz, elismervény, üzenet, számla
   transformation:
   ha a kért összeg nagyobb, mint amekkora a számla összege, **akkor** a tranzakciót törölni kell, és ki kell írni a következő üzenetet: „készpénzkeret túllépve";

ha a kért összegre van a számlán fedezet, **akkor** ...

constraints:

- a kért összeg nem lehet negatív;
- a kért összeg legfeljebb a hitelhatárral (hitel limit) lépheti át a számla összegét;
- stb.

Előny:

- A specifikáció speciális szoftvereszközökkel támogatható a formák vonatkozásában.
- Vezérfonalat nyújt a specifikáció teljesebb és ellentmondásmentesebb elkészítéséhez.

Hátrány:

- A beszélt nyelv ellentmondásos volta megmarad.
- Bizonyos feladatok megfogalmazása nehézséget okozhat az adott kereten belül.

3. *Követelmények leírására szolgáló nyelvek, grafikus modellezési eszközök használata.*
Például: PDL, PSL, SADT. Ezek programszerű nyelvek, amelyek absztrakt leírást tesznek lehetővé. Például PDL esetén az ATM specifikációja a következő:

procedure ATM **is**
PIN : azonosító kód;
No: számla száma;
SUM: számla összege;
Szolg: választható szolgáltatások;
V-card, V-PIN : bool;

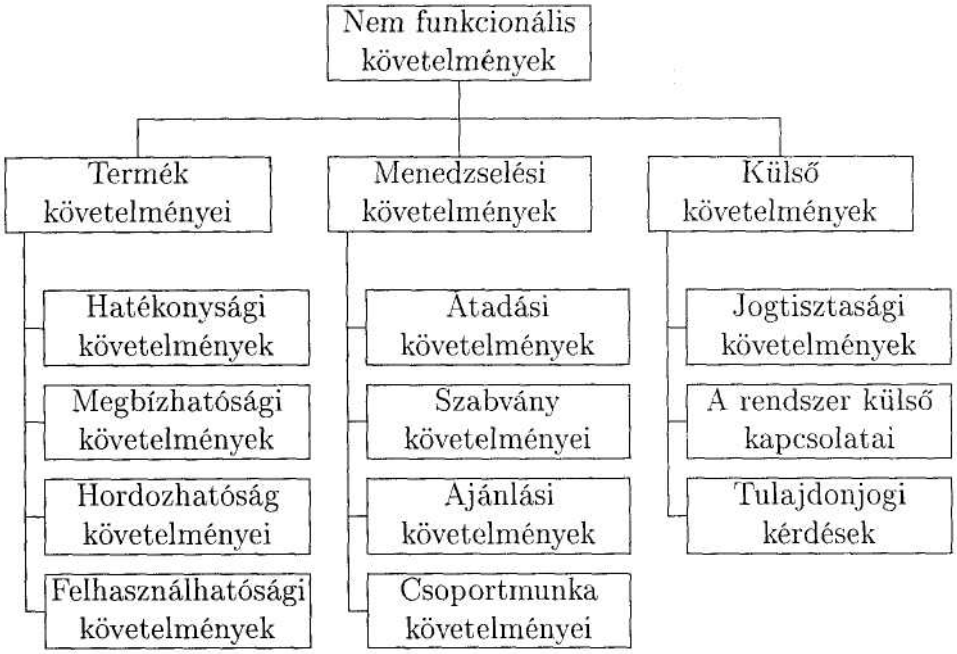
begin loop
kártya-beolvasása(No,PIN,V-card);
if V-card **then**
PIN-kód-ellenőrzése(PIN, V-pin); **if**
V-pin **then**
összeg-beolvasása(No, SUM); választható-szolgáltatások-beolvasása(Szolg); **while** egy szolgáltatás kiválasztódik **loop** írd-ki-a-választékot; vedd-át-az-igényt (Szolg); **end loop;** add-ki-a-kártyát; **end if; end if; end loop; end** ATM; Ezután jön a kiszolgálás, pénz, számla kiadása.

4. *Formális, matematikai leírások.*
Ekkor a szintaxis és szemantika egyértelműen meghatározott. Szoftvereszközökkel a specifikáció helyessége ellenőrizhető. Lehetséges formái például a következők.

- Elő-, utófeltételes forma, ha $pre(a)$ a bemenetre vonatkozó előfeltétel, $post(a, a')$ pedig az utófeltétel:
$\{pre(a)\}\ a' = f(a)\ \{post(a, a')\}$.
- Axiomatikus specifikáció; például adattípus f_s, f_c műveleteire algebrai axiómákkal:
$\alpha(a) \Longrightarrow f_s(f_c(a)) = h(a)$.

Nem funkcionális követelmények

A nem funkcionális követelményeket rendszerint három osztályba soroljuk: a termék követelményei, menedzselési követelmények, külső kö-



2.16. ábra. Nem funkcionális követelmények osztályozása

vetelemények. Az osztályokat tovább lehet bontani (2.16. ábra).

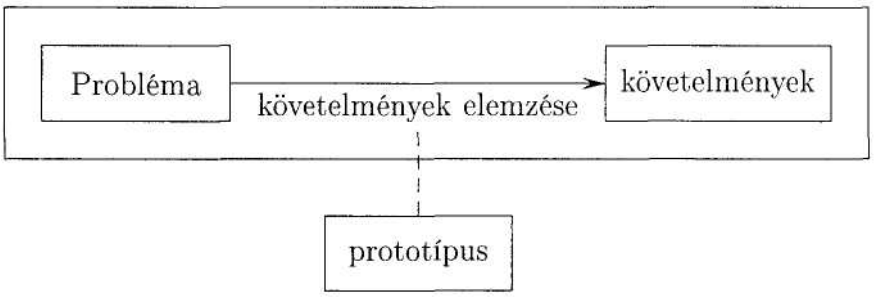
2.4.6. Követelmények elemzése és prototípus

A követelmények elemzése során a következő kérdéseket kell megvizsgálni:

- Önmagában jó-e a követelmények leírása?
  - Nincs-e benne ellentmondás? *(Konzisztencia vizsgálat.)*
  - —Teljes-e a leírás? *(Komplettség vizsgálat.)*
- A követelmények leírása megfelel-e a felhasználó által elképzelt problémának? (*Validáció vizsgálat.)*

- A követelményeket kielégítő megoldás megvalósítható-e? *{Megvalósíthatósági vizsgálat.)*
- A követelmények úgy vannak-e megfogalmazva, hogy azok tesztelhetők? *(Tesztelhetőség vizsgálata.)*
- A követelmények nem mondanak-e ellent a módosíthatóság, a továbbfejleszthetőség követelményének? *(Nyíltság kritériumainak vizsgálata.)*

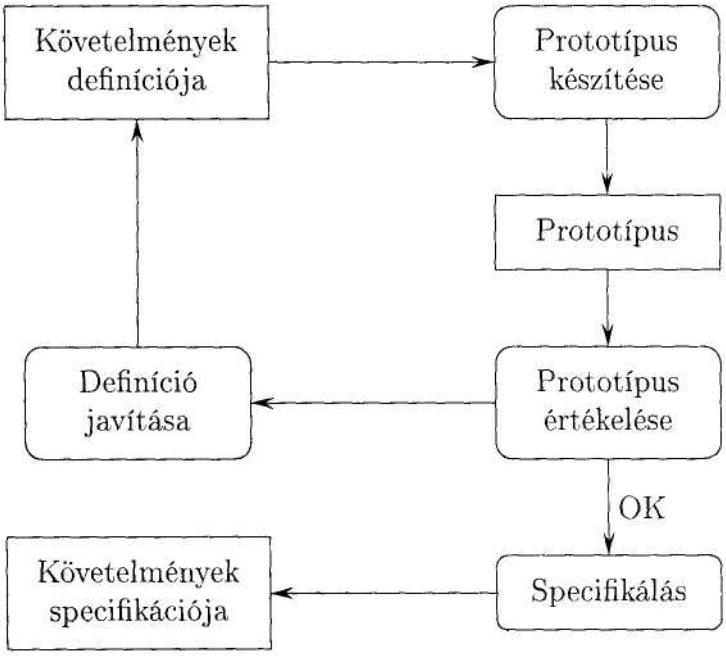
A követelmények elemzésének egyik eszköze a prototípus-készítés (2.17. ábra).



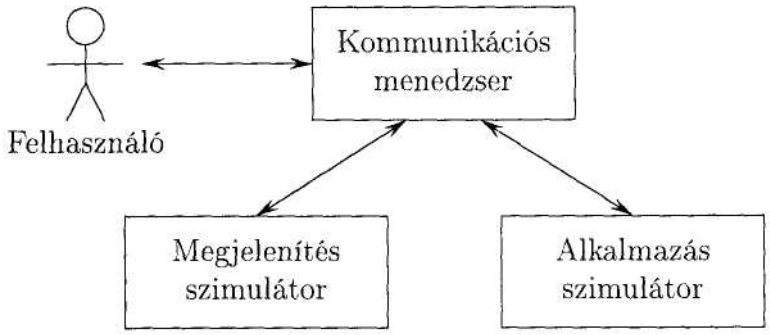
2.17. ábra. A prototípus szerepe a követelmények elemzésében

A *prototípus* magas szintű programozási környezetben létrehozott, a *külső viselkedés szempontjából helyes* megoldása a problémának. A prototípus a következő feladatokra alkalmas:

- Konzisztencia, komplettség, validáció vizsgálat.
- Félreértések tisztázása a felhasználók és a fejlesztők között.
- A prototípus alapján a felhasználó megítélheti az alkalmazás hasznosságát.
- A prototípus a felhasználók felkészítésének eszköze lehet.
- A prototípus elősegítheti a követelmények specifikálását, amint az a 2.18. ábrán látható.



2.18. ábra. A prototípus szerepe a követelmények meghatározásában



2.19. ábra. Felhasználói felület szimulációja mint prototípus

A prototípus viszonylag gyorsan megvalósítható formája a felhasználói felület szimulátora (2.19. ábra). Ennek segítségével a felhasználót be lehet vonni a követelmények meghatározásába, pontosításába.

2.4.7. Programspecifikáció

A programspecifikáció a következő kérdésekre kell, hogy válaszoljon a követelmények leírása alapján:

- Mik a bemenő adatok? (Forma, jelentés, megjelenés.)
- Mik az eredmények? (Forma, jelentés, megjelenés.)
- Mi a reláció a bemenő adatok és az eredmény adatok között, mi azok viszonya?

A specifikációs módszereket a következőképpen szokás csoportosítani:

- informális,
- formáknak alávetett,
- formális módszerek.

Az informális specifikációban a feladatot a beszélt nyelven, bizonyos előírások betartásával, rendszerezetten írjuk le. Erről és a hátrányairól már ejtettünk szót a specifikációra épülő programfejlesztési modell bemutatása során. Már szintén tárgyaltuk, és az olvasó is jól ismeri a formális módszerekkel történő specifikálást. A formáknak alávetett specifikáció a kettő között helyezkedik el, megpróbálva kiküszöbölni a másik két módszerben felmerülő problémákat. Ennek során jól és egyértelműen definiált formalizmusban írjuk le a feladatot, azonban a formalizmusban megmarad a beszélt nyelv bizonyos helyeken, noha ezt minimalizálni igyekszünk. Ugyanakkor a formalizmus nem feltétlen matematikai alapokon nyugszik, közelebb áll a hétköznapi gondolkodásmódhoz. A választott leírási mód értelemszerűen megfelel a követelmények leírása során használt leírásnak.

2.4.8. Tervezés

A tervezés során - módszertől függően - a következő kérdésekre adjuk meg a választ. (Az egyes szakaszokat az objektumelvű tervezésben használt terminológia szerint neveztük el.)

1. A statikus modell megalkotása.
    - Mi a rendszer szerkezete?
    - Milyen programegységekből épül fel a rendszer?
    - Mi az egységek feladata?
    - Mi az egységek közötti kapcsolat, reláció?
2. A dinamikus modell megalkotása.
    - Hogyan oldja meg a rendszer a problémát?
    - Milyen egységek működnek együtt a megoldás során?
    - Milyen üzenetek játszódnak le az együttműködő egységek között?
    - Mik a rendszer, az egységek állapotai?
    - Milyen események hatására valósulnak meg az állapotok közötti átmenetek?
3. A funkcionális modell megalkotása.
    - Milyen adatáramlások révén valósulnak meg a szolgáltatá sok?
    - Milyen leképezések játszanak szerepet az adatáramlások ban?
    - Mik az ajánlások az implementáció számára?
        - Implementációs stratégiára vonatkozó ajánlás.
        - Programozási nyelvre vonatkozó előírás, ajánlás.
        - Tesztelési stratégiára vonatkozó ajánlás.

A gyakorlatban a következő két tervezési módszer terjedt el:

1. procedurális (funkcionális) elvű és
2. objektumelvű.

1. A *procedurális tervezés* során a rendszer által megvalósítandó funkciókból, műveletekből indulunk ki, és ezek alapján bontjuk fel a rendszert kisebb összetevőkre, modulokra. Ezeket is a műveletek szerint bontjuk tovább. Ez a tervezési módszer rendszerint felülről lefelé halad, fokozatosan finomítva a tervet az egyes szinteken. Az ilyen irányú tervezést nevezik „felülről lefelé" tervezésnek. Ennek ellentettje az „alulról felfelé" haladó tervezés, amikor a legrészletesebb szint felől haladunk az egyre absztraktabb szintek felé. Ez úgy lehetséges, hogy egy meglevő művelet gyűjteményből indulunk ki. Ez érezhetően nehezebben összeegyeztethető a procedurális tervezés elvével.

A procedurális elvű megközelítés tipikus példája a *strukturális dekompozíció.* A strukturált tervezés eredményei:

- A program struktúradiagramja.
- Az input-output táblázat.
- A modulok definíciója.
- Az adatszótár.

2. Az *objektumelvű* tervezés esetén a rendszer funkciói helyett az adatokat állítjuk a tervezés középpontjába. A rendszer által használt adatok felelnek meg majd bizonyos értelemben az objektumoknak. A tervezés során az adatokat csoportosítjuk viselkedésük szerint, és az így kialakított osztályok, illetve egyedek közötti kapcsolatokat igyekszünk felderíteni, azok lehetséges állapotait azonosítjuk. Ezen tervezési módszer alapja az objektumelvű modellalkotás. Több olyan szoftvereszköz áll már rendelkezésre, amely nemcsak a tervezés folyamatát támogatja, hanem az elkészült terv alapján a programot, illetve annak vázát generálja.

A terv minőségének mutatói

A tervezés eredménye a *terv,* amelynek minősége nagyban meghatározza a projekt eredményességét, illetve a végső program minőségét. A továbbiakban a terv minőségét meghatározó jellemzőket tekintjük át.

Kohézió: egy programegységen belül a komponensek összetartozását fejezi ki. Mértéke az összetartozás szorosságának, erősségének a nagyságával jellemezhető.

A programrendszer terve akkor jó, ha az önálló programegységeken (modulokon) belül erős az összetartó erő.

- Laza kohézió:
  - Véletlenszerű kohézió.
  - Időbeli kohézió.
  - Logikai kohézió.
- Szoros kohézió:
  - Kommunikációs kohézió.
  - Funkcionális kohézió.

Összekapcsolódás (coupling): Az összekapcsolódás egy rendszeren belül azt fejezi ki, hogy a programegységek mennyire vannak egymással összekapcsolódva, mennyire függnek egymástól. Mértéke ennek a függő viszonynak az erősségével, nagyságának mértékével mérhető.

A programrendszer terve akkor jó, ha a programegységek egymástól függetlenek, vagy egymástól csak kevéssé függenek.

Erős az összekapcsolódás a cím szerinti adatátadás esetében, amikor a modulok egymás területeit használhatják szolgáltatásaiknak igénybevételénél.

Az adattípus, az adattípus osztály használata laza összekapcsolódást jelent.

Az öröklődés az összekapcsolódás szempontjából nem szimmetrikus eset. Azok az objektumosztályok, amelyek egy általános osztálytól attribútumokat, műveleteket vesznek át, nagyon szorosan összekapcsolódnak a szuper-osztállyal.

Módosíthatóság: A programegységek legyenek önállóak, függetlenek a rendszer más komponenseitől. Ez az elv azonban ellentmond az újrafelhasználhatóságnak.

Fejleszthetőség: A rendszerek nyitottsága ma általános követelmény. Szabványos csatlakozási felületeket kíván meg.

Bonyolultság: Ez pszichológiai bonyolultságot jelent, ami a rendszer áttekinthetőségének, megérthetőségének nehézségét fejezi ki.

2.4.9. Implementáció

Az implementáció során megválaszolandó kérdések a következők:

- *Hogyan ábrázoljuk az adatokat?* Ezt nevezzük *reprezentációnak.*

- *Hogyan valósítjuk meg a leképezéseket, eseményeket?* Ennek során el kell döntenünk, hogy:

  —milyen algoritmusokat használunk, és

  —milyen optimalizálásokat hajtunk végre.

A tervezéshez hasonlóan itt is absztrakciós szintek bevezetésével igyekszünk egyszerűsíteni a problémát. Az absztrakciós szintek rendszerint összhangban vannak a tervezéskor bevezetett absztrakciós szintekkel, igény szerint azokat finomítják tovább.

Az implementáció során a következő stratégiák terjedtek el:

- alulról felfelé (bottom-up) stratégia,

- felülről lefelé (top-down) stratégia,

- modul, alrendszer izolációs stratégia,
- objektumelvű modell alapján automatikus kódgenerálás.

Az „alulról felfelé" történő implementáció során a legalsó szintű részt készítjük el először, és innen haladunk felfelé, a magasabb szintekre. Ennek a módszernek előnye, hogy könnyű az elkészült részeket tesztelni, ugyanakkor a rendszer szerkezetének esetleges hibáit későn lehet csak észlelni. A másik módszer a „felülről lefelé" haladó implementáció. Ennek során a legfelső szintről indulunk ki, és onnan haladunk lefelé. Előnye, hogy hamar kiderülnek a szerkezeti hibák. Ugyanakkor tesztelni csak a teljes programot lehet, illetve a közbenső tesztelésekhez az alacsonyabb szinteket bizonyos értelemben szimulálni kell.

Az implementációt jelentősen felgyorsíthatja és biztonságosabbá teheti, ha már létező, kipróbált elemeket használunk fel. Ezek lehetnek eljárások vagy akár modulok. Ennek a koncepciónak egy továbbfejlesztett változata az objektumelvű megközelítésben használt öröklődés, illetve az aggregáció és a kompozíció. Ennek során egy ismert és helyes részt használunk fel újra úgy, hogy azt a szükséges tulajdonságokkal, műveletekkel kiegészítjük, egyes műveleteit pedig szükség esetén átdefiniáljuk.

Az implementáció folyamatát meghatározza, hogy milyen *programozási eszközrendszer, szoftverfejlesztési környezet* áll rendelkezésre. Egy jó környezet rendelkezik a következő jellemzőkkel:

- Az eszközök egységes rendszert alkotnak.
- Az eszközrendszer konstruktív.
- Tartalmaz rendszerkonstrukciót támogató eszközöket.
- Támogatja a csapatmunkát.

Az implementáció egyik alapvető kérdése az *implementációs stílus*. A jó programozási stílus néhány fontos eleme:

- absztrakció különböző szintjeinek alkalmazása;

- öröklődési technika használata, absztrakciós szintek hierarchikus rendszere;
- absztrakciós szintekre bontás osztályon belül (deklaráció + meg valósítás) ;
- korlátolt láthatóság;
- információ elrejtés (information hiding);
- információ beburkolás (encapsulation).

2.4.10. Verifikáció, validáció

Ebben a fázisban arra keressük a választ, hogy a rendszer eleget tesz-e a vele szemben támasztott elvárásoknak. Ez két nagy részre osztható:

Verifikáció: a specifikáció szerinti helyesség igazolása.

Validáció: annak ellenőrzése, hogy a rendszer teljesíti-e az előírt minőségi tulajdonságokat; ilyen tulajdonságok például a robusztusság, hatékonyság, bonyolultság, erőforrásigény.

Az ellenőrzés folyamatát nevezzük *tesztelésnek.* Ennek feladata a rendszer hibáinak, működési rendellenességeinek kimutatása, lokalizálása. A továbbiakban hibáról beszélünk, ha a rendszer nem a specifikáció szerint működik, azaz nem helyes (verifikáció); működési rendellenességről, ha a rendszer nem felel meg valamilyen előzetes követelménynek (validáció). Ennek megfelelően a tesztelés is két nagy fázisra osztható. Az egyik a verifikáció folyamatához, a másik a validációhoz kapcsolódik.

Nagy rendszerek esetén a tesztelés két szakaszát különböztethetjük meg.

- *Egységteszt:* Ennek során a rendszer elemeit, moduljait teszteljük külön-külön, egymástól függetlenül. A modulok által megvalósí tott szolgáltatások kerülnek ellenőrzésre.

- *Rendszerteszt:* Az egész rendszer együttes tesztje. Itt azt vizsgáljuk, hogy a (már tesztelt és önmagukban helyes) modulok miként működnek együtt, az együttműködés helyes eredményre vezet-e, a modulok eleget tesznek-e az interfészre vonatkozó előírásoknak.

Ez a két szakasz gyakran kiegészül közbenső szakaszokkal, amikor a rendszert több modulból álló alrendszerekre lehet bontani. Ekkor szokás alrendszereket is külön-külön tesztelni. Ez egyrészt egységteszt, hiszen egy adott szintű egységet tesztel; másrészt rendszerteszt, mert egységek együttműködését vizsgáljuk. Ilyen típusú tesztek több szinten is megjelenhetnek, az alrendszerek hierarchikus szerkezetének megfelelően.

A tesztelés módja szerint megkülönböztethetjük a

- *fekete doboz* tesztelést, amelynek során a tesztelendő program részt ismeretlennek tekintjük; és a

- *fehér doboz* tesztelést, amelynek során ismert a megfelelő prog ramrész.

Fekete doboz tesztelés esetén csak a hibák és működési rendellenességek felderítésére van lehetőségünk, a hibák helyének lokalizálására értelemszerűen nincs. Ebben az esetben a tesztadatok célszerű megválasztásához fel kell derítenünk a feladat eseteit (általános esetek és speciális esetek). Ki kell gyűjteni néhány adatot az általános esetekhez, és minden speciális esethez legalább egy tesztadatot kell készítenünk. (Speciális eset például egyszerű adatfeldolgozó programok esetén az üres fájl.)

Fehér doboz tesztelés esetén megfelelő tesztadatok mellett a hiba helyét is meghatározhatjuk. Ekkor is érdemes a fekete doboz tesztelés eseteit is használni, és ezeket kell úgy kiegészíteni, hogy a tesztelendő programrész minden lefutási szálán legalább egyszer végighaladjunk. (Tehát például egy elágazás összes ágán végig kell menni legalább egyszer.)

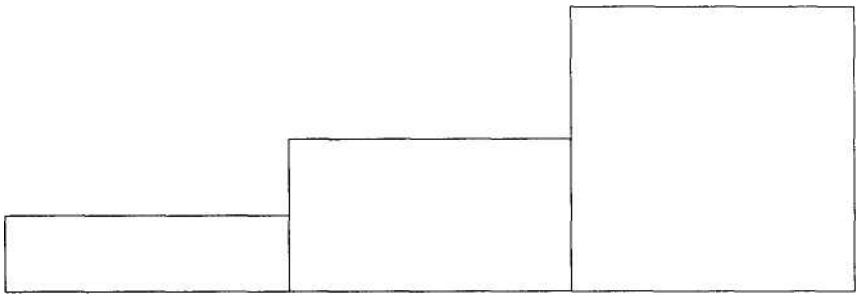
2.4.11. Rendszerkövetés és karbantartás (maintenance)

Ebbe a fázisba a kész, már próbaüzembe vagy működésbe helyezett rendszerrel kapcsolatos feladatok tartoznak. Ezeket két nagy osztályba sorolhatjuk:

- A szoftverrendszer üzembe állítása után szükségessé váló, szoft ver jellegű munkákat nevezzük *karbantartási munkáknak.*
- A *rendszerkövetés* munkája a felhasználókkal való kapcsolattar tás menedzsment jellegű, dokumentációs feladatainak ellátása.

A karbantartási munkák további csoportosítása:

- A rendszerben a használat során felmerülő hibák, az úgynevezett *rejtett hibák* kijavítása. Ezek a hibák a programkészítés külön böző fázisaiban fordulhattak elő. Kijavításuk költsége ettől is függ (2.20. ábra).
- A rendszer új üzemeltetési környezetbe helyezése, az úgynevezett *adaptációs munkák.* Ez lehet:



2.20. ábra. Rejtett hibák javítási költségének aránya a fázisok függvényében

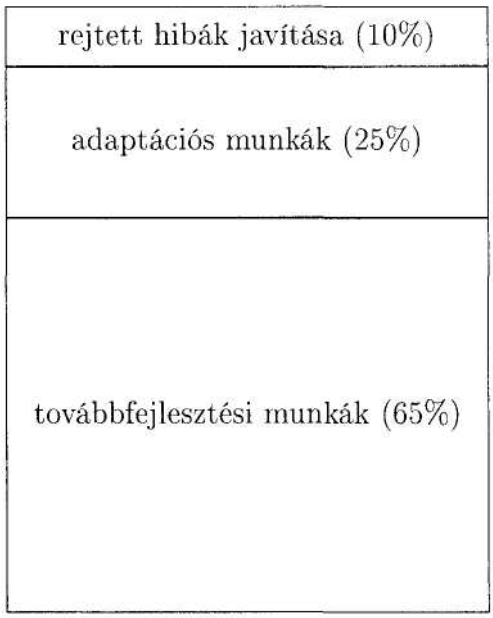
~ új hardverkörnyezetbe helyezés vagy

—új szoftverkörnyezetbe helyezés.

• A rendszer új igényeknek megfelelő módosításai, az úgynevezett *továbbfejlesztési munkák,* azaz:

- új szolgáltatások bevezetése,
- új adatbázisra történő kiterjesztések,
- hálózati szolgáltatásokra történő kiterjesztések,
- minőségi mutatók javítása.

A karbantartás három nagy feladatának költségben kifejezett arányát nehéz meghatározni. Különböző nagy cégek tapasztalatai alapján a 2.21. ábrán látható hozzávetőleges arányok találhatók meg a szakmai irodalomban.



2.21. ábra. A karbantartási feladatok költségben kifejezett aránya

Tevékenység	Költségszázalék
Rejtett hibák kijavítása	10%
Minőségi mutatók javítása	5%
Szolgáltatások módosítása, új szolgáltatások bevezetése	40%
Új adatformákhoz, új adatbázishoz igazítás	20%
Új környezetbe helyezés	25%

2.2. táblázat. A karbantartás feladatai

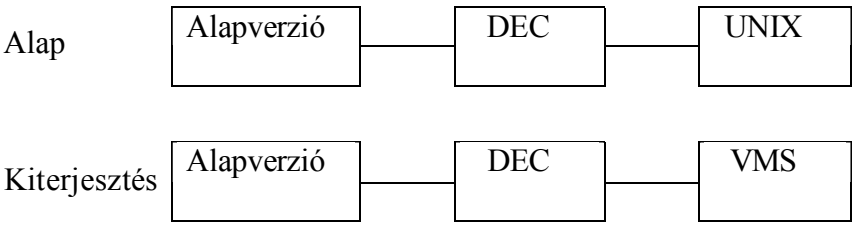
A karbantartási feladatok költségben mérhető tipikus százalékos arányát foglalja össze a 2.2. táblázat, finomabb bontásban. Természetesen ezek az adatok (az előzőekhez hasonlóan) csak iránymutató értékek, amelyeket nagy rendszerek kidolgozásának tapasztalatai alapján becsültek meg.

A fenti adatok súlyát kiemeli, ha megadjuk azt a százalékos adatot is, hogy a rendszer megvalósításának, kidolgozásának és a rendszer utóéletének (rendszerkövetés és karbantartás) során elvégzendő feladatok aránya a következő:

- megvalósítás: 30-35%,
- karbantartás: 65-70%.

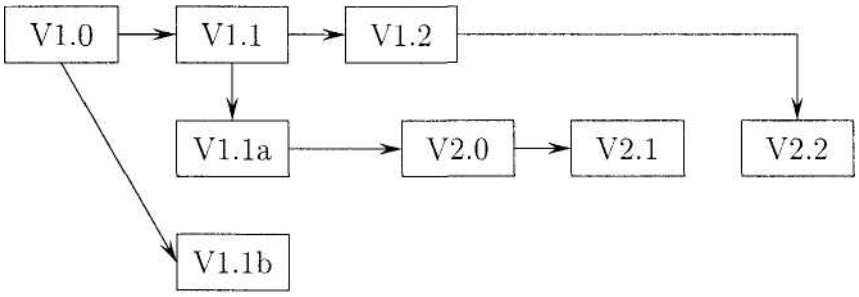
A rendszerkövetési munkákat is lehet osztályozni:

- *Konfigurációk nyilvántartási feladatai:* A tervezés során megha tározzuk az alapkonfigurációt és annak ajánlott szabványos ki terjesztéseit. Például:



A konfigurációkat nyilván kell tartani. A változásokat a verziókon és azok dokumentációján végre kell hajtani.

Verziók menedzselési feladatai: A hibajavítások és a fejlesztések során a rendszernek új verziói jönnek létre. Ez nemcsak a végrehajtandó programot jelenti, hanem az installációs programot, adatfájlokat stb. A tervezés során megtervezzük a verziók nyilvántartási rendszerét.



A rendszer üzembe állításakor létrehozzuk, illetve felújítjuk a felhasználók verzióhasználati nyilvántartását.

Dokumentáció menedzselési feladatai:

- elektronikus dokumentáció,
- kiadvány formában megjelent dokumentáció.

A rendszer generálásának feladatai: a kívánt rendszer összeállítása komponensekből.

2.4.12. Dokumentáció

Az előzőekben szó volt arról, hogy az elkészült szoftvertermék egyik lényeges eleme a *dokumentáció.* Ennek készítése ideális esetben a rendszerkészítés megfelelő fázisaiban történik. A továbbiakban a dokumentáció fajtáival, illetve a vele szemben támasztott elvárásokkal foglalkozunk.

Egy nagy méretű program önmagában nem tekinthető szoftvertermé-knek dokumentáció nélkül. Egy jó dokumentáció választ ad a programmal kapcsolatos lényeges kérdésekre:

1. Mi a kitűzött feladat, amit a programnak meg kell oldania? (Mire használható a szoftver?)
2. Milyen környezetben használható a program?
3. Milyen korlátozások között futtatható a program? Az esetleges speciális igények leírása.
4. Mi a program előélete? Verziószám, eltérések, illetve fejlesztések a korábbi verziókhoz képest. Mennyire kompatibilis a program az előző verziókkal?
5. Hogyan installálható a program?
6. Hogyan használható a program?
7. Kik készítették a programot? Hogy lehet kapcsolatba lépni ve lük? (A termékkel kapcsolatos további információk beszerzése, az esetleges hibák, rendellenességek jelentése.)
8. Milyen modulokból épül fel a program? Mi a modulszerkezete?
9. Milyen osztályok fordulnak elő, mi ezek kapcsolata? (Osztályhi erarchia, statikus diagramok.)
10. Milyen a rendszer dinamikus viselkedése? (Állapotdiagramok, szekvenciadiagramok, aktivációs diagramok, együttműködési di agramok.)
11. Az egyes osztályok miként lettek implementálva? A felhasznált fontosabb adatszerkezetek leírása. A felhasznált (sablon) osz tályok felhasználási módja. Az absztrakt típusok leképezése az adott nyelvre, fejlesztő eszközre.

12. Teszteléssel kapcsolatos információk. Milyen esetekre, hogyan és milyen környezetben került a program tesztelésre? Milyen eredményekkel?

A dokumentáció első hét pontra vonatkozó részét szokás *felhasználói leírásnak* nevezni, a többit pedig *fejlesztői leírásnak.* Sokszor ez a két rész fizikailag is elkülönül, mert rendszerint a felhasználó csak a felhasználói leírást kapja meg.

A dokumentációt nem helyettesíti a programokban ma már szokványos on-line help (súgó).

A dokumentációnak a fenti kérdésekre szabatosan, áttekinthetően kell választ adnia. Ennek érdekében azt megfelelően strukturálni kell, azaz hierarchikus egységekre: fejezetekre, a fejezeteket pedig pontokra, ezeket igény szerint további alpontokra kell bontani. Az egyes egységeket kifejező címmel kell ellátni, hogy az olvasó gyorsan megtalálhassa a számára szükséges részt.

A szabatos leírás szükségessé teszi a megoldott feladattal, illetve a programmal kapcsolatos fogalmak azonosítását, ezek pontos definícióját és későbbi használatát a dokumentációban. Nagyon sokszor találkozhatunk azzal, hogy valamit többször is hosszasan (fél mondat vagy akár mondatok) körülírnak ahelyett, hogy elneveznék, definiálnák a fogalmat, és később a nevével hivatkoznának rá. A definíciónak nem feltétlen kell formálisnak lennie, de törekedni kell a pontos leírásra. Formális definíció mellől nem hiányozhat az informális leírás sem. Csak feltétlen szükséges fogalmakat vezessünk be, mert az olvasó elveszhet a túl sok definícióban, ami olvashatatlanná teszi a dokumentációt! A bevezetett fogalom neve utaljon annak jelentésére! (Lehetőség szerint a dokumentációt lássuk el tárgymutatóval is, amely tartalmazza az egyes fogalmak definíciójának helyét a dokumentumban.)

A leírást gyakran egyszerűsíthetjük, illetve kifejező erejét növelhetjük ábrák, programképek beillesztésével, ezért ezek használata ajánlatos. Különösen hasznos lehet, ha az ábrákon alkalmazott jelölés egy ismert szabványt követ. (L.: UML!)

A felhasználói dokumentáció készítésekor figyelembe kell venni azt is, hogy kiknek készül az, és annak megfelelő szintű leírást kell készí-

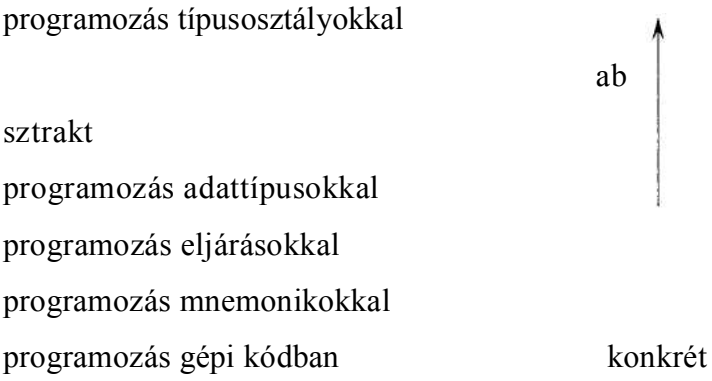
teni. Például egy irodai programcsomag esetében nem ugyanazokat a számítástechnikai ismereteket tételezhetjük fel az olvasóról, mint egy fejlesztőrendszer vagy egy matematikai programcsomag esetén. Ehhez kell igazodnia a dokumentáció jelölésrendszerének és szóhasználatának is, ugyanis akár a túl sok ismeretet feltételező, akár a túl kevés ismeretre építő dokumentáció zavaró, nehezen követhető lehet. A fejlesztői leírás esetében ez nem probléma, hiszen annak készítésekor feltehetjük, hogy az olvasó megfelelő ismeretekkel rendelkezik, csak az esetleges specialitásokra kell külön figyelmet fordítani.

Bonyolultabb rendszerek esetén hasznos lehet, ha a felhasználói leírás tartalmaz egy-két egyszerű esettanulmányt a program használatáról (tutorial). Ennek segítségével elsajátítható a rendszer alapvető funkcióinak használata, és erre építve könnyebben megismerhetők további szolgáltatások is.

2.5. Programfejlesztés modellalkotással

Mielőtt rátérnénk ennek a programfejlesztési modellnek a bemutatására, meg kell vizsgálnunk a programozás és a modellalkotás kapcsolatát. A programozás során a problémát számítástechnikai térben, úgynevezett megoldástérben szemléljük. A modellalkotás folyamata során a problémát valamilyen absztrakt térben írjuk fel és oldjuk meg. A programfejlesztés során ezt a két teret kell egymáshoz közelítenünk. Erre szolgálnak a programozásban használt a különböző absztrakciós szintek. A programozás technikája az elmúlt 50 éves gyakorlat során a konkrét megoldásoktól az absztrakt megoldások irányába haladt. Ennek az útnak a legfontosabb állomásait, szintjeit és irányát foglalja össze a 2.22. ábra. Ezek a szintek lényegében megfelelnek a számítástechnika története során megjelent programnyelvek absztrakciós szintjeinek. (Gépi kód -> Assembly -> FORTRAN, Algol -> Pascal, C -> C++, Java.)

Kezdetben a program megírása jelentette az adott probléma megoldását. Nagyobb programok esetén ez az út nem volt járható. Általá-



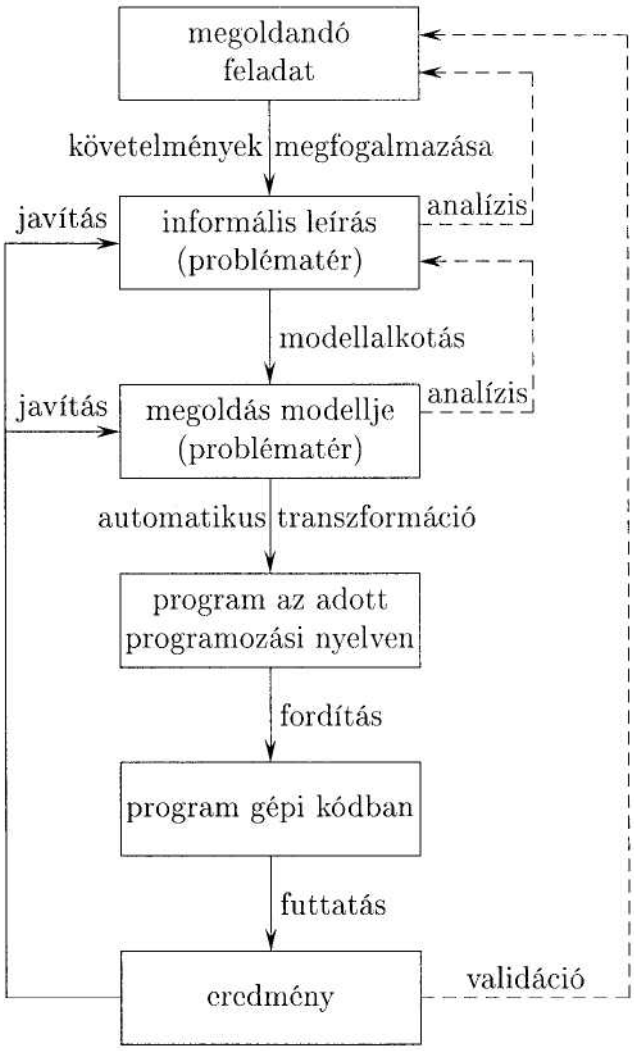
2.22. ábra. Absztrakciós szintek a programozásban

nosan kialakult emberi gyakorlat a probléma megoldásában a *modellalkotás*. Mielőtt továbbmennénk, soroljuk fel a modell néhány fontos jellemzőjét!

- A modell a valóság egy leegyszerűsített megvalósítása.
- A modell alapján a problémát jobban meg tudjuk érteni.
- A modell alapján a probléma megoldásainak tulajdonságait vizsgálni tudjuk.
- A modell alapján megalapozott, dokumentált döntést tudunk hozni a konkrét megvalósításról.

Ha összehasonlítjuk a programozást és a modellalkotást, arra az eredményre jutunk, hogy a programozás során a problémát a számítástechnikai térben (megoldástérben) oldjuk meg, a modellalkotás során pedig valamilyen absztrakt térben tesszük ugyanezt. Nagy rendszerek fejlesztésekor a tervezés során létrehozunk egy modellt, amelyben leírjuk a tervet, azaz a problémateret leképezzük egy absztrakt térre. Az implementáció során az absztrakt teret képezzük le a megoldástérre. A terek közötti transzformációk bonyolítják a folyamatot, és hibákat okozhatnak.

Az *objektumelvű modellalkotás* alapja, hogy a probléma megoldása a valós világ objektumainak a terében, a *problématérben* történik. Ez



2.23. ábra. Programfejlesztés objektumelvű modellalkotással

az egyik lényeges eleme a modellalkotással történő programfejlesztésnek is. Ekkor a tervezés során elvileg nem kell teret váltanunk. Ez rendkívüli módon leegyszerűsíti a modellezés folyamatát, és csökkenti az elkövethető hibák számát. Léteznek előre definiált, szabványos jelölés-

rendszerek (pl.: UML), amelyekkel a megoldás objektumelvű modelljét leírhatjuk. Ez a szabványos leírás veszi át a megelőző programfejlesztési modellekben szereplő formális specifikáció, illetve rendszerterv szerepét (2.23. ábra).

A modellalkotás néhány alapelvét (ökölszabályát) soroljuk fel a következőkben.

- A modellt tudni kell jól megválasztani. (Ezt a gyakorlat, tapasz talat alakítja ki.)
- Egy modell nem modell. (Több alternatívát kell megvizsgálni, mielőtt kiválasztanánk a használandót.)
- A modellnek pontosan kell tükröznie a valóságot.

Mi a modellalkotás során a továbbiakban az úgynevezett UML (Unified Modeling Language) nyelvet fogjuk használni.

Ma már rendelkezésre állnak olyan eszközök, amelyek ebből a leírásból automatikusan előállítják a program kódját, illetve annak vázát (osztályok, műveletek deklarációi) egy adott nyelven. Azonban ilyen eszköz hiányában is egyszerű, könnyen végrehajtható átalakításról van szó. Ennek megfelelően a kész kódot nem is kell változtatni, a javítások egy szinttel feljebb tolódnak. Továbbá optimális esetben nincs szükség a programkód és a megoldás modelljének összevetésére sem, így az ennek megfelelő elemzés (analízis) is elmaradhat.

2.6. A programozási technológia fejlődési iránya

Két fontos célt tűz ki maga elé a programozási technológia. Egyrészt a követelmények informális leírásából és a specifikációból kiindulva próbál egy egységes követelményleírást létrehozni; másrészt a tervezést, kódolást és verifikációt próbálja egységesen kezelni a modellalkotás és az automatikus objektumelvű programozás segítségével (2.24. ábra). Mi ez utóbbival foglalkozunk részletesen.

követelmények leírása, specifikáció } egységes követelményleírás

tervezés, kódolás, verifikáció } modellalkotás + automatikus objektumelvű programozás

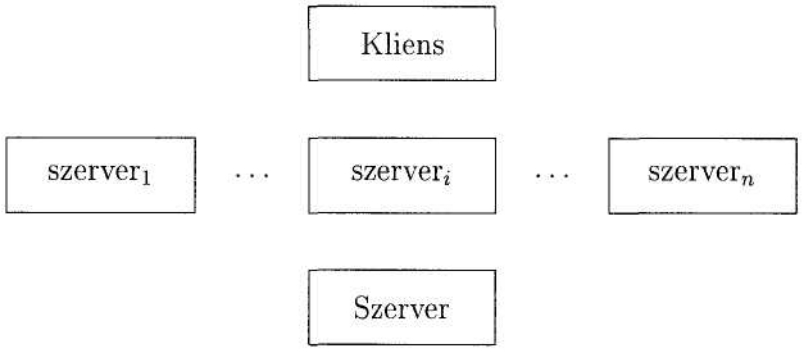
2.24. ábra. A programozási technológia célkitűzései

A jövőre nézve azt mondhatjuk, hogy a 21. századi szoftvertechnológiát várhatóan a következők jellemzik majd:

- nyílt,
- szabványokra épülő,
- objektumelvű,
- komponens alapú,
- teljes életciklust lefedő,
- elosztott rendszerű,
- távmunkában felhasználható,
- minőségbiztosítást támogató.

A komponens alapú rendszerekben a komponens fogalmát a következőképpen határozzuk meg: egy önálló részfeladat megoldására szolgáló fizikailag létező része a rendszernek, amely más egységgel helyettesíthető, amennyiben az új komponens megvalósítja a csatlakoztatáshoz előírt felületet.

Az elosztott rendszerű alkalmazások alapjában kliens-szerver rendszerek, illetve a WEB alapú alkalmazások. A kliens-szerver rendszerekben a kliens a felhasználói kommunikációt látja el, míg a szerver a megosztott erőforrásokat használja, és ezekkel nyújt szolgáltatást. A WEB alapú alkalmazásokban a kliens és a szerver közötti kapcsolat több szerveren keresztül jöhet létre (2.25. ábra).



2.25. ábra. Kliens-szerver kapcsolat WEB alapú alkalmazásokban

2.7. Unified Modeling Language (UML)

Az objektumelvű modellezés kialakulása maga után vonta a megfelelő eszközök kifejlődését is. Ezek közül kiemelt szerep jut a modellezés folyamatának és az eredményének leírását támogató rendszereknek, jelöléseknek, illetve szabványoknak. Mi az objektumelvű modellalkotás során az úgynevezett UML (Unified Modeling Language) nyelvet fogjuk használni, ezért vázlatosan áttekintjük ennek főbb jellemzőit és történetét.

Az UML egy grafikus nyelv, amelyben lehetőségünk van a probléma

- specifikációjára,
- megoldására,
- a megoldás dokumentálására.

Az UML-hez léteznek az alábbi transzformációs eszközök:

- UML —> programozási nyelv, ahol a nyelv lehet például:
  - JAVA,
  - C++,

- Smalltalk,
- Visual Basic,
- 4 GLs.

• UML —> közbeeső rétegek, amelyek lehetnek:

- EJB,
- CORBA (OMG),
- DNA (Microsoft).

Az UML objektumelvű modellezéshez használható. Az objektumelvű modellezés alapjait az 1980-as években megjelent objektumelvű módszerek, illetve objektum modellezési technikák alkotják. Maga az UML az 1990-es években alakult ki, és 2000-ben az ISO szabványa lett. Az UML fokozatos fejlődésen ment keresztül, amint azt a 2.3. táblázat mutatja.

Soroljuk fel az UML néhány jellemzőjét! Ezekkel részletesebben a 4. fejezetben foglalkozunk majd.

1. Az UML-ben létrehozott modell grafikus szemléltetést nyújt.
2. A nyelv lehetővé teszi modellek megalkotását különböző néze tekből.

Szabvány	Dátum
UML 0.8	1995.
UML 1.0	1997. jan.
UML 1.1	1997. szept.
UML 1.2	1998.
UML 1.3	1999.

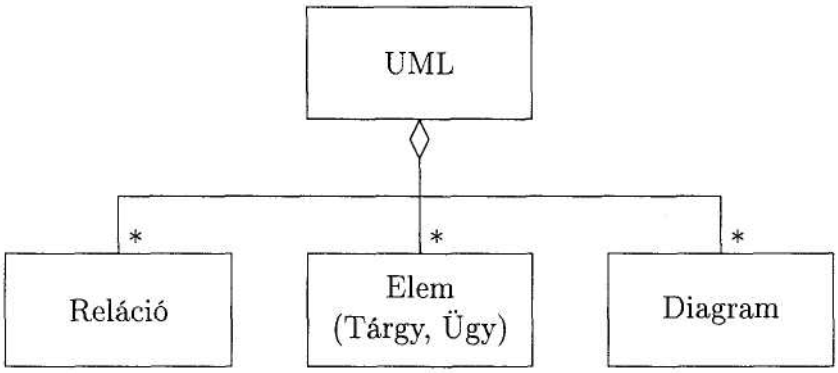
2.3. táblázat. UML szabványok

3. Alkalmas a probléma megoldásának pontos leírására.
4. Alkalmas a probléma megoldásának dokumentálására, ezen be lül:
   - a projektterv dokumentálására,
   - a szoftverkészítés fázisainak (követelmények, tervezés,...) dokumentálására,
   - a prototípus dokumentálására.
5. Az UML konstrukciós eszköz.
6. Nemcsak szoftvermodellek leírására alkalmas, de munkafolyama tok, szervezetek, különböző üzleti tevékenységek stb. leírására is.

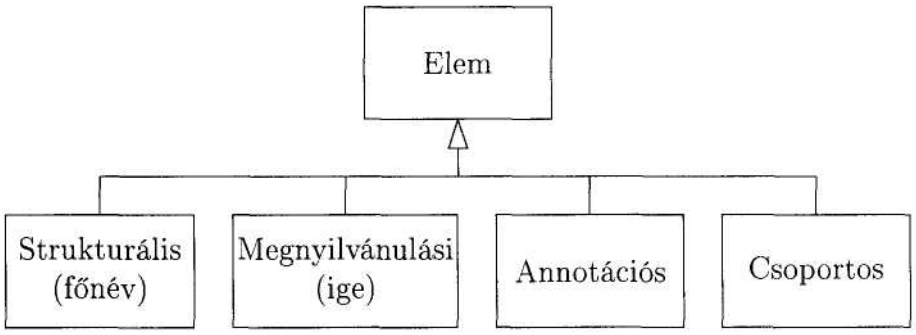
2.7.1. Az UML építőkövei

A következőkben röviden áttekintjük az UML alkotóelemeit. A leíráshoz felhasználjuk kiegészítésként az UML jelölésrendszerét is, ezzel a későbbi fejezetekben ismerkedhet meg az olvasó.

Egy UML leírás *relációkból, elemekből* (tárgy, ügy) és *diagramokból* áll (2.26. ábra).



2.26. ábra. Az UML építőkövei



2.27. ábra. Az elemek fajtái

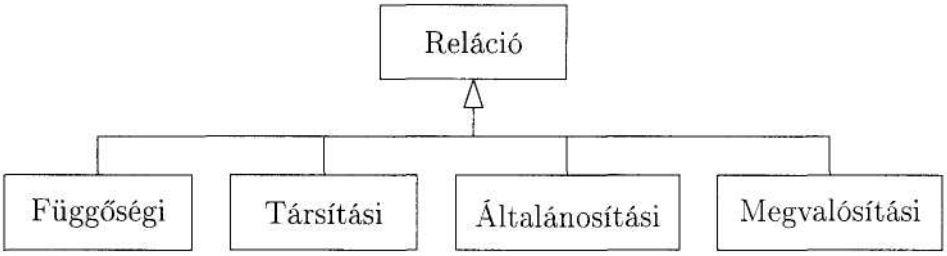
Az elemek lehetnek (2.27. ábra):

- Strukturális elemek:
  - Objektum.
  - Osztály.
  - Használati eset.

- Megnyilvánulási elemek:
  - Művelet végzése.
  - Interakció (üzenetküldés).
  - Állapotautomata.

Csoportos elemek:

- Alrendszer
- Package.
- ...



2.28. ábra. A relációk fajtái

Annotációs elemek:

- Kiegészítés.
- Megjegyzés.
- Megszorítás.

A relációkat 4 csoportba oszthatjuk (2.28. ábra):

• Függőségi reláció: szemantikai összefüggés. Például:

Társítási reláció: strukturális, szerkezeti összefüggés. Például:

Általánosítási reláció: általános és speciális kapcsolata. Például:

Megvalósítási reláció: szemantikai kapcsolat a fogalom és annak megvalósítója között. Például:

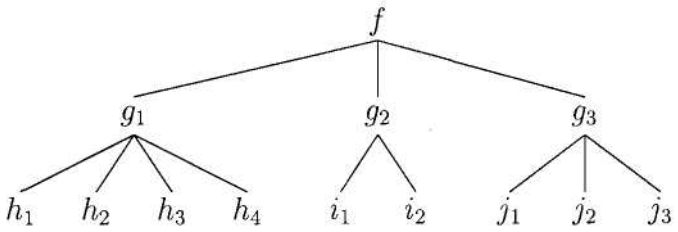
A diagramok az elemeket és a köztük fennálló relációkat szabványos jelöléssel leíró ábrák. Ezeket részletesen felsoroljuk és osztályozzuk a 4. fejezet elején.

3. Az objektumelvű programozás kialakulása

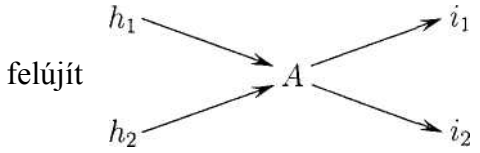
Az előzőekben már említettük, hogy két tervezési módszert szoktak megkülönböztetni a programozási technológiában: a procedurálist és az objektumelvűt. Történetileg a procedurális módszer alakult ki előbb; hiányosságainak, hibáinak kiküszöbölésére fejlesztették ki később az objektumelvű tervezést, programozást. Ezért az objektumelvű programozás bemutatásakor kell néhány szót ejteni a procedurális módszerről is.

A procedurális megközelítésnek az a lényege, hogy egy leképezésnek tekintjük a megoldandó feladatot, és ennek kiszámítási szabályát kívánjuk előállítani úgy, hogy szintenként egyszerűbb és egyszerűbb leképezésekre bontjuk azt (3.1. ábra).

Ennek során olyan leképezések léphetnek fel, amelyek közös erőforrást használnak, például egy közös táblázatot újítanak fel, illetve kérdeznek le (3.2. ábra).

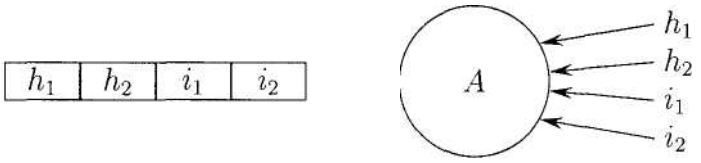


3.1. ábra. A feladat szintenkénti finomítása procedurális tervezés esetén



felújít lekérdez

3.2. ábra. Közös táblázatot (A) felújító, illetve lekérdező műveletek



3.3. ábra. Az eljárások modulba foglalása, szabványos felület bevezetése

Ha ezeknek a leképezéseknek az eljárásait egy programegységbe, modulba foglaljuk össze, akkor csökkenteni tudjuk a rejtett hibák előfordulásának valószínűségét. Erre szokták azt mondani, hogy a közös erőforrást használó eljárások között erős kohézió áll fenn. Ha az erőforráshoz való hozzáférést szabványos eljárásokkal oldjuk meg, a hiba valószínűségét tovább csökkenthetjük. Ily módon a modul más modulokkal laza kapcsolatba kerül.

A fenti példánk esetében a kohézió miatt egy modulba kerülnének a h_1, h_2, i_1, i_2 eljárások, amelyeket azután szabványos hozzáférési felülettel láthatnánk el (3.3. ábra).

Ez a gyakorlat vezetett el az absztrakt adattípus fogalmához és azoknak a programozási nyelveknek a megszületéséhez, amelyek a típusalkotás lehetőségét a programozó kezébe adták. Az így bevezetett adatabsztrakciót és absztrakt adattípusokat kiegészítve a típusöröklődés fogalmával eljutunk az objektumelvű programozás hagyományos értelmű, klasszikus meghatározásához:

objektumelvű programozás = adatabsztrakció +
absztrakt adattípus +
típusöröklődés.

Ennek alapján objektumelvű programozási nyelv az, amely a következő típusokat támogatja:

```
    <típus>  ::=  <egyszerű> | <összetett> ;
<egyszerű>   ::=  integer | real | boolean |... ;
<összetett>  ::=  vector | array | record |...|
                  <absztrakt adattípus> |
                  <öröklődéssel származtatott típus> .
```

Az objektumelvű programozás klasszikus meghatározásában megjelent az absztrakció fogalma, ezért elengedhetetlen, hogy annak fogalmát tisztázzuk.

3.1. Absztrakció

Absztrakción a programozás adott szintjén a megoldás szempontjából lényegtelen részletek elhanyagolását értjük. Két fő formáját különböztetjük meg:

1. *Paraméteres absztrakció:* a formális paraméterekhez rendelhető aktuális paraméterek konkrét tulajdonságainak elhanyagolása.

2. *Specifikáció szerinti absztrakció:* az adat konkrét ábrázolásának (reprezentáció) és a rajta értelmezett műveletek konkrét megadásának (implementáció) elhanyagolása.

A specifikáció szerinti absztrakción belül további formákat különböztetünk meg:

2.1. *Applikatív procedurális absztrakció:* Választunk egy absztrakt rendszert és az azon értelmezett műveleteket. Ezt az absztrakt rendszert használjuk fel az adat ábrázolására, és ebben a rendszerben felírjuk az adathoz tartozó műveletek algoritmusait (procedurális implementáció).

2.2. *Külső felület szerinti absztrakció:* Egy absztrakt rendszerben ábrázoljuk az adatokat. A műveletek algoritmusait nem adjuk meg, csak a bemenő adatok halmazát és az egymáshoz rendelt (bemenő adatok, eredmény adatok) párok halmazait. Ezt megtehetjük például a jól ismert elő- és utófeltételes forma segítségével.

2.3. *Axiomatikus absztrakció:* Nincs ábrázolás és nincs megvalósítás, a program jelentését axiómákkal adjuk meg. Csak olyan axiómákat kell megadni, amelyek a művelet jelentését egyértelműen meghatározzák.

3.2. Adattípus

A következőkben a másik felhasznált fogalommal, az absztrakt adattípussal foglalkozunk. Definiáljuk az egyszerű és összetett adattípus fogalmát, és ennek során az axiomatikus absztrakciót fogjuk használni. Megadjuk még az adattípusok egy komplex leírási módját is.

3.2.1. Egyszerű adattípus

Egyszerű adattípusnak nevezzük az *(A, F)* párt, ahol *A* az adatok halmaza, *T* pedig a műveletek véges halmaza. A műveletekre a következő megkötéseket tesszük:

$$\forall f \in \mathcal{F} \;:\; f : \mathcal{A}^n \to \mathcal{A} \quad (n = 0, 1, \ldots)$$
$$\exists f_0 \in \mathcal{F} \;:\; f_0 :\to \mathcal{A}\,,$$

azaz minden művelet az adatokon értelmezett, és adatot állít elő, valamint létezik (legalább) egy adatot előállító művelet.

Az adatok halmazának konstruktívnak kell lennie, azaz minden elemét elő kell tudnunk állítani véges vagy megszámlálható számú művelet felhasználásával, vagyis:

$$\forall a \in \mathcal{A} : \exists f_{i_1}, \ldots, f_{i_k} \in \mathcal{F} : a = f_{i_k}\Big(\ldots(f_{i_1}(f_0))\ldots\Big)$$

A konstrukciós műveletek halmaza, $\mathcal{F}_c \subset \mathcal{F}$ az a legkevesebb számú művelet, amelynek segítségével $\mathcal{A}$ megkonstruálható, azaz összes eleme előállítható.

Tekintsük például a nemnegatív egész számok típusát ($\mathcal{N}$)! Az adatok halmaza a természetes számok és a nulla, a műveletek a nulla előállítása, a rákövetkező és két szám összege. Azaz: $\mathcal{A} = \mathbb{N}_0$, és $\mathcal{F} = \{zero, succ, add\}$. Összefoglalva:

$$\mathcal{N} = (\mathbb{N}_0, \{zero, succ, add\}) ,$$
$$zero :\rightarrow \mathbb{N}_0 ,$$
$$succ : \mathbb{N}_0 \rightarrow \mathbb{N}_0 ,$$
$$add : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0 .$$

Nyilvánvaló, hogy $\mathcal{F}_c = \{zero, succ\}$. Az axiómák felírását az olvasóra hagyjuk. Például az egyik axióma lehet:

$$\forall n, m \in \mathbb{N}_0 : add(succ(n), m) = succ(add(n, m)) .$$

3.2.2. Összetett adattípus

Összetett adattípusnak nevezzük az $(\mathcal{A}, \mathcal{F})$ rendezett párt, amelyben $\mathcal{A} = \{A_0, A_1, \ldots, A_m\}$ és A_0 a típusobjektumok halmaza, $A_1, \ldots, A_m$ paraméterhalmazok; továbbá $\mathcal{F} = \{f_0, f_1, \ldots, f_n\}$ a műveletek véges halmaza. Az egyszerű adattípushoz hasonlóan A_0 konstruktív, és minden művelet $\mathcal{A}$-beli halmazokon értelmezett, és oda tartozó értéket vesz fel, azaz:

$$\forall f \in \mathcal{F} : f : A_{i_1} \times \ldots \times A_{i_k} \rightarrow A_j, \quad \text{ahol:}$$
$$A_{i_1}, \ldots, A_{i_k}, A_j \in \mathcal{A} \wedge \exists x \in [1..k] : i_x = 0 \vee k = 0 .$$

$\mathcal{F}_c \subset \mathcal{F}$ a konstrukciós műveletek, röviden a *konstruktorok* halmaza, a többi $\mathcal{F}_s = \mathcal{F} \setminus \mathcal{F}_c$ művelet alkotja a *szelektorok* halmazát.

Tekintsük példaként a kupacokat! A kupac paramétere a benne elhelyezendő elemek típusa. Így:

$$Kupac = (\{Kupac, Elem\}, \{üres, betesz, kivesz, csúcs\}) ,$$
$$üres :\rightarrow Kupac ,$$
$$betesz : Kupac \times Elem \rightarrow Kupac ,$$
$$kivesz : Kupac \rightarrow Kupac ,$$
$$csúcs : Kupac \rightarrow Elem .$$

A műveletek jelentését megadó axiómák felírása az olvasó feladata.

Ha ezt megtesszük, akkor megadjuk az adattípus szemantikáját is. Ezzel lesz teljes a specifikáció, azaz $SPEC = (\mathcal{A}, \mathcal{F}, \mathcal{E})$, ahol $\mathcal{E}$ a szemantika leírása a választott absztrakciós módszerek egyikével.

3.2.3. Típusosztály

Adattípusok komplex leírására szolgál a típusosztályok megadására bevezetett következő módszer. Ez egy olyan komplex leírása a típusnak, amely az adott adattípus absztrakt leírását (PAR + EXP) és konkrét leírását (IMP + BODY) is szolgáltatja. Ennek alapján típusosztálynak nevezünk egy olyan négyest, amelynek első tagja megadja a paramétereket, második tagja az adathalmazt és a rajta értelmezett műveleteket definiálja, harmadik tagja meghatározza az átvett szolgáltatásokat, negyedik tagja pedig tartalmazza a megvalósítást. Pontosabban:

Típusosztály=(PAR, EXP, IMP, BODY), ahol:

PAR = < a paraméterek tulajdonságainak leírása >;

EXP = < a típusobjektumok halmazának és a rajta értelmezett műveleteknek a neve, szintaxisa és szemantikája, valamint a korlátozások leírása >;

IMP = < a típusobjektumok ábrázolásához más osztályból átvett szolgáltatások leírása >;

BODY = < a típusosztály ábrázolása, megvalósítása >.

A fenti négyest többféleképpen is meg lehet adni, a formális specifikációtól kezdve az informális leírásig. A gyakorlatban, célszerűségi okokból, a formáknak alávetett leírás terjedt el. Ennek egy lehetséges formája:

paraméter = < átvett aktuális típusok nevei > +

típusok: < formális paraméterek nevei > ;

forma: < műveletek formái > ;

jelentés: < műveletek jelentésének leírása > ;

export =

típus: < típushalmaz neve > ;

forma: ... ;

jelentés: ... ;

korlátozás: < típushalmaz > ;

import =

típusok: < ábrázoláshoz importált típusok nevei > ;

forma: ... ; jelentés: ... ;

body =

forma: rep: < az ábrázolás formája > ;

jelentés: < jelentés konkrét leírása > ;

korlátozás: < konkrét típushalmaz > .

3.3. Típusöröklődés

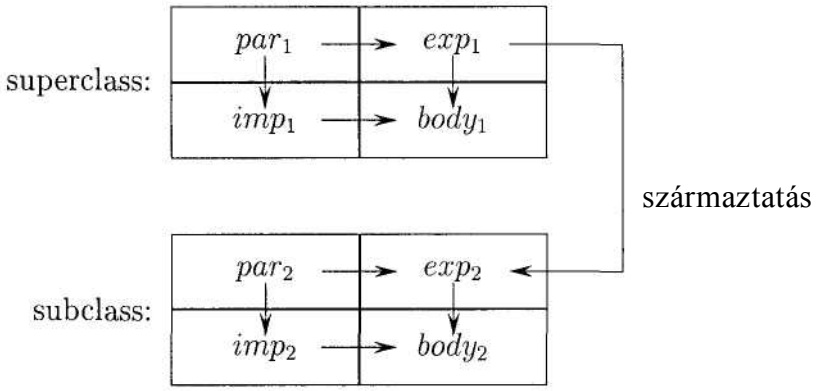
Az objektumelvű programozás klasszikus meghatározásában szereplő harmadik fogalom a típusöröklődés. Most ezzel foglalkozunk részletesebben. Két fő formája ismert:

- a specializáció és
- az újrafelhasználás.

Mindkét esetben egy létező típusosztályból, neve: *superclass,* hozunk létre, *származtatunk* egy új típusosztályt, neve: *subclass.* A két származtatás jellemzőit tekintjük át a következőkben.

3.3.1. Típusöröklődés specializációval

A superclass-ból a subclass-nak azt a származtatását, amely a következő tulajdonságokkal rendelkezik, specializációval történő öröklődésnek nevezzük (3.4. ábra):



3.4. ábra. Típusöröklődés specializációval. Adott a superclass: *(par_1, exp_1, imp_1, $body_1$),* amelyből származtatjuk a subclass-t: *(par_2, exp_2, imp_2, $body_2$)*- Az ábrában a nyilak az információ áramlásának irányát mutatják. Például a par_1 részben leírtak az exp_1 részben leírtak részét képezik.

1. A subclass átveszi a superclass összes *absztrakt tulajdonságát,* és azokat az export részben használja fel saját szolgáltatásaként.

2. A tulajdonságok átvételekor a típushalmaz, a paraméterhalmazok és a műveletek *nevei átdefiniálódhatnak, de jelentésük nem változhat meg.*

3. A subclass típushalmazának a superclass típushalmaza felel meg.

4. A superclass átdefiniált nevekkel történő leírása a subclass absztrakt leírásának részét képezi, ezért a subclass leírása:

   a. új szolgáltatásokkal bővülhet,

   b. új paraméterek definiálására is sor kerülhet,

   c. az import része módosulhat,

   d. új ábrázolási forma és új megvalósítás léphet fel.

Az elmondottak alapján: ha C_2 típusosztály C_1 specializációja és $a \in C_2$, akkor $a \in C_1$.

A specializáció következménye a

- polimorfizmus és a
- dinamikus összekapcsolás.

Polimorfizmus: a specializáció miatt minden változónak két típusa van:

1. a *statikus típus,* amelyet a deklaráció során nyer el;

2. a *dinamikus típus,* amely a deklaráció pillanatában megegyezik a statikus típussal, de a program végrehajtása során értékadáskor megváltozhat, ha egy superclass példányának adjuk értékül a subclass egy példányát.

Például legyen a C_2 típusosztály a C_1 specializációja és a deklaráció értelmében $x \in C_2$ és $y \in C_1$! A deklarációk után a változók típusai a következők:

változó	statikus típus	dinamikus típus
x	C_2	C_2
y	C_1	C_1

Legyen $f : C_2 \to C_2$, és hajtsuk végre az $y := f(x)$ értékadást! Ekkor:

változó	statikus típus	dinamikus típus
x	C_2	C_2
y	C_1	C_2

Ez egyben azt is jelenti, hogy y attribútumai és metódusai az értékadás után a C_2 osztálynak megfelelőek lesznek (l. dinamikus összekapcsolás).

Dinamikus összekapcsolás: a dinamikus típusnak megfelelő kiszámítási szabály hozzárendelése a függvényhez, attribútumhoz a végrehajtás pillanatában.

Tekintsük a következő példát a specializációval történő öröklődésre! Már ismerjük a kupacok ($Kupac$) típusát. Származtassuk ebből a kettős kupacok ($DKupac$) típusát, amelyben nemcsak a maximális, hanem a minimális elem is elérhető és kivehető! Tehát:

$$\begin{aligned} Kupac &= (\{Kupac, Elem\}, \{\ddot{u}res, betesz, kivesz, csúcs\}), \\ DKupac &= (\{DKupac, Adat\}, \\ &\quad \{d\ddot{u}res, dbetesz, kimax, max, kimin, min\}). \end{aligned}$$

A származtatás során az absztrakt részben az elnevezések megfeleltetése: *DKupac* = *Kupac* és *Adat* = *Elem*. Szintaktikai és szemantikai megegyezés áll fenn a következő esetekben: *düres* = *üres*.

dbetesz = betesz, kimax = kivesz, max = csúcs. A korlátozások változatlanok. Az absztrakt rész bővül a *kimin* és *min* műveletekkel. Az implementációs rész új import felülettel és új törzs résszel rendelkezik.

3.3.2. Típusöröklődés újrafelhasználással

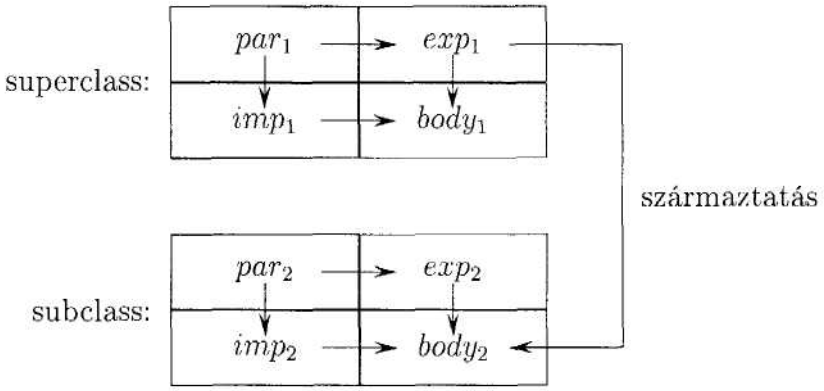
A superclass-ból a subclass-nak azt a származtatását, amely a következő' tulajdonságokkal rendelkezik, újrafelhasználással történő öröklődésnek nevezzük (3.5. ábra):

1. A subclass átveszi a superclass összes *absztrakt tulajdonságát,* és azokat a törzs részben használja fel.

2. A tulajdonságok átvételekor a típushalmaz, a paraméterhalmazok és a műveletek *nevei újradefiniálódhatnak.*

3. A subclass típushalmazának a superclass típushalmaza felel meg.

4. Az átvett műveletek jelentése nem változhat meg.



3.5. ábra. Típusöröklődés újrafelhasználással. Adott a superclass: (par_1, exp_1, imp_1, $body_1$), amelyből származtatjuk a subclass-t: (par_2, exp_2, imp_2, $body_2$). Az ábrában a nyilak az információ áramlásának irányát mutatják.

Példaként tekintsük azt az esetet, amikor a zsákok típusából származtatjuk a halmazok típusát! (A zsák olyan halmaz, amelyben minden elem multiplicitással szerepel.) Ekkor:

$$\begin{aligned} Zsák &= (\{Zsák, Elem, \mathbb{N}_0\}, \\ &\qquad \{züres, zbetesz, zkivesz, hányszor, zméret\}) , \\ Halmaz &= (\{Halmaz, Elem, \mathbb{N}_0\}, \\ &\qquad \{hüres, hbetesz, hkivesz, eleme, hméret\}) . \end{aligned}$$

A származtatás során az absztrakt rész új, az implementációs rész pedig üres lesz. A törzs rész a következőképpen alakul újrafelhasználással:

$$\begin{aligned} &Halmaz = Zsák \\ &h \in Halmaz, e \in Elem : \\ &hüres = züres \\ &hbetesz(h,e) = \begin{cases} h & , \text{ ha } eleme(h,e) = \texttt{igaz} \\ zbetesz(h,e) & , \text{ különben} \end{cases} \\ &hkivesz(h,e) = zkivesz(h,e) \\ &eleme(h,e) = hányszor(h,e) > 0 \\ &hméret(h) = zméret(h) \end{aligned}$$

4. Az objektumelvű modellezés alapjai

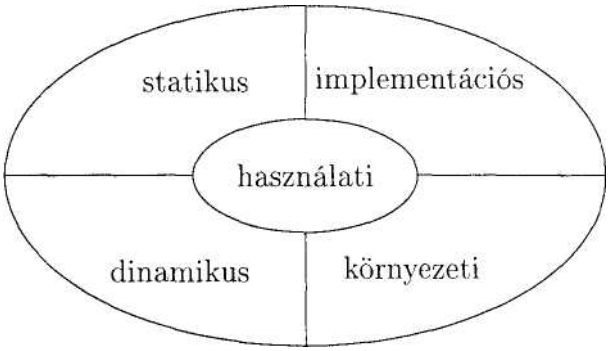
Ebben a fejezetben megvizsgáljuk, hogy milyen szempontok szerint lehet egy megoldandó problémát megközelíteni, milyen fontos a megoldás során a megfelelő szemléltetés, hogyan kapcsolódik ez az UML jelölésrendszeréhez; és megadjuk néhány alapvető fogalom definícióját.

4.1. Nézetrendszer

Az ember vizuális lény, programozási gyakorlata során számos olyan eszközt, módszert fejlesztett ki magának, amelyekkel a probléma megoldását valamilyen szempont alapján szemléltetni tudta.

A következőkben megadjuk a probléma megoldásának nézetrendszerét néhány mondatban (4.1. ábra):

- *Használati szempont:* A használati szempontból történő vizsgálat arra keresi a választ, hogy kiknek nyújt ez a rendszer szolgáltatást. Ezek lehetnek személyek, de lehetnek más rendszerek, programok is. Ebben az esetben arra kell válaszolni, hogy a rendszer szolgáltatásaival szemben támasztott követelmények teljesülnek-e.
- *Szerkezeti, strukturális, statikus szempont:* Itt arra vagyunk kíváncsiak, hogy a rendszer milyen egységekből épül fel, mi ezeknek az egységeknek a feladata, milyen kapcsolatban vannak egymással a megoldás elérésének az érdekében.

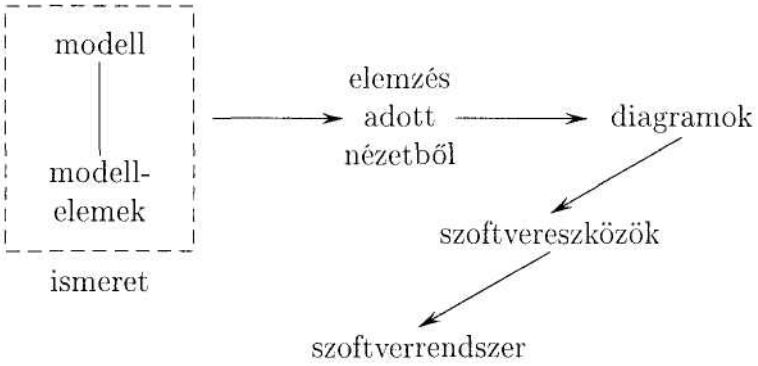


4.1. ábra. A modellalkotás nézetrendszere

- *Dinamikus szempont:* Ennek során arra keressük a választ, hogy a rendszer egyes részegységei hogyan viselkednek a megoldás során. Az egységek milyen állapotokat vesznek fel, milyen ese mények hatására változik az állapotuk, milyen a közöttük lévő együttműködés mechanizmusa, időben hogyan játszódnak le kö zöttük az üzenetek, stb. Ez a szempont az, amely a legtöbb kér désre keresi a választ.
- *Implementációs szempont:* Ekkor a megoldás megvalósításának szoftver kérdéseire keressük a választ: milyen szoftverkomponen sek vesznek részt a megoldásban, és azok között milyen kapcso latok állnak fenn.
- *Környezeti szempont:* Ilyenkor a megoldás hardver és szoft ver konfigurációjára vagyunk kíváncsiak. Azt vizsgáljuk, hogy a rendszer milyen hardver és szoftver erőforrást igényel a megol dás során.

A felsorolt nézetrendszer sokrétű, és az elemzés során a megoldás csak egy-egy szempontból szemléltethető. Ezért alakult ki sokféle szemléltető eszköz az idők folyamán. Ezek nagyrészt diagram formájában láttak napvilágot.

Az UML-ben megtestesülő filozófia az, hogy a lényeges, a gyakorlatban bevált diagramokat szabványos formában, egy egységes nyelve-



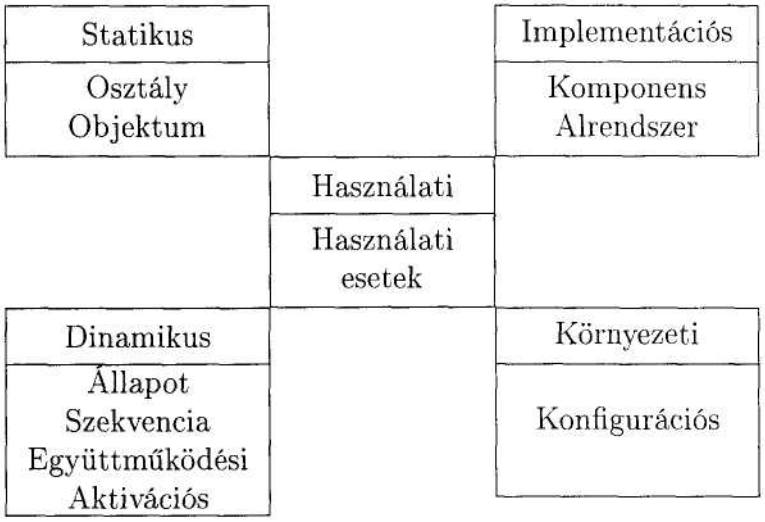
4.2. ábra. A modell transzformálása szoftverrendszerré

zetben foglalja össze. Ha ez a nyelv olyan, hogy ahhoz bizonyos szinten egyértelmű szintaxis és szemantika rendelhető, akkor szoftvereszközökkel egy adott objektumelvű programozási nyelvre áttranszformálható (4.2. ábra).

4.1.1. Az UML diagramjai

Most megadjuk, hogy a később bevezetésre kerülő UML diagramok mely szempont szerinti nézetrendszerhez kapcsolódnak (4.3. ábra):

- Statikus szempont szerint:
  - *Osztálydiagram (Class):* a rendszer objektumelvű szerkezetének leírása.
  - *Objektumdiagram (Object):* az osztálydiagram egy példányát mutatja be.
- Dinamikus szempont szerint:
  - *Állapotdiagram (Statechart):* azt mutatja meg, hogy a rendszer milyen állapotokon keresztül, milyen állapotátmenetekkel oldja meg a feladatot.
  - *Szekvenciadiagram (Sequence):* az objektumok közötti üzenetváltások időbeli menetét szemlélteti.



4.3. ábra. Az UML diagramjainak csoportosítása a nézetrendszer alapján

—*Együttműködési diagram (Collaboration): az* objektumoknak a probléma megoldásában való együttműködését mutatja be.

—*Aktivációs diagram (Activity):* a tevékenységek és az objektumok egymásra gyakorolt hatását fejezi ki (vezérlések, rendszerfunkciók).

- Implementációs szempont szerint:

—*Komponensdiagram (Component):* a komponensekből felépülő szoftverrendszert mutatja be.

—*Alrendszerdiagram:* az alrendszerek kapcsolatát írja le.

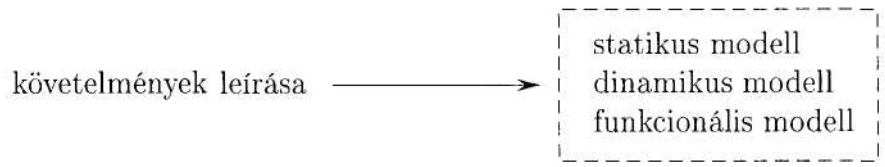
- Környezeti szempont szerint:

—*Konfigurációs diagram (Deployment):* a szoftverrendszer környezetének, a hardver-szoftver konfigurációnak a szem léltetésére szolgál.

• Használati szempont szerint:

— *Használati esetek diagramja (Use case):* a rendszernek és felhasználóinak kapcsolatát adja meg.

Az objektumelvű modellezés során a *követelmények* leírásából a megoldás három *modelljét* állítjuk elő a *problématérben* (4.4. ábra). A statikus modell a rendszer szerkezetét adja meg (statikus szempont), a dinamikus és funkcionális modell a rendszer viselkedését határozza meg (dinamikus szempont). Ezek közül elsőként a *statikus modellel* foglalkozunk.



problématér **4.4.**

ábra. A követelmények leírásából nyert három modell

A statikus modell az osztálydiagramokból és a hozzájuk tartozó osztályleírásokból, valamint az objektumdiagramokból és az ezekhez tartozó objektumleírásokból áll. Azaz:

statikus modell = osztálydiagram + osztályleírások,
objektumdiagram + objektum leírások.

A továbbiakban a statikus modell elemeivel foglalkozunk. Az osztálydiagram és az objektumdiagram megismerése előtt meg kell adnunk az objektum és az objektumosztály (röviden osztály) fogalmát.

4.2. Az objektum informális definíciója

Először felsoroljuk az objektum jellemzőit, amelyek tulajdonképpen az informális definíciót adják, majd az egyes jellemzőket külön-külön is megvizsgáljuk. Tehát az objektumokra igaz, hogy:

1. Az objektum azonosítható, az objektumok egymástól megkülön böztethetők, függetlenül azok állapotától.

2. Tulajdonságok, jellemzők, *attribútumok* tartoznak hozzá. Ezek között formális paraméterek is lehetnek.

3. Állapot tartozik hozzá. Az attribútumok konkrét értékei az ob jektum mindenkori állapotát határozzák meg.

4. Műveletek (leképezések, tevékenységek, események) tartoznak hozzá.

5. Korlátolt láthatósággal rendelkezik, azaz van *látható része,* amelyet a felhasználó ismer, és van *láthatatlan része,* amelyet a felhasználó nem ismer.

    - A látható rész az objektum külső felületét (interfészt), azaz az objektumhoz tartozó export és import műveleteknek a formáját és jelentését írja le. Ebből megtudhatjuk, hogy az objektum létezik, és hogy milyen műveletek tartoznak hozzá. (Deklaráció.)
    - A láthatatlan rész (elrejtett, burkolt rész) írja le az objektum ábrázolásának részleteit, a szolgáltatások megvalósítását. (Reprezentáció.)

6. Az objektumnak van absztrakt és konkrét megjelenési formája:

    - Az *absztrakt forma* az absztrakció valamelyik szintjének megfelelően leírt, azaz a konkrét ábrázolástól és megvalósítástól független forma.

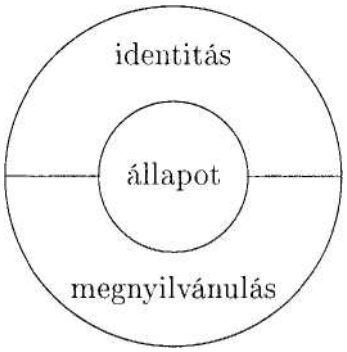
- A *konkrét forma* egy konkrét ábrázolása az objektumnak, és a hozzáférés műveleteinek ebben a formában való leírását jelenti.

7. Az objektum az osztály egy példánya. (Mi olyan objektumelvű programozással foglalkozunk, amelyben minden objektum csak az osztály egy példányaként létezik. Az objektumelvű programo zási nyelvek általában csak ezt támogatják.) Az objektum és az osztály között fennálló reláció az úgynevezett „is - a" reláció[1].

8. Az objektumot szabványos felületek veszik körül, amelyek a hoz záférések engedélyezését határozzák meg.

Ezek alapján röviden: objektum = identitás +

megnyilvánulás + állapot.



Objektumok azonosításán azt értjük, hogy a típushalmaz elemei mint objektumok megkülönböztethetők. Ezen belül az azonosítás

- történhet névvel,
- történhet olyan állításra adott válasszal, amely az adott objek tumra és csakis arra igaz.

[1]gyakran az „is - a" reláció alatt az osztályok közötti öröklődési relációt értik. Külön felhívjuk a figyelmet, hogy mi itt nem ebben az értelemben használjuk.

Az objektumhoz tartozó attribútumokra tekintsünk egy hétköznapi példát! Egy személy jellemzői, tehát egy ilyen objektum attribútumai, lehetnek a neve, neme, kora, lakcíme. Másik példaként tekinthetjük a vermeket, például:

A_1 érték : verem,

A_2 hossz : verem.

Az adattípusnál láttuk, hogy az

$a_0 \in A_0$

típusobjektum általában

$A_1, A_2, \ldots, A_n$

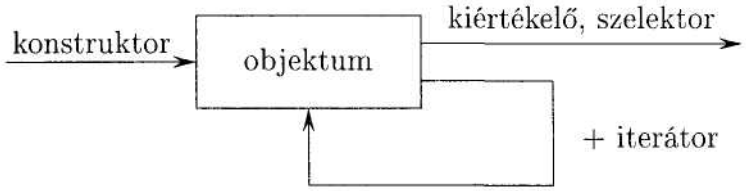
halmazok elemeiből épül fel. Az objektum is általában összetett szerkezet, amely más objektumokból épül fel; ezek az építőkövek az attribútumok aktuális értékei.

Az objektumok műveletei megfeleltethetők az absztrakt adattípusnál bevezetett műveleteknek: f_0 az üres objektum létrehozása; $f_1, \ldots, f_n$ a többi művelet.

Az objektumok műveleteit két csoportba oszthatjuk:

a. *Export műveletek* azok, amelyeket az objektum magára nézve meg enged, amelyeket más objektumok végezhetnek rajta. Például a verem magára nézve általában „megengedi" a következő műveleteket: *push, pop, top*. Az üres verem azonban nem engedi meg a *top* műveletet magára nézve. A tele verem pedig nem engedi meg magára nézve a *push* műveletet.

b. *Import műveletek* azok, amelyeket az objektum másokon végez, amelyeket igényel ahhoz, hogy az export szolgáltatásokat nyújtani tudja. Például ha a vermet egy vektorral és egy mutatóval ábrázoljuk, akkor szolgáltatásaihoz igényli a *put, access* műveleteket, amelyeket a vektoron „végez".



állapot megváltoztató **4.5. ábra.**

Az export műveletek csoportosítása

Az export műveleteket szokás további csoportokra osztani, ahogy azt a 4.5. ábra szemlélteti.

- A *konstruktőr* műveletek az objektum létrehozására, felépítésére szolgálnak. Például a verem esetében ilyen a

$$\begin{aligned} create &: \rightarrow verem, \text{ és} \\ push &: verem \times elem \rightarrow verem. \end{aligned}$$

- A *kiértékelő* műveletek az objektum bizonyos jellemzőire „kérdeznek rá". Például, hogy hány elemű vagy mekkora az objektum, vagy rendelkezik-e bizonyos tulajdonsággal:

$$\begin{aligned} length &: objektum \rightarrow \mathbb{N}_0, \\ size &: objektum \rightarrow \mathbb{N}_0, \\ has &: halmaz \times elem \rightarrow boolean. \end{aligned}$$

- A *szelektor* műveletek az objektum bizonyos részét kiemelik. Például a vektor adott indexű elemét adja az *access* művelet:

$$access : vektor \times index \rightarrow elem.$$

- Az *állapot megváltoztató* művelet az objektum attribútumának az értékét változtatja meg. Például verem esetében elem kivétele:

$$pop : verem \rightarrow verem.$$

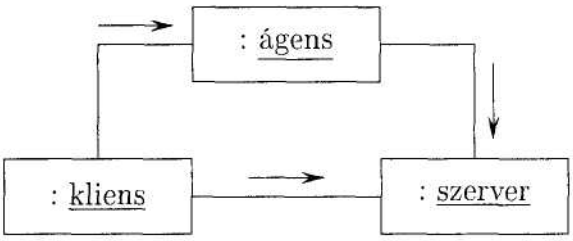
• Az *iterátor* az objektum felépítésében részt vevő komponensek bejárására szolgáló eljárás.

Az objektumokat osztályozhatjuk a műveletek típusai (az objektumok viselkedése) alapján:

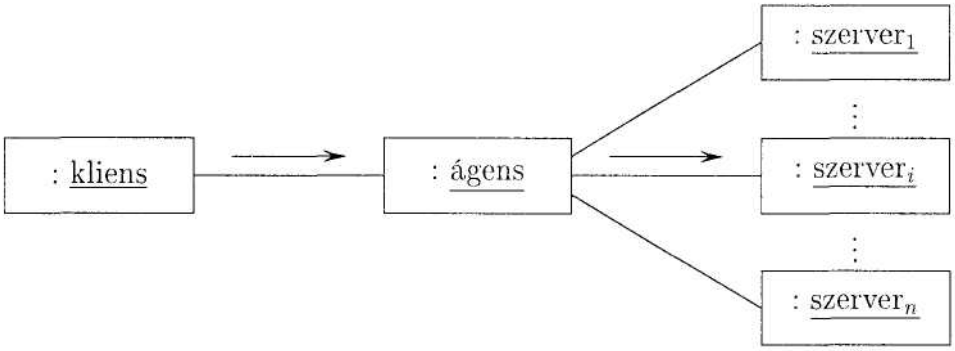
Kliens: olyan *aktív objektum,* amely csak másik objektumon végez műveleteket, de rajta mások nem végeznek műveleteket. A kliens más objektum létrejöttét, működését, megszűnését vezérelheti, de az ő működését mások nem vezérelhetik. A kliens mások szolgáltatását igénybe veszi, de másnak nem nyújt szolgáltatást. A kliensnek tehát *nincs export felülete.* Ilyen például egy óra, amely meghatározott időközönként műveleteket végez egy regiszteren, amelybe jelző „bit"-et billent be.

Szerver: olyan *passzív objektum,* amelynek csak export felülete van, azaz amelyen csak mások végeznek műveleteket, de ő másokon nem. A szerver másoktól érkező üzenetre vár, amelyben a működését kezdeményezik, szolgáltatását igénylik. A szerver másoknak nyújt szolgáltatást, de mások szolgáltatását nem veszi igénybe. A szervernek tehát *nincs import felülete.*

Ágens: általános objektum modell, amely *mind export, mind import felülettel* rendelkezik. Az ágens közvetítő szerepet tölt be a kliens és a szerver között.



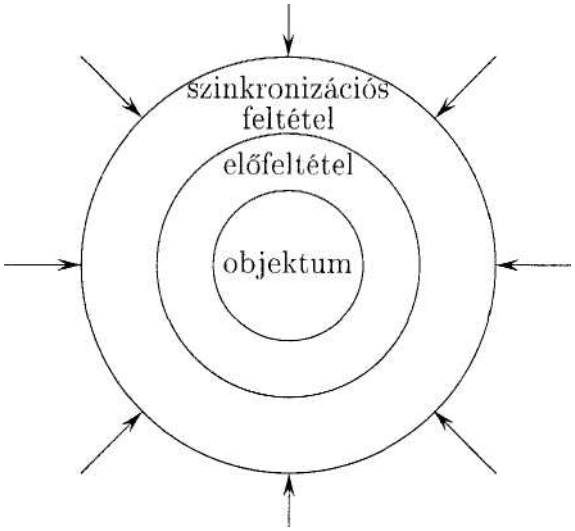
4.6. ábra. Együttműködés a kliens-ágens-szerver rendszerekben



4.7. ábra. Együttműködés az útkijelölésben az üzenetküldésnél

Ezeknek az együttműködését szemlélteti a 4.6. ábra a „kliens-ágens-szerver" rendszerekben. A kliens-ágens-szerver együttműködése az útkijelölésben az üzenetküldésnél jöhet létre (4.7. ábra).

Az objektumokat körülvevő felületeket - amelyek a hozzáférések engedélyezéséért felelnek - szemlélteti a 4.8. ábra.



4.8. ábra. Az objektumot körülvevő szabványos felületek. Erős összetartó erő érvényes belül, gyenge kapcsolódás kifelé

4.3. Az objektumosztály informális definíciója

Az objektumosztály, vagy röviden osztály, fogalmát az objektumhoz hasonló módon adjuk meg. Az osztály jellemzői:

1. *Hasonló tulajdonságú objektumok egy halmaza. A* hasonlóság az implementáció szempontjából egységesen kezelhető szerkezeti és viselkedésbeli jellemzőket jelenti.

2. *Az osztálynak van neve,* amelyet az osztályba tartozó összes ob jektum örököl. (Például egy hallgatói nyilvántartásban a hallga tók adatai egy osztályt alkotnak. Ekkor egy konkrét objektum neve: **Papp Ica,** az osztály neve: hallgató.)

3. Az osztálynak lehetnek *attribútumai, paraméterei,* amelyek az objektumoknak is közös építőkövei.

4. Tartoznak hozzá *szolgáltatások, operációk, műveletek (export fe lület),* amelyek lehetnek:

   - Objektum szintű operációk, amelyek az osztály minden ob jektumára vonatkoznak.
   - Az osztály egészére vonatkozó műveletek.

5. Az osztályhoz tartozhat *import felület,* amely az általa igényelt szolgáltatások definícióját tartalmazza.

6. Az osztály rendelkezhet *megvalósítási résszel. Í*gy az osztály spe cifikációja (3-6.) általában a következőkből áll (1. később):

   - paraméterek leírása,
   - szolgáltatások leírása,
   - import felület leírása,
   - megvalósítás leírása.

7. *Az osztálynak van látható része, és van láthatatlan része.* A *látható részt* a paraméterek, a szolgáltatások és az import felület leírása tartalmazza, a *láthatatlan részt* a megvalósítás leírása tartalmazza.

8. Az osztály lehet *absztrakt osztály.* Az absztrakt osztálynak nincs import része, és nincsen megvalósítási része sem; csak a szolgáltatások absztrakt formáját és jelentését tartalmazza[2].

9. Az osztály lehet *konkrét osztály.* A konkrét osztály minden szolgáltatásához definiált annak megvalósítása is.

10. Az osztály objektumainak attribútumaihoz és operációihoz való *hozzáférési mód 3 csoportba tartozhat:*

    - *Public,* az objektumhoz kívülről történő hozzáférést engedélyező mód.
    - *Private,* az objektumhoz kívülről történő hozzáférést nem engedélyező mód. Az attribútumok, az operációk kívülről láthatatlanok.
    - *Protected,* az objektumhoz csak az osztályon kívülről történő hozzáférést tiltó hozzáférési mód, a származtatott osztályokon belül ezek az attribútumok, operációk láthatók.

11. Az osztály lehet *paraméteres osztály (sablon).* A sablon közös formával rendelkező osztályok egy osztályát definiálja. A definíciót olyan formális paraméterekkel adjuk meg, amelyeknek

    - sem típusa,
    - sem korlátozása nincs meghatározva.

    Sablon nem lehet:

    - példány vagy

[2]Szokás absztrakt osztályról beszélni akkor is, ha van megvalósítási rész, de az nem teljes, azaz létezik olyan szolgáltatás, amelyhez nem definiált a megvalósítás.

- általánosítás (superclass).

Sablon lehet:

- specializáció (subclass),
- tartalmazhat saját formális paramétereket a definiáláshoz.

A sablon alkalmazása:

- az implementáció számára ajánlást készítünk,
- az automatikus implementációt készítjük elő.

A sablon aktualizálása:

- formális paraméterek helyébe aktuális paraméterek kerül nek,
- az aktuális paraméterek típusát megadjuk,
- a korlátokat megadjuk: «bind» (value list).

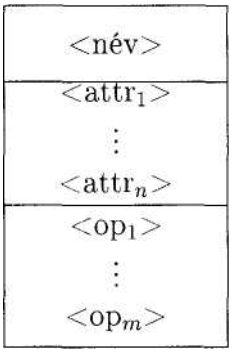
A sablon új attribútumokkal, új műveletekkel az aktualizálás során nem terjeszthető ki. Erre az öröklődés szolgál.

A következőkben részletesebben megvizsgáljuk az osztály leírását alkotó 4 összetevőt:

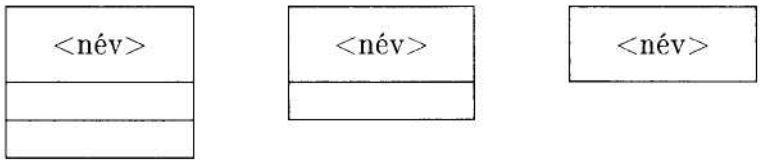
- *A paraméter rész leírása* az osztály objektumainak felépítésében szereplő jellemzők leírását tartalmazza. Ezeket a jellemzőket nevezzük *attribútumoknak,* és ezek két csoportra oszthatók:
  - —*Objektum attribútumok* azok, amelyek értékei egy-egy ob jektumra nézve specifikusak. Például az objektum állapotát meghatározó értékek.
  - —*Osztály attribútumok,* amelyeknek értékei az osztály min den objektumát jellemzik. Például az osztály objektumai nak maximális mérete.

- *A szolgáltatások leírása* az osztály által nyújtott szolgáltatások formájának és jelentésének leírását tartalmazza. Ez a leírás há rom részre tagolódik:
  - Az egyes objektumokra vonatkozó szolgáltatások, operá ciók *(objektum operációk)* leírása.
  - Az osztályra mint egészre vonatkozó operációk *(osztály ope rációk)* leírása. Például az osztály egy példányának létreho zása, az attribútumok listájának kiíratása.
  - A szolgáltatások igénybevételére vonatkozó feltételek *(kor látozások)* leírása.
- *Az import rész leírása* az osztály szolgáltatásainak megvalósí tásához szükséges, más osztályoktól igénybe vett szolgáltatások leírását tartalmazza.
- *A megvalósítás leírása* az osztály szerkezeti tulajdonságainak áb rázolását és viselkedésbeli tulajdonságainak konkrét implemen tációját írja le.

A diagramokban az osztály ábrázolásánál, jelölésénél a nevét, az attribútumok neveit és a műveletek absztrakt formáját tüntetjük fel (4.9. ábra). Absztrakt osztály esetén a nevet dőlt betűkkel adjuk meg.



4.9. ábra. Az objektumosztály jelölése



4.10. ábra. Az objektumosztály egyszerűsített ábrázolása

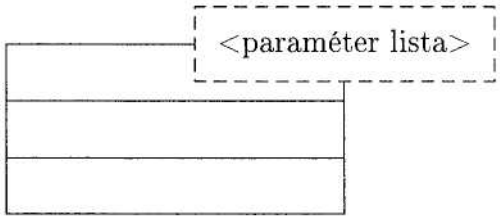
Gyakran azonban a modellalkotás során egyszerűbb formákat használunk a terv áttekinthető ábrázolásának érdekében (4.10. ábra). Ekkor az adott absztrakciós szinten nem érdekes részeket (műveletek, attribútumok) elhagyjuk.

Tekintsük konkrét példaként a kerékpárok osztályát, ahol ismerjük az egyes kerékpárok színét, típusát és azonosítóját; a lehetséges műveletek pedig a kölcsönzés és a javítás! A háromféle jelölés látható erre a konkrét példára alkalmazva a 4.11. ábrán.

4.11. ábra. Az objektumosztály ábrázolásai egy konkrét esetben

A sablon osztály jelölése a 4.12. ábrán látható. Ebben az esetben az osztály eddig megismert jelölését ki kell egészíteni a sablon paramétereivel, amelyeket a bal felső sarokban elhelyezett szaggatott vonalú téglalapban adhatunk meg.

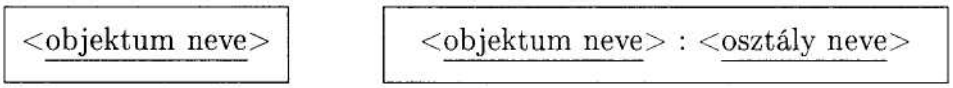
Az objektumok jelölését tartalmazza a 4.13. ábra. Az objektum neve sokszor nem elegendő az objektum azonosításához - több osztálynak lehet ugyanolyan nevű példánya -, vagy kifejezőbbé akarjuk tenni a diagramot az osztály explicit feltüntetésével. Ezekben az ese-



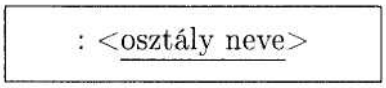
4.12. ábra. Sablon osztály jelölése

tekben az ábra jobb oldalán látható jelölést használhatjuk. Ebben a könyvben ezt a módszert követjük. Ha egy osztály tetszőleges objektumát szeretnénk jelölni, akkor azt a 4.14. ábrán látható módon tehetjük meg.

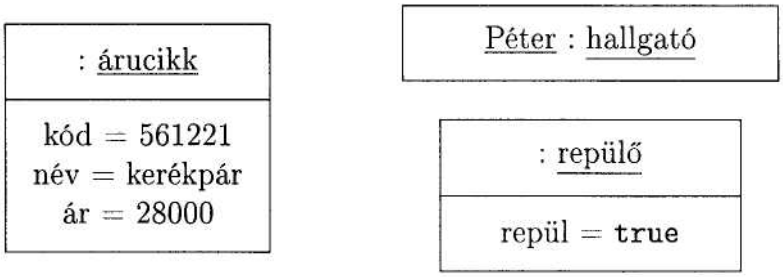
Konkrét példákat szemléltet a 4.15. ábra. Ezen látható, hogy az azonosító mellett megjelenhetnek az attribútumok értékei is.



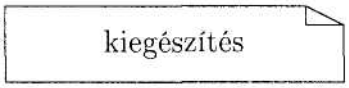
4.13. ábra. Objektumok jelölése



4.14. ábra. Osztály tetszőleges objektumának jelölése



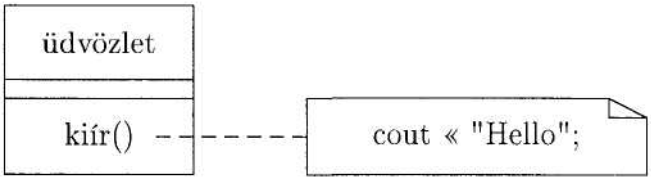
4.15. ábra. Konkrét objektumok ábrázolása



4.16. ábra. Annotáció ábrázolása

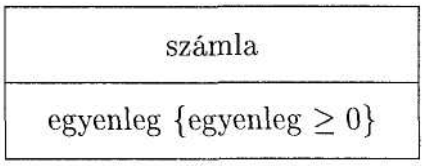
A műveletek argumentumainak meghatározása, végrehajtásának tisztázása az implementáció kérdéskörébe tartozik. Ezt támogatja az UML-ben az *annotáció*. Az annotáció az UML nyelv szemantikai kiterjesztését teszi lehetővé. Jelölése a 4.16. ábrán látható.

Egy lehetséges példa az operáció kiegészítése (4.17. ábra). Itt a művelet jelentését procedurálisan, **C++** nyelven adjuk meg az annotációban. Ez valójában a C++-ban történő implementációhoz egy ajánlásként is értelmezhető. Gyakran nem egy adott nyelven adjuk meg az annotációban szereplő implementációs javaslatot, hanem pszeudokódban.



4.17. ábra. Művelet megadása annotációval

Az attribútumok lehetséges értékeire tehetünk *megszorításokat.* Ekkor az attribútum mellé, kapcsos zárójelek közé írhatjuk a megszorítást kifejező feltételt. Például bizonyos számlák esetén a számla egyenlege nem lehet negatív (4.18. ábra).



4.18. ábra. A számla egyenlegének megszorítása

Az UML diagramokban értelemszerűen máshol is elhelyezhetünk megszorításokat. A megszorítást kifejező feltételt minden esetben kapcsos zárójelek között kell megadni.

4.4. Az osztálydiagram definíciója

Az eddigieket felhasználva már meghatározhatjuk a statikus modellben szereplő osztálydiagram fogalmát.

Az osztálydiagram a problématérben a megoldás szerkezetét leíró, összefüggő gráf[3], amelynek

- csomópontjaihoz az osztályokat,
- éleihez pedig az osztályok közötti relációkat rendeljük.

A relációk osztályok, illetve azok objektumai közötti kapcsolatot fejeznek ki. Az osztályok között a következő relációk állhatnak fenn:

- öröklődés,
- asszociáció,
- aggregáció,
- kompozíció.

Az öröklődés osztályok közötti kapcsolat, a másik három a részt vevő osztályok objektumait kapcsolja össze. A relációkkal a következő fejezetben foglalkozunk részletesen.

[3]A gráf összefüggő, hiszen az osztályoknak kapcsolódniuk kell egymáshoz, különben független rendszert alkotnának. Két osztályt ugyanakkor több él is összeköthet, mert több reláció is lehet közöttük, illetve egy relációhoz több él is tartozhat.

4.5. Az objektumdiagram definíciója

Az objektumdiagram egyszeresen összefüggő gráf, amelynek

- csomópontjaihoz az objektumokat,
- éleihez pedig az objektumok közötti összekapcsolásokat rendel jük.

A rendszerhez egy osztálydiagram tartozik, ugyanakkor egy osztálydiagramhoz több objektumdiagram tartozhat. Mindegyik objektumdiagramnak meg kell felelnie az osztálydiagramnak. A rendszer működése során dinamikusan jönnek létre, változnak és szűnnek meg objektumok, ezért az idő függvényében változhat az objektumdiagram. Az osztálydiagram a rendszer egész idejére jellemző, az objektumdiagram egy pillanathoz köthető.

Az objektumdiagramban az osztályok helyébe azok példányai, az objektumok kerülnek. Az összekapcsolások az osztálydiagramban szereplő relációk példányai, és átveszik a megfelelő tulajdonságokat. Az objektumdiagram összekapcsolásai multiplicitás nélküliek, mert az osztálydiagramban szereplő relációk multiplicitásának megfelelő számú objektum jelenik meg az adott helyen. Az öröklődési reláció nem jelenik meg az objektumdiagramban, hiszen ebben a diagramban konkrét objektumok szerepelnek.

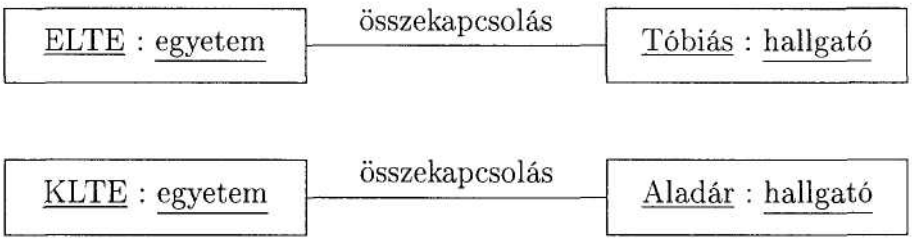
5. Objektumosztályok közötti kapcsolatok

Az előző fejezet végén említettük, hogy az objektumosztályok között kapcsolatok, relációk állhatnak fenn, amelyek közül mi négyfélét különböztetünk meg: asszociáció, aggregáció, kompozíció és öröklődés. Ezeket mutatjuk be ebben a fejezetben.

5.1. Társítási reláció, asszociáció

Ez a legáltalánosabb reláció két osztály között. Az asszociáció két osztály közötti absztrakt reláció, amely kétirányú társítást fejez ki. A reláció absztrakt volta azt jelenti, hogy a reláció konkretizálása osztályok objektumainak összekapcsolásában valósul meg.

Konkrét esetben összekapcsolásról beszélünk (5.1. ábra), míg absztrakt esetben társításról (5.2. ábra).



5.1. ábra. Objektumok közötti összekapcsolás

5.2. ábra. Objektumosztályok közötti társítás

Az asszociáció informális definíciója:

1. Asszociáció: két vagy több osztály objektumainak valamilyen relációval történő *összekapcsolása,* azaz:

$$\text{asszociáció}(C_1, C_2) = \{(obj_1, obj_2) \mid obj_1 \in C_1 \wedge obj_2 \in C_2 \wedge rel(obj_1, obj_2)\}.$$

2. Az asszociáció lehet *reflexív,* azaz azonos osztályon belüli objektumok összekapcsolását is megengedi.

3. Az asszociációhoz társulhat annak neve, *azonosítója.*

4. Az összekapcsoláshoz *irány* is társulhat, amely az aktív objektumtól a passzív objektum felé mutat, azaz a reláció értelmezésének irányát adja meg. Jele: ►.

5. Az asszociáció részleteinek leírása a hozzá *társult osztályban* kaphat helyet.

6. Az összekapcsolt objektumoknak lehet *multiplicitása* is:

   - pontosan i, jele: i;
   - i és j közötti, jele: $i..j$;
   - 0 vagy több, azaz valamennyi, jele: $*$;
   - legalább i, jele: $i..*$.

7. Az asszociációban részt vevő objektumnak lehet *szerepe* is:

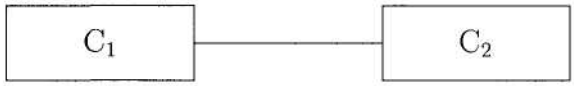
   • névvel azonosított szerep,

- kiemelt szerep,
- sorrendiségi szerep.

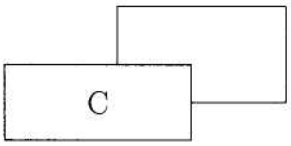
8. Az asszociációhoz *minősítő* társulhat, amelynek értékei az osz tály objektumait a társítás szempontjából diszjunkt partíciókhoz rendelik.

9. Az asszociáció esetén megadhatjuk a *navigálhatóságot.* Előfordulhat, hogy a társított osztályok objektumai nem ismerik egymást kölcsönösen, csak az egyik osztály objektumai érhetik el a másik osztályba tartozó objektumokat. Ezt fejezhetjük ki ezzel a tulajdonsággal. Ha nem tüntetjük fel, akkor kölcsönös elérhetőséget tételezünk fel.

Az asszociáció jelölése látható az 5.3. ábrán. Ezt a jelölést kiegészíthetik az informális definícióban megadott egyéb jellemzők, például multiplicitás. Ezeket a későbbi példák során mutatjuk be. A definíció szerint az asszociáció lehet reflexív. A reflexív asszociáció jelölését az ■5.4. ábra mutatja.

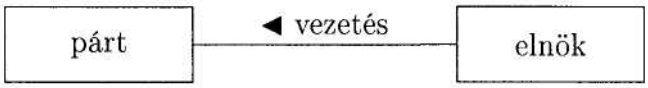


5.3. ábra. Objektumosztályok között fennálló asszociáció jelölése



5.4. ábra. Reflexív asszociáció jelölése

Példa: Tekintsük például a pártokat, amelyeket egy osztályba szervezünk, és a pártok elnökeit, akiket egy másik osztályba szervezünk! A két osztály objektumait a *„vezetés"* reláció kapcsolja össze. Feltesszük, hogy minden pártnak pontosan egy elnöke van, és egyvalaki csak egy pártnak az elnöke lehet. Az osztálydiagram az 5.5. ábrán látható.



◀

5.5. ábra. A pártok és elnökeik kapcsolatát leíró osztálydiagram. Itt a jel a reláció értelmezésének irányát mutatja.

5.1.1. A multiplicitás jelölése

A relációban az osztályok objektumai közül általában több példány is részt vehet, azaz:

$$\text{asszociáció}(C_1^m, C_2^n), \qquad m \geq 0 \wedge n \geq 0 \ .$$

A lehetséges eseteket és a megfelelő jelöléseket vizsgáljuk egy-egy példán keresztül a következőkben. A példában a *személy* és a *telefonszám* osztályokat kapcsoljuk össze asszociációval, azaz C_1 a személyek osz-

Az $m = 1 \wedge n = 1$, „egy az egy”-hez kapcsolat esete látható az 5.6. ábrán.

Az $(m = 0 \vee m = 1) \wedge n = 1$ esetet tartalmazza az 5.7. ábra.

Az $m \geq 0 \wedge n = 1$ esete látható az 5.8. ábrán.

Az $m \geq i \wedge n = 1$ esetet szemlélteti az 5.9. ábra.

Az $m = i \wedge n = 1$ eset látható az 5.10. ábrán.

Az $i \leq m \leq j \wedge n = 1$ esetet szemlélteti az 5.11. ábra.

5.6. ábra. Minden személynek van telefonszáma, és pontosan egy.

tálya, C_2 pedig a telefonszámoké.

5.7. ábra. Mindenkinek van telefonszáma, de lehet olyan szám, amelyhez nem tartozik személy (pl.: cégé), és egy számhoz legfeljebb egy személy tartozik.

5.8. ábra. Mindenkinek van telefonszáma, és egy telefonszám több személynek is (akár 0) lehet a közös telefonszáma.

5.9. ábra. Minden telefonszám legalább *i* db személy közös tulajdona.

5.10. ábra. Minden telefonszám pontosan *i* db személy közös tulajdona.

5.11. ábra. Minden telefonszámhoz *i* és *j* közötti számú személy tartozik.

5.1.2. Az objektum szerepének jelölése

Az informális definíció során említettük, hogy az asszociációval összekapcsolt osztályok objektumai különféle szerepeket tölthetnek be. A szereppel kapcsolatban az alábbi fogalmak jelölését kell bemutatni:

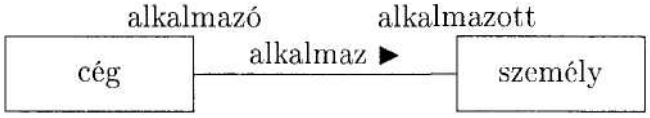
- a szerep megnevezése,
- kiemelt szerep,
- sorrendiségi szerep,
- több szerep megnevezése.

A szerep megnevezése

Az általános jelölés látható az 5.12. ábrán.

5.12. ábra. A C_1 és C_2 osztályok között fennálló asszociációban C_1 objektumai a <szerepnév> szerepet töltik be

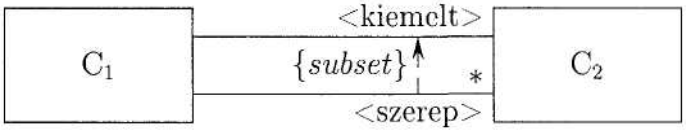
Tekintsük példaként a cégek és a személyek kapcsolatát! A két osztály a *cég* és a *személy*, amelyek asszociációban állnak egymással aszerint, hogy mely cég kit alkalmaz. Az *alkalmaz* relációban a cégek az *alkalmazó*, a személyek pedig az *alkalmazott* szerepet töltik be. A multiplicitástól tekintsünk most el! Ekkor a megfelelő diagram az 5.13. ábrán látható.



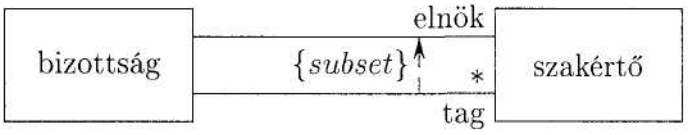
5.13. ábra. A cégek és személyek közötti alkalmazás relációban a cégek az alkalmazó, a személyek pedig az alkalmazott szerepét töltik be

Kiemelt szerep

Ebben az esetben az osztály egy vagy több objektuma a relációban, a többitől eltérő módon, más szerepet is betölt. Ennek jelölését tartalmazza az 5.14. ábra.



5.14. ábra. A C_1 és C_2 osztályok között fennálló asszociációban C_2 objektumai a <szerep> szerepet töltik be, de van egy vagy több objektum, amely a <kiemelt> szerepet is betölti

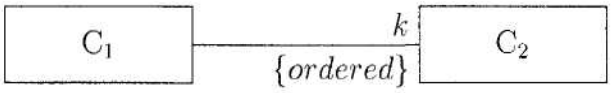


5.15. ábra. A bizottság és a szakértők közötti relációban a szakértők a tag szerepét töltik be, és minden bizottságnak van egy tagja, aki a bizottság elnöke is

Konkrét példaként tekintsük a szakértői bizottságokat! Ekkor a bizottságok osztálya kapcsolatba hozható a szakértők osztályával. A szakértők szerepe ebben a relációban a tagság, de van minden bizottságnak egy olyan tagja, aki egyben elnök is. Ezt szemlélteti az 5.15. ábra.

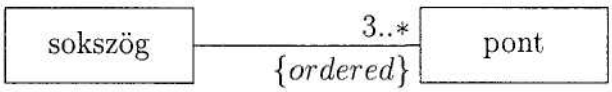
Sorrendiségi szerep

Ebben az esetben megadjuk, hogy hány objektum veszt részt a relációban, és az *ordered* alapszó feltüntetésével jelezzük, hogy ezek kötött sorrendben vesznek abban részt (5.16. ábra).



5.16. ábra. A sorrendiségi szerep jelölése *(k* > 1)

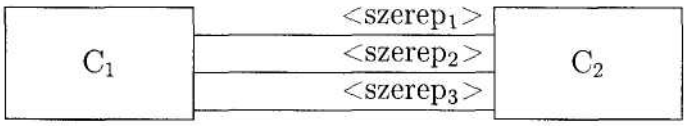
Tekintsük példaként a sokszögeket és a pontokat! A két osztály objektumai közötti kapcsolat kifejezhető, ha a sokszög csúcsait valamilyen sorrendben (óramutató járásával ellenkező irányban) adjuk meg. Egy sokszöghöz legalább három csúcspont tartozik. Az ennek megfelelő osztálydiagramm látható az 5.17. ábrán.



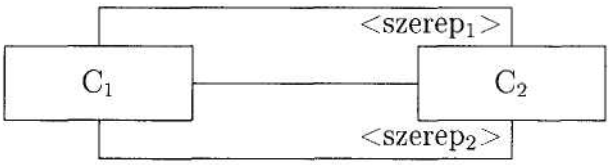
5.17. ábra. A sokszögek és a pontok közötti relációban a pontok sorrendiségi szerepének szemléltetése

Több szerep megjelölése

Ebben az esetben a két osztály közötti asszociációt több vonallal szemléltetjük, és az egyes vonalakra ráírjuk a megfelelő szerepeket, amint az az 5.18. ábrán látható. Ha az áttekinthetőség úgy kívánja, másik elrendezést is használhatunk (5.19. ábra).



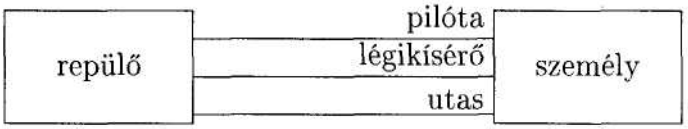
5.18. ábra. Több szerep jelölése



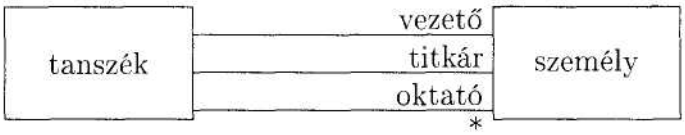
5.19. ábra. Több szerep jelölése más elrendezésben

Példánkban a C_1 osztálynak feleljen meg a repülők osztálya, C_2-nek pedig a személyeké! A személyek többféleképpen is kapcsolatba hozhatók egy repülővel: lehet köztük pilóta, légikísérő vagy utas (5.20, ábra).

Egy másik példában tekintsük az egyetemi tanszékeket! Egy tanszéknek egy vezetője és egy titkára van, továbbá több oktató dolgozik ott, de ezek mindegyike személy (5.21. ábra).



5.20. ábra. A repülők és a személyek közötti kapcsolat (multiplicitások nélkül)



5.21. ábra. A tanszékek és a személyek közötti kapcsolat

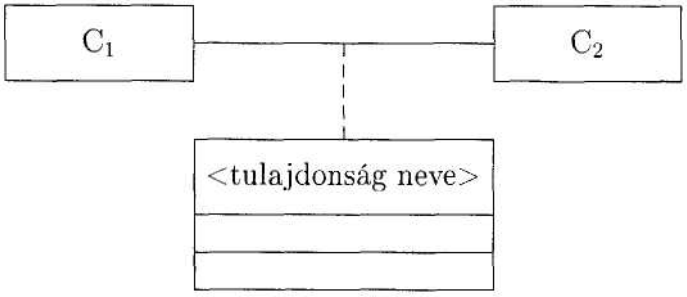
5.1.3. Az asszociációhoz kapcsolódó további jelölések

A reláció tulajdonságainak megadása

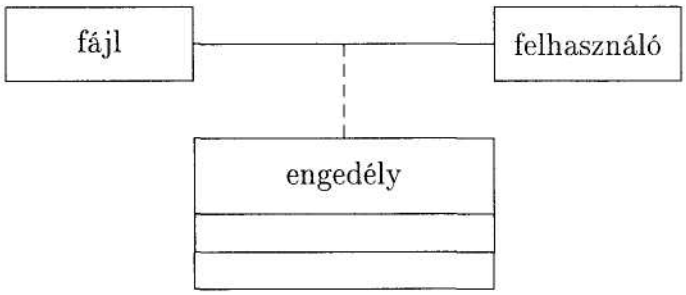
A reláció tulajdonságait, a relációra vonatkozó korlátozásokat rendszerint nem tudjuk elhelyezni a reláció mellett, noha rendkívül fontos lenne a feltüntetésük, jelölésük. Ezeket ilyenkor egy külön osztályban adjuk meg. Ez látható az 5.22. ábrán.

Példaként tekintsük a fájlokat és a felhasználókat! A két osztály objektumai közötti kapcsolat fontos jellemzője az engedély, amely a felhasználók hozzáférését szabályozza a fájlokhoz (5.23. ábra). Az engedély típusát (például Pál írásra is jogosult használni a „Hallgatói adatok" nevű fájlt) külön osztályban adhatjuk meg.

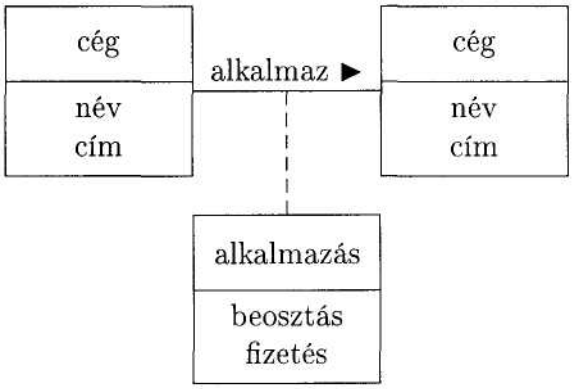
Az osztály bizonyos attribútumait, amelyek a relációhoz köthetők, a reláció tulajdonságaként tüntethetjük fel egy osztály keretében, Így mentesítjük az osztályok közötti kapcsolatot a terjedelmes felsorolástól. Erre példa a cégeknél dolgozó alkalmazottak alkalmazásukhoz



5.22. ábra. A reláció tulajdonságának megadása külön osztályban



5.23. ábra. A fájlok és felhasználók közötti kapcsolat, és az engedély jellemző jelölése



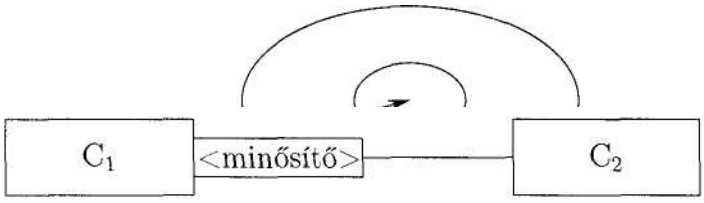
5.24. ábra. A cégek és alkalmazottak közötti kapcsolat tulajdonságainak kiemelése külön osztályba

köthető tulajdonságainak, adatainak (pl.: beosztás, fizetés) külön osztályban történő megadása (5.24. ábra).

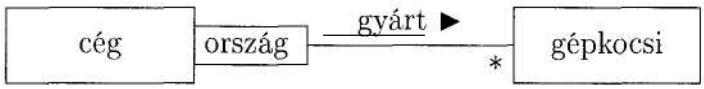
A reláció minősítése

A minősítő konkrét értékei a relációban részt vevő konkrét objektumok egy példányát, részhalmazát azonosítják (5.25. ábra). A diagramokban alkalmazott jelölést szemlélteti az 5.26. ábra.

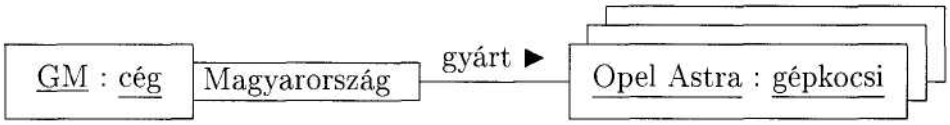
Példaként a multinacionális cégeket említjük meg, ahol a minősítő azt az országot azonosítja a relációban, amelyben a szóban forgó cégek



5.26. ábra. A minősítő jelölése a diagramokban



5.27. ábra. A multinacionális cégek és a gépkocsik kapcsolata



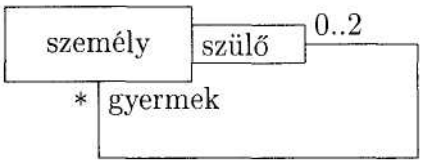
5.25. ábra. A minősítő az osztály objektumainak egy részhalmazát azonosítja

5.28. ábra. A multinacionális cégek és a gépkocsik kapcsolatának konkrét esete

működnek. Ilyen példa a gépkocsik és multinacionális gyártóik közötti reláció (5.27. ábra). Egy konkrét esetnek megfelelő objektumdiagram látható az 5.28. ábrán.

Reflexivitás

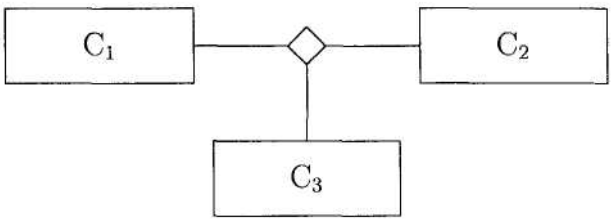
A definícióban azt mondtuk, hogy az asszociáció lehet reflexív. Ennek szemléltetésére tekintsük a személyek osztályát! (Élő személyekkel foglalkozunk.) Minden személynek lehetnek gyermekei tetszőleges számban, és minden személynek vannak szülei, akik nem feltétlen élnek. Egy szülő lehet anya, illetve apa (5.29. ábra).



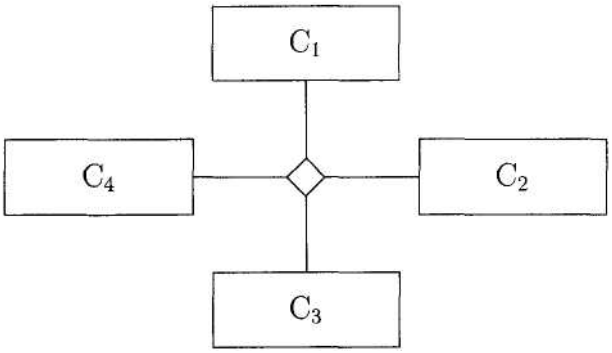
5.29. ábra. A személyek reflexív kapcsolata. Egyrészt lehet valaki gyermek, másrészt lehet szülő is (szülő = {anya, apa}).

Több osztály között fennálló asszociáció

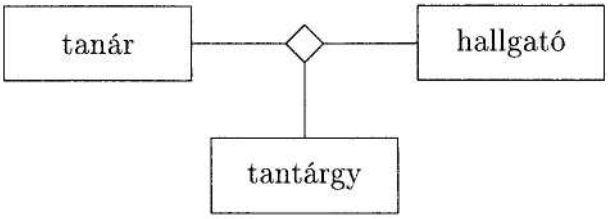
Az asszociáció nem feltétlen két osztályt kapcsol össze. Előfordulhat, hogy több osztályt hozunk kapcsolatba. Ekkor ezt a vonalak metszéspontjában elhelyezkedő rombusszal szemléltetjük (5.30. ábra, 5.31. ábra).



5.30. ábra. Három osztályt összekapcsoló asszociáció jelölése, azaz a diagram megfelel az asszociáció(C_1,C_2,C_3)-nak



5.31. ábra. Négy osztályt összekapcsoló asszociáció jelölése, azaz a diagram megfelel az asszociáció(C_1,C_2,C_3,C_4)-nek



5.32. ábra. A tanárok, hallgatók és tantárgyak közötti kapcsolat

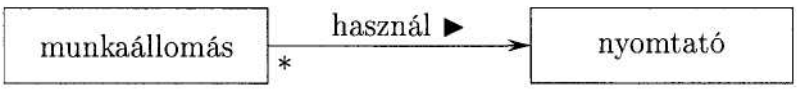
Egy lehetséges példa három osztály közötti asszociációra a tanárok, hallgatók és tantárgyak között fennálló kapcsolat (5.32. ábra).

Navigálhatóság

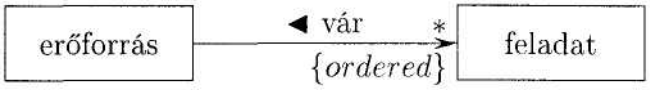
Említettük, hogy a társított osztályok objektumai nem feltétlen ismerik egymást kölcsönösen. Ha az ismeret csak egyirányú, akkor lehetőségünk van ennek kifejezésére (5.33. ábra), amivel az implementáció számára is útmutatást adunk.

5.33. ábra. A navigálhatóság jelölése. Ebben az esetben csak C_1 objektumai ismerik C_2 objektumait.

Tegyük fel, hogy egy rendszerben több munkaállomásról lehet egy nyomtatót használni! Ekkor a nyomtatónak nem feltétlen kell ismernie a munkaállomásokat, elég a munkaállomásoknak elérniük a nyomtatót, hiszen így ki tudják nyomtatni a kívánt dokumentumokat. Ezt szemlélteti az 5.34. ábra.



5.34. ábra. A munkaállomások és a nyomtató kapcsolata



5.35. ábra. Egy erőforrásnál kiszolgálásra várakozó feladatok

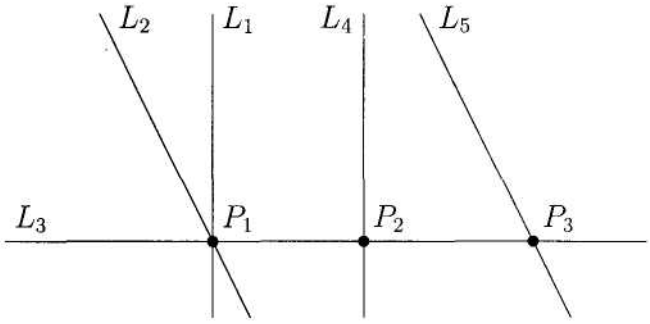
A navigálhatóság iránya nem feltétlen esik egybe a reláció irányával. Például egy erőforrásnál kiszolgálásra várakozhatnak feladatok. Ha a reláció neve „vár", akkor az irány a feladat felől mutat az erőforrás felé. Ugyanakkor elegendő az erőforrásnak ismernie a feladatokat, hiszen így mindig végre tudja hajtani a következőt (5.35. ábra).

5.1.4. Példák asszociációra

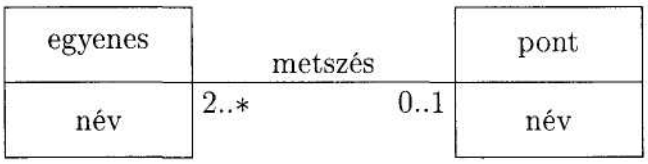
Az első példa síkbeli egyenesek és pontok kapcsolatát kifejező asszociáció. A síkbeli egyenesek lehetnek párhuzamosak, azaz nem metszik egymást; illetve metszik egymást: akkor legalább két egyenesnek van ebben szerepe, és ebben az esetben van egy közös metszéspontjuk is. A reláció konkrét esetei láthatók az 5.36. ábrán.

Ezt felhasználva megadhatjuk az egyenesek és a pontok osztályát összekapcsoló reláció diagramját (5.37. ábra).

A második példa az étkező filozófusok problémája. Ez szavakban megfogalmazva a következő: *n* filozófus ül egy kör alakú asztal körül., és van *n* villa az asztalon, bármely két szomszédos filozófus között



5.36. ábra. A síkbeli egyenesek és pontok



5.37. ábra. A síkbeli egyenesek és pontok asszociációjának osztálydiagramja

pontosan egy. Minden filozófus egy bizonyos ideig gondolkodik; ezután megpróbálja felvenni a tőle balra és jobbra eső villát, és ha ez sikerül, eszik, majd leteszi a villákat.

A problémát leírhatjuk n darab folyamat: $F_0, F_1, \ldots, F_{n-1}$ és n darab erőforrás: $V_0, V_1, \ldots, V_{n-1}$ segítségével. A folyamatok megfelelnek a filozófusoknak, az erőforrások a villáknak. Az elmondottak alapján az i. filozófusnak megfelelő folyamat ($i-1$ indexelés modulo n értendő):

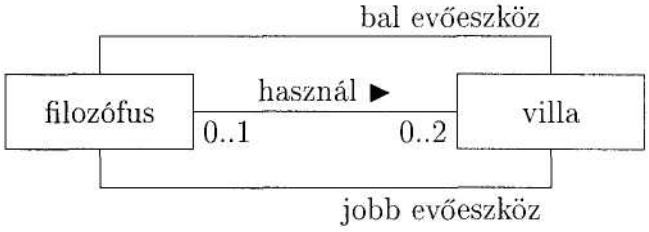
F_i : *while* `true` *do*
gondolkodik;
(< V_{i-1} lekötése > || < V_i lekötése >)
eszik;
(< V_{i-1} elengedése > || < V_i elengedése >)
od

Az eddig elmondottak alapján két osztályt azonosíthatunk:

- a *filozófusok* és
- a *villák* osztályát.

A köztük fennálló reláció: a filozófus *használja* a villát.

Vizsgáljuk meg a multiplicitás kérdését! Ezt megtehetjük, ha megvizsgáljuk a filozófus használatában levő villák számát a tevékenységei során:



5.38. ábra. Az étkező filozófusok problémájának osztálydiagramja

Tehát a filozófus objektumok közül 0 vagy 1, a villa objektumok közül 0, 1 vagy 2 vesz részt a kapcsolatban. Az ennek megfelelő osztálydiagram látható az 5.38. ábrán.

5.2. Aggregáció

Az asszociáció az osztály objektumainak egymáshoz rendelését fejezi ki. Az egymáshoz rendelés különböző erősségű kapcsolódást jelenthet. Az asszociációt általában egymástól független osztályok társításának a kifejezésére használjuk. Az aggregáció ennél erősebb kapcsolat, amely olyan jellegű kapcsolatokat fejez ki, mint:

- egész és annak részei,
- felépítmény és annak komponensei.

Az aggregáció informális definíciója:

1. Az aggregáció egy *speciális asszociáció.*
2. Az aggregációs reláció azt fejezi ki, hogy az egyik osztály objek tumai részét képezik egy másik osztály objektumainak:

$$A \text{ is an aggregation of } B \;=\; \{(a,b) \mid a \in A, b \in B, b \text{ is a part of } a\}.$$

3. Az aggregáció *tranzitív:*

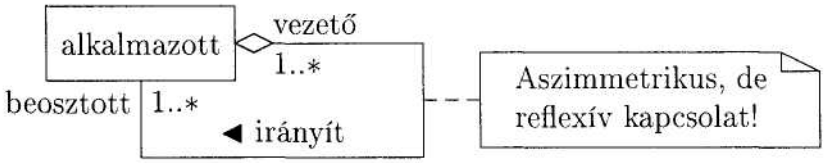
$$A \text{ is an aggregation of } B \wedge B \text{ is an aggregation of } C \rightarrow A \text{ is an aggregation of } C.$$

$$A \text{ is an aggregation of } B \rightarrow \neg(B \text{ is an aggregation of } A).$$

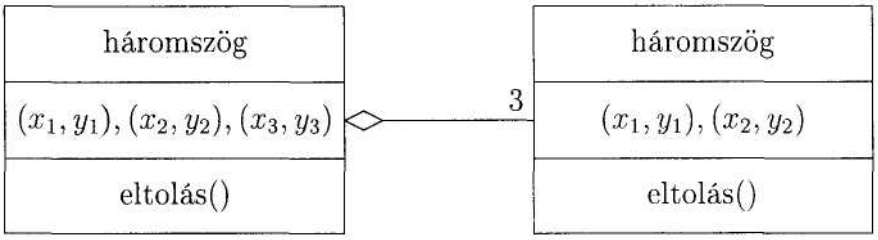
5. Az aggregáció lehet reflexív (5.40. ábra).
6. Ha *(A* is an aggregation of *B)* és *f* az *A* osztály egy szolgáltatása, akkor *f* a *B* osztálynak is egy szolgáltatása lesz (5.41. ábra).
7. Ha (A is an aggregation of B) és *attr* az *A* osztály egy att ribútuma, akkor *attr* a *B* osztálynak is egy attribútuma lesz (5.41. ábra).
8. Ha *A* és *B* osztályok között aggregációs kapcsolat áll fenn, akkor az *A* osztály objektumai és a *B* osztály objektumai *egymástól függetlenül is létezhetnek.*
9. Különböző' aggregátumoknak lehetnek közös komponensei (5.42. ábra).

Az aggregáció jelölése látható az 5.39. ábrán. Reflexív aggregációra példa az 5.40. ábra. Közös szolgáltatásra és attribútumra mutat egy esetet az 5.41. ábra, míg közös komponenseket szemléltet az 5.42. ábra.

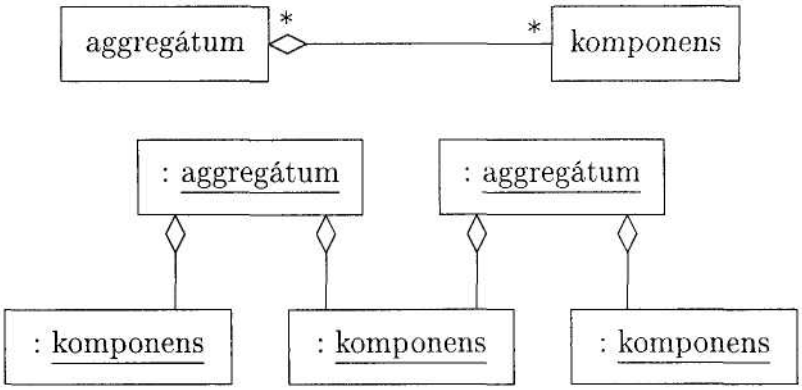
5.39. ábra. Az *(A* is an aggregation of *B)* reláció jelölése



5.40. ábra. Példa reflexív aggregációra

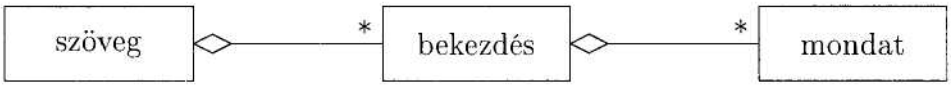


5.41. ábra. Közös attribútum és szolgáltatás aggregáció esetén



5.42. ábra. Közös komponensek különböző aggregátumok esetén

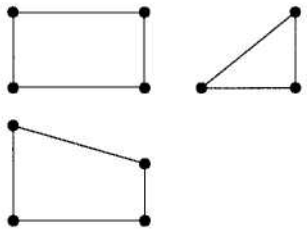
Aggregációra első példaként tekintsük az írott szövegeket! Ezek bekezdésekből állnak, a bekezdések pedig mondatokból. Így három osztályunk lesz: a szöveg, a bekezdés és a mondat, amelyekből a fentiek alapján az 5.43. ábrán látható osztálydiagramot kapjuk.



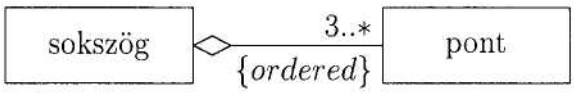
5.43. ábra. A szöveg felépítése bekezdésekből és mondatokból

A következő példában vizsgáljuk meg a síkbeli sokszögek és a sík pontjainak viszonyát! A sokszögek csúcsai pontok - méghozzá megfelelő sorrendben, ahogy az asszociáció tárgyalásakor már láttuk -, így a két osztály között aggregációs kapcsolat áll fenn. Ha feltesszük, hogy egy síkbeli pont csak egy sokszöghöz tartozhat, és csak ilyen pontokat veszünk figyelembe (5.44. ábra) - diszkrét eset -, akkor az 5.45. ábrán látható osztálydiagramhoz jutunk.

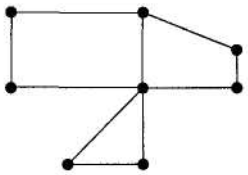
Tekintsünk most el az előbbi egyszerűsítéstől, azaz most már megengedjük, hogy egy pont akárhány sokszöghöz tartozzon! Továbbra is csak olyan pontot vegyünk figyelembe, amelyik legalább egy sokszög csúcspontja (5.46. ábra)! Ekkor az 5.47. ábra osztálydiagramja lesz az eredmény.



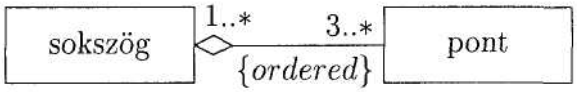
5.44. ábra. Síkbeli sokszögek és pontok kapcsolata, diszkrét eset



5.45. ábra. A sokszögek és pontok aggregációs kapcsolata a diszkrét esetben



5.46. ábra. Síkbeli sokszögek és pontok kapcsolata általában, átfedéses eset



5.47. ábra. A sokszögek és pontok aggregációs kapcsolata az átfedéses esetben

5.3. Kompozíció

Az aggregációs társítások között a legerősebb kapcsolat a kompozíciós kapcsolat.

A kompozíció informális definíciója:

1. A kompozíció egy *speciális aggregáció.*

2. A kompozíciós kapcsolat azt fejezi ki, hogy az egyik osztály ob jektumai a másik osztály objektumait *fizikailag tartalmazzák:*

$$A \text{ composition of } B = \{(a, b) \mid a \in A, b \in B, a \text{ contains } b\}.$$

3. A kompozíciós kapcsolat és az attribútum jellegű kapcsolat két objektum között szemantikailag azonos, csupán grafikai megje lenítésük különböző. Az attribútum jelölése valójában a kompo zíció jelölésének egy leegyszerűsített formája.

4. Egy komponens objektum legfeljebb csak egy aggregációs objek tumhoz tartozhat.

5. Az aggregációnak tetszőleges számú kompozíciója lehet.

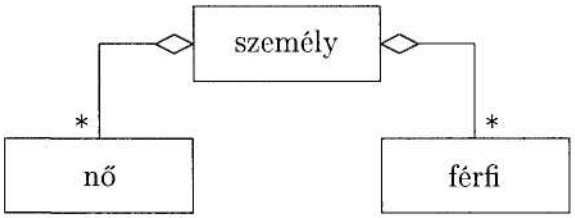
5.48. ábra. Az *(A* composition of *B)* reláció jelölése

6. Az aggregációs objektum és annak komponensei azonos életciklusban léteznek, azaz egyszerre jönnek létre, és egyszerre szűnnek meg.

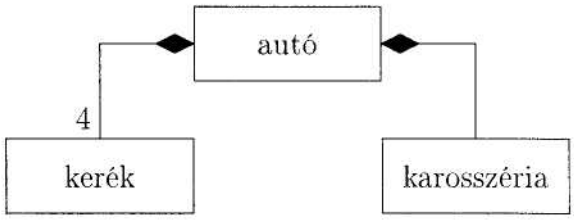
A kompozíció jelölése látható az 5.48. ábrán.

Vizsgáljuk meg egy-egy példán keresztül az aggregáció és a kompozíció közötti különbséget! Elsőként tekintsük a személyek, illetve a nők és férfiak osztályát! Azt mondhatjuk, hogy aggregációs kapcsolat áll fenn a személyek és nők, illetve a személyek és férfiak között, hiszen minden nő és férfi egyben személy is. Ugyanakkor ebben az esetben fizikai tartalmazás nem áll fenn, tehát kompozícióról nem lehet szó (5.49. ábra).

Másodikként foglalkozzunk az autók, kerekek és karosszériák osztályával! Tudjuk, hogy egy autónak négy kereke és egy karosszériája van, továbbá ez fizikai tartalmazást is jelent. Az autó ezek nélkül nem is létezhet, így a kompozíció minden feltétele megvan. Ebben az esetben tehát a kompozíció a megfelelő reláció (5.50. ábra).



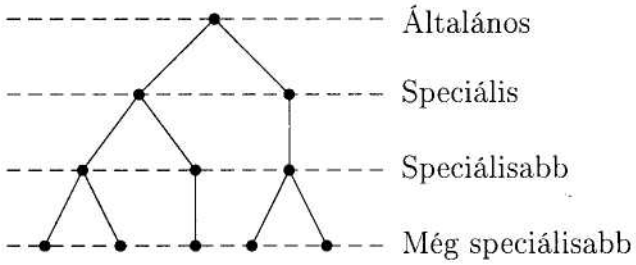
5.49. ábra. A személyek és a nők, illetve férfiak közötti aggregáció



5.50. ábra. Az autó és a kerék, illetve karosszéria közötti kompozíció

5.4. Általánosítás és specializáció

A modellalkotásban ez a reláció nem pontosan azonos a típusosztály esetében tárgyalt öröklődéssel, annál általánosabb fogalom. Az általánosítás a modellalkotásban az általános tulajdonságokkal rendelkező dolog (superclass, ősosztály) és a kevésbé általános, speciálisabb dolog (subclass, származtatott osztály) között fennálló reláció. Az általánosítás egy klasszifikációs megközelítés. Ennek az a lényege, hogy először létrehozzuk az általános tulajdonságokkal felruházott osztályt, majd annak a tulajdonságait átvéve származtatjuk a speciálisabb tulajdonságokkal rendelkező osztályt. Ezt az eljárást tovább folytatva az osztályokat egy hierarchikus szerkezetbe rendezzük. Többszörös öröklődést is megengedve az osztályokat egy fa struktúrába rendezhetjük. A fa struktúrában a gyökérelemtől távolodva bonyolultsági szinteket hozhatunk létre (5.51. ábra).



5.51. ábra. Öröklődési fa szerkezet

Ezt a relációt a tulajdonságai alapján a következő módon definiálhatjuk:

1. A reláció egy általános konstrukció és egy speciális konstrukció közötti kapcsolatot fejez ki. Mi csak azzal az esettel foglalkozunk, amikor a konstrukció osztályt jelent.

2. A reláció azt fejezi ki, hogy a speciális osztály az általános osztályból származtatással jön létre.

3. A származtatásnál a származtatott speciális osztály:

   - átveszi az általános osztály tulajdonságait (név, attribútum, operáció, asszociáció);
   - az átvett jellemzőkhöz, attribútumokhoz, operációkhoz stb. bővítésként új jellemzőket vezethet be;
   - az átvett jellemzőket újrafogalmazhatja, de úgy, hogy a speciális osztály objektumai az általános osztály helyére is behelyettesíthetők lehessenek:

     $b \in B \wedge B$ is a specialization of $A \rightarrow b \in A$.

4. A származtatás az öröklődési technika felhasználásával történik, amelynek speciális esetei a következők:

   - specifikációs (interfész) öröklődés, amikor az általános osztály absztrakt tulajdonságait veszi át a származtatott osztály;
   - implementációs öröklődés, amelynek során az általános osztály konkrét tulajdonságait veszi át a származtatott osztály, az absztrakt tulajdonságokkal együtt.

5. A származtatás nem szimmetrikus.

6. A származtatás nem lehet reflexív.

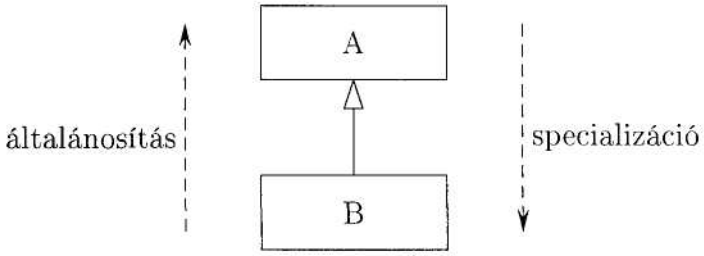
7. Az általánosítás és specializáció új osztályok létrehozásának egy technikája, amely absztrakt és konkrét osztályok létrehozására egyaránt alkalmas.

8. A specializáció lehet többszörös, ekkor egy általánosításból több származtatott jön létre.

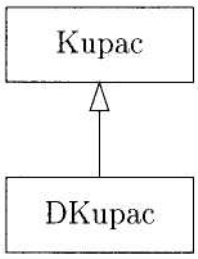
9. Az általánosítás is lehet többszörös, amikor a származtatás több általánosításból történik.

10. A származtatás az úgynevezett *„is a kind of*" reláció.

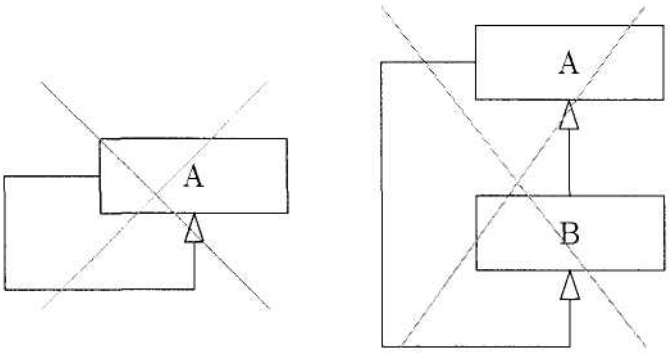
Az általánosítás és a specializáció jelölése látható az 5.52. ábrán. Példaként tekinthetjük a már bemutatott kupacok *(Kupac)* és kettős kupacok *(DKupac)* osztályát (5.53. ábra).



5.52. ábra. A *(B* is a specialization of *A)* reláció ábrázolása osztálydiagramokban



5.53. ábra. A kupacok *(Kupac)* és a kettős kupacok *(DKupac)* osztálya közötti reláció

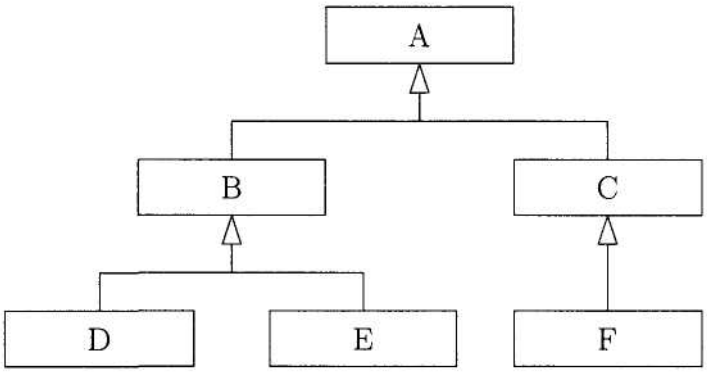


5.54. ábra. Specializáció esetén nem fordulhatnak elő ezek az esetek. A bal oldali esetben ugyanis reflexív, a jobb oldali esetben pedig szimmetrikus lenne a reláció, ami ellentmond a definíciónak.

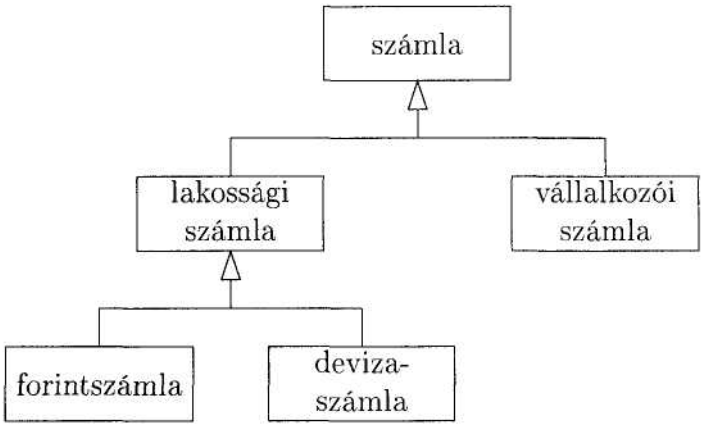
A definícióban kikötöttük, hogy az általánosítás, illetve a specializáció nem lehet sem reflexív, sem szimmetrikus, azaz az 5.54. ábrán látható esetek nem fordulhatnak elő.

5.4.1. Többszörös specializáció

Ebben az esetben az absztrakciós szintek hierarchikus szerkezetét hozzuk létre, az osztályokat fa struktúrába szervezve (5.55. ábra). A fa



5.55. ábra. Többszörös specializáció jelölése



5.56. ábra. A számlák és a specializációval kapott osztályok

alacsonyabb szintjén elhelyezkedő osztály specializációja a gyökérből hozzá vezető úton elhelyezkedő osztályoknak, beleértve a gyökeret is.

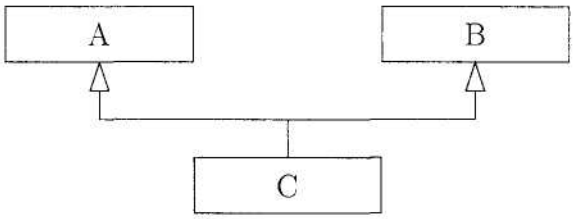
Példaként tekintsük a bankszámlákat, röviden számlákat! Ezeket tovább lehet bontani lakossági és vállalkozói számlákra. A lakossági számlákon belül megkülönböztethetünk forint- és devizaszámlákat. Ezt szemlélteti az 5.56. ábra.

Az általános osztály megalkotása jó absztrakciós készséget kíván meg. A nehézség a specializáció során jól használható általános tulajdonságok megragadásában van. A speciális osztály megalkotása ennek birtokában már viszonylag egyszerű feladat, ha az alkalmazás területén mély ismeretekkel rendelkezik a modellalkotó.

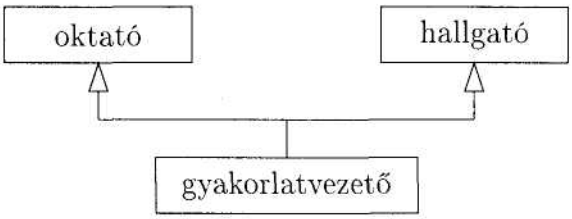
5.4.2. Többszörös általánosítás

Ebben az esetben egy osztály több osztály tulajdonságaival rendelkezik, ennek megfelelően a diagramokban ez „fordított fa"-ként jelenik meg (5.57. ábra).

Az egyetemen oktatók is és hallgatók is vezetnek gyakorlatot. A gyakorlatvezető tehát rendelkezik mind az oktatók, mind a hallgatók bizonyos tulajdonságaival (5.58. ábra).



5.57. ábra. Többszörös általánosítás ábrázolása



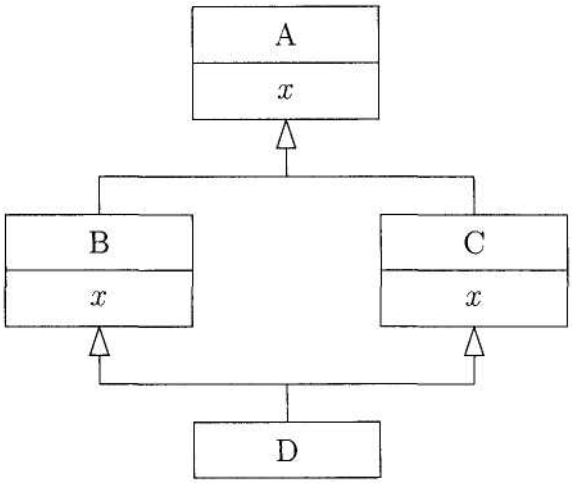
5.58. ábra. A gyakorlatvezető rendelkezhet az oktatók és a hallgatók tulajdonságaival is

Egynél több általánosítás alkalmazása csak megfelelő korlátozó feltételek mellett teszi lehetővé az átvett információk összekeveredés nélküli kezelését. Ezért az objektumelvű programozási nyelvek több általánosítás alkalmazását nem ajánlják. (Például: JAVA, ADA95.)

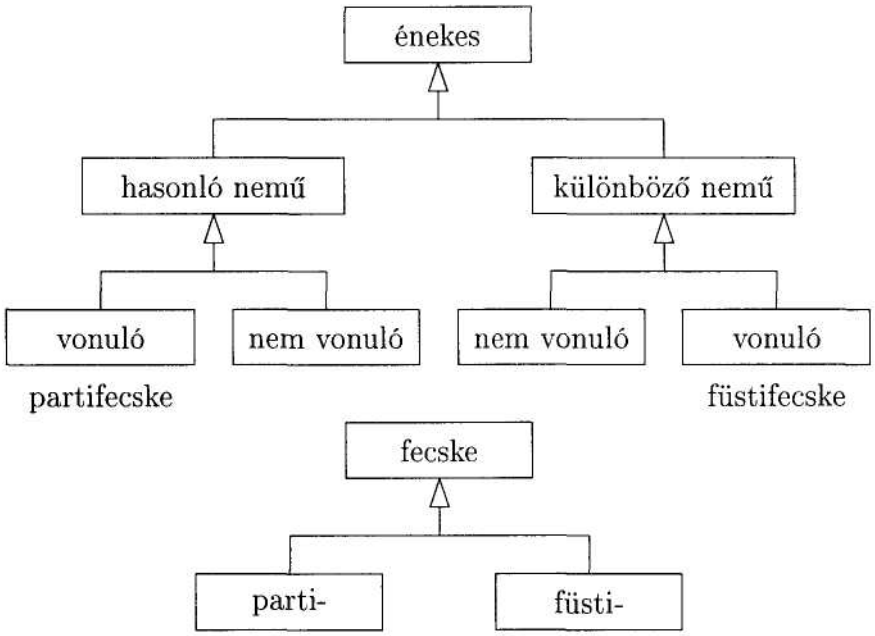
Ezzel analóg módon, a többszörös specializáció során merül fel a *kovariancia problémája, azaz* a különböző utakon származtatott tulajdonságok kezelésének problémája. Ezt általában egymástól független kritériumok alapján történő származtatással próbálják kezelni, amikor a kovariancia nem magában a probléma természetében gyökerezik.

A probléma szemléltetésére tekintsük az 5.59. ábra diagramját! A „D" osztály egy *d* objektuma rendelkezik az *x* attribútummal. Ugyanakkor *d.x* jelentheti az „A", a „B" vagy a „C" osztályhoz tartozó attribútumot, ebben az esetben ez nem egyértelmű.

Az osztályozás problémája számos tudományterületen felmerül. Például tekintsük a biológiában a madarak osztályozásának a problémáját, azon belül pedig az énekes madarak osztályozását! Bizonyos elhanyagolással az 5.60. ábrán látható osztálydiagramhoz jutunk.

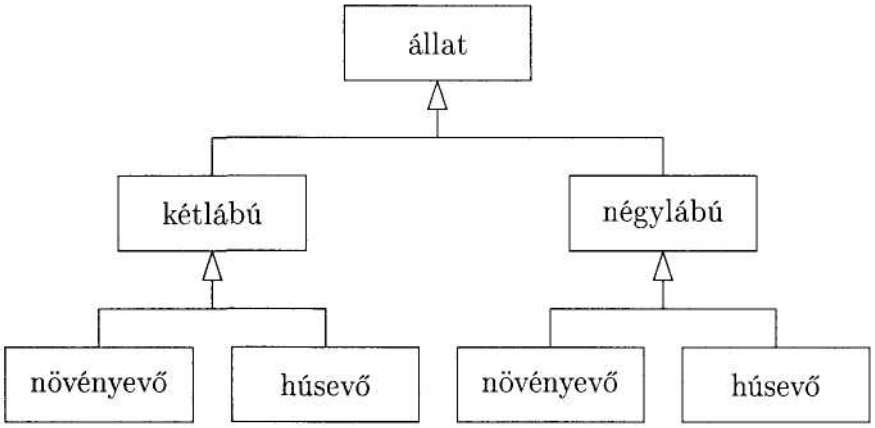


5.59. ábra. A többszörös specializáció során fellépő kovariancia probléma

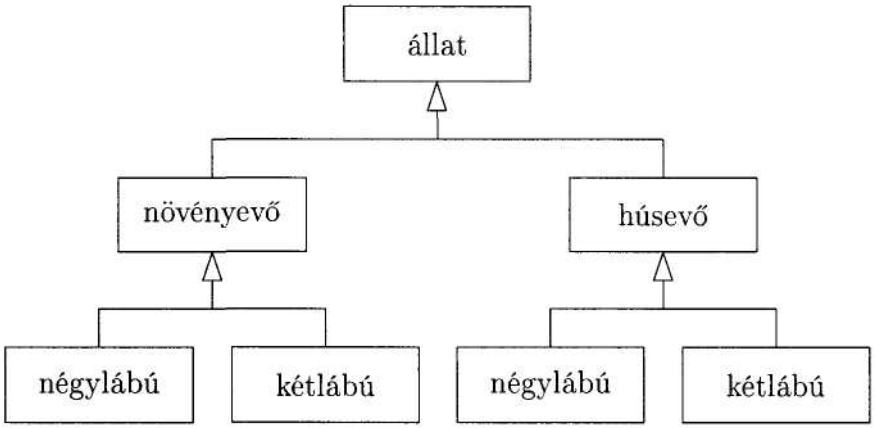


5.60. ábra. Az énekes madarak osztályozása. A partifecske például hasonló nemű, vonuló; a füstifecske pedig különböző nemű, vonuló.

Egy másik probléma a hierarchikus kapcsolatok meghatározása. Például az állatokat osztályozhatjuk először a lábuk száma alapján, majd aszerint, hogy mivel táplálkoznak (5.61. ábra). De ezt megtehetjük fordítva is, azaz osztályozhatunk először a táplálkozás, majd ezután a lábszám alapján (5.62. ábra).



5.61. ábra. Az állatok osztályozása: lábszám, aztán táplálkozás szerint



5.62. ábra. Az állatok osztályozása: táplálkozás, aztán lábszám szerint

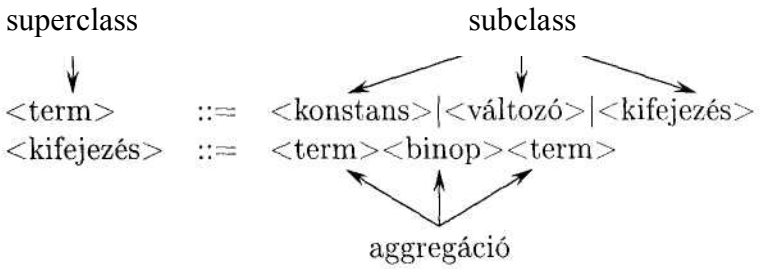
5.5. Osztálydiagramok készítése

A fejezet eddigi részeiben megismertük az osztályok közötti relációkat és jelölésüket. Felmerül a kérdés: ezek után hogyan lehet egy problémához tartozó osztálydiagramot megalkotni a követelmények elemzése alapján. Ezzel fogunk most foglalkozni, feladatok megoldása során.

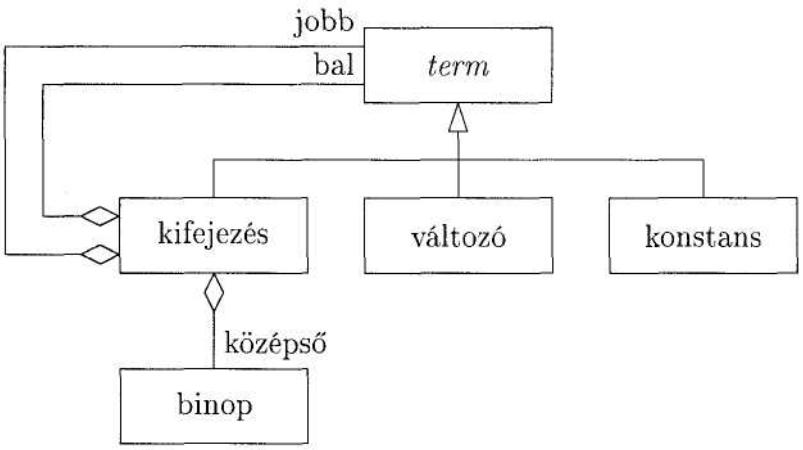
Általában kétféle követelményleírással találkozunk. Ha a leírás matematikailag jól meghatározott, akkor a diagram elkészítése viszonylag könnyű feladat. Ezt szemlélteti az első feladat. Ha a leírás nem formális, akkor a mondatok analizálásával juthatunk eredményre. Először megpróbáljuk elkülöníteni a szövegben a probléma szempontjából lényeges főneveket, amelyek az osztályoknak felelnek meg, majd a kapcsolatokat derítjük fel a leírás alapján. Ezt a folyamatot pontosítjuk a harmadik feladat előtt, amelyik már nem formális leírással megadott.

5.1. feladat:
Tekintsük a programozási nyelvekből jól ismert *„term"* egyszerűsített definícióját! Ez lehet konstans, változó vagy kifejezés. A kifejezés két termből áll, amelyeket egy műveleti jel, operátor választ el. Adjuk meg a term osztálydiagramját!

Megoldás: Írjuk fel a leírás alapján a BNF-et (5.63. ábra)! Ebben az öröklődési reláció a „vagy" kapcsolat, az aggregációs reláció pedig az „és" kapcsolat formájában jelenik meg. A termeknek egy absztrakt osztály felel meg. Ennek alapján adódik az osztálydiagram (5.64. ábra).



5.63. ábra. A term BNF leírása a definíció alapján



5.64. ábra. A term osztálydiagramja

5.2. feladat:

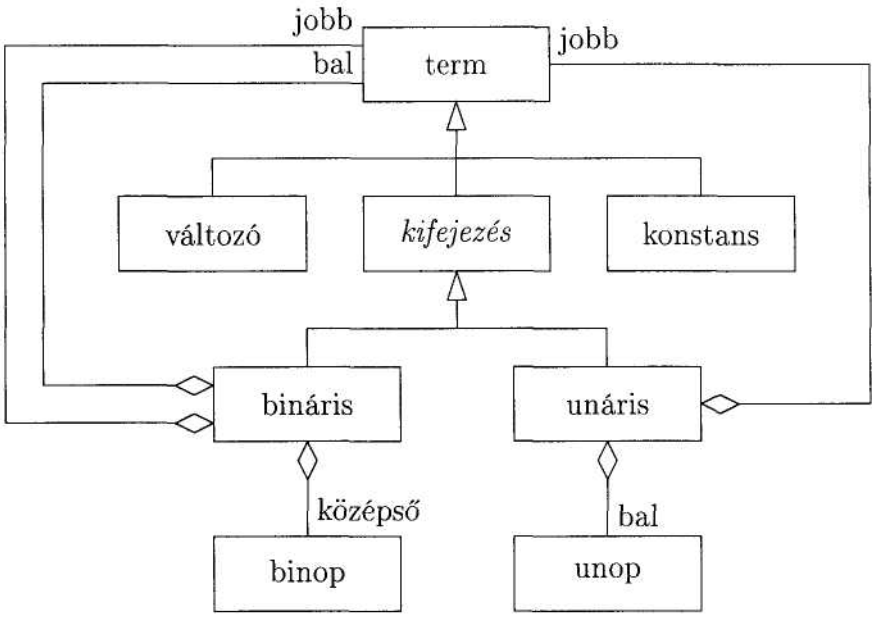
Módosítsuk az előző feladatot úgy, hogy nemcsak bináris, hanem unáris operátor is szerepelhet egy kifejezésben!

Megoldás: Írjuk fel a módosításnak megfelelő BNF-et! A term nem változik, csak a kifejezés. A bináris kifejezés felel meg az előző feladatban szereplő kifejezés fogalmának. A másik lehetséges kifejezés az unáris, amelyet egy unáris műveleti jel és egy term alkot.

```
   <term>  ::=  <konstans>|<változó>|<kifejezés>
<kifejezés>  ::=  <bináris>|<unáris>
  <bináris>  ::=  <term><binop><term>
   <unáris>  ::=  <unop><term>
```

Innen az előzőhöz hasonló módon jutunk az 5.65. ábra osztálydiagramjához. A kifejezés osztály egy absztrakt osztály lesz, amelyből származtatjuk a bináris és unáris kifejezések osztályát.

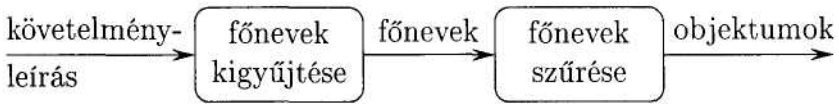
Sokkal nehezebb feladat az osztálydiagram megalkotása akkor, ha a szövegesen megadott követelmények elemzése alapján kell azt végrehajtani. Az elemzés végrehajtására a következő útmutató adható.



5.65. ábra. A term osztálydiagramja bináris és unáris operátorok esetén

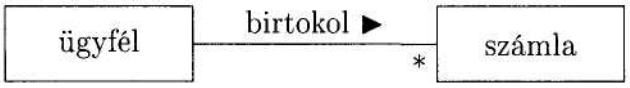
Az osztálydiagram megalkotásának lépései:

1. A követelmények elemzése alapján meghatározzuk azokat a főneveket, amelyek potenciálisan objektumjelöltek lehetnek.



2. Objektumok leírása, tulajdonságjellemzőik meghatározása. Ez tartalmazza a tulajdonságok, műveletek és relációk felderítését. Például a banki számlák esetén:

   - Tulajdonságok: tulajdonos azonosítója, számla típusa (fo lyószámla, lekötött számla), ...
   - Műveletek: felújítás, lekötés, ...



5.66. ábra. A számlák és az ügyfelek közötti kapcsolat

- Relációk: a számlát bankban tárolják, egy ügyfélnek több számlája is lehet, ...

3. Objektumok osztályba sorolása hasonló tulajdonságaik alapján.
4. Az osztályok közötti relációk meghatározása. Például a számla és az ügyfél között asszociációs kapcsolat áll fenn, amelynek neve „birtokol" (5.66. ábra).
5. Kezdeti osztálydiagram megszerkesztése.
6. Az objektumok attribútumainak, a relációk tulajdonságainak, szerepeknek, multiplicitásoknak a meghatározása.
7. Általánosítás segítségével az osztálydiagram hierarchikus szer kezetének kialakítása, áttekinthetőségének növelése, egyszerűsí tése.
8. Osztályleírások elkészítése.

5.3. feladat:

Készítsük el az irányított gráf osztálydiagramját a következő leírás alapján! Az irányított gráf csomópontokból és élekből áll. Az élek csomópontból csomópontba mutatnak, egy csomópontba több él is mutathat be, és egy csomópontból több él is mutathat ki. A bemenő él egy csomópontba, a kimenő él pedig egy csomópontból mutat be, illetve ki. Egy pont is lehet gráf. A gráf síkbeli alakzat, ahol az élet két pont koordinátáival (első honnan, második hova mutat), a csomópontot a síkbeli koordinátákkal ábrázoljuk.

Megoldás: A leírás első mondata alapján három osztályt határozhatunk meg:

- irányított gráf,
- él,
- csomópont.

Az utolsó mondat egy relációt ad rneg, a hozzá tartozó szerepekkel, illetve megadja a csomópont attribútumait. Az előző mondat tisztázza a multiplicitások kérdését. Összefoglalva:

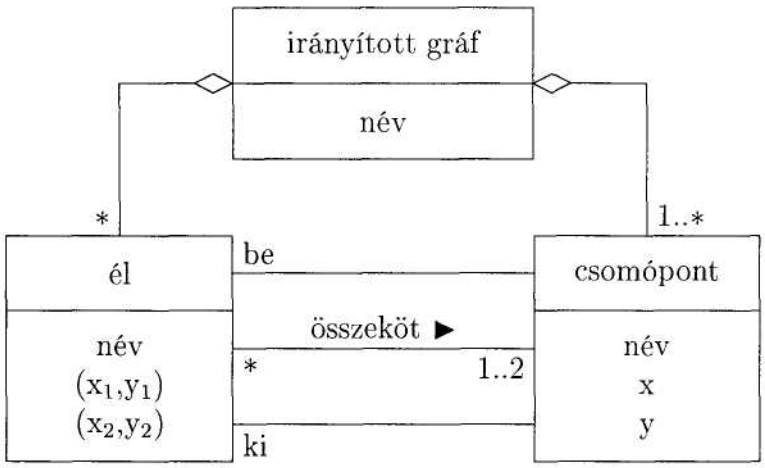
Reláció: él csomópontokat *köt össze.*

Szerep: csomópontba *be* mutat az él, illetve csomópontból *ki* mutat az él.

Multiplicitások: a bemenő és a kimenő élhez egy-egy csomópont tartozik. A hurok élek egy pontot kötnek össze önmagával. Egy izolált ponthoz nem tartozik él.

Attribútumok: név, illetve név és koordináták.

Ennek alapján egy lehetséges megoldás látható az 5.67. ábrán.



5.67. ábra. Az irányított gráf osztálydiagramja

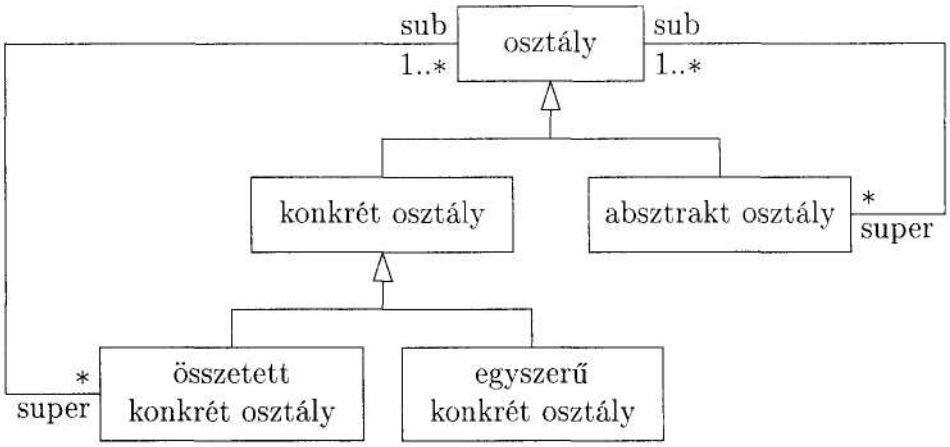
5.4. feladat:
Készítsük el az osztály fogalmának következő definíciója alapján az osztálydiagramot! Az osztály fogalmának hierarchikus szerkezetét a következő leírás alapján definiálhatjuk:

Az osztály lehet absztrakt vagy konkrét osztály. A konkrét osztály lehet egyszerű osztály, vagy lehet összetett osztály. Az absztrakt osztályból öröklődéssel származtathatók osztályok. Az összetett konkrét osztályból is származtathatók öröklődéssel osztályok. Az az osztály, amelyből egy másikat származtathatunk, „super" szerepet, az pedig, amely a származtatott, „sub" szerepet játszik az öröklődési relációban. Egy öröklődésben mindkét szerep többszörös is lehet, de absztrakt vagy összetett konkrét osztályból legalább egy osztályt származtatunk.

Megoldás: A leírás alapján az *osztályok:* osztály, absztrakt osztály, konkrét osztály, összetett konkrét osztály, egyszerű konkrét osztály.

A *relációk:* öröklődés, osztályok társítása az öröklődési reláció szerint. Itt fellép a szerep, amely sub, illetve super; és a multiplicitás, amelynek értékeit a szöveg utolsó mondata határozza meg.

A fenti elemzés alapján készült osztálydiagram látható az 5.68. ábrán.

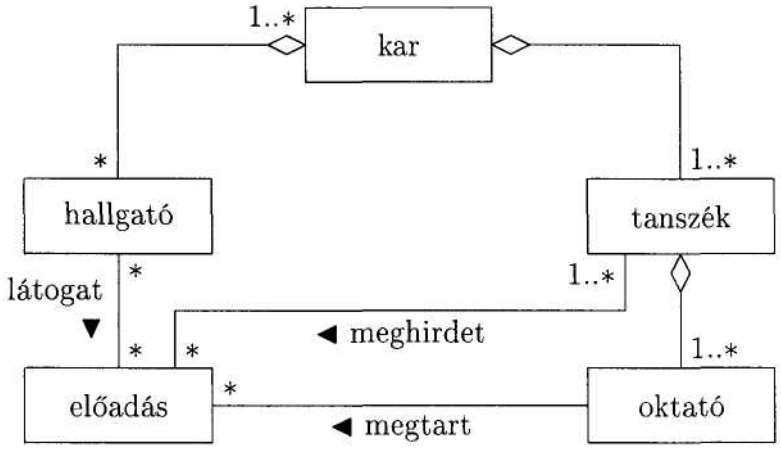


5.68. ábra. Az osztály osztálydiagramja

5.5. feladat:

Készítsük el az osztálydiagramot a következő leírás alapján! A kar tanszékekből és hallgatókból áll. Az oktatók alkotják a tanszékeket. Egy tanszék csak egy karhoz tartozhat. Minden tanszéknek van legalább egy oktatója. A hallgató több karnak is lehet hallgatója. Az oktató csak egy tanszéknek lehet az oktatója. A tanszékek hirdetik meg az előadásokat, de az oktatók tartják meg azokat. A hallgatók ezeket az előadásokat látogatják.

Megoldás: Az osztályok: kar, hallgató, tanszék, oktató, előadás. Ezek kapcsolata látható az 5.69. ábrán. A relációk és a multiplicitások adódnak a feladat szövegéből.

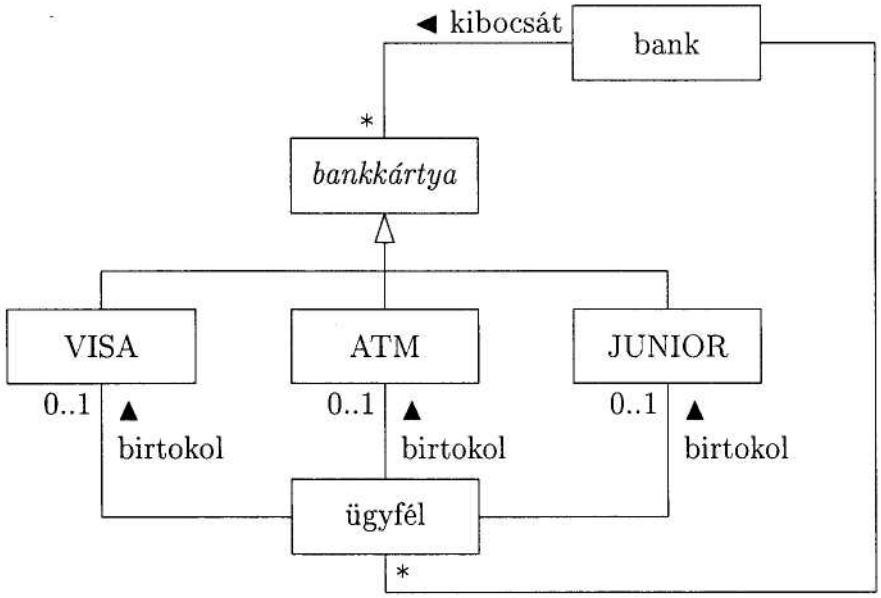


5.69. ábra. Karon belüli kapcsolatok

5.6. feladat:

Készítsük el az osztálydiagramot a következő szöveg alapján! A bank bankkártyákat bocsát ki. A bank által kibocsátott bankkártyák háromfélék: VISA, ATM, JUNIOR. Minden ügyfélnek bármelyik kártyából lehet egy kártya a tulajdonában.

Megoldás: A leírás alapján a bankkártya egy absztrakt osztály, amelynek konkrét formái a VISA, ATM, JUNIOR osztályok objektumai. A banknak a bankkártyákkal és az ügyfelekkel van asszociációs kapcsolata. A multiplicitás a szöveg alapján egyértelmű (5.70. ábra).



5.70. ábra. A bank, bankkártyák és ügyfelek kapcsolatát leíró osztálydiagram

5.7. feladat:

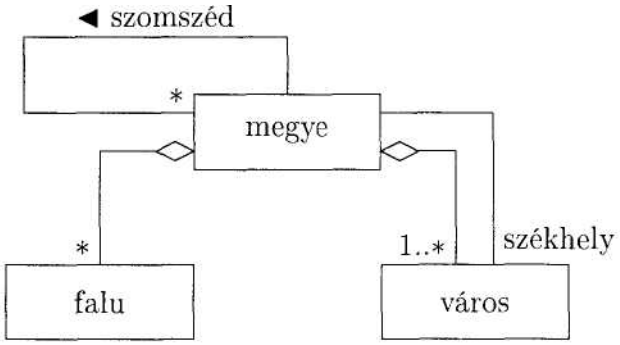
Készítsük el az osztálydiagramot a következő leírás alapján! A falvak és városok megyéket alkotnak. A megyének van székhelye, amely város. A feldolgozás szempontjából fontos, hogy mely megyék szomszédosak.

Megoldás:

- A megyék tehát falvakból és városokból állnak.
- A városok között van egy, amelynek az a szerepe, hogy székhely.
- Végül a megyék között fontos a szomszédsági reláció.

A fenti elemzés alapján az 5.71. ábrán látható osztálydiagramot kapjuk.

A feladatban nem szerepelt, de be lehet vezetni a települések osztályát, amely egy absztrakt osztály lenne, és ebből származtatnánk specializációval a falvak és a városok osztályát. A diagram megfelelő módosítása az olvasó feladata.



5.71. ábra. A megyék, falvak és városok kapcsolata

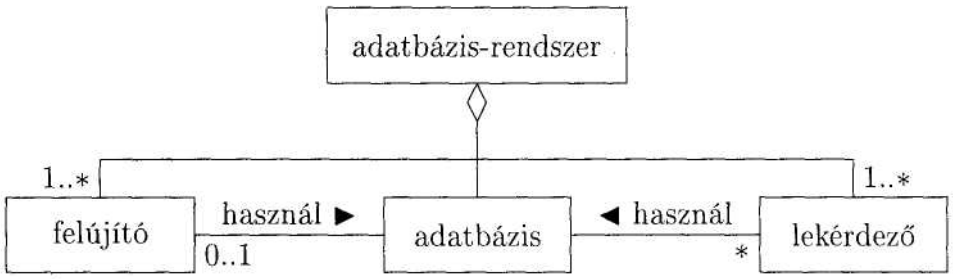
5.8. feladat:

Készítsük el az adatbázis-rendszer osztálydiagramját a következő leírás alapján! Az adatbázis-rendszer adatbázisból, az adatbázist felújító folyamatokból és az adatbázist lekérdező folyamatokból áll. Az adatbázist egyszerre csak egy felújító folyamat használhatja, a lekérdező folyamatok közül azonban az adatbázist több folyamat is használhatja egyidejűleg.

Megoldás: Az elemzés alapján az *osztályok:* adatbázis-rendszer, adatbázis, felújító, lekérdező. A *relációk:*

- Az adatbázis-rendszer aggregátuma az adatbázisnak, a felújító és a lekérdező folyamatoknak. Itt a folyamatok multiplicitása többszörös.
- A folyamatok használják az adatbázist. A szöveg alapján a multiplicitás a felújító folyamatoknál 0 vagy 1, a lekérdező folyamatoknál többszörös.

Ezek alapján az 5.72. ábrán látható osztálydiagramhoz jutunk.



5.72. ábra. Az adatbázis-rendszer osztálydiagramja

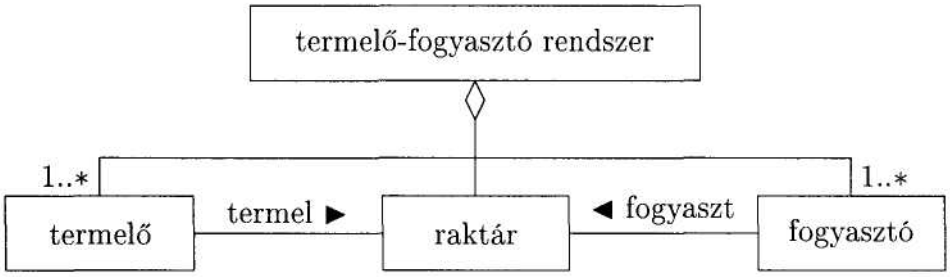
5.9. feladat:

Készítsük el a termelő-fogyasztó rendszer osztálydiagramját a következő leírás alapján! A termelő-fogyasztó rendszer egy raktárból, több termelő folyamatból és több fogyasztó folyamatból áll. A raktárban azonban egyszerre csak egy termelő helyezhet el árut, és a raktárból egyszerre csak egy fogyasztó szállíthat ki árut.

Megoldás: A szöveg elemzése alapján az *osztályok:* termelő-fogyasztó rendszer, raktár, termelő, fogyasztó. A *relációk:*

- A termelő-fogyasztó rendszer a raktár, a termelő és a fogyasztó objektumok aggregátuma. A fogyasztó és a termelő multiplici tása többszörös.
- A termelés és a fogyasztás asszociációk. A multiplicitás itt mind a két esetben egyszeres.

Így az 5.73. ábrán látható osztálydiagramhoz jutunk.



5.73. ábra. A termelő-fogyasztó rendszer osztálydiagramja

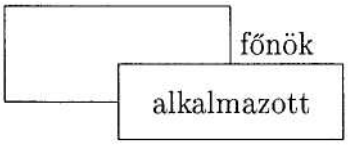
A relációk közötti összefüggéseket összegzi a következő feladat.

5.10. feladat:
Jelölje K a kompozíciók, AG az aggregációk, AS az asszociációk és S a specializációk halmazát! Írjuk fel a négy halmaz között fennálló matematikai relációkat!

Megoldás: $K \subset AG \subset AS \ \wedge \ AS \cap S = \emptyset$.

5.11. feladat:
Egy konkrét vállalkozás adatait dolgozzuk fel. Rajzoljuk fel az osztálydiagramot a következő mondat alapján! Az alkalmazottak között van egy, aki a főnök.

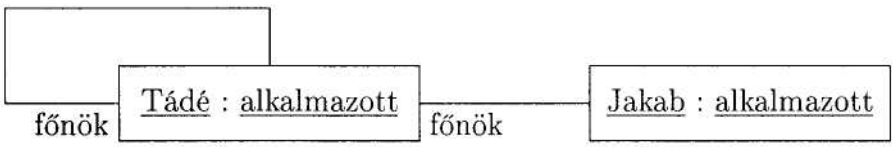
Megoldás: A megoldás az 5.74. ábrán látható.



5.74. ábra. A vállalkozás osztálydiagramja

5.12. feladat:
Tegyük fel, hogy *Tádé* a főnök, és *Jakab* egy alkalmazott! Rajzoljuk fel a fenti osztálydiagram alapján azt az objektumdiagramot, amely ezekre az objektumokra vonatkozik!

Megoldás: A megoldás az 5.75. ábrán látható.

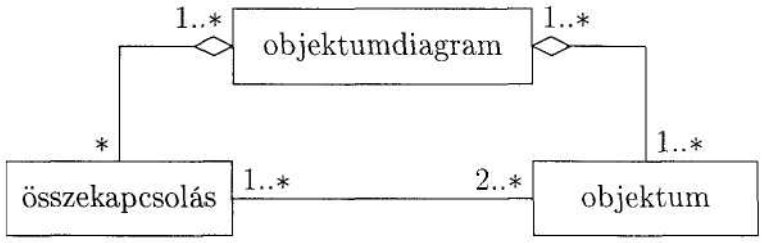


5.75. ábra. Az 5.74. ábrán látható osztálydiagramhoz tartozó objektumdiagram

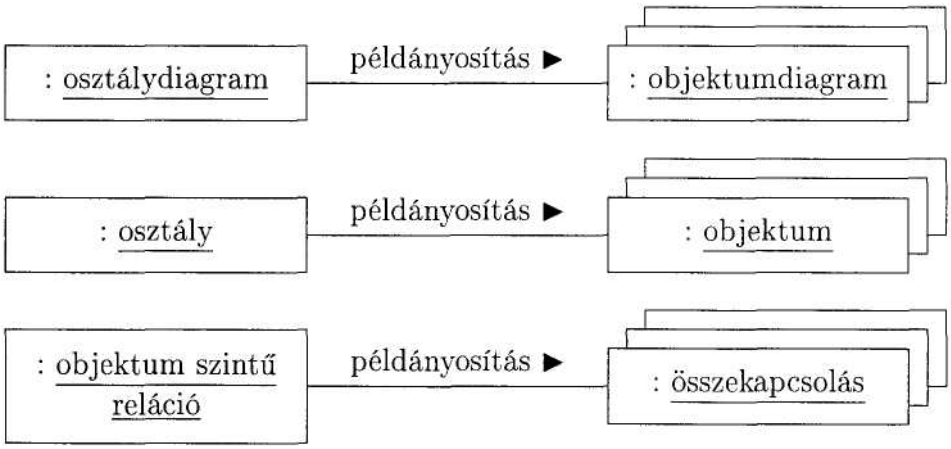
5.5.1. Az osztálydiagramok és az objektumdiagramok kapcsolata

Az előző fejezet végén megadtuk az objektumdiagram definícióját, és vázlatosan áttekintettük az objektumdiagram és az osztálydiagram kapcsolatát. Az objektumdiagram összetevőit és kapcsolataikat leírhatjuk az UML segítségével is (5.76. ábra), csakúgy mint az osztálydiagramból példányosítással történő előállítást is (5.77. ábra).

A továbbiakban először a két diagram kapcsolatát vizsgáljuk meg részletesebben, majd ezt felhasználva azzal foglalkozunk, hogy az osztálydiagram ismeretében miként határozhatjuk meg az objektumdiagramot.



5.76. ábra. Az objektumdiagram



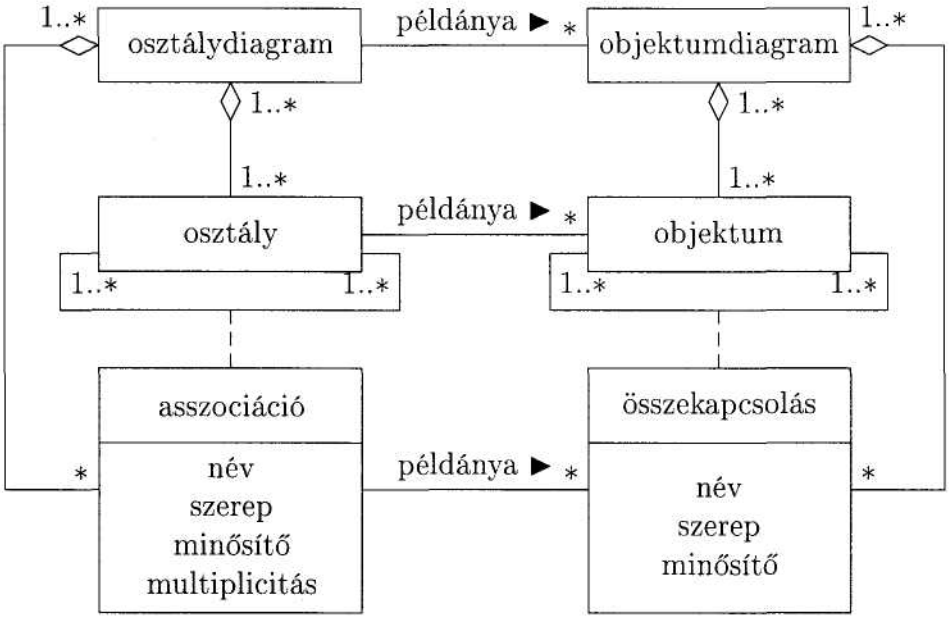
5.77. ábra. Az osztálydiagram és az objektumdiagram kapcsolata

Foglaljuk össze röviden a diagramok jellemzőit, és ennek alapján készítsük el az osztálydiagram és az objektumdiagram kapcsolatának osztálydiagramját!

Az osztálydiagram osztályokból és az osztályokat összetársító (aszszociáció) relációkból áll. Az osztálydiagram legalább egy osztályt tartalmaz. A társítás több osztályhoz több osztályt is társíthat. Ugyanaz az osztály és társítás több osztálydiagramhoz is tartozhat. A társításnak vannak tulajdonságjellemzői, ezek: név, szerep, minősítő, multiplicitás.

Egy osztálydiagramhoz több objektumdiagram tartozhat. Ennek során az osztály helyébe annak példányai, az objektumok; a társítás helyébe pedig a társítás példányai, az összekapcsolások lépnek. Az összekapcsolás átveszi a megfelelő társítás tulajdonságait (5.78. ábra).

A vállalkozásokkal kapcsolatos feladatban már láttunk egy egyszerű példát arra, hogy miként lehet osztálydiagram alapján egy konk-

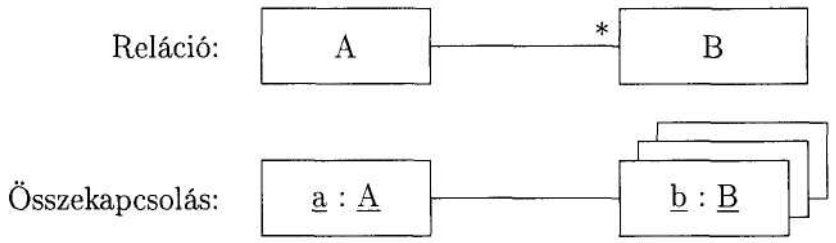


5.78. ábra. Az osztálydiagram és az objektumdiagram kapcsolata

rét esetnek megfelelő objektumdiagramot elkészíteni. Ez egy egyszerű példa a reflexív asszociációra. A továbbiakban összefoglaljuk az objektum szintű relációk példányosításával kapcsolatos tudnivalókat és ismereteket, majd ezt és az osztály- és objektumdiagram közötti kapcsolatot egyszerű példákon szemléltetjük.

Az objektum szintű relációk példányosítása:

- A reláció azonosítóját az összekapcsolás átveszi.
- A reláció multiplicitásának értékével megegyező számú objektum jelenik meg az összekapcsolásban.
- Ha a reláció multiplicitásának értéke tetszőleges érték, akkor az összekapcsolásnál ez a következő módon jelenik meg:

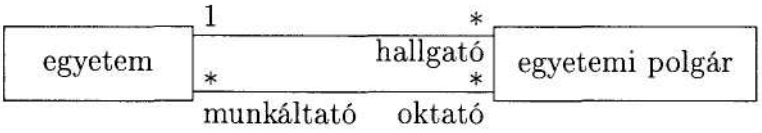


- A reláció irányának megfelelően vesznek részt az objektumok aktív, illetve passzív szereplőként az összekapcsolásban. Tehát az irányt az összekapcsolás a relációtól szemantikailag és jelölésben is átveszi.
- A reláció speciális esetének tulajdonságát az összekapcsolás sze mantikailag és jelölésben is átveszi (aggregáció, kompozíció).
- A reláció „ordered" tulajdonsága az összekapcsolásban részt vevő objektumok sorszámmal kifejezett szerepében nyilvánul meg; az „első" sorszám tetszőlegesen választható meg.
- Minősítéssel ellátott reláció esetén a minősítőnek az adott objek tum szempontjából éppen aktuális értékét kapja meg az össze kapcsolás.

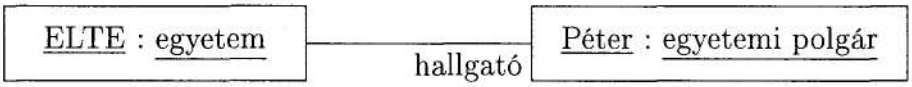
- Az öröklődési reláció az objektumdiagramban nem jelenik meg. Megjelennek viszont a konkrét objektumok mellett azok a szerepnevek, amelyek az adott aggregáció esetében az objektumok konkrét szerepét jelentik.

Első példaként tekintsük az egyetem és az egyetemi polgárok osztályát! Az egyetemi polgár lehet hallgató vagy oktató. Egy egyetemen sok hallgató és sok oktató lehet. Tegyük fel, hogy egy hallgató csak egy egyetemre járhat, egy oktató pedig több egyetemen is oktathat, és ideiglenesen lehet, hogy akár egy egyetemen sem oktat! Az egyetem az oktatók munkáltatója. Ennek a leírásnak megfelel az 5.79. ábrán látható osztálydiagram.

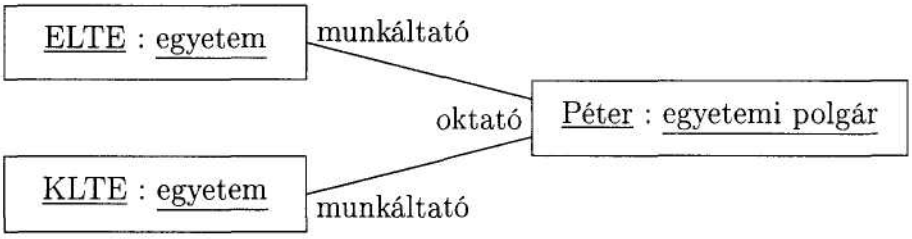
Ha Péter az ELTE hallgatója, akkor ebből az 5.80. ábrán látható objektumdiagramhoz jutunk. Ha pedig Dr. Papp az ELTE és KLTE oktatója, akkor az 5.81. ábra lesz a megfelelő objektumdiagram.



5.79. ábra. Az egyetem és az egyetemi polgárok kapcsolata



5.80. ábra. Az objektumdiagram egy hallgató esetén



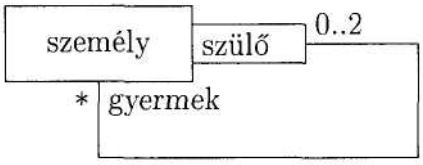
5.81. ábra. Az objektumdiagram egy oktató esetén

A szerep tehát megjelenik az objektumdiagramban is; a multiplicitás pedig konkretizálódik, azaz annyi objektum jelenik meg az összekapcsolásnál, amennyi a multiplicitás konkrét értéke az adott esetben. A mi példánkban azt néztük, hogy Dr. Papp melyik egyetemnek az oktatója, és az ábra szerint ez az ELTE és a KLTE esetében áll fenn.

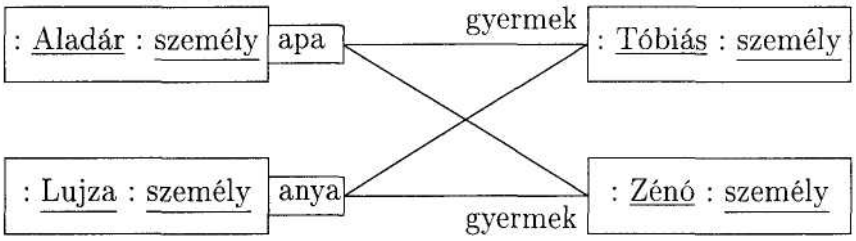
A következő példa reflexív asszociáció esetén mutatja meg az osztálydiagram és az objektumdiagram viszonyát. Az osztály legyen az élő személyek osztálya, a reflexív kapcsolat pedig a szülő-gyermek viszony! Egy személynek lehet akármennyi gyermeke, és legfeljebb két élő szülője, akik közül az egyik anya, a másik apa (5.82. ábra).

Legyen Aladár és Lujza apja, illetve anyja Tóbiásnak és Zénónak! Ekkor az 5.83. ábrán látható objektumdiagramhoz jutunk. Ez a példa szemlélteti, hogy a minősítő konkrét értéke is megjelenik az objektumdiagramban a konkrét objektum mellett, amelyet az azonosít.

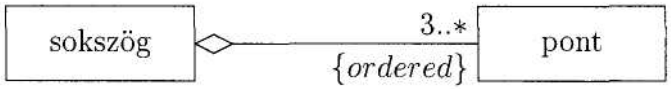
A következő példában egy olyan esetet vizsgálunk az osztálydiagram és a konkrét objektumdiagram viszonyára, amikor sorrendiségi szerep is előfordul az osztálydiagramban. Tekintsük a sokszögek és a csúcspontjaik osztályát (5.84. ábra)!



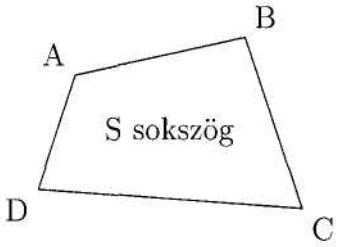
5.82. ábra. Szülő-gyermek viszony a személyek között



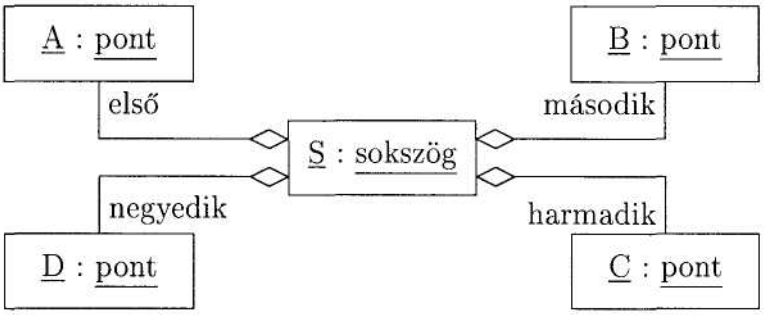
5.83. ábra. Az 5.82. ábrán látható osztálydiagramhoz tartozó lehetséges objektumdiagram



5.84. ábra. A sokszögek és a csúcspontjaik közötti reláció



5.85. ábra. Egy konkrét *S* sokszög

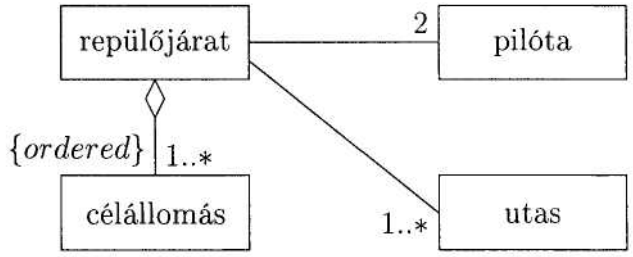


5.86. ábra. Az 5.85. ábrán látható sokszögnek megfelelő objektumdiagram

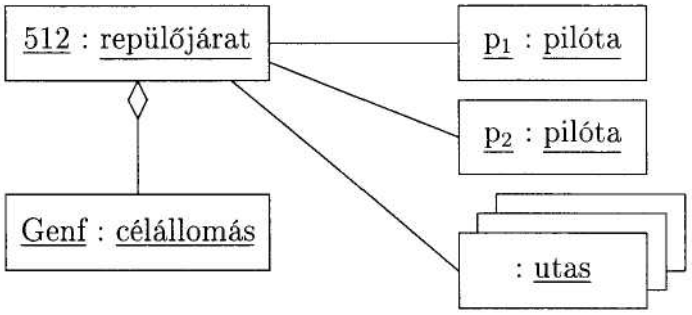
Egy konkrét *S* sokszög (5.85. ábra) esetén adott sorrend szerint külön-külön vesszük fel az objektumokat a diagramba (5.86. ábra).

Valójában itt csak az a lényeg, hogy az objektumok milyen sorrendben vesznek részt az aggregátum felépítésében, függetlenül attól, hogy melyik közülük az első sorszámmal jelölt. Az „ordered" szerep tehát egy önkényesen megválasztott sorrendnek megfelelő sorszámozást jelent az objektumdiagramban.

Utolsó példaként tekintsük a repülőjáratok 5.87. ábrán látható lehetséges modelljét! Ebben egy repülőjáratot pontosan két pilóta vezet, legalább egy utas használja, és van valahány célállomása. A célállomá-



5.87. ábra. Repülőjáratok egy lehetséges modellje



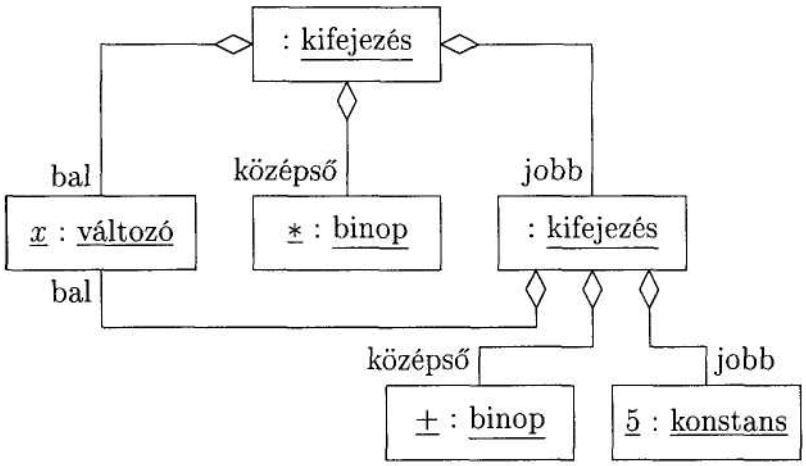
5.88. ábra. Egy konkrét repülőjárat objektumdiagramja

sok sorrendje lényeges. Egy ehhez tartozó objektumdiagram lehet az 5.88. ábrán látható diagram. Ebben csak egyetlen célállomás szerepel, ezért a sorrendiséggel nem kell foglalkozni.

5.13. feladat:
Készítsük el a konkrét objektumdiagramját az *x* * *(x* + 5) kifejezésnek a korábban felírt osztálydiagram (5.64. ábra, 135. oldal) alapján!

Megoldás: Ez tehát egy olyan term, ami kifejezés. A kifejezés bal oldalán ugyancsak egy term áll, amely viszont már változó. A jobb oldalon ismét term áll. Ez a term azonban kifejezés. Ennek a kifejezésnek a bal oldalán konstans, a jobb oldalán változó áll. Ennek az elemzésnek az alapján az 5.89. ábrán látható objektumdiagramhoz jutunk.

Az öröklődési reláció tehát az objektumdiagramban nem jelenik meg. Az objektumok mellett a konkrét osztály található. Az öröklődés osztály szintű reláció, ezért nem szerepelhet az objektumdiagramban.



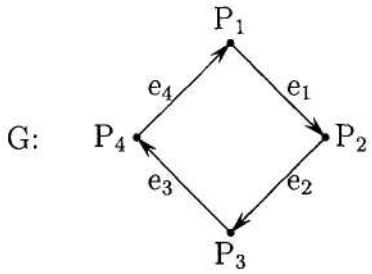
5.89. ábra. Az *x* * *(x* + 5) kifejezésnek megfelelő objektumdiagram

Megjelennek viszont a konkrét objektumok mellett azok a szerepnevek, amelyek az adott aggregáció esetében az objektumok konkrét szerepét jelentik.

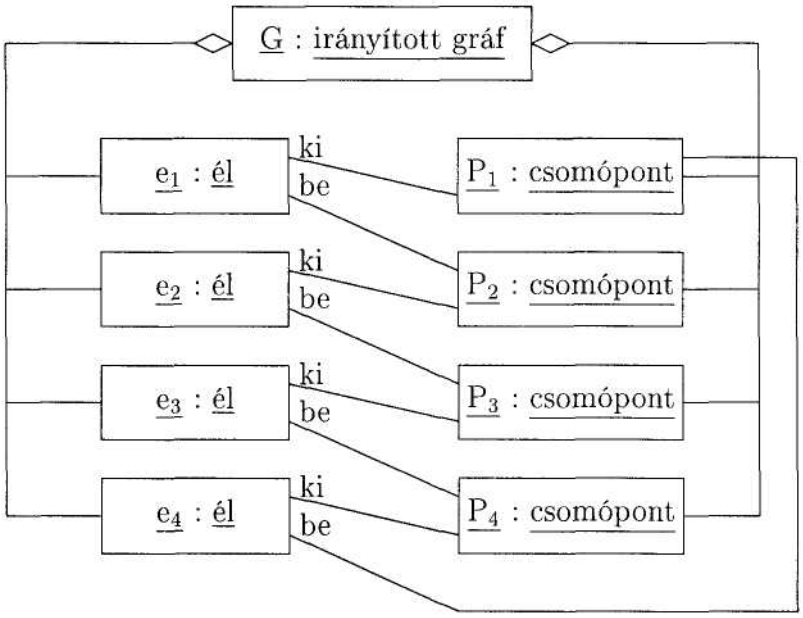
5.14. feladat:

Tekintsük az 5.90. ábrán látható konkrét gráfot, és készítsük el annak objektumdiagramját az 5.67. ábrán (138. oldal) szereplő osztálydiagram alapján!

Megoldás: Értelemszerűen az 5.91. ábrán látható diagramhoz jutunk. (Az attribútumokat nem tüntetjük fel.)



5.90. ábra. Egy konkrét irányított gráf



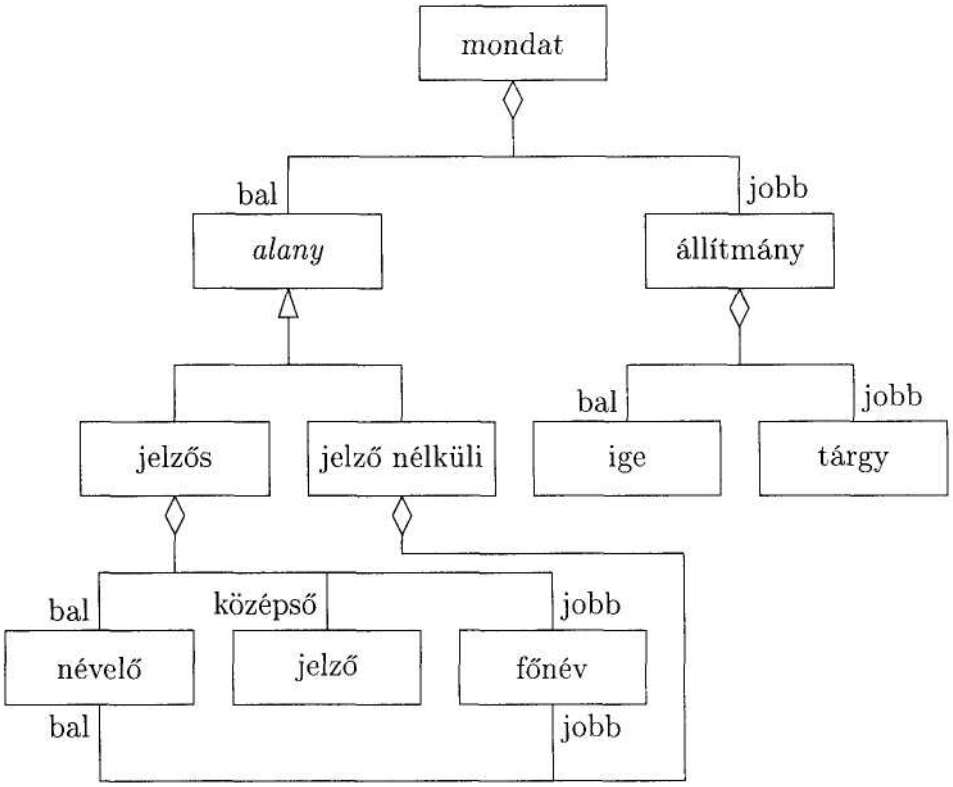
5.91. ábra. Az 5.90. ábrán szereplő irányított gráf objektumdiagramja

5.15. feladat:

Készítsük el az osztálydiagramot a következő leírás alapján! A mondat alanyból és állítmányból áll, ahol az alany az állítmányt megelőzi. Az alany lehet névelővel ellátott, jelzős vagy jelző nélküli főnév. Az állítmány igéből, és az utána álló tárgyból áll. A névelő a főnév előtt áll.

Megoldás: A szöveg elemzése alapján az objektumosztályok: mondat, alany, állítmány, jelzős (alany), jelző nélküli (alany), névelő, jelző, főnév, ige, tárgy. Ezek formális szintaktikai felírása:

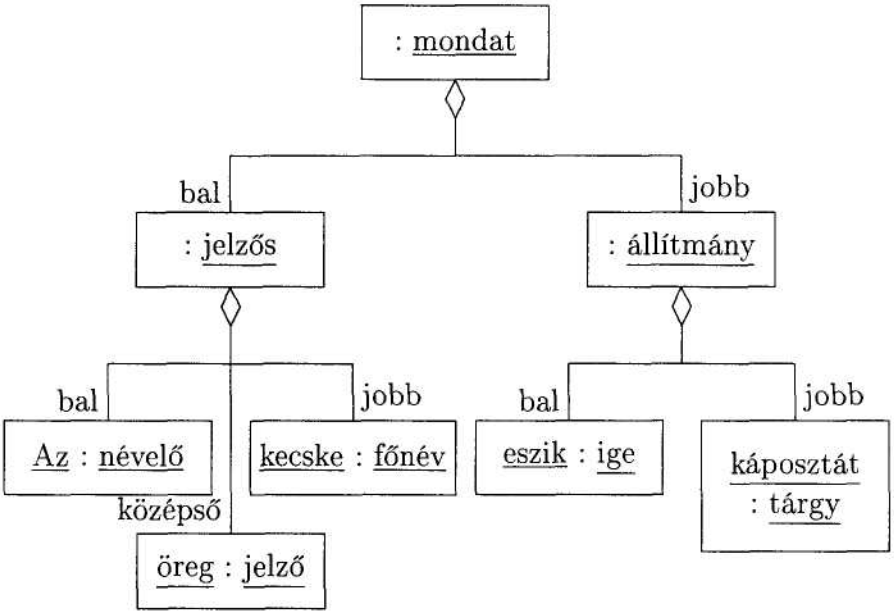
```
<mondat>          ::= <alany> <állítmány>;
<alany>           ::= <jelzős> | <jelző nélküli>;
<jelzős>          ::= <névelő> <jelző> <főnév>;
<jelző nélküli>   ::= <névelő> <főnév>;
<állítmány>       ::= <ige> <tárgy>.
```



5.92. ábra. A mondat felépítését megadó osztálydiagram

Ahogy azt már láttuk, a BNF leírásban előforduló „vagy" kapcsolatnak öröklődés, az „és" kapcsolatnak aggregáció felel meg a diagramban. Az alany osztály absztrakt osztály, amelynek konkrét esetei a jelzős, illetve jelző nélküli osztályok. Ennek alapján az 5.92. ábrán szereplő osztálydiagramhoz jutunk.

5.16. feladat:
Az előző feladatban előállított osztálydiagram alapján készítsük el a következő mondat objektumdiagramját! **Az öreg kecske eszik káposztát.**



5.93. ábra. „Az öreg kecske eszik káposztát." mondatnak megfelelő objektumdiagram

Megoldás: Először azonosítsuk a mondat alkotóelemeit!

```
     Az : névelő;
   öreg : jelző;
 kecske : főnév;
  eszik : ige;
káposztát : tárgy.
```

Ezt felhasználva az 5.93. ábrán látható objektumdiagramhoz jutunk.

5.17. feladat:
Készítsük el az osztálydiagramot a következő leírás alapján!

A program blokkokból áll. A blokk utasítások egy véges sorozata. Az utasítás lehet értékadás, feltételes elágazás vagy ciklus. A ciklus egy logikai kifejezésből és az utána álló blokkból áll. Az elágazás egy

logikai kifejezésből és az azt követő értékadásból, valamint az értékadást követő blokkból áll. A logikai kifejezéshez és az értékadáshoz változókat és konstansokat használunk fel.

Megoldás: A szöveg alapján a következő osztályokat azonosíthatjuk: program, blokk, utasítás, feltételes elágazás, értékadás, ciklus, logikai kifejezés, változó, konstans. Relációk:

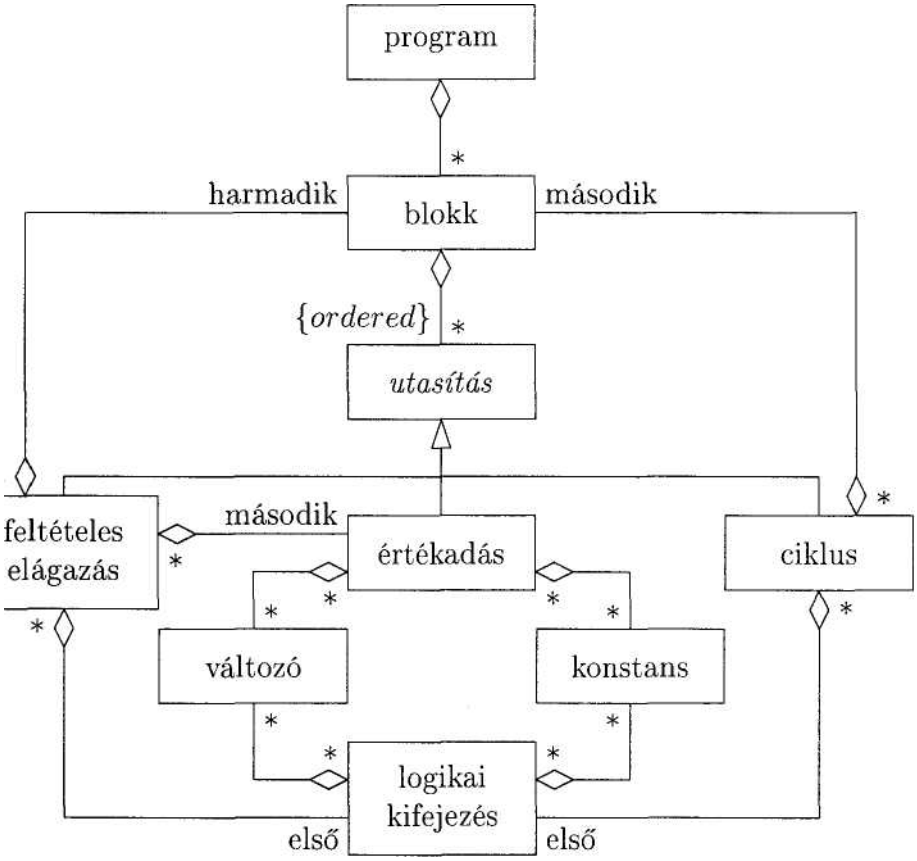
- Az utasítás egy általánosítás - absztrakt osztály -, amelynek konkrét megjelenési formái a felsorolt utasítások.
- Az értékadás és a változó, illetve az értékadás és a konstans kö zött a szöveg alapján aggregáció azonosítható.
- A logikai kifejezés és a változó, illetve a logikai kifejezés és a konstans között szintén aggregáció áll fenn.
- Az osztályok között fennálló további kapcsolat mind aggregáció, ahol a komponensek sorrendje lényeges.

Szerepek:

- Az utasítások komponensei sorrendben első, második, illetve har madik szerepet vállalva alkotják az utasítást.
- A blokk utasítások véges sorozata. Az utasítások tehát sorrend ben egymás után állva képeznek blokkot. Nincs megadva, hogy hányan. Tehát csak azt írhatjuk, hogy „sorrendben". Erre szolgál az *ordered.*

Multiplicitások: A szövegben szereplő többes számok alapján felismerhető, hogy az aggregátumoknak hány komponensük van. Ugyanakkor egy komponens tetszőleges számú aggregátum része lehet a program definíciója miatt.

A fenti elemzés alapján az 5.94. ábra osztálydiagramjához jutunk.



5.94. ábra. A program osztálydiagramja

5.18. feladat:

Készítsük el az előző feladatban megadott osztálydiagram alapján az

```
S :   begin
      x ← 5;
      if x < y then
          y ← x
      else
```

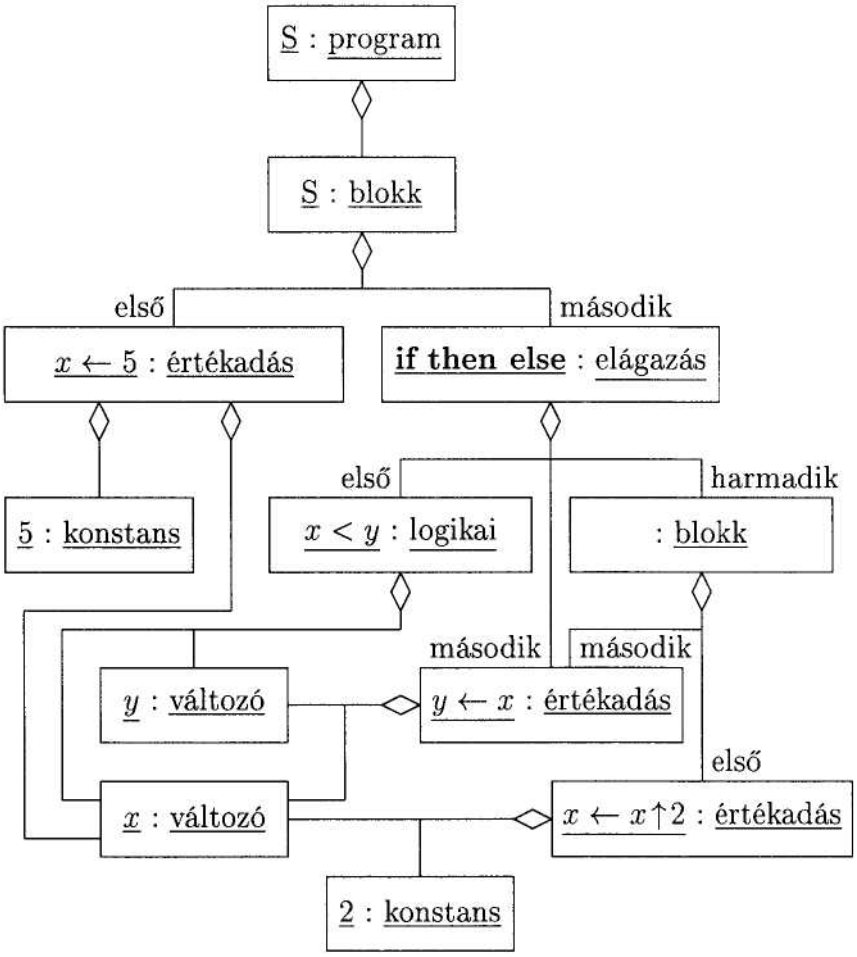
```
        begin
            x ← x ↑ 2;
            y ← x
        end
    fi;
    end;
```

program objektumdiagramját, ahol **begin** ... **end** a blokkot jelenti!

Megoldás: Ha a megfeleltetéseket a következők szerint alkalmazzuk, az 5.95. ábrán szereplő objektumdiagram adódik eredményül.

- A program egyetlen blokkból áll.
- A blokk két utasítást tartalmaz, amelyek közül az első az $x \leftarrow 5$ értékadás, a második egy feltételes elágazás.
- A feltételes elágazás logikai kifejezése az $x < y$ feltétel, a második tagot adó értékadás az $y \leftarrow x$, a harmadik tag egy blokk.
- A harmadik tagot alkotó blokk két értékadást tartalmaz, amelyek közül az $x \leftarrow x \uparrow 2$ az első, az $y \leftarrow x$ a második tag. A második tag megegyezik a feltételes elágazás értékadásával.
- A programban az x, y változók és a 2, 5 konstansok szerepelnek az értékadásokban, illetve a logikai kifejezésben.

Az objektumdiagramban az „első", „második" neveket használtuk a sorrendiségi szerep meghatározásához. Ezek a nevek megegyeznek az osztálydiagramban található szerepnevekkel, de ez nem okoz problémát.



5.95. ábra. Az „S" program objektumdiagramja. (A feltételes elágazást „elágazás"-ként, a logikai kifejezést „Logikai"-ként rövidítettük.)

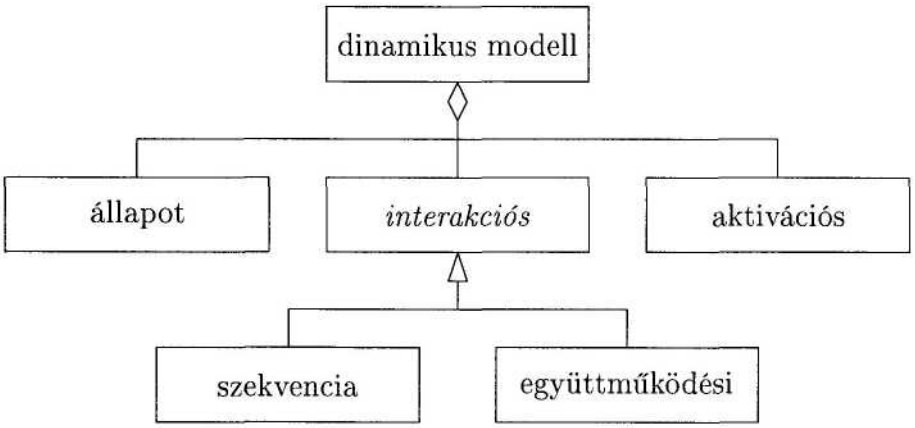
6. Állapotdiagram

Az előző fejezetben megismerkedtünk a statikus modell alkotóelemeivel. Ebben a fejezetben az objektumelvű modellezés egy másik fontos elemének, a dinamikus modellnek a bemutatását kezdjük el.

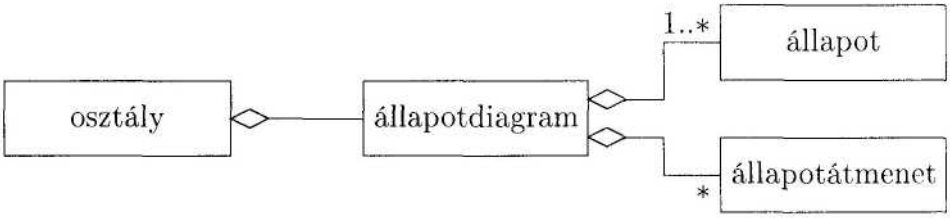
A dinamikus modell három fő részből tevődik össze: az állapotdiagramokból, az állapotok leírásából és az események leírásából. Azaz:

dinamikus modell = állapotdiagramok+
állapotok leírása +
események leírása.

Az események leírására szolgál a szekvencia-, együttműködési és aktivációs diagram. A szekvencia- és az együttműködési diagram az objektumok interakcióját határozza meg (6.1. ábra).



6.1. ábra. A dinamikus viselkedést leíró diagramok csoportosítása



6.2. ábra. Osztály és állapotdiagram kapcsolata

Az osztály objektumainak dinamikus viselkedését a probléma megoldása során az állapotdiagram írja le. Az állapotdiagram egy állapotautomatát ábrázol. Az osztály dinamikus tulajdonságai alapján egy állapotautomata, amely állapotoknak és állapotátmeneteknek az összessége (6.2. ábra).

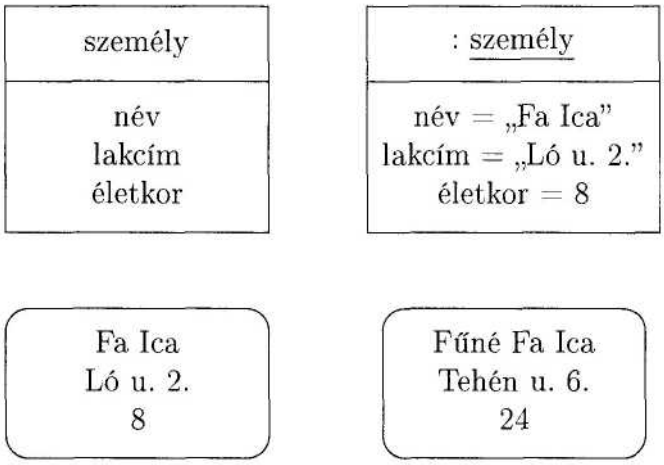
A dinamikus modell leírásának értelmezéséhez be kell vezetnünk néhány alapfogalmat és a hozzájuk tartozó jelöléseket. A bevezetendő alapfogalmak:

- állapot,
- esemény + jelölésrendszer,
- állapotdiagram.

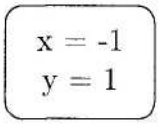
Minden objektumnak van egy életciklusa. Az objektum létrejön, aktív életet él, azaz más objektummal működik együtt különböző feladatok megoldásában, majd megsemmisül. Számos esetben azonban végtelen folyamatot vizsgálunk. Ilyenkor az objektum megsemmisülése nem játszik szerepet a vizsgálatban. Az objektum aktív állapotaiban különböző objektumokkal kerül kapcsolatba, és ennek eredményeként különböző *állapotokba* kerül.

Állapot: Az objektum állapotát az attribútumok konkrét értékeinek n-esével jellemezzük (6.3. ábra).

Ugyanígy ha síkbeli pontokat koordinátáikkal jellemzünk, akkor egy pont lehetséges állapota látható a 6.4. ábrán, amely eltolás vagy forgatás hatására megváltozhat.



6.3. ábra. Példa az objektumok állapotának jellemzésére. A felső részen látható az osztály és az objektum jelölése, alul pedig egy konkrét objektum két állapota (lánykori és házasság utáni).



6.4. ábra. Egy pont állapota

Esemény: Eseménynek nevezzük azt a tevékenységet, történést, amely valamely objektum állapotát megváltoztatja.

Eseményre példa lehet az előző esetekben a házasságkötés vagy az eltolás, forgatás, illetve a vektorokra értelmezett *put* művelet, amely a vektor adott indexű elemét megváltoztatja:

put : *vektor* x *index* x *elem* —> *vektor* ,

vagy ha az útkereszteződésnél a lámpa pirosra vált. (Ekkor ugyanis az útkereszteződésnek mint objektumnak az állapota megváltozik.)

Az objektum állapota rendszerint nem egy adott pillanatban képezi a vizsgálat tárgyát. A személyekre vonatkozó példa esetén az lehet érdekes számunkra, hogy az adott személy általános iskolás-e, azaz

objektumunk olyan állapotban van-e. Ezt az állapotot egy időintervallummal jellemezhetjük:

7 < életkor < 14.

Általában az osztály objektumainak viselkedése kapcsán az azonos állapotban lévő objektumokat egy halmazba vonjuk össze, és állapotdiagramjaikat egy, az osztály objektumainak életciklusát reprezentáló diagrammal adjuk meg. Ezt fejezi ki a következő definíció.

Osztályhoz rendelt állapot: Azokhoz az objektumokhoz rendelt állapotokat, amely objektumok az eseményekre, az események egy bizonyos halmazára azonos módon reagálnak (pl. nem engedik meg magukra, vagy fordítva), egy halmazba vonjuk össze, és ezt a halmazt osztályhoz rendelt állapotnak nevezzük. *Az osztályhoz rendelt állapot tehát az állapotoknak egy halmaza, amely az állapotinvariánssal jellemezhető:*

<állapot neve> = {(attri,..., attr„) | /(attri,..., attr„)} .

Például a verem egyik állapota a normál állapot, amelynek az invariánsa:

Például a verem egyik állapota a normál állapot, amelynek az invariánsa:

$$0 < length(v) < n .$$

Mi a következőkben állapot alatt osztályhoz rendelt állapotot értünk.

Az állapotok és az események időhöz vett viszonyát a következőképpen jellemezhetjük (6.5. ábra):

- az állapothoz időintervallum tartozik,
- az eseményhez pedig időpont tartozik.

Mi a következőkben állapot alatt osztályhoz rendelt állapotot értünk.

Az állapotok és az események időhöz vett viszonyát a következőképpen jellemezhetjük (6.5. ábra):

- az állapothoz időintervallum tartozik,
- az eseményhez pedig időpont tartozik.

6.5. ábra. Az állapot és az esemény viszonya az időhöz

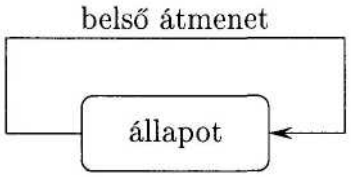
6.1. Az állapot informális definíciója

Az állapot a következő tulajdonságokkal rendelkezik:

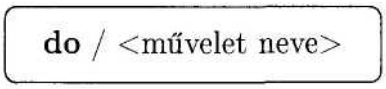
1. Az állapotnak van azonosítója. Az állapotok egymástól megkülönböztethetők, például van nevük. A verem esetében: „normál", „tele", ... Nem csak név azonosíthat egy állapotot. Lehet az azonosító egy vagy több attribútumának konkrét értéke, illetve ezen értékeket meghatározó állítás, állapotinvariáns. Az állapot jelölése egy lekerekített sarkú téglalap, amelybe az azonosító kerül, ahogy az például a 6.3. ábrán látható.

2. Az állapot általában esemény, eseménysorozat hatására jön létre. Ezeket az eseményeket megelőző eseményeknek (pre-events) nevezzük. Például a normál állapot a verem esetében a „create", „push" események eredményeként jön létre először; az iskolás állapot pedig a beiratkozás hatására. Egy speciális, rendszeren kívüli állapot a kezdeti állapot.

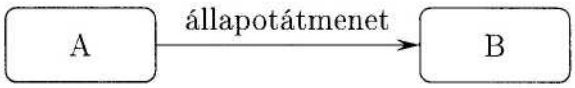
3. Az állapot időben mindaddig fennmarad, amíg az objektumok attribútumainak értékei kielégítik az állapothoz rendelt invariánst. Az állapot fennállása során belső átmenetek fordulhatnak elő. Ezek nem változtatják meg az objektum állapotát, azaz érvényes marad továbbra is az állapotinvariáns:



4. Az állapotokat gyakran nehéz megfelelően kifejező névvel ellátni. Ezért gyakran azonosítóként használjuk a belső tevékenységek, belső műveletek nevét. Ennek jelölése:



5. Az állapot megszűnése egy esemény hatására következik be. A megszűnéshez egy eseménysorozat kötődhet (rákövetkező események, post-events). Azaz az objektum egy külső állapotátmenet hatására egy másik állapotba kerül. Jele:



6. Az objektum megszűnése ugyancsak egy állapotátmenet hatására következik be. Ekkor az objektum egy rendszeren kívüli, úgynevezett befejező állapotba kerül.

Ennek a definíciónak az alapján az állapotot a következő formáknak alávetett leírással adhatjuk meg:

state: <az állapot azonosítója>;

comment: <az állapot rövid magyarázó leírása>;

pre-events: <az állapotot előidéző, megelőző események azonosítóinak listája>;

invariant: <az állapot invariánsának leírása>;

post-events: <az állapot megszűnéséhez kötődő, rákövetkező események listája>.

Példaként írjuk fel ennek megfelelően az ébresztőóra „csengetés" állapotát!

state:
ébresztő csengetés;

comment:
amikor az előzetesen beállított időpont bekövetkezik, a csengő megszólal, és 10 másodpercig csörög;

pre-events:
idő beállítás (ébresztési idő);
minden más esemény, ami nem „idő beállítás törlés";
aktuális idő = ébresztési idő;

invariant:
idő beállítás bekapcsolva és
ébresztési idő < aktuális idő < ébresztési idő + l0mp;

post-events:
aktuális idő = ébresztési idő +10mp —> visszaállítás (ébresztési idő).

6.2. Az esemény informális definíciója

Az eseményeket a következők jellemzik:

1. lehet paraméter nélküli (például az „enter" gomb megnyomása);
2. lehet paraméteres (például vermek esetén a *push(s,e));*
3. az események között sorrendiség állhat fenn, azaz beszélhetünk:
   - megelőző eseményről, illetve
   - rákövetkező eseményről;
4. az eseménynek előfeltétele is lehet.

Az esemény előfeltételét megadhatjuk például az esemény paramétereire értelmezett feltétellel: *{pre(a)} a′ = f(a)*.

Általában az események a mindennapi életben is sorrendben egymáshoz kötődnek, például:

- reggel kimegyek az utcára (megelőző esemény);
- hőmérséklet fagypont alatti (feltétel);

- felhúzom a kesztyűmet (esemény).

Az esemény formáknak alávetett leírása a fentiek alapján a következő lehet:

event: <esemény neve, azonosítója>;

comment: <az esemény jelentésének rövid leírása>;

parameters: <a paraméterek listája>;

precondition: <az esemény bekövetkezését szükségszerűen megelőző állítás >;

pre-events: <az eseményt megelőző esemény>. Ennek alapján

az esemény általános formája:

<esemény>(<paraméterek>)[<feltétel>]/<megelőző esemény>,

ahol a feltétel olyan állítás, amely vonatkozhat

- paraméterekre (előfeltétel), formája:

  {pre(a,p)} a' = f(a,p);

- a megelőző eseményekre (szinkronizációs feltétel).

Példaként tekintsük a MALÉV MA-416-os London-Prága-Budapest járatát, az esemény legyen a gép megérkezése Budapestre! Ekkor:

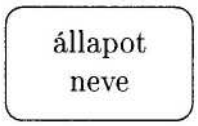
megérkezik[Londonból felszállt^Prágába leszállt^Prágából felszállt].

Egy másik példa lehet a vermek esetén a „push" művelet:

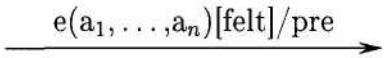
push(v,e) [0 < *length(v)* < *n]*.

6.3. Az állapotdiagram definíciója

Az eddigiek felhasználásával definiálhatjuk az állapotdiagram fogalmát. Az állapotdiagram egy összefüggő irányított gráf, amelynek csomópontjaihoz az állapotokat rendeljük, éleihez pedig az eseményeket. A rendszer működésének vizsgálata során csak olyan állapotokat vizsgálunk, amelyek létrejönnek, azaz elérhetők a kezdő állapotból, ezért a gráf összefüggő. Ugyanakkor egy állapotátmenet több esemény hatására is létrejöhet, ezért két csúcsot több irányított él is összeköthet. Az állapot jelölése a 6.6. ábrán, az élekhez rendelt átmeneteket okozó események jelölése - általános formában - a 6.7. ábrán látható. Ezenkívül fontos még a rendszeren kívüli állapotok jelölése, amely a 6.8. ábrán látható.



6.6. ábra. Állapot jelölése az állapotdiagramban

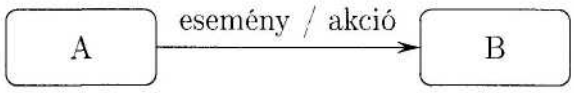


6.7. ábra. Állapotátmenetet okozó esemény jelölése az állapotdiagramban

6.8. ábra. A speciális rendszerállapotok, kezdő és befejező állapot, jelölése az állapotdiagramban

6.4. Esemény és akció

Az eseményt mi úgy definiáltuk, hogy az egy időpillanat alatt játszódik le. Számos esetben azonban az esemény végrehajtása időben elhúzódhat. Ilyenkor célszerű megkülönböztetni az eseményt és az akciót egymástól. Ekkor az *akció* az, ami egy időpillanathoz kötődik. Az állapotok közötti átmenetet tehát esemény vagy akció eredményezheti (6.9. ábra).



6.9. ábra. Az állapotátmenetet általában vagy esemény, vagy akció eredményezi

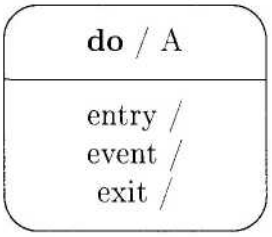
Az állapotok közötti átmenet, a tranzakció valójában egy relációt fejez ki két állapot között. Az első állapotban az objektum kezdeményezi a kilépés akcióját, amely elindítja azt az eseményt, eseménysorozatot, amely a második állapotba történő belépést eredményezi. Ha az első állapotból történő kilépés és a másodikba történő belépés ugyanaz az időpillanat, akkor akcióról beszélünk, ellenkező esetben eseményről. Események esetén célszerű az eseményhez állapotot rendelni.

Valójában arról van szó, hogy az eseményeknek általában három fázisuk van, amelyek közül kettő akció:

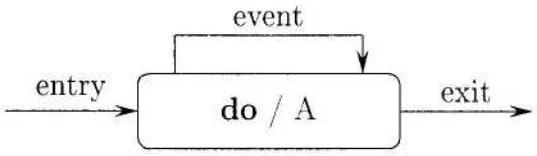
- Egy *entry fázis,* a belépés akciója, amely elindítja azt az ese ményt, eseménysorozatot, amelynek hatására létrejön egy ese ményhez rendelt állapot.

- Egy *event fázis,* amely az adott állapothoz kötődik, azaz belső események sorozata, amely az adott állapothoz kötődő belső ál lapotokat jelenti.

- Egy *exit fázis,* a kilépési akció, amely az esemény befejezését, a hozzá rendelt állapotból való kilépést eredményezi.



6.10. ábra. Az esemény három fázisának jelölése

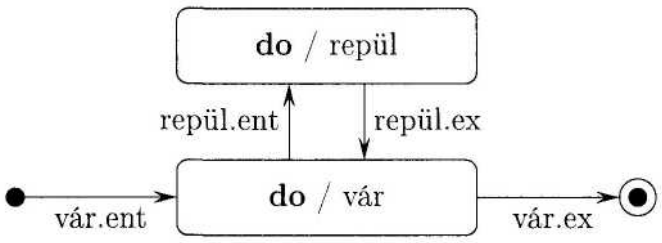


6.11. ábra. Az esemény folyamatosságának szemléltetése

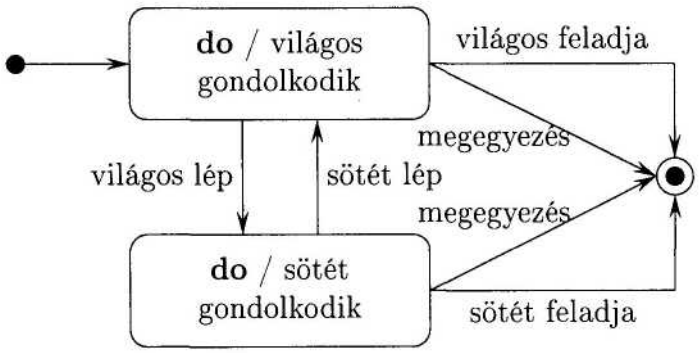
Ezt szemlélteti a 6.10. ábra, ahol az eseményhez rendelt állapot neve „do / A".

Az esemény folyamatosságát a 6.11. ábrával is szemléltetni kívánjuk.

Példaként tekintsük a repülőjárat-nyilvántartásnak egy leegyszerűsített modelljét (6.12. ábra)! A repülők a regisztráció után a felszállásra várnak, repülnek, a következő állomáson várnak, stb., az utolsó állomáson várnak, majd befejezik a tevékenységet, kijelentkeznek. A várakozás és a repülés időben elhúzódhat, azaz ezek események. A regisztráció felel meg a várakozás belépési akciójának, a kijelentkezés



6.12. ábra. Repülőjárat-nyilvántartás állapotdiagramja



6.13. ábra. Egy sakkjátszma lefolyásának állapotdiagramja

pedig a kilépési akciónak. A repül esemény entry fázisa a felszállás, exit fázisa pedig a leszállás. Az ábrában az entry fázist „ent"-ként, az exit fázist „ex"-ként rövidítettük. A továbbiakban is ezt a jelölési konvenciót használjuk.

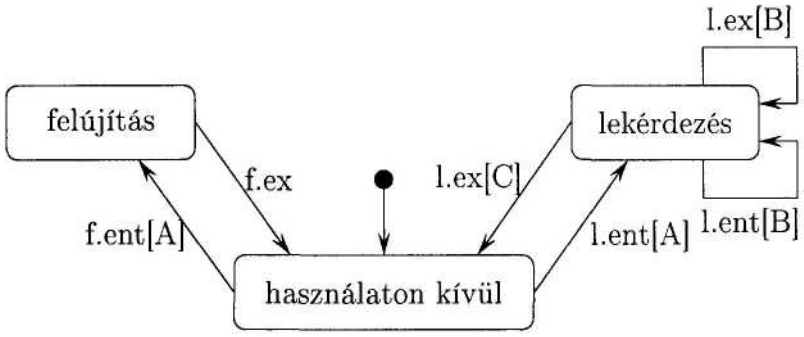
A következő példa egy sakkjátszma nyilvántartása, és az állapotdiagram a játszma lefolyásának legabsztraktabb szintjét mutatja be (6.13. ábra).

Összetettebb példaként készítsük el az előző fejezet 8. feladatához (143. oldal) tartozó állapotdiagramot! Emlékeztetőül a feladat szövege: Egy adatbázis-rendszer adatbázisból, az adatbázist felújító folyamatokból és az adatbázist lekérdező folyamatokból áll. Az adatbázist egyszerre csak egy felújító folyamat használhatja, lekérdező folyamatok közül azonban az adatbázist több folyamat is használhatja egyidejűleg.

Az osztálydiagram az 5.72. ábrán (143. oldalon) látható. Ezt felhasználva határozzuk meg a rendszer lehetséges állapotait!

- Egy felújító kizárólagosan használja. Állapot neve: *felújítás.*
- Egy vagy több lekérdező használja, kizárva annak használatából a felújítókat. Állapot neve: *lekérdezés.*

  Sem felújító, sem lekérdező nem használja az adatbázist. Állapot neve: *használaton kívül.*



6.14. ábra. Az adatbázis-kezelő rendszer állapotdiagramja. A felújítót „f"-fel, a lekérdezőt „l"-lel rövidítettük.

A rendszer állapotai között az állapotátmenetek: felújító$_i$.ent, felújító$_i$.ex; lekérdező$_i$.ent, lekérdező$_i$.ex; ahol: i = 1,..., n, j = 1,..., m; és n a felújítók, m pedig a lekérdezők száma. A felújítókra a diagramban röviden „f"-fel, a lekérdezőkre pedig „l"-lel hivatkozunk majd.

Így a 6.14. ábrán látható állapotdiagramhoz jutunk, ahol az állapotátmenetek feltételei:

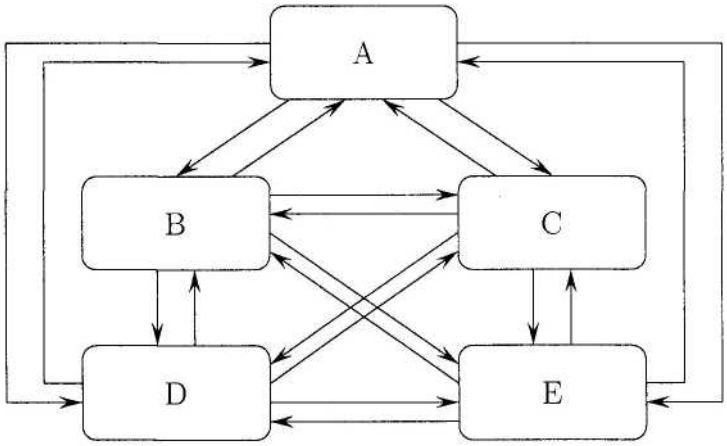
A: sem felújító, sem lekérdező nem használja az adatbázist; B:

felújító nem, de lekérdező használja az adatbázist;

C: sem felújító, sem lekérdező nem használja a folyamaton kívül az adatbázist (ő az utolsó).

6.5. Az állapotdiagram bonyolultsága

Az állapotdiagram viszonylag egyszerű esetekben is áttekinthetetlenné válhat. Tekintsük azt az állapotdiagramot, amelynél 5 állapot van, és mindegyik állapotból mindegyik másik állapotba történhet átmenet (6.15. ábra)!



6.15. ábra. Öt állapot közötti összes átmenet

Az ábra már ebben az esetben is áttekinthetetlen, bonyolult. Általános esetben, *n* állapot esetén, ilyenkor az állapotátmenetek száma *n* * (n — 1). Noha ennek a szélső esetnek az előfordulási valószínűsége elenyésző, az állapotok számának növekedésével az állapotdiagram áttekinthetetlenné válik. Ezért szükséges az így előálló bonyolultságot kezelnünk, nevezetesen egyszerűbbé és áttekinthetőbbé tenni az állapotdiagramot.

A bonyolultság csökkentésére két általános módszer létezik:

- az állapotok általánosítása és
- az állapotok aggregációja.

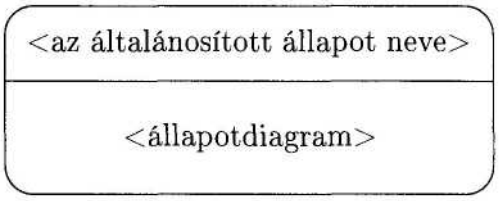
Mindkét esetben állapotokat vonunk össze egyetlen állapotba, és az állapotdiagramban az összevont állapotot tüntetjük fel. Az összevont állapotot megadhatjuk akár függetlenül, külön diagramban is. Így egy adott szinten egyrészt az állapotok száma csökken, másrészt az összevont állapotok belső átmenetei sem jelennek meg. A következőkben ezeket a módszereket tárgyaljuk.

6.5.1. Állapotok általánosítása

Az általánosított állapot a következő tulajdonságokkal rendelkezik:

1. Az általánosított állapot véges számú részállapot összessége.
2. A részállapotok örökölhetik az általánosított állapot tulajdonsá gait:
   a. attribútumait (állapotjellemzőit);
   b. eseményeit, akcióit.
3. Az általánosított állapot állapotinvariánsa a részállapotok álla potinvariánsainak diszjunkciója, azaz az objektum az általáno sított állapotban mindig valamelyik részállapotban van.
4. A részállapotok lehetnek általánosított állapotok is.
5. Az általánosított állapothoz állapotdiagram tartozik, amelyben a részállapotok közötti átmeneteket tüntetjük fel.
6. Az általánosított állapot állapotdiagramjában a részállapotok hoz azok az állapotátmenetek tartoznak, amelyek az általánosí tott állapotot nem változtatják meg.
7. Az állapotdiagramban kell lennie legalább egy olyan részállapot nak, amely az általánosított állapot *„entry"* akcióját örökli. Az állapotdiagramban a kezdő állapotból a megfelelő részállapotba vezet él.
8. Ha az általánosított állapot rendelkezik *„exit"* akcióval[1], akkor az állapotdiagramban kell lennie legalább egy olyan részállapotnak,

[1] Vannak olyan esetek, amikor a rendszer egy „végtelen folyamat", azaz nem fejeződik be. (Például folyamatirányító rendszerek.) Bizonyos objektumok életciklusa megegyezik a rendszerével, ezért azok sem szűnnek meg létezni, azaz mindig valamilyen állapotban vannak. Ekkor a megfelelő állapotáltalánosításhoz nem tartozik *„exit"* akció.



6.16. ábra. Az általánosított állapot jelölése

amely az általánosított állapot *„exit"* akcióját örökli. Minden olyan állapot, amelyhez az általánosított állapotot megváltoztató átmenet tartozik, örökli ezt az akciót. Az állapotdiagramban ilyen részállapotból vezet él a befejező állapotba.

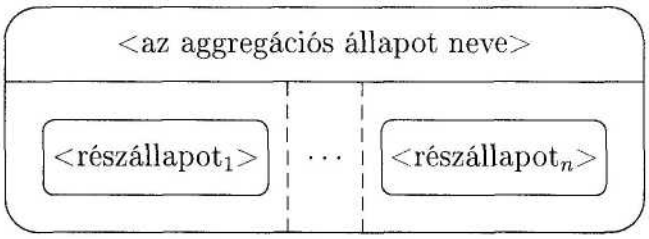
9. Az általánosított állapot megfelelő objektuma az *„entry"* akció hatására létrejön, a megfelelő objektum pedig az állapotdiagram *„exit"* akciójának hatására semmisül meg.

Az általánosított állapot jelölése a 6.16. ábrán látható.

6.5.2. Állapotok aggregációja

Az aggregációval előállított állapot a következő tulajdonságokkal rendelkezik:

1. Az aggregációval létrejövő állapot egymástól független részálla potok egy véges halmaza.
2. Minden részállapothoz állapotdiagram tartozik.
3. A részállapotok lehetnek általánosított állapotok is.
4. Minden részállapothoz tartozó állapotdiagramban kell lennie leg alább egy olyan állapotnak, amelybe a részállapot az aggregátum *„entry"* akciójának hatására kerül, azaz örökli ezt az akciót. A megfelelő állapotdiagramban a kezdő állapotból ilyen állapotba vezet él.



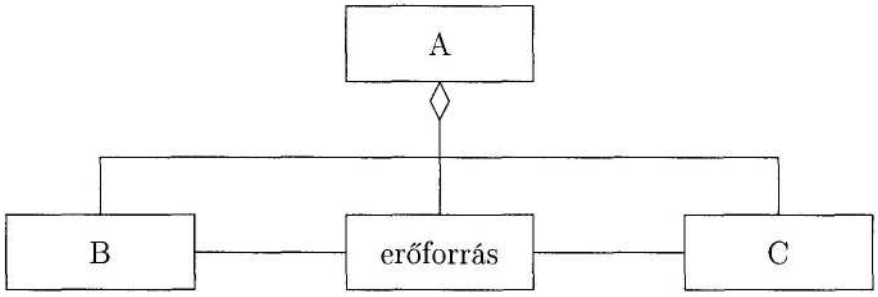
6.17. ábra. Az aggregációs állapot jelölése

5. Ha az aggregátum rendelkezik *„exit"* akcióval, akkor minden részállapothoz tartozó állapotdiagramban kell lennie legalább egy olyan állapotnak, amely az aggregátum *„exit"* akcióját örökli. A megfelelő állapotdiagram ilyen állapotából vezet él a diagram befejező állapotába.

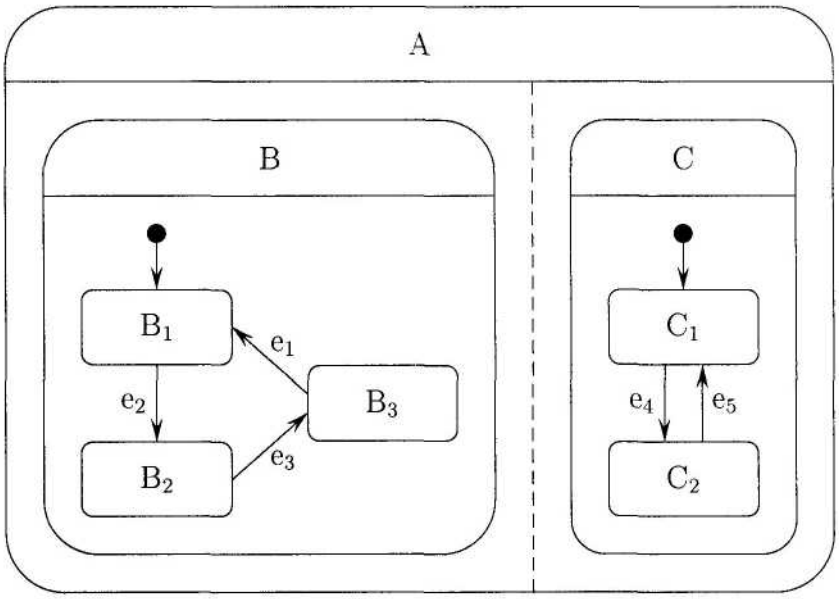
6. Az aggregációval létrejött állapot invariánsa a részállapotok in variánsainak konjunkciója, azaz az aggregációs állapot objek tuma az aggregációt alkotó részállapotokban egyidejűleg létezik.

7. Az állapotok aggregációja az állapoton belüli állapotdiagramok közötti párhuzamosság egy megjelenési formája.

Az állapotaggregáció jelölése a 6.17. ábrán látható. Példaként tekintsük a 6.18. ábrán látható aggregációt! Ehhez tartozhat a 6.19. ábrán szereplő aggregációs állapot, ha B és C az osz-



6.18. ábra. Egy egyszerű aggregáció

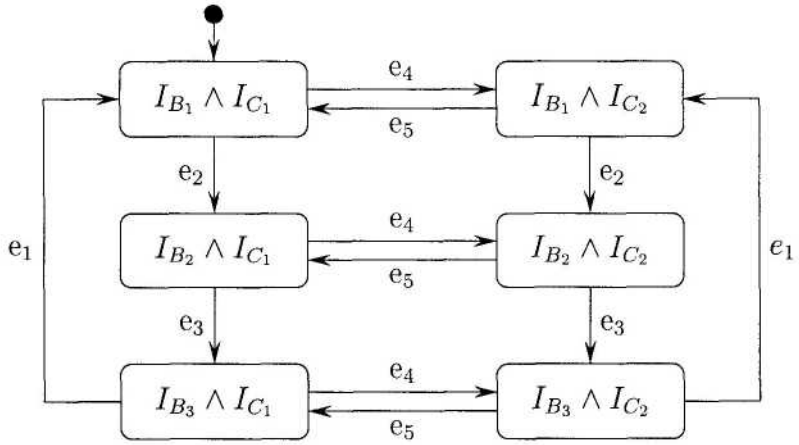


6.19. ábra. A 6.18. ábra aggregációjának megfeleltethető állapotdiagram

tálydiagramban szereplő megfelelő osztályokhoz tartozó állapotáltalánosítások.

Jelölje Ix_k az X_k állapot állapotinvariánsát. Ekkor a 6.19. ábrán szereplő aggregátum a 6.20. ábra állapotdiagramjával ekvivalens. (Például az I_{B_X} ^ I_{C_1} állapot megfelel annak, hogy B a B_1 és C a C_1 állapotban van egyidejűleg.) Látható, hogy az állapotaggregáció és az általánosítás használatával mennyivel áttekinthetőbb állapotdiagramhoz jutottunk még ebben az egyszerű esetben is.

Az aggregációban részt vevő komponensek azonban rendszerint nem egymástól függetlenül, hanem egymással együttműködve vesznek részt a feladat megoldásában. Például az előző példában szereplő állapotdiagram lehet egy kölcsönös kizárási modell, amelyben a B osztálybeli objektumnak elsőbbsége van a C osztálybeli objektummal szemben, amikor az erőforráshoz akar hozzáférni. Ekkor például lehetnek B állapotai a következők:



6.20. ábra. A 6.19. ábrával ekvivalens állapotdiagram

B_1 használja az erőforrást, B_2:

nem használja az erőforrást, B_3:

igényli az erőforrást;

C állapotai pedig a következők:

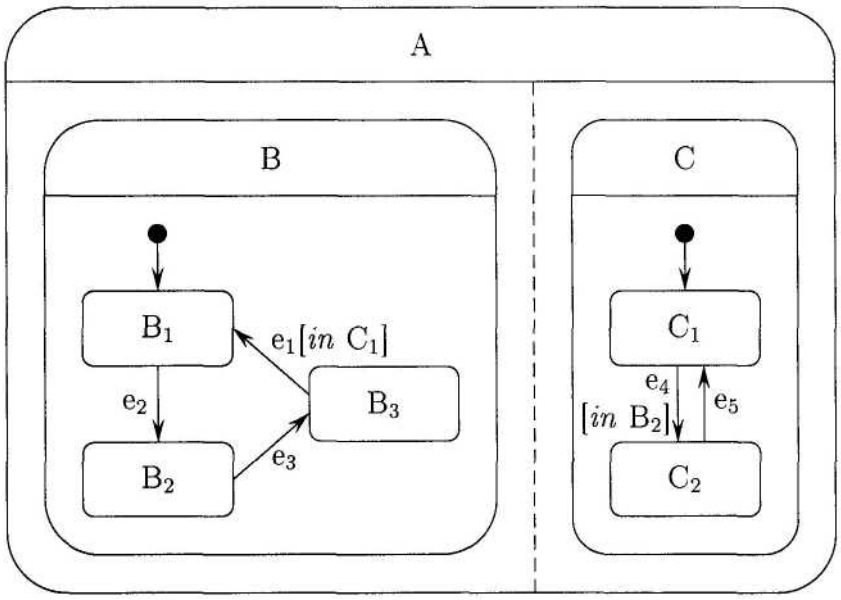
C_1: nem használja az erőforrást,

C_2: használja az erőforrást. Az állapotátmenetekre pedig a

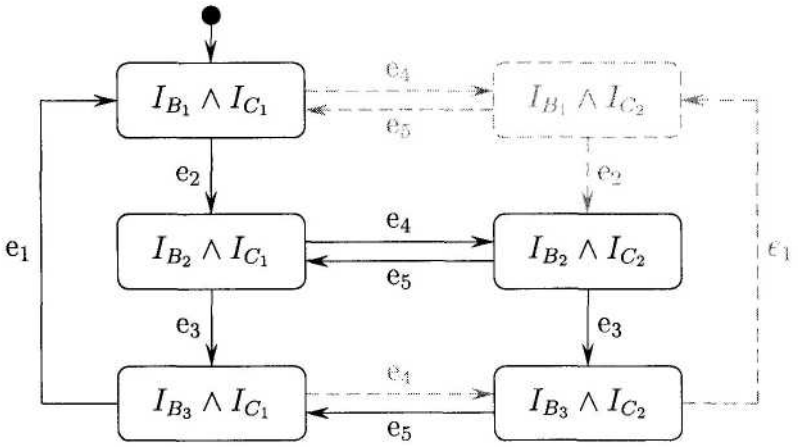
következő korlátozások állnak fenn:

- e_1[C_1 állapotban van C],
- e_4[B_2 állapotban van B].

Ha a jelölésünk következetes, akkor „C_1 állapotban van C" helyett írhatjuk röviden: *„in* C_1". Ekkor az állapotaggregáció használatával a 6.21. ábrán látható diagram lesz az eredmény, a részletes állapotdiagramot pedig a 6.22. ábra szemlélteti.



6.21. ábra. A 6.19. ábrának megfelelő állapotdiagram kölcsönös kizárás esetén

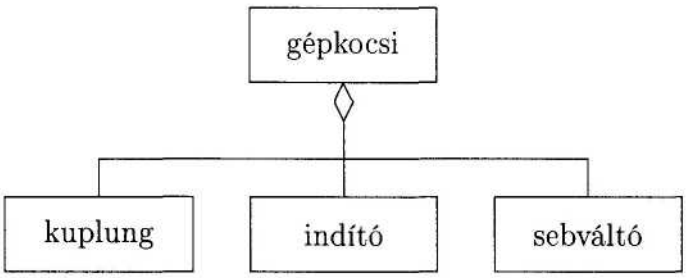


6.22. ábra. A 6.21. ábrának megfelelő részletes állapotdiagram. Szaggatott vonallal és szürke színnel jelöltük azokat az állapotokat és átmeneteket, amelyek nem kerülnek be a diagramba a kölcsönös kizárási feltétel miatt.

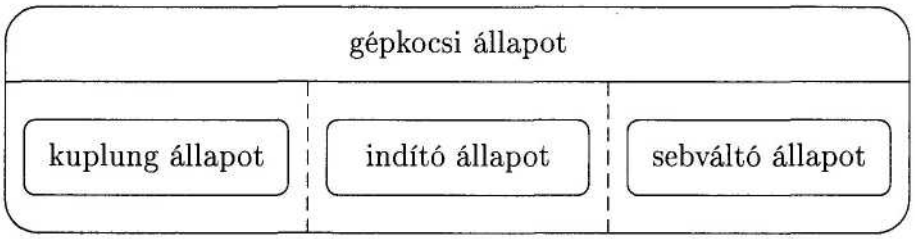
Egy lehetséges példa állapotaggregációra a gépkocsi, ami - leegyszerűsítve - álljon a következő egységekből: kuplung, indító és sebességváltó (röviden sebváltó) (6.23. ábra). Ekkor a gépkocsi állapotát leírhatjuk ezen három összetevő állapotainak aggregációjaként (6.24. ábra).

Ezek után csak az aggregációban szereplő általánosított állapotokat kell megadnunk, azaz:

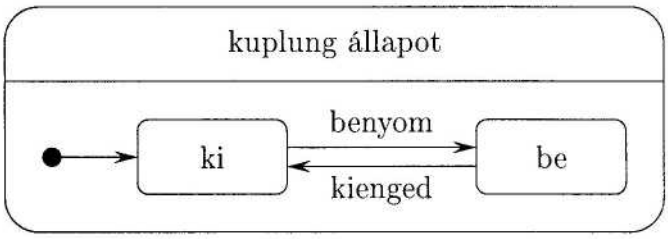
- a kuplung állapotát, amelynek konkrét állapotai a benyomott állapot és a kiengedett állapot (6.25. ábra);
- az indító állapotát, amelynek konkrét esetei legyenek például az üres, az indítás és a menet állapot (6.26. ábra);
- és a sebváltó állapotát, amely általában három állapot általá nosítása: ezek az üres, a hátramenet és az előremenet állapotok (6.27. ábra).



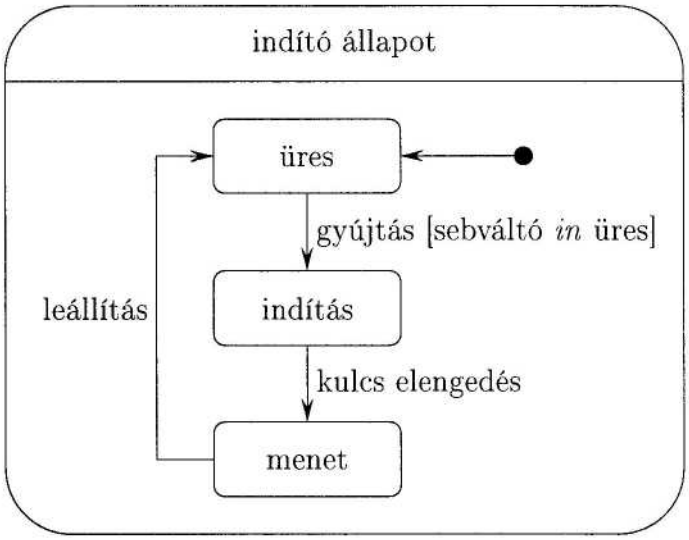
6.23. ábra. A gépkocsi osztály



6.24. ábra. A gépkocsi állapotainak megadása aggregációval

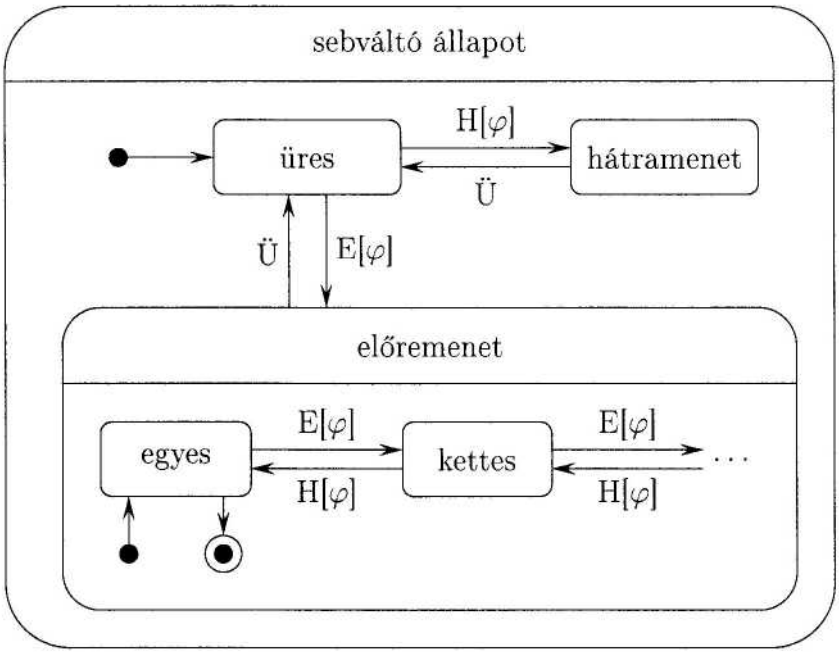


6.25. ábra. A kuplung állapotdiagramja



6.26. ábra. Az indító állapotdiagramja

A sebváltó előremenet állapota szintén állapotáltalánosítás, mivel konkrét esetei az egyes, kettes, ... menetek. Az egyszerűség érdekében tegyük fel, hogy adott fokozat esetén csak a következő, illetve az előző fokozatba tudunk váltani! Az események legyenek „E" és „H"! Természetesen nincs akadálya annak, hogy az összes előremeneti állapot örökölje az „előremenet" általánosított állapot belépési és kilépési eseményeit, noha az ábrán csak az egyes fokozat esetén tüntettük ezt fel. Sebességet csak benyomott kuplunggal lehet váltani, ezért minden ilyen esemény („E", „H") feltétele: kuplung *in* be. A sebességváltót



6.27. ábra. A sebességváltó állapotdiagramja (φ = kuplung *in* be)

bármikor üresbe lehet kapcsolni („Ü" esemény).

Térjünk vissza a már tárgyalt adatbázis-rendszer problémájához! (Előző fejezet 8. feladat, 143. oldal, illetve ez a fejezet 172. oldal.) Módosítsuk a feladatot úgy, hogy ha egy felújító folyamat használni akarja az adatbázist, akkor annak elsőbbsége van a lekérdező folyamatokkal szemben!

A megoldás során az osztálydiagram nem változik, csak az állapotdiagram, hiszen a kiegészítés a rendszer működését érinti, és az állapotdiagram írja le azt.

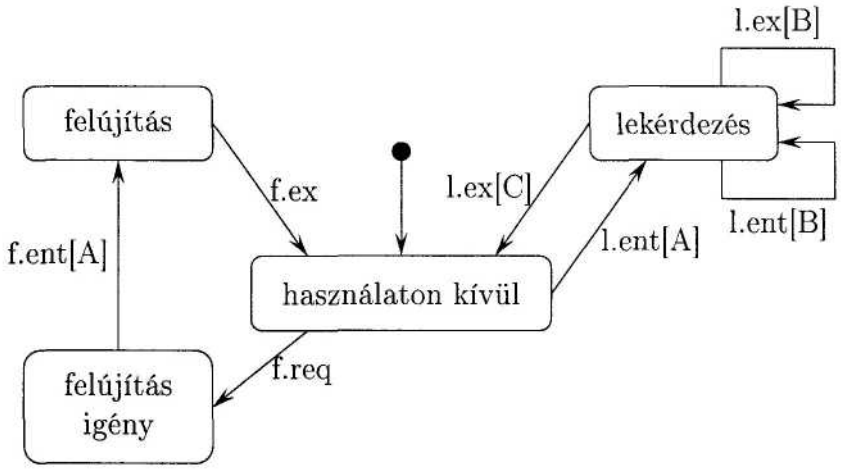
Az első megoldásban próbáljuk az előzőleg létrehozott állapotdiagramot (6.14. ábra, 173. oldal) módosítani! Az elsőbbség biztosítása érdekében be kell vezetnünk egy új eseményt és egy új állapotot az eddigiek mellé. Az új eseményben a felújító folyamatok igénylik az adatbázis használatát. Az esemény neve legyen „req". Az állapotok:

- *felújítás:* egy felújító kizárólagosan használja az adatbázist;
- *felújítás igény:* egy vagy több felújító igényli a használatot (ez az új állapot);
- *lekérdezés:* egy vagy több lekérdező használja, kizárva a haszná latból a felújítókat;
- *használaton kívül:* sem felújító, sem lekérdező nem használja az adatbázist.

A rendszer állapotai között az állapotátmenetek: felújító$_1$.ent, felújító$_1$.ex, felújító$_1$.req; lekérdező$_j$.ent, lekérdező$_j$.ex; ahol: i = 1,..., n; j = 1,..., m, és n a felújítók, m pedig a lekérdezők száma. A felújítókra a diagramban röviden „f-fel, a lekérdezőkre pedig „l"-lel hivatkozunk majd.

Így a 6.28. ábrán látható állapotdiagramhoz jutunk, ahol az állapotátmenetek feltételei nem változtak az előzőhöz képest.

Az állapotaggregációt felhasználva egy másik megoldáshoz jutunk. A prioritásos adatbázis-rendszer állapotát tekintsük a felújítók és a



6.28. ábra. Az adatbázis-kezelő rendszer állapotdiagramja. A felújítót „f"-fel, a lekérdezőt „l"-lel rövidítettük.

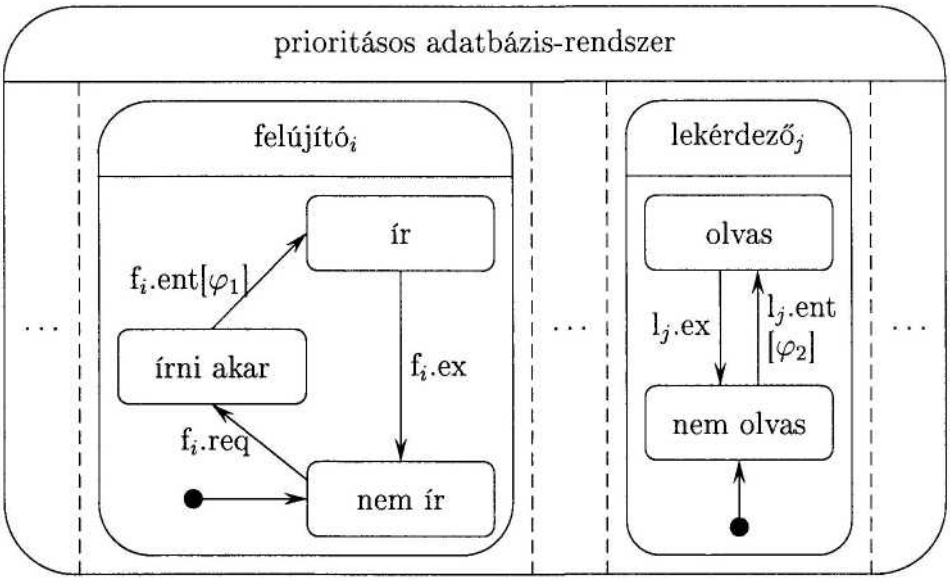
lekérdezők aggregátumának (6.29. ábra)! A felújítók esetében 3 állapotot különböztethetünk meg: nem ír, írni akar és ír. Lekérdezők esetén két állapot lehetséges: olvas és nem olvas. Az állapotátmenetek feltételei értelemszerűen adódnak.

- Az *i.* felújító akkor kezdhet írni, ha egyetlen lekérdező sem olvassa, és egyetlen felújító sem írja az adatbázist, azaz

$$\varphi_1 = \forall j \in [1..m] : \text{lekérdező}_j \; in \text{ nem olvas} \wedge \forall k \in [1..n] : \neg(\text{felújító}_k \; in \text{ ír}) \; .$$

- A *j.* lekérdező akkor olvashat, ha egyetlen felújító sem ír, és nem is akar írni, azaz

$$\varphi_2 = \forall i \in [1..n] : \text{felújító}_i \; in \text{ nem ír} \; .$$



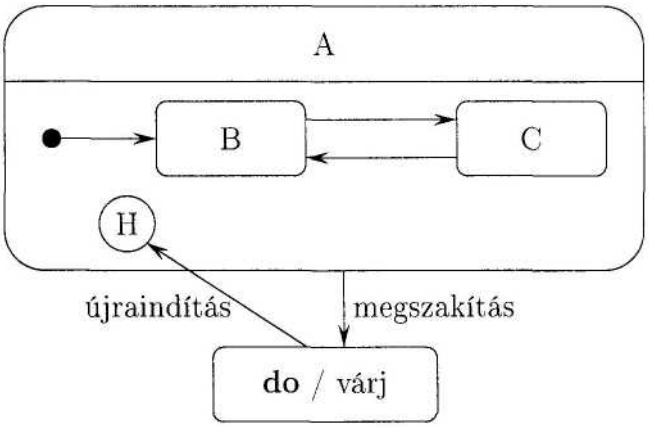
6.29. ábra. Az adatbázis-kezelő rendszer állapotdiagramja. Az eseményekben a felújítót „f"-fel, a lekérdezőt „l"-lel rövidítettük.

Az állapotdiagram bonyolultságának kezelésére bevezetett állapotáltalánosítás és állapotaggregáció miatt általánosabban kell megfogalmaznunk az állapot fogalmát, mint ahogy azt az informális definíció során tettük. Az ott leírtakat ki kell egészítenünk, és az eddigi rendszerállapotokat bővítenünk kell egy újabbal, amint azt a definíció utáni példában látni fogjuk. Ezen állapotokat a továbbiakban pszeudoállapotoknak nevezzük majd.

6.6. Az állapot általános fogalma

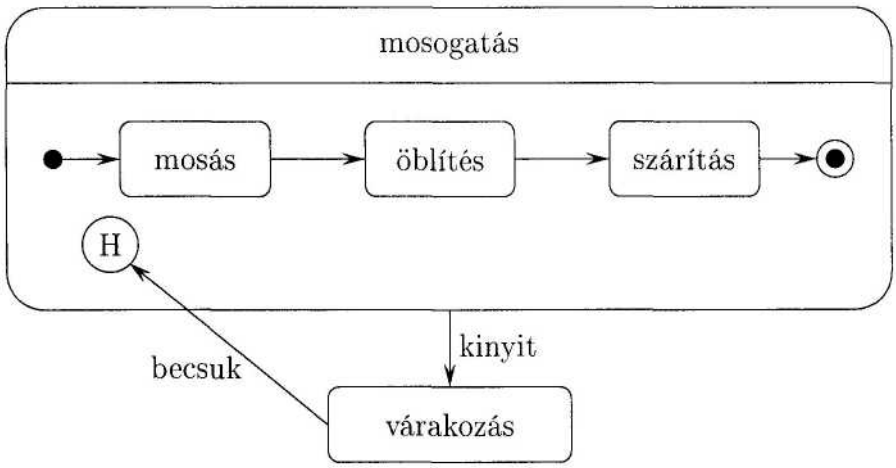
1. Az állapotnak van azonosítója.
2. Esemény hatására következik be.
3. Az állapot mindaddig fennmarad, amíg az objektumok attribú tumai az állapot invariánsát kielégítik.
4. Az állapot megszűnéséhez eseménysorozat kötődik.
5. Az állapot lehet részállapotok általánosítása. Ilyenkor az álla pothoz állapotdiagram is kötődik.
6. Az állapot lehet más állapotok aggregációja.
7. Az állapot lehet pszeudoállapot:
    a. Kezdéskor a külső állapot.
    b. Befejezéskor a külső állapot.
    c. Hisztorizációs állapot, melyhez hisztorizációs indikátor tár sul.

Az állapot hisztorizációs indikátorának, illetve a hisztorizációs állapotnak a jelölése a 6.30. ábrán látható. Ez lehetőséget ad arra, hogy egy adott pillanatban felfüggesszük az állapotokat, majd bizonyos várakozás után újra a felfüggesztés pillanatától folytatódjon az állapotok változása.



6.30. ábra. Az állapot hisztorizációs indikátora és a hisztorizációs állapot

Példaként tekintsük a mosogatógép működését! A mosogatás állapota a mosás, öblítés és szárítás állapotok egymásutánja. Azonban ha kinyitjuk a gépet, akkor az aktuális tevékenység felfüggesztésre kerül, és ha egy bizonyos idő eltelte után becsukjuk a gépet, akkor pontosan onnan folytatódik minden, ahol megszakadt (6.31. ábra).



6.31. ábra. A mosogatógép állapotdiagramja

6.1. feladat:

Készítsük el a következő leírás alapján a bróker rendszer állapotdiagramját, felhasználva az állapotaggregációt és -általánosítást!

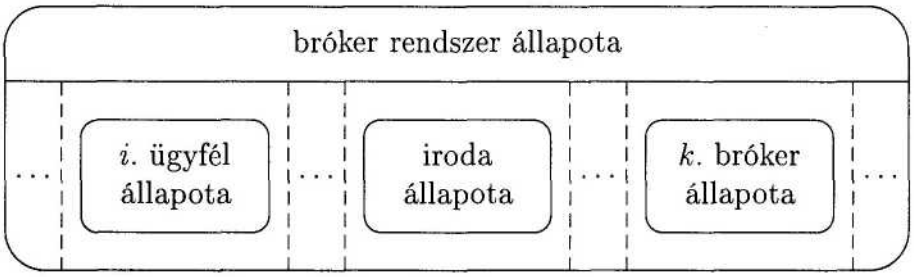
A brókerirodába egymás után érkeznek az ügyfelek, hogy megbízásokat adjanak le részvények vásárlására. A brókerek ezeket a megbízásokat folyamatosan teljesítik.

A brókerirodában két alkalmazott veszi át a megbízásokat, és adja tovább a brókereknek. Az alkalmazott egyszerre vagy egy ügyféllel, vagy egy brókerrel foglalkozik. Amíg az alkalmazottól egy megbízást bróker nem vesz át, addig az alkalmazott újabb megbízást nem vesz át ügyféltől.

Megoldás: A feladatban a kliensek, az ügyfelek és a brókerek mind az iroda szolgáltatásait veszik igénybe. Az iroda tehát szerver szerepet tölt be. Valójában itt egy termelő-fogyasztó rendszerről van szó, amelyben az iroda a raktár, ahol a megbízások várakoznak arra, hogy a brókerek átvegyék azokat. A raktár két megbízás tárolására alkalmas.

A bróker rendszer állapota ennek megfelelően egy aggregációs állapot, amelyet az ügyfelek állapota, a brókerek állapota és az iroda állapota együttesen határoz meg. Ezen a szinten a 6.32. ábra állapotdiagramjához jutunk.

Az ügyfélnek két állapota van: vagy várakozik, vagy kiszolgálják az irodában, azaz megbízást ad át. A leírás alapján nem lényeges nyilvántartani azt, hogy az ügyfelet melyik alkalmazott szolgálja ki, ennek a megoldás szempontjából nincs jelentősége. A megrendelések végre-



6.32. ábra. A bróker rendszer állapotdiagramja

hajtásának sorrendjére sincs semmilyen megkötés.

Az ügyfél akkor veheti igénybe a szolgáltatást, ha van szabad ügyintéző. A leírásban arról sincs szó, hogy az ügyfelek milyen sorrendben vehetik igénybe a szolgáltatást, ez tehát véletlenszerűen történik. Ezért képezhetnek egy végtelen ügyfélfolyamatot.

A brókernek is két állapota lehet: vagy várakozik (megbízást teljesít), vagy kiszolgálják (megbízást vesz át). A bróker állapotdiagramja csak abban különbözik az ügyfél állapotdiagramjától, hogy ő akkor veheti igénybe az iroda szolgáltatását, ha van el nem intézett megbízás.

Az irodának három állapota lehet annak megfelelően, hogy hány alkalmazott szabad, azaz hány alkalmazottnál nincs megbízás.

Soroljuk fel ezek után az állapotokat, az állapotátmenetek eseményeit és az átmenetek feltételeit! Egyúttal az ábrázolás céljából vezessünk be ezek számára rövid jelöléseket.

Az iroda állapotai:

- üres: két alkalmazott szabad, azaz nincs náluk megbízás;
- normál: egy alkalmazott szabad;
- tele: egyik alkalmazott sem szabad.

Az iroda állapotának változását előidéző események:

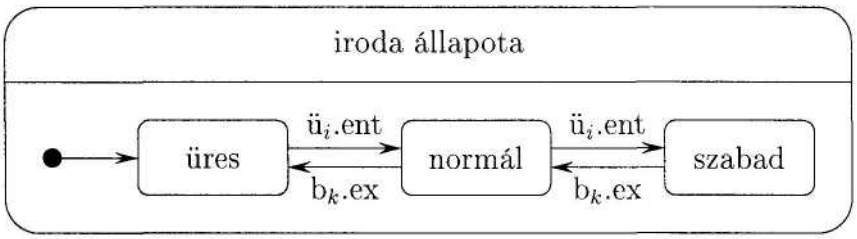
- $ü_i$.ent: egy ügyfél megkezdte a megbízást, eggyel kevesebb szabad hely lesz.
- b_k.ex: egy bróker befejezte a megbízás átvételét, eggyel több szabad hely lesz.

Ennek alapján a 6.33. ábrán látható állapotdiagramhoz jutunk.

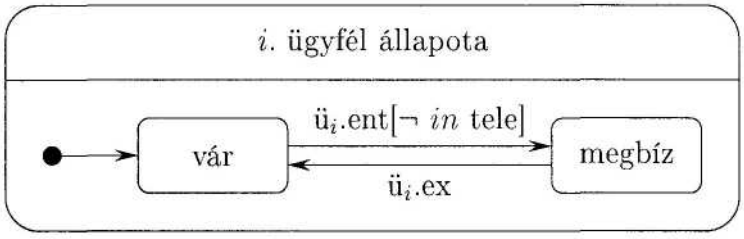
Az ügyfél állapotai:

- vár: az ügyfél arra vár, hogy legyen szabad alkalmazott;
- megbíz: az ügyfél átadja az alkalmazottnak a megbízást.

Az *i.* ügyfél állapotait változtató események és feltételeik:



6.33. ábra. Az iroda állapotdiagramja



6.34. ábra. Az ügyfelek állapotdiagramja

- $ü_i$.ent[*in* tele]: az *i.* ügyfél akkor léphet be, ha van szabad alkalmazott;
- $ü_i$.ex: az ügyfél a megbízás átadása után feltétel nélkül távozhat.

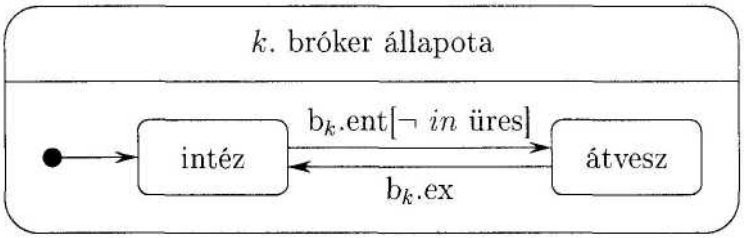
Így az *i.* ügyfél állapotát kapjuk (6.34. ábra). A brókerfolyam állapotai:

- átvesz: megbízást vesz át;
- intéz: megbízást teljesít.

Az állapotokat megváltoztató események:

- b_k.ent[*in* üres]: akkor vehet át megbízást, ha van megbízás az alkalmazottnál;
- b_k.ex: a megbízást feltétel nélkül elviheti.

A *k.* bróker állapotdiagramja látható a 6.35. ábrán.



6.35. ábra. A bróker állapotdiagramja

6.2. feladat:

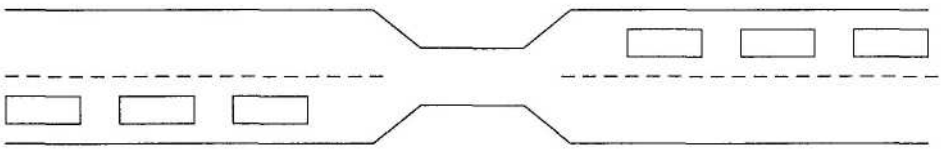
Tegyük fel, hogy programot kívánunk készíteni a forgalom lámpával történő vezérlésére, a következő leírás alapján!

Egy útszűkületben csak egy sávban közlekedhetnek a járművek, felváltva jobbról balra, illetve balról jobbra.

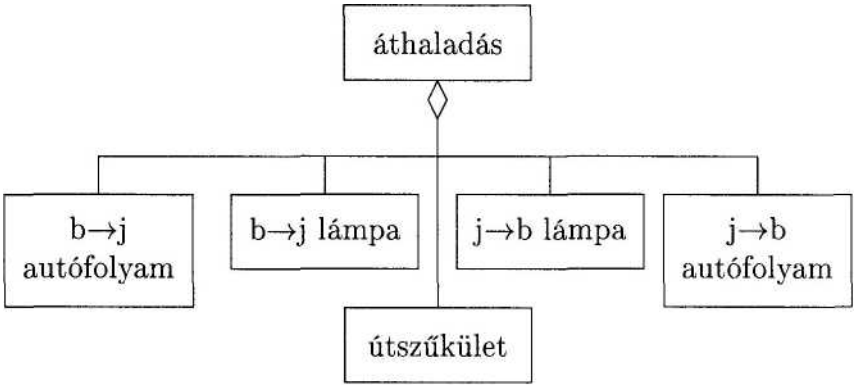
A forgalmat mindkét oldalon lámpa irányítja. A lámpa zöldre akkor vált, ha az ellenkező irányból a lámpa piros. A zöld színt a ciklusidő lejártakor sárga váltja fel. Sárgáról pirosra akkor vált a lámpa, ha az útszűkületben már nincs jármű.

A probléma vázlatos ábrázolása a 6.36. ábrán látható.

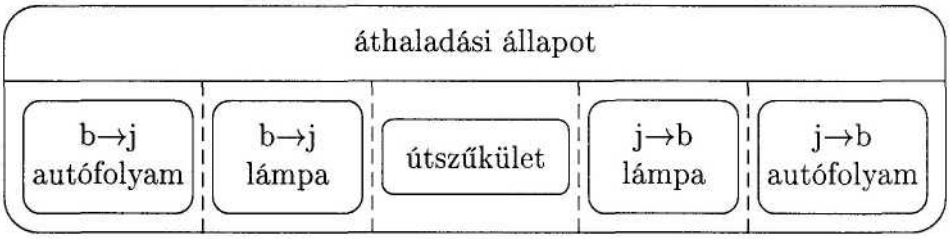
Megoldás: A leírás alapján az útszűkület olyan erőforrás, amelyet a járművek vagy csak az egyik irányból, vagy csak a másik irányból vehetnek egyszerre igénybe. A két igénybevétel között irányváltás történik, amikor az útszűkületben már nem tartózkodhat jármű. Az útszűkületben történő *áthaladás* tehát a balról jobbra haladó járműfolyamból, a jobbról balra haladó járműfolyamból, a hozzájuk tartozó lámpákból, valamint az útszűkületből mint az igénybe vett erőforrásból áll. Ennek alapján a 6.37. ábrán látható osztálydiagramhoz jutunk.



6.36. ábra. A forgalom szabályozása jelzőlámpával egy útszűkületben

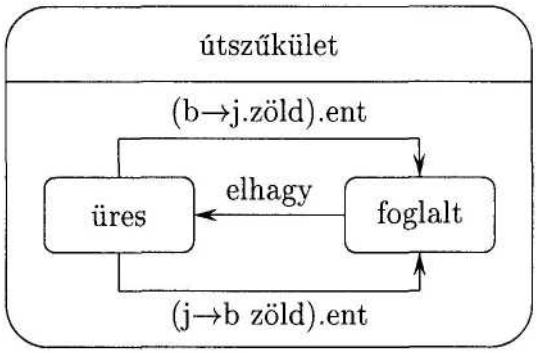


6.37. ábra. Az útszűkületben történő áthaladás osztálydiagramja

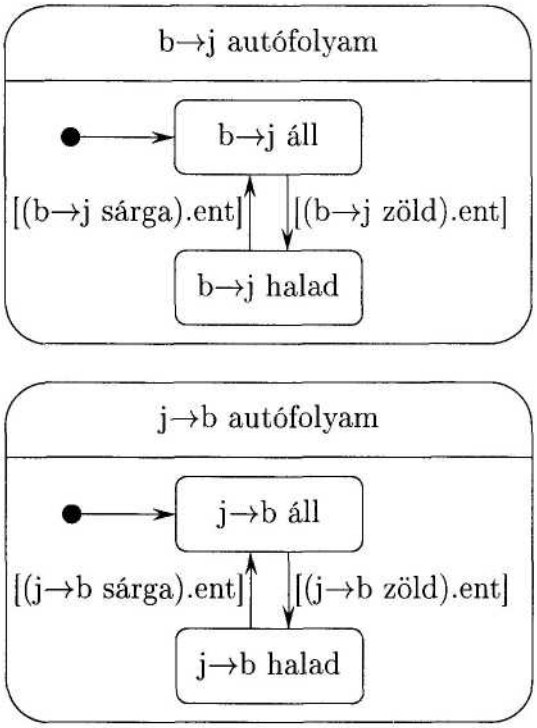


6.38. ábra. Az áthaladási állapot

Így az áthaladási állapot egy állapotaggregáció lesz (6.38. ábra). Az útszűkületnek két állapota lehet (6.39. ábra): tartózkodik benne jármű



6.39. ábra. Az útszűkület állapotdiagramja

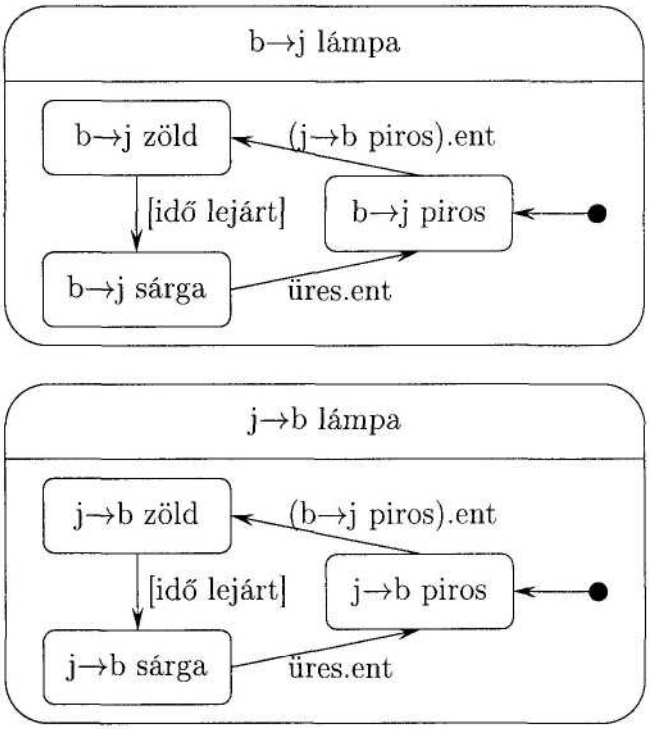


6.40. ábra. Az autófolyamok állapotdiagramjai

(foglalt), vagy nincs benne jármű (üres). Ha valamelyik lámpa zöldre vált, akkor foglalttá válik, és üres lesz, ha az utolsó jármű is elhagyja. (Tegyük fel, hogy ezt érzékelni tudjuk, és az érzékelés eredménye az „elhagy" akció!)

A gépkocsifolyamok állapotát a megfelelő lámpa állapota határozza meg. Az autók a lámpa előtt állnak, vagy haladnak, kezdetben állnak. Ha a lámpa zöldre vált, akkor haladnak, és ha a lámpa sárgára vált, akkor megállnak (6.40. ábra).

A lámpának három állapota lehet, az aktuális színnek megfelelően. Csak akkor válthat zöldre a lámpa, ha az ellenkező lámpa pirosra váltott. Sárgára az idő letelte után vált, és piros lesz, ha az útszűkület kiürül (6.41. ábra).



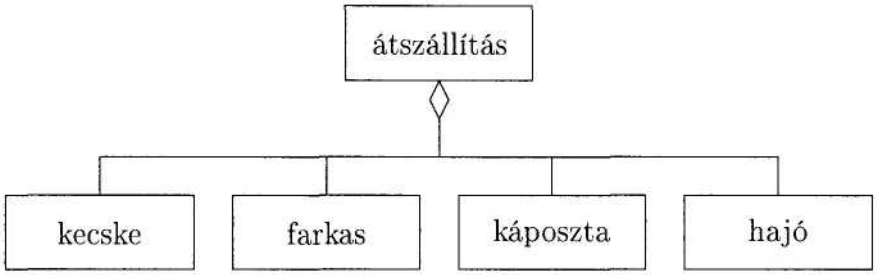
6.41. ábra. A jelzőlámpák állapotdiagramjai

6.3. feladat:

Oldjuk meg a kecske, farkas és káposzta folyón történő átszállításának jól ismert problémáját! Ez lehet egy olyan szimulációs program, amely a problémában rögzített szabályok szerint folyamatosan szimulálja az átkelést. A szabályok a következők:

1. A kecske a káposztával nem maradhat azonos parton, ha a hajós nincs jelen, mert megeszi a káposztát.
2. A farkas nem maradhat azonos parton a kecskével, ha a hajós nincs jelen, mert megeszi a kecskét.

Megoldás: A leírás alapján az objektumok, illetve az osztályok a következők lesznek: átszállítás, kecske, káposzta, farkas, hajó. Az átszál-

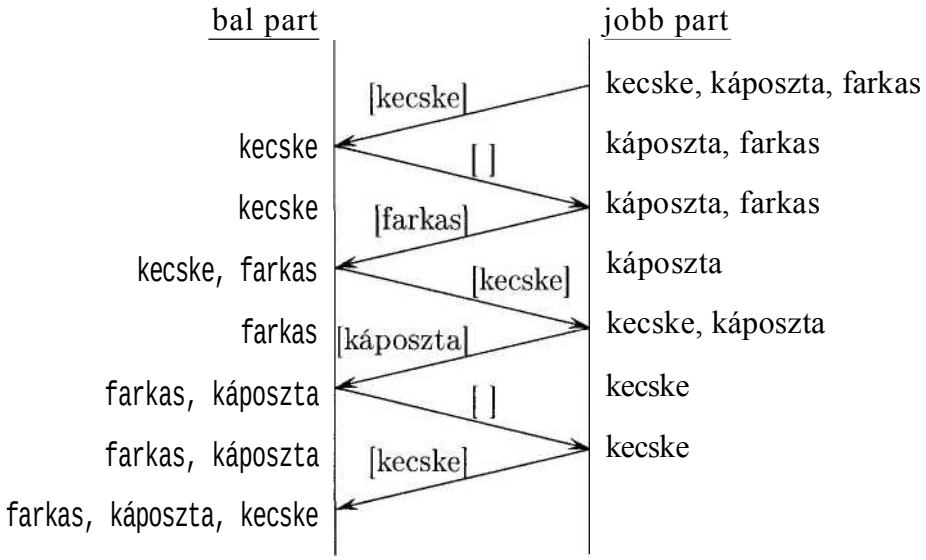


6.42. ábra. A kecske-farkas-káposzta átszállításának osztálydiagramja

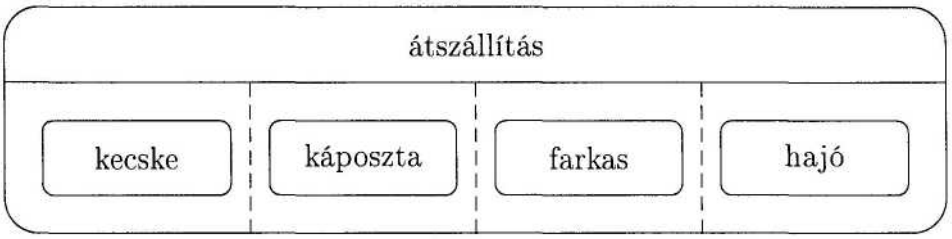
lítás az a rendszer, amelyet a kecske, káposzta, farkas és hajó objektumok alkotnak. Ez a rendszer tehát egy aggregátum. Osztálydiagramja a 6.42. ábrán látható.

A probléma egy lehetséges megoldása látható a 6.43. ábrán.

Az átszállítás állapotát a kecske, a káposzta, a farkas és a hajó állapota együttesen határozzák meg. Ezért az átszállítás állapota a



6.43. ábra. A kecske-farkas-káposzta probléma egy lehetséges megoldása. A csónak tartalmát [ ] jelek között tüntettük fel, [ ] jelenti az üres csónakot.



6.44. ábra. Az átszállítás állapotdiagramja

felsorolt állapotok aggregációja (6.44. ábra).

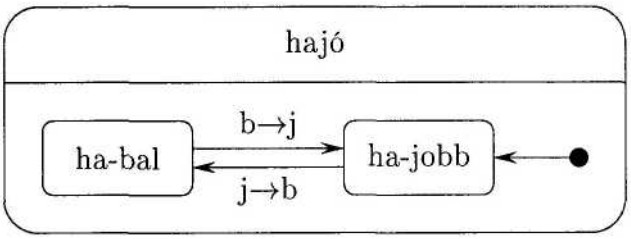
A hajó állapota: A hajó a folyó két partja között ingázik. A probléma megoldása szempontjából az a lényeges, hogy melyik parton kötött ki. Ennek megfelelően az állapotok:

- hajó a bal parton kikötött : ha-bal,
- hajó a jobb parton kikötött : ha-jobb.

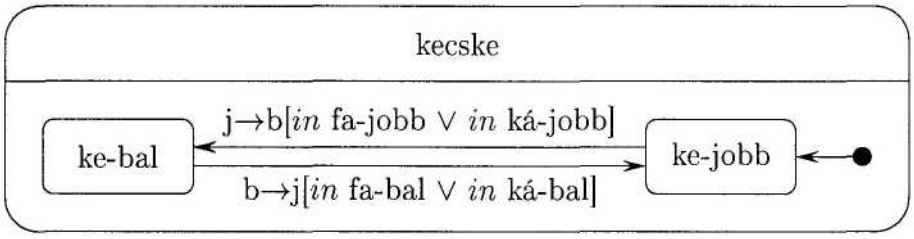
Az állapotot megváltoztató események pedig:

- átkelés balról-jobbra : b—>j,
- átkelés jobbról-balra : j—>b.

Az átkelés valójában időben elhúzódó esemény, azonban a modellben ezt egyszerűsíthetjük, és akcióként is tekinthetünk rá. Így a hajó állapotait a 6.45. ábra diagramja írja le.



6.45. ábra. A hajó állapotdiagramja



6.46. ábra. A kecske állapotdiagramja

A kecske állapota: A kecskének is két állapota van, attól függően, hogy melyik parton tartózkodik. Az állapotokhoz invariánsokat rendelünk, amelyekkel kifejezzük, hogy a kecske csak akkor tartózkodhat az egyik parton, ha a hajó ott van, vagy nincs azon a parton sem a farkas, sem a káposzta. Az invariánsokkal kifejezzük, hogy a hajót akkor használja a kecske, ha a farkas mellől kell menekülnie, vagy ha a káposzta kerül felügyelet nélkül veszélybe a jelenléte miatt. Az állapotok és invariánsaik a következők.

- A kecske a bal parton van: ke-bal, invariánsa:
  in ha-bal V *(in* ká-jobb A *in* fa-jobb).

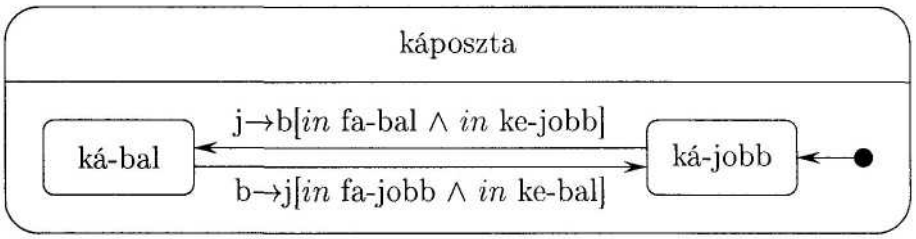
- A kecske a jobb parton van: ke-jobb, invariánsa:
  in ha-jobb V *(in* ká-bal ^ *in* fa-bal).

Az állapotok közötti átmenetet az átkelések adják, az egyetlen megszorítás, hogy az invariánsoknak fenn kell állniuk. Ezt biztosíthatjuk, ha a kecskét mindig átszállítjuk, amikor a farkas vagy a káposzta vele azonos parton van (6.46. ábra).

A káposzta állapota: A kecskéhez hasonlóan a káposztának is két állapota van, amelyeket invariánsokkal jellemezhetünk.

- A káposzta a bal parton van: ká-bal, invariánsa:
  in ha-bal V *in* ke-jobb.

- A káposzta a jobb parton van: ká-jobb, invariánsa:
  in ha-jobb V *in* ke-bal.



6.47. ábra. A káposzta állapotdiagramja

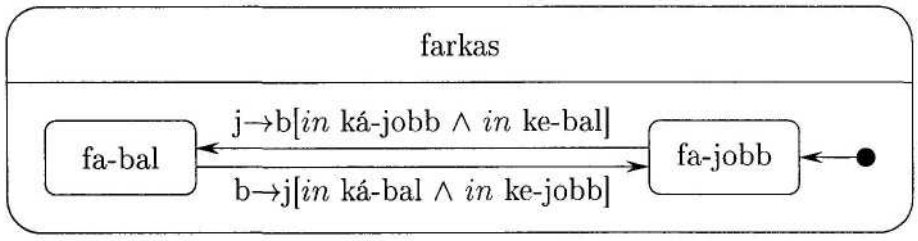
Az invariánsok kifejezik, hogy a kecske és a káposzta csak akkor lehet ugyanazon a parton, ha a hajó is ott van.

Az állapotok közötti átmeneteket az átkelések adják, amelyekhez feltételt rendelve biztosíthatjuk, hogy a kecske és a káposzta állapotainak invariánsa is fennálljon. Ezért a káposztát akkor kell átszállítani, ha a kecske azonos parton van, és nincs azon a parton a farkas (6.47. ábra).

A farkas állapota: Az eddigiekhez hasonlóan eljárva meghatározhatjuk az állapotokat és invariánsaikat.

- A farkas a bal parton van: fa-bal, invariánsa:
  in ha-bal V *in* ke-jobb.
- A farkas a jobb parton van: fa-jobb, invariánsa:
  in ha-jobb V *in* ke-bal.

Az invariánsok alapján kapjuk, hogy farkast át kell vinni a kecskéhez a káposzta mellől (6.48. ábra).



6.48. ábra. A farkas állapotdiagramja

6.4. feladat:

Készítsük el a következő leírás alapján a termelő-fogyasztó rendszer állapotdiagramját!

A termelő-fogyasztó rendszer raktárból, termelőből és fogyasztóból áll. A termelő az általa előállított árut akkor helyezheti el a raktárban, ha van üres hely. A raktárban *n* darab üres hely van. A fogyasztó akkor szállíthat ki a raktárból árut, ha a raktárban van áru.

Megoldás: A termelő-fogyasztó rendszer állapotát a termelő állapota, a fogyasztó állapota és a raktár állapota együttesen határozza meg. A termelő-fogyasztó rendszer állapota tehát ezek aggregációja (6.49. ábra).

A raktár állapota egy általános állapot. Ennek konkrét eseteit a hozzáférés szempontjából a következőképpen határozhatjuk meg:

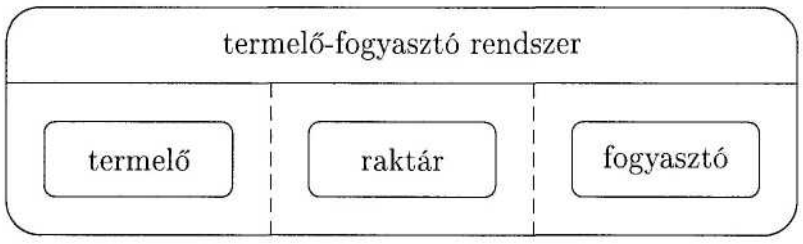
1. Üres állapot, amikor a raktárban az üres helyek száma *(hely)* megegyezik a raktár kapacitásával *(n* > 0):

   üres[hely = n] .

2. Normál állapot, amikor a raktár már nem üres, és még nincs tele:

   normál[0 < *hely* < *n]* .

3. Tele állapot, amikor az üres helyek száma 0:



6.49. ábra. A termelő-fogyasztó rendszer állapotdiagramja

tele[hely = 0] .

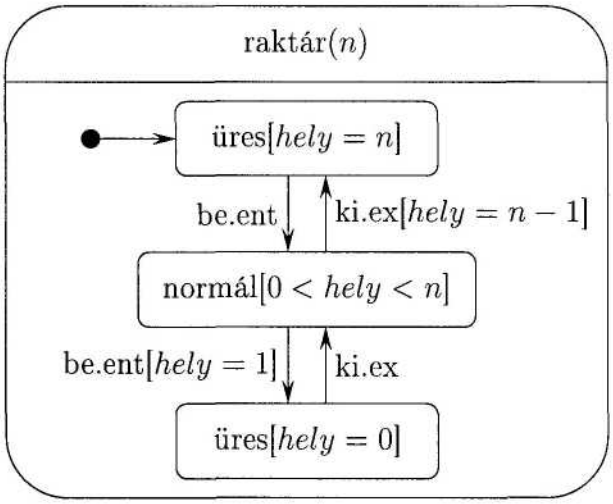
A raktárban állapotváltozásokat a következő események okoznak:

1. Áru elhelyezése az üres raktárban, aminek eredményeként a normál állapot jön létre $n > 1$ esetén. Legyen ez az akció: be.ent!
2. Áru elhelyezése a normál állapotú raktárban, aminek eredményeként az megtelik (üres helyek száma 0 lesz): *be.ent[hely* = 1].
3. Áru kivétele a teli raktárból, aminek eredményeként a normál állapot jön létre $n > 1$ esetén. Legyen ez: ki.ex!
4. Áru kivétele a normál állapotú raktárból, aminek eredményeként az kiürül (üres helyek száma *n* lesz): *ki.ex[hely* = *n* — 1].

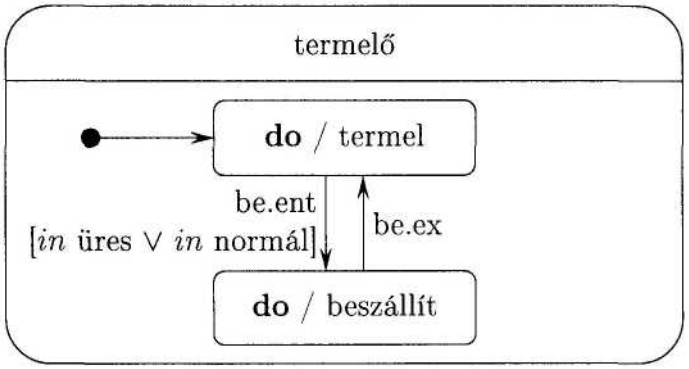
Ennek alapján a 6.50. ábra állapotdiagramját kapjuk eredményül.

A termelő állapotai:

1. Az az állapot, amikor a termelő az árut előállítja. Legyen ez:
   do / termel! Ebbe az állapotba a termelő feltétel nélkül átmehet.



6.50. ábra. A raktár állapotdiagramja



6.51. ábra. A termelő állapotdiagramja

2. Az az állapot, amikor a termelő az árut elhelyezi a raktárban. Legyen ez:
   do / beszállít!
   Ebbe az állapotba akkor mehet át a termelő, ha van üres hely a raktárban, azaz ha a raktár állapota *üres* vagy *normál.*

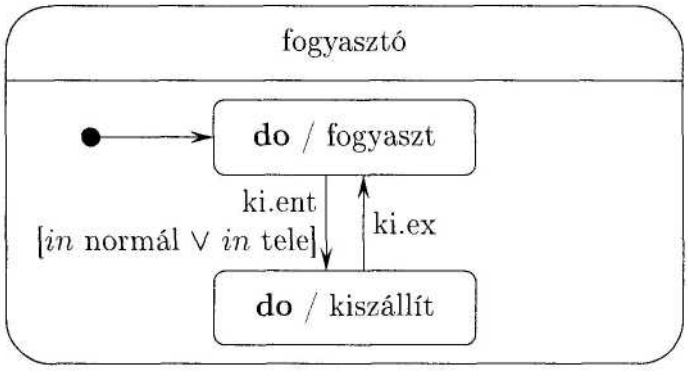
A beszállítás eseményéhez tartozó belépési és kilépési akciókra röviden „be.ent" és „be.ex" néven hivatkozunk. A termelő állapota tehát a 6.51. ábrán látható diagram lesz.

A fogyasztó állapotai:

1. Az az állapot, amikor az árut kiszállítja a raktárból. Legyen ez:
   do / kiszállít!
   Ebbe az állapotba akkor léphet be, ha van áru a raktárban, azaz:
   ki.ent[*in* normál V *in* tele].

2. A kiszállított áru feldolgozásának állapota:
   do / fogyaszt. Ebbe az állapotba feltétel nélkül átmehet az áruval.

A kiszállít eseményre a belépési és kilépési fázisban röviden „ki" néven hivatkozunk. A fogyasztó állapotdiagramja a 6.52. ábrán látható.



6.52. ábra. A fogyasztó állapotdiagramja

6.5. feladat:

A hallgatók adatait egy hallgatói nyilvántartó rendszer tartalmazza. A hallgatók adatait többféle módon lehet felújítani, például lakcímváltozás, szakpárváltozás stb. A hallgató adatait tehát több folyamat is felújíthatja. Hasonlóan a hallgatók adatait több folyamat is lekérdezheti, kiértékelheti. Egyszerre azonban legfeljebb egy folyamat dolgozhat ugyanannak a hallgatónak az adatain. Ha egy felújító folyamat aktualizálni kívánja egy hallgató adatait, akkor a lekérdezéssel meg kell várni a felújítás befejezését.

1. Készítsük el a hallgatói rendszer osztálydiagramját!

2. Készítsük el a rendszer állapotdiagramját, felhasználva az álla potaggregációt és az állapotáltalánosítást!

Megoldás: Először a követelmények elemzése alapján az osztálydiagramot készítjük el.

Az osztályok: hallgatói rendszer, hallgatói adatbázis, felújító folyamat, lekérdező folyamat.

A relációk:

- a hallgatói rendszer egy aggregáció, amelynek komponense a többi osztály;

- a lekérdező objektumok az adatbázist lekérdezik, azaz asszociációs kapcsolatban állnak vele;
- a felújító folyamatok felújítják az adatbázist, tehát ebben az esetben szintén asszociációról van szó.

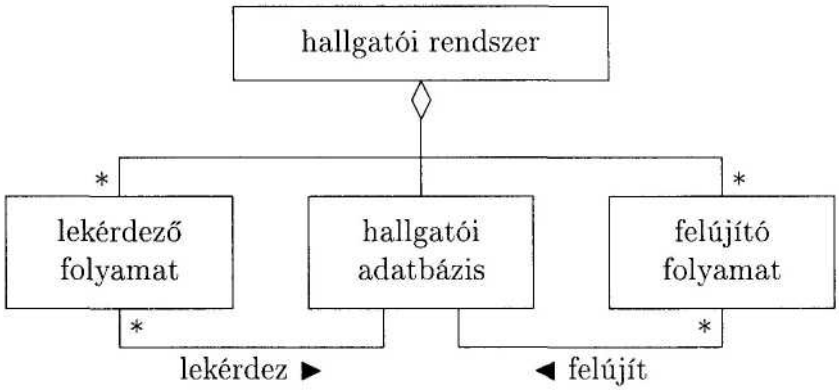
Multiplicitás:

- az aggregációban egy adatbázis és több felújító, illetve lekérdező folyamat vesz részt;
- egyszerre több felújító, illetve lekérdező folyamat dolgozhat az adatbázison, csak egy hallgató adatait használhatja egyszerre egy folyamat.

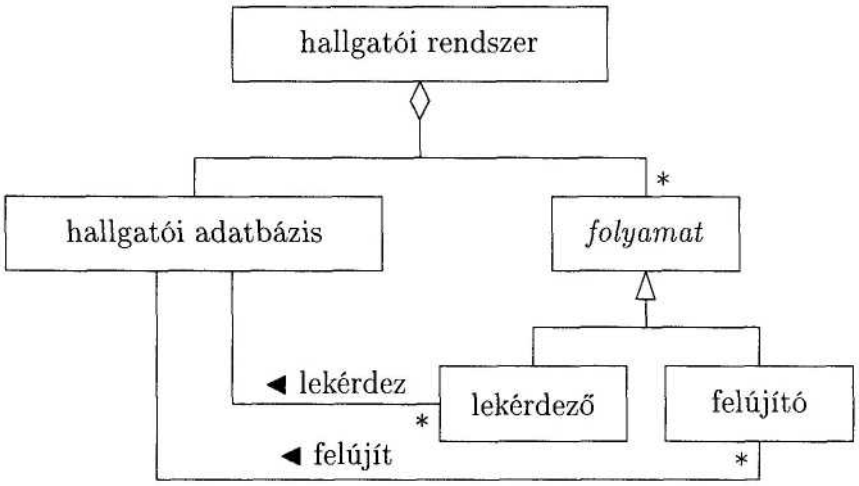
Az eddigieket összegezve a 6.53. ábrához jutunk.

Természetesen más megoldás is lehetséges. Például a 6.54. ábrán szereplő osztálydiagram is helyes megoldás. Az eltérés oka, hogy a lekérdező, illetve a felújító folyamatokat egy közös folyamat specializációjának tekintjük.

Az osztálydiagram után térjünk át az állapotdiagram megalkotására! Egyszerre csak egy folyamat dolgozhat ugyanannak a hallgatónak az adatain. Ha egy felújító folyamat aktualizálni kívánja egy hallgató



6.53. ábra. A hallgatói rendszer és adatbázis kapcsolata



6.54. ábra. A hallgatói rendszer osztálydiagramja

adatait, akkor a lekérdező folyamat csak az aktualizálás után férhet hozzá a szóban forgó hallgató adataihoz. A lekérdező folyamat állapotai:

1. Az *l* azonosítóval rendelkező folyamat lekérdezi a *k-val* $(k \in$ *kulcsok)* azonosított hallgató adatait:

   do / lekérdez$_l(k)$.

2. Az *l* azonosítóval rendelkező folyamat nem végez lekérdezést, hanem feldolgozza a *k* azonosítójú hallgató adatait:

   do / feldolgoz$_l(k)$.

A felújító folyamat állapotai:

1. Az / azonosítóval rendelkező folyamat előkészíti a i-vel $(j \in$ *kulcsok)* azonosított hallgató adatainak felújítását:

   do / előkészít$_f(j)$.

2. Az / azonosítóval rendelkező folyamat bejelenti az igényét arra, hogy a *j*-vel azonosított hallgató adatait felújíthassa:

   do / igényel$_f(j)$.

3. Az / azonosítóval rendelkező folyamat felújítja a *j* kulcsú hallgató adatait:
   do / felújít$_f$ (*j*).

Ezek után vizsgáljuk meg az állapotmeneteket és azok feltételeit!

A lekérdező folyamat:

a. A lekérdező folyamat induláskor do / feldolgoz, *(k)* állapotba kerül. Ebbe az állapotba feltétel nélkül, bármikor átmehet.

b. A do / lekérdez$_i$(*k*) állapotba akkor mehet át, ha:

   1. nincs olyan lekérdező folyamat, amely a *k* kulcsú hallgató adatait kérdezi le:

$$\nexists i : in \text{ lekérdez}_i(k) \equiv A(k) ;$$

   2. nincs olyan felújító folyamat, amely a *k* kulcsú hallgató adatait újítja fel:

$$\nexists i : in \text{ felújít}_i(k) \equiv B(k) ;$$

   3. nincs olyan felújító folyamat, amely a *k* kulcsú hallgató adatainak felújítására vár:

$$\nexists i : in \text{ igényel}_i(k) \equiv C(k) .$$

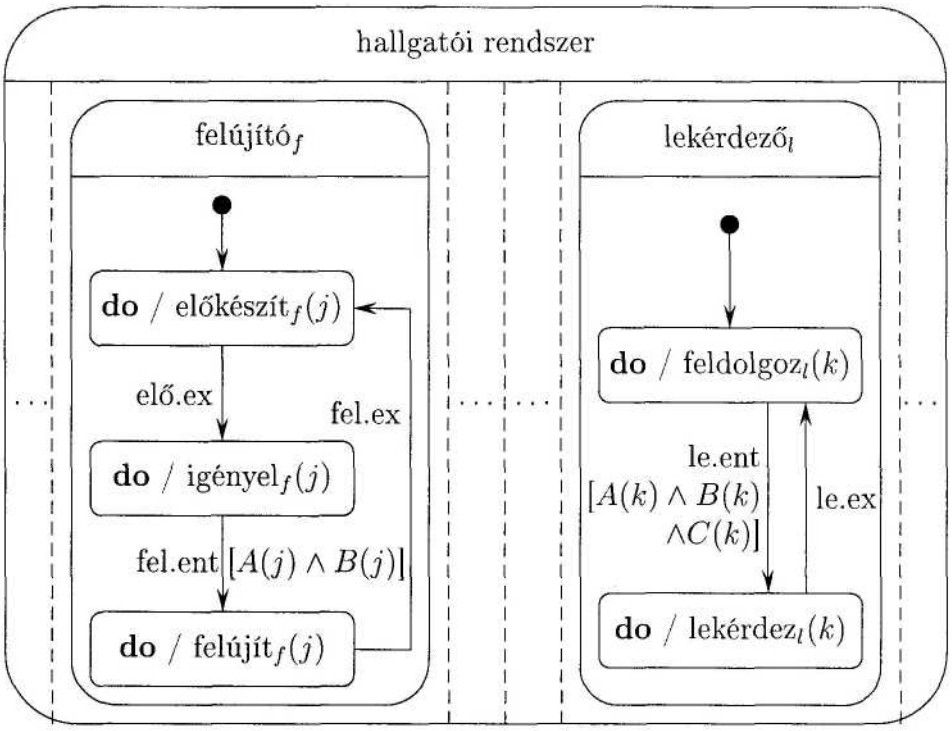
A felújító folyamat:

a. A felújító folyamat induláskor a do / előkészít$_f$*(j)* állapotba kerül. Ebbe az állapotba bármikor, feltétel nélkül átmehet.

b. A do / igényel$_f$*(j)* állapotba is bármikor átmehet a folyamat.

c. A do / felújít$_f$*(j)* állapotba akkor mehet át a folyamat, ha:

   1. nincs olyan lekérdező folyamat, amely a *j* kulcsú hallgató adatait kérdezi le, ennek a feltétele: *A(j);*



6.55. ábra. A hallgatói rendszer állapotdiagramja

2. nincs olyan felújító folyamat, amely a *j* kulcsú hallgató adatait újítja fel, ennek a feltétele pedig: *B(j)*.

Összegezve az eddigieket a 6.55. ábrán látható állapotdiagramhoz jutunk.

6.6. feladat:

Körgyűrűk mentén szolgáló szervizkocsik vezérlésére kell programot készíteni. A körgyűrű segélyhelyekből és a segélyhelyen szervizszolgáltatást nyújtó szervizkocsiból áll. A segélyhelyek egymástól egységnyi távolságra helyezkednek el a körgyűrű mentén, és számuk *m*. Van egy olyan segélyhely, amely a kiszolgálás szempontjából kitüntetett szerepet tölt be.

1. Készítsük el a körgyűrű osztálydiagramját! Töltsük ki az osztá lyok attribútum és műveleti részeit is!

2. Ha a szervizkocsi egy segélyhelyről elmegy, akkor azonnal a kö vetkező hívás helyére megy. A kiszolgálás szempontjából az egyik helyről a másik helyre történő átjutásának időigényét el lehet hanyagolni, ezt nem kell külön állapotként kezelni. Készítsük el ennek alapján a körgyűrű állapotdiagramját!

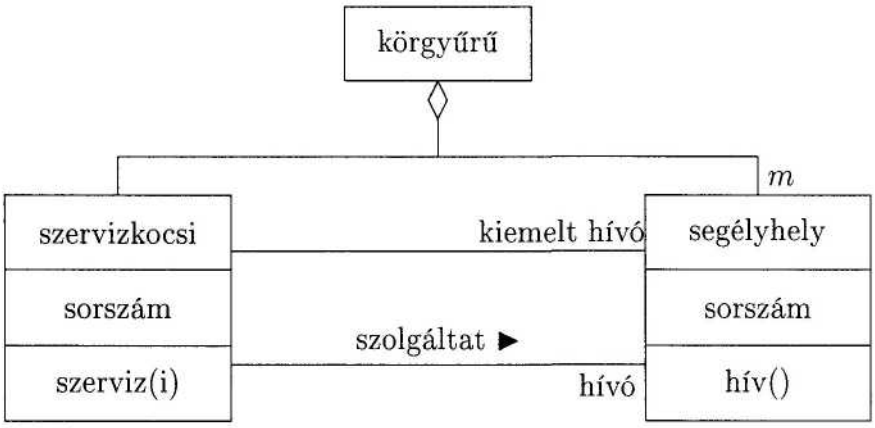
Megoldás: Az osztálydiagram elkészítéséhez határozzuk meg az osztályokat, attribútumaikat és kapcsolataikat!

Az osztályok: körgyűrű, szervizkocsi, segélyhely. Az osztályokhoz tartozó attribútumok és műveletek:

- a segélyhely esetén ez lehet annak azonosító sorszáma;
- a szervizkocsi esetén az, hogy éppen hol tartózkodik, ami szintén sorszám:
- a szervizkocsi esetén lehetséges művelet a szolgáltatás művelete, a *szerviz(i),* ahol *i* a szolgáltatás helye;
- a segélyhely esetén lehetséges művelet a hívás művelete, *hív.*

A relációk:

- a körgyűrű objektum egy szervizkocsi és *m* segélyhely aggregá ciója;
- a szervizkocsi szolgáltatásnyújtás révén kerül kapcsolatba a se gélyhellyel, amely egy asszociációs kapcsolat, mindkét oldalon egyszeres multiplicitással;
- a segélyhelynek az a szerepe ebben a kapcsolatban, hogy kéri a szolgáltatást, ő a hívó fél; és van egy olyan fél is, aki kiemelt szerepet játszik.

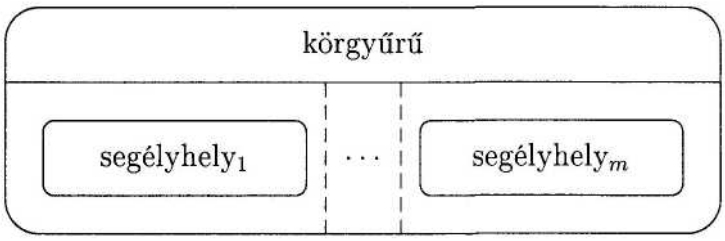


6.56. ábra. A körgyűrű osztálydiagramja

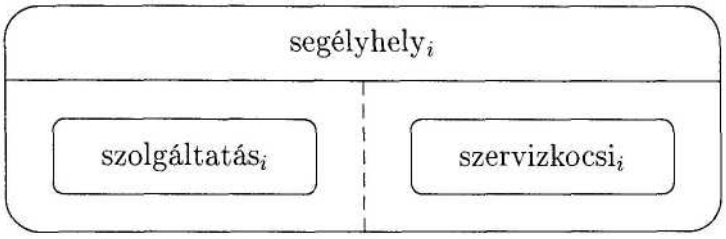
Ennek alapján a 6.56. ábrán szereplő osztálydiagramhoz jutunk.

Térjünk át a feladat második részére, az állapotdiagramra! A körgyűrű állapotát nyilván a segélyhelyek állapota együttesen határozza meg; van-e javítási kérés, ott van-e a szervizkocsi, stb. A szervizkocsi állapotát a segélyhelyek állapota határozza meg, ugyanis annak része, hogy ott tartózkodik-e a szervizkocsi. A körgyűrű állapota tehát a segélyhelyek állapotainak aggregációja (6.57. ábra).

A segélyhely állapotát az határozza meg, hogy ott van-e a szervizkocsi, és mi a szolgáltatási állapot. Ez tehát egy állapotaggregáció, amelynek részei a szolgáltatás és szervizkocsi állapota (6.58. ábra).



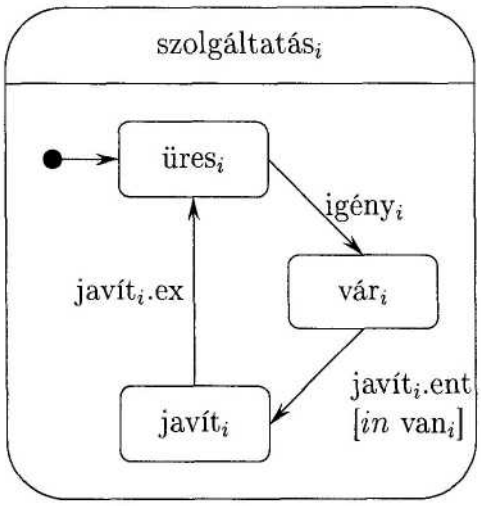
6.57. ábra. A körgyűrű állapotdiagramja



6.58. ábra. A segélyhely állapotdiagramja

A szolgáltatáson belül három állapot lehetséges:

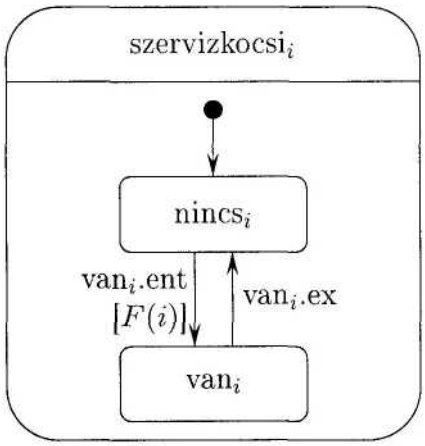
- üres: nincs szervizre igény, ez a kezdeti állapot;
- vár: van szolgáltatási igény, amelybe az „igény" esemény hatására kerülünk;
- javít: a szervizkocsi ezen a segélyhelyen végez szolgáltatást, ami csak akkor következhet be, ha a szervizkocsi ezen a helyen tar tózkodik (6.59. ábra).



6.59. ábra. A szolgáltatás állapotdiagramja

$$F(i) \equiv \; in\ \text{nincs}_1 \wedge \ldots \wedge in\ \text{nincs}_m \wedge in\ \text{vár}_i \; .$$

Ennek alapján a szervizkocsi állapotdiagramjának a 6.60. ábrán látható diagram felel meg.



6.60. ábra. A szervizkocsi állapotdiagramja

7. Szekvenciadiagram

A rendszer működésének leírására szolgál a szekvenciadiagram. A probléma megoldása során az objektumok egymásnak üzeneteket küldenek. Az üzenetek időbeli sorrendjének szemléltetése gyakran megkönnyíti a probléma megoldásának megértését. Ennek a feladatnak egyik eszköze a szekvenciadiagram.

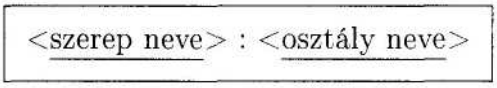
A szekvenciadiagram alapfogalmai, komponensei a következők:

- osztályszerep,
- osztályszerep életvonala,
- aktivációs életvonal,
- üzenet.

Ezeket fogalmakat és jelöléseiket tekintjük át a továbbiakban.

7.1. Osztályszerep

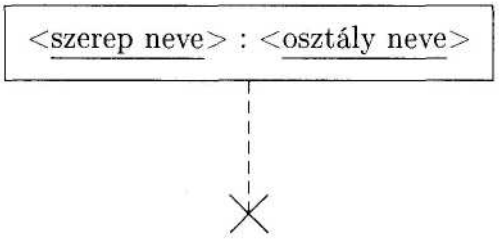
Az osztály szerepét az osztályok közötti üzenetben megtestesítheti az osztály egy vagy több objektuma, amelyek az üzenetküldés szempontjából konform módon járnak el. Az osztályszerep megnyilvánulhat az osztályok egy halmazának megtestesítőjeként is (például generic form). Az osztályszerep jelölése a 7.1. ábrán látható.



7.1. ábra. Az osztályszerep jelölése

7.2. Osztályszerep életvonala

Az életvonal az osztályszerep időben való létezését jelenti. Az osztályszerep életvonalának jelölése a 7.2. ábrán látható.



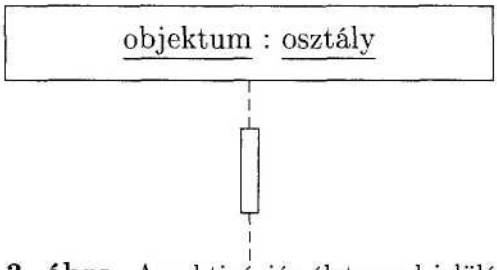
7.2. ábra. Az osztályszerep életvonalának jelölése, ahol x az osztályszerepet megtestesítő objektum megszűnését jelenti

7.3. Az osztályszerep aktivációs életvonala

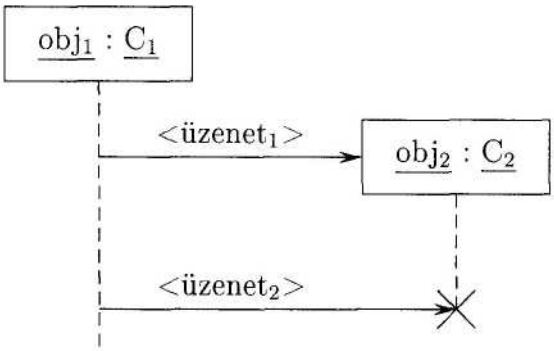
Az aktivációs életvonal az osztályszerepnek azt az állapotát jelöli, amelyben az osztályszerep megtestesítői műveletet hajtanak végre, és más objektumok vezérlése alatt állnak. Az aktivációs életvonal jelölése a 7.3. ábrán látható.

7.3.1. Az objektum létrehozása és megsemmisítése

Egy objektum létrejöhet egy másik objektum létrehozó üzenetének a hatására, és megsemmisülhet, ha a másik objektum egy törlést jelentő üzenetet ad ki. Jelölése a 7.4. ábrán látható.



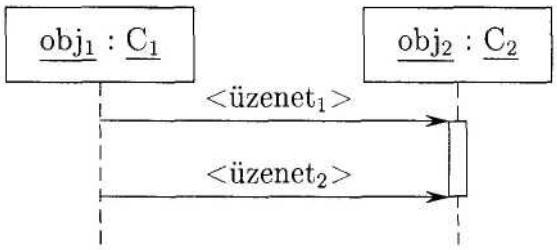
7.3. ábra. Az aktivációs életvonal jelölése



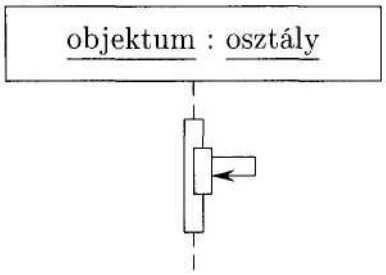
7.4. ábra. Objektum létrehozása és megsemmisítése: az obj_1 objektum első üzenete létrehozza az obj_2 objektumot, a második üzenet megsemmisíti

7.3.2. Az objektum aktivációja

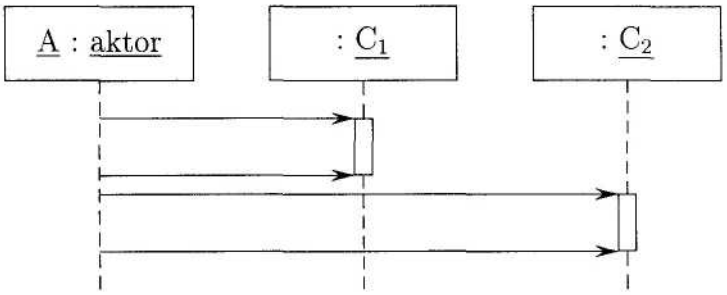
Életvonala során az objektum aktív módon viselkedhet, amíg valamilyen tevékenységet végrehajt. Az aktív állapotot előidézheti egy másik objektum üzenete, illetve üzenettel meg is szüntetheti azt (7.5. ábra).



7.5. ábra. Az objektum aktivációjának jelölése; ha szükséges, az aktív szakasz mellett feltüntethető a tevékenység neve



7.6. ábra. A rekurzív aktiváció jelölése



7.7. ábra. A centrális aktiváció és az aktor jelölése

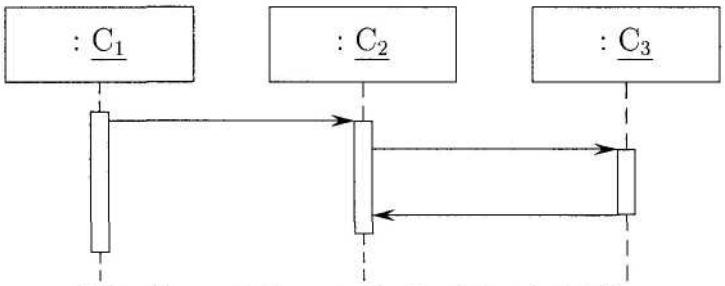
7.8. ábra. Az aktor jelölése másként

Az objektum aktivációjának egy speciális esete a *rekurzív aktiváció,* amikor egy objektum saját magát aktiválja (7.6. ábra).

Az aktiváció lehet *centrálisan vezérelt.* Centrális esetben minden objektumot egy *aktor* objektum aktivizál (7.7. ábra).

Megjegyzés: Az aktor egy másik szokásos jelölése látható a 7.8. ábrán.

Az aktivizáció lehet decentralizáltan vezérelt (7.9. ábra).



7.9. ábra. A decentralizált aktiváció jelölése

7.4. Üzenettípusok

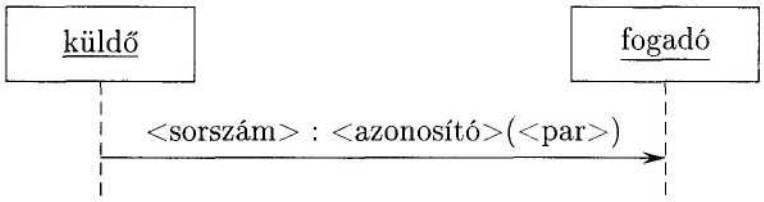
Az eddigiekben láttuk, hogy az objektumok aktivációs életvonalának alakulását az üzenetek szabályozzák. A következőkben ezen üzeneteket osztályozzuk annak alapján, hogy miként viszonyulnak az objektumokhoz. Az üzenet az objektumok közötti információátadás formája. Az üzenet egy példányának átadása általában egy esemény bekövetkezése. Az üzenet küldésének az a célja, hogy az objektum működésbe hozza a másik objektumot. Az üzenet azok között az objektumok között jöhet létre, amelyek az objektumdiagramban kapcsolatban állnak. Az üzenet az esemény egy példánya. Az üzenet küldése egy olyan akció, amelynek eredménye egy végrehajtható utasítás. Az üzenetnek van azonosítója (neve, szövege), lehet paramétere, sorszáma.

7.4.1. Egyszerű üzenet

Egy aktív objektum üzenetet küld egy passzív objektumnak. Az aktív objektum átadja a vezérlést a passzív objektumnak. Ez tulajdonképpen egy közönséges eljáráshívás (7.10. ábra).

7.4.2. Szinkronizációs üzenet

A küldő objektum elküldi az üzenetet, és a küldő blokkolt állapotba kerül, amíg a fogadó nem fogadta az üzenetet. A küldő várakozik, amíg a szinkronizációs feltétel nem teljesül (7.11. ábra).

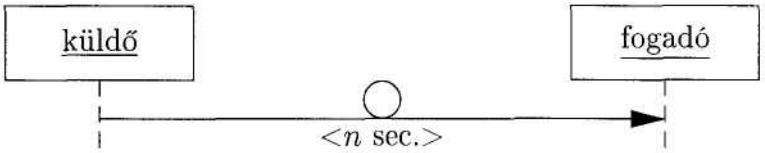


7.10. ábra. Egyszerű üzenet jelölése

7.11. ábra. Szinkronizációs üzenet jelölése

7.4.3. Időhöz kötött várakozás

A küldő legfeljebb a megjelölt ideig várakozik arra, hogy a fogadó fogadja az üzenetet. Ha ennyi idő alatt ez nem következik be, akkor folytatja a tevékenységét (7.12. ábra).



7.12. ábra. Időhöz kötött várakozás jelölése

7.4.4. Randevú üzenet

A fogadó várakozik arra, hogy a küldő üzenetet küldjön neki. A fogadó előbb várakozó állapotba helyezi magát a fogadáshoz (7.13. ábra).

7.13. ábra. Randevú üzenet jelölése

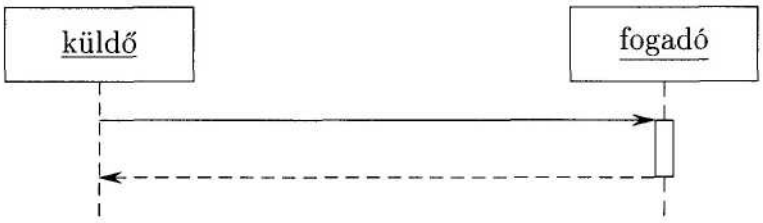
7.4.5. Aszinkron üzenet

A küldő folyamat nem szakad meg, nem érdekli őt, hogy mikor kapta meg a fogadó az üzenetet (7.14. ábra).

7.14. ábra. Aszinkron üzenet jelölése

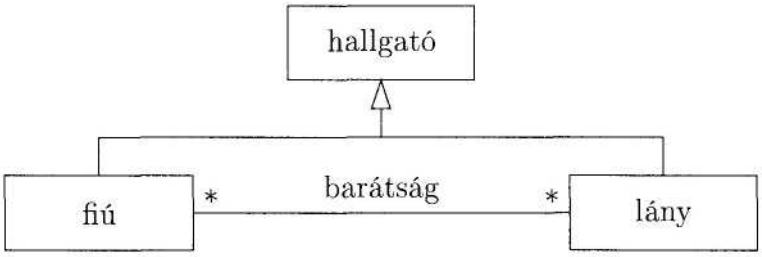
7.4.6. Visszatérési üzenet

Az üzenetet egy aktivizált objektum küldi az aktivizáló objektumnak akkor, amikor befejezi a tevékenységet, és a vezérlést visszaadja. Ezt az üzenetet gyakran nem tüntetjük fel a szekvenciadiagramban, mert rendszerint kiolvasható a többi üzenetből és az aktív szakaszokból. Vannak azonban olyan esetek, amikor ennek feltüntetése célszerű. A jelölés a 7.15. ábrán látható.



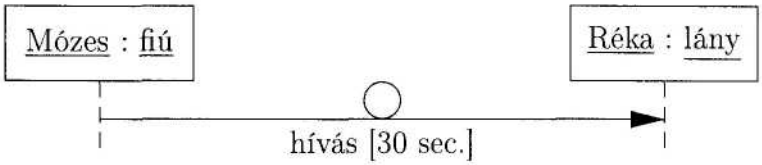
7.15. ábra. A visszatérési üzenet jelölése: a szaggatott nyíl mutatja a visszatérést

A definíciók után nézzünk néhány példát az egyes üzenetekre! Az első két példában a hallgatók osztályát használjuk, amelyből specializációval kapjuk a fiú és lány hallgatókat, akik barátságban lehetnek egymással (7.16. ábra).

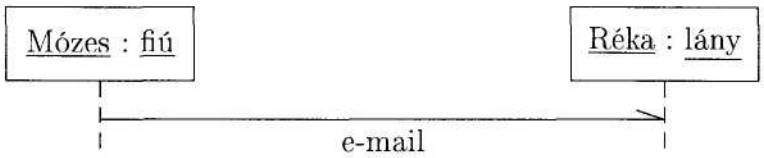


7.16. ábra. A hallgatókat és a barátság kapcsolatot leíró osztálydiagram

Időhöz kötött várakoztatásra példa, amikor Mózes telefonon felhívja Rékát, és 30 másodpercig vár arra, hogy Réka felvegye a telefont (7.17. ábra). Aszinkron üzenetre egy példa, ha Mózes e-mail-t küld Rékának, majd tanul tovább (7.18. ábra).



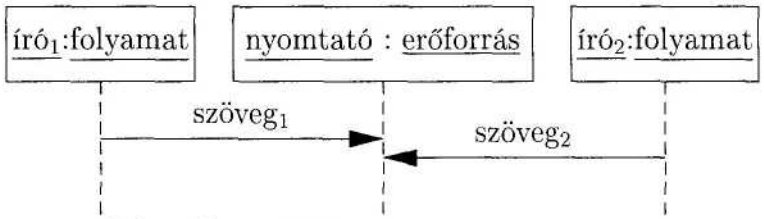
7.17. ábra. Példa időhöz kötött várakozásra



7.18. ábra. Példa aszinkron üzenetre

A szinkronizációs üzenetre bemutatandó példához tekintsünk egy kölcsönös kizárásos modellt (7.19. ábra)! Ennek a modellnek az alapján ha ketten kívánják használni például a nyomtatót - ami az erőfor-

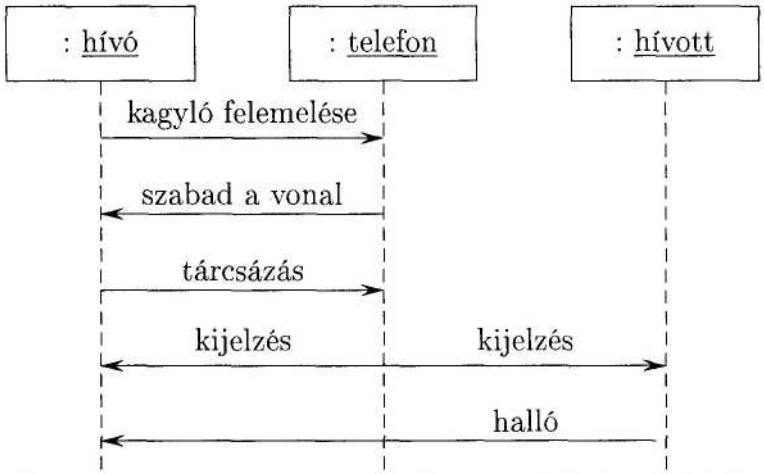
7.19. ábra. Kölcsönös kizárásos rendszer osztálydiagramja



7.20. ábra. Példa szinkronizációs üzenetre

rás szerepét tölti be - a szöveg kiírására, akkor a 7.20. ábrán látható diagram írja le a kialakult helyzetet.

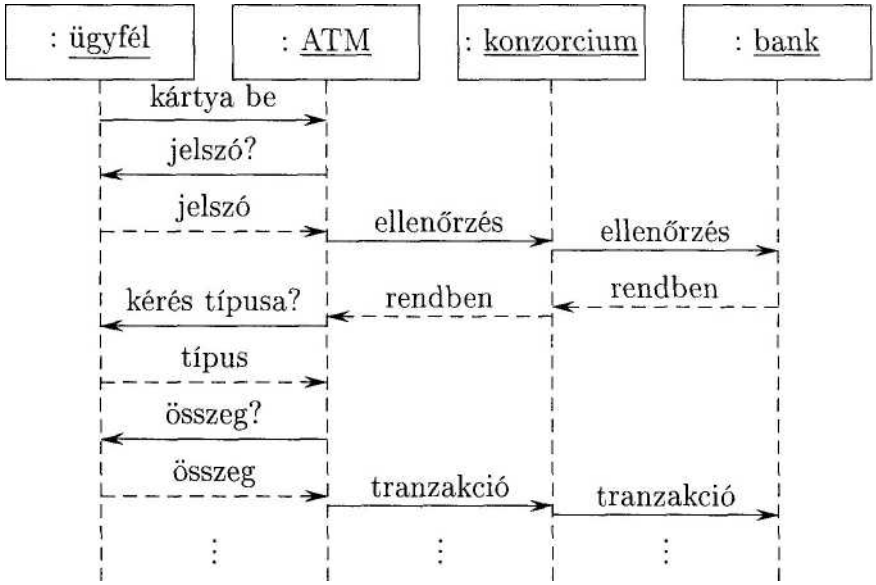
Példa a decentralizált vezérlésre a szokásos telefonkapcsolat szekvenciadiagramja (7.21. ábra), amely egyben szemlélteti az objektumok



7.21. ábra. Példa decentralizált vezérlésre: a telefonkapcsolat létrejötte

közötti üzenetek időbeli lejátszását is.

A következő példa legyen az ATM készülék működését szemléltető szekvenciadiagram, amikor szintén csak az üzenetek időben lejátszódó sorrendjét kívánjuk szemléltetni. Ez egy újabb példa a decentralizáltan vezérelt rendszerre (7.22. ábra).



7.22. ábra. Az ATM készülékek működését leíró szekvenciadiagram

7.5. A szekvenciadiagram kiegészítései

Az eddig bevezetett jelölések felhasználásával viszonylag egyszerű feladatok esetén is nagy méretű, bonyolult és nehezen áttekinthető szekvenciadiagramokhoz jutunk. Ezért célszerűnek látszik az eddigi jelöléseket kiegészíteni, hogy az áttekinthetőséget növeljük. Az UML jelenlegi változatában erre nincs szabványos módszer, legfeljebb utalások bizonyos lehetőségek megvalósítási módjaira. Mi az *MSC* (Message Sequence Chart) jelölési konvencióit használjuk, amelyek jó alapot adnak a rendszer teljesítményének elemzéséhez [11]. Ezek a kiegészítések

lehetővé teszik a hierarchikus szerkezet, a választás, az iteráció és a párhuzamosság ábrázolását.

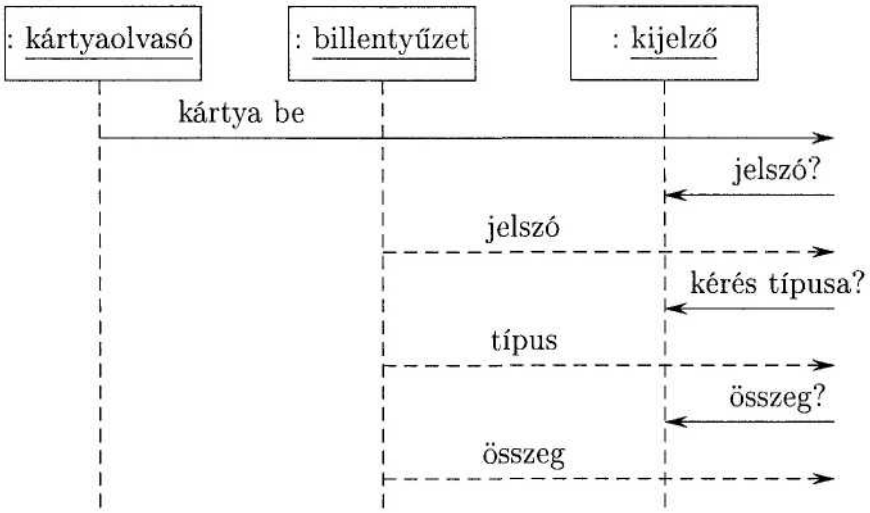
7.5.1. A hierarchia ábrázolása

Két módszerrel tudunk új szerkezeti szinteket bevezetni egy szekvenciadiagramba: példány dekompozícióval és hivatkozással.

Példány dekompozíció esetén a szekvenciadiagram egyik objektumát, illetve osztályszerepét egy másik szekvenciadiagramban fejtjük ki részletesen. A részletezés során az objektum vagy szerep helyét több objektum vagy szerep veszi át, és pontosítjuk, hogy melyik üzenetet melyik részobjektum küldi, illetve fogadja. Lényeges megszorítás, hogy az üzenetek sorrendje a dekompozíció során nem változhat meg.

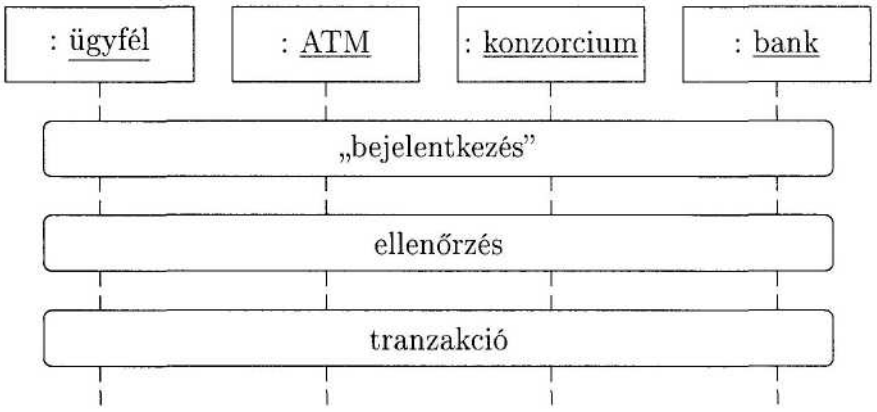
Például az ATM készülék működését leíró szekvenciadiagramban (7.22. ábra) az ügyfél objektum igény esetén tovább bontható kártyaolvasó, billentyűzet és kijelző objektumokra (7.23. ábra).

Hivatkozás esetén a szekvenciadiagram egy időintervallumnak meg-



7.23. ábra. Az ügyfél objektum felbontása kártyaolvasó, billentyűzet és kijelző objektumokra

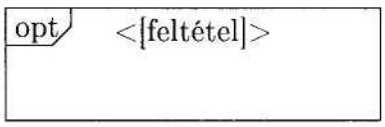
felelő részét kiemeljük egy külön diagramba. Az időintervallumnak megfelelő rész rendszerint egy tevékenységnek felel meg. A hivatkozást egy lekerekített sarkú téglalappal jelöljük, ami egy névvel (azonosítóval) rendelkezik. A kifejtést tartalmazó szekvenciadiagramot ezzel a névvel látjuk el. Az ATM készüléknél maradva a 7.22. ábra szekvenciadiagramjában például három részt különíthetünk el: „bejelentkezés", ellenőrzés és tranzakció (7.24. ábra). A részeknek megfelelő szekvenciadiagramokat nem adjuk meg, azok adódnak az eredeti diagramból.



7.24. ábra. Az ATM használatának hierarchikus részekre bontása hivatkozással

7.5.2. A választások ábrázolása

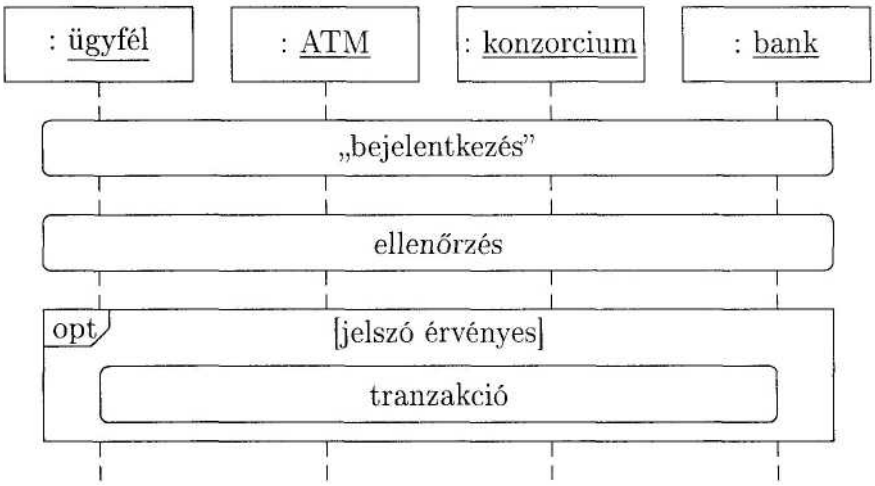
Egy szekvenciadiagramon belül lehetséges, hogy bizonyos rész üzeneteire csak akkor kerül sor, ha valamilyen feltétel teljesül. Ezt nevezzük opcionális résznek. A részt egy téglalapba zárjuk, a téglalap bal felső



7.25. ábra. Az opcionális rész jelölése

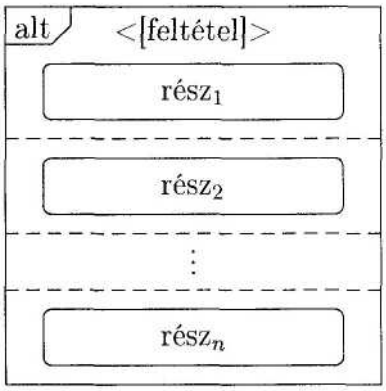
sarkába az *opt* szót írjuk, és a téglalap felső részének közepére írjuk a feltételt (7.25. ábra).

Például az ATM esetében csak akkor kell tranzakciót végrehajtani, ha az ellenőrzés után tudjuk, hogy a jelszó érvényes (7.26. ábra).

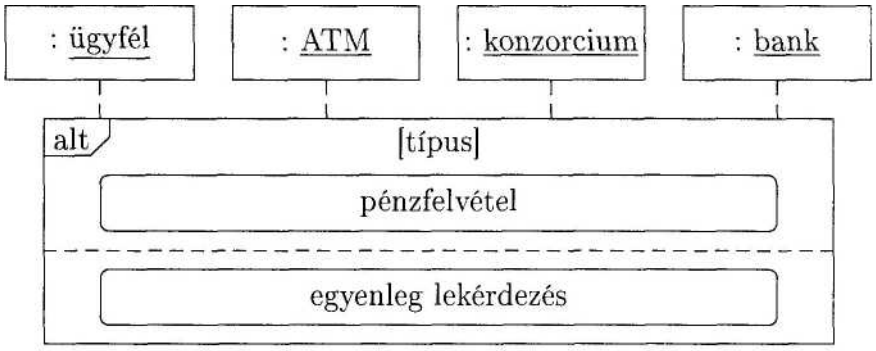


7.26. ábra. A tranzakció opcionális végrehajtása

Az is előfordulhat, hogy egy kifejezés értékétől függően eltérő részek lehetnek a szekvenciadiagramban. Ezt nevezzük alternatív résznek. Ezt a részt is téglalapba zárjuk, a bal felső sarokba az *alt* szó



7.27. ábra. Az alternatív rész jelölése



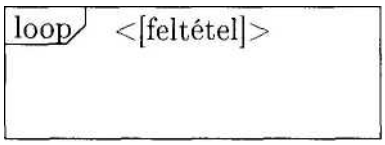
7.28. ábra. A tranzakciós rész egy alternatív szerkezet, amelynek feltétele a kérés típusa

kerül, a kifejezést a téglalap felső részének közepére írjuk, az egyes részeket pedig szaggatott vonallal választjuk el egymástól (7.27. ábra). Az ATM esetében mást kell tennünk akkor, ha a kiválasztott művelet a pénzfelvétel, illetve az egyenleg lekérdezése (7.28. ábra).

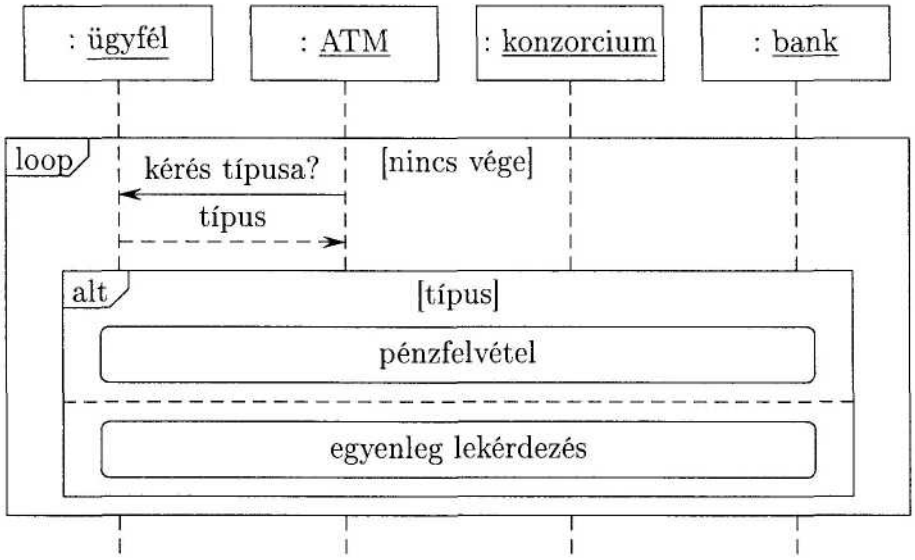
7.5.3. Az iteráció ábrázolása

Egy rendszer működése során bizonyos tevékenységek ciklikusan ismétlődhetnek. Ennek megfelelően a szekvenciadiagram egyes részeinek üzenetei is többször fordulhatnak elő. Az ilyen részt nevezzük iterációs résznek, ciklusnak. A szekvenciadiagramban ezt a részt egy téglalapba foglaljuk, a bal felső sarokba a *loop* szót írjuk, a ciklus feltételét pedig a téglalap felső részének közepére helyezzük (7.29. ábra).

Ha az ATM készülék példájában figyelembe vesszük, hogy több műveletet is végre lehet hajtani egy sikeres bejelentkezés után, akkor



7.29. ábra. Az iterációs rész jelölése



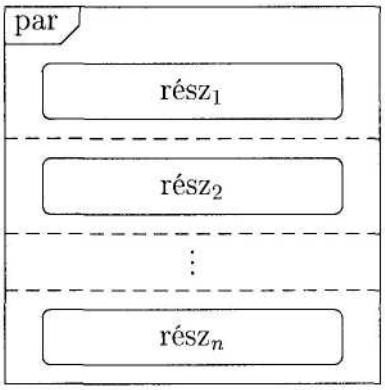
7.30. ábra. Az ATM készülék szekvenciadiagramjának tranzakciós része

a tranzakciós rész egy ciklus lesz, amelyben szerepel az előzőekben megismert alternatív rész (7.30. ábra).

7.5.4. A párhuzamosság ábrázolása

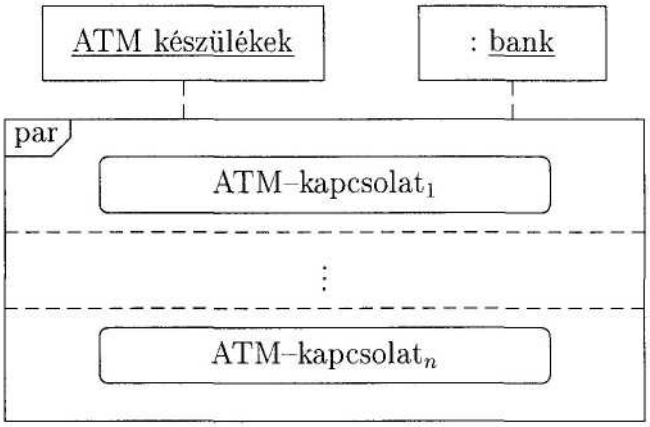
Párhuzamos, osztott rendszerek esetén bizonyos folyamatok egy időben zajlanak. Ezekben az esetekben a szekvenciadiagramban is biztosítanunk kell a párhuzamosság megjelenését. Az egymással egy időben zajló részeket egy téglalapon belül szaggatott vonallal választjuk el egymástól, a téglalap bal felső sarkába a *par* szó kerül (7.31. ábra). Az egyes részeken belül az üzenetek sorrendje rögzített, csak a különböző részekben szereplő üzenetek időbeliségére nem teszünk megkötést.

Az ATM-mel kapcsolatos példánál maradva, a gyakorlatot figyelembe véve egy bankban egyidejűleg több aktív ATM-kapcsolat lehet, és mindegyik kapcsolat tevékenységei párhuzamosan folynak. Ha a szekvenciadiagramban az ATM készülékeket összevontan kezeljük, és ezenkívül csak a bankot vesszük figyelembe, akkor a 7.32. ábrán lát-



7.31. ábra. A párhuzamosság jelölése

ható szekvenciadiagramhoz jutunk. Az egyes ATM-kapcsolatok üzeneteit az eddigiekből ismerjük.

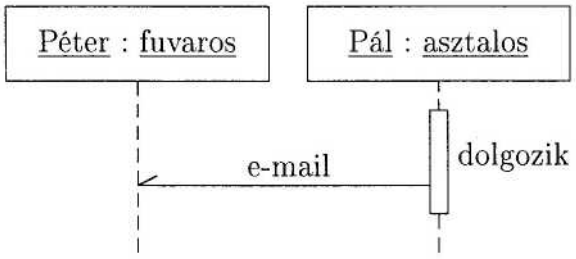


7.32. ábra. Több ATM-kapcsolat párhuzamos kezelése

7.1. feladat:

Pál asztalos, Péter fuvaros. Pál hozzákezd az asztal elkészítéséhez. Amikor már biztosan tudja, hogy mikor fejezi be a munkát, e-mail üzenetben közli Péterrel a szállítás időpontját. Készítsük el a szöveg alapján a szekvenciadiagramot!

Megoldás: Pál az asztalos osztály egy objektuma: Pál : asztalos. Hasonlóképpen: Péter : fuvaros. Pál esetében azt a szerepét, amikor az asztalt készíti, meg kell különböztetni attól, amikor nem készíti. Ezért Pál aktív szerepe legyen az, hogy asztalt készít! Pál életvonala során ebben az aktív szerepben - annak befejezése előtt - e-mail üzenetet küld Péternek. Az e-mail üzenet elküldése nem jár azzal a következménnyel, hogy Pálnak meg kellene szakítania a munkáját. Nem kerül blokkolt állapotba. Ez az üzenet aszinkron üzenet. Ha az üzenet tartalmától eltekintünk, akkor a 7.33. ábrán látható szekvenciadiagramhoz jutunk.



7.33. ábra. Az asztalos üzenetküldése a fuvarosnak

7.2. feladat:

Az óvodások tornagyakorlatát az egyik óvó néni, Jutka néni, sípjelekkel vezérli, miközben a másik óvó néni, Marika néni, zöld zászló magasba emelésével ad engedélyt az utasítás végrehajtására.

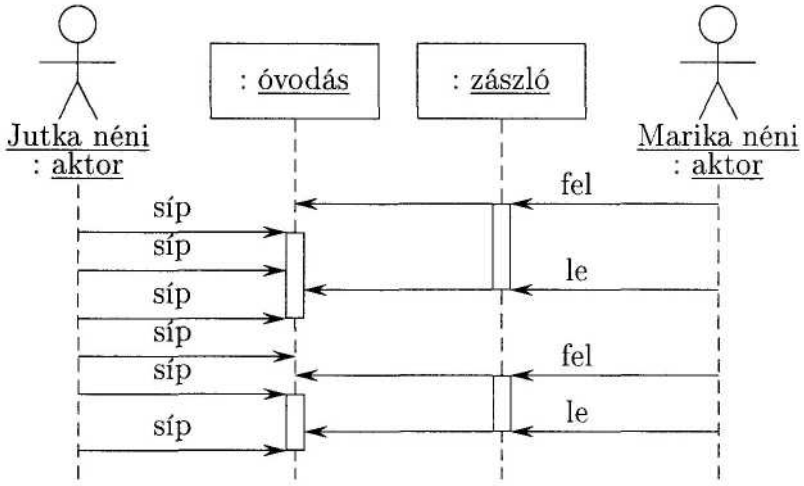
- Az óvodásoknak, ha ülnek, a sípjelre fel kell állniuk, de csak akkor, ha a zöld zászló is a magasban van.
- Az óvodásoknak, ha állnak, a sípjelre le kell ülniük, de csak ak kor, ha a zöld zászló nincs a magasban.

Készítsük el a játék menetét szemléltető szekvenciadiagramot!

Megoldás:

- A játékot Jutka néni és Marika néni vezérli. Ők az aktorok.
- Jutka nénitől jövő üzenet a sípszó.
- Marika nénitől jövő üzenet a zászló felemelése, illetve leengedése; röviden: fel, le.
- Az ovisok két szerepet játszanak: állnak vagy ülnek. Kézenfekvő ezek közül az egyiket aktív szerepnek választani. Legyen ez az álló szerep!
- A zászlónak két szerepe van: vagy magasban van, vagy leenged ték. A zászló esetében azt a szerepet választjuk aktívnak, amikor az a magasban van.

A fenti megállapítások figyelembevételével egy lehetséges szekvencia-diagramot szemléltet a 7.34. ábra.



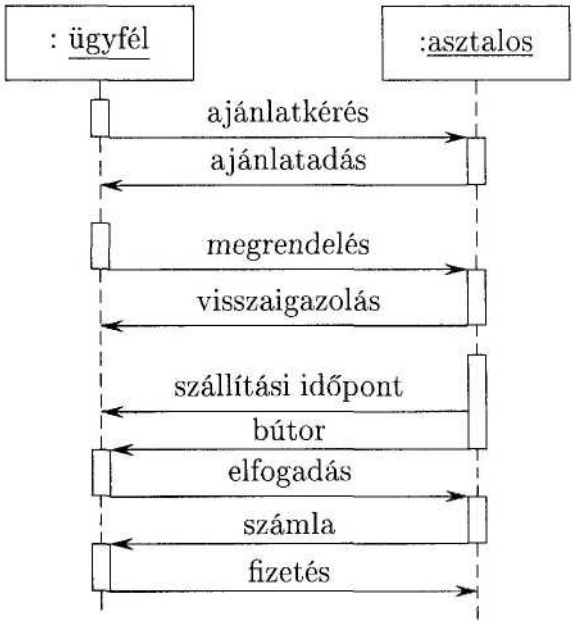
7.34. ábra. Az óvodások tornagyakorlatát leíró Szekvenciadiagram

7.3. feladat:

Készítsük el a következő probléma szekvenciadiagramját!

1. Az ügyfél konyhabútort kíván rendelni. Kialakítja az elképzelé sét, majd amikor elkészült vele, ajánlatot kér az asztalostól.
2. Az asztalos tanulmányozza a kérést, azután elküldi az ajánlatát az ügyfélnek.
3. Az ügyfél később, amikor az ideje engedi, megvizsgálja az ajánlatot, majd megrendelést ad át az asztalosnak.
4. Az asztalos ellenőrzi a megrendelést, majd a tanulmányozás után visszaigazolja azt.
5. Amikor a munka sorra kerül, az asztalos elkezdi a bútor elkészítését.
6. Amikor az asztalos már látja, hogy a rnunka mikor fejeződik be, értesíti az ügyfelet a bútor kiszállításának időpontjáról.
7. A megadott időpontban az asztalos kiszállítja az ügyfélnek az árut.
8. Az ügyfél ellenőrzi a kapott árut, kipróbálja azt, majd értesíti az asztalost, hogy elfogadja a bútort.
9. Az asztalos elkészíti a számlát, és megküldi azt az ügyfélnek.
10. Az ügyfél megszerzi a szükséges készpénzt, és kifizeti a számlát.

Megoldás: A leírás alapján értelemszerűen adódik a 7.35. ábra szekvenciadiagramja.



7.35. ábra. A bútorbeszerzés folyamatát leíró szekvenciadiagram

7.4. feladat:

Készítsük el a kisvasút mozdonyának vezérlésére írandó program modelljét, ahol a programnak a következő követelményeket kell kielégítenie!

1. A mozdony motorját, sebességváltóját és lámpáját kell egy-egy mikroprogramnak vezérelnie.
2. A sebességváltó lehet üres állapotban, előremenetben vagy hát ramenetben; de mind előremenetben, mind hátramenetben két sebességfokozatba kapcsolható.
3. A motor egyenlő sebességgel forog, ha áram alá helyezzük.
4. A lámpát akkor lehet csak bekapcsolni, ha a motor jár.
5. A rendszert a játékos kattintásokkal vezérli úgy, hogy külön gombot használ az előremenet, illetve a hátramenet vezérlésére.

6. Ha a játékos kattint egyet, akkor a kiválasztott gombnak megfe lelő sebességfokozatba kapcsol, és elindítja a motort.

7. Azonos gombon történő újabb kattintásokkal a sebességfokoza tokat váltogathatja.

8. Ellenkező irányhoz tartozó gombon történő kattintással a sebes ségváltó üresbe kerül, és a motor leáll. Ekkor a lámpa is elalszik, ha netán be volt kapcsolva.

9. A lámpát kettős kattintással lehet be- vagy kikapcsolni.

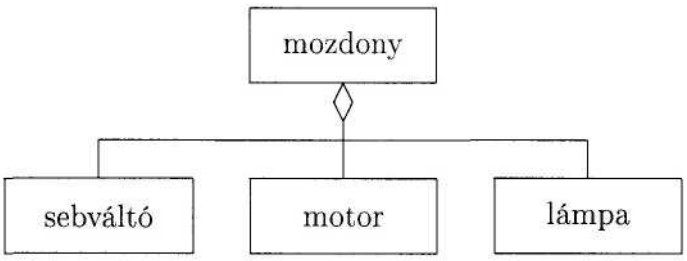
Készítsük el az osztálydiagramot, a szekvenciadiagramot és az állapotdiagramokat!

Megoldás:

Osztálydiagram:

Először a követelmények elemzése alapján határozzuk meg az objektumokat, amelyeket osztályokba foglaljunk össze, és rajzoljuk meg az osztálydiagramot! Vegyük sorba a követelmények szövegét!

a. A mozdony motorját, sebességváltóját és lámpáját kell egy-egy mikroprogramnak vezérelni. Itt aggregációról van szó: a mozdonyt a sebváltó, a motor és a lámpa alkotja (7.36. ábra).



7.36. ábra. A mozdony osztálydiagramja

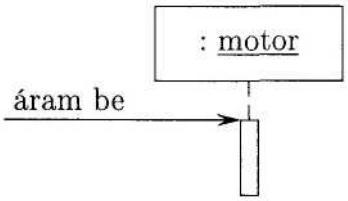
A statikus modell ismeretében készítsük el a követelmények elemzése alapján a szekvenciadiagramot a megoldás menetének szemléltetésére! Vegyük sorba a követelmények megfelelő pontjait!

Vezessük be az osztályszerepek jelölésére a következőket:

- : motor : az aktív rész jelöli, hogy a motor jár;
- : lámpa : az aktív rész jelöli, hogy a lámpa világít;
- előre$_1$: a sebességváltó előremenet első fokozatban van;
- előre$_2$: a sebességváltó előremenet második fokozatban van;
- üres : a sebességváltó üresben van;
- hátra$_1$: a sebességváltó hátramenet első fokozatban van;
- hátra$_2$: a sebességváltó hátramenet második fokozatban van!

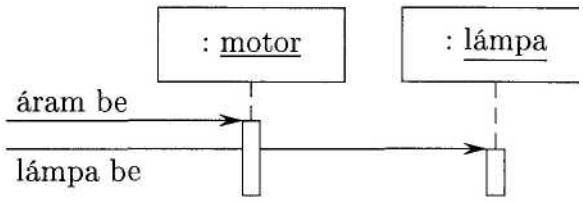
A sebességváltó esetén a rövidség kedvéért csak a szerep nevét adjuk meg, az osztály megnevezését elhagyjuk.

b. A motor egyenlő sebességgel forog, ha áram alá helyezzük, azaz a motort az áram bekapcsolásának eseménye aktivizálja (7.37. ábra).



7.37. ábra. A motor működése

c. A lámpát csak akkor lehet bekapcsolni, ha a motor jár. A lámpát tehát csak olyan időpontban lehet aktivizálni, amikor a motor is aktív (7.38. ábra).

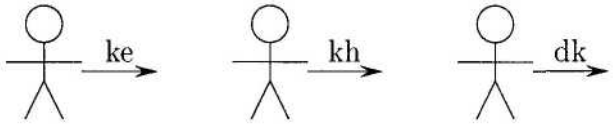


7.38. ábra. A motor és a lámpa működésének kapcsolata

d. A rendszert a játékos kattintásokkal vezérli úgy, hogy külön gombot használ az előremenet és a hátramenet vezérlésére. Jelöljük ezeket az üzeneteket röviden:

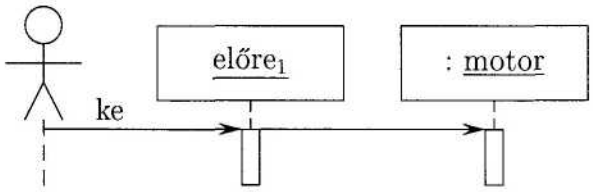
- kattintás előre : ke,
- kattintás hátra: kh,
- kettős kattintás : dk!

A játékos az aktor szerepét tölti be ezen üzenetek alatt (7.39. ábra).



7.39. ábra. A játékos üzenetei

e. Ha a játékos kattint egyet, akkor a kiválasztott gombnak megfelelő sebességfokozatba kapcsol, és elindítja a motort, ha az nem járt még (7.40. ábra).

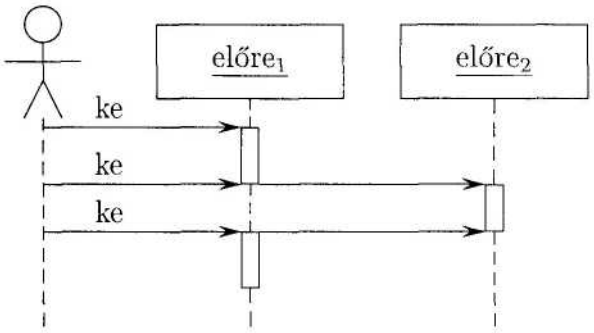


7.40. ábra. A játékos előre kattintás üzenetének hatása, ha a motor áll

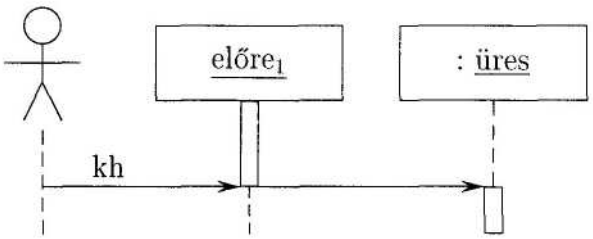
f. Azonos gombon történő kattintásokkal a sebességfokozatokat váltogathatja a játékos (7.41. ábra).

g. Ellenkező irányhoz tartozó gombon történő kattintással a sebes ségváltó üresbe kerül, és a motor leáll (7.42. ábra). Ekkor a lámpa is elalszik, ha netán be volt kapcsolva.

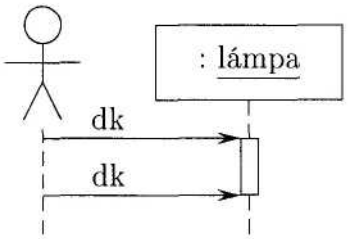
h. A lámpát a kettős kattintásokkal lehet be- vagy kikapcsolni (7.43. ábra).



7.41. ábra. A játékos előre kattintás üzenetének hatása a sebességfokozatra

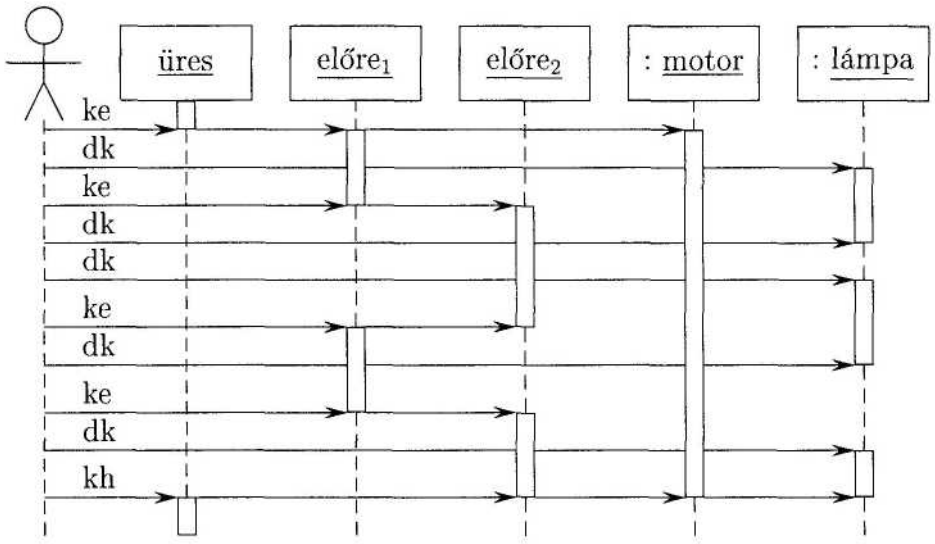


7.42. ábra. Ellentétes irányú gomb kattintásának hatása a sebességváltóra



7.43. ábra. A játékos kettős kattintásainak hatása a lámpára

Az eddigieket összevetve azt kapjuk, hogy a rendszer működését a 7.44. ábrán látható diagram szemlélteti. Itt az egyszerűség kedvéért eltekintettünk a hátrameneti állapotoktól, amelyek az előremeneti állapotokhoz hasonlóak.



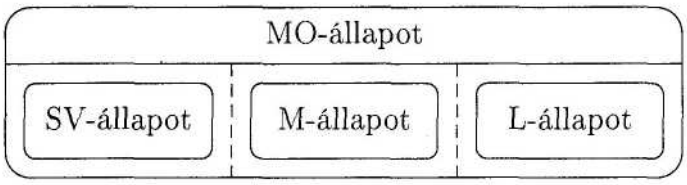
7.44. ábra. A rendszer működésének szekvenciadiagramja

Készítsük el most az állapotdiagramokat! i.

A mozdony állapotát (MO-állapot)

- a sebváltó állapota: SV-állapot,
- a motor állapota: M-állapot,
- és a lámpa állapota: L-állapot

együttesen határozza meg. Ezért a mozdony állapota ezen állapotok aggregációja (7.45. ábra).

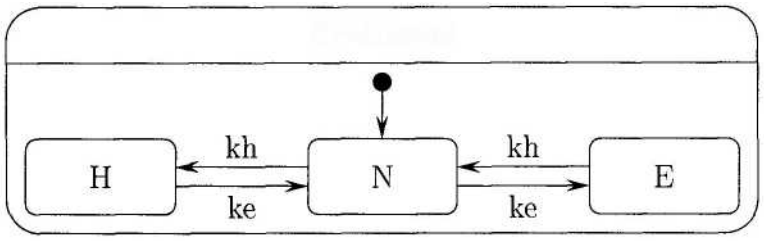


7.45. ábra. A mozdony állapotdiagramja

A sebességváltónak három állapota lehet:

- üres állapot : N,
- előremeneti állapot : E,
- hátrameneti állapot : H.

A sebességváltó állapota a fenti állapotok általánosítása, amelyek között az átmeneteket a megfelelő kattintások idézik elő (7.46. ábra).



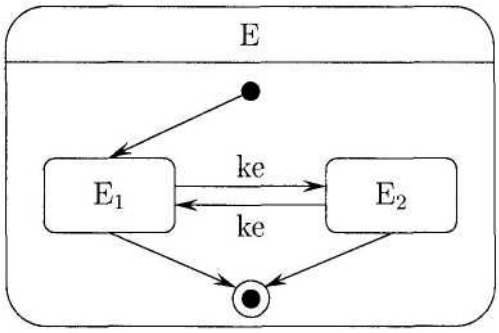
SV-állapot

7.46. ábra. A sebességváltó állapotdiagramja

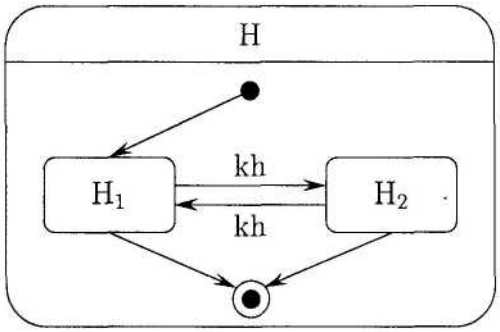
Az előremeneti állapotnak, valamint a hátrameneti állapotnak két-két konkrét állapota lehet:

- az előremeneti állapot (E) részállapotai:
  - —előremenet 1. fokozatban : E_1,
  - —előremenet 2. fokozatban : E_2;
- a hátrameneti állapot (H) részállapotai:
  - —hátramenet 1. fokozatban : H_1,
  - —hátramenet 2. fokozatban : H_2.

Ennek megfelelően kapjuk az előremeneti és a hátrameneti állapotok állapotdiagramjait (7.47. és 7.48. ábra), figyelembe véve, hogy a részállapotok közötti átmeneteket a megfelelő kattintások váltják ki.



7.47. ábra. Az előremeneti állapot (E) diagramja



7.48. ábra. A hátrameneti állapot (H) diagramja

A motornak két állapota lehet:

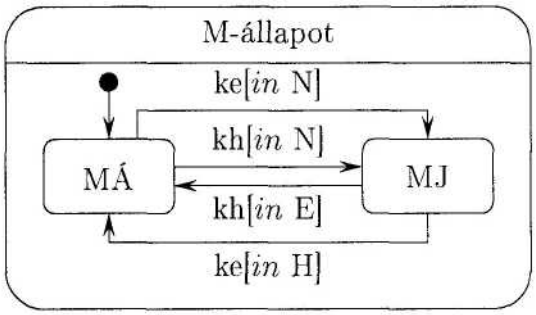
- a motor áll : MÁ,
- a motor jár : MJ.

A motor állapotai közötti átmenetek:

1. A motor elindításainak feltétele, hogy egy kattintáskor sebességváltó üres állapotban van, azaz
    - kh[in N] vagy
    - ke[in N].

2. A motor megállításának feltétele, hogy egy kattintáskor sebességváltó az ellenkező iránynak megfelelő állapotban van, azaz
   - kh[in E] vagy
   - ke [in H].

Ennek alapján a 7.49. ábra állapotdiagramjához jutunk.



7.49. ábra. A motor állapotdiagramja

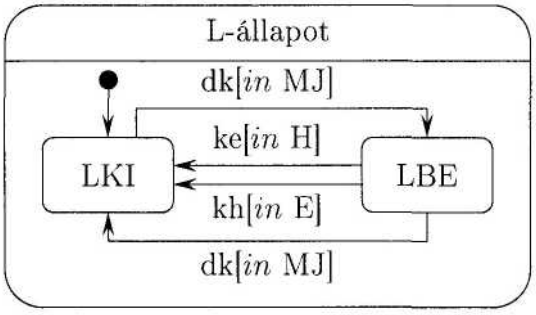
A lámpának két állapota lehet:

- a lámpa be van kapcsolva : LBE,
- a lámpa ki van kapcsolva : LKI.

A lámpa állapotai közötti átmenetek:

1. A lámpa bekapcsolásának feltétele:
   - kettős kattintással a LKI állapotból lépünk ki, amikor a motor jár: *dk[in* MJ].
2. A lámpa kikapcsolásának feltétele:
   - kettős kattintással LBE állapotból lépünk ki, amikor a mo tor jár: dk[in MJ];
   - megszűnik az áram, mert a menetiránnyal ellentétes gom bon kattintunk: kh[in E] vagy ke[in H].

Így a 7.50. ábrán látható állapotdiagramhoz jutunk.

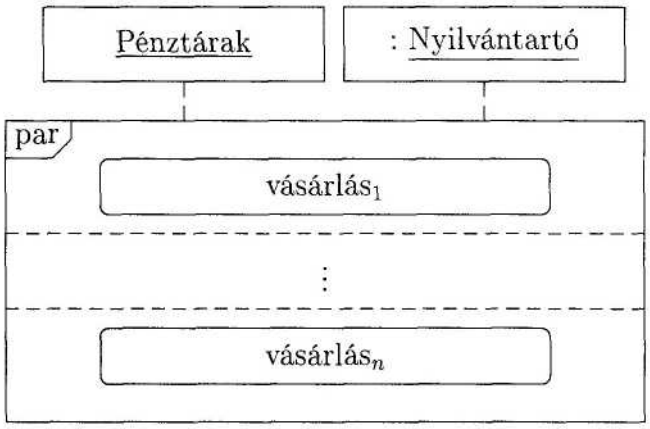


7.50. ábra. A lámpa állapotdiagramja

7.5. feladat:

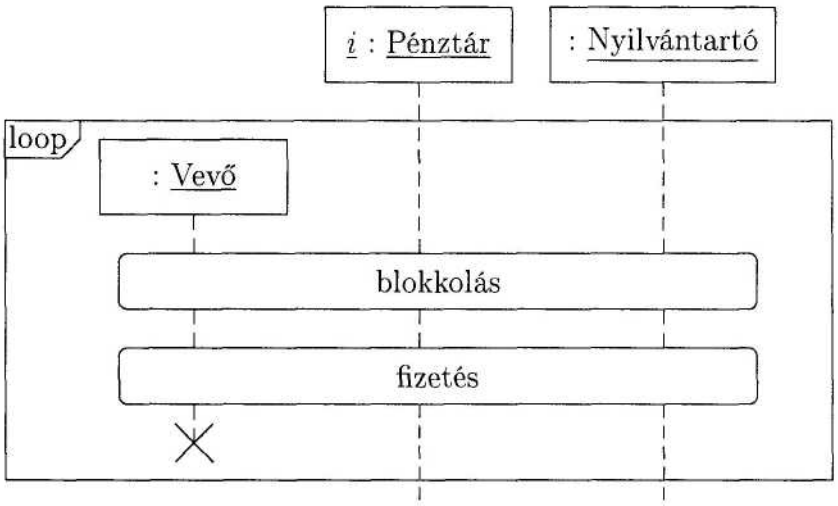
Készítsünk szekvenciadiagramot, amely a következő áruházi vásárlást modellezi! Az áruházban több pénztárnál fizethetnek a vevők, de egy vevő csak egy pénztárnál. A pénztárak párhuzamosan működnek. Egy vevő a kiválasztott pénztárnál felrakja az árukat a szalagra, ahonnan a pénztáros elveszi. A vonalkód alapján a központi nyilvántartásból megállapítja az áru nevét és árát, amit a blokkra nyomtat. Ha végzett a vevő összes árujával, akkor összegzi az árakat. Ezután a vevő fizet. A fizetés történhet készpénzzel vagy kártyával. Készpénzes fizetés esetén a vevő átadja a pénzt, és ha ez meghaladja a blokkon szereplő összeget a pénztáros visszaad. Kártyás fizetés esetén a vevő átadja a kártyát, amit a pénztáros „lehúz", azaz a banknak elküldi a kártyaszámot. Ezután megadja az összeget, amit a bank és a vevő is ellenőriz. Ha rendben van, akkor a vevő megadja a PIN-kódot, amit a bank ellenőriz, és az ellenőrzés eredményét továbbítja a pénztárosnak. Ha a kód rendben van, a pénztáros elismervényt nyomtat, amit átad vevőnek. Ezt a vevő aláírva visszaadja. Ezután a pénztáros átadja az elismervény másolatát. Mindkét fizetési mód esetén a pénztáros a blokkot is átadja. Ezután a pénztáros a következő vevővel foglalkozik.

Megoldás: Ha a pénztárakat összevonva kezeljük, akkor két szerepet különíthetünk el: a nyilvántartót és a pénztárakat. Ha a működő pénztárak száma n, akkor *n* vásárlási tevékenység zajlik párhuzamosan (7.51. ábra).



7.51. ábra. A szekvenciadiagram a legfelső szinten

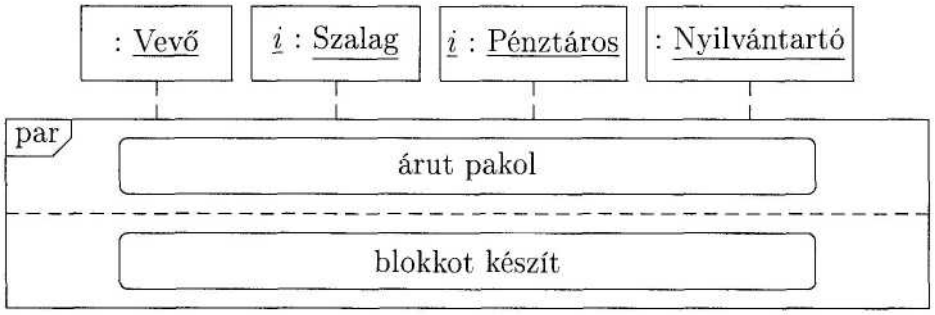
Ezt a diagramot kell finomítani, amelynek során példány dekompozícióval meg kell adnunk a pénztárakat részletesen, másrészt az egyes vásárlási tevékenységeket kell pontosítani (7.52. ábra). Az *i.* vásárlás pontosítása során a pénztárakat fel kell bontani az *i.* pénztárra és az



7.52. ábra. Az *i.* pénztár tevékenységei, vásárlási

ott található vevőkre. A tevékenység egy ciklus, amelyben a pénztár sorban szolgálja ki a vevőket. Egy kiszolgálás két nagy részre osztható: blokkolásra és fizetésre. Egy adott cikluson belül mindig egy vevővel foglalkozunk, aki tulajdonképpen ekkor „jön létre" és „szűnik meg".

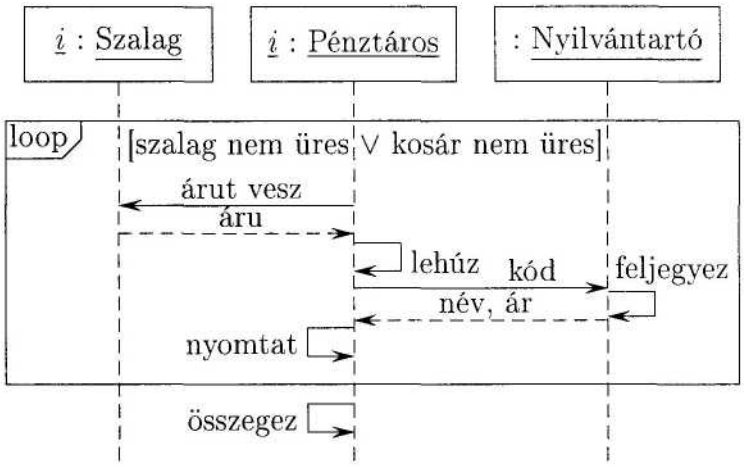
A blokkolás során a pénztárt tovább bontjuk egy pénztárosra és egy szalagra. A tevékenység során a vevő és a pénztáros párhuzamosan dolgozhat. A vevő a szalagra pakol, a pénztáros a szalagról veszi az árukat, és blokkol (7.53. ábra).



7.53. ábra. A blokkolás folyamata az *i.* pénztárnál

Az áruk pakolása egy ciklus, amelyben a vevő árut rak a szalagra, amíg ki nem ürül a kosara (7.54. ábra). A pénztáros amikor blokkot készít, egy ciklusban árukat vesz a szalagról. Az áru vonalkódját leolvassa, és a kódot a nyilvántartóhoz továbbítja. Innen megkapja a nevet és az árat, amit kinyomtat a blokkra. Közben a nyilvántartó

7.54. ábra. Az áru pakolása



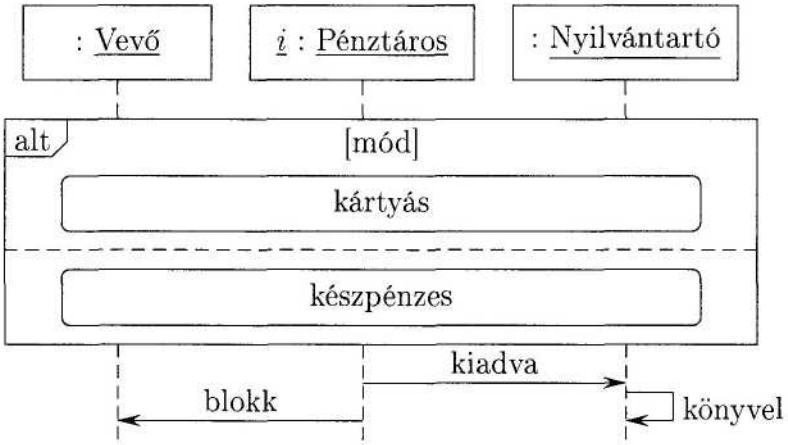
7.55. ábra. Egy blokk készítése

feljegyzi az árucikk kódját, hogy a vásárlás végén csökkenteni tudja a készletet. Az iteráció addig tart, amíg a pénztáros a vevő összes áruját fel nem dolgozza, azaz amíg a kosár nem üres, vagy a szalag nem üres. Ezután egy összegzést készít (7.55. ábra). Az egyszerűség érdekében nem vettük figyelembe az esetleges hibákat, amelyek például egy áru lehúzásakor keletkezhetnek. A továbbiakban sem kezeljük a hibás eseteket.

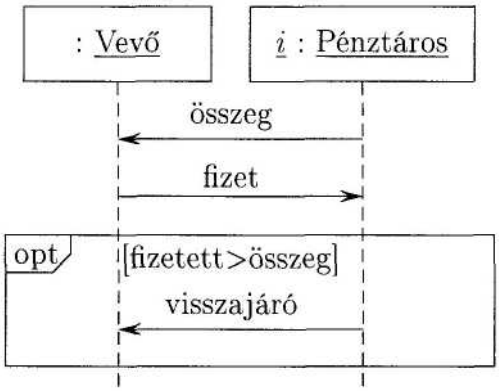
A fizetés lehet kártyás vagy készpénzes. Mindkét esetben az összeg átvétele után a pénztáros üzenet küld a nyilvántartónak, hogy az áruk tényleg ki lettek adva, majd a vevőnek átadja a blokkot. A nyilvántartó a feljegyzések alapján módosítja a készletet (7.56. ábra).

Készpénzes fizetés esetén a pénztáros megadja az összeget, ezután a vevő fizet. Ha a kifizetett összeg nagyobb, mint a blokkon szereplő összeg, akkor a visszajárót átadja a pénztáros (7.57. ábra).

Kártyás fizetés során a vevő átadja a kártyát. Ezt a pénztáros lehúzza, és a kártyaszámot elküldi a banknak. Ezután az összeget is, amelyről a bank eldönti, hogy kifizethető-e. Ezután a pénztáros megadja a vevőnek az összeget, amit az ellenőriz, és egy gomb megnyomásával nyugtáz a bank felé. Ezután a vevő megadja a banknak a



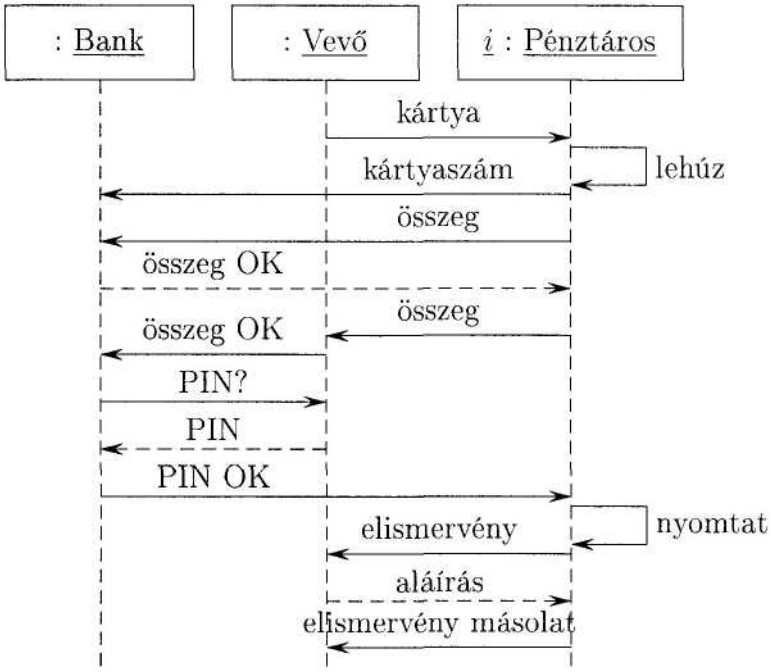
7.56. ábra. Fizetés az *i.* pénztárnál



7.57. ábra. A készpénzes fizetés szekvenciadiagramja

PIN-kódot, amit az ellenőriz, és erről értesíti a pénztárost. A pénztáros kinyomtatja az elismervényt, és átadja azt a vevőnek. A vevő ezt aláírva visszaadja. Végül a pénztáros átadja az elismervény másolatát (7.58. ábra).

Az olvasó feladata végiggondolni, hogy az esetleges hibák, rendellenességek kezelésével miként lehet a diagramokat kiegészíteni. A következőkben megadunk néhány lehetséges esetet, amelyeket egy valós rendszerben figyelembe kell venni.



7.58. ábra. A kártyás fizetés szekvenciadiagramja

- Az áru lehúzásakor nem sikerül a vonalkódot módosítani.
- Nincs elég készpénz a vevőnél, és árukat kell „visszavenni", amíg az összeg már megfelelő.
- Kártyás fizetés esetén nem sikerül a kártyát leolvasni, és néhány szor újra kell próbálkozni.
- Az összeg túllépi a kártyával fizethető maximális összeget, és vissza kell venni árukat, vagy nem lehet kártyával fizetni.
- Nem jó a megadott PIN-kód, és újra kell próbálkozni.
- Ha nem lehet kártyával fizetni, akkor meg kell próbálni a kész pénzes fizetést. Ha arra nincs lehetőség, az egész vásárlást törölni kell.

8. Funkcionális modell

Az eddigiekben láttuk, hogy a statikus modell arra válaszol, hogy mivel történik valami a megoldás során; a dinamikus modell eddig megismert részei pedig arra, hogy mikor történik valami. A dinamikus modell további részei, amelyeket együttesen funkcionális modellnek nevezünk, arra válaszolnak, hogy mi történik, és eközben milyen adatok és hogyan mozognak. Összefoglalva a funkcionális modell általában a feladat megoldása érdekében egymás után végrehajtandó tevékenységek folyamát írja le, a tevékenységek, leképezések között áramló adatokkal együtt.

A dinamikus modell részét képező interakciós diagramok közül az előző fejezetben a szekvenciadiagrammal ismerkedtünk meg. Ebben a fejezetben elsőként az interakciós diagramok másik fajtáját, az együttműködési diagramot mutatjuk be.

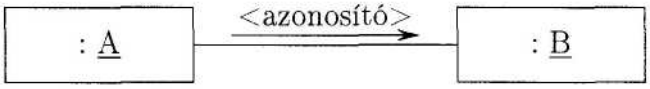
8.1. Együttműködési diagram

Az együttműködési (kollaborációs) diagram azt hivatott bemutatni, hogy miként működnek együtt az osztályok objektumai a probléma megoldásában, milyen üzenetek cseréje révén valósul meg ez az együttműködés.

Az együttműködés azok között az objektumok között valósul meg, amelyeket az objektumdiagramban asszociációs kapcsolatok kötnek össze. A diagram mutatja ezt az összekapcsolást és az ehhez tartozó üzenetváltásokat, ezért az együttműködési diagram az objektumdiag-

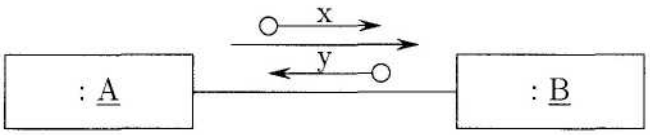
ram bizonyos értelemben vett kiterjesztésének tekinthető.

Az üzenet küldését egy nyíl mutatja, amely a társítás mellett kap helyet, és a címzett irányába mutat. Az üzenet azonosítója a nyíl mentén helyezkedik el (8.1. ábra).



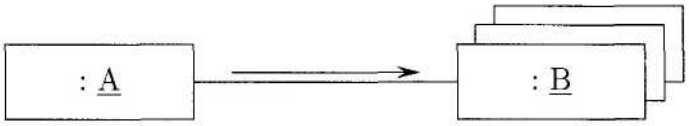
8.1. ábra. Üzenet jelölése az együttműködési diagramban

Az üzenetnek lehet argumentuma és eredménye. Ezeket egy kis körből induló nyíl mellett helyezzük el, ahol a nyíl az információ áramlásának irányát mutatja (8.2. ábra).



8.2. ábra. Üzenetek argumentuma és eredménye

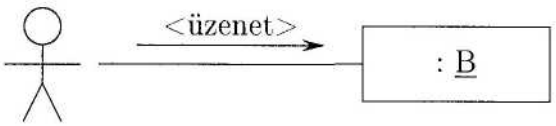
Az üzenetnek lehet egy osztályon belül több címzettje is. Ekkor ezt három, egymáshoz képest elcsúsztatott téglalappal jelöljük (8.3. ábra).



8.3. ábra. Üzenet több címzettel

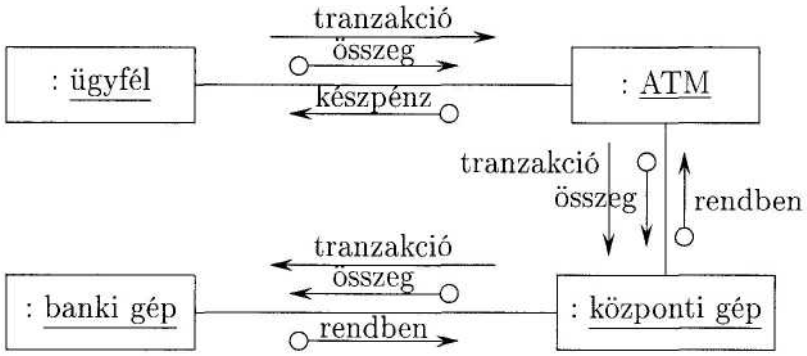
Az együttműködésben részt vehet egy rendszeren kívüli objektum is, amely aktor szerepet játszik. Az első üzenet, amely az együttműködést elindítja, ettől az objektumtól származik (8.4. ábra).

Első példaként tekintsük az ATM készüléken folytatott pénzügyi tranzakciót! Ennek keretében az ügyfél, a falon elhelyezett ATM készülék, egy központi számítógép és egy banki számítógép működik közre



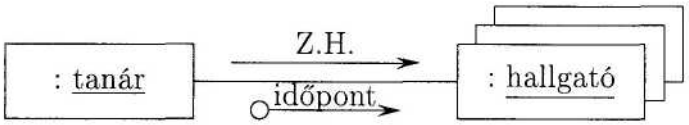
8.4. ábra. Aktor üzenete

a tranzakció lebonyolításában. Az egyszerűség kedvéért csak a pénz kiadásának igénylésével, az ahhoz kötődő adatok jogosságának ellenőrzésével, a kért összeg megnevezésének útjával és a készpénz kiadásával foglalkozzunk! Ekkor a 8.5. ábrán látható diagram írja le az objektumok együttműködését.



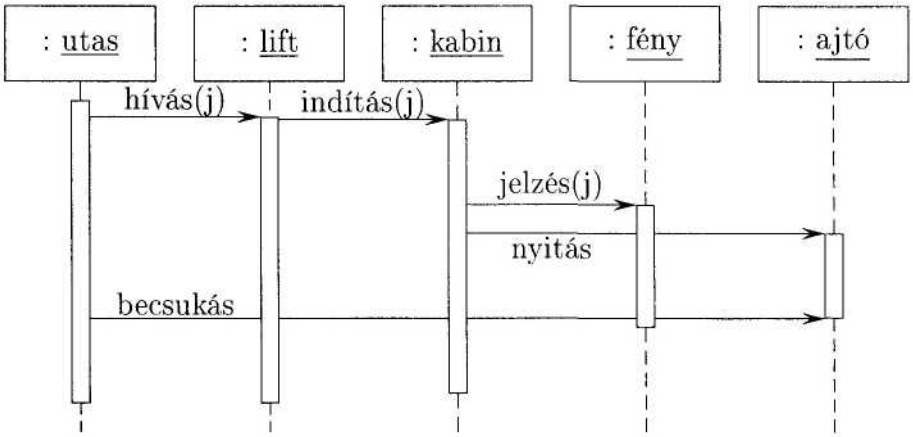
8.5. ábra. Az ATM készüléken történő pénzfelvétel

A következő példa legyen az, hogy egy tanár közli a hallgatókkal a zárthelyi időpontját (8.6. ábra)!

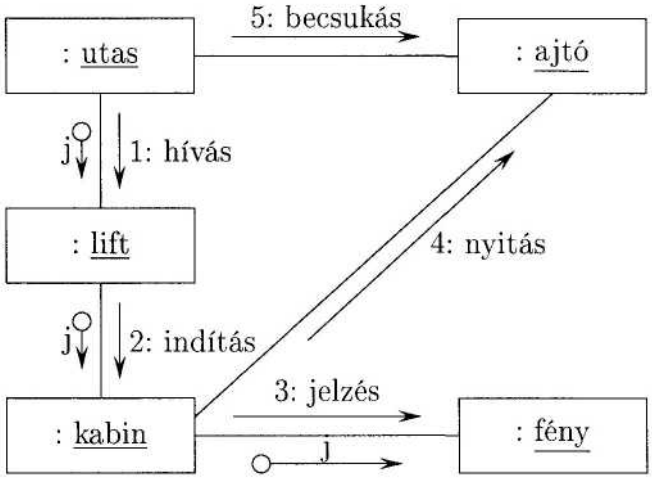


8.6. ábra. A tanár közli a Z.H. időpontját a hallgatókkal

Az utolsó példában egy lift működését leíró szekvenciadiagramot (8.7. ábra) és együttműködési diagramot (8.8. ábra) adjuk meg. A példában csak a lift hívásától a szállítás megkezdéséig terjedő részt



8.7. ábra. Lift hívásának szekvenciadiagramja



8.8. ábra. Lift hívásának együttműködési diagramja

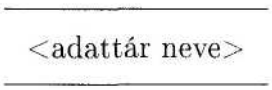
vizsgáljuk. Az utas egy adott *j* szintről hívja a liftet. Ennek hatására a kabin elindul, és ha megérkezik a szintre, akkor fényjelzést ad a lift. Ezután kinyitja az ajtót, amit az utas becsuk, miután beszállt.

A következőkben egy hagyományos, de az UML-ben nem szereplő diagramot mutatunk be, az *adatfolyam-diagramot*. Ez a funkcionális modell egyik hagyományos leírási módja.

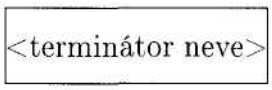
8.2. Adatfolyam-diagram

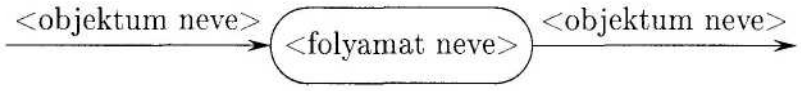
Az adatfolyam-diagram alapfogalmai a következők:

1. *Adattár* (data store): objektumai szerverek, amelyek tehát másokon nem végeznek műveletet. Jele:



2. *Terminátor :* objektumai aktorok vagy ágensek. Jele:

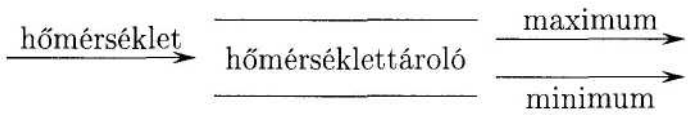




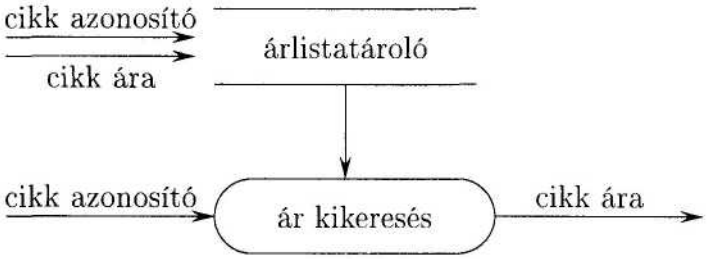
A következőkben megadjuk néhány egyszerű példa adatfolyam-diagramját. Elsőként tekintsük a hőmérséklettárolót, amelyben például napi hőmérséklet-értékeket tárolunk, és a nap végén kiválaszthatjuk a napi minimum és maximum hőmérsékletet (8.9. ábra)!

A következő példában az áruházban árusított cikkek árait tartjuk nyilván egy adattárban, és az előzetesen tárolt adatok alapján keressük ki az egyes cikkek árait (8.10. ábra).

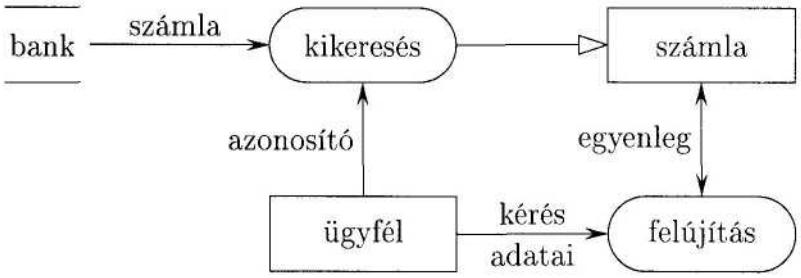
Vizsgáljuk meg egy banki számla felújításának folyamatát! Az ügyfél megadja az azonosítóját, amelynek alapján a bankból megtudjuk a



8.9. ábra. A hőmérséklettároló adatfolyam-diagramja



8.10. ábra. Az ár meghatározásának adatfolyam-diagramja

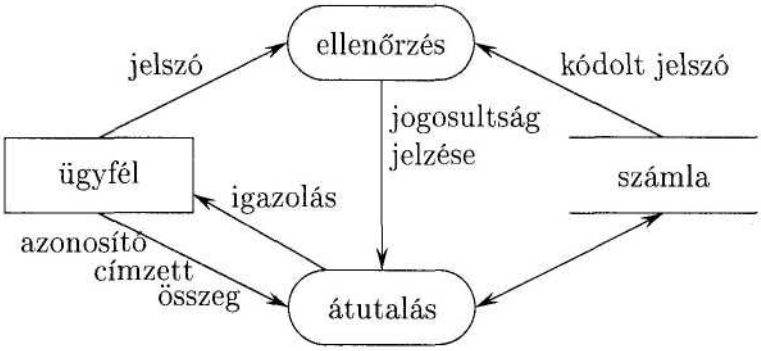


8.11. ábra. Számla felújítása a bankban

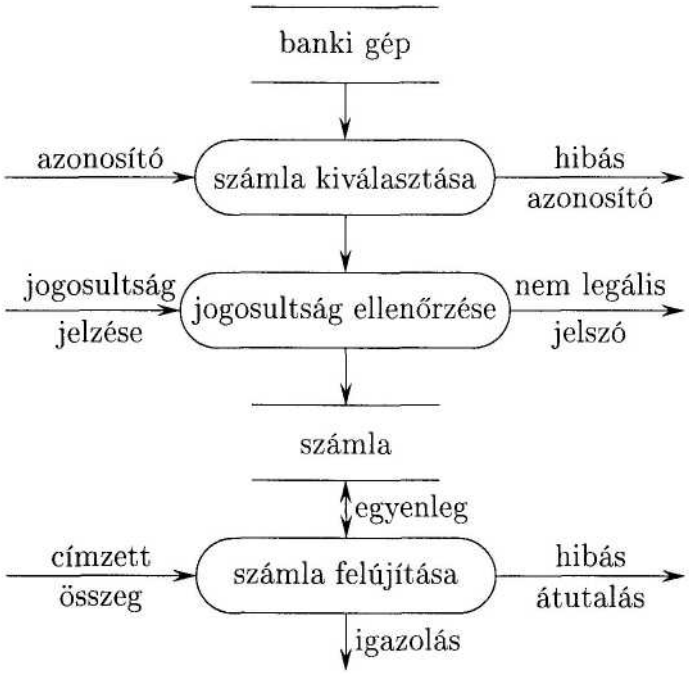
számlát. Az ügyfél megadja a kérés adatait, amiket felhasználva a kikeresett számla egyenlegét megváltoztatjuk. Látható, hogy a 8.11. ábra adatfolyam-diagramja jól leírja a folyamatot.

Végül rajzoljuk fel a banki átutalás adatfolyam-diagramját! Az ügyfél megadja az azonosítóját, a címzettet és az összeget, amelynek alapján a számlát módosítani lehet. A felújításhoz meg kell adni a jelszót is, amelyet külön ellenőrizni kell. Végül az ügyfelet értesíteni kell a felújításról. Ez a folyamat látható a 8.12. ábrán.

Az adatfolyam-diagram általában szintekre tagolt. Tekintsük például az átutalás folyamatát a 8.12. ábra diagramjában! Ezt tovább lehet bontani a számla kiválasztására, a jogosultság ellenőrzésére és a tranzakció végrehajtására. Itt már figyelembe vehetjük az esetleges hibákat is. A részletes megvalósítás lehet például a 8.13. ábrán látható adatfolyam-diagram.



8.12. ábra. A banki átutalás adatfolyam-diagramja



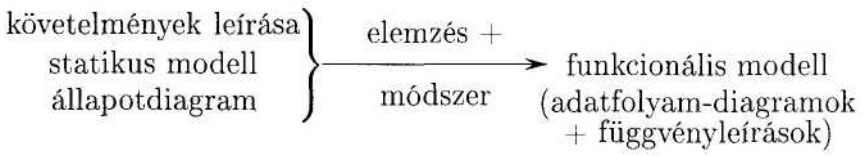
8.13. ábra. Az átutalás részletes adatfolyam-diagramja

8.3. A funkcionális modell megalkotása

A funkcionális modell létrehozásakor adott

- a követelmények leírása,
- a statikus modell és
- az állapotdiagram.

Ezek elemzésével és egy ismert módszer felhasználásával jutunk a funkcionális modellhez, amely tartalmazza az adatfolyam-diagramokat és



a függvényleírásokat (8.14. ábra).

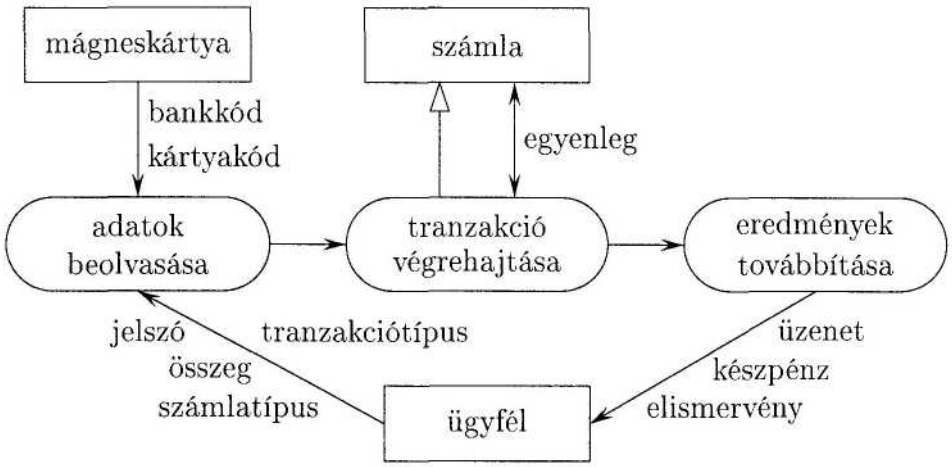
8.14. ábra. A funkcionális modell létrehozása

Az előállítási módszer négy fő lépésből áll:

1. Bemenő adatok és eredmény adatok meghatározása, kiegészítése osztályonként.
2. Adatfolyam-diagramok meghatározása.
3. A függvények leírása.
4. Optimalizációs kritériumok meghatározása.

Tekintsük példaként egy banki ügyfél készpénzfelvételét egy ATM automatából! Ekkor az adatfolyam-diagram a 8.15. ábrán látható diagram lesz.

Nézzük meg, ennél a példánál maradva, miként írhatjuk le a számlát felújító függvényt! Meg kell adnunk a nevét, hogy honnan hova képez, a leképezés módját és a műveletre érvényes megszorításokat.



8.15. ábra. Készpénzfelvétel ATM automatából

számla-felújítása

function:	számla-felújítása : (számla, összeg, tranzakciótípus) —> (készpénz, elismervény, üzenet, számla);
input:	számla, összeg, tranzakciótípus; készpénz,
output:	elismervény, üzenet, számla;
transformation:	- ha a kért összeg nagyobb, mint amekkora a számla összege, a tranzakciót törölni kell, és ki kell írni a következő üzenetet: "készpénzkeret túllépve"; - ha a kért összegre van a számlán fedezet, akkor végre kell hajtani a tranzakciót, és ki kell adni a készpénzt és az elismervényt az ügyfélnek;
constraints:	- ... - a kért összeg nem lehet - ... negatív; - ... - a kért összeg legfeljebb a hitelhatárral (hitel limit) lépheti át a számla összegét.

8.4. Aktivációs diagram

Az aktivációs diagram a hagyományos adatfolyam-diagram egy módosított változata, amely része az UML-nek. Az aktivációs diagram a probléma megoldásának lépéseit szemlélteti, a párhuzamosan zajló vezérlési folyamatokkal együtt.

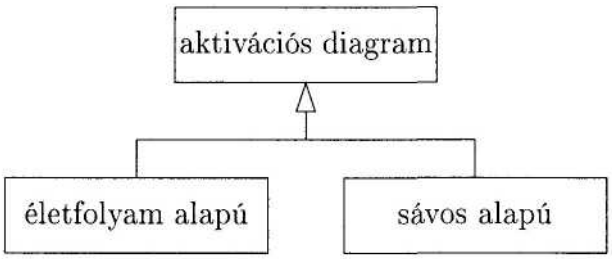
Az aktivációs diagram az állapotdiagram egy változatának is tekinthető', amelyben az állapotok helyére a végrehajtandó tevékenységeket tesszük, és az átmenetek a tevékenységek befejezésének eredményeként valósulnak meg.

Az aktivációs diagramot elsősorban az üzleti életben előforduló szervezetek munkafolyamatainak modellezésére használják.

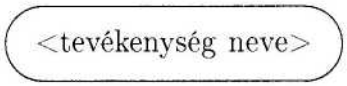
Kétfajta aktivációs diagramot szokás megkülönböztetni: az életfolyam alapút és a sávos alapút (8.16. ábra).

Az életfolyam alapú diagram esetén az objektumoknak életvonalakat feleltetünk meg, amelyeket ugyanúgy jelölünk, mint a Szekvenciadiagram esetében. Tehát az objektum egy téglalapba kerül, és az ebből kiinduló szaggatott függőleges vonal az életvonala. Erre lehet elhelyezni az objektum tevékenységeit, időben felülről lefelé haladva. Különböző objektumok tevékenységeinek sorrendjét nyilak jelzik.

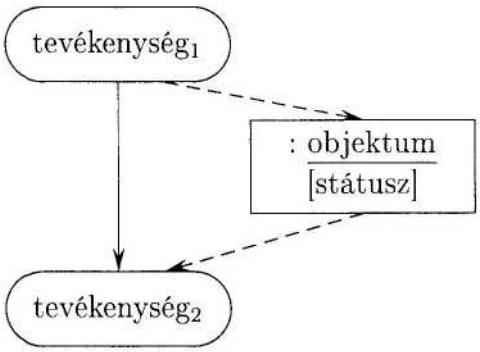
A sávos alapú diagramokban egy-egy függőleges sáv felel meg egy-egy objektumnak. A tevékenységek sorrendjét itt is nyilak adják meg. Az állapotdiagramból ismert kezdőállapot adja meg a kezdetet, a végállapot pedig a befejeződést.



8.16. ábra. Aktivációs diagramok fajtái



8.17. ábra. A tevékenységek jelölése az aktivációs diagramokban

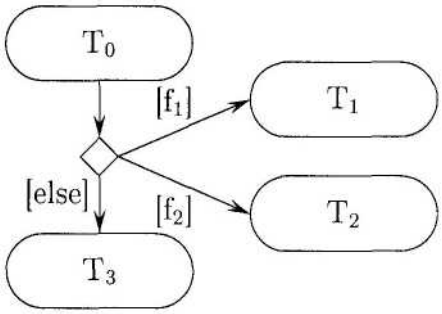


8.18. ábra. Tevékenységek sorrendjének és a továbbított adatoknak a jelölése

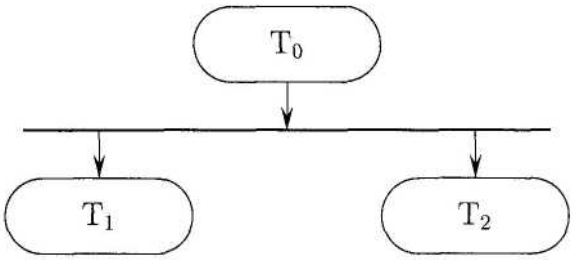
Az aktivációs diagram elemei a tevékenységek, folyamatok, amelyeket egy ovális téglalappal jelölünk (8.17. ábra). Ha egy tevékenységet egy másik tevékenység követ közvetlenül, akkor a két tevékenységet nyíllal kötjük össze. Ha adatot (objektumot) ad át egy tevékenység egy másik tevékenységnek, akkor a küldő tevékenységből szaggatott nyíl vezet az objektumot reprezentáló téglalaphoz, és a téglalaptól szaggatott nyíl mutat a fogadó tevékenységre. A téglalapban szögletes zárójelek között megadhatjuk az objektum állapotát, státuszát is (8.18. ábra).

Lehetőség van arra, hogy bizonyos feltételek teljesülése esetén eltérő tevékenységeket hajtsunk végre, illetve tevékenységek végrehajtását feltételekhez kössük. Ekkor egy rombuszt kell elhelyeznünk a diagramban, amelyből kivezető nyilakra írjuk a feltételeket (8.19. ábra). Az a tevékenység kerül végrehajtásra, amelyhez tartozó feltétel teljesül. Ha egyik feltétel sem áll fenn, akkor az „else" feltételhez tartozó tevékenységet kell végrehajtani, ha van ilyen eset.

A sávos alapú aktivációs diagram lehetőséget biztosít párhuzamos



8.19. ábra. Feltétel jelölése az aktivációs diagramokban

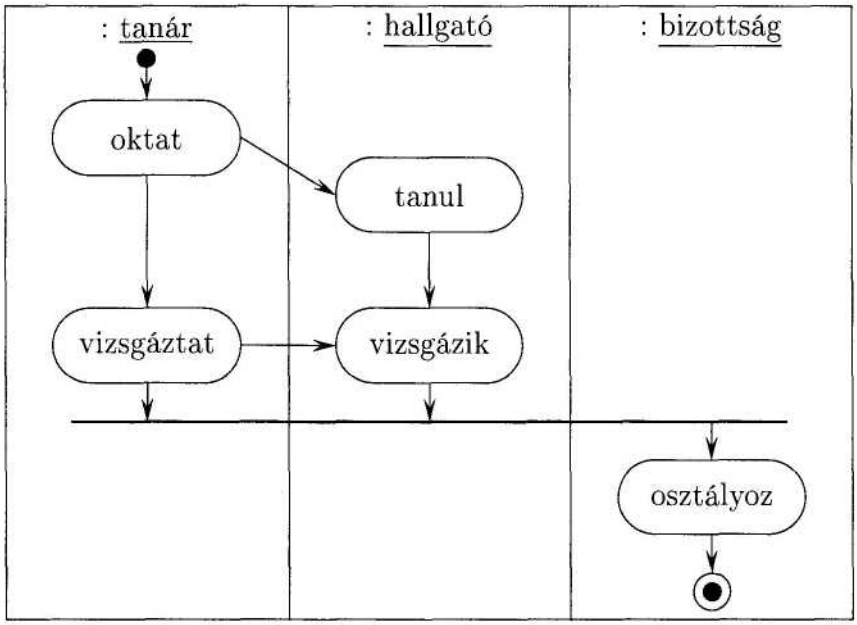


8.20. ábra. Folyamatok szinkronizációja

folyamatok leírására. Ekkor a folyamatok közötti szinkronizációt egy vastagított vízszintes vonallal jelöljük (8.20. ábra). A jelölés azt fejezi ki, hogy a vonalba mutató folyamatok mindegyikének be kell fejeződnie, mielőtt a vonalból kiinduló tevékenységek megkezdődnének.

A következőkben néhány példán keresztül szemléltetjük az aktivációs diagram jelölésrendszerét és alkalmazhatóságát.

Tevékenységek szinkronizációjára példa az egyetemi oktatók, hallgatók és vizsgabizottságok tevékenysége egy adott félévben. A tanár a szorgalmi időszakban oktat; a hallgató ekkor és a vizsgát megelőzően tanul. A vizsgán egyszerre vizsgáztat a tanár, és vizsgázik a hallgató. (A bizottság értelemszerűen a vizsgáztató tanárokból áll.) A vizsga végén, amikor a tanár is befejezte a vizsgáztatást és a hallgató is a vizsgázást, a bizottság osztályoz (8.21. ábra).



8.21. ábra. A záróvizsga aktivációs diagramja

A következő példákban az árurendelés és a házhoz szállítással összekötött áruvásárlás aktivációs diagramját adjuk meg.

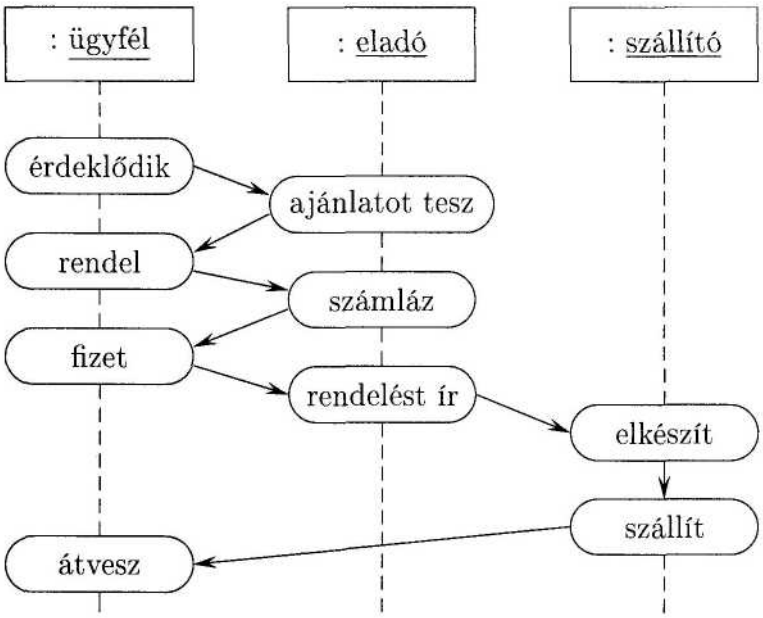
Az első esetben ügyfél először érdeklődik, és az eladó tesz egy ajánlatot. Ha az ügyfélnek ez megfelel, akkor az ajánlat alapján megrendeli az árut. Az eladó a megrendelés alapján elkészíti a számlát, amit elküld az ügyfélnek. Az ügyfél a számlát kifizeti, és ezután az eladó elkészíti a megrendelést, amit elküld a szállítónak. A szállító elkészíti, majd házhoz viszi az árut. Tehát a tevékenységek és a végrehajtók:

- érdeklődik - ügyfél,
- ajánlatot tesz - eladó,
- rendel - ügyfél,
- számláz - eladó,

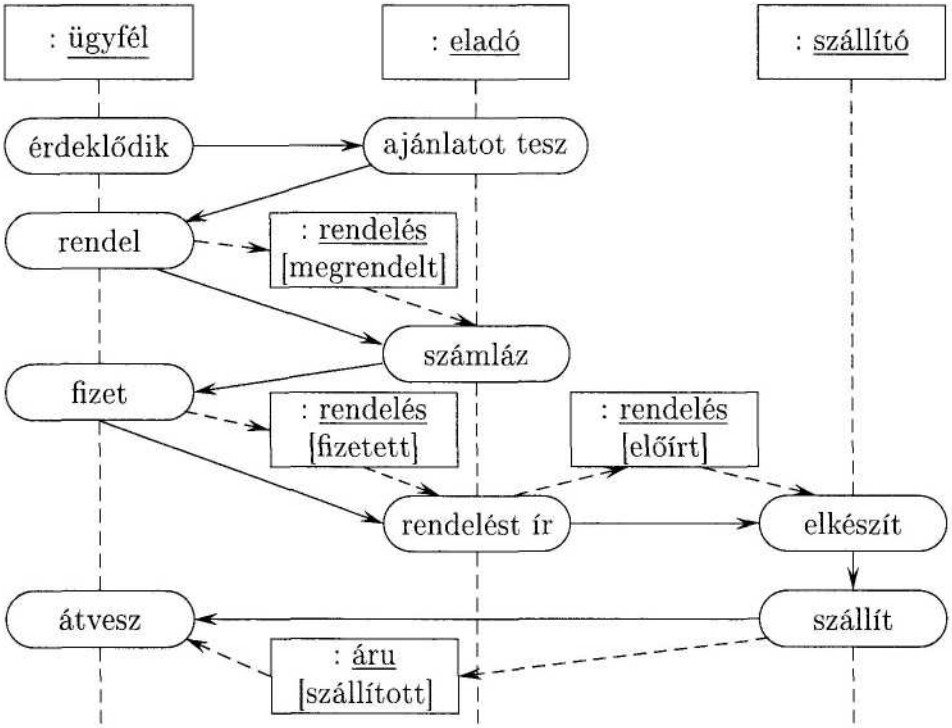
- fizet - ügyfél,
- rendelést ír - eladó,
- elkészít - szállító,
- szállít - szállító,
- átvesz - ügyfél.

A 8.22. ábrán látható életfolyam alapú aktivációs diagramról jól leolvasható a folyamat menete.

Ezt a diagramot kiegészíthetjük objektumfolyamokkal a teljesebb leírás érdekében (8.23. ábra). Két objektum jelenik meg ebben a diagramban.



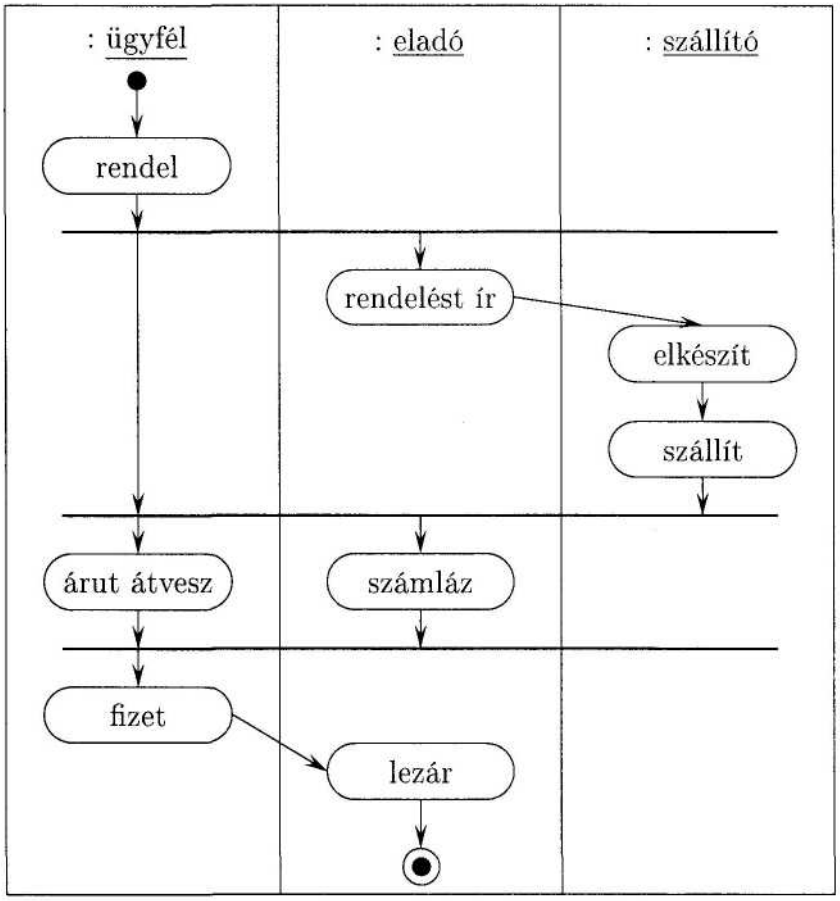
8.22. ábra. Árurendelés és házhoz szállítás életfolyam alapú aktivációs diagramja



8.23. ábra. Árurendelés és házhoz szállítás életfolyam alapú aktivációs diagramja, objektumfolyamokkal együtt

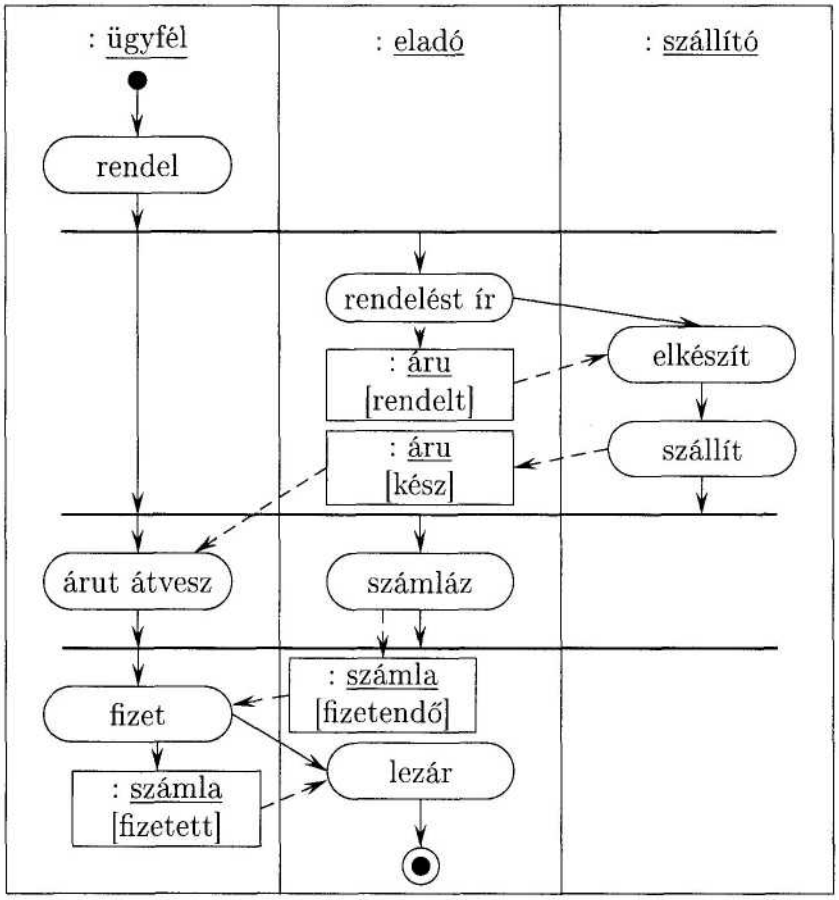
- A rendelés, amelynek három állapota lehet. Kezdetben az ügyfél megadja a kívánságait, ez a megrendelt állapot. Fizetés után az állapot fizetett lesz, majd az eladó küldi el a pontos előírásokat a szállítónak.
- Az áru, amely a szállítás után jelenik meg, és megfelel a rende lésnek.

Egy hasonló probléma aktivációs diagramjának a sávos változata látható a 8.24. ábrán, illetve a sávos változat az objektumfolyamokkal együtt a 8.25. ábrán. Ebben az esetben azonban a vásárlás utánvéttel történik, azaz a vevő csak az áru átvétele után kapja meg a számlát, és fizet; továbbá az ügyfél rendelésével indul a folyamat.



8.24. ábra. Árurendelés és házhoz szállítás sávos alapú aktivációs diagramja

Ennek megfelelően az első tevékenység a rendelés, amit az ügyfél hajt végre. Ezért az ügyfél sávjába kerül az állapotdiagramoknál megismert kezdeti állapot, és ebből vezet nyíl a rendelés tevékenységhez. A rendelés befejezése után az eladó megírja a rendelést, amit elküld a szállítónak. A szállító elkészíti az árut, és leszállítja azt. Ezután árut az ügyfél átveszi, az eladó ezzel egyidőben elkészíti a számlát. Ha mindkét tevékenység befejeződött, akkor az ügyfél fizet. Az eladó ezt

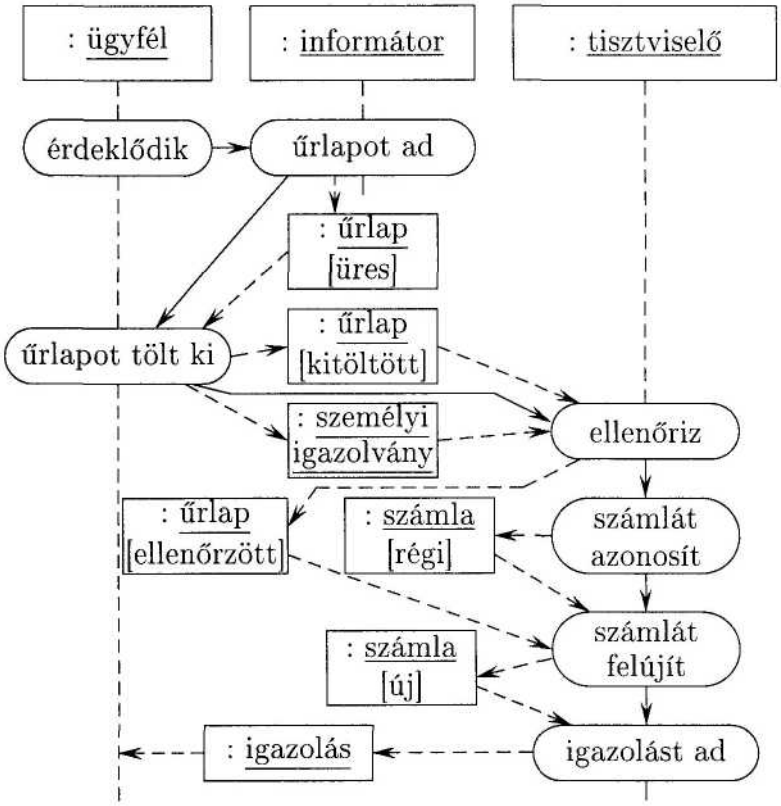


8.25. ábra. Árurendelés és házhoz szállítás sávos alapú aktivációs diagramja, objektumfolyamokkal együtt

követően lezárja a rendelést. Ez az utolsó tevékenység, ezért innen egy nyíl vezet a befejező állapotba.

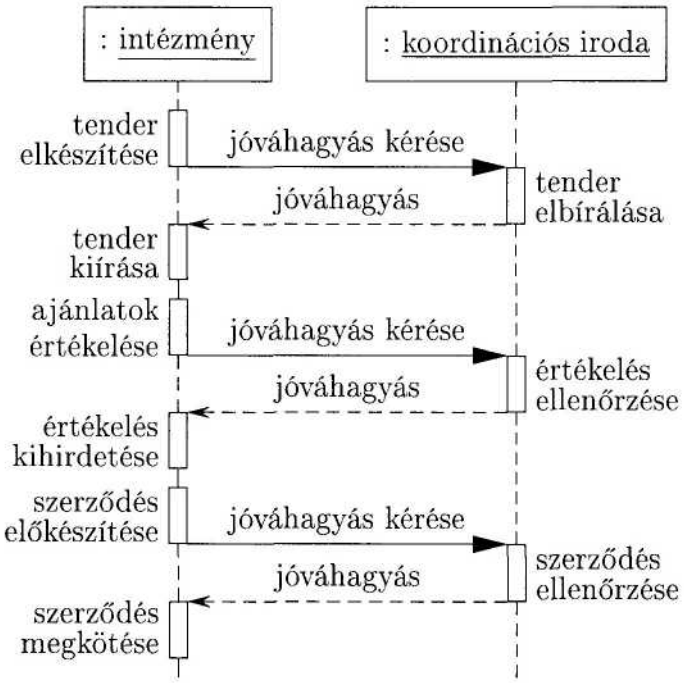
Két objektumot vettünk fel az objektumfolyamokkal kiegészített diagramban, az árut és a számlát. Az áru két állapota a „rendelt" és a „kész". A számlának is két állapota lehet: „fizetendő" és „fizetett".

A következő példa egy bankban megrendelt átutalást és annak végrehajtását leíró aktivációs diagram (8.26. ábra). Az ügyfél az információnál érdeklődik, ahol megkapja a megfelelő űrlapot. Ezt kitöltve, a személyi igazolvánnyal együtt, átadja a banki tisztviselőnek, aki ellenőrzi az űrlapot, kiválasztja a megfelelő számlát, és az átutalási űrlap alapján felújítja azt. Végül ad egy igazolást az ügyfélnek.



8.26. ábra. Banki átutalás megrendelése és végrehajtása

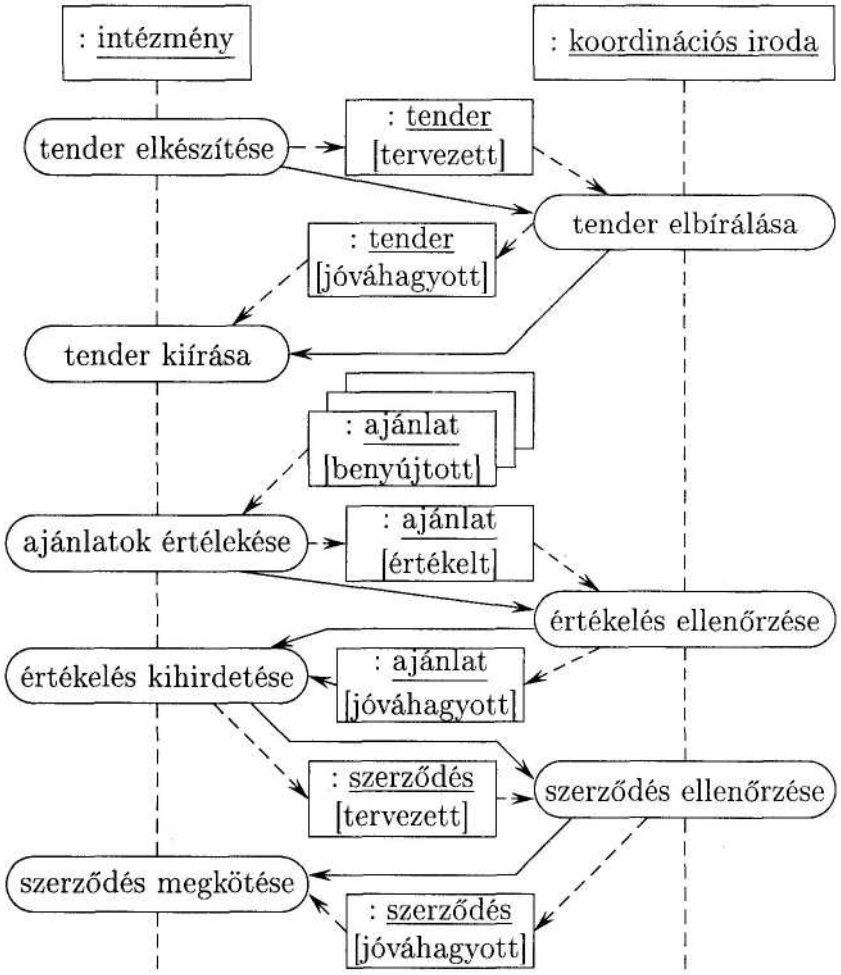
A következő példa a közbeszerzésekkel kapcsolatos. Először a közbeszerzési eljárás szekvenciadiagramját készítjük el, amely a jóváhagyott megvalósítási terv alapján a szerződéskötésig terjedő részt mutatja be (8.27. ábra). Az intézmény elkészíti a tendert, amit elküld



8.27. ábra. A közbeszerzési eljárás szekvenciadiagramja, jóváhagyott megvalósítási terv alapján, a szerződéskötésig

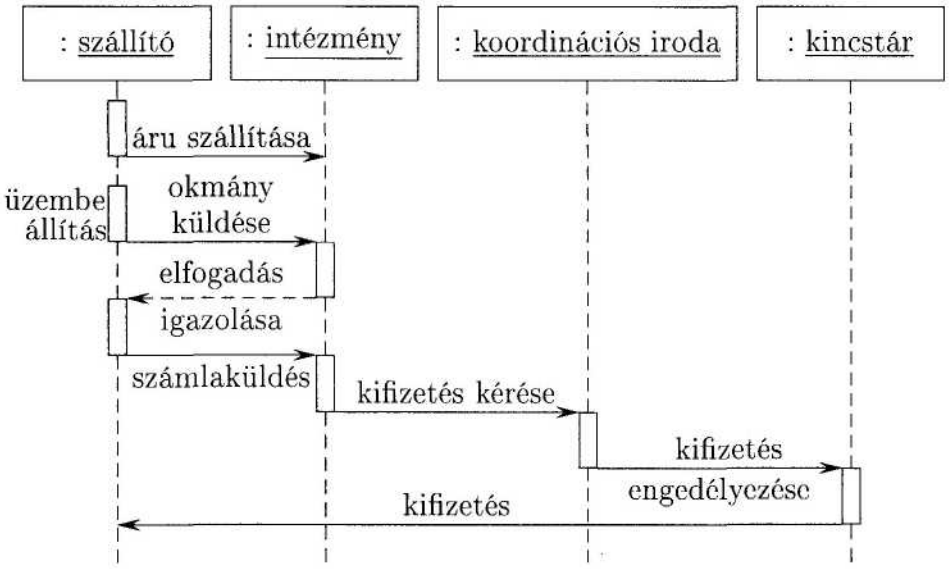
elbírálásra a koordinációs irodának. Ha az iroda engedélyezi, akkor kiírja a tendert, és az ajánlatok beérkezése után értékeli azokat. Az ajánlatok értékelését elküldi az irodának, ahol jóváhagyják a választást. Ezután az intézmény elkészíti a szerződés tervezetét, amit az iroda szintén ellenőriz. Ha elfogadja, akkor az intézmény megköti a szerződést.

Ezután az ugyanerre a részre vonatkozó életfolyam alapú aktivációs diagramot mutatjuk be, az objektumfolyamokkal kiegészítve. A tevékenységek megegyeznek a szekvenciadiagramban szereplő tevékenységekkel. Három objektum jelenik meg a diagramban: a tender, az ajánlat és a szerződés. A tendernek két állapota lehet: tervezett (miután az intézmény elkészítette) és jóváhagyott (ha az iroda elbírálta). Az ajánlat állapotai: benyújtott (a pályázók küldik az intézménynek), értékelt



8.28. ábra. A közbeszerzési eljárás aktivációs diagramja, jóváhagyott megvalósítási terv alapján, a szerződéskötésig

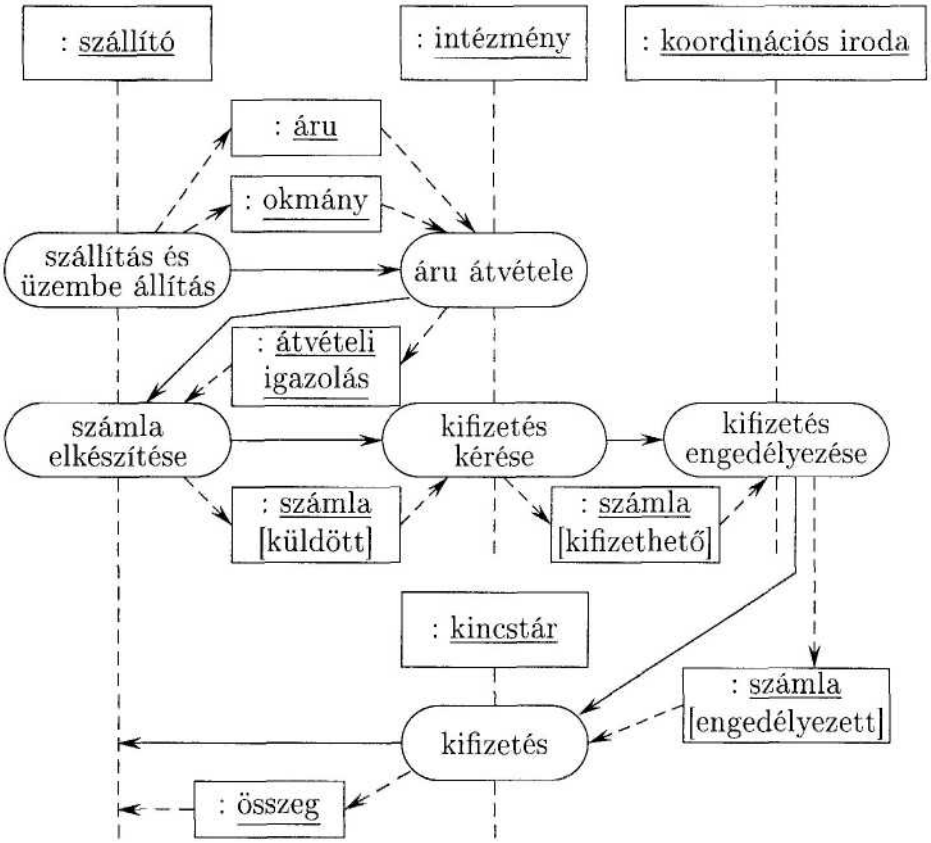
(az értékelés után) és jóváhagyott (az iroda ellenőrzése után). A szerződés két állapota: tervezett (az értékelés kihirdetését követően) és jóváhagyott (az iroda ellenőrzése után). Az eddigiek alapján a 8.28. ábra diagramjához jutunk.



8.29. ábra. A közbeszerzés során leszállított áru és szolgáltatás számlájának kifizetését szemléltető szekvenciadiagram

A közbeszerzés során leszállított áru és szolgáltatás számlájának kifizetését leíró szekvenciadiagramot készítjük el a következő lépésben (8.29. ábra). A szállító átadja az árut, majd üzembe is helyezi azt az intézménynél. Ezután elküldi az üzembeállítási okmányt, amelyen az intézmény igazolja az átvételt. A szállító elküldi a számlát az intézménynek, amit az továbbít az irodának. Az iroda engedélyezi a kifizetést a kincstárnál, ahol ezt meg is teszik.

Végezetül pedig ennek a résznek az életfolyam alapú aktivációs diagramját adjuk meg, az objektumfolyamokkal együtt. Az objektumfolyamokban megjelenő objektumok: áru, okmány, átvételi igazolás, számla, összeg. A számlának három állapota lehet: küldött, kifizethető, engedélyezett (8.30. ábra).



8.30. ábra. A közbeszerzés keretében leszállított áru kifizetésének aktivációs diagramja

8.5. Végrehajtási gráf

A létrehozandó rendszer működésének elemzése sokszor elkerülhetetlen, ha annak teljesítményével kapcsolatos kérdésekre kell felelnünk. Az eddig bevezetett eszközök erre nem alkalmasak. Ezért bevezetjük a *végrehajtási gráf* fogalmát [11], amely ugyan nem része az UML-nek, de könnyen létrehozható a már ismert diagramok felhasználásával.

A végrehajtási gráfot a rendszer működésének egyes forgatókönyveihez rendeljük. A forgatókönyv a használati esetek egy példánya.

Ha a rendszer teljesítményét akarjuk vizsgálni, akkor a teljesítmény szempontjából meghatározó forgatókönyveket vesszük figyelembe.

A végrehajtási gráf egy irányított gráf, amelynek csúcsai a rendszer tevékenységeit, élei pedig a tevékenységek közötti sorrendet ábrázolják. A rendszer tevékenysége egy feldolgozási lépés, amely valamilyen feladatot hajt végre a rendszerben. Ez lehet művelethívások gyűjteménye, illetve egyszerű utasítás is.

A gráf csúcsait megkülönböztetjük annak alapján, hogy milyen feldolgozási lépésnek feleltethetőek meg.

Egyszerű csúcs: egy funkcionális komponens, amely az adott szinten tovább nem bontható.

Kiterjesztett csúcs: olyan feldolgozási lépés, amelyet egy másik végrehajtási gráfban fejtünk ki részletesen.

Ismétlési csúcs: az ezt követő csúcsok n-szer ismétlődnek; a ciklus utolsó csúcsából ebbe a csúcsba mutat él.

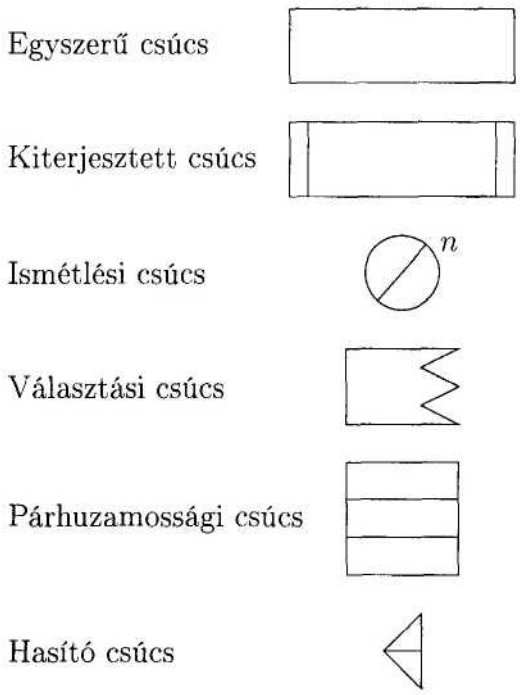
Választási csúcs: az ehhez csatolt csúcsok végrehajtása feltételes; minden esethez tartozik egy végrehajtási valószínűség.

Párhuzamossági csúcs: a csatolt csúcsok tevékenységei párhuzamosan hajtódnak végre, és mindegyiknek be kell fejeződnie, mielőtt ebből a csúcsból továbblépnénk.

Hasító csúcs: a csatolt csúcsok párhuzamosan futó új szálakat reprezentálnak, amelyek befejeződése előtt is továbbmehetünk.

A csúcsok jelölése látható a 8.31. ábrán.

A végrehajtási gráf előállítását a következő egyszerű példa szemlélteti. A hallgatók egy gépteremben levő számítógépeket használnak feladataik elvégzéséhez. Egy hallgató először belép a gép rendszerébe, majd a feladatait végzi el, végül kilép a rendszerből. Háromféle feladata lehet egy hallgatónak: internethasználat, fejlesztés és levelezés. Minden esetben kiválasztja a megfelelő programot a feladathoz. Az internetezést és a fejlesztést ezen a szinten nem bontjuk tovább. A



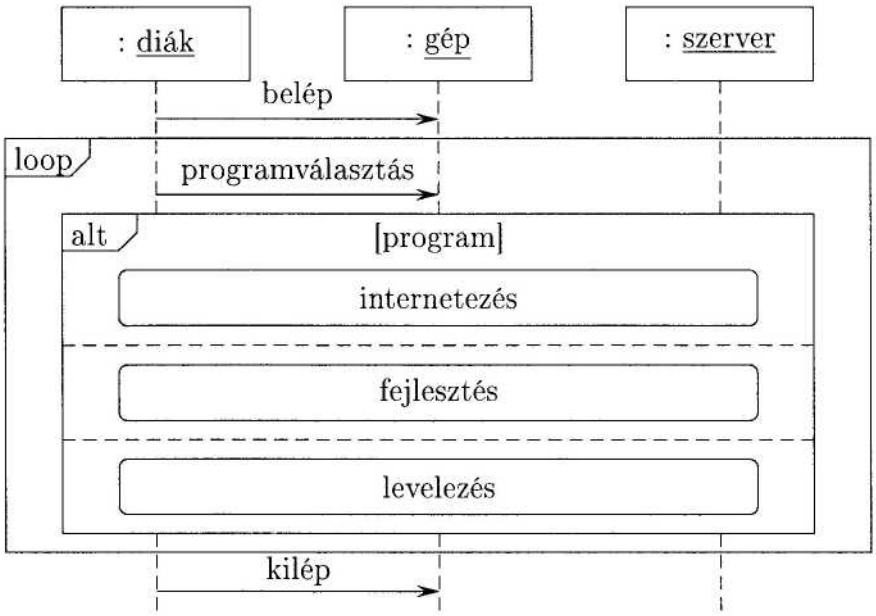
8.31. ábra. A végrehajtási gráf különböző típusú csúcsainak jelölése

levelezés során előbb bejelentkezik a szerverre, üzeneteket olvas vagy ír, a végén pedig kijelentkezik.

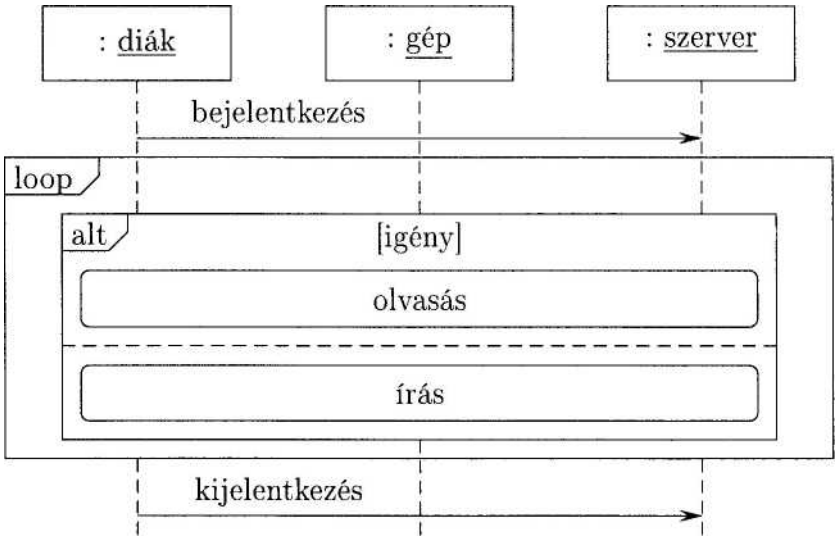
A leírás alapján elkészíthetjük akár a szekvenciadiagrarnot, akár az aktivációs diagramot. Mindkét diagram könnyen átalakítható végrehajtási gráffá. A szekvenciadiagram kiegészítéseit felhasználva egyszerűbb a probléma leírása, ezért mi ezt az utat választjuk.

A leírásból értelemszerűen adódik a 8.32. ábra szekvenciadiagramja, amelynek a levelezés részét tartalmazza a 8.33. ábrán látható diagram.

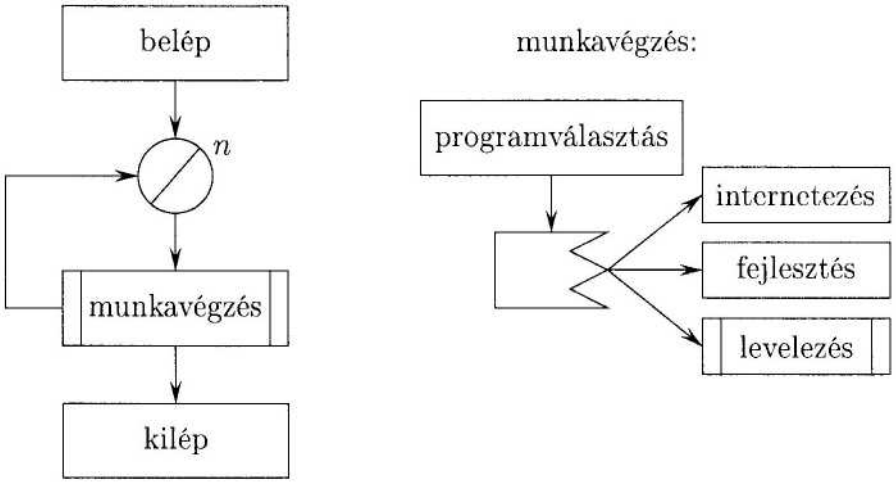
A Szekvencíadiagram alapján adódik a végrehajtási gráf, hiszen egy diagramban előforduló iterációnak egy ismétlési csúcs és a ciklusmagot leíró kiterjesztett csúcs felel meg, az alternatív szerkezetnek pedig egy választási csúcs. Ennek megfelelően a legfelső szintű végrehajtási gráf



8.32. ábra. Egy hallgató munkája a számítógépes laborban

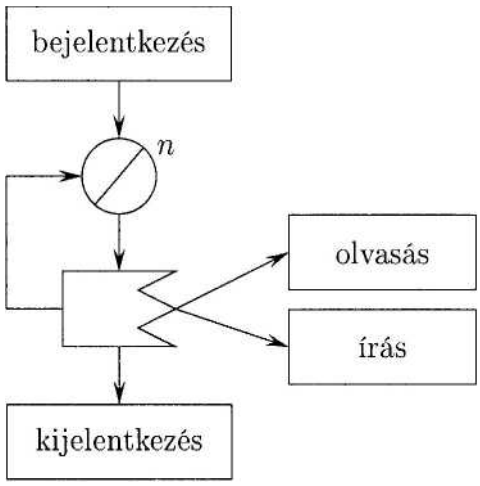


8.33. ábra. A levelezés folyamata



8.34. ábra. A legfelső szintű végrehajtási gráf (bal oldalon) és a munkavégzést leíró gráf (jobb oldalon)

látható a 8.34. ábrán. Ebben a munkavégzés egy kiterjesztett csúcs, amelyet az ábra jobb oldali része tartalmaz. Ezen belül a levelezést is külön ki kell fejtenünk (8.35. ábra).



8.35. ábra. A levelezés végrehajtási gráfja

Párhuzamos rendszerek esetén az eddig megismert csúcsokat ki kell egészítenünk szinkronizációs csúcsokkal, amelyek két csoportra oszthatók. Az egyik csoport csúcsai a hívó folyamatra vonatkoznak, a másik típusú csúcsok a hívott folyamathoz kapcsolhatók.

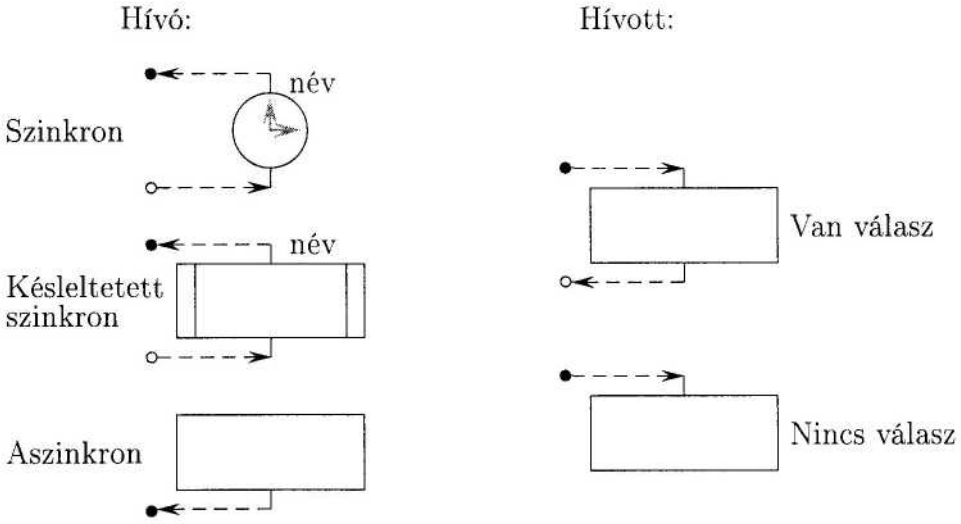
A hívó folyamatra vonatkozó csúcsokat adjuk meg először.

Szinkron hívás: a hívó a válaszra vár.

Késleltetett szinkron hívás: a hívó a válaszra vár, közben azonban a feldolgozás zajlik.

Aszinkron hívás: nincs válasz, nem kell várni.

A hívott folyamatra vonatkozó csúcsoknál csak azt kell jelölnünk, hogy van-e válasz. A jelöléseket tartalmazza a 8.36. ábra.



8.36. ábra. A végrehajtási gráf szinkronizációs csúcsai

A jelöléseket a következő példán szemléltetjük. Egy üzemben a műhelyt egy raktárból látják el a termeléshez szükséges anyagokkal, árucikkekkel. A raktárban *n* különböző árucikket tárolnak, amelyeket az $\{1,\ldots,n\}$ értékekkel azonosítanak. Minden cikkről nyilvántartják

a raktáron található mennyiséget. A raktári nyilvántartáson három művelet hajtható végre.

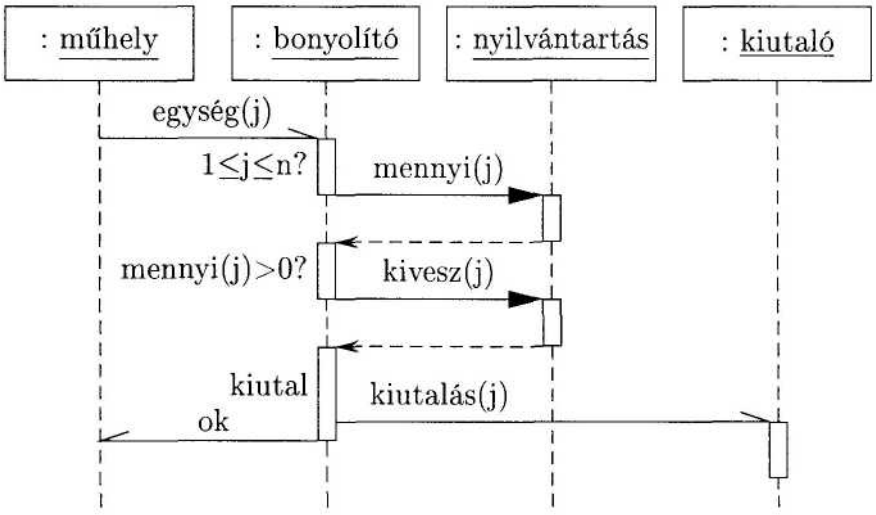
- Egy darab *j* azonosítójú áru elhelyezése a raktárba: *betesz(j).*
- Egy darab *j* azonosítójú áru kivétele a raktárból: *kivesz(j).*
- A *j* azonosítójú áru raktári készlete: *mennyi(j).*

(Az eddigiek alapján látható, hogy a raktári nyilvántartás tulajdonképpen egy zsák.)

A nyilvántartás egy központi számítógépen helyezkedik el. Ugyanezen a központi gépen található a kiutalást kezelő program (kiutaló) is. A *j* azonosítójú árucikk rendelését a műhely egy terminálon (PC) keresztül kezdeményezi az *egység(j)* kéréssel, amit a kéréseket lebonyolító programnak (bonyolító) továbbít. Ez egy hálózati szerveren található, és ellenőrzi a kérést. Ennek során megnézi, hogy olyan egységet kérnek-e, ami a raktárkészlet profiljába tartozik (1 <= *j* <= n), és ha igen, akkor van-e raktáron a kért egységből *(mennyi{j)* > 0). A teljesíthető rendelést a bonyolító kezeli, azaz kezdeményezi a készlet megváltoztatását, és a kiutalást; a rendelés teljesítéséről értesíti a műhelyt.

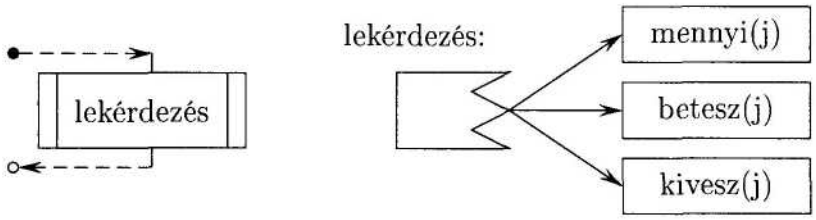
Először a szekvenciadiagramot készítjük el. Ebben négy objektum játszik szerepet: a műhely, a bonyolító, a nyilvántartás és a kiutaló. A műhely indítja el az üzeneteket az igény megadásával. A kérés teljesítése szempontjából más szerepe nincs, ezért nem is jelölünk az életvonalán aktív szakaszokat. Várhatóan még akkor kér árut, amikor a termelés nem akad meg, ezért ezt egy aszinkron hívással modellezhetjük. A bonyolító aktív szakaszai sorrendben: a kérés ellenőrzése, a raktári készlet ellenőrzése, kiutalás könyvelése. A nyilvántartáshoz küldött üzenetei szinkron üzenetek, a válasz megérkezéséig nem folytathatja a tevékenységét. A kiutalónak és a műhelynek aszinkron módon küldhet üzenetet. A szekvenciadiagram megadásakor feltesszük, hogy a kérés teljesíthető (8.37. ábra).

A Szekvenciadiagram felhasználásával elkészítjük a végrehajtási gráfot. Három objektum esetében kell ezt megadnunk. Ezek a bonyo-

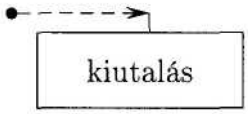


8.37. ábra. Egy teljesíthető kérés szekvenciadiagramja

lító, a nyilvántartás és a kiutaló. A nyilvántartás minden művelete, nevezzük ezeket együttesen lekérdezésnek, egy hívott objektum választ adó művelete lesz. A lekérdezésen belül három művelet lehet, amelyeket nem bontunk tovább (8.38. ábra). A kiutalónak egy válasz nélküli művelete van, a kiutalás (8.39. ábra).

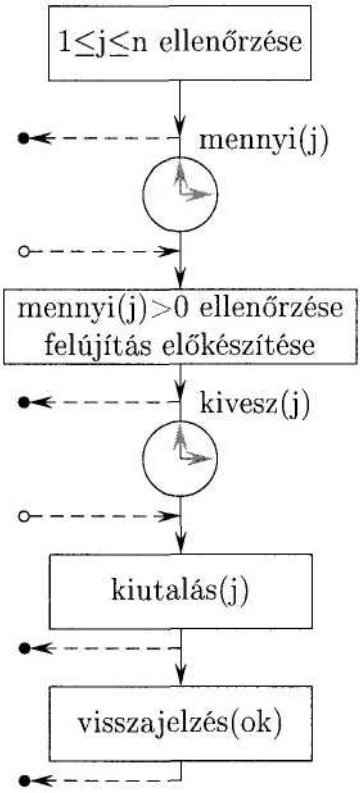


8.38. ábra. A nyilvántartás végrehajtási gráfja



8.39. ábra. A kiutaló végrehajtási gráfja

A bonyolító esetében az első tevékenység egy ellenőrzés, amit a nyilvántartás *mennyi* műveletének szinkron hívása követ. Ezután az eredményt ellenőrizni kell, és a felújítást elő kell készíteni. Ezt követi a nyilvántartás *kivesz* műveletének szinkron hívása. Végül a kiutalás és a visszajelzés aszinkron végrehajtása következik (8.40. ábra).



8.40. ábra. A bonyolító végrehajtási gráfja

9. Implementációs szempont szerinti diagramok

A rendszer elkészítésekor figyelembe kell venni az implementációra vonatkozó megkötéseket, korlátozásokat. Az UML ezek leírására két diagramot vezetett be a komponens- és az alrendszerdiagramot.

9.1. Komponensdiagram

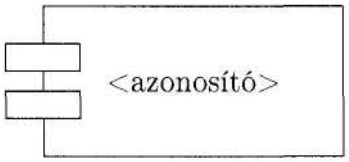
A komponensdiagram a probléma megoldására szolgáló rendszer tulajdonságait implementációs szempont szerint fejezi ki. A komponensdiagram alapfogalmai:

- komponens,
- reláció.

A *komponens* a rendszer egy fizikailag létező és kicserélhető része, feltéve, hogy a kicseréléséhez az új komponens csatlakozási felületét a környezettel konform módon valósítjuk meg, azaz a környezethez az új komponens csatlakozási felületét hozzáillesztjük.

A komponens informális definíciója:

1. A komponensnek van azonosítója.
2. A komponens az objektumok egy osztálya, amely a fizikai ob jektumokat definiálja.



9.1. ábra. A komponens jelölése komponensdiagramokban

3. A fizikai objektumok a szoftvermodulok fejlesztési vagy végre hajtási formáiban léteznek.

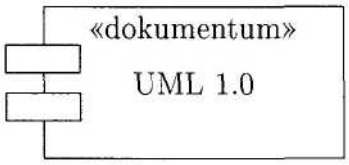
4. A fejlesztési forma lehet szöveges forma, a fordítás fázisaiban létező kódformák valamelyike.

5. A végrehajtási forma a modul végrehajtásra kész formája.

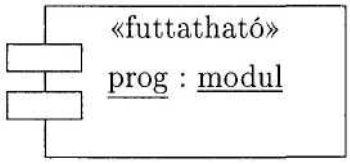
6. A komponenshez tartozhat egy sztereó típus, amelyeknek jelen tése rögzített, és a létezési formát fejezi ki. Például: «dokumentum», «forrás», «bináris» stb.

A komponens jelölése a 9.1. ábrán látható.

Első példaként tekintsük az UML 1.0 változatának egy dokumentumát! A megfelelő komponens a 9.2. ábrán látható. A következő példa



9.2. ábra. Az UML 1.0 változatának dokumentuma

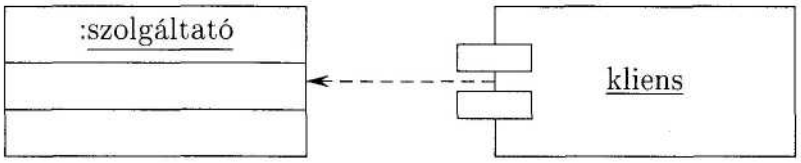


9.3. ábra. Végrehajtásra kész „prog" modul

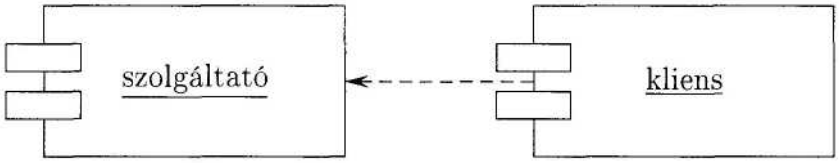
legyen egy „prog" nevű, végrehajtásra kész formában megjelenő modulnak megfelelő komponens (9.3. ábra)!

A komponensek közötti relációkat két csoportba sorolhatjuk:

- fejlesztés során fennálló reláció (9.4. ábra),
- meghívási reláció (9.5. ábra).



9.4. ábra. A szolgáltatást nyújtó komponens egy osztály, amely csak a fejlesztés során létezik



9.5. ábra. Egy kliens komponens és az annak szolgáltatást nyújtó komponens között fennálló reláció

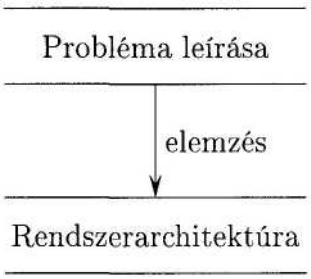
Megjegyzés: 9.4. és 9.5. ábrában a szaggatott nyíl függőségi viszonyt fejez ki, amely az UML jelölésben másutt is értelmezve van.

9.2. A rendszer statikus szerkezete

Bonyolult, összetett feladatok sokszor egymással lazán összefüggő részfeladatokból állnak. A részfeladatok önállóan megoldhatók, de a megoldás során feltétel lehet a másik részfeladat megoldásának állapota, esetleg az egyik részfeladat megoldásához igénybe vesszük a másik

részfeladat megoldásának eredményeit. Ilyen alrendszerekből álló feladat például az operációs rendszer a számítógépen, amely olyan alrendszerekből áll, mint a fizikai alrendszer, amely a hardverrel kapcsolatos feladatokat látja el; a logikai alrendszer, amely a hardver szolgáltatásait nyújtja a felhasználó számára; a fájlkezelő rendszer; stb.

Ilyen alrendszerekből épülnek fel napjaink nagy szolgáltató rendszerei is. Gondoljunk például a banki szolgáltató rendszerre! Ennek vannak olyan alrendszerei, amelyek az ügyfél rendelését szolgálják ki. Ilyen a telefonon történő rendelés lebonyolítása, a mágneskártyával történő vásárlás, az automatából történő pénzfelvétel kártyával, a bankban a személyes ügyintézés kiszolgálása és végül az ügyfél számlájával kapcsolatos konkrét műveletek (lekötés, befizetés, kifizetés stb.) végrehajtása. Ilyen összetett feladatok megoldására programrendszereket hozunk létre, amelyek alrendszerekből állnak. A probléma megoldásának alrendszerekből álló szerkezetét a probléma elemzése során határozzuk meg, amelynek modelljét szokás rendszerarchitektúrának nevezni (9.6. ábra).



9.6. ábra. A rendszerarchitektúra meghatározása

A következőkben a rendszerarchitektúrával kapcsolatos alapfogalmakat definiáljuk objektumelvű programozás esetén.

9.2.1. Alrendszer

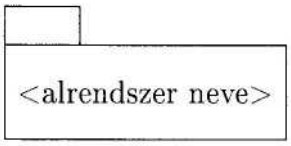
Az alrendszer (subsystem) logikai (szolgáltatási) szempontok alapján egységként kezelt szoftverkomponensek rendszere. Ilyen például egy

csomag (package) vagy a fájlrendszer egy operációs rendszerben.

Az alrendszer informális definíciója:

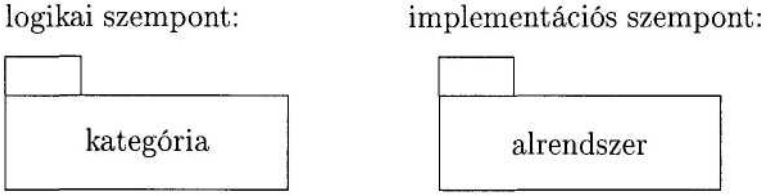
1. Az alrendszer önálló névvel ellátott programegység.
2. Az alrendszer logikai szempontok alapján (fizikailag létező) szoft verkomponensek egységként kezelt rendszere.
3. Az alrendszert modulok, illetve más alrendszerek alkotják.
4. Az alrendszer önállóan oldja meg feladatát.
5. Az alrendszer modulokban megvalósított osztályának objektu mai a feladat megoldása során más alrendszerek szolgáltatásait az alrendszerek között definiált szabványos kapcsolat útján ve hetik igénybe.

Az alrendszer jelölése a 9.7. ábrán látható.

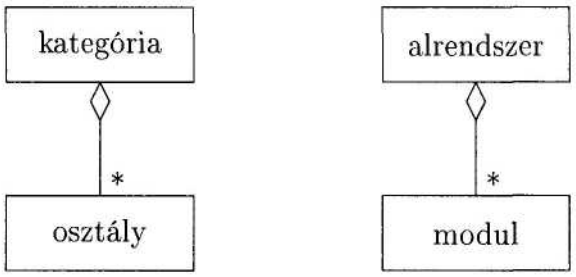


9.7. ábra. Az alrendszer jelölése

Az alrendszer logikai szempontból kategória. Alrendszerről az implementáció szempontjából szoktunk beszélni (9.8. ábra). A kategória osztályokból áll, míg az alrendszer modulokból épül fel (9.9. ábra).

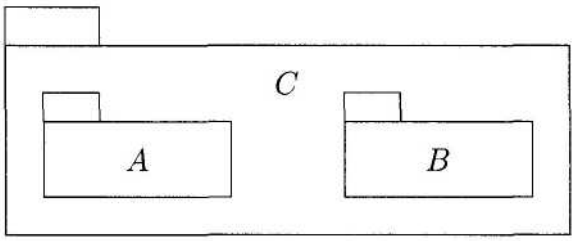


9.8. ábra. A kategória és az alrendszer kapcsolata



9.9. ábra. A kategória és az alrendszer kapcsolata felépülésük alapján

Az alrendszer relatív fogalom. Az alrendszer más alrendszerekből és osztályokból állhat. Például ha *C* egy olyan alrendszer, amely *A* és *B* alrendszereket tartalmazza, akkor ezt a 9.10. ábra fejezi ki.



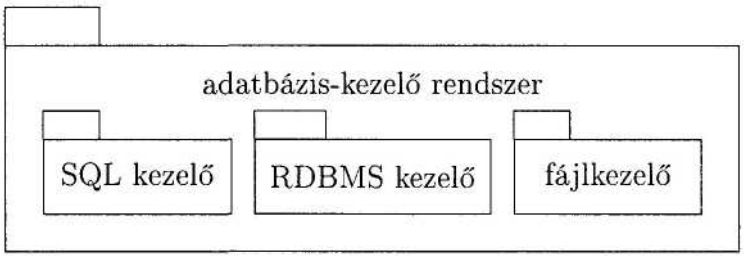
9.10. ábra. Alrendszereket tartalmazó alrendszer

Tipikus példa az adatbázis-kezelő csomag, amely rendszerint három alrendszert tartalmaz. Ezek lehetnek a következők:

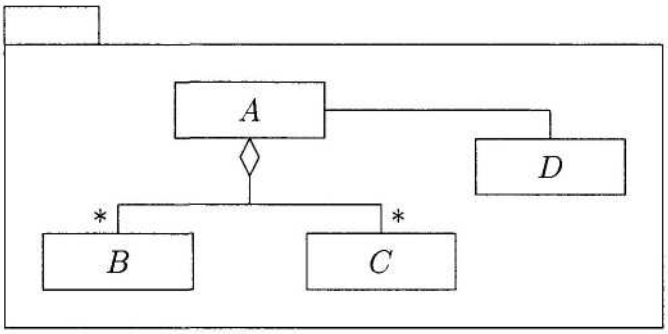
- SQL lekérdező nyelvet kezelő,
- relációs adatbázist (RDBMS) kezelő és
- fájlkezelő alrendszer (9.11. ábra).

Ha az alrendszer relációkkal összekapcsolt osztályok moduljait tartalmazza, akkor például a 9.12. ábra szemléltetheti az adott esetet.

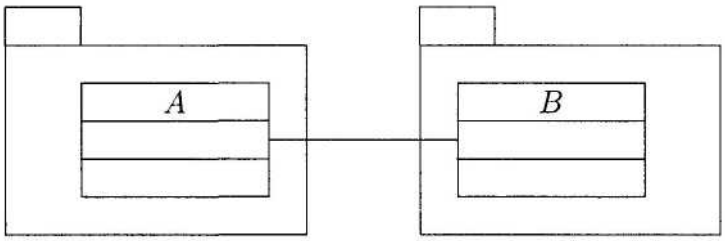
Különböző alrendszerek osztályai között asszociációs kapcsolatok állhatnak fenn (9.13. ábra).



9.11. ábra. Adatbázis-kezelő rendszer alrendszerei

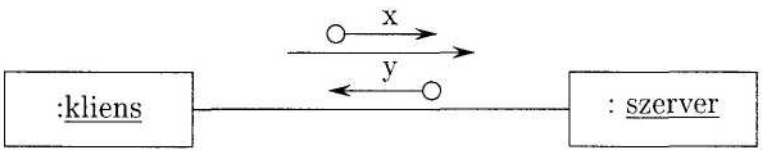


9.12. ábra. Alrendszer relációkkal összekapcsolt modulokból

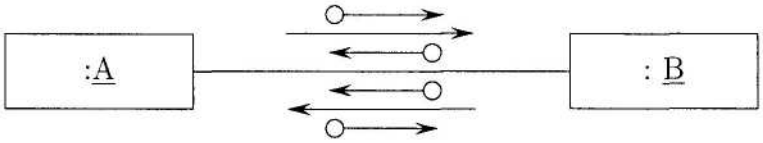


9.13. ábra. Különböző alrendszerek osztályai közötti asszociáció jelölése

Az alrendszerek objektumai egy ilyen kapcsolatban különböző módon viselkedhetnek. Lehetnek kliens, ágens vagy szerver tulajdonságú objektumok. A tulajdonság alapján lehet a kapcsolatokat osztályozni (9.14. és 9.15. ábra). A kapcsolatokat jellemezzük a következőkben, majd ezeket felhasználva megvizsgáljuk a különböző architektúrákat és azok tulajdonságait.



9.14. ábra. Kliens-szerver viszony



9.15. ábra. Egyenrangú, ágens-ágens viszony

1. **Ügyfél-szolgáltató** (kliens-szerver) kapcsolat.

   Az *ügyfél* szerepében fellépő objektum jellemzői a következők:

   —Ismeri a szolgáltatót, azaz tudja, hogy a számára szükséges szolgáltatásokat melyik objektumosztálytól igényelheti.

   —Ismeri a szolgáltatás igénybevételének formáját.

   —Tudja, hogy mit nyújt a szolgáltatás, azaz mi annak a jelentése.

   —A szolgáltatás teljesítésének részletei rejtve maradnak számára.

   A *szolgáltató* szerepében fellépő objektum jellemzői a következők:

   —Nem ismeri az ügyfelet, azt sem tudja, hogy az melyik alrendszerhez, annak melyik osztályához tartozik.

   —Nem tud semmit arról, hogy az ügyfél milyen célból veszi igénybe a szolgáltatásait.

   —Ismeri a szolgáltatás megvalósításának módját.

2. **Egyenrangú partner** (peer to peer) **kapcsolat.**

Mindkét alrendszer objektumainak jellemzői a következők:

— A számára szükséges szolgáltatásokról tudja, hogy azt melyik osztály objektuma nyújtja számára.
— Ismeri a szolgáltatás igénylésének formáját.
— Tudja, hogy mi a szolgáltatás jelentése.
— Nem ismeri a szolgáltatás végrehajtásának részleteit.

Az alrendszerekből álló rendszereket ezen kapcsolatok alapján a következőképpen szokás osztályozni:

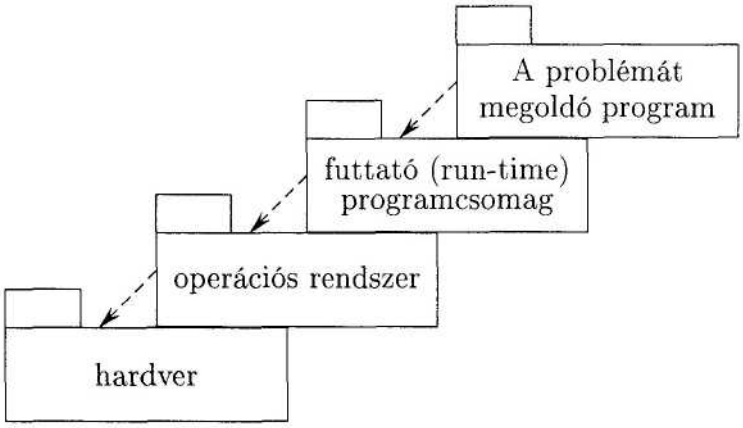
- rétegszerkezetű architektúra (layer),
- particionális szerkezetű architektúra,
- vegyes szerkezetű architektúra.

A rétegszerkezetű architektúra jellemzői:

1. Az alrendszerek egymásra épülő virtuális világot alkotnak.
2. Minden virtuális világ egy réteg.
3. A virtuális világ az alatta elhelyezkedő világokat ismeri, azoknak a szolgáltatásait igénybe tudja venni. Objektumai ekkor kliens ként viselkednek.
4. Egy virtuális világ a fölötte elhelyezkedő rétegekről nem ren delkezik ismeretekkel. Objektumai ezen rétegek felé szerverként viselkednek.

A tiszta rétegszerkezetű architektúra jellemző példája a hagyományos programfuttató rendszer (9.16. ábra).

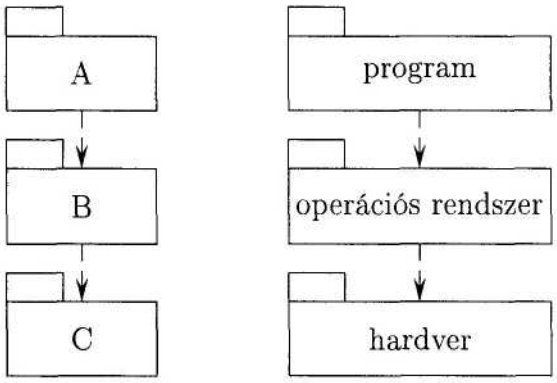
Rétegszerkezetek esetén meg szoktunk különböztetni nyílt rétegszerkezetet és zárt rétegszerkezetet.



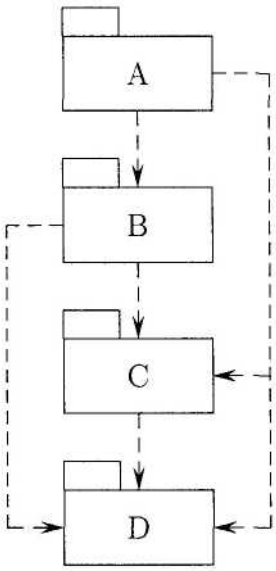
9.16. ábra. Programfuttató rendszer architektúrája

1. *Zárt rétegszerkezet* esetén a magasabb szinten elhelyezkedő ré tegek objektumai csak a közvetlenül alattuk lévő réteg objektu mainak szolgáltatásaihoz férnek hozzá (9.17. ábra).

2. *Nyílt rétegszerkezet* esetén egy adott szintű réteg objektumai több alacsonyabb szinten elhelyezkedő réteg objektumainak szol gáltatásait is ismerhetik és használhatják (9.18. ábra).



9.17. ábra. A zárt rétegszerkezet általános modellje látható az ábra bal oldalán, a jobb oldalon pedig egy konkrét példa



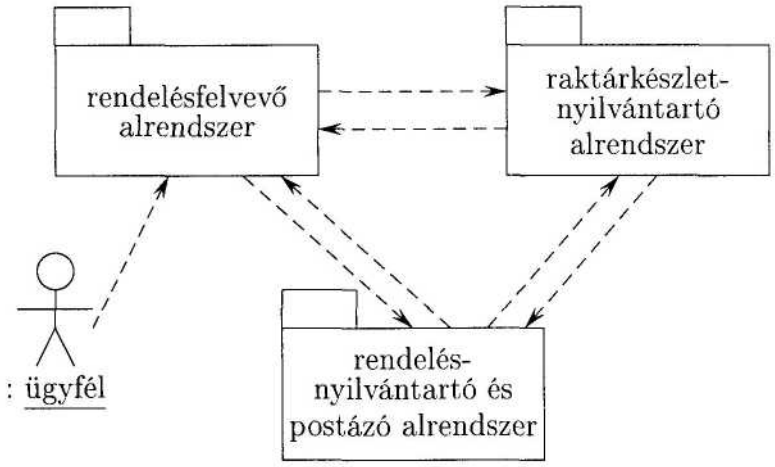
9.18. ábra. Nyílt rétegszerkezet

A particionális szerkezetű architektúra jellemzői:

1. A kapcsolat szempontjából egymással egyenrangú alrendszerek ből alkotott szerkezet.

2. Az egyenrangúság azt jelenti, hogy bármely alrendszer objek tuma egyenrangú partnerkapcsolatban állhat bármelyik másik alrendszer objektumával, azaz egymás szolgáltatásait kölcsönö sen igénybe vehetik.

A particionális szerkezetű architektúra egy példája a csomagküldő szolgáltatás rendszere (9.19. ábra). A három alrendszer egymás szolgáltatásait kölcsönösen használja. A rendelésfelvevő rendszer például ellenőrzi, hogy van-e a raktáron a rendelt mennyiség, majd a rendelést továbbítja a rendelés-nyilvántartónak. A raktárkészlet-nyilvántartó értesíti a rendelésfelvevőt a rendelhető áruk listájának változásáról. A



9.19. ábra. A csomagküldő szolgáltatás rendszere

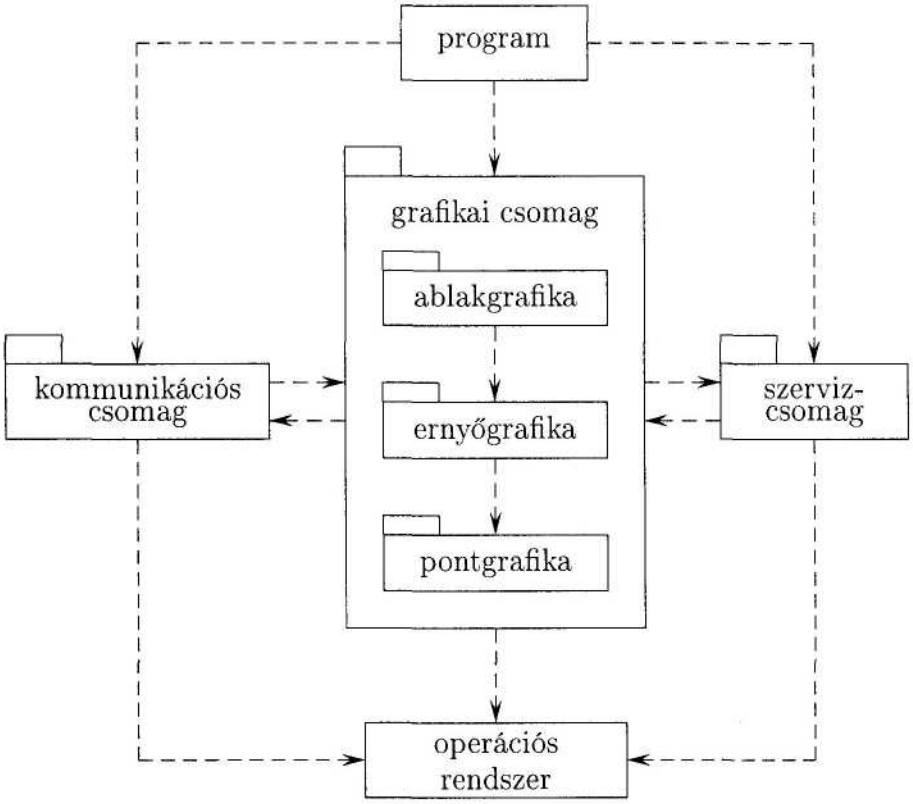
rendelés-nyilvántartó módosítja a raktárkészletet, és értesíti a rendelésfelvevőt a rendelés kielégítéséről.

A vegyes szerkezetű architektúra jellemzői:

1. A vegyes szerkezetű architektúra rétegszerkezetből és particio nális szerkezetből áll.
2. A vertikális szerkezet rétegszerkezet.
3. A horizontális szerkezet particionális szerkezet.

A vegyes szerkezetű architektúra egy tipikus példája az ablaktechnikára épülő programfuttató rendszer (9.20. ábra). A rendszer rétegei:

1. a program,
2. egy particionális szerkezetű réteg,
3. az operációs rendszer.



9.20. ábra. Az ablaktechnikára épülő programfuttató rendszer

A particionális szerkezet részei:

a. a kommunikációs csomag,

b. a grafikai csomag,

c. a szervizcsomag.

Ezen belül a grafikai csomag rétegszerkezetű, amelyben

- ablakgrafika,
- ernyőgrafika,

- pontgrafika

szerepel.

10. Az UML további diagramjai

Az UML diagramjai közül kettőről még nem beszéltünk: a környezeti és a használati esetek diagramjáról. Ezekről lesz szó ebben a fejezetben.

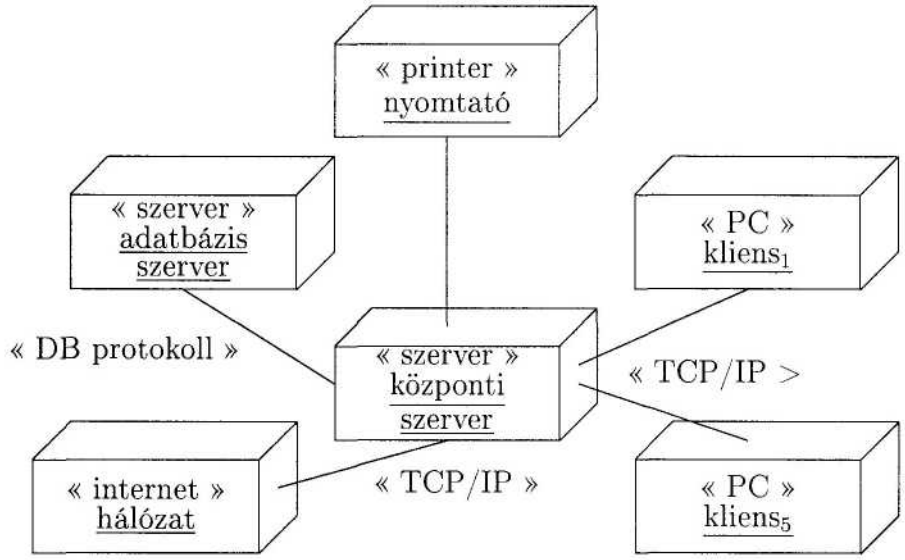
10.1. Környezeti diagram

A környezeti diagram a rendszer topológiáját írja le. Erre egy konkrét példát mutatunk (10.1. ábra). A kockák ábrázolják rendszer egységeit, az összeköttetések pedig az egységek közötti kapcsolatot és azok jellegét adják meg.

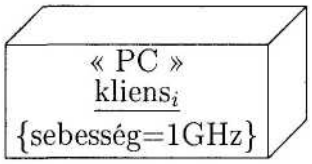
Az ábrát pontosabban is meg lehetne adni. Például az egységek jellegét is meg lehet határozni. Ha a nyomtatóról, illetve az egyik kliensről többet tudunk, akkor fel lehet tüntetni a megfelelő dobozon belül a következőket:

- színes nyomtató : HP Super Jet,
- $kliens_i$: Pentium III PC stb.

Lehetőség van az egyes egységekre megszorításokat tenni, és ezzel tovább pontosítani a leírást. Ekkor a megszorítást kapcsos zárójelek közé kell írni, mint például az a 10.2. ábrán látható.



10.1. ábra. Egy általános példa a környezeti diagramra, amely a rendszer topológiáját mutatja egy konkrét esetben



10.2. ábra. Egy egységre tett megszorítás jelölése

10.2. Használati esetek diagramja

A használati esetek diagramja a felhasználók szempontjából kívánja szemléltetni azt, hogy a rendszer miként működik, függetlenül attól, hogy a szolgáltatásait hogyan valósítja meg. Tehát azt szemlélteti, hogy mit tud a rendszer, függetlenül attól, hogy azt miként hajtja végre. A rendszer által nyújtott szolgáltatásokat együtt mutatja be azokkal, akik ezekkel a szolgáltatásokkal kapcsolatba kerülnek.

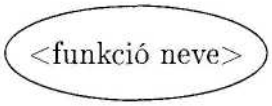
Ez a diagram eszköze annak, hogy a felhasználók és a rendszer fejlesztői a rendszerrel szemben támasztott követelményeket azonos

módon értelmezzék.

A diagram részei:

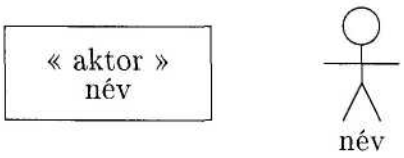
- használati esetek,
- felhasználók,
- felhasználási relációk.

A használati esetek a rendszer funkcióinak összefoglalásai, szolgáltatási egységek. Ez az egység az akcióknak egy olyan sorozata, amelyekkel a rendszer a felhasználók egy csoportjával működik együtt. Például egy információs rendszerben ilyen lehet a számlamozgásokról történő szolgáltatás az ügyfelek számára. A használati eset jelölése a diagramban a 10.3. ábrán látható. A használati eseteket egy téglalapba foglaljuk, ez a téglalap jelzi a rendszer határát.



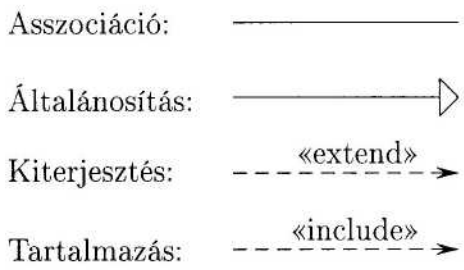
10.3. ábra. Használati eset jelölése

A felhasználók az adott rendszeren kívüli egységek, más programrendszerek, alrendszerek, osztályok, illetve személyek lehetnek. Ezek aktor szerepet töltenek be. Ezek jelölése a diagramban a 10.4. ábrán szerepel.



10.4. ábra. Felhasználók jelölése

A felhasználási relációk kapcsolják össze a használati eseteket a felhasználókkal. A relációk egymással is kapcsolatban állhatnak, amit a diagramban fel lehet tüntetni. A lehetséges relációk a következők.



10.5. ábra. A felhasználási relációk jelölése

Asszociáció: egy felhasználó és egy használati eset közötti kapcsolatot (kommunikációt) jelez.

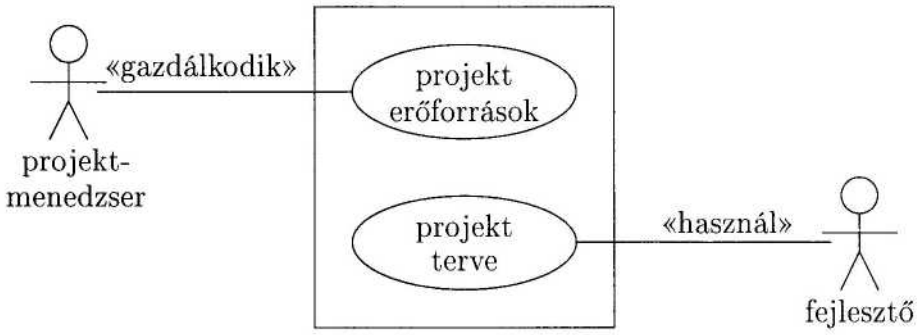
Általánosítás: az egyik használati eset a másik általánosabb formája.

Kiterjesztés: az egyik használati eset a másikat terjeszti ki. Ennek során viselkedéseket illeszt be megadott beszúrási pontoknál.

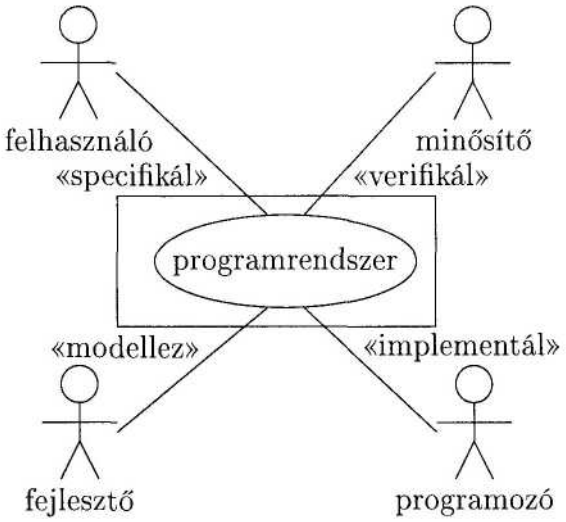
Tartalmazás: az egyik használati eset tartalmazza a másik viselkedését.

Jelölésük a 10.5. ábrán látható.

A felhasználási relációkat a felhasználás módjának jelölése egészítheti ki, alapszavak felhasználásával. A jelöléseket a 10.6. ábrán látható példán foglaljuk össze.



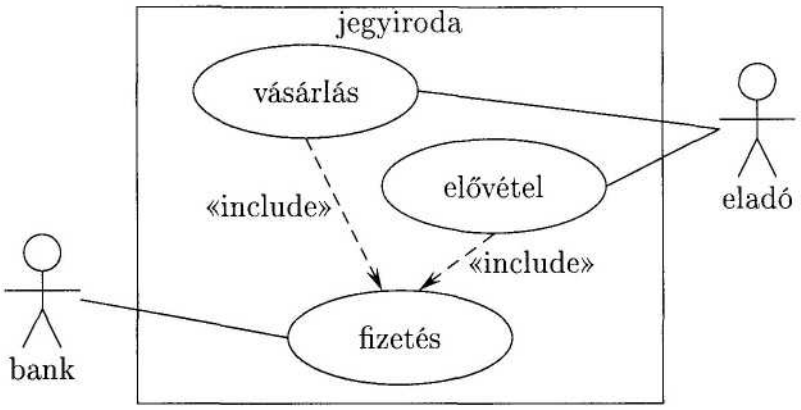
10.6. ábra. Egyszerű példa használati esetek diagramjára



10.7. ábra. A programrendszer és a projekt tagjainak kapcsolata

A használati esetek diagramja az áttekinthetőség céljából rendszerint szintekre tagolt. Például a programrendszer - mint használati eset - és a projekt tagjainak kapcsolatát a fejlesztés során szemlélteti a legfelső szinten a 10.7. ábra diagramja.

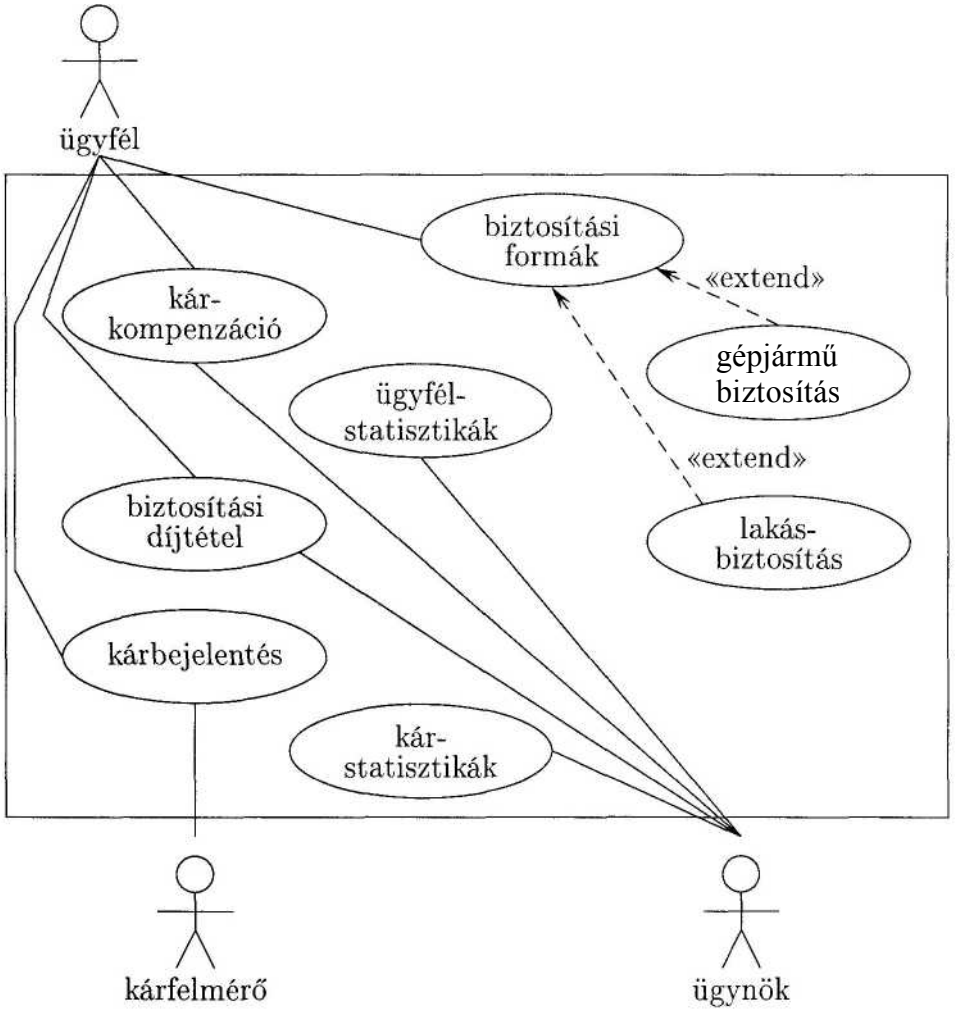
Bankkártyás jegyvásárlás esetén (10.8. ábra) jegyet vehetünk az



10.8. ábra. Jegyirodában történő bankkártyás vásárlás

eladónál az aktuális előadásra vagy elővételben. Mindkét eset tartalmazza a fizetést, amit a bankon keresztül lehet lebonyolítani.

Végezetül megadjuk egy biztosítási rendszer használati esetek diagramját (10.9. ábra). Az aktorok: az ügyfél, az ügynök és a kárfelmérő. Ezek kapcsolata a rendszer szolgáltatásaival az ábráról leolvasható.

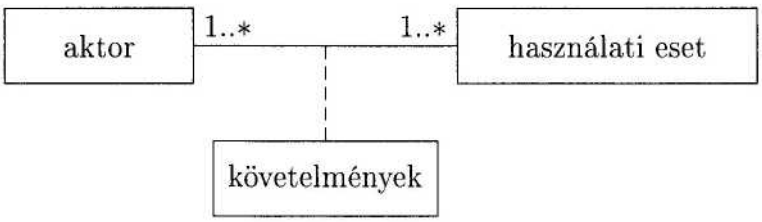


10.9. ábra. Biztosítási rendszer kapcsolatai

11. Modellalkotás a programfejlesztésben

Végezetül foglaljuk össze, milyen szerepet játszik az UML alapú modellalkotás egy programrendszer létrehozásában, melyek az egyes szakaszok! Ahol csak lehet, az UML jelöléseit használjuk a leírásban.

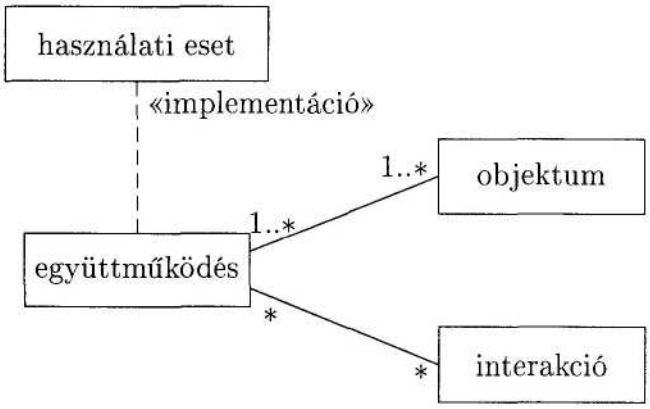
Az első szakasz a követelmények kifejtése (11.1. ábra). Ennek eredménye a *használati esetek diagramja,* amely a rendszer funkcióit, globális szolgáltatásait és azoknak a felhasználóival (aktorokkal) együtt történő szemléltetését nyújtja. A diagramban az egyes szolgáltatásokkal (aktor - használati eset összekapcsolásokkal) szemben támasztott követelményeket, elvárásokat írjuk le.



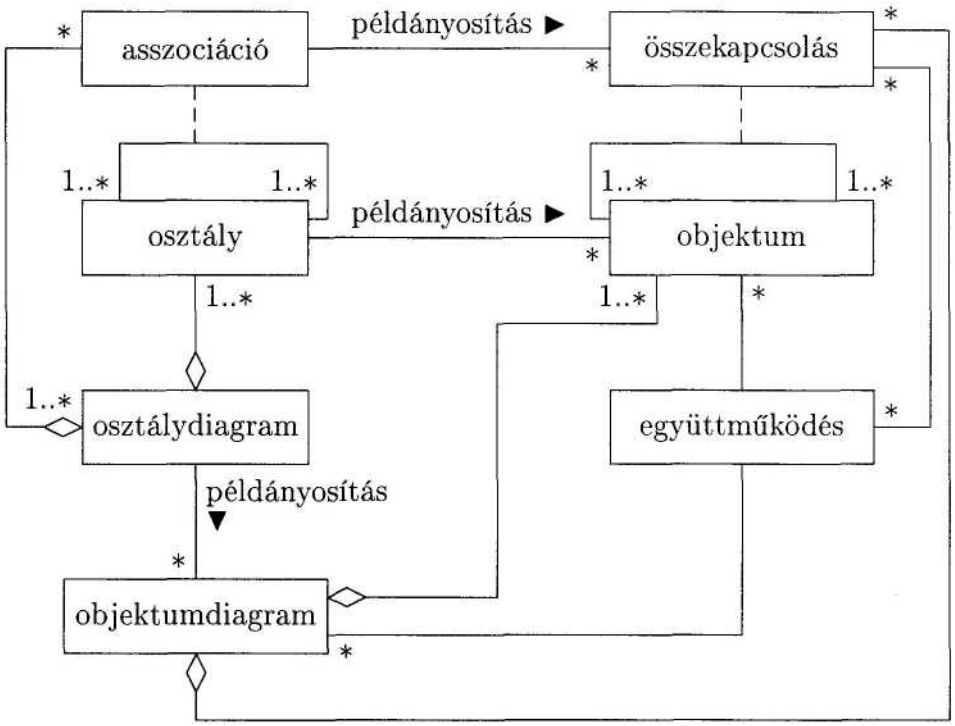
11.1. ábra. Követelmények kifejtése

Ezután következik a használati esetek objektum alapú formára alakítása (11.2. ábra). Ennek során feltárjuk a használati esetekben azonosítható objektumokat, az egymásra gyakorolt hatásokat, együttműködéseket a szolgáltatás megvalósítása során.

A következő lépés a megoldás szerkezetének meghatározása, amint a 11.3. ábrán látható. Ebben az *osztálydiagram* a probléma megol-



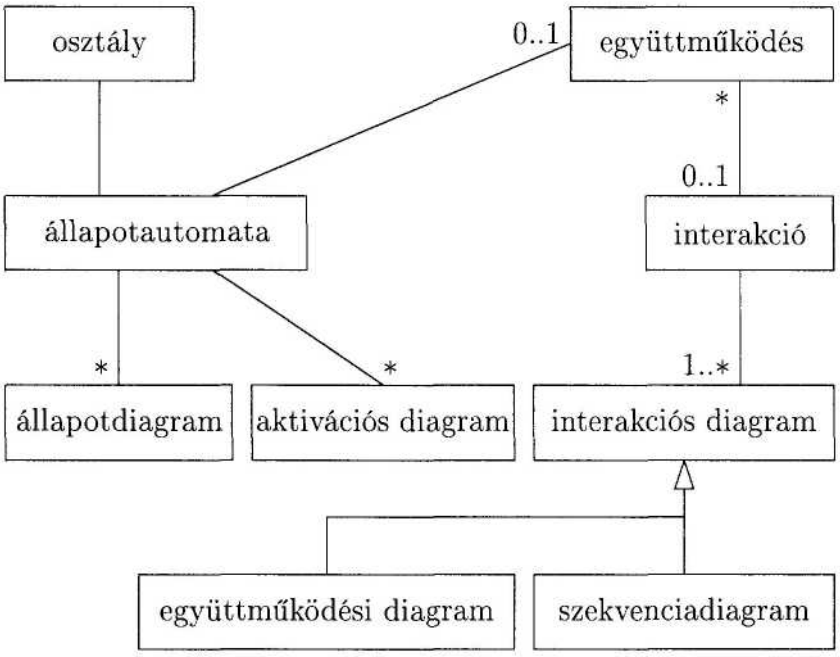
11.2. ábra. Használati esetek objektum alapúvá alakítása



11.3. ábra. Megoldás szerkezetének meghatározása

dásának statikus szerkezetét fejezi ki, osztályok és osztályok között fennálló relációk formájában. Az *objektumdiagram* pedig az osztálydiagram egy példányaként szemlélteti a probléma megoldásának statikus szerkezetét, objektumok és azok összekapcsolódásainak formájában. Az objektumok közötti együttműködés az objektumdiagramban szereplő összekapcsolásokon keresztül valósulhat meg.

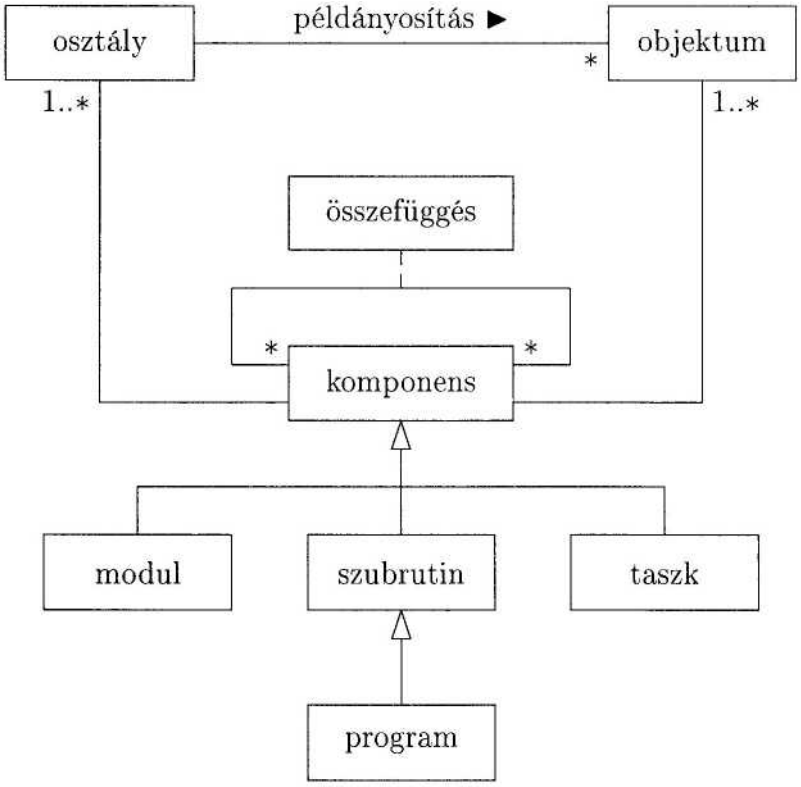
Ezután a megoldás viselkedésének (jelentésének) meghatározása következik (11.4. ábra). Ebben az *állapotdiagram* a probléma megoldásában részt vevő osztályok objektumainak a viselkedését mutatja, állapotautomata formájában. Az *aktivációs diagram* azt szemlélteti, hogy az objektumok a probléma megoldása során időrendi sorrendben hogyan, milyen tartalmú üzenetekkel aktiválják (késztetik cselekvésre) egymást. Az objektumok közötti együttműködést az interakciós diagramok írják le. A *szekvenciadiagram* a probléma megoldása során az



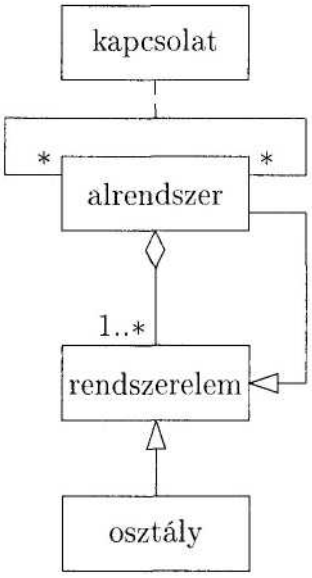
11.4. ábra. Megoldás viselkedésének meghatározása

objektumok között időrendi sorrendben lejátszódó üzenetváltásokat szemlélteti. Az *együttműködési diagram* az objektumok közötti üzenetváltásokat szemlélteti, az üzenetek tartalmára (az információáramlásra) és nem az időbeliségre helyezve a hangsúlyt.

Az utolsó lépés az implementáció. A *komponensdiagram* az implementációs környezetben mutatja, az egyes komponensek helyét és egymáshoz való viszonyát (11.5. ábra). Az *alrendszerdiagram* az egységek szervezési módját szemlélteti, az alrendszerek közötti előre meghatározott kapcsolatok leírásával együtt (11.6. ábra). A *konfigurációs diagram* azt mutatja, hogy a probléma megoldására szolgáló komponensek hol helyezkednek el a hardver környezetben.



11.5. ábra. Komponensdiagramok szerepe az implementációban



11.6. ábra. Alrendszerdiagramok szerepe az implementációban

11.1. Szemléltető példa

A következőkben egy egyszerű példán mutatjuk be az objektumelvű terv elkészítését és annak implementálását.

A buszközlekedés állomások között zajlik. Minden állomást jellemez a neve. Az állomások között buszjáratok közlekednek. Minden járatot egy szám azonosít, és jellemzője a maximálisan szállítható utasok száma. Egy járat legalább két állomáson megáll, az állomások sorrendje rögzített. A buszközlekedésben utasok vesznek részt. Minden utasnak van neve, és egy kiinduló állomásról szeretne egy másik állomásra eljutni egy járattal (átszállás nélkül).

Az utasok bizonyos időközönként megérkeznek a kiinduló állomásra, és ott várnak. Ha olyan buszra lehet felszállni az állomáson, amelyen még van hely, és meg fog állni a célállomáson, akkor az utas felszáll. A felszállás során előbb azok az utasok szállnak fel, akik előbb érkeztek az állomásra. Ha a busz megérkezik a célállomásra, akkor az utas leszáll, és befejezi a tevékenységét. Az utasok az első megfelelő

buszra felszállnak, nem foglalkoznak egyéb kritériummal (menetidő, ... stb.). A buszjáratok a menetrend szerint indulnak, azaz érkeznek meg az első állomásukra. Induláskor minden busz üres. Egy állomásra érve előbb leszállnak a megfelelő utasok a buszról, majd felszállnak az utasok a buszra. Ha a felszállás befejeződik, akkor a járat a következő állomásra megy, ahová a menetrendben megadott időpontban érkezik. Ha ez volt az utolsó állomás, akkor az utasok leszállása után a járat befejezi a tevékenységét.

- Készítsük el a buszközlekedés osztálydiagramját! Az osztályok nál adjuk meg az attribútumokat is!
- Készítsünk az osztálydiagram alapján objektumdiagramot a kö vetkező esetre! Három állomás között járnak a járatok. Az állo mások nevei: a, b, c. Két buszjárat jár: az egyes számú, amelyik 2 utast szállíthat; és a kettes számú, amelyik 3 utast szállíthat. A menetrend:

idő	állomás	járat
8:00	a	1
8:15	c	1
8:20	b	2
8:25	b	1
8:30	a	2
8:35	c	2
8:40	a	1

Az objektumdiagram az állomások és a buszok közötti kapcsolatot adja meg a menetrend alapján!

Készítsük el a buszközlekedés állapotdiagramját!

Készítsünk szekvenciadiagramot, amely szemlélteti az előző menetrend és a következő táblázat alapján az üzeneteket 7:55 és 8:16 között!

idő	utas	indul	cél
7:55	A	a	c
7:56	B	a	b
7:57	C	a	c
8:00	D	c	a

- Készítsünk együttműködési diagramot, amely szemlélteti azokat az üzeneteket az átadott adatokkal együtt, amelyek egy járat állomásra érkezése után kerülnek elküldésre!

- Készítsünk programot, amely alkalmas a buszközlekedés szimu lálására! A program jelenítse meg a járatok, az állomások ak tuális állapotait, és az utolsó időegység alatt bekövetkezett ese ményeket! A forgalmi adatokat egy szövegfájlból olvassuk be, amely tartalmazza a szimuláció kezdetének időpontját, az állo mások adatait, a járatok adatait, majd az utasok adatait. Az utasok a kiinduló állomásra érkezés sorrendjében szerepeljenek a fájlban!

Megoldás:

A leírás alapján három osztályt azonosíthatunk:

- állomás,

- járat,

- utas.

Az állomás leírásban szereplő attribútuma a név, a járaté pedig a szám és a maximálisan szállítható utasok száma, és végül az utasoké a név. Az osztályok közötti relációk:

- Egy járat részét képezik az állomások mint megállók. Ez egy aggregációs kapcsolat, amelyben legalább két állomás vesz részt meghatározott sorrendben. Egy állomás ugyanakkor tetszőleges számú járathoz tartozhat. A navigálhatóság szempontjából azt

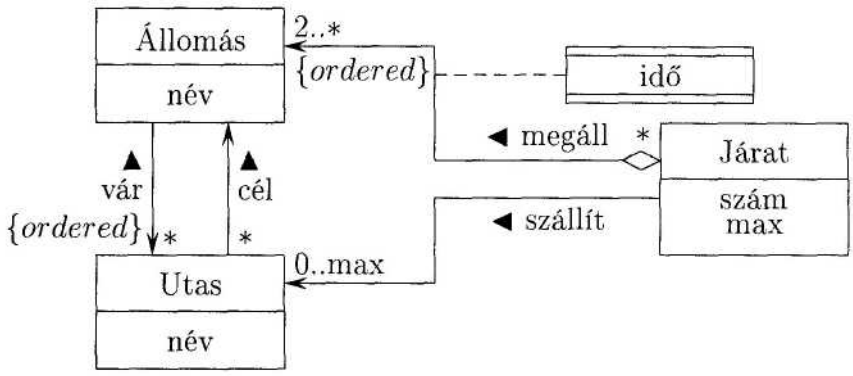
mondhatjuk, hogy csak a járatnak kell ismernie a megállóit. Ugyanakkor ennek a relációnak jellemzője minden egyes esetben az állomásra érkezés időpontja. Ezt egy társult osztályban fejezhetjük ki.

- Egy járat utasokat szállít, ez asszociáció. Egy járat legfeljebb adott számú utast szállíthat (egyszerre nem lehet rajta akár mennyi), egy utas pontosan egy járatot használ. Itt a járat felől akarjuk elérni az utasokat.

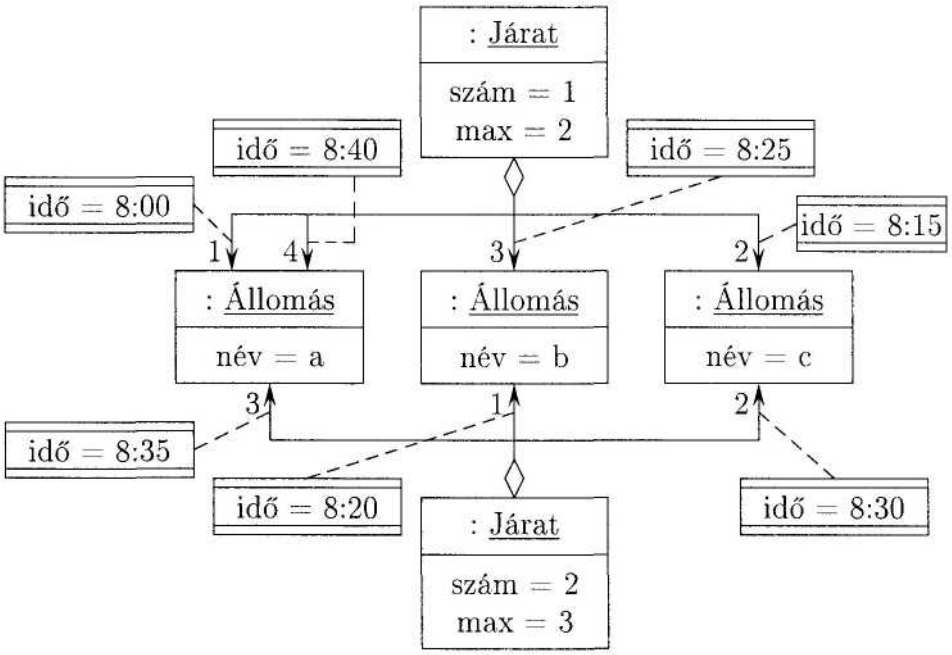
- Az állomás és az utas osztályok között két reláció azonosítható. Egyrészt minden utas egy kiinduló állomáson várakozik, más részt minden utasnak egy állomás a célja. Mindkét esetben egy utashoz pontosan egy állomás tartozik, és egy állomáshoz tet szőleges számú utas. A várakozás esetében az állomásról kell tudnunk elérni az ott várakozó utasokat, méghozzá az érkezés sorrendjében. A cél esetében a navigálhatóságnak az utas felől kell az állomás felé mutatnia.

Ennek alapján a 11.7. ábra osztálydiagramjához jutunk.

A kapcsolatok megvalósításához nyelvi szinten újabb attribútumokkal kell kiegészíteni az osztályokat. A járat esetében két, mutatókat tartalmazó sorozatra van szükség, az állomás esetében egy mu-



11.7. ábra. A buszközlekedés osztálydiagramja

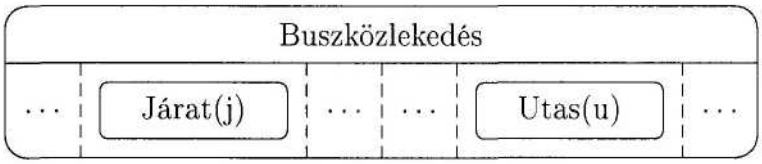


11.8. ábra. A buszközlekedés egy objektumdiagramja

tatókat tartalmazó sorozatra, az utas esetében pedig egy állomás nevére. Az attribútumok nevei egyezzenek meg a relációk neveivel! A továbbiakban így hivatkozunk ezekre.

Az objektumdiagramban 2 járat és 3 állomás objektum szerepel. A járatokat a *megáll* aggregációs kapcsolat köti össze az állomásokkal, a menetrendben szereplő időpontok adják meg a sorrendet (11.8. ábra).

A buszközlekedés állapotait a járatok és az utasok állapotai határozzák meg (11.9. ábra).

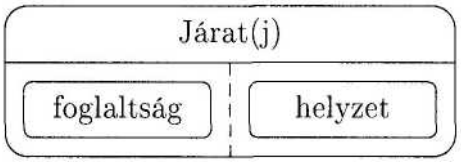


11.9. ábra. A buszközlekedés állapotai

Egy járat állapotai két részből állnak (aggregáció):

- a járat telítettsége, *foglaltság,*
- a járat helyzete, *helyzet* (11.10. ábra).

(A *j* paramétert a továbbiakban nem tüntetjük fel, ahol nem szükséges. Az állapotokat mindig a *Járat(j)* állapotainak kell tekinteni.)



11.10. ábra. Egy járat állapotai

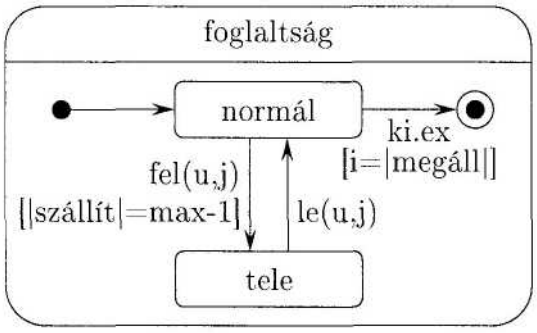
A foglaltságon belül két állapotot különböztethetünk meg. A *normál* állapotban van még hely a buszon, azaz a szállított utasok száma nem éri el a maximumot; a *tele* állapotban már nincs szabad hely, azaz a szállított utasok száma megegyezik a maximummal. Ezt a két állapothoz tartozó állapotinvariáns fejezi ki:

$$I(normál) \;:\; |szállít| < max$$
$$I(tele) \;:\; |szállít| = max$$

Az állapotváltozásokat az utasok felszállása és leszállása, röviden fel és le, okozza. A kezdeti állapot a normál, és ha az utolsó megállóban az utasok leszálltak (1. helyzet), akkor a járat befejezi a működését (11.11. ábra).

A járat helyzetét három állapot jellemzi:

- Az éppen aktuális állomáson kiszállnak az utasok, *ki.* Ez egy paraméteres állapot, ahol a paraméter az aktuális állomás. Vezessünk be egy új attribútumot, *i,* amely az aktuális állomás indexét adja meg a megállók sorozatában! Ekkor az állapot paramétere: *megáll[i}.* Az *i* kezdetben 1.



11.11. ábra. A járat foglaltsága

- Ezután az aktuális állomáson beszállnak az utasok, *be(megáll[i]).*
- A beszállás befejezése után a járat a következő állomásra *megy.* Ennek az entry fázisában *i* értéke eggyel nő.

A járat kezdeti állapota a *ki(megáll[i]),* hiszen kezdetben $i = 1$. A járat akkor fejezi be a működését, amikor az utolsó megállóban kiszálltak az utasok, azaz *ki(megáll[i]) .ex[i = |megáll|].*

A *be* és a *ki* paraméteres állapotokat is egy-egy invariáns jellemzi. Az állapotok egészen addig fennmaradnak, amíg az utasok *le,* illetve *fel* akciói meg nem szüntetik az invariánst. Az állapotokból történő kilépésnél *(.ex)* az áttekinthetőség kedvéért nem tüntetjük fel a paramétert az állapotdiagramban.

Az invariánsok formális felírásához vezessünk be néhány fogalmat és jelölést, amelyekkel az állítások tömörebben megfogalmazhatóak:

- az u utas indexe az a állomás várakozási sorában legyen $\sigma(u, a)$;
- a járat hátralévő megállóinak halmaza legyen μ;
- az a állomáson várakozó első i utas céljainak a halmaza legyen $\omega(a, i)$!

A bevezetett fogalmak definíciói formálisan:

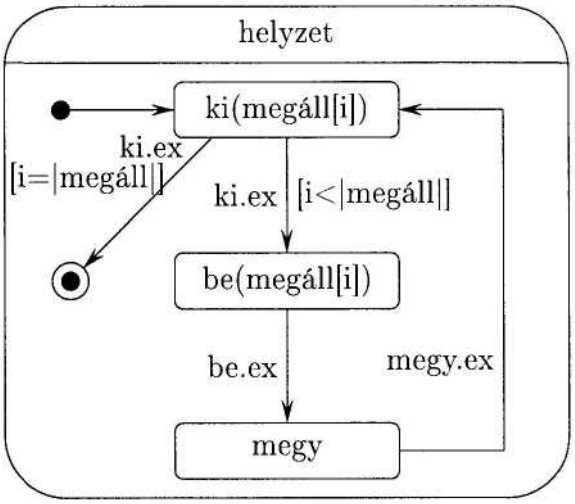
$$\sigma(u, a) \quad : \quad a.vár[\sigma(u, a)] = u \ ;$$

$$\begin{aligned} \mu &= \{megáll[k] \mid k \in [i+1..|megáll|]\} \ ; \\ \omega(a,i) &= \{a.vár[k].cél \mid k \in [1..i]\} \ . \end{aligned}$$

Ezek felhasználásával felírhatóak az állapotinvariánsok. A *ki* állapot addig áll fenn, amíg van a járaton olyan utas, aki az aktuális állomásra jött. A *be* esetében azt kell figyelni, hogy van még hely, és vár olyan utas az állomáson, aki oda akar menni, ahová a járat. A *megy* állapotból a megfelelő idő letelte után lépünk ki, azaz ebben maradunk, amíg az aktuális idő, *t,* kisebb az érkezési időnél.

$$\begin{aligned} I(ki(a)) &: \exists k \in [1..|szállít|] : szállít[k].cél = a \\ I(be(a)) &: in\ normál \wedge \omega(a, |a.vár|) \cap \mu \neq \emptyset \\ I(megy) &: t < megáll[i].idő \end{aligned}$$

A járat helyzetét megadó állapotdiagram látható a 11.12. ábrán.



11.12. ábra. Egy járat helyzete

Egy utas esetében két állapotot különböztethetünk meg:

- a kezdeti a állomáson várakozik, *vár (a),*
- a *j* járaton utazik, *utazik(j).*

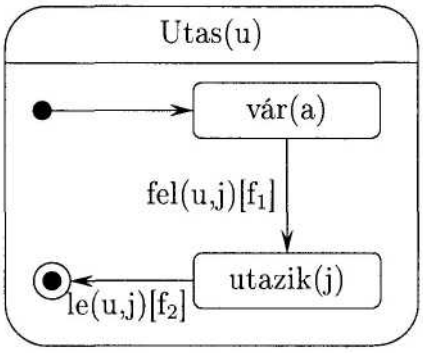
A kezdeti állapot a várakozás, amelyből a *fel* akció hatására megy át az utazásba. Ennek feltétele, hogy az adott járatra a megfelelő állomáson fel lehet szállni *(be),* a járat megáll az utas célállomásán, és az utas előtt nincs olyan utas a várakozók között, aki oda menne, ahová a járat. Ezt fejezi ki az f_1 feltétel. Az utazást a *le* akció hatására fejezi be az utas, amelynek feltétele, hogy a járat a célállomáson legyen a leszállási állapotban *(ki).* Ezt fejezi ki az f_2 feltétel. Ezután az utas befejezi a tevékénykedését.

Az állapotátmenetek feltételei:

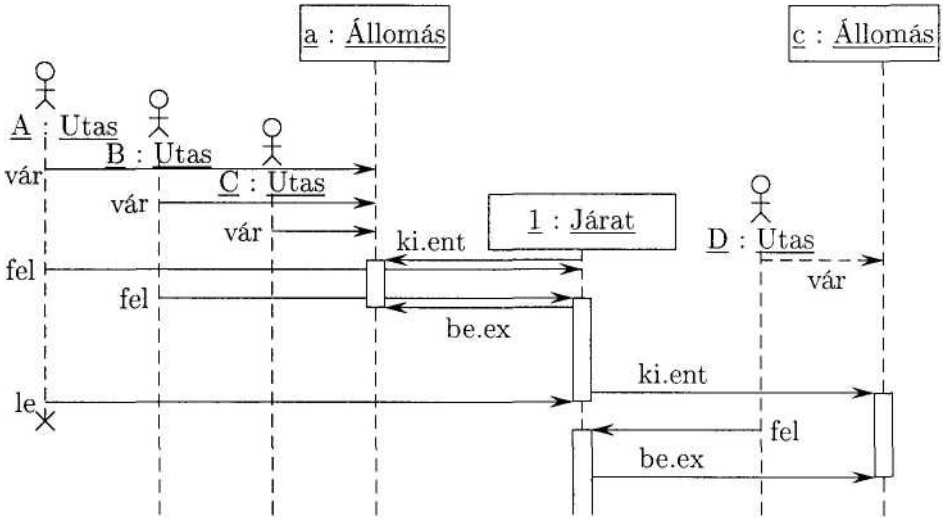
$$f_1 \ : \ járat(j)\ in\ be(a) \wedge u.cél \in járat(j).\mu \wedge \omega(a, \sigma(u,a) - 1) \cap járat(j).\mu = \emptyset$$

$$f_2 \ : \ járat(j)\ in\ ki(u.cél)$$

Az utas állapotdiagramja a 11.13. ábrán látható. A szekvenciadiagramban a megadott táblázatban szereplő 4 utas, az egyes számú járat és az a és c állomás objektumok szerepelnek. Az



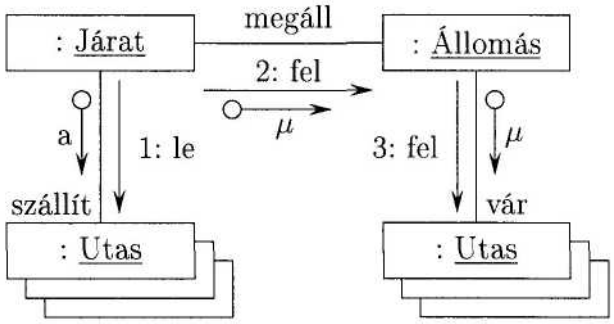
11.13. ábra. Egy utas állapotai



11.14. ábra. A buszközlekedés egy szekvenciadiagramja

utasok aktor szerepet töltenek be. Az állomás aktív szerepe legyen az, amikor ott busz tartózkodik, a járat aktív szerepe pedig az, amikor tele van (11.14. ábra)!

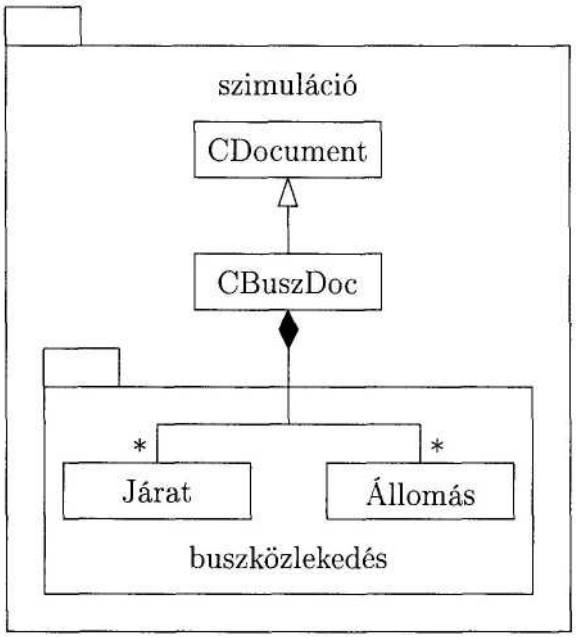
Az együttműködési diagramban fel kell tüntetni a járat objektumot, az általa szállított utasoknak megfelelő objektumokat (ezek száma tetszőleges), az aktuális állomást és az ott várakozó utasokat. A járat és az első utascsoport közötti összekapcsolás a szállít relá-



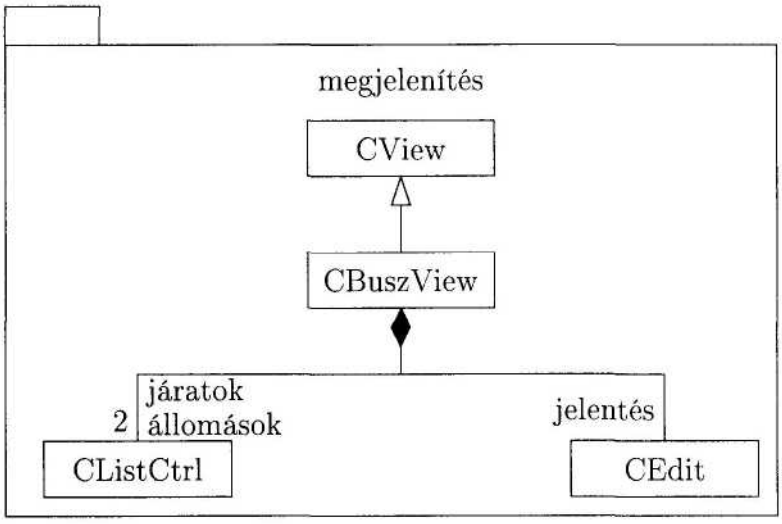
11.15. ábra. Egy járat állomásra érkezését leíró együttműködési diagram

ció, amelynek mentén küldi a járat az első üzenetet az utasoknak. Az üzenet a *le,* az átadott adat pedig az állomás neve. A járatot az állomással a megáll reláció kapcsolja össze, és a járat az állomásnak küldi a második üzenetet, *fel,* amelyhez a hátralévő állomások halmaza társul adatként. Ezt az üzenetet az állomás továbbítja az adattal együtt a várakozó utasoknak, ez lesz a harmadik üzenet. A *le,* illetve a *fel* üzenetekre az utasok megfelelően, a saját akcióikkal reagálnak (11.15. ábra).

A szimulációs programot Windows operációs rendszerre, Visual C++ környezetben készítjük el. A program az MFC által biztosított alrendszerekre bomlik. Az egyik a *megjelenítés* kezelése, a másik az *adatok* kezelése, azaz a tulajdonképpeni szimuláció. Az adatok kezelése a *CDocument* osztályból származtatott *CBuszDoc* osztályban valósul meg. Ez tartalmazza a buszközlekedés alrendszer osztályait, a már megismert kapcsolatokkal, amelyeket itt nem ismétlünk meg (11.16. ábra).



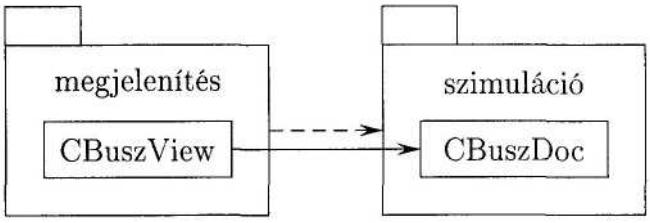
11.16. ábra. A szimulációs alrendszer



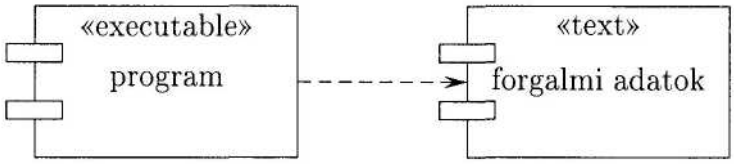
11.17. ábra. A megjelenítő alrendszer

A megjelenítés a *CView* osztályból származtatott *CBuszView* osztály feladata. Ebben szerepel az utolsó időegység alatt lejátszódott események leírása, *jelentés;* a járatok listája, *járatok;* és az állomások listája, *állomások.* A jelentés egy szerkesztőablakban, *CEdit,* tartalmazza a megfelelő szöveget, a két lista egy-egy report stílusú *CListCtrl* (11.17. ábra).

A két alrendszer alkotja a programot. A megjelenítő alrendszer használja a szimulációs alrendszer adatait a CBuszView és a CBuszDoc osztályok közötti kapcsolaton keresztül (11.18. ábra).



11.18. ábra. A buszközlekedést szimuláló program alrendszerei



11.19. ábra. A buszközlekedést szimuláló rendszer komponensei

A program alkotja a rendszer egyik komponensét. A másik komponens a forgalom adatait (állomások, járatok, utasok) tartalmazó fájl (11.19. ábra).

Az eddigieket felhasználva készítsük el a programkódot! Itt most csak a szimulációval foglalkozunk, a megjelenítés a megfelelő MFC eszközök felhasználásával viszonylag egyszerűen kivitelezhető.

A programban mutatókat tartalmazó sorozatokat kell tudnunk kezelni az osztálydiagramban szereplő relációk ábrázolásához. Szükséges utasok és állomások mutatóit is kezelnünk. Ennek érdekében vezessünk be egy sablonosztályt, amelynek paramétere a mutató alaptípusa, és rendelkezik a megfelelő műveletekkel (elem hozzávétele, lekérdezés, eltávolítás, ...)! A sorozat nem szabadítja fel a mutatókat, ez a felhasználó felelőssége. Ez lehetőséget ad arra, hogy a mutatókat egyik sorozatból a másikba helyezzük át, illetve arra is, hogy egy mutató több sorozatban is szerepelhessen. Ez utóbbi tulajdonság különösen fontos lesz a járatok és az állomások közötti *megáll* reláció megvalósításánál, hiszen több járat is megáll ugyanazon az állomáson. A leírtaknak megfelelő sablonosztály legyen a *Sequence* osztály, amely a következő felülettel rendelkezik.

```
template <class Item> class Sequence
{
public:
    Sequence();
    virtual ~Sequence();
    void Add(Item *i);
    Item *operator[](unsigned int i);
    Item *operator[](unsigned int i) const;
```

```
    void RemoveAt(unsigned  int  i);
    void RemoveAll();
    int   Size()  const                { return(size) ;  }
protected:
    Item              **content;
    unsigned int      size;
    unsigned int      act  max;
    void      Resize() ;

    Sequence(const  Sequence &src)                    {};
    Sequence &operator=(const  Sequence &src)       {};
};
```

Az utasoknak megfelelő *Utas* osztály implementációja adódik az osztálydiagramból.

```
class Utas {
public:
    Utas(const CString &n, const CString &c);
    virtual ~Utas()        {};
    CString Nev() const    { return(nev); }
    CString Cel() const    { return(cel) ;
} protected:
    CString   nev;
    CString   cel;


Utas::Utas(const  CString &n,  const CString &c)
{
    nev = n;       cel = c;
```

Az *Állomás* osztály esetében a *vár* relációt a bevezetett *Sequence* osztály segítségével valósítjuk meg. Az ehhez kapcsolódó műveletek: a

várakozó utasok számának lekérdezése, az *i.* várakozó utas lekérdezése, az *i.* utas felszállása egy adott buszra, új utas felvétele a várakozási sorba.

```
class Allomas

public:
    Allomas(const CString &n)          { nev = n; }
    virtual ~Allomas();
    CString Nev() const                { return(nev); }
    int Utasszam() const               { return(var.Size() ); }
    Utas *Var(int i)                   { return(var[i]); }
    Utas *Fel(int i);
    void UjUtas(Utas *u)               { var.Add(u); }
protected:
    CString          nev;
    Sequence<Utas>   var;


Allomas::~Allomas ()

    for ( int i = 0; i < var.Size(); i++ ) delete var[i] ;


Utas *Allomas::Fel(int i)
{
    Utas   *u = var[i];
    var.RemoveAt(i);
    return(u);
```

A *megáll* reláció megvalósításához be kell vezetnünk egy új osztályt, legyen ez *Megálló,* amely megadja az állomás mutatóját és az érkezés idejét. A járatok csak ezen keresztül érhetik el az állomásokat, ezért az állomások megfelelő műveleteit itt biztosítanunk kell.

```
class Megallo
{
public:
    Megallo(Allomas *a, int mikor);
    virtual ~Megallo()     {};
    CString Nev() const   { return(all->Nev()); }
    int Utasszam() const   { return(all->Utasszam()); }
    Utas *Var(int i)      { return(all->Var(i)); }
    Utas *Fel(int i)      { return(all->Fel(i)); }
    int Ido() const       { return(ido); }
protected:
    Allomas *all;
    int    ido;


Megallo::Megallo(Allomas *a, int mikor)
{
    ido = mikor;       all = a;
```

A *Járat* osztály reprezentációjában szereplő attribútumok és a vonatkozó műveletek adódnak az osztálydiagramból. Ezt ki kell egészíteni egy olyan művelettel, amely egy idő múlását jelzi. Legyen ez a művelet az *Órajel!* Ennek paramétere az aktuális időpillanat, és egy jelentés, amelyhez a járattal kapcsolatos tevékenységeket kell hozzávennünk. A járat *ki,* illetve *be* állapotának (leszállás és felszállás) fe-

meg, ezért ezek belső műveletek. Az állapot paramétere az aktuális állomás, ami a sorozat első eleme. (A már érintett állomásokat elhagyjuk a sorozatból.) Egy másik segédművelet, $Tartalmaz$, feleljen meg $cél \in \mu$ feltételnek, azaz annak, hogy az adott állomás szerepel-e leljen meg a Le, illetve *Fel* művelet! Miután ezek állapotnak felelnek

a megmaradt úticélok között! A műveletek implementációja adódik az állapotdiagramból, illetve egy egyszerű lineáris keresés a tartalmazás esetében.

```
class Jarat
{
public:
    Jarat(int s, int m)         { szam = s; max = m; }
    virtual ~Jarat();
    int Utasszam() const        { return(szallit.Size()); }
    int Megalloszam() const     { return(megall.Size()); }
    void UjMegallo(Megallo *m) { megall.Add(m); }
    void Orajel(int ido, CString &jelent);
    int Szam() const            { return(szam); }
    Megallo *Megall(int i)      { return(megall[i]); }
    Utas *Szallit(int i)        { return(szallit[i]); }
    int Max() const             { return(max); }
protected:
    int         szam;
    int         max;
    Sequence<Megallo>   megall;
    Sequence<Utas>      szallit;

    void Le(CString &jelent);
    void Fel(CString &jelent);
    bool Tartalmaz(const CString &s) const;
};

Jarat::~Jarat()
{
    int    i;
    for ( i = 0; i < szallit.Size(); i++ )
        delete szallit[i];
    szallit.RemoveAll();
    for ( i = 0; i < megall.Size(); i++ )
        delete megall[i];
    megall.RemoveAll();
}
```

```
        if ( Tartalmaz(megall[0]->Var(i)->Cel()) )
        {
            u = megall[0]->Fel(i);
            szallit.Add(u);
            // jelentés kiegészítése
        }
        else
            i++;
    }
}

bool Jarat::Tartalmaz(const CString &s) const
{
    for ( int i = 1; i < megall.Size(); i++ )
        if ( megall[i]->Nev() == s )    return(true);
    return(false);
}
```

A *CBuszDoc* osztály tartalmazza a szimuláció adatait és a megjelenítéshez szükséges adatokat. Ez az osztály felelős az idő kezeléséért is. Ennek megfelelően a következőkkel kell kiegészítenünk a fejlesztőkörnyezet által létrehozott osztályt. Le kell tudnunk kérdezni az aktuális időponthoz tartozó eseményeket (jelentés), a járatokat és az állomásokat. Reprezentálnunk kell az aktuális időt, az állomásokat, a járatokat, az adatokat tartalmazó fájlt és a jelentést. Az adatok esetében tudnunk kell, hogy mi a soron következő utas állomásra érkezési ideje, így akkor olvasunk be adatot, ha szükséges. Belső műveletek az idő kezelése, ezen belül az érkező utasok beolvasása, illetve az elején az állomások és a járatok beolvasása.

```
class CBuszDoc : public CDocument
{
...
public:
    CString Jelentes() const { return(jelentes); }
```

```
    int    Jaratszam() const { return(jaratok.Size()); }
    Jarat *Jaratok(int i)    { return(jaratok[i]); }
    int Allomasszam() const  { return(allomasok.Size()); }
    Allomas *Allomasok(int i) const
                             { return(allomasok[i]); }
    //{{AFX_VIRTUAL(CBuszDoc)
    public:
    virtual void Serialize(CArchive& ar);
    virtual void DeleteContents();
    //}}AFX_VIRTUAL
protected:
    int                 ido;
    Sequence<Allomas>   allomasok;
    Sequence<Jarat>     jaratok;
    ifstream            adat_file;
    bool                nyitott_adat_file;
    int                 adat_ido;
    CString             jelentes;
protected:
    void Orajel();
    void UtasOlvasas();
    void AllomasOlvasas();
    void JaratOlvasas();
    int  AllomasIndex(const CString &n) const;
...
};
```

Most csak a szimuláció szempontjából érdekes Orajel műveletet adjuk meg, a többi művelet implementációja értelemszerű, és nem is tartoznak a modellezett részhez. Egy időegység leteltekor be kell olvasni az esetlegesen érkező utasokat, a járatoknak el kell küldeni a megfelelő üzenetet, és a megszűnt járatokat törölni kell.

```
void CBuszDoc::Orajel()
{
    int    i;
    ido++;
    if ( ido % 100 == 60 )    ido += 40;    // 760 -> 800
    CString    info;
    CString linefeed;
    linefeed.FormatMessage("\n");
    info.Format("Idő: %d", ido);
    jelentes = info + linefeed + linefeed;
    UtasOlvasas();
    for ( i = 0; i < jaratok.Size(); i++ )
        jaratok[i]->Orajel(ido, jelentes);
    i = 0;
    while (i < jaratok.Size() )
        if ( jaratok[i]->Megalloszam() == 0 )
        {
            delete jaratok[i];
            jaratok.RemoveAt(i);
        }
        else    i++;
}
```

12. Tervminták, keretek

Objektumelvű rendszer tervezése nehéz és bonyolult feladat, különösen ha újrafelhasználható rendszert, illetve tervet szeretnénk létrehozni. A tervnek egyszerre kell megfelelnie a konkrét probléma sajátosságainak, ugyanakkor elég általánosnak is kell lennie ahhoz, hogy más, később felmerülő feladatok megoldása során is alkalmazható legyen, elkerülve vagy legalábbis minimalizálva az esetleges újratervezést.

Jó tervek létrehozásához nagy gyakorlatra van szükség. Miért van az, hogy kezdő szakemberek rendszerint nem tudnak olyan jó, újrafelhasználható terveket készíteni, mint a nagy gyakorlattal rendelkezők? Ennek egyik legfontosabb oka, hogy a tapasztalt tervezők rendszerint bevált tervek alapján hozzák létre egy új rendszer tervét. Egy-egy jó tervet használnak újra meg újra, ezek ismerete és használata teszi őket jó szakemberekké. Ezeket a terveket, tervrészeket nevezzük *tervmintáknak* (design patterns). Hasonló módon használhatóak a tervezésben, mint ahogy a programozás során a kód egyes részeit kód-újrafelhasználás segítségével állítjuk elő.

12.1. Tervminta

Általában egy tervminta a következő négy alapvető elemből épül fel:

Név - Ezzel hivatkozunk a tervezési feladatra és annak megoldására. Ez rendszerint egy-két szó, amely utal a funkcióra. Lehetővé teszi, hogy a tervezést magasabb absztrakciós szinten végezzük.

Feladat - Itt le kell írni, hogy milyen esetekben alkalmazható a minta. A leírás tartalmazza a problémát, annak környezetét és az esetleges peremfeltételeket.

Megoldás - A minta alkotóelemeinek, azok kapcsolatainak, együttműködésüknek a leírása. Ez egy absztrakt leírás, amely az általánosság érdekében nem tartalmazza az implementációt.

Következmények - A minta eredményeinek és a meghozott kompromisszumoknak (futási idő, tárkapacitás) a leírása.

12.1.1. Tervminták megadása

A következőkben összefoglaljuk, miként adhatunk meg egy-egy tervmintát. Az UML diagram ugyan fontos eleme ennek, de korántsem elégséges önmagában. A mintákat a megoldandó feladat alapján osztályokba soroljuk (pl.: létrehozási, szerkezeti, viselkedési), és ezt szintén fel kell tüntetnünk a meghatározás során.

1. *A minta neve és osztálya:* egy jó névnek utalnia kell a felhasz nálás területére, a megoldott feladatra.
2. *Cél:* rövid leírása annak, hogy mi a minta által megoldott feladat vagy feladatosztály.
3. *Más nevek:* további, mások által használt nevei a mintának, ha van ilyen.
4. *Motiváció:* egy esettanulmány, amely bemutatja a tervezési fel adatot, és hogy miként oldja meg azt a minta a benne szereplő osztályok és objektumok segítségével.
5. *Felhasználhatóság:* annak leírása, hogy milyen esetekben lehet a mintát alkalmazni.
6. *Szerkezet:* itt kell megadni a megfelelő UML diagramot vagy diagramokat.

7. *Elemek:* a mintában előforduló osztályok, objektumok és szere peik felsorolása.

8. *Együttműködés:* annak bemutatása, hogy a minta elemei miként működnek együtt a szerepük megvalósítása érdekében.

9. *Következmények:* itt kell választ adni azokra a kérdésekre, hogy miként éri el a minta a célját; milyen kompromisszumok árán; a rendszer szerkezetének milyen összetevőit lehet függetlenül vál toztatni.

10. *Implementáció:* itt kell megadni az implementációval kapcsolatos észrevételeket, megjegyzéseket, ajánlásokat, megszorításokat.

11. *Példa kód:* kódrészletek, amelyek bemutatják, miként lehet a minta egyes elemeit egy adott nyelven megvalósítani.

12. *Ismert használat:* példák a minta előfordulásaira működő rend szerekben.

13. *Rokon minták:* a hasonló minták nevei, a fontosabb különbségek felsorolása, illetve annak a megadása, hogy mely más mintákkal célszerű együtt használni.

A tervminta megadásának elemei közül néhány (10-13) értelemszerűen elmaradhat.

Az, hogy mely terveket tekintünk tervmintáknak, relatív fogalom. Az egyetlen követelmény, hogy az többször felhasználható legyen. Ezen belül egy adott fejlesztői közösség dönthet. Ugyanakkor léteznek jól bevált tervminták. Ezek egy részéről ad áttekintést a [10] könyv. A könyvben szereplő számos minta közül mutatunk be egyet itt példaként. Ezt is rövidítve, vázlatosan, bizonyos részek elhagyásával tesszük meg.

Név, osztály: Figyelő (observer); viselkedési (behavioral).

Cél: Olyan egy-több függőség megadása objektumok között, amelyben ha egy objektum állapotot vált, akkor az összes tőle függő objektumot értesíteni és módosítani kell.

Más nevek: dependents, publish-subscribe.

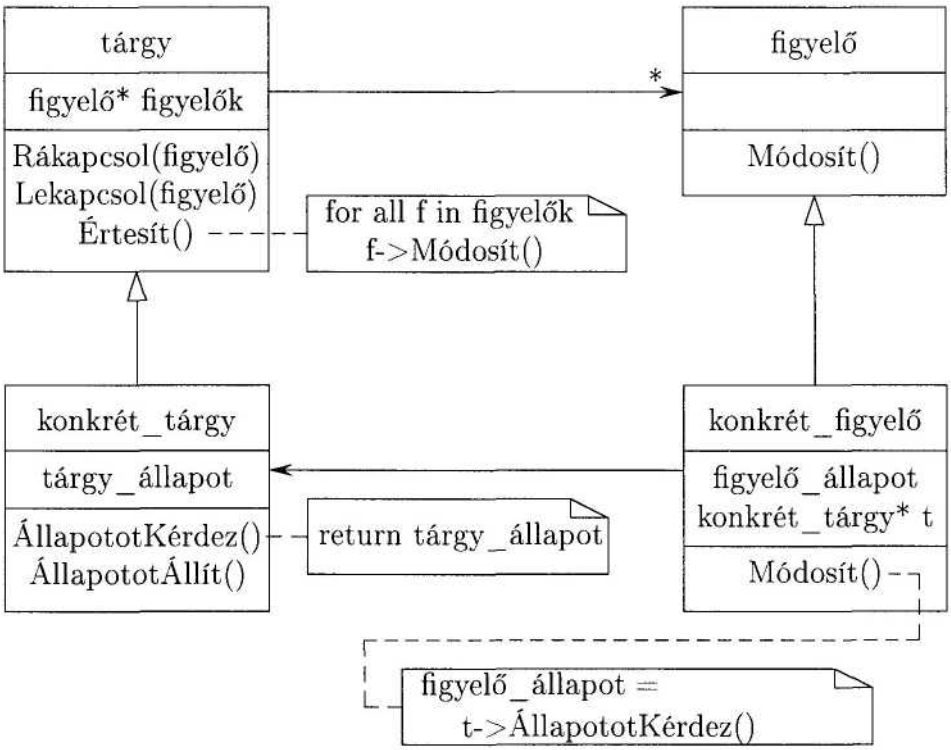
Motiváció: Egy gyakori mellékhatás, ami egy rendszer együttműködő osztályokra bontása során felmerül, a kapcsolatban álló objektumok közötti konzisztencia fenntartásának szükségessége.

Erre egy példa, amikor egy statisztika százalékos adatait akarjuk megjeleníteni táblázat, oszlopdiagram vagy kördiagram formájában. Egyidejűleg több formában is láthatók az adatok. Ekkor a statisztika a megjelenítés tárgya, a diagramoknak pedig mindig annak aktuális állapotát kell mutatniuk, anélkül hogy a diagramok egymásról tudnának. Ha bármelyiken változtatás történik, akkor az megjelenik a statisztikában, és az értesíti az összes diagramot. Ebben a példában a statisztika a *tárgy* (subject), a diagramok pedig a *figyelők* (observers).

Felhasználhatóság: A figyelő tervminta használható az alábbi esetekben:

- Ha egy absztrakcióban két olyan tényező szerepel, amelyek kö zül az egyik függ a másiktól. Ha ezeket külön objektumoknak tekintjük, akkor lehetséges egymástól független változtatásuk és újrafelhasználásuk.
- Amikor egy objektum állapotának megváltoztatása más ob jektumok változtatását teszi szükségessé, és nem ismert ezen objektumok száma.
- Amikor egy objektumnak üzenetet kell küldenie más objektu moknak, amelyekről nem tehetünk fel semmit; azaz nem akar juk szorosan összekapcsolni ezeket az objektumokat.

Szerkezet:

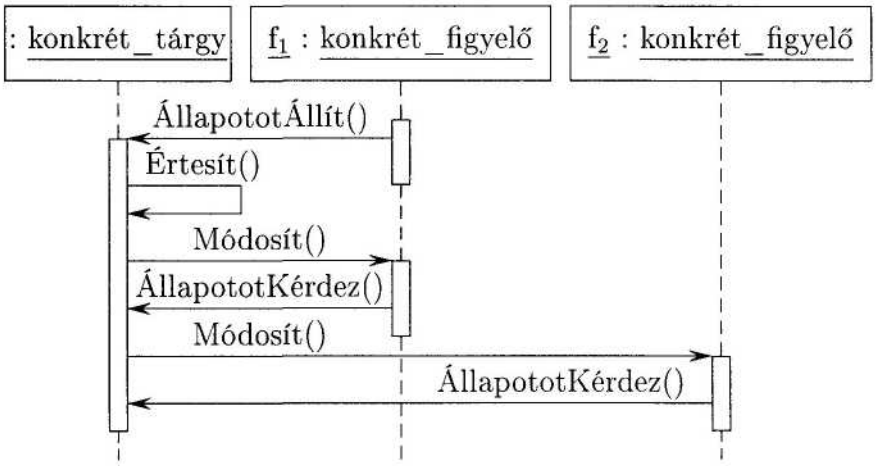


Elemek:

- tárgy: Ismeri a figyelőit, amelyek száma tetszőleges. Lehetősé get biztosít figyelő objektumok hozzávételére és eltávolítására.
- figyelő: Meghatározza a módosítási felületét azoknak az ob jektumoknak, amelyeket értesíteni kell.
- konkrét tárgy: Tartalmazza a konkrét figyelők számára ér dekes állapotot. Állapotváltozás esetén értesíti a figyelőit.

- konkrét_figyelő: Hivatkozik a konkrét_tárgy objektumra. Tartalmazza az állapotot, amelynek konzisztensnek kell lennie a tárgyéval. Implementálja a figyelő módosítási felületét, ezzel biztosítva a konzisztenciát.

Együttműködés: A konkrét_tárgy értesíti a figyelőit olyan változás esetén, amely inkonzisztenciát okozhatna az aktuális állapot és a figyelői állapota között. Az értesítés után a konkrét figyelő lekérdezheti a tárgy állapotát. Ennek segítségével állítja helyre a konzisztenciát. Ezt szemlélteti a következő szekvenciadiagram, két figyelő esetén.



A tervminta leírásának befejezésétől eltekintünk. Az osztálydiagramban feltüntettük a kapcsolatokat megvalósító pointereket is. Ezek közül a *figyelők* egy listára mutat, amelynek elemeit sorban (például egy iterátor segítségével) el tudjuk érni.

Egy lehetséges alkalmazás az idő megjelenítése a számítógépen. Ekkor az óra a megfelelő műveletekkel kiegészítve a tárgy objektumból örököltethető, figyelő pedig lehet ennek digitális vagy analóg megjelenítése.

12.2. Keret

A tervmintáknál speciálisabb, de nagyobb újrafelhasználható tervezési egységek a *keretek* (frameworks). A keretek egyre elterjedtebbé és fontosabbá válnak. Ezeket használják fel legtöbbször az objektumelvű rendszerekben.

A keret együttműködő osztályok halmaza, amelyek egy újrafelhasználható tervet alkotnak meghatározott osztályba tartozó szoftverek számára. A keret megszabja a rendszer szerkezetét. Definiálja a teljes szerkezetet, azt osztályokra és objektumokra particionálja, megadja azok feladatait, az együttműködésük módját és a vezérlés folyamatát. A keret magában foglalja a felhasználási terület közös tervezési döntéseit. A keret előre definiálja ezeket a tervezési paramétereket, így használójának csak az adott rendszer specialitásaival kell foglalkoznia. Ennek megfelelően keretek használata esetén terv-újrafelhasználásról beszélünk, nem pedig kód-újrafelhasználásról, noha rendszerint egy keret tartalmaz konkrét osztályokat, amelyek azonnal használhatók.

Keretek használata esetén a program (szerkezetileg) fő részét használjuk fel, és azokat a kódrészleteket kell megírni, amelyeket az hív. Azaz adott nevű és paraméterezésű műveleteket kell előállítani, így csökkentve a meghozandó tervezési döntések számát. Ennek eredményeképpen nemcsak a rendszer készül el gyorsabban, hanem hasonló feladatot megoldó programok szerkezete is hasonló lesz. Ezért egyszerűbb lesz azok karbantartása, és egységesebbeknek tűnnek majd felhasználóik számára is.

A tervminták is és a keretek is tervezési általánosítások, de három lényeges eltérés van köztük:

1. *A tervminták absztraktabbak, mint a keretek.* A keretekhez tartozik programkód, a tervminták esetén legfeljebb példákat lehet megadni.

2. *A tervminták kisebb szerkezeti egységek, mint a keretek.* Egy keret rendszerint több tervmintát tartalmaz, de ez fordítva nem áll fenn.

3. *A tervminták kevésbé specializáltak, mint a keretek.* A keretekhez mindig tartozik egy speciális felhasználási terület. Ezzel szemben egy tervminta gyakorlatilag tetszőleges területen használható. A tervminta nem írja elő az egész rendszer szerkezetét, szemben a kerettel.

A tervminták és a keretek is *komponensek* segítségével épülnek fel, és ezek biztosítják az egyszerű újrafelhasználhatóságot. Komponensen értjük egy rendszer (fizikai) kicserélhető részét, amely megfelel egy adott felületnek (interfész), és egyben ilyen hozzáférési felületet biztosít is. Komponenseket könnyebben fel lehet használni, ha kiegészítjük azokat olyan leírással, amely válaszol a következő kérdésekre: milyen szolgáltatásokat biztosít a komponens; milyen elvárásoknak kell megfelelnie; melyek a függőségi viszonyai; ...

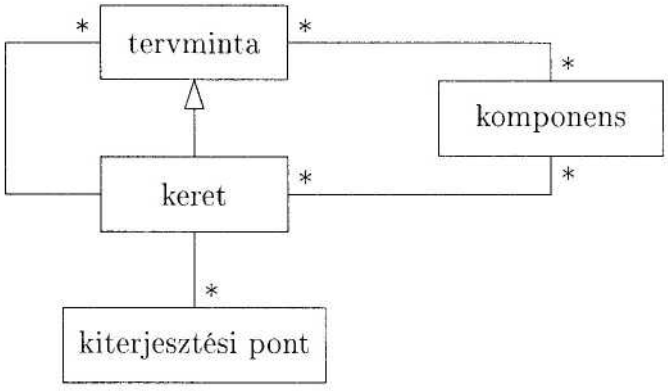
A keretek ezenkívül még rendelkeznek *kiterjesztési pontokkal* is. Ezek adják meg a keret azon részeit, amelyeket módosítani kell egy adott felhasználás során, azaz leírják, hogy miként lehet egy keretet kiterjeszteni, illetve testre szabni. Ennek megfelelően, a tervmintákkal ellentétben, egy keretet a felhasználás során ki is lehet terjeszteni, illetve az alapértelmezett viselkedését módosítani lehet.

A tervminták, keretek, komponensek és kiterjesztési pontok kapcsolatát szemlélteti a 12.1. ábra. Látható, hogy a keretekben specializált tervminták szerepelnek. Egy keret akár több tervmintát is használhat. Komponens előfordulhat tervmintában és keretben egyaránt. Egy komponens több tervmintának, illetve keretnek is alkotóeleme lehet, és egy tervminta, illetve keret felhasználhat több komponenst is. Egy kerethez több kiterjesztési pont tartozhat.

Egy keret létrehozása során a következő tevékenységeket kell elvégezni:

1. A keret felhasználási területének vagy területeinek azonosítása.

2. A keret elnevezése. A névnek utalnia kell a felhasználás módjára és lehetőleg a keret szerkezetére is.



12.1. ábra. Tervminták, keretek, komponensek és kiterjesztési pontok kapcsolata

3. A keret által támogatott alapvető használati esetek meghatározása.
4. A kerettel kapcsolatot tartó aktorok azonosítása.
5. Olyan ismert tervminták vagy egyéb működő megoldások azonosítása, amelyek segíthetik a keret fejlesztését.
6. A keret alapvető felületeinek és komponenseinek megtervezése, és feladatok, illetve aktorok hozzárendelése a hozzáférési felület részeihez.
7. A keret felületeinek alapértelmezett implementációja.
8. A keret kiterjesztési pontjainak leírása és dokumentálása.
9. Tesztesetek és teszttervek létrehozása.

A tervminták és keretek jelentősen felgyorsíthatják a tervezés folyamatát, ugyanakkor használatuk kezdetben jelentős ráfordítást igényel, elsősorban a keretek esetében. Ennek oka a keret szerkezetének megismerésében rejlik. Ennek a kezdeti ráfordításnak az idejét lehet csökkenteni ismert elemek felhasználásával. Ezért fontos, hogy minél

több közismert tervmintát, illetve megoldási módot használjunk fel, és az egész keretet megfelelően dokumentáljuk. Ebben az esetben ugyanis a keret nagyobb egységekből épül fel, magasabb absztrakciós szintet használ, így a megfelelő ismeretekkel rendelkező szakember számára könnyebben áttekinthető.

13. A programtermék minőségi mutatói

A második fejezetben tárgyaltuk, hogy a szoftvertechnológia célkitűzése előírt minőségű programtermék, előre meghatározott határidőre, előre meghatározott költségen történő előállítása. Kérdés, hogy egy elkészült szoftver minőségét miként lehet vizsgálni, értékelni. A válaszhoz meg kell állapodni bizonyos mutatókban, amelyekkel a minőséget minél egzaktabban jellemezni lehet. A programtermék minőségi mutatói:

- programhelyesség,
- megbízhatóság, robusztusság,
- hatékonyság,
- pszichológiai bonyolultság,
- egyéb jellemzők, például kezelhetőség, újszerűség stb.

Ezen mutatók közül az első kettő a program *funkcionális minőségét* jellemzi. Először ezekkel foglalkozunk részletesebben. Ezután a pszichológiai bonyolultságot vizsgáljuk röviden. Végül megmutatjuk, hogyan használható a végrehajtási gráf a rendszer futási idejének becslésére. A többi mutatót nem tárgyaljuk, azok ugyanis nagyban függnek a konkrét alkalmazástól, így általános elemzés nehezen lenne adható.

13.1. Programhelyesség

Ez a mutató a specifikáció és a program közötti kapcsolatot minősíti. Tegyük fel, hogy adott egy feladat specifikációja a következő formában:

$X = \{x \mid p(x)\}$	bemenő adatok halmaza,
$Y = \{y \mid q(y)\}$	eredmény adatok halmaza,
$X \to Y$ reláció	$\{(x, y) \mid p(x) \wedge q(y) \wedge r(x, y)\}$,

ahol p, q, r állítások!

Tegyük fel továbbá, hogy adott egy S program, amelynek programfüggvénye f_S! Ekkor azt mondjuk, hogy az S program a p, q, r specifikáció szerint helyes, ha

- végrehajtása befejeződik, és
- $p(x) \wedge q(f_S(x)) \wedge r(x, f_S(x)) =$ `igaz`.

A specifikáció szerinti helyesség ellenőrzésére több módszer is ismert (Dijkstra, Floyd, Hoare), ezek tárgyalása azonban meghaladja e könyv lehetőségeit.

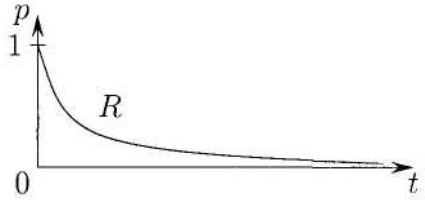
13.2. Megbízhatóság, robusztusság

Ez a fogalom a programmal szemben támasztott követelmények és a program viszonyának minőségét fejezi ki. Annak valószínűségével mérhető, hogy a program nem nyújt hibás szolgáltatást.

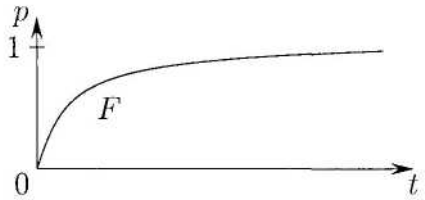
A helyesség és a megbízhatóság két külön fogalom. Egy program lehet helyes és mégsem megbízható, és fordítva. Ugyanis vegyünk egy helyes programot és egy olyan bemenő adatot, amely nem megengedett, azaz $x \notin X$! Ebben az esetben a helyesség nem mond semmit, ugyanakkor a megbízhatóság értelmében ekkor sem nyújthat helytelen szolgáltatást (nem abortálhat) a program. Fordítva: a programunk

megbízható, de nem helyes, ha olyan x G X bemenő adat esetén, amikor túlcsordulás lépne fel, akkor erre figyelmeztet, és nem hajtja végre a műveletet.

A megbízhatóság időben változó, amit jellemezhetünk a megbízhatósági függvénnyel (13.1. ábra), illetve a meghibásodási függvénnyel (13.2. ábra). A megbízhatósági függvény, *R,* annak a valószínűségét adja meg, hogy a program nem nyújt hibás szolgáltatást adott idő elteltéig; a meghibásodási függvény, *F,* pedig annak a valószínűsége, hogy hibás szolgáltatást nyújt.



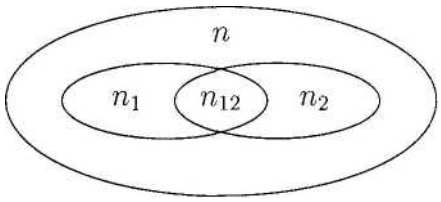
13.1. ábra. A megbízhatósági függvény, *R.* A függőleges tengely a valószínűség, a vízszintes az idő.



13.2. ábra. A meghibásodási függvény, *F.* A függőleges tengely a valószínűség, a vízszintes az idő.

A megbízhatóság mértékét meghatározhatjuk a programban lévő rejtett hibák számának becslésével. Erre egy lehetséges módszer a következő.

Tegyük fel, hogy adott két azonos felkészültségű csapat a programban lévő rejtett hibák keresésére! A programban összesen *n* rejtett hiba van, amelyből az egyik csapat n_1*-et,* a másik pedig n_2-t talált meg, és ezek között n_{12} közös (13.3. ábra).



13.3. ábra. Rejtett hibák elhelyezkedése

Ha feltesszük, hogy a két csapat azonos hatékonysággal dolgozik,

$$\frac{n_{12}}{n_1} = \frac{n_2}{n} \quad \text{és} \quad \frac{n_{12}}{n_2} = \frac{n_1}{n},$$

ahonnan a megbízhatóság mértéke (a rejtett hibák számának reciproka):

$$\frac{1}{n} = \frac{n_{12}}{n_1 n_2} .$$

akkor

13.3. A program bonyolultsága

Ennek lényeges összetevője a program szerkezeti (architekturális) bonyolultsága. A program akkor bonyolult, ha nehéz annak a működését megérteni.

Akkor könnyű egy program működését megérteni, ha

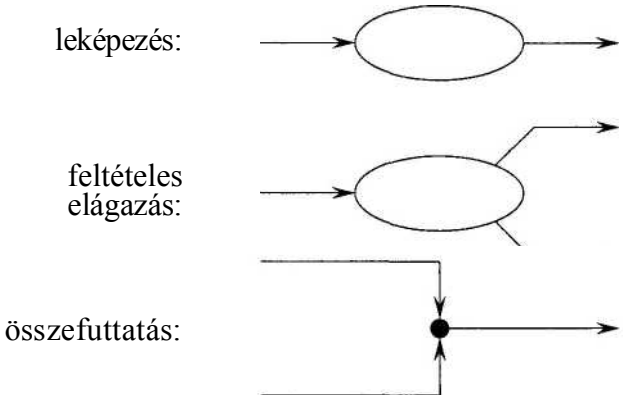
- egyszerű,
- az egymásra épülő absztrakciós szintek jól definiáltak,
- világosan elkülönül egymástól az egységek közötti kommuniká ciós felület és az egység szolgáltatásainak megvalósítása,
- az architektúra bővelkedik elegáns megoldásokban.

Felmerül a kérdés, hogy miként lehet mérni a program bonyolultságát. Ha megvizsgáljuk, hogy mivel lehet ezt becsülni, akkor két kézenfekvő lehetőség adódik:

- utasítások számával és/vagy
- döntések számával.

A döntések számának felhasználásán alapuló bonyolultságbecslésre mutatunk egy módszert a következőkben.

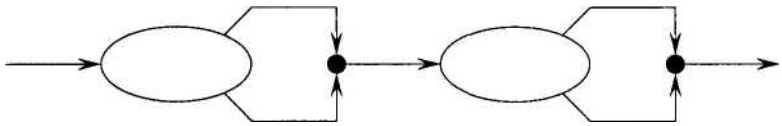
Tegyük fel, hogy a programban leképezések, feltételes elágazások (döntések) és összefuttatások szerepelnek (13.4. ábra), és minden csomóponton vezet legalább egy út a bemenő éltől a kimenő élig!



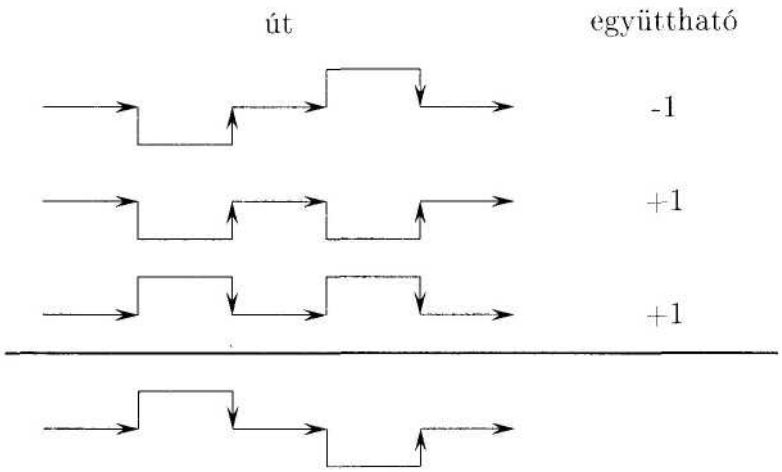
13.4. ábra. Program szerkezeti elemei

Ebben az esetben a programon átvezető, lineárisan független utak számának maximuma: $\pi+1$, ahol π a programban lévő döntések száma.

Például a 13.5. ábrán látható program esetében a független utak száma 3, ugyanis a 13.6. ábra alsó útja megkapható a felső három út lineáris kombinációjaként, a feltüntetett együtthatók segítségével.



13.5. ábra. Példa program



13.6. ábra. Lehetséges utak

13.4. Teljesítményelvű szoftvertechnológia

Nagy rendszereknél lényeges szempont lehet a rendszer hatékonysága. Hiába működik egy rendszer hibátlanul, ha az egyes műveletek elvégzésének ideje nagyobb egy előre meghatározott értéknél. Ha az időkorlátot átlépi a rendszer, akkor esetleg használhatatlanná válik. Ez a probléma nyilvánvaló valós idejű (például folyamatirányító) rendszerek esetén, de más esetben is fontos lehet. Gondoljunk például egy internetes áruházra, ahol az érdeklődők kiszolgálása nem tarthat túl hosszú ideig, mert akkor senki sem vásárolna ott! A gyakorlatban sajnos ez már előfordult, ugyanis egy ilyen árusítással foglalkozó cég nem tudta kielégíteni az igényeket, sőt még fogadni sem tudta az érdeklődőket. Ez komoly kereskedelmi visszaesést és bizalomvesztést eredményezett.

A rendszer hatékonyságát kétféleképpen lehet kezelni. Az egyik esetben megtervezzük és létrehozzuk az adott rendszert, majd megvizsgáljuk, hogy megfelel-e a hatékonysági követelményeknek. Ezt nevezzük reagáló módszernek. A másik esetben már a tervezés során megpróbáljuk a hatékonyságot elemezni, és csak olyan tervet fogadunk el, amely megfelel a követelményeknek. Ez az aktív megközelítés.

A gyakorlatban sokáig a reagáló megközelítés volt érvényben. A módszer követőinek érve, hogy először hozzunk létre egy helyesen működő rendszert, és amikor ez már megvan, ráérünk a hatékonysággal foglalkozni, ha egyáltalán szükség van rá. Ha a hatékonyság javítása elérhető az adott terv keretein belül, akkor ezzel a módszerrel nincs is különösebb gond. Az igazán nagy problémát az jelenti, ha a terv alkalmatlan a hatékonyság növelésére. Ekkor új tervet kell készíteni, és a rendszer fejlesztését tulajdonképpen elölről kell kezdeni. Ugyanakkor semmi sem garantálja, hogy sikerül egy megfelelő rendszert létrehozni. Sokszor a megoldási javaslat ebben az esetben a hardver fejlesztése, ami rendkívül költséges lehet egy-egy nagyobb rendszer esetén.

Az aktív megközelítés egyik legújabb eredménye a teljesítményelvű szoftvertechnológia *(SPE* - Software Performance Engineering). Ennek lényege, hogy nem igényel bonyolult matematikai apparátust, és illeszkedik az UML alapú objektumelvű tervezéshez. Néhány olyan projekt, amelyben ezt használták, és a teljesítmény lényeges tényező volt, bizonyította, hogy a végső költsége ennek a módszernek jóval kisebb, mint a reagáló megközelítésnek.

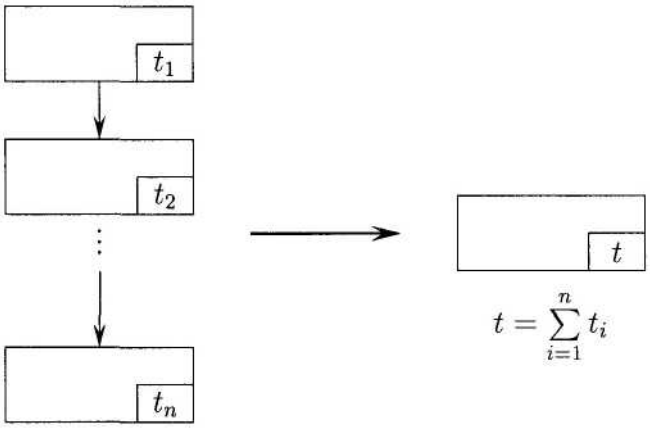
Első lépés a teljesítmény szempontjából lényeges használati esetek azonosítása. Ezekhez az esetekhez forgatókönyvet kell készíteni. Ennek eszköze a szekvenciadiagram. A forgatókönyvek alapján el kell készíteni a végrehajtási gráfokat, amelyek alkalmasak a futási idő elemzésére. Az elemzés során meghatározhatjuk az optimális, a legrosszabb és az átlagos eset futási idejét. Ennek ismeretében tudunk válaszolni arra a kérdésre, hogy elég hatékony-e a rendszer. Arra is lehetőség van, hogy megmondjuk, létre lehet-e hozni a megadott paraméterekkel rendelkező rendszert. Ha nem, akkor jelezhetjük a megrendelőnek ezt, és módosítani lehet az elvárásokon, mielőtt még akár a fejlesztő, akár a megrendelő lényeges ráfordításokat hajtott volna végre.

Az idők becslése a végrehajtási gráf egyszerűsítésén alapul. Tegyük fel, hogy minden egyszerű csúcsban szereplő művelet végrehajtási idejét meg tudjuk becsülni! Ez nem feltétlen pontos szám, lehet alsó korlát, felső korlát és átlagos eset is. Ezeket a csúcsokat „számított" csúcsokra cseréljük. Egy számított csúcsnak megfelelő téglalap

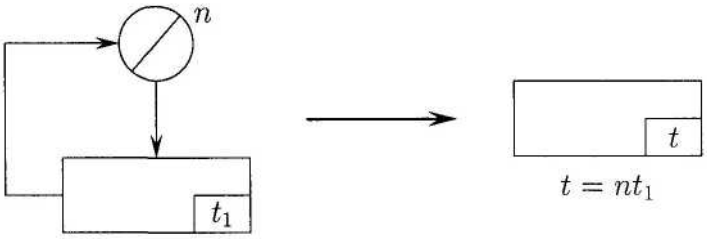
jobb alsó sarka tartalmazza a végrehajtási időt.

A végrehajtási gráf alapvető szerkezetei a szekvencia, az iteráció és a választás. Szekvencia esetén a benne szereplő számított csúcsokat helyettesíteni lehet egy számított csúccsal, amelynek ideje a szekvenciában szereplő csúcsok idejeinek összege (13.7. ábra). Iteráció esetén a ciklusmag idejét meg kell szorozni a végrehajtások számával (13.8. ábra). Mindhárom (optimális, legrosszabb, átlagos) idő meghatározása esetén ezt kell tennünk, csak a megfelelő időt kell figyelembe venni a felhasznált számított csúcsokban.

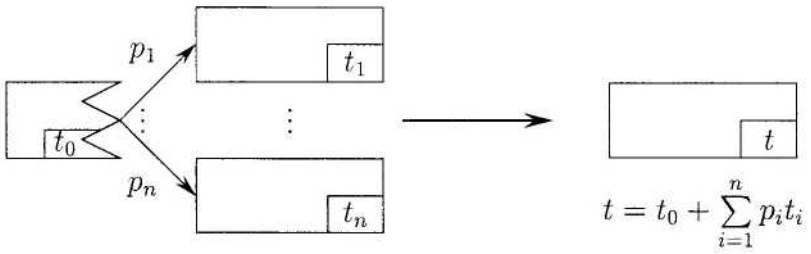
Választás esetén figyelembe kell vennünk a feltétel kiértékelési idejét, amit növelni kell a legrövidebb idejű esettel az optimális, a leghosszabb idejű esettel a legrosszabb idő meghatározásához. Az átlagos



13.7. ábra. Egyszerűsítés szekvencia esetén



13.8. ábra. Egyszerűsítés iteráció esetén



13.9. ábra. Egyszerűsítés az átlagos idő kiszámításához választás esetén

idő számításakor a kiértékelési időhöz az egyes ágak valószínűségekkel súlyozott idejét kell adnunk (13.9. ábra).

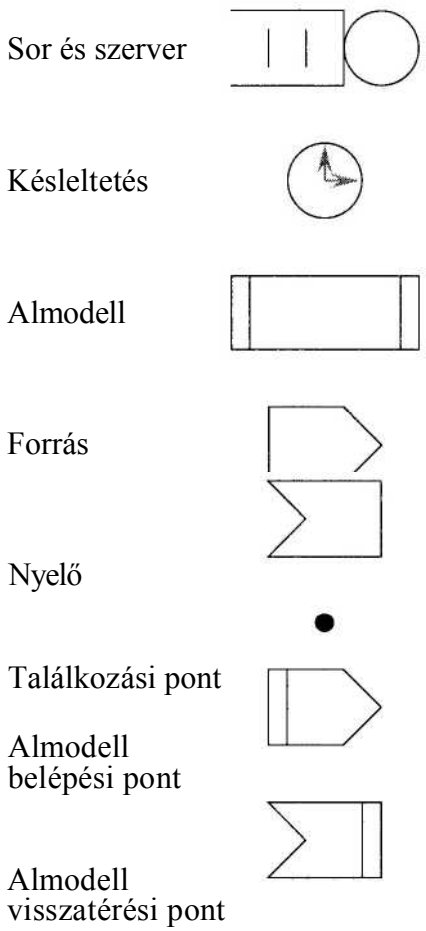
Az egyszerűsítéseket addig kell folytatni, amíg egyetlen csúcs marad. A csúcs ideje tartalmazza a becsült végrehajtási időt. Az egyszerűsítéseket célszerű automatizálni, léteznek erre megfelelő eszközök. Itt most csak szekvenciális rendszerekre illusztráltuk a módszert. Párhuzamos rendszerekben a párhuzamossági és hasító csúcsokat is kezelni kell, és a szinkronizációt is figyelembe kell venni. Ha a párhuzamos tevékenységek egymástól függetlenek, azaz nincs szinkronizáció, akkor a becslés egyszerű, hiszen csak a leghosszabb időt kell figyelembe vennünk, hogy megkapjuk a párhuzamos rész futási idejét. Ez a leghosszabb idejű tevékenység nem feltétlen ugyanaz az optimális, a legrosszabb és az átlagos esetben. Szinkronizáció esetén a szinkronizációs pontok közötti szakaszok idejét határozhatjuk meg hasonlóan, és ezeket kell összegeznünk.

Az eddigiek során feltettük, hogy egy egyszerű csúcsnak megfelelő tevékenység idejét meg tudjuk becsülni. Rendszerint ezek a tevékenységek egyszerűek, de a futási idő megállapítása során több összetevőt kell figyelembe vennünk. Ezért a becslés érdekében be kell vezetnünk *végrehajtási modelleket.* Célszerű olyan modellt választani, amelyben figyelembe lehet venni, hogy a folyamatok nem önmagukban léteznek. Egy időben több folyamat futhat, és ezek közös erőforrásokat használhatnak. Az erőforrásokra esetleg várakozni kell, ami növelheti a futási időt, és ez befolyásolja a rendszer viselkedését.

A választott modellben sorral rendelkező kiszolgáló egységeket kö-

tünk össze egy hálózatba (*QNM* - queueing network model). Egy gráffal szemléltetjük ezt a modellt is. A modell alapfogalmai, illetve a gráf csúcsai a következők (13.10. ábra).

Szerver: egy komponens, amely valamilyen szolgáltatást nyújt a szoftvernek (processzor, lemez, hálózati elem). Rendszerint egy sor tartozik hozzá.



13.10. ábra. A végrehajtási modell alapfogalmainak jelölése

Sor: ebben várakoznak a kiszolgálandó feladatok. Ha a szerver foglalt egy feladat érkezésekor, a feladat ebbe a sorba kerül, és itt vár, amíg a szerver szabad nem lesz.

Forrás: a tevékenység kezdete, eredete.

Nyelő: a tevékenység befejezése, kilépés.

Késleltetés: aktív erőforrás sor nélkül.

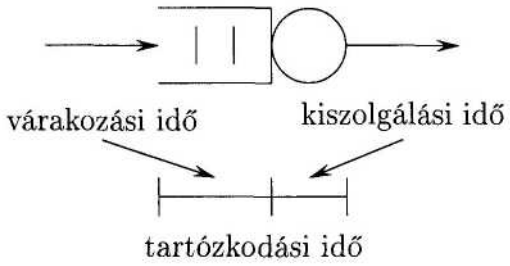
Találkozási pont: több él is bevezethet ebbe a csúcsba, és innen több él is kiindulhat.

Almodell: ez a csúcs egy később részletesen kifejtett modellt reprezentál.

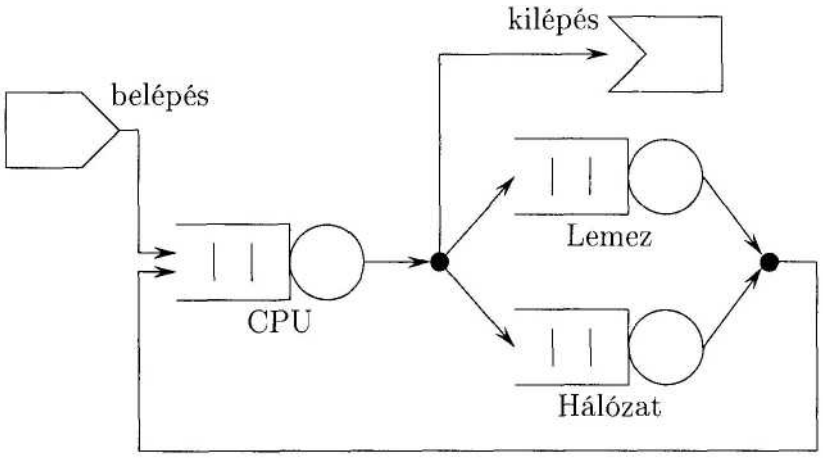
Almodell belépési pont: az almodellel leírt rész tevékenységeinek kezdete.

Almodell visszatérési pont: az almodellel leírt rész tevékenységeinek vége.

A futási idő meghatározásánál figyelembe kell venni, hogy egy folyamat mennyi ideig tartózkodik egy kiszolgáló egységnél. A tartózkodási idő a várakozási idő és a kiszolgálási idő összege (13.11. ábra). Egy tevékenység idejének kiszámításakor a tartózkodási időket kell összegeznünk.



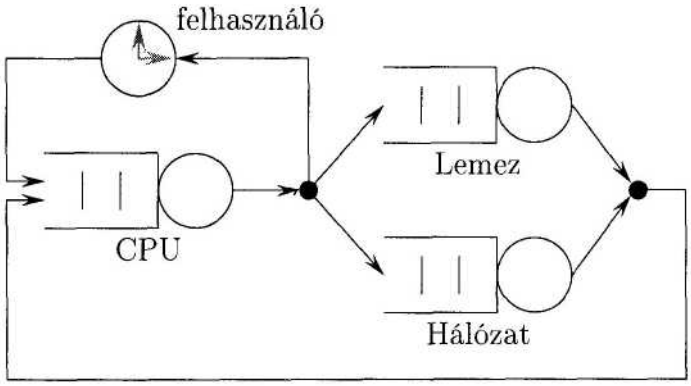
13.11. ábra. A futási idő meghatározásakor a tartózkodási időt kell használni



13.12. ábra. Nyílt modell

A 13.12. ábra egy egyszerű nyílt modellt tartalmaz. Ebben szerepel három szerver, amelyekhez egy-egy sor tartozik. A szerverek a központi egység, a merevlemez és a hálózat. A modell nyílt, mert a belépési ponton keresztül bármikor újabb tevékenység léphet be a modellbe. Egy tevékenységnél azt kell meghatároznunk, hogy melyik erőforráshoz hányszor kerül a végrehajtás során.

Zárt modell (13.13. ábra) esetén a felhasználó dönt arról, hogy mikor indít el egy újabb tevékenységet. Ezt a döntést az éppen végre-

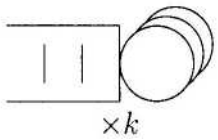


13.13. ábra. Zárt modell

hajtás alatt álló tevékenységről kapott információk befolyásolhatják.

Nyílt modell esetén valamilyen valószínűségi tényező figyelembevételével lehet a modellbe belépő feladatok gyakoriságát meghatározni, zárt modell esetén a felhasználó tevékenységeit kell elemezni. A modellben egy időben „végrehajtott" feladatok jellege és száma befolyásolja az egyes sorok hosszát, ami kihat a futási időre. Ezért szükséges a feladatokra vonatkozó becslés.

Előfordulhat, hogy több szerverhez tartozik egy közös sor. Ez nem ugyanaz, mint több, külön-külön sorral rendelkező szerver, mert ott amikor egy folyamat bekerül egy sorba, akkor már eldőlt, hogy melyik szerverhez kerül. (Nem mindegy, hogy például egy postán az összes ablakhoz egy sor tartozik, vagy minden ablaknál van külön-külön sor.) Ezt az esetet ezért meg kell különböztetnünk jelölésben is (13.14. ábra). Egy lehetséges példa egy többprocesszoros gép, ahol a feladatok egy sorba kerülnek, de mindig az éppen szabad processzor hajtja végre a feladatot.



13.14. ábra. Több szerver egy sorral

A következő példában az egyszerű csúcsban szereplő tevékenység, egy banki tranzakció végrehajtása a központi gépen. A tranzakciót hálózaton keresztül hajtják végre, és így is kell az információt továbbítanunk. A banki gép szempontjából vizsgáljuk az esetet, ezért a tranzakció beérkezésével nem foglalkozunk. Ebben az esetben a 13.12. ábrán látható nyílt modell használható a becsléshez. A tranzakció végrehajtása a modellen belül négy részre bontható, mindegyikhez meghatározhatjuk a szükséges erőforrásokat.

1. Felhasználó azonosítása: CPU, lemez.
2. Tranzakció ellenőrzése: CPU, lemez.
3. Eredmény rögzítése: CPU, lemez.

4. Eredmény közlése: CPU, hálózat.

Ha ismerjük, hogy a tevékenységek az egyes erőforrásoknál mennyi ideig tartózkodnak *(t_{cpu},t_{lemez},$t_{hálózat}$),* akkor a tevékenység futási idejét is megbecsülhetjük. Esetünkben ez $4t_{cpu} + 3t_{lemez} + t_{hálózat}$.

Tárgymutató

A
absztrakció, 75 paraméteres, 75
specifikáció szerinti, 75
applikatív, 75 axiomatikus, 76 külső felület szerinti, 76
absztrakt adattípus, 74
absztrakt osztály, 97
adatabsztrakció, 74
adatfolyam-diagram, 249
adattípus, 76
egyszerű, 76
művelet, 76
konstrukciós, 77
szelekciós, 77
összetett, 76, 77
aggregáció, 120
akció, 170
aktivációs diagram, 88, 254
szinkronizáció, 256
aktor, 214, 246
alrendszer, 278
particionális szerkezetű, 285
rétegszerkezetű, 283 vegyes szerkezetű, 286
alrendszerdiagram, 88
annotáció, 102 asszociáció, 105 attribútum, 90, 96

Á
ágens, 94, 281
állapot, 90, 162, 165
aggregáció, 176
jelölése, 177
általános fogalom, 186
általánosítás, 175
általánosított, 175
jelölése, 176
hisztorizációs, 186
jelölése, 187
jelölése, 169
osztályhoz rendelt, 164
pszeudo, 186
rendszer, 169
állapotdiagram, 87, 169
bonyolultság, 173
állapotinvariáns, 164
általánosítás, 126
többszörös, 130

D
dinamikus modell, 49, 161
dinamikus összekapcsolás, 81, 82
dinamikus szempont, 86, 87
dinamikus típus, 81
dokumentáció, 59
fejlesztői, 61
felhasználói, 61

E
együttműködési diagram, 88, 245
esemény, 163, 167
általános forma, 168
fázisok, 170
jelölése, 169
evolúciós modell, 31

forgatókönyv, 266 funkcionális
modell, 49, 245
megalkotása, 252
függőségi táblázat, 19

H
használati esetek diagramja, 89
használati szempont, 85, 89

implementáció, 30, 52
implementációs stílus, 53
implementációs stratégia, 52
alulról felfelé, 52
felülről lefelé, 52
modul izolációs, 53
objektumelvű, 53
implementációs szempont, 86, 88
irány, 106

K
karbantartás, 30, 56
kategória, 279 keret, 327
kliens, 94, 281 kohézió, 74
komponens, 275
komponensdiagram, 88, 275
kompozíció, 124
konfigurációs diagram, 88
konkrét osztály, 97
konstruktor, 77 korlátok
diagramja, 22 korlátolt
láthatóság, 90 kovariancia
probléma, 131 környezeti
szempont, 86, 88
követelményelemzés, 45
követelményleírás, 30, 39, 89
funkcionális, 42
nem funkcionális, 44

M
megszorítás, 102
megvalósíthatósági tanulmány, 39
mérföldkövek, 19 minősítő, 107
modell, 63, 89
alternatíva, 34
cél, 34
evolúciós, 31
kockázat, 34
kockázatkezelés, 34
korlátozás, 34

spirális, 33
validáció, 34
vízesés. 30
modellalkotás, 63
modul, 74
multiplicitás, 106

N
navigálhatóság, 107, 117
nézetrendszer, 85, 87

O
objektum, 90
absztrakt forma, 90
aktiváció, 214
rekurzív, 214 aktív,
94 azonosítás, 91
deklaráció, 90 felület,
91, 95 interfész, 90
jelölése, 100 konkrét
forma, 91 külső felület,
90 létrehozás, 212
megsemmisítés, 212
művelet, 90, 92
állapot változtató, 93
export, 90, 92, 93
import, 90, 92
iterátor, 94
kiértékelő, 93
konstruktor, 93
szelektor, 93
passzív, 94
reprezentáció, 90
objektumdiagram, 87, 104
objektumelvű nyelv, 75
objektumelvű programozás, 74
objektumelvű tervezés, 50
objektumosztály, 96 osztály.
96
absztrakt, 97
attribútum, 96
export felület, 96
import felület, 96
jelölése, 99
konkrét, 97
megvalósítás, 96
műveletek, 96
név, 96
private, 97
protected, 97
public, 97
sablon, 97
jelölés, 100
specifikáció, 96
osztálydiagram, 87, 103
osztályszerep, 211
aktivációs életvonal, 212
életvonal, 212

Ö
öröklődés, 126 öröklődési
technika, 127

polimorfizmus, 81
problématér, 63, 89
procedurális tervezés, 50, 73

programfejlesztési modell, 24
programkészítési fázisok, 30
programspecifikáció, 48
projekt, 17
projektmenedzselés, 16, 17
prototípus, 45, 46

R
reláció, 71
    általánosítási, 71
    függőségi, 71
    megvalósítási, 72
    minősítés, 114
    multiplicitás, 108
    navigálhatóság, 117
    reflexivitás, 115
    társítási, 71
    több osztály közötti, 116
    tulajdonság, 113
rendszerkövetés, 30, 56
reprezentáció, 52
részfeladat, 19
rétegszerkezet, 283
    nyílt, 284
    zárt, 284

S
sablon, 97 specializáció, 80, 126
    többszörös, 129
specifikáció, 27, 30 spirális modell, 33 statikus modell, 49, 89 statikus szempont, 85, 87 statikus típus, 81
strukturális dekompozíció, 50
subclass, 80 superclass, 80

Sz
származtatás, 80, 127
szekvenciadiagram, 87, 211
kiegészítések, 220 hierarchia, 221 iteráció, 224
párhuzamosság, 225
választás, 222 szelektor, 77
szerep, 106, 109 kiemelt, 110
megnevezés, 109 sorrendiségi, 111 több,112
szerkezeti szempont, 85
szerver, 94, 281 szintenkénti finomítás, 73 szoftverkrízis, 15 szoftverprojekt, 17
szoftvertechnológia, 15, 16

társítási reláció, 105
társult osztály, 106
terv, 51
    bonyolultság, 52
    fejleszthetőség, 52
    kohézió, 51
    módosíthatóság, 52
    összekapcsolódás, 51
tervdiagram, 21

tervezés, 30, 49
objektumelvű, 50, 73
procedurális, 50, 73
tervminta, 321 tesztelés, 27, 54
egységteszt, 54
fehér doboz, 55
fekete doboz, 55
rendszerteszt, 55
típusosztály, 78
ábrázolás, 79
átvett szolgáltatások, 79
paraméter, 78
típusobjektumok, 78
típusöröklődés, 74, 80
specializáció, 80
újrafelhasználás, 80, 83

Ú
újrafelhasználás, 80, 83

Ü
üzenet, 215, 246
argumentum, 246
aszinkron, 217
címzett, 246
egyszerű, 215
időhöz kötött várakozás, 216
irány, 246
randevú, 216
szinkronizációs, 215
visszatérési, 217

V
validáció, 30, 54
verifikáció, 30, 54
végrehajtási gráf, 266
aszinkron hívás, 271
egyszerű csúcs, 267
egyszerűsítés, 337
hasító csúcs, 267
ismétlési csúcs, 267
késleltetett szinkron hívás, 271
kiterjesztett csúcs, 267
párhuzamossági csúcs, 267
számított csúcs, 337
szinkron hívás, 271
választási csúcs, 267
végrehajtási modell, 339
almodell, 341 belépési pont, 341 visszatérési pont, 341
forrás, 341
késleltetés, 341
nyelő, 341
sor, 340
szerver, 340
találkozási pont, 341
vízesés modell, 30

Irodalomjegyzék

[1] Ian Sommerville: Software Engineering,
Addison-Wesley, 1983 [742],
ISBN-0201-42765-6
(5. kiadás utánnyomásai: 1996, 1997, 1998).
Magyar kiadás: Szoftverrendszerek fejlesztése,
Panem Könyvkiadó, 2002 [752],
ISBN-963-545-311-6 [2] Bertrand Mayer: Object Oriented Software Construction,
Prentice Hall PTR, 1997 [1254],
ISBN-0-13-629155-4.

[3] Jacques Loeckx, Hans-Dieter Ehrich, Markus Wolf : Specification of Abstract Data Types,
John Wiley and Sons, Inc., 1996 [260],
ISBN-0-471-95067-X. [4] James Rumbaugh, et al.: Object Oriented Modelling and Design,
Prentice Hall International Inc., 1991 [500],
ISBN-0-13-630054-5. [5]
Pierre-Alain Muller: Instant UML,
WROX Press Ltd, 1997 [343],
ISBN-1-861000-87-1
(UML 1.1-es verzió). [6] Sinan Si Albir: UML in a Nutshell. A Desktop Quick Reference,
O'REILLY, 1998 [273],
ISBN-1-56592-448-7.

[7] Grady Booch, James Rumbaugh, Ivar Jakobson: The Unified Modeling Language User Guide,
Addison-Wesley Longman, Inc., 1999 [482],
ISBN-0-201-57168-4.

[8] Murray Cantor: Object Oriented Project Management with UML,
Wiley Computer Publishing, John Wiley & Sons Inc., 1998 [343],
ISBN-0-471-25303-0.

[9] Hans-Erik Erikson, Magnus Penker: Business Modelling with UML,
Wiley Computer Publishing, John Wiley & Sons, Inc., 2000 [459],
ISBN-0-471-29551-5.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, *Addison-Wesley Longman, Inc.,* 1995 [395], ISBN-0-201-63361-2.

[11] Connie U. Smith, Lloyd G. Williams: Performance Solutions - A Practical Guide to Creating Responsive, Scalable Software,
Addison-Wesley, Pearson Education, Inc., 2002 [510],
ISBN-0-201-72229-1.