Objektum orientált szoftverfejlesztés

Kondorosi, Károly

Szirmay-Kalos, László

László, Zoltán

Objektum orientált szoftverfejlesztés

Kondorosi, Károly
Szirmay-Kalos, László
László, Zoltán

Az eredeti mű a ComputerBooks Kiadó gondozásában jelent meg. Az elektronikus kiadás az NKTH által lebonyolított Felsőoktatási Tankönyv- és Szakkönyv-támogatási Pályázat keretében készült, a DocBook XML formátumot Bíró Szabolcs készítette.

Ajánlás

Könyvünk 1997-es megjelenése óta sok kritikát, de még több pozitív visszajelzést kaptunk. Számos oktatási intézményben látjuk a kötelező, vagy ajánlott irodalmak listáján, és – örömünkre – nem csak az informatika szakokon. Ez azt bizonyítja, hogy az objektumorientált megközelítés egyre inkább hat az informatika alkalmazási területein, és egyre inkább képes betölteni azt a szerepét, hogy alapja lehessen az alkalmazási területek szakértői és az informatikusok által egyaránt érthető formális rendszermodelleknek. Megtisztelő, hogy könyvünk a DIGIT2005 digitális szakkönyvpályázaton támogatást nyert, és így internetes kiadásban is elérhetővé válik.

Ugyanakkor nem kis fejtörést okozott számunkra, hogy hogyan reagáljunk az eltelt tíz esztendő szakmai fejlődésére, hiszen a szoftverfejlesztés az informatika egyik legdinamikusabban fejlődő területének és egyben üzletágának bizonyult ebben az időszakban. Ennek megfelelően új irányzatok, módszerek, eszközök, fogalmak jelentek, jelennek meg, amelyek közül nem egyszerű kiválasztani a lényegeseket, a maradandókat. A komponens-technológia, az aspektus-orientált és az intencionális programozás, a versengő és egymással kölcsönhatásban fejlődő Java és .NET technológiák, az agilis szoftverfejlesztés, a C# nyelv, az analízis-, architekturális és tervezési minták, az új, integrált fejlesztő környezetek (mint például a Visual Studio, vagy az Eclipse) – mind-mind új, lényeges elemekkel színesítették a palettát, és ismeretük elengedhetetlen egy képzett informatikus számára. A szakma egyik legnagyobb hatású konzorciuma, az Object Management Group (OMG), számos szabványt, ajánlást dolgozott ki, amelyek eredményeként a módszertanok, jelölésrendszerek egységesedtek, a fogalmak tisztábbá váltak. Az egységes modellező nyelv (Unified Modelling Language, UML), a modellvezérelt architektúra (Model Driven Architecture, MDA), az objektum metamodell (Meta-Object Facility, MOF), az objektumok együttműködésének elosztott rendszerekben is alkalmazható szabványa (Common Object Request Broker Architecture, CORBA), az interfészleíró nyelv (Interface Definition Language, IDL), széles körben elterjedt szabványokká váltak. A konzorciumnak a szakma legnagyobb piaci szereplői is tagjai, így a szabványok gyakorlati alkalmazása és a forgalmazott termékekben való megjelenése is biztosított. Az OMG dokumentumainak jelentős része nyílt, elérhető a www.omg.org portálon.

Az internetes kiadás előkészítésekor irreális célkitűzés lett volna minden lényeges újdonság tárgyalása, akár csak felületesen is. Valamilyen mértékű átdolgozást azonban feltétlen szükségesnek láttunk, hiszen – egy tankönyvtől elvárhatóan – a jelölésrendszernek alkalmazkodnia kell a szabványokhoz, a példaprogramoknak pedig lefuttathatóknak kell maradniuk a mai rendszereken is.

Az internetes kiadást tehát az eredeti könyvhöz képest a következők jellemzik:

- Megtartottuk az eredeti célkitűzést, azaz bemutatjuk az objektumorientált szoftverfejlesztés alapjait: az analízist, tervezést és a C++ nyelvű implementációt.
- A bevezető, áttekintő fejezetekben csak ott változtattunk, ahol az új eredmények alapján a szöveg feltétlen korrekcióra szorult.
- Az OMT (Object Modelling Technique) módszertan és jelölésrendszer helyett az UML-t alkalmazzuk. Ennek megfelelően az adatfolyamokat (dataflow) nem tárgyaljuk, a használati eseteket (use-case) pedig bevezetjük.
- A C++ nyelv bemutatásakor és a mintafeladatok implementációiban ma elterjedten használt nyelvi környezetet veszünk alapul, így az olvasó a közölt programokat könnyebben fordíthatja és futtathatja az általa elérhető számítógépeken.

Ismételten köszönjük mindazoknak, akik észrevételeikkel, tanácsaikkal segítették munkánkat. Külön köszönjük Dr. Goldschmidt Balázs munkáját, aki ábráinkat az OMT jelölésrendszerről UML-re alakította. Ugyancsak megkülönböztetett köszönet illeti Bíró Szabolcsot, aki az internetes megjelenésre alkalmas formátumra alakította szövegeinket és ábráinkat.

Reményeink szerint a felfrissítés a könyv hasznára válik, és mind az oktatók és hallgatók, mind a gyakorlati szakemberek hasznos olvasmánya marad az elkövetkező években is.

Budapest, 2007. február

A szerzők

Tartalom

Előszó

A szoftverfejlesztés folyamata a programozás történetének mintegy fél évszázada alatt jelentős átalakulásokon esett át – néhány kiválasztott guru mágikus ténykedését a szoftver ipari előállítása váltotta fel. Az "ipari" termeléshez szigorú technológiai előírásokra, hatékony termelőeszközökre és a hajdani guruk helyett mind a technológiát, mind pedig az eszközöket jól ismerő, fegyelmezett szakembergárdára van szükség. A szoftverfejlesztés során a szabványos technológiai előírásokat az ún. fejlesztési módszertanok fogalmazzák meg, az eszközöket pedig a CASE rendszerek és a programozási nyelvek jelentik. A módszertanok alkalmazása során megértjük a megoldandó problémát és körvonalazzuk a számítógépes megvalósítás mikéntjét. Napjainkban számos módszertan vetélkedik egymással, amelyek közül az alkalmazási területek jelentős részénél az ún. objektum-orientált megközelítés került vezető pozícióba. Ennek oka talán az, hogy a többi módszerrel szemben az objektum-orientált elvek nem a számítástechnika elvont fogalmait kívánják ráerőltetni az életre, hanem megfordítva az élet természetes és számítástechnika-mentes működését hangsúlyozzák a feladatok megoldása során.

Az objektum-orientált szemlélettel megragadott feladatok programjait olyan programozási nyelveken érdemes implementálni, amelyek maguk is segítik az objektum-orientált gondolkozást, különben a programfejlesztés utolsó fázisában, az implementáció során esetleg elveszítenénk az objektum-orientált megközelítés számos előnyét. Számos objektum-orientált programozási nyelv létezik, melyek közül messze a C++ nyelv a legelterjedtebb.

Ezen könyv az objektum-orientált szoftverfejlesztés fázisait kívánja bemutatni, a megoldandó probléma megértésétől kezdve a megoldás menetének körvonalazásán át egészen az implementáció részletes kidolgozásáig. A fejlesztés különböző fázisait igen színvonalas idegen nyelvű munkák tárgyalták ezen könyv megjelenése előtt is. A könyvek egy része az analízis és tervezés lépéseit ismerteti, míg más művek a C++ nyelv szintaktikáját és szemantikáját mutatják be. A mi könyvünk főleg abban kíván újat adni, hogy a szoftvertechnológiai lépéseket és a C++ nyelv ismertetését összekapcsolja, lehetőséget teremtve arra, hogy a gyakorló programfejlesztő számára egy egységes kép alakuljon ki. Ezzel reményeink szerint elkerülhető lesz az a – oktatói tapasztalataink alapján elég gyakori – hiba, hogy a fejlesztő külön-külön remekül kezeli az objektum-orientált analízis és tervezés lépéseit, és jól ismeri a C++ nyelvet is, de C++ programját mégsem az elkészült tervekre építi, így az analízis és tervezés hamar felesleges és értelmetlennek látszó teherré válik számára. A könyvben ismertetett objektum-orientált analízis és tervezés döntő részben a ma legelterjedtebb OMT (Object Modelling Technique) módszertanra épül, amelyet kiegészítettünk és összekapcsoltuk az implementációval.

Terjedelmi korlátok miatt a könyv nem törekedhet teljességre az implementációs eszköz, a C++ nyelv bemutatásánál. Egyrészt ismertnek tekinti a C++ nyelvnek az összes C nyelvtől örökölt konstrukcióját, másrészt pedig nem tárgyal néhány C++ újdonságot (például a kivételek (*exception*) kezelése).

Ajánljuk ezt a könyvet mind a diákoknak, mind pedig a gyakorló rendszertervezőknek, programfejlesztőknek és programozóknak, ha már jártasságot szereztek a C programozási nyelvben. Reményeink szerint ezen könyv segítségével a kezdő C++ programozók megtanulhatják a nyelvet és az objektum-orientált fejlesztés módszertanát, de haszonnal forgathatják a könyvet a C++ nyelvet már jól ismerők is.

A könyv a Budapesti Műszaki Egyetem Műszaki Informatika és Villamosmérnöki karain a szerzők által előadott "Objektum-orientált programozás", "Szoftver technológia", "Objektum-orientált szoftverfejlesztés" tárgyak anyagára és a B. Braun fejlesztőintézetnél tartott tanfolyam anyagára épül. Hálásak vagyunk hallgatóinknak és a B. Braun fejlesztőinek, akik az előadásokon feltett kérdéseikkel, megjegyzéseikkel sokat segítettek a könyv szerkezetének és tárgyalásmódjának finomításában. Végül hálával tartozunk a B. Braun fejlesztőintézet épületében működő felvonónak, amely a harmadik mintafeladatot ihlette.

Budapest, 1997.

A szerzők

1. fejezet - 1. Bevezetés a szoftverfejlesztésbe

A szoftver karrierje egyelőre felfelé ível. Az alig néhány évtizedes történet nem mentes viharoktól és ellentmondásoktól. Éppen csak elkezdődött a történet és máris krízisről beszéltek. Ma is tart a vita, hogy ebből sikerült-e kilábalnia. Egy dolog biztos, a szoftver egyike az utóbbi évek legdinamikusabban fejlődő üzletágainak. Előállításával – amatőrök és profik – valószínűleg több millióan foglalkoznak. Hogy csinálják? Hogy kellene csinálniuk?

A szoftver előállítása tudományos vizsgálódások tárgya is. A publikációk száma hatalmas. Kezdetben programozási technikák, később módszerek, aztán módszertanok, paradigmák jelentek meg. Ma tudományos körökben is talán a legnépszerűbb módszertan és paradigma az objektumorientáltság.

Tekintve, hogy a szoftver ma vitathatatlanul tömegtermék, előállítási módszereit egyre inkább indokolt technológiáknak nevezni.

1. 1.1. Szoftvertechnológiák

Az objektumorientált szoftverfejlesztés a lehetséges és használatos szoftvertechnológiák egyike. A szókapcsolat szokatlan; érdemel némi vizsgálódást, hogy a szoftverrel kapcsolatosan miként értelmezhető a technológia fogalma, milyen sajátosságok jelennek meg a hagyományos technológiákhoz képest.

Kezdjük a technológia általános fogalmával!

A technológia valaminek az előállításával foglalkozik. Általában megkövetelünk bizonyos ismérveket ahhoz, hogy ezt a kifejezést használjuk, nem tekintünk mindenféle barkácsolást technológiának.

A társadalom által a gyakorlatban felvetett problémák megoldására szolgáló dolgok tudományos ismeretek alkalmazásával történő, gazdaságos előállításának mikéntjét nevezzük technológiának.

A definícióban minden szónak különös jelentősége van, ezért érdemes a mondatot alaposan elemezni. A technológia lényegében dolgok előállításának *mikéntje; módszerek, eszközök, technikák* együttese, amelyek alkalmazásával véges számú lépésben a kívánt dologhoz jutunk. A mikénthez két jelző is tartozik, nevezetesen a tudományosság és a gazdaságosság. A *tudomány* eredményeinek alkalmazásától remélünk garanciát arra, hogy a módszerek időtől és tértől függetlenek, *megbízhatóak, megismételhetőek* lesznek. A *gazdaságosság* az a szempont, amelynek alapján a lehetséges megoldások közül választunk. Fontos a definícióban szereplő *dolog* szó és jelzője is. *Dolog* az, aminek előállítására a technológia irányul. A definícióban kikötjük, hogy a technológia csak olyan dolgok készítésének módjával foglalkozik, amely dolgok a *gyakorlatban előforduló problémák* megoldását célozzák. További szűkítést jelent a *társadalmi* vonatkozás is. Ez szűkebb értelemben azt jelenti, hogy az előállítandó dolgok iránt társadalmi igény nyilvánul meg, tágabban pedig azt, hogy mind a dolognak, mind a technológiának magának komoly egyéb társadalmi (jogi, környezeti, etikai, stb.) vonatkozásai is vannak. Az előállítandó dolgokra és a technológiára vonatkozó társadalmi elvárások törvényekben, szabványokban és ajánlásokban fogalmazódnak meg.

Az elmúlt néhány évtized egy új területen, az *információ-feldolgozás* területén vetett fel egyre növekvő társadalmi igényeket. A dolgok (termékek), amelyek ezeket az igényeket kielégítik, összefoglalóan *információ-feldolgozó rendszerek* névvel jelölhetők.

Az információ fogalmának tisztázása nem egyszerű, a filozófusokat is komoly feladat elé állítja. Mindenesetre nem anyagi természetű valamiről van szó, de az információ tárolása, feldolgozása, továbbítása az anyag törvényszerűségeit kihasználó eszközökkel lehetséges. Az információt mindig valamilyen anyagi dolog, fizikai jellemző hordozza, amit hordozónak vagy közegnek (média) nevezünk. A közeg kiválasztásán túl az ábrázoláshoz meg kell állapodnunk abban is, hogy a fizikai jellemző mely értékeit, milyen jelentéssel használjuk.

Az információnak egy adott közegen, adott szabályok szerint történő ábrázolását az információ reprezentációjának nevezzük.

Ugyanannak az információnak többféle reprezentációja lehetséges. Gondoljunk csak arra, hogy valamiről értesülhetünk például az újságból, az írott szöveg elolvasása útján, de ugyanaz az információ megszerezhető úgy is, hogy meghallgatjuk a rádió híreit! Az információ-feldolgozó rendszerekben az eszközök anyagi jellege, fizikai működése a megoldandó feladat szempontjából közömbös, egyébként csak annyiban lényeges, amennyiben az információ ábrázolásához, az információval végzett műveletek konkrét végrehajtásához ennek ismeretére szükség van.

A mai információ-feldolgozó rendszerek általános felépítését az 1.1. ábrán láthatjuk.

1.1. ábra

A rendszer magja egy (esetleg több összekapcsolt) általános célú számítógép, amelyre ráépül egy általános célú programcsomag, az alapszoftver (operációs rendszer, adatbázis-kezelő, hálózatkezelő szoftver stb.). Ezeket a komponenseket építőelemeknek tekinthetjük. A harmadik réteg – a feladatspecifikus felhasználói szoftver – az, amelynek létrehozása a feladat megoldásának döntő része. Az információ-feldolgozási problémák megoldói az esetek többségében felhasználói szoftver készítésével foglalkoznak, jóllehet munkájuk eredményeként egy *rendszer* – hardver-szoftver együttes – oldja meg a feldolgozási feladatokat. Ugyanezen oknál fogva beszélhetünk az információ-feldogozó rendszerek létrehozásával kapcsolatosan elsősorban szoftvertechnológiáról.

Természetesen – az anyagi technológiákhoz hasonlóan – az alapanyagok és a komponensek készletét önmagukban is folyamatosan fejlesztjük. Bizonyos speciális feladatok pedig ennek ellenére sem oldhatók meg kész elemek összeépítésével. A szoftvertechnológia módszerei természetesen az alapszoftver fejlesztése során is alkalmazhatók, sőt nagyrészük még a hardverfejlesztésben is, hiszen ilyenkor is információ-feldolgozó rendszereket kell létrehoznunk. A kialakult szóhasználat szerinti *szoftvertechnológia*, *szoftverfejlesztés* (software engineering), *rendszerfejlesztés* (system engineering) és *információtechnológia* fogalmak által jelölt területek jelentősen átfedik egymást. A továbbiakban akkor használjuk a *rendszer* fogalmat, ha hangsúlyozni kívánjuk mondandónk érvényességének kiterjeszthetőségét a hardver-szoftver együttesekre is.

Vizsgáljuk meg a szoftver sajátosságait, értelmezzük vele kapcsolatosan a technológia fogalmát, és értékeljük a szoftvertechnológia mai helyzetét!

A "mi a szoftver?" kérdésre adandó válasz nem egyszerű, ha arra gondolunk, hogy szoftvervásárlás címén általában mágneses anyagokat és könyveket szoktunk kapni. Azonosítható-e a szoftver a mágneslemezzel? Nyilvánvalóan nem, a lemez csak a szoftver hordozója. Hasonlóképpen hordozónak tekinthetők a nyomtatott dokumentációk. A szoftver tehát az információhoz hasonló tulajdonságokat mutat. Valóban, a szoftvert értelmezhetjük úgy, mint azt az információt, amely megmondja, hogy egy (vagy több) adott berendezést hogyan kell működtetni egy feladat megoldása érdekében.

Elkerülendő a filozófia csapdáit, a szabványok a **szoftvert** mint programok, adatok és dokumentációk együttesét definiálják, amelyek különféle anyagi formát ölthetnek (**reprezentációk**). Maga a szoftver szellemi termék, és mint ilyen, számos – a technológiák szempontjából furcsa – tulajdonsággal rendelkezik. Anyagtalanságának

fontos következménye, hogy az anyag ismert törvényei (Newton törvény, Maxwell egyenletek stb.) rá nem érvényesek. Ennek egyik jelentős előnye, hogy a szoftver anyagi értelemben nem avul, nem kopik és több éves használat után is ugyanannyi hiba van benne, mint a megvételekor. A szoftvert reprezentációi hordozzák, de az igazi érték természetesen nem a hordozó. Általában nagyon egyszerű a reprezentációk másolása, hiszen úgy lehet azokat sokszorozni, hogy annak az "eredetin" nincs nyoma. Ez a tulajdonság kis mértékben a technológia megszokott fogalmát is módosítja. Amíg az anyagi technológiák legfontosabb célja a minőség megőrzése a sorozatgyártásban (legyen a százezredik darab is olyan, mint az első), addig a szoftver esetében a reprezentációk többszörözése általában nem jelent különösebb gondot. A szoftvertechnológiák esetében a "tömeggyártás" nem az egyedi példányok előállítását, hanem sokkal inkább szoftverek különféle változatainak, verzióinak szisztematikus és követhető elkészítését jelenti.

Kérdés, hogy a mai szoftverkészítési gyakorlat a tudományosság és a gazdaságosság kritériumainak megfelel-e. Erősen vitatható, hogy a szoftverkészítés mai általános gyakorlata kellően kidolgozott, tudományos alapokon nyugszik. A piac óriási felvevőképessége és bizonyos – elsősorban a minőség területén mutatott – igénytelensége még ma is eltűri a "mindegy-hogy-hogyan" programkészítést, sőt alkalmanként jobban értékeli, mint az alapos, igényes szakmai munkát. A témában mutatkozó zűrzavart az is jelzi, hogy szemben az építészettel, ahol csak a tervezői névjegyzékbe felvett építészkamarai tagok tervezhetnek önállóan, szoftvert bárki készíthet, akár szakmai képesítés nélkül is.

A technológia hiányára egyébként a gazdaság hívta fel a figyelmet valamikor a 60-as évek végén a *szoftverkrízis* felismerésével. A krízis lényege, hogy a szoftverek fejlesztői minden költség- és időkeretet rendszeresen túlléptek, s mindezek ellenére sem voltak képesek megbízható és a felhasználói követelményeket legalább elfogadható szinten teljesítő szoftvert előállítani. Hitelesnek tartott vizsgálati adatok szerint az elkészült szoftvereknek – többszöri javítás után is – kevesebb, mint 25 %-át vették használatba. A szakértők a probléma tanulmányozása során arra a következtetésre jutottak, hogy a krízis eredendő okai a fejlesztés módszerességének és szervezésének (menedzsment) hiányosságaiban keresendők. Ez a felismerés lökést adott a nagy rendszerek uralására alkalmas módszertanok és programozási nyelvek fejlődésének, valamint a technikai aspektusokon túlmenően a hatékony munkaszervezési (menedzsment) módszerek kialakításának. Ekkor elkezdődött a szoftverfejlesztés technológiájának kialakulása.

Mára számos fejlesztési módszertant dolgoztak ki és publikáltak. Ezek némelyikéhez számítógépes támogató eszközöket is kifejlesztettek (CASE – *Computer Aided Software Engineering* – számítógéppel támogatott szoftver mérnökség). A CASE eszközöket és módszereket egyre több helyen alkalmazzák. Sok országban a minőség biztosítására szabványokat és ajánlásokat vezettek be. Tudományos intézetekben kutatásokat folytatnak a bizonyíthatóan helyes programok készítésének – matematikai szigorúsággal rögzített – módszereit illetően.

Összefoglalásul leszögezhetjük, hogy a mai szoftvergyártás még elég messze áll attól, hogy "jól technologizáltnak" nevezzük, de határozott lépések történtek a szükséges irányba.

2. 1.2. A fejlesztés elvi alapjai

Elsőként tisztázzuk, mit is értünk fejlesztésen!

Minden terméknek van élettörténete (életciklusa), amely a rá vonatkozó igény felmerülésétől a termék használatból való kivonásáig (feledésbe merüléséig) tart. A ciklus elnevezés főként azokban az esetekben indokolt, amikor egy terméknek rendre újabb, továbbfejlesztett változatait állítják elő. Ilyenkor minden továbbfejlesztési lépést úgy tekinthetünk, mint az élettörténet megismétlődését, azaz ciklusokról beszélhetünk. Valamennyi életciklus-típuson belül megtalálható az a tervező, kísérletező tevékenység, amelyik jellegzetesen az új vagy módosított termék előállítására vonatkozó igény megszületésétől a gyártás megindításáig tart.

Fejlesztésen egy termék életciklusának azt a szakaszát értjük, amelyik a termék előállítására vagy módosítására vonatkozó igény felmerülésétől a gyártás megindításáig tart.

A fejlesztési szakaszban részint magát a terméket (gyártmányfejlesztés), részint a technológiát (gyártástervezés) kell megtervezni és kísérleti úton igazolni. Természetesen a termék és a technológia kölcsönösen hatnak egymásra.

Felidézve mindazt, amit az előző pontban a szoftverről mondtunk, vegyük vizsgálat alá a szoftver élettörténetét!

A szoftver elnevezés a kemény/lágy (*hard/soft*) ellentétpárból származik. Az ellentétpár azt tükrözi, hogy – szemben a számítógép merev, a gyártás után már nem változtatható jellegével – a szoftver könnyen, rugalmasan,

akár "házilag" is módosítható. Ez a könnyű változtathatóság csábít is a változtatásokra. A szoftver életének ciklikus jellegét a módosítások, továbbfejlesztések sorozata adja.

Azt is megállapíthatjuk, hogy a szoftver élettörténetében az egyes szakaszok súlya jelentősen eltér a hagyományos termékeknél megszokottól. Miután a gyártás – azaz a példányok előállítása – nem okoz különösebb gondot, ezzel szemben a termék maga bonyolult, így az életciklus döntő része a fejlesztés, ezen belül is a termékfejlesztés. Ennek megfelelően *a szoftvertechnológia* nem a gyártás miként jére, hanem *a termékfejlesztés miként jére koncentrál, erre próbál szisztematikus módszereket adni.*

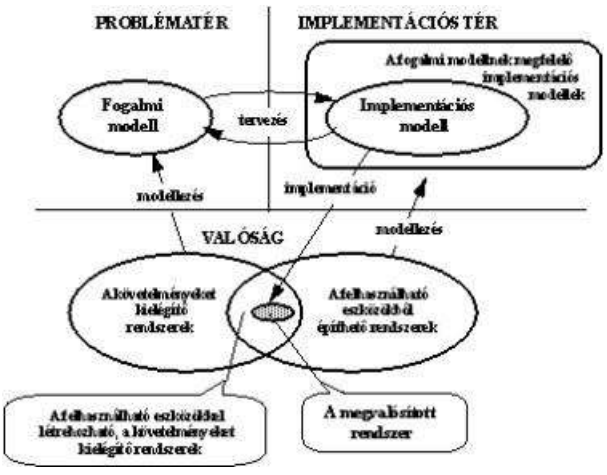
2.1. 1.2.1. A szoftverfejlesztés alapproblémái

A szoftverfejlesztés olyan folyamat, amely során egy adott igényt kielégítő szoftver első példányát létrehozzuk. A folyamat indulhat "nulláról", azaz úgy, hogy nincs jelentős előzménye, nincs meglévő, hasonló szolgáltatású rendszer a birtokunkban; de indulhat úgy is, hogy egy meglévő szoftver továbbfejlesztését határozzuk el. Fejlesztésről csak akkor beszélünk, ha a folyamat során jelentős újdonságokat hozunk létre.

A fejlesztés folyamatának három jellegzetes lépését különíthetjük el: a **modellezést**, a **tervezést** és az **implementációt**. Ezt a három tevékenységet az 1.2. ábra alapján értelmezhetjük.

A fejlesztés jelentős része gondolati síkon, azaz a modellek síkján folyik. A valóságos dolgokról bennünk élő képeket nevezzük *modelleknek*, ennek a képnek a kialakítását pedig *modellezésnek*. Egy modell sohasem tükrözi a valóság minden apró részletét, hiszen mindig valamilyen céllal alkotjuk. A modellben azokra a tulajdonságokra koncentrálunk, amelyek a cél elérése szempontjából fontosak. Ezért a modell szükségképpen absztrakt (ld. később). Milyen összefüggés áll fenn a modell és a valóságos dolgok között? Ugyanarról a valóságos dologról több modellt is alkothatunk, amelyek mindegyike más-más szempontból emel ki lényeges tulajdonságokat. Fordított irányban: egy modellnek több valóságos dolog is megfelel. Ezek a valóságos dolgok a modellben figyelmen kívül hagyott (lényegtelennek tartott) tulajdonságaikban térnek el egymástól.

A szoftverfejlesztés során valamely speciális feladat megoldására univerzális eszközöket használunk, azaz speciális rendszerként viselkedő számítógépes rendszert hozunk létre. Eközben mind a speciális rendszerről kialakított modellre, mind a felhasználható számítástechnikai eszközökről kialakított modellre szükségünk van, a cél eléréséhez pontosan ennek a két modellnek az elemeit kell megfeleltetnünk egymásnak.



1.2. ábra

Vizsgáljuk meg, milyen viszony áll fenn egy termék és a létrehozása során születő modelljei között!

Az 1.2. ábrán megjelenő szimbólumok három térrészben, három "világban" helyezkednek el. Az ábra alsó része a valóságot jelöli. A bal felső térrész az úgynevezett **problématér**, ahol a speciális feladathoz kötődő gondolatkörben mozoghatunk. Ebben a térben alakíthatjuk ki a létrehozandó rendszer **fogalmi modelljét.** A jobb felső térrész az úgynevezett **implementációs tér**, amelyet a megvalósításhoz felhasználható eszközök bennünk élő képe határoz meg. Ebben a térben helyezkedik el az **implementációs modell**, amely esetünkben a megvalósítandó rendszert, mint számítástechnikai eszközök valamilyen együttesét írja le.

Igen sok olyan rendszer létezhet a valóságban, amelyik kielégíti a támasztott követelményeket. A fogalmi modell ezek mindegyikének megfeleltethető. Ugyancsak igen sok rendszer létezhet, (most ne firtassuk, hogy ezek egyáltalán véges sokan vannak-e), amelyeket az implementációs eszközökből el tudnánk készíteni. Ezek közül azok lesznek a *lehetséges megvalósítások*, amelyek egyben a támasztott követelményeknek is megfelelnek. A *megvalósított valóságos rendszer* a lehetséges megvalósítások egyike. Ennek a konkrét rendszernek egy implementációs modell felel meg. Ugyanennek az implementációs modellnek azonban több – egymástól lényegtelen részletekben különböző – valóságos rendszer is megfeleltethető.

A fogalmi- és az implementációs modellek megfeleltetését vizsgálva megállapíthatjuk, hogy egy fogalmi modellnek több implementációs modell is megfeleltethető (a lehetséges megvalósítások implementációs modelljei). A fogalmi modellhez tartozó, neki megfeleltethető *legkedvezőbb* implementációs modell létrehozása a **tervezés** feladata.

A fejlesztés kezdetekor általában a problématér fogalmaival kifejezett igényekből, vagyis a fogalmi modell egy változatából indulunk ki, és a követelményeket kielégítő valóságos rendszerhez kell eljutnunk. A legtöbb gondot ennek során az okozza, hogy a szóban forgó rendszerek *bonyolultak*, amiből számos probléma származik. Rögtön elsőként említhetjük, hogy a bonyolult modellek *áttekinthetetlenek*, megértésük, kezelésük nehézkes. Nincs olyan zseniális tervező, aki egy rendszernek, amelynek programja több tízezer forrássorból áll, és sok (mondjuk tíznél több) processzoron fut, valamennyi részletét egyidejűleg fejben tudná tartani, illetve egy-egy tervezői döntés minden kihatását átlátná. A bonyolult rendszerek modelljeit és a modellek közötti megfeleltetéseket csak több lépésben tudjuk kidolgozni. Minden lépés hibalehetőségeket rejt magában, mégpedig minél bonyolultabb a rendszer, annál nagyobb a hibázás esélye. Az áttekintés nehézségein túl komoly probléma a *résztvevők információcseréje* is. A rendszerrel szemben támasztott követelményeket általában nem a fejlesztők, hanem a felhasználók (megrendelők) fogalmazzák meg, maga a fejlesztés pedig általában csoportmunka. Igen jó lenne, ha a folyamat minden résztvevője pontosan ugyanazt a gondolati képet tudná kialakítani önmagában az amúgy igen bonyolult rendszerről. Sajnos ezt nagyon nehéz elérni, nagy a félreértés veszélye. (Tapasztalatok szerint a probléma egyszemélyes változata is létezik: igen jó lenne, ha egy fejlesztő néhány hét elteltével fel tudná idézni ugyanazt a képet, amit korábban kialakított magában a rendszerről.)

A bonyolultságon valahogyan uralkodni kell.

A felvetett problémák alapján három kérdést veszünk részletesebb vizsgálat alá. Az első az, hogy milyen módon tudunk bonyolult rendszerekről áttekinthető, kezelhető modelleket alkotni. A második, hogy mit tehetünk a modellek egyértelműsége és ellenőrizhetősége érdekében. Ez a vizsgálódás a rendszerről készülő dokumentumok (reprezentációk) tulajdonságainak vizsgálatához vezet, miután saját korlátos emlékezetünk, valamint a résztvevők közötti információcsere egyaránt megköveteli a következetes dokumentálást. A harmadik kérdés, hogy a fejlesztés folyamatát hogyan célszerű megszervezni. Milyen lépéseket milyen sorrendben hajtsunk végre annak érdekében, hogy a nagyobb buktatókat elkerüljük, az elkövetett hibák minél előbb kiderüljenek, és minden fázisban fel tudjuk mérni, hogy a munkának mekkora részén vagyunk túl?

2.2. 1.2.2. Uraljuk a bonyolultságot!

A bonyolultságot általában úgy érzékeljük, hogy nagyon sok mindent kellene egyszerre fejben tartanunk, és ez nem sikerül. Figyelmünket hol az egyik, hol a másik részletre koncentráljuk, és a váltások közben elfelejtjük az előzőleg tanulmányozott részleteket. A bonyolultság uralása érdekében olyan modelleket kellene alkotnunk, amelyek lehetővé teszik, hogy az egyidejűleg fejben tartandó információ mennyiségét csökkenthessük, a tervezés közben hozott döntések kihatását pedig az éppen áttekintett körre korlátozzuk. Erre két alapvető eszköz áll rendelkezésünkre, a részletek eltakarása (*absztrakció*), illetve a probléma (rendszer) egyszerűbb, egymástól minél kevésbé függő részekre bontása (*dekompozíció*). Ezzel a két gondolkodási technikával olyan struktúrákat hozhatunk létre, amelyeken mozogva figyelmünket a rendszer különböző részeire, különböző nézeteire irányíthatjuk.

Az absztrakció olyan gondolkodási művelet, amelynek segítségével a dolgok számunkra fontos jegyeit elvonatkoztatjuk a kevésbé fontosaktól, az általánosítható tulajdonságokat az egyediektől.

Más szavakkal: az *absztrakció* műveletével eltakarjuk a szükségtelen, zavaró részleteket, így egy modellből kevesebb részletet tartalmazó új modellt állítunk elő. Az *absztrakció szintjét* a részletezettség aprólékossága jellemzi. Minél nagyobb, bonyolultabb, összetettebb dolgot tekintünk eleminek, azaz a vizsgálat szempontjából pillanatnyilag tovább nem oszthatónak, annál magasabb absztrakciós szintről beszélhetünk. Fordítva: ahogy közeledünk az apró részletekhez, egyre konkrétabbak leszünk. Egy modellből a *finomítás* műveletével

állíthatunk elő egy alacsonyabb absztrakciós szintű, részleteiben gazdagabb modellt. Természetesen az absztrakciós szintek nem abszolútak, hiszen csak egymáshoz képest van jelentésük.

Dekompozíciónak nevezzük egy rendszer együttműködő, egyszerűbb részrendszerekre bontását, ahol a részrendszerek együttesen az eredeti rendszernek megfelelő viselkedést mutatnak.

Egy dekompozíciós lépésben általában részrendszereket definiálunk, valamint meghatározzuk a részrendszerek együttműködésének módját. Ezután a részrendszereket egymástól függetlenül fejleszthetjük tovább. A dekompozíció közvetlenül nem jár az absztrakciós szint csökkentésével. Az azonban igaz, hogy a dekompozíció általában egy finomítási lépés után válik szükségessé, amikor a feltáruló részletek már áttekinthetetlenné válnak. Fordított irányban: a részekre bontás egy absztrakciós lépésben eltűnhet, hiszen éppen az válhat érdektelenné, hogy milyen részrendszerekből áll a teljes rendszer.

Az absztrakció és a dekompozíció az ember ösztönös gondolkodási technikája. Létezésüket a beszélt nyelvben is megfigyelhetjük. Amikor fogalmakat használunk, akkor konkrét dolgok absztrakt modelljeivel van dolgunk. Amikor például autóról beszélünk, akkor bármilyen márkájú, típusú, színű, méretű, korú autóra gondolhatunk. Amikor az autó működését magyarázzuk, célszerűen a szerkezetét vázoljuk fel, a motort, a kereket, a féket stb.. Ezeket a szerkezeti egységeket azután külön-külön tárgyalhatjuk tovább.

Az absztrakciót és a dekompozíciót mind a fogalmi, mind pedig az implementációs modell kialakításakor bevethetjük. Így tulajdonképpen nem egyetlen fogalmi modellel és egyetlen implementációs modellel dolgozunk, hanem mindegyik egy-egy modellsorozattá válik. A sorozat tagjai finomítással, illetve absztrakcióval állíthatók elő egymásból.

Szemléltessük az elmondottakat egy vasúti helyfoglalási rendszer példáján! A fogalmi modell magas absztrakciós szintű elemei a következők: jegy, helyjegy, vonat, indulási és célállomás, kocsiosztály, számla, foglalás, törlés, fizetés. Ha kísérletet teszünk a fogalmak tartalmának definiálására, akkor alacsonyabb absztrakciós szintre kerülünk, közelebb a konkrétumokhoz. Például elemezhetjük a helyjegyen vagy a számlán szereplő adatokat (vonatszám, kocsiszám, ülőhely sorszáma stb.), vagy a foglalás törlésének mechanizmusát. Végül eljuthatunk a helyjegyen vagy a számlán szereplő szövegekig, illetve a vasúti alkalmazott elemi tevékenységéig (például a jegy lepecsételése), mint konkrétumig. De mit is tekintsünk konkrétumnak? Hiszen elemezhetnénk tovább a pecsételés közben a bélyegző mozgását, a jól olvasható lenyomathoz szükséges nyomóerőt stb. Valószínűleg mindannyian egyetértünk azonban abban, hogy a megoldandó feladat szempontjából ezek már lényegtelen (nem releváns) részletek. A pecsételést nyugodtan tekinthetjük tovább nem részletezendő elemi műveletnek.

Természetesen a szoftver saját világában is különböző absztrakciós szintű fogalmakat használhatunk. Magas absztrakciós szintű fogalmak például az alkalmazói program, az operációs rendszer, az adatbázis-kezelő. Alacsonyabb szintűek a fájl, az ütemező, a lekérdező parancs. Az absztrakciós szintet tovább csökkentve olyan fogalmakig jutunk el, mint a rekordszerkezet vagy a programmodul. Folytathatjuk a finomítást az elemi adatok (int, char) és vezérlő szerkezetek (switch, while, for) irányában. Konkrétumnak például a szoftver egy közismert programozási nyelven leírt kódját tekinthetjük. Ennek további finomítása, például a fordító által előállított gépi utasítássorozat, vagy a futtatható program bitsorozata már többnyire érdektelen.

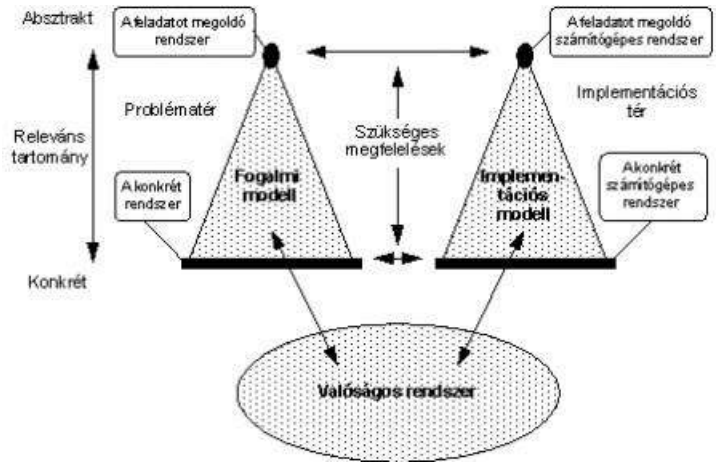
A fenti példából azt a következtetést is levonhatjuk, hogy minden feladathoz a problématérben és az implementációs térben egyaránt tartozik az *absztrakciós szintnek egy releváns tartománya*. Ezen belül érdemes a problémát megragadni. Sem a túl általános, sem a túl aprólékos modell nem használható.

Láthattuk, hogy a vasúti helyfoglaló rendszer legkülönbözőbb absztrakciós szintjei megfogalmazhatók a problématér elemeivel (célállomás, kocsiszám, pecsételés), és semmi szükség sem volt olyan számítógépes fogalmakra, mint a fájl, a rekord vagy a bit. A helyfoglalás működik számítástechnika nélkül is. A szoftver fejlesztése során kell megtalálnunk a fogalmi modell elemeinek megfelelőit a számítógépes világban, az implementációs modellben. A teljes helyfoglaló rendszer egy összetett hardver-szoftver együttes lesz. A járatokat nyilvántartó alrendszernek megfelel egy adatbázisokkal dolgozó programrendszer. A vonatokat például fájlokkal reprezentálhatjuk, amelyekben egy-egy rekord az ülőhelyeknek felel meg. A foglalás folyamata egy számítógépes üzenetváltással írható le.

A fogalmi és az implementációs modell különböző absztrakciós szintjeinek szemléltetését az 1.3. ábrán láthatjuk.

Ez az ábra az 1.2. ábra módosított változata, amely minden térben csupán egyetlen rendszert ábrázol. A fogalmi és az implementációs modellt most egy-egy háromszög jelöli. A háromszögek nem is egyetlen modellt, hanem modellsorozatokat jelképeznek. Minden vízszintes metszetre elképzelhetünk egy modellt. A háromszögek csúcsa a releváns tartomány legmagasabb absztrakciós szintjének felel meg. Lefelé haladva az absztrakciós szint csökken, a háromszög egyre szélesedő vízszintes metszete pedig a feltáruló egyre több részletet jelképezi. A háromszögek alapja a konkrétnak tekinthető modellt jelöli (a releváns tartomány alsó határa). A továbbiakban általában nem hangsúlyozzuk a modellsorozat jelenlétét, de a fogalmi, illetve az implementációs modellen modellsorozatokat fogunk érteni.

Felvetődik a kérdés, hogy a finomítással, illetve a dekompozícióval mekkora lépésekben haladjunk. Egy-egy lépésben mennyit tárjunk fel a részletekből, hány részre osszunk fel egy magasabb szinten egységnek látszó elemet? Erre ismét az emberi gondolkodás tanulmányozása alapján adhatunk választ. Tulajdonképpen az a kérdés, hogy figyelmünket hány különböző dolog között tudjuk még viszonylag kényelmesen megosztani. Természetesen a válasz egyéni képességektől, a fejben tartandó dolgok bonyolultságától erősen függ, de különböző vizsgálatok megkísérelték a jellemző érték meghatározását. Sommerville szerint [Som89] ez a szám hét, más publikációk ennél valamivel nagyobb értéket is megengednek. Általában senki nem jelöl meg ötnél kisebb és harmincnál nagyobb számot. Ez azt jelenti, hogy az emberi megértésre szánt modelleket és azok leírásait úgy célszerű megszerkeszteni, az absztrakciós szinteket oly módon egymásra építeni, hogy egyidejűleg ne kelljen tíz körüli számnál több fogalmat fejben tartani.



1.3. ábra

A szoftver fejlesztése során egyrészt létre kell hozni a fogalmi, másrészt az implementációs modell különböző absztrakciós szintű változatait (a modellsorozatokat). Harmadrészt pedig el kell végezni a két modell megfeleltetését.

Helyesen megtervezett és létrehozott rendszerben a legmagasabb absztrakciós szintű fogalmi modell (nevezetesen a "feladatot megoldó rendszer") szükségképpen megfelel a legmagasabb absztrakciós szintű implementációs modellnek (nevezetesen a "feladatot megoldó számítógépes rendszer"-nek). Ugyanígy szükségképpen fennáll a megfelelés a legalacsonyabb (konkrét) szinten is. Ha ezek a megfelelések nem állnának fenn, akkor a rendszer nem teljesítené feladatát. A szinteket áthidaló struktúra azonban nem szükségképpen hasonló a két térben. Helyfoglaló rendszerünkben például nem biztos, hogy találunk egy fájlt és egy vonatot, amelyek kölcsönösen és egyértelműen megfeleltethetők egymásnak. Az azonban biztos, hogy a rendszer kimenetein konkrét helyjegyeknek megfelelő nyomtatott bizonylatokat kell kapnunk. A tapasztalat azt mutatja, hogy ezek a közbülső megfeleltetések nem szükségszerűek, fennállásuk rendkívül hasznos.

A fejlesztési módszertanok eddigi történetéből egyértelműen leszűrhető tendencia, hogy a bonyolultságot akkor tudjuk uralni, vagyis akkor tudunk kezelhető, módosítható, karbantartható szoftvert előállítani, ha a problématér és az implementációs tér fogalmait minden absztrakciós szinten minél inkább igyekszünk megfeleltetni egymásnak.

Más szavakkal: a problématér bármilyen absztrakciós szintű fogalma legyen felismerhető és elkülöníthető az implementáció megfelelő szintű modelljében, sőt magában az implementációban is. E tendencia jegyében

alakult ki a könyv tárgyát képező objektumorientált módszertan is. Sajnálatosan ezen elv következetes betartását megnehezíti, hogy a fogalmi modellt tükröző implementáció hatékonysága nem mindig kielégítő.

2.3. 1.2.3. A leírás szigorúsága

Amikor a fejlesztés során létrehozunk egy modellt, az adott nézetből, adott absztrakciós szinten ábrázolja a rendszert. Ahhoz, hogy ezt a modellt később ismételten fel tudjuk idézni, valamint másokkal meg tudjuk ismertetni, dokumentálnunk kell, létre kell hoznunk a modellnek emberi értelmezésre szánt leírását. Azt mondtuk, hogy a modell a valóság gondolati képe, és mint ilyen, személyhez kötődik. Kérdés, hogy tudunk-e a modellről olyan leírást készíteni, amelyet mindenki azonosan értelmez.

Teljes általánosságban a kérdésre nagyon nehéz lenne válaszolni, de beérjük kevesebbel is, megelégszünk azzal, hogy csak azok értsék azonosan, akik a rendszer létrehozásában valamilyen módon részt vesznek.

A kérdésre igen választ csak akkor kaphatunk, ha a leírás (reprezentáció) egyértelmű, pontos és kimerítő. Ezt úgy érhetjük el, hogy azok, akiknek a leírást értelmezniük kell, megegyeznek bizonyos tartalmi és formai szabályok szigorú betartásában, azaz a leírást **formalizálják.**

Formálisnak nevezzük az olyan reprezentációt, amely csak pontosan definiált fogalmakat, szerkezeteket és műveleteket használ, és a definíciók megadásának formáit is rögzíti.

A szoftver konkrét implementációs modelljének szintjén (ami nem más, mint a forrásnyelvű leírás) a szabályok adottak, ezeket a programozási nyelv definiálja. A programnyelvet a programozók és a fordítóprogramok is értelmezik. A magasabb absztrakciós szinteken készülő dokumentumokat azonban – különösen a fejlesztés kezdeti szakaszában – igen különböző előismeretekkel, felkészültséggel rendelkező személyek (leendő felhasználók, menedzserek, fejlesztők) értelmezik. Ezért a szabályok esetleg kimerülnek annak megkötésében, hogy a dokumentum milyen beszélt nyelven készüljön (például készüljön magyarul). Ez pedig, mint látni fogjuk, aligha tekinthető formálisnak, és komoly félreértések forrása lehet.

A különböző kifejezési módok skáláján a legkevésbé formálisnak az élő beszédben előforduló szerkezeteket tekintjük. Szándékosan kerültük a mondat kifejezést, mivel a "kötetlen" beszédben gyakorta nem is használunk egész mondatokat. Már maga a kötetlen szó is a formáktól való függetlenséget jelenti. Az élő beszéd szerkezeteinek értelmezése erősen függ a szituációtól, amelyben a szavak, mondattöredékek elhangzanak. Továbbá a közölt információk tárolása sem megoldott (hangfelvétel persze készíthető, de ennek kezelése, a szövegkörnyezet felidézése nehézkes).

Nem véletlen, hogy az életben valamennyi fontos dolgot írásban rögzítünk. Például egy jogszabály sem egyéb, mint az életviszonyok szabályozásának formalizált leírása. Ismert, hogy még a jól tagolt, paragrafusokba szedett, korrektül fogalmazott jogi kijelentéseket is meglehetősen tágan lehet értelmezni.

A reprezentációk köznyelvi, szöveges megfogalmazásának problémája kettős. Egyrészt a köznyelvben használt fogalmak nem elég egyértelműek, másrészt a nyelvi elemekből képezhető szerkezetek szabályai sem elég szigorúak. A fogalmi egyértelműség problémájának érzékeltetésére gondoljunk a számítógépes rendszerekre vonatkozó követelmények között gyakran előforduló "optimális", "gyors" "rugalmasan bővíthető" stb. megfogalmazásokra, amelyek önmagukban aligha értelmezhetők. Az irodalmi művekben – sőt már az általános iskolai fogalmazásokban is – elvárt stílusjegyek – mint például szinonimák használata szóismétlések helyett – egy szoftver dokumentum értelmezését bizony nem teszik könnyebbé. A szerkezeti egyértelműség problémájára pedig csak egyetlen példa: élettapasztalatunkat félretéve próbáljuk eldönteni, hogy vajon a *"Megetette a lovat a zabbal."* kijelentésben a ló ette-e a zabot, vagy a zab a lovat.

A formalizáltság mértékének növelésétől végeredményben azt várjuk, hogy a reprezentáció pontosan és egyértelműen leírja a modellt, ami pedig reményeink és szándékaink szerint minden lényeges vonásában pontosan tükrözi a valóságot. A valóság és annak formalizált reprezentációja közötti egyezőséget pedig azért keressük, mert ha ez fennáll, akkor a formalizmuson értelmezett és igazoltan helyes műveleteket végrehajtva a valóságra vonatkozó korrekt következtetésekre juthatunk.

A formalizált reprezentáció gyakorta nem szöveges, hanem például grafikus formát ölt. Egy adott jelölésrendszer betartásával készített ábra nem csak a formalizáltság szempontjából, hanem kifejező erejét tekintve is igen kedvező tulajdonságokat mutat. Minden leírásnál többet jelent például egy ház esetében mondjuk az alaprajz, vagy a homlokzat képe. Akik láttak szabadalmi leírásokat, pontosan tudják, hogy milyen bonyolult, *"... azzal jellemezve, hogy..."* alakú mondatokban lehet csak egy nagyon egyszerű ábrát szövegesen

leírni. Érdemes utánagondolni, hogy vonalakat és köröket rajzoló program futásának eredménye (az ábra) és a program szövege (nagyon formális leírás) között kifejező erőben mekkora különbség van. Nem véletlenül tartjuk az ábrákat a szövegnél kifejezőbbnek. Mivel a rajz – műszaki, építészeti vagy villamos kapcsolási rajz – lényegesen szigorúbb szabályoknak felel meg, mint az élő szöveg, ezért a rajzból kiindulva jól definiált, ellenőrzött lépések segítségével hamar eljuthatunk a realizálásig. Gondoljunk a villamos kapcsolási rajz alapján nyomtatott áramköri lapot szerkesztő rendszerekre!

A legszigorúbban formalizált leírásoknak a matematikai modelleket tekintjük.

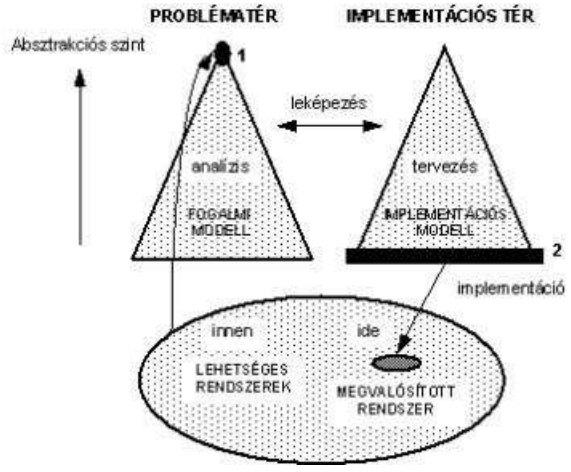
2.4. 1.2.4. A fejlesztés folyamata

Egy "nulláról induló" szoftverfejlesztés általában úgy kezdődik, hogy a valóságban zajló folyamatok megfigyelése alapján felmerül egy probléma, aminek megoldása lehetségesnek és célszerűnek látszik valamilyen számítógépes rendszer alkalmazásával. Tekintve, hogy a valóságot eleve a fogalmi modell alapján figyeljük és értjük meg, kiindulási pontunk általában a fogalmi modell "csúcspontja" (lásd 1.4. ábra). Gondolatainkban megjelennek a rendszer körvonalai, mégpedig igen magas absztrakciós szinten. Ezeket a körvonalakat általában igen kevéssé formalizált leírással tudjuk megadni.

Mindez például történhet úgy, hogy a leendő felhasználó megkeres bennünket, és élő beszédben, a saját terminológiáját használva előadja meglehetősen ködös elképzeléseit arról, hogy mit várna a rendszertől. Általában még a kitűzendő feladat tekintetében is tanácsokat kér. Innen kell eljutnunk addig, hogy a problémát megoldó rendszer elkészül, és a valóságban működik. Közben – az absztrakciós réseket áthidalva – ki kell alakítanunk a teljes fogalmi és a teljes implementációs modellt, valamint a konkrét implementációs modell alapján létre kell hoznunk a konkrét, a valóságba átültetett implementációt.

Az 1.4. ábrán nyomon követhetjük a fenti folyamatot, és – a korábbiakhoz képest már kissé finomítva – értelmezhetjük a fejlesztési folyamat különböző tevékenységeit.

A valóság gondolati képének kialakítását **modellezésnek**, a fogalmi modell szerkezetének kialakítását **analízisnek**, az implementációs modell szerkezetének kialakítását **tervezésnek** nevezzük. A konkrét megvalósítást **implementációnak**, a fogalmi és az implementációs modell megfeleltetését pedig **leképezésnek** nevezzük. A tervezés megjelölést gyakran tágabb értelemben használjuk, ilyenkor beleértjük a leképezést is (lásd 1.2. ábra).

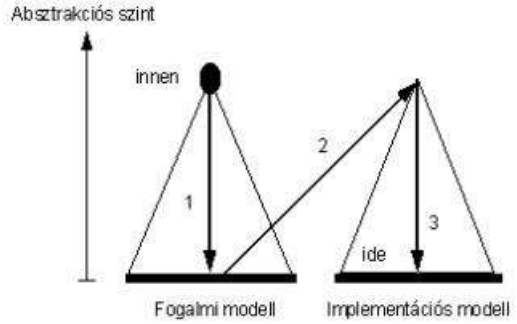


1.4. ábra

A legtöbb esetben a folyamatot az (*1*) jelű pontból indítva a (*2*) jelű pontba kell eljutnunk, miközben mindkét teljes modellt megalkotjuk, azaz mindkét háromszöget kitöltjük. Ezt az utat többféleképpen bejárhatjuk. A bejárás két szélsőséges példáját láthatjuk az 1.5. és az 1.6. ábrákon.

Az 1.5. ábra szerint haladva először a fogalmi modellt alakítjuk ki teljes egészében – analizálunk – méghozzá úgy, hogy a létrehozott reprezentációk absztrakciós szintjét fokozatosan csökkentjük. Ezt felülről-lefelé **(top-down)** haladásnak nevezzük. Az absztrakciós szintet úgy csökkentjük, hogy döntéseket hozunk arra nézve, hogy az addig nem részletezett fogalmakat, tulajdonságokat és műveleteket milyen módon építjük fel egyszerűbb

komponensekből. A legmagasabb absztrakciós szintről tehát döntések sorozatával jutunk a konkrétumok szintjére. Minden döntés finomítja a modellt, újabb részleteket tár fel belőle. Minden döntéshez ki kell választanunk valamit, amiről döntünk. Ezt a valamit **domináns fogalomnak** vagy **strukturáló objektumnak** nevezzük. Az így végrehajtott döntési sorozat pedig **lépésenkénti finomítás** néven ismert.

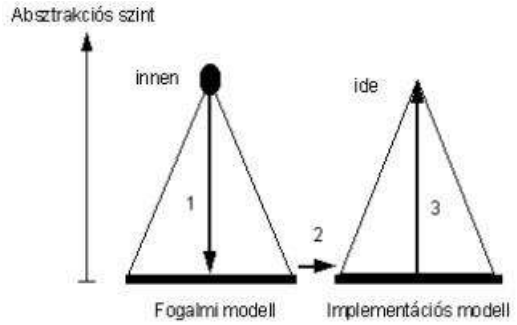


1.5. ábra

Az alábbiakban egy példán szemléltetjük a finomítás folyamatát. Tekintsünk egy házat, amely szobákból áll. Ha egy szobát konkrétabbá akarunk tenni – azaz precízebb, a részleteire kitérő leírást kívánunk adni – akkor választanunk kell egy szempontot, ami szerint finomítunk. Amennyiben az épület alakja érdekes számunkra, akkor a domináns fogalom a geometria lesz, és a szobát mint geometriai alakzatot írjuk le. Ha lakberendezői szempontból vizsgálódunk, akkor a bútorzattal jellemezzük a szobát. Ha üvegessel tárgyalunk, őt minden bizonnyal a nyílászárókról kell tájékoztatnunk.

Az 1.5. ábra szerint először lépésenkénti finomítással kialakítjuk a teljes fogalmi modellt. Ezután áttérünk az implementációs modellre, ahol ugyancsak ezt a stratégiát követve haladunk. Kérdés, hogy az implementációs modell finomítása közben eltekinthetünk-e a már kész fogalmi modelltől. Nyilvánvalóan nem, hiszen a konkrét implementációnak ugyanúgy kell viselkednie, mint a konkrét fogalmi modellnek. Az implementációs modell finomítása közben tehát előbb-utóbb meg kell céloznunk a konkrét fogalmi modell leképezését. Sajnos semmiféle garancia sincs arra nézve, hogy az implementációs modell finomításának kezdeti szakaszában ne hozzunk olyan rossz döntéseket, amelyek megnehezítik a fejlesztést.

Egy másik stratégia látható a 1.6. ábrán. A kezdeti (*1*) szakaszban hasonlóan járunk el, mint az előbb. Ezt követően azonban az implementációs modellt alulról-felfelé **(bottom-up)** építkezve hozzuk létre. A (*2*) lépés során az implementációs modell konkrétumaiból összerakjuk a fogalmi modell konkrétumait **(szintézis)**, majd az implementáció kezelhetősége érdekében egyre magasabb absztrakciós szintű implementációs fogalmakat (például adatbázisok, modulok, fájlok, taszkok stb.) alkotunk. Ez a stratégia sem garantálja a fogalmi és az implementációs modell struktúrájának hasonlóságát, csak a feltétlenül szükséges megfeleltetéseket biztosítja. Emiatt nehéz a program megértése és a fogalmi modellben történő legkisebb változtatás is az implementáció jelentős mértékű átdolgozását teheti szükségessé.

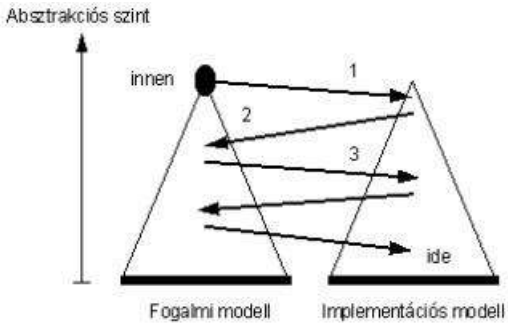


1.6. ábra

A fenti két alapeset problémáinak kiküszöbölésére számos módszertan javasolja az 1.7. ábra szerinti stratégiát. Az alkalmazott gondolatmenet: egy kicsit analizálunk, egy kicsit tervezünk, és ezt ismételgetjük. Végrehajtunk egy finomítási lépést a fogalmi modellen, meggondoljuk ennek következményeit, kihatását az implementációs

modellre, majd ennek figyelembevételével az implementációs modellen is végrehajtunk egy finomítási lépést. Ezt ismételgetve végül olyan megoldáshoz jutunk, amelyben a fogalmi modell és az implementációs modell szerkezetileg – azaz a legkülönbözőbb absztrakciós szinteken is – fedi egymást. A módszer óriási előnye, hogy az implementációban felismerhetők lesznek a problématér fogalmai. Emiatt a szoftver könnyebben megérthető, ha pedig módosítás válik szükségessé, tudjuk hova kell nyúlnunk.

Az utóbb vázolt stratégia alkalmazásának nehézsége abban áll, hogy a fogalmi modellek a felhasználási területtől függően a legváltozatosabb alakokat ölthetik. Vannak szakmák, amelyek előszeretettel írják le problémáikat például differenciálegyenletek formájában, mások élő nyelven fogalmazott szabályok tömegével adják meg, mit várnak a rendszertől. Megkönnyíti a helyzetünket, ha minél előbb az analízist olyan irányba tudjuk terelni, a fogalmi modellben pedig olyan dolgokat és szerkezeteket tudunk bevezetni, amelyeket különösebb nehézségek nélkül át tudunk ültetni az implementációs modellbe. Ha az átültetés szabályai ismertek (például hasonló feladat megoldása során már eredményesen alkalmaztuk őket), akkor munka közben el is hagyhatjuk a gyakori kitekintgetést az implementációs térre, hiszen ismert, hogy az analízis eredményeként kapott fogalmi modell különösebb gondok nélkül átvihető az implementációs térbe. Ezekben az esetekben az analízis és a tervezés között nem mindig húzható éles határvonal. A könyv tárgyát képező objektumorientált módszertan szintén ezt a gondolatmenetet követi.

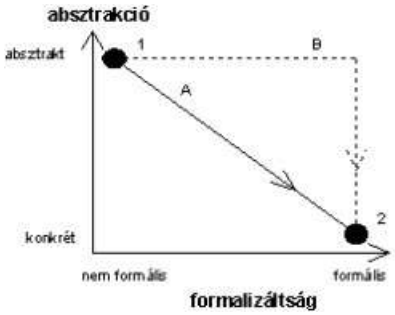


1.7. ábra

Akár a problématérben, akár az implementációs térben dolgozunk, a modellek, illetve leírásaik kialakulása egy másik szempontból is vizsgálható, az absztrakció és a formalizáltság függvényében (lásd 1.8. ábra). A fejlesztés kiinduló pontja általában az ábrán látható, (*1*) jelű pont. Ez a felhasználó által használt, igen magas absztrakciós szintű, egyáltalán nem formális leírás. Innen kell eljutnunk a (*2*) jelű pontba, ami a programszöveget jelenti, amely az implementációs térben konkrét számítástechnikai fogalmakat használ és szigorúan formalizált.

Az 1.8. ábra alapján is bevezethetünk két fontos fogalmat. A formalizáltság növelését a gyakorlatban **specifikálásnak** nevezzük. Azt a tevékenységet, amikor az absztrakciós szintet csökkentve egy magasabb absztrakciós szintű elemet alacsonyabb szintű elemek segítségével állítunk elő, **tervezésnek** hívjuk. A fogalmakat ilyen értelemben használva nem foglalkozunk azzal, hogy a tevékenység éppen melyik térben, melyik modellen zajlik.

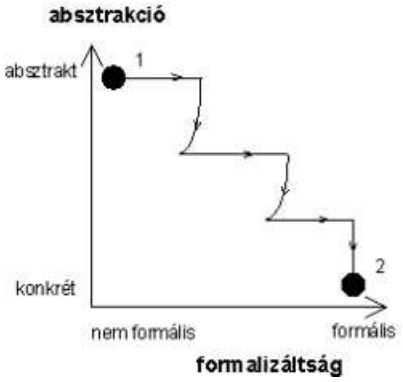
Kérdés, hogy az (*1*) jelű pontból melyik úton juthatunk el a (*2*) pontba. Az ábrán a két, nagybetűvel azonosított útvonal közül a *B* lenne az elvileg legjobb választás. Ebben az esetben a felhasználó által használt magas absztrakciós szintű fogalmak segítségével olyan szigorú, formális leírást készítünk, amely alkalmas arra, hogy jól definiált transzformációs lépéseket végrehajtva (akár számítógépes támogatással) közvetlenül a kódhoz jussunk. Tipikusan így oldunk meg feladatokat az elemi matematika és fizika köréből: Tanulmányozva a feladat szöveges (kevéssé formális) formáját felírjuk az egyenletet (szigorú formalizmus), majd az egyenleten a matematika által megengedett lépéseket végrehajtva – az egyenletben megfogalmazott állítás igazságtartalmát megőrző átalakításokkal – az egyenletet olyan alakra hozzuk, amely számunkra a megoldást jelenti (például az egyenletben szereplő ismeretlent kifejezzük). A *B* útvonalon haladva egyetlen specifikációs lépésben eljutunk oda, ahonnan már a tervezés következhet. Ez a megoldás azonban csak a triviálisan egyszerű esetekben használható, mivel jelenleg nem áll rendelkezésünkre olyan szigorú formalizmus, amellyel egy bonyolultabb fogalmi modell a maga magas absztrakciós szintű fogalmaival ésszerű terjedelemben leírható lenne.



1.8. ábra

Az *A* úton való haladás azt jelentené, hogy egyidejűleg specifikálunk és tervezünk is. Úgy tűnik azonban, hogy az emberi gondolkodás nem tud egyszerre mindkét irányba haladni, ezért a gyakorlatban a specifikációs és tervezési lépések egymást váltogatják az 1.9. ábrán látható módon. Az ábrán a specifikálást jelző vízszintes szakaszokat a tervezés követi. Az utolsó tervezési lépés (a kódkészítés) kivételével a tervezés nem kizárólag az absztrakció csökkentését jelenti, mivel a tervezés során egy magasabb absztrakciós szintű elemet konkrétabbakból építünk fel. Így az építőelemként használt konkrétabb elemek szintjén a terv kevésbé formális, további specifikálást igényel. A korábbi "szobás" példával jól megvilágítható a jelenség. Tervezzük meg, hogy egy szobában hol legyenek a nyílászárók! Legyen a tervezői döntésünk az, hogy kiválasztjuk, melyik falra kerüljön ablak! Ez az ablak még nem specifikált, nem rögzítettük a pontos helyét, a méretét, a típusát, a gyártóját stb., azaz közvetlenül a döntés után az ablakot is tartalmazó szoba összességében kevésbé specifikált, mint az ablaktalan szoba a döntés előtt.

A technológiákban megjelenő szoftverfejlesztési gyakorlat végső soron nem más, mint a fentiekben bemutatott elvi folyamatoknak irányítható, menedzselhető formája.



1.9. ábra

3. 1.3. A szoftver életciklusa

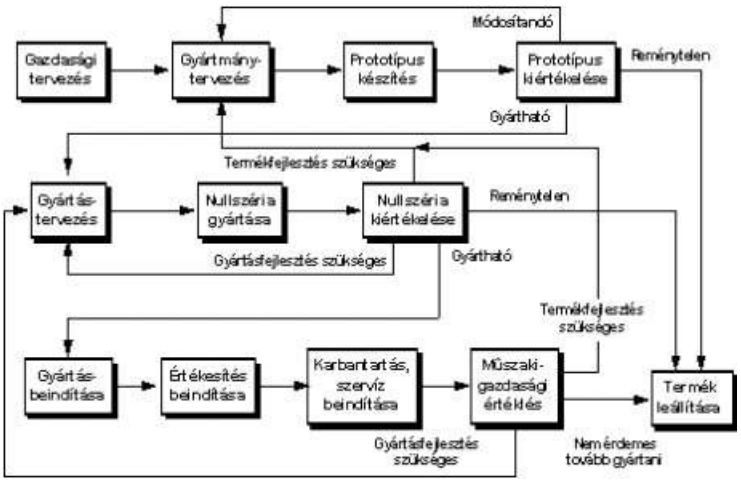
Az előző két pontban meghatároztuk a technológia és a termék fogalmát, valamint tisztáztuk a fejlesztés elvi alapjait. Jelen pontban – kiindulva a termékek szokásos életciklusából – vizsgálat alá vesszük a szoftver életciklusát és bemutatjuk annak különböző modelljeit. A fejezet végén a minőségbiztosítás néhány kérdésével is foglalkozunk.

3.1. 1.3.1. Termékek életciklusa

Mint már említettük, a termékek sorsát a rájuk való igény felmerülésétől a termék forgalomból való kivonásáig (feledésbe merüléséig) az életciklus írja le. A hagyományos (egyszerűbb, anyagi jellegű) termékek életciklusának jellegzetes szakaszai például a gyártmánytervezés, a prototípuskészítés, a gyártástervezés, a nullszéria gyártása, a gyártás, a karbantartás.

Hosszabb időn át gyártott tömegcikkek esetén mind a termék, mind az előállítás technológiája többször módosul a termékről szerzett tapasztalatok alapján, valamint a technikai fejlődés következtében. Ilyen életciklust láthatunk az 1.10. ábrán.

Egyedi, bonyolult termékek gyakori jellegzetessége, hogy az első működő példány egyben az egyetlen is. Ilyenkor a tervezés szerepe megnő, a fejlesztés költségei nem oszlanak meg a sorozatban előállított darabok között. Nem véletlen, hogy az ilyen típusú termékek megrendelői különös jelentőséget tulajdonítanak a szállító cég referenciáinak.



1.10. ábra

Az életciklus tehát magas szinten írja le egy termék kezelésének *folyamatát*, azaz a termékkel kapcsolatos teendőket és azok sorrendjét. Az életciklus fogalmának bevezetése megkönnyíti a szükséges munkák felmérését, megtervezését és kiértékelését. Tudatos feltérképezése, sőt megtervezése a termék előállításával foglalkozó cég alapvető érdeke. Megfigyelhető az a tendencia, hogy a gazdasági tervezés egyre inkább a termék teljes élettartamára kiterjedő tevékenységek költségoptimumát keresi a lokális (például a csak gyártási, vagy az anyagköltségekre vonatkozó) optimumok helyett.

Vizsgáljuk meg ezek után a szoftver azon tulajdonságait, amelyek az életciklus jellegét befolyásolják!

3.2. 1.3.2. A szoftver életciklusának jellegzetességei

A 1.2. pont bevezetésében megállapítottuk, hogy a szoftver életének súlyponti szakasza a fejlesztési, ezen belül is a termékfejlesztési szakasz. A szoftver élettörténetét leíró kezdeti modellek ezért szinte kizárólag az első működő példány elkészültéig terjedő szakaszt vették vizsgálat alá. Mára a helyzet jelentősen megváltozott. A szoftverről kiderült, hogy folyamatos változásokra hajlamos, kezeléséhez olyan életciklus-modellek szükségesek, amelyek ezt a tulajdonságát is tükrözik. Ennek ellenére az életciklus termékfejlesztési szakaszának elsődlegessége (dominanciája) ma is fennáll.

A szoftver változási hajlamát két tényező okozza. Egyrészt a változtatások technikailag könnyen kivitelezhetők. Ezért a felhasználók elvárják, hogy egyre újabb igényeiket gyorsan kövessék a szoftver újabb változatai. Másrészt a szoftvertermék általában sokkal bonyolultabb, mint a hagyományos anyagi jellegű termékek. Ezért nehéz a "nulláról induló" fejlesztés folyamán már az első példányt úgy elkészíteni, hogy az minden igényt kielégítsen. A szoftver változási hajlama ezért nemcsak azt eredményezi, hogy a nagypéldányszámú szoftverek verziói sűrűbben követik egymást, mint ahogyan az anyagi jellegű termékeknél megszoktuk, hanem azt is, hogy életük során az egyedi szoftverek is számos változtatáson, módosításon, bővítésen esnek át.

Amennyire a kivitelezés viszonylagos egyszerűsége csábít a gyakori változtatásra, annyira óvatosságra intetnek a bonyolultságból fakadó nehézségek. A tapasztalatok megmutatták, hogy egy bonyolult rendszerben minden módosítás rendkívül veszélyes. Ugyanis gyakran nemcsak a kívánt elsődleges hatást érjük el, hanem olyan kellemetlen mellékhatások is jelentkezhetnek, amelyekre nem gondolunk a módosítás megtervezésekor. A szoftverszállítók gyakorta nem adják át a megrendelőnek a szoftver forrásnyelvű reprezentációját. Ezen ténynek

csak egyik oka a szerzői jog védelme. A másik ok az, hogy a szállítók el akarják kerülni a szoftver átgondolatlan megváltoztatását.

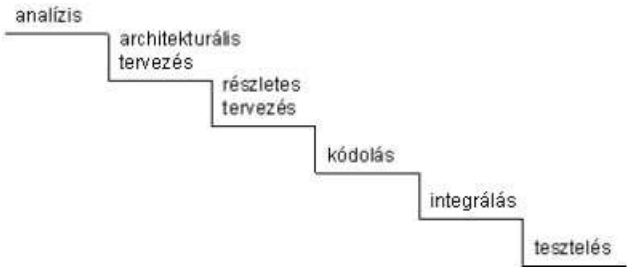
Ugyanakkor a fejlesztők elemi érdeke, hogy a felmerülő módosítási igényeket minél gyorsabban és minél biztonságosabban ki tudják elégíteni, egy-egy új igény kielégítésére pedig minél gyorsabban tudjanak – lehetőleg már meglévő komponensekből – új rendszert készíteni. Ezért lett a szoftvertechnológia két kulcsszava a módosíthatóság és az újrafelhasználhatóság.

A ma használatos, igen bonyolult szoftverek egy-egy újabb változatának előállítása igen nagy – az első verzió előállításával összemérhető – feladat. Ezért marad meg az állandó változást tükröző, ciklikus élettörténeten belül a termékfejlesztési szakasz dominanciája.

Vizsgáljuk meg, milyen jellegzetes élettörténeti sémák jöttek létre a szoftver projektek követésére! Elöljáróban is hangsúlyozzuk, hogy ezek olyan alapsémák, amelyek egy-egy konkrét projekt esetén egymással kombinálva is alkalmazhatók. Az életciklus-sémákat is modelleknek szokás nevezni. Természetesen itt nem a rendszer modelljéről, hanem a teljes projekt folyamatának modelljéről van szó.

3.3. 1.3.3. A vízesésmodell

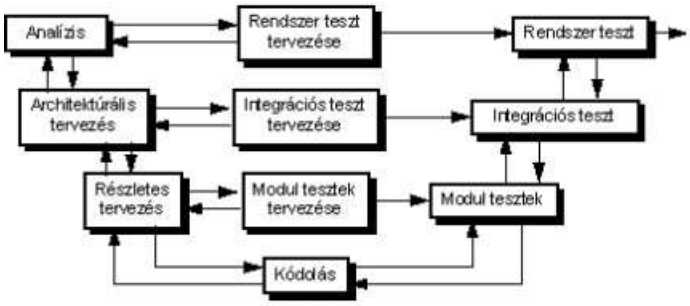
A szoftverfejlesztés problémáinak felismerése után a projekt folyamatának leírására kialakult legrégebbi modell a **vízesésmodell**, amelyet **fázismodellnek** is nevezünk. Nevét a szemléltető ábra jellegzetes alakjáról kapta (1.11. ábra). A modell a termékfejlesztésre koncentrál, azaz az első működő példány előállításáig terjedő szakasz lefolyását ábrázolja. Ma általában a változtatásokat is kezelő ciklikus modellekbe ágyazva használják, önmagában legfeljebb nagy egyedi szoftverek esetén fordul elő.



1.11. ábra

A modell a fejlesztési folyamat kiindulási pontjának a követelményeket tekinti. Az első fejlesztési ciklusban ezek a követelmények új követelményként jelennek meg. A későbbi változatoknál azonban már nagyrészt a felhasználói, üzemeltetési tapasztalatok alapján módosított korábbi követelményekből indulunk ki. A vízesésmodellben megjelenő első szakasz az analízis, amelynek eredményeként létrejön egy specifikáció. A specifikáció alapján három további lépésben hozzuk létre a szoftvert. Az elsőt architekturális, a másodikat pedig részletes tervezésnek nevezzük. A két tervezési lépés a rendszerről alkotott modellek absztrakciós szintjében különbözik. A részletes tervezés végeredményeként egymástól függetlenül, önállóan kódolható részek specifikációit állítjuk elő. A következő lépés a kódolás, amibe beleértendő az önállóan kódolt részek tesztje is. Ezen lépésen belül húzódik a tervezés és az implementáció határvonala, azaz itt térünk át a gondolati síkról a valóságba. Korábban, a fejlesztés elvi modelljének tárgyalásakor az implementációval nem foglalkoztunk. Valójában a fejlesztés igen fontos szakasza a vízesésmodellben **integrációnak** nevezett fázis, amelyben az önállóan kódolt és kipróbált részekből állítjuk össze a teljes rendszert, illetve a korábbi, meglévő kódot és a módosítások eredményeként létrehozott kódot összeillesztjük. Az integrációhoz szorosan kapcsolódik a teljes rendszerre kiterjedő ellenőrzés, tesztelés.

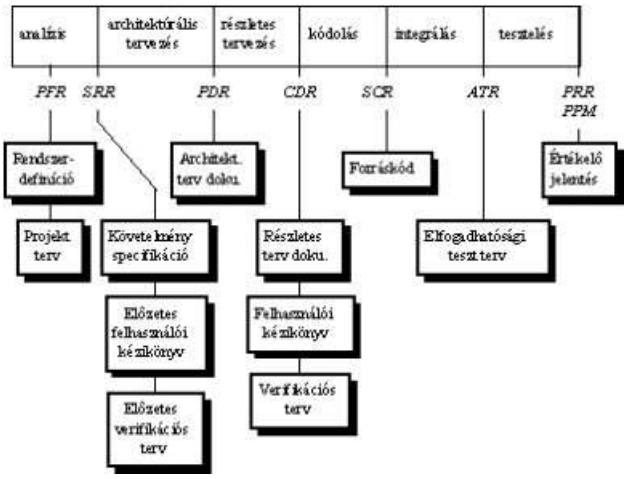
A szemléltető ábrán a lépcsőzés egyfelől a tevékenységek sorrendjét mutatja, másfelől pedig azt kívánja ábrázolni, hogy – amint a vízesésen sem tudunk felfelé haladni – a megelőző fázisokra nincs visszatérés. Valójában ilyen visszalépésekre gyakran van szükség, elsősorban akkor, ha rájövünk, hogy valamelyik korábbi fázisban hibáztunk (visszacsatolásos vízesésmodell). A visszalépés költségei annál nagyobbak, minél nagyobb a lépés, azaz minél korábbi fázistól kezdődően kell a már elvégzett munkánkat módosítani, javítani.



1.12. ábra

A vízesésmodell egy másik, továbbfejlesztett változata az úgynevezett V-modell, amelyik az implementációs és tesztelési szakaszt a *V* alakzat jobbra felfelé haladó ágára helyezi, és részletezi (lásd 1.12. ábra). Minden tervezési fázis után a fejlesztés két ágon fut tovább. Egyrészt megkezdődik a következő tervezési fázis, másrészt – lehetőleg egy független csoport munkájaként – megkezdődik a megfelelő tesztfázis tervezése.

A vízesésmodell gyakorlatban történő alkalmazásához definiálni kell, hogy melyek az egyes fázisok befejezésének kritériumai. Mikor mondhatjuk azt, hogy most elérkeztünk például az analízis fázis végére? A fázisok elhatárolására szükségünk van, hiszen ellenkező esetben kevés az esélyünk, hogy meg tudjuk válaszolni azt a kérdést, hol tartunk a fejlesztésben idő és költségek tekintetében. A fejlesztési folyamat áttekinthetetlensége esetén válik igazzá Murphy törvénye, miszerint a fejlesztés alatt álló szoftver készültségi foka bármely időpillanatban 90%.



1.13. ábra

A fázisok határait az ajánlások és a szabványok bizonyos dokumentációk (reprezentációk) meglétéhez, továbbá ezek áttekintését szolgáló összefoglaló és értékelő megbeszélések (**mérföldkövek**) megtartásához kötik. Az előírások a dokumentumok elnevezésén kívül részletesen szabályozzák azok pontos tartalomjegyzékét is. Az egyik legszigorúbb szabvány, az USA Védelmi Minisztériuma által kidolgozott DoD-2167 számú szabvány, amelyik több mint hatvan dokumentáció meglétét írja elő.

Az 1.13. ábrán a fázismodellhez kapcsolódó dokumentációk és mérföldkövek szerepelnek. Az ábrán az említett DoD-2167 szabvány egy nagyon leegyszerűsített változata látható.

Az egyes fázisok határainál és az analízis fázis alatt *DŐLT* betűkkel feltüntettük a mérföldkő elnevezésének hárombetűs kódját. Az alábbiakban összefoglaljuk az ábrán szereplő betűszavak angol és magyar értelmezését.

PFR	*Product Feasibility Review*	Megvalósíthatósági vizsgálat

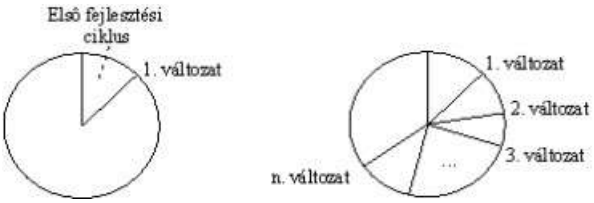
SRR	*Software Requirements Review*	Követelményelemzés
PDR	*Preliminary Design Review*	Az architektúra-tervek áttekintése
CDR	*Critical Design Review*	A részletes tervek áttekintése
SCR	*Source Code Review*	A forráskód felülvizsgálata
ATR	*Acceptance Test Review*	Az elfogadhatósági teszt áttekintése
PRR	*Product Release Review*	Forgalombahozatal előtti áttekintés
PPM	*Project Post-Mortem Review*	Projektértékelés

3.4. 1.3.4. Az inkrementális fejlesztési modell és a prototípus

A vízesésmodell igen jó szolgálatot tett annak tudatosításában, hogy a szoftverfejlesztés nem azonos a programozással. A modell így önmagában a nagy, egyedi rendszerek fejlesztési folyamatának kezelésére alkalmas. Korlátai akkor jelentkeztek, amikor kiderült, hogy a szoftver változási hajlama még az egyedi rendszerek esetén sem szorítható merev fázishatárok közé. A leggondosabb tervezés ellenére is csaknem minden rendszer átadásakor, vagy próbaüzeme során felmerülnek olyan módosítási igények, amelyek még a követelményekre is visszahatnak. Az általános célú, nagypéldányszámú szoftverek esetén a módosítási igények pedig folyamatosan jelentkeznek.

A gyakorlatban a legtöbb szoftver életének nagyobbik részét teszi ki – és a költségek nagyobb részét is igényli – az egyre újabb változatok, verziók előállítása, a módosítások végrehajtása. Ezt a tevékenységet karbantartás (*maintenance*) néven szokás említeni. Az új szoftver fejlesztése ebben a folyamatban mindössze egy speciális esetet képvisel, az első változat előállítását. Ekkor a "semmiből" hozzuk létre az első verziót, ezt követően pedig a már meglévő változatok módosításával állítjuk elő az újabbakat. Az 1.14. ábrán a szoftver teljes élettartama alatt szükséges anyagi és élőmunka ráfordításokat a kör területe jeleníti meg. Jól látható, hogy az első fejlesztésre a ráfordításoknak csak töredéke esik.

Tekintettel arra, hogy egy változat – főleg az első – önmagában is elég nagy lehet ahhoz, hogy évekig készüljön, a fejlesztés közben megszerzett ismeretek fényében szükségessé válhat a követelmények módosítása. Célszerű lehet ezért egy változatot is több lépésben, alváltozatok egymásutánjaként előállítani. Ezt a folyamatot nevezzük a fejlesztés **inkrementális modelljének.**



1.14. ábra

Az inkrementális fejlesztés során módunk van arra, hogy a rendszer problematikus részeit fejlesszük ki először. Ha ez sikerrel járt, akkor ehhez hozzáfejleszthetjük az újabb és újabb részeket. A szoftver problematikus részeinek kísérletezésre alkalmas megvalósítását nevezzük **prototípusnak.**

A tapasztalatok szerint prototípus készítésére kétféle okból lehet szükség:

- nem vagyunk biztosak abban, hogy valamilyen részproblémát adott technikai-gazdasági keretek között meg tudunk oldani,

- a felhasználó nem tudja pontosan meghatározni az elvárásait, vagy nem vagyunk biztosak benne, hogy a követelményeket azonosan értelmezzük.

A szoftverfejlesztői gyakorlatban főként az utóbbi eset gyakori. Korábban elemeztük a fejlesztők és a felhasználók információcseréjének nehézségeit. A félreértések elkerülésének legjobb módja, ha minél előbb be tudjuk mutatni, és kísérletezésre rendelkezésre tudjuk bocsátani a rendszer viselkedését és kezelési stílusát érzékeltető prototípust. Ennek érdekében leggyakrabban a felhasználói felületek készülnek el a prototípus változatban.

Az inkrementális fejlesztési folyamatban tehát – szemben a vízesésmodellel – a rendszer nem egyenletesen fejlődik teljes szélességében, hanem bizonyos részekkel előreszaladunk egészen a megvalósításig, tapasztalatokat szerzünk, és az elkészült részekhez fejlesztjük hozzá a még hiányzókat. A két modell természetesen kombinálható. Egy-egy inkrementum, vagy a prototípus fejlesztése végbemehet a vízesésmodell alapján.

3.5. 1.3.5. Spirálmodellek

A szoftver életének ciklikus jellegét, a folyamatos változásokat talán legszemléletesebben a **spirálmodell** írja le. A modell több változatban is definiált, egy változatát az 1.15. ábrán mutatjuk be.

Az életciklus a ponttal jelölt helyről indul, és négy térrészen át fut körbe. Az (*1*) jelű térrész többé-kevésbé megfeleltethető az analízis, illetve a nagyvonalú tervezés tevékenységének. Új eleme a modellnek, hogy több alternatíva tudatos felállítását javasolja, amelyek mindegyikét ki kell dolgozni olyan részletezettségig, hogy értékelésüket el lehessen végezni. Ezzel kezelhetővé válik az 1.2. ábrán bemutatott több lehetséges megfeleltetés. Ugyancsak új elem, hogy a második térrészben gazdasági szemléletű kockázatelemzés folyik. A lehetséges megoldások közül a minimális kockázattal rendelkezőt tekintjük a legkedvezőbbnek. A harmadik térrész a kiválasztott megoldás megvalósítását, azaz a részletes tervezést és az implementációt jelöli. A megvalósítás és üzemeltetés tapasztalatainak értékelése alapján dönthető el, hogy szükséges-e módosítás, továbbfejlesztés, és ha igen, ennek alapján az új célok is kitűzhetők.

1.15. ábra

A spirálmodell akár nagyobb léptékben az új verziók ciklusainak leírására, akár kisebb léptékben az inkrementális fejlesztés szakaszainak leírására egyaránt jól használható. Az egyes szakaszokban az eddig megismert modellek is alkalmazhatók.

3.6. 1.3.6. Az újrafelhasználhatóság

Valamennyi fejlesztés során természetes igény, hogy a korábbi fejlesztői tevékenység eredményeit ismételten hasznosítsuk. Az **újrafelhasználhatóság** a szoftvertechnológiák varázsszava, amelytől sorsunk jobbra fordulását és a szoftver krízis leküzdését reméljük. Más technológiákhoz képest a szoftver esetén az újrafelhasználhatóság látszólag sokkal nagyobb hangsúlyt kap. Ennek oka, hogy az ipari technológiák lényegesen kialakultabbak, kikristályosodtak meghatározott funkciójú alkatrészek, részegységek és lezajlott a szabványosodás folyamata. Ugyanakkor a szoftver területén nehezen találjuk meg (pontosabban véljük megtalálni) az ismételten használható komponenseket.

Újrafelhasználható komponenst a szoftverreprezentációk valamennyi absztrakciós szintjén találhatunk. Egy szoftverrendszer tulajdonképpen alkalmazói programok együttesének is tekinthető. A jogi formaságokat

rendezve teljes programokat is beépíthetünk saját rendszerünkbe. Megfordítva is igaz ez a gondolat. Általában jól ismert, elfogadott rendszerprogramokra (operációs rendszerek) épülnek rá a mi alkalmazásaink. A PC-s világban a programjainkat DOS vagy Windows környezetben futtatjuk. Ezt természetesnek tartjuk, és elfeledkezünk arról, hogy valójában az operációs rendszerek újrafelhasználásával állunk szemben. Ez akkor válik szembetűnővé, ha teljes rendszert kell szállítanunk, és kénytelenek vagyunk az operációs rendszer jogtiszta változatát megvásárolni.

Manapság a szabadon felhasználható programok (freeware) száma meg sem becsülhető. Ezeket készítőik azzal a szándékkal adják közre, hogy újból felhasználjuk őket. Alkalmazásukat azonban korlátozza, hogy még nem alakult ki olyan rendezett környezet, amelyben legalább áttekintést kaphatnánk a választékról. Ezen programok minősége és megbízhatósága ugyancsak nem problémamentes.

Jól ismert az újrahasznosítás fogalma az eljárások, illetve a függvények szintjén. Mi más is lehetne egy C függvény meghívása, mint a könyvtárbeli program ismételt hasznosítása? Pontosan ezért érdemes megtanulni, hogy milyen függvények is állnak a rendelkezésünkre, hiszen ezen ismeretek ismételten a hasznunkra válhatnak.

A fenti példák mindegyike a szoftver ugyanazon reprezentációjának – nevezetesen a kódnak – az újrahasznosítását volt hivatva bemutatni. A fejlesztésről kialakított képünk szerint hasznos és szükséges lehet más reprezentációk ismételt felhasználása is. Az analízis során készített dokumentációk csak akkor használhatók fel ismételten, ha az új feladat a régebbihez nagyon hasonló, akkor is inkább csak orientáló jelleggel. A tervezés közben készült anyagok még mindig túl erősen kötődnek az alkalmazáshoz, ezért ritkán újrafelhasználhatóak.

Az ismételt használat lehetőségét jelentősen növeli a konkrét alkalmazástól való függetlenség. Nem véletlen, hogy elsősorban az alkalmazástól kevéssé függő felhasználói felületek és az absztrakt adatszerkezetek (listák, fák stb.) kezelésére alakultak ki a hagyományos kódnál magasabb szintű reprezentációk újrahasznosításának technikái, amelyek általában az objektumorientáltságon alapulnak.

Mint a fentiekből kiderült, az újrahasznosíthatóság alapvető feltétele, hogy a hasznosítható komponensekről, azok paramétereiről, alkalmazási feltételeiről minden ismeret rendszerezett, könnyen hozzáférhető formában álljon rendelkezésre, és maga a komponens is egyszerű technikával legyen alkalmazható. A hardvert tervező szakemberek számára rendelkezésre állnak a különböző gyártók katalógusai, amelyekben rendben megtalálhatók a komponensekkel kapcsolatos ismeretek. A szoftverkomponensek tekintetében még nem léteznek ilyen jellegű katalógusok, többek között azért sem, mert a kész komponensekből való építkezésnek nincsenek olyan kialakult technikái, mint az áramkörök összeépítésének. Ezért a programtervező dilemma elé kerül, hogy mikor jár el gyorsabban, olcsóbban? Ha megpróbálja felderíteni, hogy van-e, és ha igen, akkor hol, mennyiért, milyen minőségben a feladatához felhasználható kész szoftverkomponens, vagy ha inkább saját maga készíti el azt?

Az újrahasznosíthatóság ebből a szempontból ma a menedzsmenttel szembeni kihívás – sikerül-e mielőbb összeállítani a hardveresekéhez hasonló katalógust. A technológusok feladata pedig a katalógusba bekerülő elemek és az építkezési technológia kidolgozása, lehetőleg egységes elvek alapján. Mai kilátásaink szerint ezen egységes elv az objektumorientáltság lesz.

3.7. 1.3.7. Minőségbiztosítás a szoftverfejlesztésben

Az utóbbi évtizedben a termékek minőségbiztosításának problémája Európaszerte központi kérdéssé vált. (A folyamat az Egyesült Államokban és Japánban már valamivel korábban elkezdődött.) Az **ISO 9000** sorozatú szabványok megjelenése és egyre szélesebb körű elfogadása minden termékelőállítással foglalkozó vállalkozást arra késztet, hogy a termékelőállítás vállalkozáson belüli folyamatát vizsgálat alá vegye, értékelje és elfogadtassa [ISO87]. Jelentősebb rendelésekre egy vállalkozás ugyanis egyre inkább csak akkor számíthat, ha megszerezte valamely neves minősítő szervezet tanúsítványát, amely igazolja, hogy a termékelőállítási folyamat a vállalkozáson belül a szabvány követelményeinek milyen mértékben felel meg. Nem kivétel ez alól a szoftver mint termék sem, bár sajátos jellegénél fogva a szoftver minőségének megragadása, számszerűsítése és mérése nem könnyű feladat.

Valamely termék minőségét végső soron az jellemzi, hogy mennyire teljesíti a felhasználó elvárásait. Ez egy konkrét példányról utólag (kidobásakor) már viszonylag egyszerűen eldönthető, azonban a felhasználót inkább a vásárláskor adható előzetes becslések érdeklik. A termék vásárláskor történő kipróbálása javítja az esélyeket, de – különösen a bonyolult, hosszú élettartamú termékek esetén – a teljes használati időre még mindig sok bizonytalanság marad. Tapasztalatok szerint bizonyos gyártók termékeiben jobban megbízhatunk, mint másokéban. Sok felhasználó hosszabb időn át összegyűjtött tapasztalatai alapján a megbízhatónak tartott gyártók

termékeiben csupán elvétve találkozunk rejtett hibákkal. Ha mégis hibát találunk, kiterjedt, készséges szervizszolgálat segít át a problémán, a használat közben felmerülő újabb igényeinket egyszerű bővítésekkel kielégíthetjük, stb.

Mitől alakulnak ki ezek a különbségek az egyes gyártók termékei között még hasonló alapanyagok, hasonló technológiák alkalmazása esetén is?

A szabványok kidolgozását megelőző hosszas vizsgálódások eredményei szerint a termékek minősége "utólag" nem javítható. A termékek előállításának teljes folyamata, sőt a fejlesztési ciklusok egymásra építésének módja, azaz a teljes életciklus minden lépése, befolyásolja a termék minőségét.

A minőségbiztosítás lényeges elemei például

- a minőségellenőrzés minden fázisban, a beérkező alapanyagok és részegységek vizsgálatától kezdődően a kibocsátásig;

- az eredmények dokumentált megőrzése, rendszeres kiértékelése és ennek alapján a gyenge pontok felderítése;

- a karbantartási- és szerviztevékenység tapasztalatainak rendszeres kiértékelése és figyelembevétele a fejlesztési célok kitűzésekor;

- a fenti tevékenységek szervezeti, jogi kereteinek kialakítása.

Ezért a minőségbiztosítási szabványok célkitűzése az, hogy a termékek kezelésének teljes folyamatát és a gyártók teljes működési rendjét veszik vizsgálat alá, és ennek alapján minősítik a *gyártókat*.

A szoftver termékek tekintetében Európában általában az **ISO 9000-3** ajánlásra hivatkoznak [ISO91], míg az Egyesült Államok területén a Carnegie Mellon Egyetem Szoftvertechnológiai Intézete (Software Engineering Institute, *SEI*) által kidolgozott minősítési rendszer (**Capability Maturity Model,** *CMM*) a leginkább elfogadott [Hum1].

Az ISO 9000-3 elsődlegesen egy ellenőrző listaként használható, amelyik összefoglalja, hogy milyen problémákkal kell foglalkozni a minőségbiztosítás keretén belül. Három csoportban tárgyalja a minőségügyi rendszer elemeit:

- a *keretek* (alapdefiníciók, szervezeti formák, felelősségi körök),

- *az életciklus tevékenységei* (azok a tevékenységek, amelyek egy-egy projekthez kötődnek, például a követelményspecifikáció, a fejlesztési folyamat megtervezése, a minőségtervezés, a tervezés és az implementáció stb.)

- a *kiegészítő tevékenységek*, amelyek célja az életciklus egyes fázisainak hatékonyabb végrehajtása (például dokumentációk ellenőrzése, adatgyűjtés a minőségi paraméterekhez, mértékek definiálása, mérés, fejlesztőeszközök beszerzése, készítése, értékelése, oktatás stb.).

A CMM modell a szervezetben folyó tevékenység elemzéséhez ad szempontrendszert, amelynek kiértékelése alapján a szervezetben zajló munkafolyamat öt kategória valamelyikébe sorolható:

– *a kezdetleges,*

– *a megismételhető,*

– *a jól meghatározott,*

– *a szervezett,*

– *az optimalizált*.

Az utolsó két kategória már számszerűsített jellemzőket is megkövetel a munkafolyamat, illetve a termékminőség értékelésére. Az *optimalizált* kategória elérésének feltétele pedig az, hogy legyen kialakult módszer a többféle technológia közötti rugalmas választásra, sőt magának a munkafolyamatnak megváltoztatására és a projekthez történő igazítására is.

A fenti szabványok és ajánlások nem kötik meg az alkalmazható fejlesztési módszertanokat. Rögzítik azonban, hogy kellően definiált, kezelhető módszertanokat kell alkalmazni. Így az objektumorientált módszertanoknak is csak kellően kidolgozott, menedzselhető, mérhető jellemzőkkel igazolt formái számíthatnak sikerre.

2. fejezet - 2. Az objektumorientáltság fogalma

1. 2.1. Út az objektumig

A szoftverfejlesztés elvi alapjainak tárgyalásakor megmutattuk, hogy a tervezés folyamán sorozatos döntésekkel konkretizáljuk az éppen tervezendő dolgot. A döntést a választott domináns fogalomnak, vagy strukturáló objektumnak megfelelően hozzuk meg. Jelen fejezetben azt kívánjuk bemutatni, hogy a megbízható, korrekt szoftverkészítésre való törekvés eredményeként az idők folyamán miként változott a domináns fogalom, és ez miként vezetett el – szinte szükségszerűen – az objektumorientáltság kialakulásához.

1.1. 2.1.1. A kezdetektől az absztrakt adatstruktúrákig

A szoftverfejlesztés kezdeteit – talán pontosabb programkészítést említeni – a strukturálatlanság jellemezte. Az általában nagyon alacsony szintű (többnyire assembly esetleg valamilyen autokód) nyelveken írt programok nélkülözték a tervezési meggondolásokat, a döntések nem tudatosan választott strukturáló objektumokon alapultak. A program írója a megoldandó feladatot egy bonyolult adatmanipulációnak tekintette. A program ezt tükrözte is, minden utasítás egy újabb adatmanipulációt végzett el. A programozó gondolatait az a szemlélet hatotta át, hogy a feladat megoldásához még mely adatokon milyen változtatást kell végrehajtani. A változókat programozás közben szükség szerint vettük fel, esetenként a programkód közepébe beleírva. A vezérlő szerkezetek még nem tisztultak le. Az alkalmazásukkal kapcsolatosan olyan közszájon forgó szabályok alakultak ki, hogy például ciklus belsejébe nem illik beugrani. Ezen a helyzeten az első magas szintű programozási nyelvek (FORTRAN, ALGOL) megjelenése sem sokat változtatott. Amint a megoldandó feladatok bonyolultabbá váltak, egyre inkább csak kivételes képességű, "zsenigyanús" programozók voltak képesek eligazodni a kialakuló szövevényes programszerkezetekben, ők is elsősorban csak a sajátjukban. Egy-egy meghatározó egyéniség kiválása a munkából akár a projekt kudarcát is okozhatta, nem is szólva arról, hogy a program megírása és belövése után a legkisebb módosítás is gyakorlatilag áttekinthetetlen következményekkel járt. A szaporodó kedvezőtlen tapasztalatok nyomán a programozók és tervezők kezdtek *szoftverkrízisről* beszélni.

1.1.1. 2.1.1.1 Strukturált programozás

A krízisből való kilábalásra tett első tudatos lépések egyike – talán a legjelentősebbnek is nevezhetjük – a **strukturált programozás** módszerének megfogalmazása volt.

A módszert elméletileg megalapozó E. W. Dijkstra elmélete szerint [Dij1] elkészítendő programunkat elgondolhatjuk egy olyan absztrakt gépnek – nevezzük *A*-nak –, amelynek egyetlen utasítása van: *"oldd meg a feladatot"*. Ez az utasítás pontosan azt teszi, amit a programunktól elvárunk. Mivel nincs ilyen gépünk, kénytelenek vagyunk egy kevésbé okos absztrakt gép által nyújtott szolgáltatásokból előállítani a feladat megoldását. Definiálunk hát egy egyszerűbb *B* absztrakt gépet (meghatározzuk az utasításkészletét), és ezzel az utasításkészlettel elkészítjük az előző (*A*) gépet "szimuláló" programot, azaz a feladat megoldását. Ezután újra feltehetjük a kérdést, van-e ilyen *B* gépünk. Ha nincs, az eljárást meg kell ismételni, azaz egy még egyszerűbb *C* absztrakt géppel és egy programmal most a *B* gépet kell szimulálni. Az eljárást tovább folytathatjuk, szükség szerint újabb, egyre egyszerűbb *D, E, F* stb. gépeket definiálhatunk. Akkor vagyunk készen, ha olyan utasításkészlethez jutunk, amelyik már létezik egy konkrét gépen, illetve egy olyan programozási nyelvben, amelynek fordítóprogramja rendelkezésünkre áll.

A gondolatmenetet a 2.1. ábrán szemléltetjük. Láthatóan absztrakt gépek hierarchiáját hoztuk létre, ahol a hierarchia szintjeit **rétegnek** nevezzük. Bármely két réteg határán lefelé tekintve egy absztrakt gépet, felfelé tekintve pedig egy erre írott programot látunk. Minden réteg csak a közvetlen alatta elhelyezkedő réteget, mint "utasításkészletet" látja, így független annak további finomításától, azaz az alsóbb rétegek kialakításától.

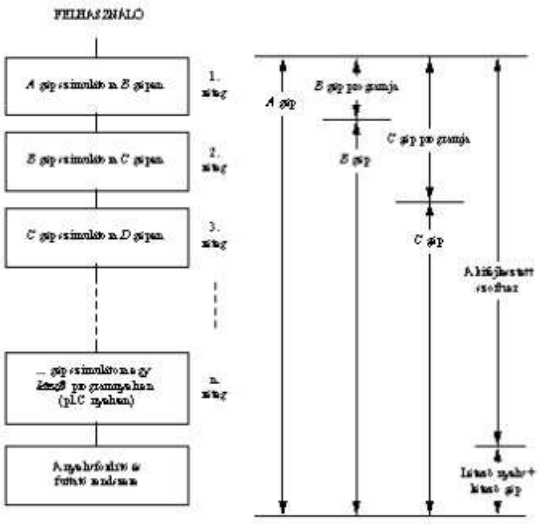
A gondolatot kiegészíthetjük azzal, hogy egy adott szint utasításait nem kell feltétlen egyetlen absztrakt géphez tartozónak tekintenünk, hanem egymással együttműködő gépeket is elképzelhetünk. Ezek a gépek a továbbiakban egymástól függetlenül finomíthatók, amennyiben a köztük lévő kapcsolatot pontosan definiáltuk.

A strukturált programozás gondolatmenetének következetes alkalmazásától az alábbi előnyök várhatók:

− A kialakuló programszerkezet áttekinthető lesz, hiszen egy-egy réteg önmagában vizsgálható, anélkül, hogy a teljes problémát és a teljes programot szem előtt kellene tartani.

− Az egyes finomítási lépésekhez (azaz az egyes rétegekhez) különálló tervezői döntés tartozik. Ez egyrészt a döntés hatását csak a következő rétegre korlátozza, másrészt – ha a döntés jól dokumentált – megkönnyíti azon pontok megtalálását a programban, amelyeket egy-egy módosítás során meg kell változtatni.

− A rétegszerkezet megteremti a hordozhatóság lehetőségét. Ha bármely két réteg határán szétválasztjuk a teljes programot, akkor a felső részt átvihetjük egy másik rendszerre, amennyiben a lefelé eső részt, mint absztrakt gépet, ott valahogyan létre tudjuk hozni.



2.1. ábra

Az előnyök mellett a módszer néhány problémája is hamarosan tudatosult. *Az egyik* alapvető gondot az okozta, hogy a rétegek kialakítása önmagában nem oldotta meg az áttekinthetőség problémáját, hiszen néhány lépés után a kialakuló utasításkészlet már igen bonyolulttá vált. Ilyenkor a több együttműködő gépre való bontás segített, de ez egyben újabb problémákat is felvetett. Az együttműködő gépek kapcsolatát ugyanis pontosan definiálni kellett. Ezt megtehettük a közösen használt *adattér* (globális változók) nagyon pontos – az aktuális réteg absztrakciós szintjénél sokkal konkrétabb – definíciójával, vagy az egyes gépek mások (többiek) által használható *eljárásainak* pontos – ugyancsak az egyébként aktuális absztrakciós szintnél lényegesen konkrétabb – specifikációjával. Ha egy adott rétegben több gépet definiálunk, akkor egy bonyolultsági fok fölött nehezen követhető, hogy a fölötte lévő réteg mely funkciói melyik gép utasításait használják. Ennek ismerete főként a módosítások végrehajtásához kellett, annak követéséhez, hogy egy módosítás a teljes program mely részeire lesz hatással. Ezen problémák megoldására a strukturált programozás egy végletekig letisztított elvi modelljét alakították ki, amely szerint valamely réteg által képviselt virtuális gép valamennyi utasítását tekintsük a következő, eggyel alacsonyabb szinten különálló gépnek. Ez a tiszta modell a rendszer funkcióit (eljárásait) egy fa struktúrába rendezi.

A másik fő probléma a hatékonyság és a strukturáltság ellentmondásában rejlik. Egy program hatékonyságát – meglehetősen szűk értelemben – jellemezhetjük az általa megoldott feladat és a felhasznált erőforrások (tárigény, processzoridő stb.) viszonyával. A szigorú rétegszerkezet (minden réteg csak a közvetlenül alatta elhelyezkedő réteg "utasításkészletét" használhatja) már önmagában is komoly hatékonysági problémákat vet fel. Gyakran előfordul ugyanis, hogy egy magasabb szintű rétegben olyan egyszerű műveletekre is szükség lenne, amelyekre az alacsonyabb szintek is számítanak, tehát az csak több réteggel lejjebb válik "elemi utasítássá". Ha ilyenkor következetesen betartjuk a strukturált programozás elveit, akár több rétegen keresztül is csupán közvetítő szerepet betöltő, önálló érdemi funkcionalitással nem rendelkező eljárásokat cipelünk magunkkal. Ha engedményeket teszünk, és az alacsonyabb szintek eljárásainak használatát is megengedjük, elveszítjük a "bármelyik réteghatár mentén szétszedhetjük" elv rugalmasságát és áttekinthetőségét.

További hatékonysági probléma merül fel a rétegenkénti több gépre bontás során, különösen a tiszta fa struktúrájú eljárásrendszer kialakítása esetén. Programunk ugyanis általában egy (néha több, egymáshoz hasonló) valóságos processzoron fog futni, ahol minden információ (adat és utasítás) azonos közegen, azonos formában tárolódik. A gépközeli szinteken tehát bármelyik absztrakt utasítás hasonló elemekből kell, hogy építkezzen. Így a fa levelei felé haladva a különböző ágakon elhelyezkedő eljárások egyre nagyobb valószínűséggel használnak közös műveleteket. Sőt, a tárigény csökkentése érdekében törekedünk is a közös műveletek kialakítására, hiszen ezeket elegendő egyetlen példányban megvalósítani. Ebben a pillanatban azonban a fa már nem fa többé, hiszen több ágnak van közös levele.

A strukturált programozás elvei alapján kialakult programfejlesztési gyakorlat megtalálta az ésszerű kompromisszumokat a fenti problémák kezelésére. A megoldás során a fogalmi modell és az implementációs eszközök közötti "rést" egyrészt felülről lefelé haladva a strukturált programozás elvei szerint, másrészt alulról felfelé haladva, az implementációs eszközök fölé húzott "simító" rétegekkel lehetett áthidalni. Ez a megközelítés már nem zárta ki, hogy a magasabb szintű rétegek közös műveleteket használjanak.

Mai szemléletünk igen sok elemét a strukturált programozás gondolatköréből örököltük, bár a gyökereket a problémamegoldás elméletével foglalkozó korábbi munkákban is megtaláljuk (például [Pol1]).

Az egyik legfontosabb ilyen gondolat a bonyolultság kezelése az **absztrakció** és a **dekompozíció** (részekre bontás) alkalmazásával. Mind az absztrakció, mind a dekompozíció azt a célt szolgálja, hogy figyelmünket koncentrálni tudjuk, csökkentve a tervezés során egyidejűleg fejben tartandó információ mennyiségét. Az absztrakció ezt a részletek eltakarásával, a dekompozíció pedig a probléma egymástól függetlenül kezelhető, egyszerűbb részekre bontásával éri el.

Figyeljük meg, hogy a strukturált programozás módszerének alkalmazásakor egyrészt igen magas szintű absztrakciót alkalmazunk: a legmagasabb szintű gép a feladat valamennyi részletét elfedi. Ezután fokozatosan tárjuk fel a részleteket. Úgy, hogy *mindig csak az éppen aktuális réteg feladataival kell foglalkoznunk*, vagyis azzal, hogy a következő, alsóbb réteg utasításaival hogyan szimuláljuk az aktuális rétegnek megfelelő virtuális gépet. Az absztrakciós szint csökkentésével feltáruló részletek áttekintésének terhét tehát a dekompozíció egy formájával, a rétegekre bontással enyhítjük. Ha egy-egy szintet még így is túl bonyolultnak találunk, akkor egyetlen rétegen belül is alkalmazhatjuk a dekompozíciót, azaz együttműködő gépekre bonthatjuk a réteg által reprezentált gépet.

A dekompozíciót úgy is felfoghatjuk, hogy elkerítjük, leválasztjuk a rendszer (feladat, probléma) egy részét, és definiáljuk, mi látható belőle, hogyan viselkedik a külvilág (a rendszer többi része) felé. Egyszersmind el is rejtjük a belsejét, hiszen azt a továbbiakban a rendszer többi részétől függetlenül kívánjuk finomítani.

Az ismertetett gondolatmenettel a strukturált programozás módszere fogalmazta meg először a szoftverfejlesztéssel kapcsolatosan a *felülről lefelé történő tervezés, a lépésenkénti finomítás* és az *információelrejtés* elvét. A kezdetben strukturált *programozásnak* nevezett irányzat alapján – a "nagybani programozás" jelentőségének felismerését és előtérbe kerülését követően – kialakult a strukturált *rendszertervezési módszertan* is (például [DeM78]). A strukturált módszertanokban a tervezői döntések domináns fogalma a *funkcionalitás*, illetve az azt reprezentáló *eljárás*. Az eljárás azonban már nem csupán a kód egyszerűsítésére és az ismétlések kiküszöbölésére szolgált, hanem a műveletek absztrakciójaként elsősorban a megértés eszközévé vált.

Jóllehet a strukturált programozás központi fogalma az eljárás, azt is felismerték a tervezők, hogy az adatok absztrakciójára is szükség van. Az *"oldd meg a feladatot"* utasítás nyilván bonyolult manipulációkat hajthat végre egy bonyolult adattéren, aminek a részleteit éppen úgy nem vizsgáljuk, mint ahogyan az eljárás részleteit sem. Dijkstra bemutatja [Dij1], hogy amikor konkrétabbá tesszük az adatteret (például döntünk arról, milyen típusú és szerkezetű változókat használunk valamilyen információ tárolására), akkor az kihatással van minden olyan eljárásra, amelyik használni akarja ezt az információt. Egy konkrétabb virtuális gép utasításkészlete úgy is kialakulhat, hogy valamilyen adatabsztrakciót teszünk konkrétabbá, és ezen konkretizálás hatását érvényesítjük az adatot használó eljárásokban. Az eljárások mellett, amelyek a műveletek absztrakciójának kitűnő eszközei, az adatabsztrakció jelölésére is leíró eszközök bevezetése lett szükséges.

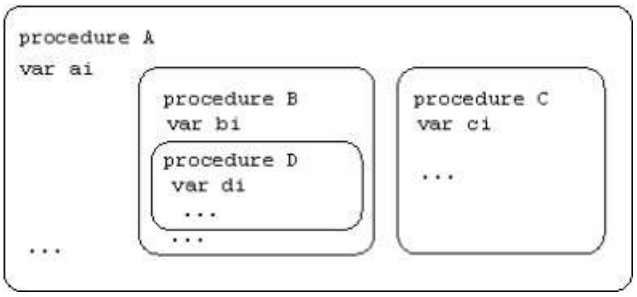
A programozási módszertannal kapcsolatos elméleti eredmények a nyelveken is éreztették hatásukat. A strukturált programozás elveit – az ALGOL-ra emlékeztető szigorú blokkszerkezettel és láthatósági szabályokkal tetézve – leginkább a PASCAL nyelv testesíti meg. A PASCAL program adatok és eljárások szigorú hierarchiába rendezett összessége. A nyelv blokkos szerkezete nemcsak az észszerű memóriakihasználást, hanem a tervezési folyamatban kialakult hierarchikus modell leírását is szolgálja, ezzel az

információ elrejtésének eszköze. A PASCAL nyelv az adatszerkezetek leírása területén is jelentős újdonságokat hozott. Gazdag szerkezetépítő és rugalmas típusdefiníciós lehetőségeket nyújt, amivel az adatabsztrakció kifejezését is segíti.

Az egymásba ágyazott blokkok struktúrája oly módon szabályozza a láthatóságot, hogy egy blokkból nézve látható valamennyi, az adott blokkban deklarált változó és eljárás fejléce, valamint a blokkot közvetlenül magában foglaló külső blokkban deklarált eljárások fejléce (sorrend miatt esetleg forward deklaráció szükséges) és valamennyi, külső blokkban deklarált változó.

A 2.2. ábránkon a B eljárást tartalmazó blokkból körülnézve látható a bi változó a B, a C és a D eljárások fejléce, valamint az ai változó.

A blokk értelmezéséhez az is hozzátartozik, hogy a blokkban deklarált változók élettartama csak a blokkban levő program futásának idejére korlátozódik. Ebből következik, hogy az eljárás befejeződése után az eljárásban deklarált változók elvesztik az értéküket. Az eljárás meghívásakor a változók értéke definiálatlan.



2.2. ábra

A blokk megvalósítása megfelel a szigorúan skatulyázott, strukturált absztrakt gép modelljének, de emellett komoly hátrányai is vannak. Annak következménye, hogy a változók élettartamát az őket tartalmazó blokk futási idejére korlátozzuk az, hogy minden olyan változót, amelynek értékére a blokkba történő ismételt belépéskor számítunk (ún. **statikus változók**), a blokkon kívül kell deklarálni. Ezáltal az ilyen változók a külső blokk számára láthatóvá válnak.

A skatulyázás támogatja az eljárások fastruktúrában való elrendeződését. Ha ugyanazt az eljárást több, különböző blokkokban deklarált eljárásból is hívni akarjuk – azaz a fastruktúrát hálóssá kívánjuk tenni – akkor az ismertetett láthatósági szabályok miatt az eljárást többszörözni kell. Ugyanazon eljárás több példányban történő létezése esetén a példányok közötti azonosság egy körültekintés nélkül – nem az összes példányon – elvégzett módosítással könnyen megsérthető. Ez a hatékonyság romlásán túl még inkonzisztenciához (következetlenség) is vezet.

A skatulyázott blokkstruktúra megnehezíti a program önállóan fordítható részekre bontását (szegmentálását), hiszen a fordításkor valamennyi külső blokkban deklarált változót ismerni kell. Nagyobb programok, bonyolultabb feladatok esetén ez bizony komoly hátrány. Nem véletlen, hogy a szoftverkrízis leküzdésének egy másik jellegzetes áramlata pontosan a feladat külön kezelhető (önállóan fordítható és tesztelhető) részekre bontását, azaz a dekompozíciót helyezte a középpontba. Ez az áramlat a **moduláris programozás** volt.

1.1.2. 2.1.1.2. Moduláris programozás

A moduláris programozás központi fogalma a **modul**, amely a rendszer valamilyen, a fejlesztés során önálló egységként kezelt, cserélhető részét jelöli. A szoftverfejlesztésben a modul a program valamely részét tartalmazó forrásnyelvű fájl, amely önállóan lefordítható és tesztelhető. A fordítóprogram előállítja a modul úgynevezett tárgykódját, ami fizikai megjelenését tekintve ugyancsak egy fájl, valamint egy fordítási listát. A modulok tárgykódját tartalmazó fájlokból a szerkesztő program (*linker*) állítja össze a teljes rendszert.

A modul cserélhető jellegéből következően jól meghatározott csatlakozó felületen (*interfészen*) keresztül kapcsolódik a környezetéhez. A hardver világában az interfész csatlakozó szerelvényeket jelent, amelyeken meghatározott bemeneti és kimeneti jelek haladnak át. A szoftver világában az interfésznek a modul által definiált, nyilvánossá (láthatóvá) tett azonosítók, és a modul által feltételezett, a külvilágban létező azonosítók felelnek meg. Az azonosítók a modulon belül bármit jelölhetnek (hívható eljárásokat, változókat, konstansokat

stb.). Valamennyi modul egyenrangú abban a tekintetben, hogy a modul maga szabja meg, hogy a benne definiált változók és függvények közül melyek lesznek kívülről láthatók. Így a modul az információ elrejtésének kitűnő eszköze.

A moduláris programfejlesztésre alkalmas programozási nyelvek közül a legismertebbek a FORTRAN, a C és a koncepciót talán a legkövetkezetesebben támogató MODULA-2. A nyelvi megvalósításokban a modul jelentősen különbözik a blokktól, hiszen a modul jellegénél fogva statikus. Így a statikus változók használata ezekben a nyelvekben természetes dolog.

A moduláris programozás kezdeti időszakában a program részekre bontása egyszerűen a méret alapján történt. Ha a program kezdett túlságosan nagy lenni, a javítások utáni újrafordítások egyre több időt emésztettek fel. Ilyenkor a programot egyszerűen két részre osztották, mindegyikben láthatóvá téve a benne deklarált azonosítókat. Később, amint a programot alkotó modulok száma kezdett növekedni és a kapcsolatokat realizáló azonosítók kezelése, áttekintése nehézségeket okozott, vizsgálat alá vették, hogy mit célszerű egy modulba összefogni. Az eredmény általános megfogalmazása: *a modul belső kötése (kohéziója) legyen minél erősebb, a modulok közötti csatolás pedig minél gyengébb*. Azaz, ami a feladatban összetartozik, az kerüljön egy modulba, a külön modulba kerülő részek pedig legyenek minél függetlenebbek.

Ennek az általános megfogalmazásnak elméletileg tiszta megfelelőjét az *egyetlen adatszerkezeten egyetlen funkciót* megvalósító modulban vélték megtalálni. Hasonlóan a strukturált programozás tiszta fastruktúrájához, ez a "steril" elv is számos problémát vetett fel. Először is az így keletkező modulok elég kicsik. Másrészt, ha a modul által elrejtett adatszerkezetből több példány kell, akkor az egész modult több példányban kell elkészíteni. Harmadrészt – és talán ez a leglényegesebb, mivel egy adatszerkezeten általában többféle műveletet is el kell végezni – az adatszerkezeteket mégiscsak láthatóvá kell tenni a többi művelet, azaz más modulok számára. Ebben a pillanatban azonban az adatszerkezetet definiáló modul és az abba belelátó modul függetlensége megszűnik.

1.1.3. 2.1.1.3. Absztrakt adatszerkezetek

Az *"egy adatszerkezeten egy funkció"* alapú modul koncepcióját az **absztrakt adatstruktúrák** megjelenésével az *"egy adaton több funkció"* elve váltotta föl, azaz egyetlen egységbe fogtak össze egy adatszerkezetet és a rajta végrehajtandó műveleteket. Az így kialakuló modulból csak a műveleteket realizáló eljárásokat és függvényeket tették láthatóvá. Így a modul egy adatabsztrakciót testesít meg: anélkül, hogy a névvel (a modul nevével) jelölt adatszerkezet bármilyen részletét ismernénk, műveleteket tudunk végezni rajta.

Az absztrakt adatstruktúrák használata a funkcionalitásra épülő korábbi tervezési elvek radikális megváltoztatását jelentette, ugyanakkor a strukturált és a moduláris programozási áramlat tanulságait egyesítette. Az absztrakt adatstruktúrákra alapozott tervezés domináns fogalma az adatszerkezet. Az adatstruktúra egy adott absztrakciós szinten tökéletesen leírható a rajta értelmezett műveletekkel, függetlenül attól, hogy a struktúrát hogyan valósítjuk meg (*implementáljuk*). Absztrakt adatstruktúrák leírására jól használható az algebra, mint azt a következő példa is igazolja.

Írjuk le absztrakt adatstruktúraként a jól ismert vermet (*stack*)! A megszokott veremmutató mozgatására való hivatkozás helyett leírjuk azokat a műveleteket, amelyeket a vermen szoktunk végrehajtani. Elsőként precízen megadjuk a műveletek szintaktikáját, vagyis azt, hogy mely művelet milyen bemenetekből (a művelet értelmezési tartománya) milyen kimenetet (értékkészlet) állít elő. Azt sem kívánjuk definiálni, hogy milyen elemeket akarunk a veremben tartani, végül is ez nem tartozik a verem működésének lényegéhez. Itt a veremben tárolt "valamiket" nevezzük *elem*-nek. Ne foglalkozzunk a rendelkezésre álló tároló véges méretével sem!

Művelet	**Értelmezési tartomány.**	**Értékkészlet**
NEW	( )	verem
PUSH	(verem, elem)	verem
POP	(verem)	verem
TOP	(verem)	elem

EMPTY	(verem)	boolean

A műveletek értelmezése legyen a következő. A NEW-val egy új, üres vermet hozunk létre. A PUSH-sal egy elemet helyezünk a verembe. A POP-pal a veremről levesszük az utoljára behelyezett elemet, eredménye egy módosult verem, a levett elem elvész. Nem üres verem esetén a TOP megmondja, hogy mi volt az utoljára behelyezett elem, anélkül, hogy a vermet megváltoztatná. Az EMPTY legyen igaz, ha a verem üres, és hamis, ha nem az.

A fent említett szemantikát leírhatjuk olyan algebrai formulákkal, ahol a műveletek egymás utáni végrehajtásának eredményét rögzítjük. A kisbetűvel szedett *verem* és *elem* konkrét vermet és elemet jelölnek.

1. EMPTY (NEW ( )) = igaz
2. TOP (NEW ( )) = nem definiált
3. POP (NEW ( )) = NEW ( )
4. EMPTY (PUSH (verem, elem)) = hamis
5. TOP (PUSH (verem, elem)) = elem
6. POP (PUSH (verem, elem)) = verem

Az axiómák tartalma szövegesen a következőképp fogalmazható meg:

1. Az új verem üres.
2. Az új veremben nincsen egyetlen elem sem.
3. Az üres veremnél nincsen üresebb.
4. A verem csakis és kizárólag PUSH művelettel tölthető.
5. A TOP az utoljára behelyezett elemet mutatja meg.
6. A PUSH a korábban a verembe rakott elemeket és azok sorrendjét nem változtatja.

Mint látható, az algebrai definícióban nincs utalás a szerkezet implementálására vonatkozóan. A fenti műveletek a verem lényegét jelentik és egyaránt leírják a számítógép stackjét, de ugyanígy a bőröndöt vagy a zsákot is.

Az absztrakt adatszerkezet, mint domináns fogalom alkalmazásával ugyanazon feladatra más, sokszor világosabb, áttekinthetőbb megoldást kaphatunk, mint a funkcionális megközelítéssel. Az alábbi, R. Mitchelltől származó példa [Mit92] ezt kívánja bizonyítani.

1.2. 2.1.2. Funkcionális kontra adatorientált tervezés

Készítsük el a következő feladatot megoldó program vázlatát!

Egy bemeneti szövegfájl szavainak abc-sorrendjében nyomtassuk ki, hogy a szavak mely sorokban fordulnak elő.

Ha a bemenet:

```
Ez a sor az első sor,
Ez a sor pedig a második sor.
```

akkor a kimeneten az alábbi sorok jelenjenek meg:

a	1 2 2
az	1
első	1
Ez	1 2
második	2
pedig	2
sor	1 1 2 2

A funkcionális megoldás

A feladatot végiggondolva megállapíthatjuk, hogy a kimenetet csak akkor kezdhetjük kinyomtatni, ha már a teljes bemenő fájlt végigolvastuk, hiszen a bemenő fájl utolsó sorában állhat az a szó, amelyik a kimenet első sorába kerül. Következésképp a funkciókra koncentrálva a feladat három lépésben oldható meg. Elsőként beolvassuk a fájlt, és annak alapján egy táblázatot építünk fel. Ezt követően a táblázatot abc-szerint rendezzük. Utolsó lépésben kinyomtatjuk a rendezett táblázatot. Egy C-hez hasonló pszeudonyelven a programunk a következőképp nézne ki:

```
main( ) {
  build(infile, table);
  sort(table);
  print(outfile, table);
}
```

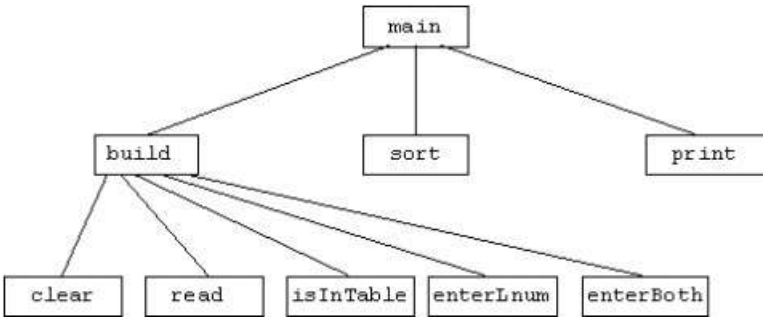
A három funkcióból az elsőt választjuk további finomításra. A build függvény – amelynek feladata a bemeneti fájlból a táblázat elkészítése – strukturált szöveggel történő leírása az alábbi:

```
if (infile üres) then return (empty table)
else
   for (minden különböző szóra)
   (a table-ba egy bejegyzést készíteni, amelyben tároljuk a szót és az
    előfordulásaihoz tartozó sorszámokat).
```

Ugyanez pszeudonyelven leírva:

```
void build (infile_type& infile, table_type& table) {
   int linenumber = 0;
   clear(table);
   while (!eof(infile)) {
      linenumber++;
      while (true) {
         if (read(infile, word) == EOL) break;
         else
         if (isInTable(word, table)
            enterLnum(word,linenumber,table);
         else enterBoth(word, linenumber, table);
      }
   }
}
```

Az eddig elkészült program struktúraábráját láthatjuk a 2.3. ábrán. A struktúraábrában azt jelöljük, hogy egy függvény mely más függvényeket hív meg.



2.3. ábra

A funkcionalitásra koncentrálva, az algoritmusokat elrejtettük, ugyanakkor döntéseket hoztunk az adatstruktúrákat érintően. Példának okáért úgy döntöttünk, hogy a table táblázatban a különböző szavakat csak egyszer tároljuk. Ezt a döntésünket a build függvény tervezése kapcsán hoztuk meg, mégis látható lesz a build függvény hatáskörén kívül is, mi több, meghatározza mind a sort, mind a print függvények működését. Amennyiben párhuzamosan, egymástól függetlenül kívánjuk fejleszteni a három részprogramot (build, sort, print), úgy a munka csak a – meglehetősen bonyolult – table adatszerkezet pontos definiálása után kezdődhet. Az adatszerkezeten történő legkisebb módosítás valamennyi programrészre hatást gyakorol.

Az adatszerkezet-orientált megoldás

Ezen tervezési elv szerint első feladatunk az adatstruktúrák azonosítása, és mindazon absztrakt műveletek meghatározása, amelyek a szerkezet adott absztrakciós szintű használatához szükségesek. Mivel a bemenet olvasása közben gyűjtött adatainkat tárolni kell, az nyilvánvalónak tűnik, hogy a table egy adatszerkezet lesz. A rajta végrehajtandó műveleteket nem azzal kívánjuk meghatározni, hogy a táblázatot hogyan fogjuk implementálni (tömbbel, listával, fával, stb.), hanem azzal, hogy a szerkezeten a feladat megoldása szempontjából milyen műveleteket kell végrehajtani.

Kevésbé nyilvánvaló, hogy a bemeneti szövegfájl is absztrakt struktúrának tekinthető, mivel belső szerkezetét tekintve annyit bizonyosan tudunk róla, hogy sorokból, sorokon belül pedig szavakból épül fel. A bemenő fájl ilyen szintű szerkezetét maga a feladat szövege definiálja, ugyanakkor nem foglalkozik például azzal a kérdéssel, hogy mit is tekintünk szónak. Mi ezt a szerkezetet is a rajta végrehajtandó műveletekkel határozzuk meg. A kimeneti fájl szintén absztrakt adatstruktúrának tekintendő.

Ezen elemzéseket összefoglalva megállapíthatjuk, hogy három adatstruktúrában gondolkodhatunk, amelyeket a következőképp nevezünk el:

```
InFile    bemeneti fájl
Table     táblázat
OutFile   kimeneti fájl
```

A továbbiakban több – egyre jobb – kísérletet teszünk az InFile-on elvégzendő műveletek definiálására. Első próbálkozásként induljuk az alábbi két művelettel:

```
void getWords(wordList);
BOOL endOfFile( );
```

A getWords függvénynek bemenő paramétere nincs, kimenete legyen a bemenő fájl következő sorának szavaiból képzett lánc. Az endOfFile a hagyományos *eof* függvényeknek felel meg. A két műveletet felhasználva a tervezett programunk beolvasó része a következő formában állítható elő:

```
int linenumber = 0;
while (! endOfFile( )) {
   linenumber ++;
   getWords(wordList);
   while (wordList nem üres) {
      (következő szót venni a szóláncból);
      (tárolni a szót és/vagy  a linenumbert-t a
      table-n értelmezett műveletekkel)
   }
}
```

Ha elgondolkodunk az implementálhatóságon, azonnal belátjuk, hogy a szólánc kimenet nem volt szerencsés választás. A getWords függvényt használó programnak és a getWords készítőjének egy meglehetősen bonyolult struktúrában kell kiegyeznie, ami a szerkezet precíz definiálásával jár. Ekkor nem használhatunk magasabb absztrakt szintet, mint a nyelvi konstrukciók, vagyis nagyjából ugyanott vagyunk, mint a funkcionális megközelítésnél.

Második lehetőségként próbáljuk meg az átadandó szerkezetet egyszerűsíteni.

```
void getWord(word);
BOOL endOfLine( );
BOOL endOfFile( );
```

Ennél a választásnál a getWord függvény a következő szót adja meg. A két boolean típusú függvény a szokásos jelentésű. A fentebb szereplő programrész így módosul:

```
int linenumber = 0;
while (! endOfFile( )) {
   linenumber ++;
   while (! endOfLine( )) {
      getWord(word);
      (tárolni a szót és/vagy linenumbert-t a table-n értelmezett műveletekkel)
   }
}
```

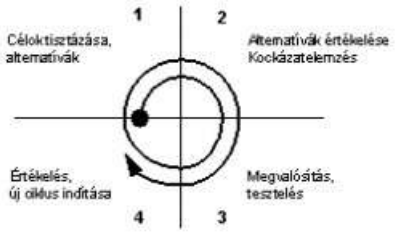
Ezzel a megoldással sikerült elkerülnünk a bonyolult szólánc használatát. Az InFile műveletei azonban még mindig nem tekinthetők elég jónak, mivel a bemenő fájl magánügyének tekinthető sorváltás felkerült a program szintjére. A sorok kezelése, a sorszámok számlálása a program felelőssége, nem pedig az adatszerkezeté. Gondoljunk arra az esetre, ha a bemenő fájlban értelmezett a folytatósor. Definiálhatjuk például, hogy ha a sor első két karakterén felkiáltójelek állnak, akkor ezt a sor tekintsük az előző folytatásának. Megengedhetjük a többszörösen folytatott sort is. Ezen esetekben a sor program szintjén történő értelmezése igen kellemetlen, hiszen a bemenő fájlt nemcsak hogy szó szinten, de karakter szinten is ismerni kell. Világos, hogy a sorváltás értelmezését kizárólagosan az InFile adatszerkezet hatáskörébe kell utalni.

Az utolsó változat műveletei:

```
void get(word, linenumber);
BOOL morePairs( );
```

Harmadik kísérletre sikerült olyan műveleteket találnunk, amelyekkel a bemenő fájl fizikai és logikai szerkezetét – azon túl, hogy a fájl sorokból és azon belül szavakból áll – sikerült eltakarni a külső szemlélő elől. A get függvény egy szó/sorszám párost ad vissza, míg a morePairs igaz lesz mindaddig, amíg van további szó a fájlban.

A definiált absztrakt struktúrát tekinthetjük a fizikai bemeneti fájl egyfajta transzformáltjának is. Feltételezve, hogy a bemeneti fájl tartalma a következő:



absztrakt adatszerkezetet használva a transzformált fájlt az alábbinak látjuk:

Így az InFile adatszerkezetet kezelő program:

```
while ( morePairs( )) {
   get(word, linenumber);
   (tárolni a szót és/vagy linenumbert-t a table-n értelmezett műveletekkel)
}
```

Elhagyva a didaktikus megközelítést, definiáljuk a Table szerkezetet a következő műveletekkel:

```
void store(word, linenumber);
void retrieve(word, linenumber);
BOOL moreData( );
```

A store művelet segítségével tudjuk a táblázatban a szó/sorszám párokat elhelyezni. A retrieve művelettől azt várjuk, hogy a Table táblába store-ral addig betett párosok közül a névsor szerint következőt adja. Ezzel a definícióval nem döntöttük még el, hogy a táblában milyen lesz a tárolási módszer, hogy az azonos szavakat egyszer vagy többször tároljuk, valamint azt sem rögzítettük, hogy mikor történjen meg a sorbarendezés.

Az *OutFile* adatszerkezeten mindössze egyetlen

```
void print(word, linenumber);
```

műveletet definiálunk, és így szabadon kezelhetőnek hagyjuk az output formázását. Nem kell a program szintjén gondolnunk a hosszú sorok és a laptördelések kezelésére.

Az absztrakt szerkezetek alkalmazásával keletkező program összességében alig bonyolultabb, mint a funkcionális változat első szintje. A függvények neve elé pont közbeiktatásával írt adatszerkezet nevet jelen pillanatban tekintsük egy fölösleges (redundáns) jelölésnek, amelynek célja emlékeztetni arra, hogy a függvény melyik adatszerkezethez tartozik.

```
while (InFile.morePairs( )) {
   InFile.get(word, linenumber);
   Table.store(word, linenumber);
}
while (Table.moreData( )) {
   Table.retrieve(word, linenumber);
   OutFile.print(word, linenumber);
}
```

Az adatszerkezet-orientált megoldást választva azonnal ki lehet adni a munkát az adatstruktúráknak megfelelően három részre felbontva. A részek között a kapcsolat nagyon laza, mindössze a szót és a sorszámot kell egyeztetni, amely adatszerkezetek lényegesen egyszerűbbek lesznek, mint a funkcionális dekompozíció esetében a tábla. Lényegében a programot sikerült olyan részekre bontani, amelyek közötti összetartó erő (kohézió) nagy és a részek közötti kapcsolat (csatolás) gyenge.

1.3. 2.1.3. Irány az objektum!

Az adatstruktúra domináns fogalomként történő alkalmazásával – a fenti példához hasonlóan – sok esetben szellemes megoldáshoz juthatunk. Ha az így tervezett programot meg kell valósítanunk, akkor a példányosítás lehetőségének korlátai a láthatósági viszonyok megváltozását is eredményezhetik.

Példaként induljunk ki a korábban definiált veremből, amelyet az alábbi műveletekkel adtunk meg.

Művelet	**Értelmezési tartomány**	**Értékkészlet**
NEW	( )	verem
PUSH	(verem, elem)	verem
POP	(verem)	verem
TOP	(verem)	elem
EMPTY	(verem)	boolean

Amíg egyetlen példányunk van a veremből, addig a műveletekben a veremre való hivatkozás felesleges, hiszen a művelet nem vonatkozhat másra. Ebben az esetben a vermet implementáló adatstruktúra és a függvények egyetlen C programmodulban definiálhatók, amelyből a külvilág számára csak a verem.h-ban *extern*-ként megadott függvények láthatók.

```
VEREM.C:
   static struct Verem { ... };
   void New( ) { ... };
   void Push(struct elem * x) { ... };
   void Pop(struct elem * x) { ... };
   void Top (struct elem * x) { ... };
   BOOL Empty( ) { ... };

VEREM.H:
   extern void New( );
   extern void Push(struct elem * x);
   extern void Pop(struct elem * x);
   extern void Top (struct elem * x);
   extern BOOL Empty( );
```

A vermet használó program:

```
#include "verem.h"

struct elem { ... } a,b,c;
...
```

```
New( );
Push(&b);
...
Top(&c);
if (Empty( )) ...
```

Természetesen a verem számára definiálni kell az elem struktúrát, amit célszerű a program deklarációs (header) fájljából átvenni.

Ha két vagy több vermet használunk, akkor meg kell adni, hogy melyik veremre vonatkozzon a műveletünk. Ezt két módon tehetjük meg. Az egyik megoldás szerint annyi veremmodult hozunk létre különböző nevek alatt (például a műveletek neve elé illesztett előtaggal jellemezve: APush, BPush, illetve Averem, Bverem stb.), ahányra csak szükségünk van. Ennek a megoldásnak az a hátránya, hogy a programunkban csak statikusan deklarált vermeket használhatunk, hiszen a vermek neve és száma fordítási illetve szerkesztési időben kerül meghatározásra. Dinamikusan nem hozhatunk létre vermet, hiszen ez annyit tenne, hogy a programunkhoz futási időben kellene újabb modult kapcsolni. A megoldás előnye, hogy a vermek belső szerkezete rejtve marad. Elviekben a különböző vermekhez különböző implementációs módszerek tartozhatnak. Például az Averem lehet egy tömb, míg a Bverem dinamikus szerkezet.

A másik megoldás az, hogy egyetlen vermet implementáló függvény-csoportot hozunk létre, amelynek kívülről átadandó paramétere lesz a verem:

```
void New(struct Verem * v) { ... };
void Push(struct Verem * v; struct elem * x) { ... };
void Pop(struct Verem * v; struct elem * x) { ... };
void Top (struct Verem * v; struct elem * x) { ... };
BOOL Empty(struct Verem * v;) { ... };
```

Ezen megoldás előnye, hogy a függvények egyetlen példányban jelennek meg, ugyanakkor nagy hátránya, hogy megsértjük az információk elrejtésének elvét, hiszen az adatszerkezetek az alkalmazók számára láthatóvá válnak. Nemcsak láthatók, de a felhasználók felelősségévé válik a veremtípusú változók korrekt definiálása és kezelése.

A fenti példából látható, hogy az absztrakt adatszerkezetek alkalmazásának központi problémája a példányosítás, azaz miként tudunk definiált tulajdonságokkal (műveletekkel) leírt adatszerkezetekből tetszőleges számú példányt létrehozni. Példányosítást alkalmazunk akkor, ha például megadjuk az

```
int a, b;
```

kifejezést. Ekkor keletkezik két olyan példány (a és b néven) az int adatszerkezetből, amelyeken az int-re vonatkozó műveletek végrehajthatók. Az int belső szerkezete rejtett, az egyszerű felhasználás során nem szükséges ismernünk a tárolás szabályát (például byte-sorrend, vagy 2-es komplemens módszer). A műveletek a programozási nyelvekben a szabványos típusokhoz – így az int-hez is – előre meghatározottak.

A programozási nyelvek egy részében bevezették a felhasználó által definiálható felsorolás típusú változókat. Ezen egyszerű szerkezetek konstansainak nevét adhatja meg a felhasználó. A műveletek rögzítettek és nagyjából az értékadásra, az összehasonlításra, az előző és a következő érték választására, illetve az értéknek az egészek halmazára történő leképezésére korlátozódnak.

A programozási nyelvekben a változókon értelmezett és megvalósított műveletek megnevezésének és jelölésének értelmezése is fontos tanulságokkal szolgál. A nyelvekben implementált különböző típusokon végezhető műveleteket – általában akkor, ha szemantikailag azonos értelmű – azonos néven érhetjük el. A művelet elvégzésének módját és az eredmény típusát az operandus típusa határozza meg. Gondoljunk itt arra az esetre, hogy önmagában a "+" műveleti jelből nem dönthető el, hogy hatására milyen eredmény születik, hiszen az eredményt az határozza meg, hogy a műveletet milyen típus változón alkalmaztuk. Hasonló módon értelmezhetjük például a ki- és beviteli folyamatokat elvégző függvényeket és eljárásokat, amelyeknek különféle paraméterei lehetnek és az eredményük is a paraméterek típusaitól függ.

A programozási nyelvekben ábrázolt műveletek jelölésében is számos, nehezen magyarázható korláttal találkozunk. Ilyen a műveleti jel és az operandusok elrendezésének viszonya. Hagyományosan a kétoperandusú matematikai műveletek úgy ábrázoljuk, hogy a műveleti jel az operandusok közé kerül. Ezt *infix* jelölésnek nevezzük. Például:

```
a + b; cx / 5.23; b1 or b2; t * u; egyik = másik;
```

A felhasználó által definiált szerkezeteken (változókon) csak részben engedélyezett az infix jelölés használata. Általában megengedett az értékadás és bizonyos esetekben a relációs operátorok (egyenlőség, kisebb, nagyobb) használata. Ugyanakkor, ha definiálunk például egy komplex szám típust, mint két valósat tartalmazó rekordot, azon nincs lehetőségünk az összegzés infix jelölésére. A komplex összegzést csak mint függvényt tudjuk definiálni, azaz a műveleti jelet (a függvény nevét) kötelezően az operandusok elé kell tenni. Ezt *prefix* jelölésnek nevezzük. Azaz

```
cmplplus(a, b)
```

A programnyelvek irányából közelítve célszerűnek látszik olyan szerkezetek kidolgozása, amelyek ugyanúgy példányosíthatók, mint a változók, ugyanakkor a rajtuk értelmezett műveletek is példányosodnak. Ez abban nyilvánul meg, hogy az elvégzendő műveletet nem csak a művelet jele vagy megnevezése határozza meg, hanem a művelet az adatszerkezettel együtt értelmezett.

Az említett tulajdonságokkal rendelkező szerkezeteket objektumoknak tekinthetjük.

2. 2.2. Az objektum fogalma

Az előző pontban a programozók, programtervezők mindennapi gyakorlatából (implementáció közeli praxis) vezettük le azokat az igényeket és elvárásokat, amelyek az objektum programnyelvi megjelenéséhez vezettek. A cél az volt, hogy egy objektumra vonatkozó *minden információ a program egyetlen helyén jelenjen meg* – legyen az akár egy adatszerkezet, akár egy algoritmus kifejtése egy adott absztrakciós szinten – és ez az információ a program többi részén – sőt más programokban is – könnyen újrahasználható legyen.

Az előző fejezetben a rendszeranalízis és a tervezés szemszögéből vizsgálódva jutottunk el ahhoz az igényhez, hogy *a problématér és az implementáció szerkezete minél inkább hasonlítson egymásra,* azaz a problématér (és így a valóság) objektumai jelenjenek meg az implementációban is.

A szisztematikus programtervezés alkalmazása során tudatosult, hogy egy információ-feldolgozási probléma két oldalról közelíthető, az algoritmusok, illetve az adatok oldaláról. Ez a két oldal szorosan összefügg, egyik a másik nélkül nem áll meg, és szoros kölcsönhatásban vannak [Wir82]. A feladatok jellegétől függően egyik vagy másik oldal nagyobb hangsúlyt kaphat (lásd. adatbázis-kezelés kontra gyors Fourier transzformáció). A gyakorlott rendszertervezők és programozók mindennapi munkájuk során számos elvet ösztönösen használnak az elmondottak közül, sokan alakítottak ki olyan stílust önmaguknak, ami igen hasonló az objektumorientáltsághoz. Sőt az is kiderült, hogy mind az adatok, mind az algoritmusok oldaláról közelítők sok tekintetben hasonló megoldásokra jutottak. Ezért is lehetett átütő sikere azoknak a megfogalmazott módszertanoknak, valamint azon új programozási nyelveknek, amelyek ezeket az elveket rendszerezetten összefoglalják, és alkalmazásukat támogatják. A szakirodalom tanúsága szerint az objektumorientáltság mára az egyik legnépszerűbb, legelfogadottabb paradigma a számítástechnikában.

Paradigmának nevezünk egy világszemléletet, látás- és gondolkodásmódot, amelyet az adott fogalomkörben használt elméletek, modellek, és módszerek összessége jellemez.

Az **objektumorientáltság** tehát egy szemléletmód, paradigma, amelynek alapján több rendszerfejlesztési módszertant is kidolgoztak és publikáltak. Ezek a módszertanok a modellezéstől az implementációig sőt a karbantartásig átfogják a teljes fejlesztési folyamatot. Az a rendszer, amelynek fejlesztésére, leírására a módszertanok használhatók, értelmezhető a szoftver rendszernél lényegesen tágabban is, jelenthet egy integrált hardver-szoftver együttest vagy egy szervezetet is.

Az objektumorientált módszertan alkalmazásával a kifejlesztendő rendszert együttműködő objektumokkal modellezzük, a tervezés és az implementáció során pedig ezen objektumokat "szimuláló" programegységeket alakítunk ki.

Az analízis során a rendszert együttműködő objektumok összességeként modellezzük. Például a minket körülvevő világ olyan objektumokkal írható le, mint emberek, házak, városok, autók stb. Ugyanezen világ jellemezhető olyan objektumokkal is mint óvoda, iskola, egyetem, tanár, diák. A modell objektumait az határozza meg, hogy a rendszer milyen vonatkozásait akarjuk megjeleníteni, az objektum modell a valóság mely szeletét reprezentálja. Az objektumok valamilyen absztrakciót tükröznek.

Ez a modellalkotás a köznapi gondolkodásban is megfigyelhető. Az emberek a világ dolgait objektumokként kezelik. Emberek milliói képesek autót vezetni anélkül, hogy pontos képük lenne arról, mi történik a motorház fedele alatt. Az autó számukra jól definiált kezelői felülettel rendelkező objektumok csoportja, amelyek az emberrel és egymással együttműködve összességében "autó"-ként viselkednek. Az autó használata szempontjából nem különösebben érdekes, hogy a motor benzines, dízeles vagy a hajtóerőt esetleg kalickába zárt mókusok szolgáltatják.

Az objektumorientált szemlélet alkalmazásával, a valóság és a modell kapcsolatának szorosabbá tételével nagymértékben megkönnyítjük a valóság megértését. Másfelől a világ gyakorta ismétlődő dolgainak a szoftver objektumokban történő modellezésével a vágyott cél, az újrafelhasználhatóság irányába is jelentős lépést tehetünk. Így reményeink szerint az objektumorientáltság a szoftverkrízis leküzdés-ének egyik lehetséges eszköze lehet.

Történetileg az objektumorientált szoftverfejlesztés területén publikált legismertebb módszertanok Rumbaugh, Booch, Shlaer, Mellor és Coad nevéhez kapcsolódnak [Rum91], [Boo86], [Boo94], [Shl92], [Coa90]. Az eltérő részletektől, hangsúlybeli különbségektől, eltekintve a módszertanok mindegyike az objektummal kapcsolatos közös fogalmakra épít, amelyeket a fejezet további részében foglalunk össze. Elsősorban a szemléltetés a célunk, a szigorúan formális definíciókat csak a szükséges mértékig alkalmazzuk.

2.1. 2.2.1. Az objektum

Az objektum fogalmát sokan, sokféleképpen határozták meg. Néhány ezek közül:

- Az objektum egy valós vagy absztrakt, azonosítható, egyedi entitás, amely a vizsgált környezet egy jól definiált szereplője.
- Egy objektum más objektumokra hatást gyakorol és más objektumok hatással vannak rá.
- Objektumként definiálható bármi, aminek pontos határai meghúzhatók.
- Az objektum jellemezhető a rajta értelmezett műveletekkel és a rajta elvégzett műveletek hatását rögzítő állapottal.

A sokféle definíció mindegyike megragad valami lényegeset az objektumból, de önmagában egyik sem teljes. Az objektum olyan közmegegyezéses meghatározása még nem is alakult ki, amilyeneket a hosszú ideje művelt, fogalomrendszerükben megállapodott szakterületeken megszoktunk. Fogadjuk el a továbbiakra az alábbi definíciót, amelynek alapján az objektummal kapcsolatos legfontosabb fogalmakat be tudjuk vezetni:

Az objektum egy rendszer egyedileg azonosítható szereplője, amelyet a külvilág felé mutatott viselkedésével, belső struktúrájával és állapotával jellemezhetünk.

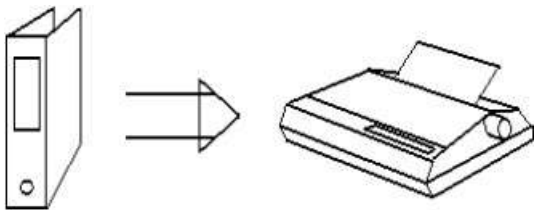
Az objektum tehát a rendszerben olyan szerepet játszik, amilyenre a rendszer feladatainak ellátásához szükség van. Ezen szerep betöltése a rajta kívül álló megfigyelő, illetve együttműködő számára az objektum viselkedésében, azaz valamiféle cselekvések, tevékenységek végrehajtásában nyilvánul meg. A külső szemlélő (többi szereplő) nem lát bele az objektumba, a struktúrára és az állapotra csak a viselkedésből következtethet. Ennek megfelelően azt sem látja, hogy a viselkedés során milyen az objektum belső működése. Ez az információelrejtés elvének következetes alkalmazása, amit **egységbe zárásnak** (*encapsulation*) nevezünk. Az objektum egységbe zárja az állapotát tároló adatokat és azok szerkezetét, valamint a rajtuk végrehajtott, az objektum viselkedését meghatározó műveleteket. Az objektum tervezőjének feladata a belső struktúra, a lehetséges állapotok kialakítása és a viselkedés programozása.

2.1.1. 2.2.1.1. Az objektum felelőssége

Az objektumtól elvárjuk, hogy maradéktalanul eljátssza a rárótt szerepet, azaz rendelkezzen minden olyan ismerettel és képességgel, amire ehhez szükség van, de semmivel sem többel. Másként fogalmazva, az

objektumhoz **felelősség** rendelhető, illetve rendelendő, amelynek vállalása és teljesítése az objektum feladata. A **lokális felelősség elve** szerint minden objektum felelős önmagáért.

Vizsgáljuk meg, hogy a következő példában szereplő objektumokat milyen felelősséggel célszerű felruházni! Legyen az egyik objektumunk egy fájl. A fájl, amit a 2.4. ábrán egy irattartó képvisel, tetszőleges formátumú lehet, tartalmazhat ASCII kódú szövegeket és/vagy különféle grafikus formátumokban megadott grafikát. A másik objektum egy mátrixnyomtató. A kérdés, hogy ki legyen felelős a fájl tartalmának korrekt kinyomtatásáért? Melyik objektumot milyen felelősség terheljen?



Egy "okos", felelős nyomtató programjának felépítése a következőképp írható le.

```
void print(file) {
   switch kind_of(file) {
      case ascii:     print_ascii(file); break;
      case msword20:  print_msword20(file); break;
      case tiff:      print_tiff(file); break;
      case bmp:       print bmp(file); break;
      case .....
   }
}
```

A nyomtató így egyaránt alkalmas lenne szabványos ASCII kódolt, MS Word for Windows 2.0 szöveges fájlok és tiff valamint bmp grafikus formátumok kezelésére. Azaz többféle bonyolult adatszerkezet fogadására és értelmezésére lenne képes, így benne nagyon sok tudást kellene felhalmozni. Ez felesleges, sőt értelmetlen, hiszen a formátumok változásával együtt kellene változni a nyomtatók programjainak, feltéve, hogy egy új formátum megjelenésekor nem akarunk új nyomtatót vásárolni. A sok, különféle formátum elfogadásának felelősségét vállalni ésszerűtlen, hiszen a felelősség növekedésével a használhatóság nem nő, sőt inkább csökken.

Ha a nyomtatókat úgy árulnák, hogy azok különféle dokumentum formátumokat ismernek fel, komoly zavarba kerülnénk a vásárláskor. Nehezen tudnánk előre megmondani, hogy mondjuk egy év múlva milyen formátumot akarunk majd nyomtatni, másrészt a nem használt formátumok okán nem biztos, hogy meg akarunk fizetni olyan szolgáltatást, amire semmi szükségünk sincs.

Észszerűbb megoldás, ha a nyomtatókhoz keresünk egy olyan fogalomkészletet, amelyik mindenféle szöveg, ábra, kép leírására alkalmas. Ennek elemei lehetnek például a karakterkészlet, az adott felbontású pontmátrix, a színek, a lapméret stb. A nyomtató felelőssége legyen annyi, hogy hibátlanul valósítsa meg a hozzárendelt emulációt (szabványos, IBM Proprinter, PostScript stb.), azaz adjon egy precíz felületet, amely egyike az általánosan elterjedteknek, és amely a fenti fogalomkészlettel specifikálható.

A formátummal kapcsolatos ismereteket a fájlnak kell tárolni, hiszen ki tudhatná jobban magánál a fájlnál, hogy mit is tartalmaz. A fájlnak, mint objektumnak tudnia kell magán elvégezni a rá vonatkozó műveleteket, például törlést vagy nyomtatást. A belső szerkezet a külvilág számára legyen rejtett. Minden új formátummal való bővítés úgy képzelhető el, hogy a fájl tartalmazza az új formátum nyomtatásának (a szabványos printer felületre való leképezés) programját is. Ezek után egy print_yourself műveletet a fájl bármikor végre tud hajtani. Ezzel mind a fájlt felhasználó alkalmazásokat, mind pedig a fájl által használt objektumokat (pl. nyomtató) egyszer s mindenkorra függetlenné tettük a fájl formátumának változásaitól.

Összefoglalva, a kérdésre – miszerint melyik objektumhoz milyen felelősséget rendeljünk – azt válaszolhatjuk, hogy a fájlnak ismernie kell saját szerkezetét, tudnia kell magát kinyomtatni egy szabványos emulációt megvalósító nyomtatón. A nyomtató felelőssége a definiált interfész és az emuláció pontos megvalósítása.

2.1.2. 2.2.1.2. Az objektum viselkedése

Az objektum **viselkedése** az általa végrehajtott tevékenységsorozatban nyilvánul meg, a rendszer működése pedig az objektumok összjátékában.

Viselkedésének jellegét tekintve egy objektum lehet **aktív** vagy **passzív.** A *passzív* objektum mindaddig nem tesz semmit, amíg valamilyen környezeti hatás nem éri, például üzenetet nem kap egy másik objektumtól. Az *aktív* objektum folyamatosan működik, valamilyen tevékenységet végrehajt, aminek során más objektumokat is működésre bír, azokra hatást gyakorol.

2.1.3. 2.2.1.3. Üzenetek

A hatást az objektumok egymásnak küldött **üzeneteken (***message***)** keresztül fejtik ki. Feltételezzük, hogy az objektumok olyan "infrastruktúrával" rendelkező környezetben működnek, amely az üzeneteket pontosan továbbítja, és egyelőre nem foglalkozunk azzal, hogy hogyan. (Valójában ennek az "infrastruktúrának" a megteremtése az implementáció igen fontos problémája lesz. Egyszerű esetekben, például amikor a teljes szoftver egyetlen processzoron fut, az üzenetküldés általában nem más, mint a célobjektum adott eljárásának meghívása.) Az üzenet szerepe kettős, egyrészt az objektumok közötti *adatcsere* eszköze, másrészt az objektumok működésének *vezérlésére* szolgáló eszköz (például működésbe hoz egy passzív objektumot).

Minden objektum az üzenetek egy meghatározott készletét képes elfogadni és értelmezni. Az üzenet két fontos komponenssel rendelkezik: egyrészt van *neve*, másrészt vannak *paraméterei*, amelyeket az üzenet aktuális tartalmának is tekinthetünk.

Az üzenet név az üzenet fajtáját azonosítja az objektum számára. A név ugyancsak alkalmas statikus jellegű, az objektum tervezésének és implementálásának pillanatában ismert információ átadására. Egy adott nevű üzenet a működés során természetesen többször elküldhető, más-más vagy akár ugyanazon paraméterekkel. Ezek az üzenet konkrét példányai, amelyek dinamikusan, a működés során keletkeznek. Egy objektum általában több, különböző nevű üzenet fogadására képes. Az, hogy ezek közül egy adott pillanatban melyik érkezik, fontos információ az objektum számára. A különböző nevű üzenetek az objektum különböző reakcióit váltják ki. Ezért mondhatjuk, hogy az üzenet neve által hordozott információ az objektum számára vezérlő jellegű.

Az üzenet neve azt is meghatározza, hogy milyen paraméterek kapcsolhatók az üzenethez. A paraméterek alkalmasak, a működés közben keletkező olyan dinamikus információk átadására, amelyek az objektum tervezésekor és implementálásakor, sőt esetleg még a konkrét példány létrehozásakor sem ismertek. Azonos nevű, különböző paraméterekkel kapott üzenetek az objektum azonos reakcióját indítják el, ami azonban a paraméterek értelmezését követően természetesen különbözőképpen folytatódhat. Paraméter akár újabb objektum is lehet.

Érdekes kérdések – amit egy-egy konkrét módszertan alkalmazásakor tisztázni kell –, hogy aki az üzenetet kapja, tudja-e, hogy az kitől érkezett, illetve képes-e egy objektum szelektíven egy adott üzenetet csak adott másik objektumtól (vagy objektumoktól) fogadni.

Ugyancsak jelentős kérdés, hogy megengedünk-e egyidejűleg több aktív objektumot a rendszerben. Az objektumorientált szemlélet ezt szinte természetessé teszi. Mégis – a hagyományos, egygépes rendszerek és fordítóprogramok általános használatából adódóan – a konkurens objektumok modellezése, de főként az ilyenek implementációja, nem tekinthető még ma sem egyszerű rutinfeladatnak. Erre vonatkozóan a módszertanok nagy része is csak igen nagyvonalú eligazítást ad.

Ha konkurens objektumokkal dolgozunk, a modellezéskor a következő problémákkal kell szembenéznünk:

– *Indulhat-e* a rendszer eleve *több aktív objektummal*, ha igen, ezeket hogyan ábrázoljuk?

– Megengedünk-e *aszinkron üzenetküldéseket*, másként fogalmazva, megengedjük-e, hogy egy objektum valamely üzenet elküldése után ne várakozzon a válaszra, hanem további műveleteket hajtson végre?

– Elfogadhat-e egy objektum újabb üzenetet, miközben még az előző feldolgozását nem fejezte be? (Megengedjük-e az *objektumon belüli konkurenciát*?)

2.1.4. 2.2.1.4. Események

A módszertanok egy része az *üzenet helyett* az objektumok kölcsönhatásainak, együttműködésének modellezésére az **esemény (event)** fogalmát vezeti be. *Eseménynek az azonosítható, pillanatszerű történést nevezzük*, azaz a történés folyamatának részletei érdektelenek számunkra.

Az eseményeket az objektumokhoz és az üzenetekhez hasonlóan osztályozhatjuk, nevet adhatunk nekik és paramétereket rendelhetünk hozzájuk. Hasonlóan az üzenetekhez, a név statikus, a paraméterek pedig dinamikus információt hordoznak. A paraméterek leírják az esemény bekövetkezésének körülményeit.

Az eseményeket objektumok hozzák létre. Itt is feltételezünk egy olyan "infrastruktúrát", amelyik megoldja, hogy más objektumok értesüljenek az esemény megtörténtéről és hozzájussanak az esemény paramétereihez.

Az események használatakor még több olyan nyitott kérdés merül fel, amit egy konkrét módszertan alkalmazásakor tisztázni kell. Tekintsünk néhányat ezek közül:

– van-e az eseménynek címzettje (azaz tudja-e irányítani az eseményt kiváltó objektum, hogy ki reagáljon az eseményre);

– eldöntheti-e egy objektum (vagy a tervező) és ha igen, akkor hogyan, hogy az objektum mely eseményekre reagál;

– egy eseményre csak egyetlen objektum reagálhat-e, vagy többen is.

Az egyik módszertan [Rum91] például az *eseményt* használja az objektumok együttműködésének modellezésére, azonban olyan jelentéssel, ami a korábban bemutatott *üzenet* fogalomhoz áll közelebb (objektumok eseményeket küldenek egymásnak paraméterekkel).

Elképzelhető az üzenet és az esemény együttes használata is, amivel finom különbségeket tudunk tenni. Tükrözheti például a modell, hogy egy eseményt egy objektum észlel, és erről üzenetküldéssel értesít egy másik objektumot, amely számára viszont az üzenet érkezése önmaga is esemény. Gyakran egy-egy üzenet érkezésekor a fogadó objektum számára csak az üzenet megérkezésének ténye fontos (paraméter nélküli üzenetek esetén biztosan). Ezért az üzenet érkezését a fogadó oldalán természetes módon eseménynek tekinthetjük, amelyet az üzenet küldője váltott ki (generált).

Egy példán szemléltetve az elmondottakat tegyük fel, hogy egyetlen kommunikációs eszközünk a posta, amely különböző küldeményeket továbbít. Ha valamit hoz a postás, akkor üzenetet kaptunk. A küldemény típusa (levél, csomag, pénzesutalvány, stb.) megfeleltethető az üzenet nevének. Az adott típusú küldemény megérkezése esemény (például "levelet kaptam"), ami független a levél tartalmától. Néha a tartalom nem is fontos, hiszen megállapodhatok valakivel abban, hogy ha elküld egy üres borítékot, az azt jelenti, hogy "Ibolyának ötös ikrei születtek és mindenki jól van". Az esetek többségében azonban a levél tartalmának fontos szerepe van, ami befolyásolja a további viselkedésünket. A levél tartalma megfelel az üzenet paramétereinek. A küldeményben kaphatunk olyan dolgokat is, amelyek maguk is objektumok, azaz viselkednek, üzeneteket értenek meg (a paraméterek objektumok is lehetnek).

2.1.5. 2.2.1.5. Metódusok

Az objektumhoz érkező üzenet hatására az objektum valamilyen cselekvést hajt végre, annak megfelelően, hogy milyen felelősségeket rendeltünk hozzá. Ha egy objektum képes fogadni egy adott nevű üzenetet, akkor erre az üzenetre az üzenet neve által meghatározott **metódus** (**method**) végrehajtásával reagál. Az objektum viselkedésének pontos leírása (implementációja) metódusainak kódjában található. A metódusokra szokás operációként is hivatkozni.

Az üzenet tehát megmondja, hogy MIT kell csinálni, a metódus pedig azt, hogy HOGYAN.

Ugyanazon objektum azonos tartalmú üzenetekre különféleképpen is reagálhat attól függően, hogy korábban milyen hatások érték. Ha például a barátomnak küldök egy üzenetet, hogy adja kölcsön az X könyvet, arra azzal reagál, hogy eljuttatja hozzám a kért művet. Ha anélkül, hogy a könyvet visszaadnám két hét múlva ismét az X könyvet kérő üzenetet küldök, akkor nem könyvet, hanem figyelmeztetést kapok.

2.1.6. 2.1.1.6. Állapot

Ha egy objektum azonos tartalmú üzenetre különféleképpen tud reagálni, (mint a fenti példában a barátom, vagy a lift, amelyik a hívásomra attól függően indul el fölfelé vagy lefelé, hogy éppen hol áll), akkor az objektumban szükségképpen valamilyen nyoma marad a korábbi üzeneteknek illetve azok sorrendjének. Ezt a nyomot az objektum **állapotának** nevezzük, melynek valahol az objektumban meg kell jelenni. Vagyis az objektum a külvilág üzeneteire adandó válaszokat meghatározó metódusokon kívül belső állapotokat is tartalmaz. Az objektumban tehát egyaránt benne foglaltatik (*encapsulated*) a viselkedés és az állapotinformáció. Az objektum által végrehajtott metódus megváltoztathatja a belső állapotot. Az objektum állapotváltozóit kívülről közvetlenül nem láthatjuk, így nem is érhetjük el azokat. Ez a gondolat teljes mértékben egybeesik az absztrakt adatszerkezeteknél elmondottakkal. Az objektum szintén az információ elrejtésének egyik eszköze.

Az objektum állapotát az **attribútumai** tárolják. Az attribútumok **értékei** az objektum élete során változhatnak, ezért a szoftverben szokásos módon változókkal jelenítjük meg azokat.

Polimorfizmus

Valamely rendszerben természetesen különböző objektumok is kaphatnak azonos tartalmú üzenetet. Az üzenetre adott választ meghatározza az objektum viselkedése. Ismét egy példával világítanánk meg a jelenséget. Tételezzük fel, hogy Sanghajban élő nénikénknek a születésnapjára virágot szeretnénk küldeni. Ennek az a kulturált módja, hogy telefonon felhívjuk a virágküldő szolgálatot, és átadunk nekik egy üzenetet. Az üzenet tartalmazza a nénikénk címét, a virág kiszállításának dátumát, a csokorban szereplő virágok felsorolását és a bankszámlánk számát. A szolgálat felveszi az üzenetet, majd végrehajtja a feladathoz tartozó metódust. Ennek megfelelően most már ők hívják fel a sanghaji virágküldő partnert és átadják az üzenetünket, csak a bankszámlánk számát cserélik le a sajátjukra. A sanghaji virágküldő ugyanarra az üzenetre, amire a budapesti virágküldő telefont ragadott, most csokrot fog készíteni és biciklis futárral elviszi a nénikénknek. Láthatjuk, hogy a végrehajtandó metódus attól (is) függ, hogy ki kapta meg az üzenetet.

Ha egy objektum úgy küldhet üzenetet egy másik objektumnak, hogy nem kell figyelemmel lennie arra, hogy ki az üzenet vevője, akkor **polimorfizmusról** beszélünk. A polimorfizmus kifejezés magyarul többalakúságot jelent. A többalakúság megnyilvánulhat abban, hogy egy adott üzenetet különféle objektumok is fel tudnak ismerni, de értelmezhetjük úgy is, hogy egy üzenet attribútumai különfélék lehetnek. A programozásban mindkét változat meglehetősen elterjedt és egyik sem kötődik közvetlenül az objektumorientáltsághoz. Gondoljunk arra, hogy egy valós és egy egész változónak (különféle objektumok) egyaránt küldhetünk olyan üzenetet, hogy "+4". A művelet eredményének típusa attól függ, hogy mi volt a változó típusa. A második esetre példa a PASCAL nyelv write eljárása. A nyelv filozófiájától teljesen eltérően a write paramétereinek száma változó lehet, és a paraméterek értékei nagyon különböző nyelvi elemek lehetnek (különféle típusú változók és konstansok, még sztring is).

Összefoglalásul leszögezhetjük, hogy az *objektum* olyan modellje a világ egy részének, amely a számára kívülről érkező üzenetekre reagálva valahogyan *viselkedik* (*behavior*). Az objektumnak van egy kívülről nem látható belső *statikus struktúrája* (*structure*), amely az állapotok értékét rögzítő *attribútumokkal* definiálható. Beszélhetünk az objektum *állapotáról* (*state*), amely a *struktúrát egy adott pillanatban kitöltő értékek* halmaza, azonban ez időről időre dinamikusan változhat.

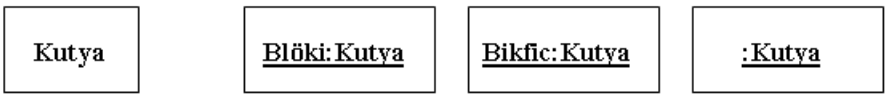
2.2. 2.2.2. Osztályok és példányok

Objektumokkal modellezett világunkban több egymással kommunikáló objektum található. Számos ezek közül nagyon "hasonlít" egymásra. Viselkedésük és struktúrájuk megegyezik, állapotuk azonban különbözhet. A megegyező viselkedésű és struktúrájú objektumok egy közös minta alapján készülnek, amit **osztálynak (class)** nevezzük. Az osztály tehát az azonos viselkedésű és struktúrájú objektumok gyárának, forrásának tekinthető. Az osztály és az objektumosztály kifejezések szinonimák. Az osztály megfelelői a beszélt nyelvben a gyűjtőnevek, mint például az ember, az autó, a ház, az ablak stb.

Az objektum – a viselkedését és struktúráját definiáló osztály egy **példánya (instance).** Mindegyik objektum egy önálló, létező egyed. Ez annyit jelent, hogy az egyébként azonos osztályból származó objektumok létükből eredően megkülönböztethetők, függetlenül pillanatnyi állapotuktól. Mindegyik objektum "ismeri" saját osztályát, amelyikből származik. Ez igen fontos kitétel, mivel az objektumorientált programozási nyelvekben gyakorta futási időben kell meghatározni egy objektum osztályát. Az objektum és a példány kifejezések egymással felcserélhetőek.

Az osztályok és objektumok ábrázolására a módszertanok grafikus technikákat is bevezettek. A jelölésrendszer, a rajztechnika lassanként egységesedett, kalakult a Unified Modeling Language (UML) leíró nyelv és grafikus jelölőrendszere. Az Object Management Group (OMG) konzorcium keretei között folyó szabványosítási munka eredményeként az UML folyamatosan fejlődő, élő szabvánnyá vált. Általában mind az osztályt, mind pedig az objektumot ábrázoló elem maximum három részből áll: tartalmazza az osztály és/vagy objektum nevét, az attribútumok nevét vagy azok értékét, továbbá a metódusok vagy műveletek felsorolását. Könyvünkben általában az UML [Rum91] jelölését követjük.

A 2.5. ábrán a baloldalon álló doboz a *Kutya* osztályt jelöli. A mellette álló három doboz három *kutya* objektumot (példányt) jelöl. Az első két esetben a *kutya* objektumok neve ismert, ez azonosítja a példányt, kettőspont után pedig az osztály neve áll, amelyikből az objektum származik. A harmadik objektum egy általános példányt jelöl, ahol a név nem ismert (vagy nem jellemző), csupán a kettőspont után álló osztály egy példányára utalunk.



2.5. ábra

Természetesen az osztály esetében is van lehetőség az attribútumok jelzésére. A most már két mezőre osztott négyszög felső részében szerepel az osztály megnevezése, alatta pedig az attribútumok elnevezése és típusa (2.6. ábra).

2.6. ábra

Az osztály fel szokták tüntetni a metódusokat is, mint ahogy az a 2.7. ábrán látható.

2.7. ábra

A 2.8. ábra összefoglaló jelleggel bemutatja az osztálydiagram egy elemének felépítését. Az attribútumok elnevezése mellett megadhatjuk azoknak típusát és az előre definiált kezdőértéket is. A metódus nevének megadását követheti a zárójelbe tett paraméterlista, valamint a metódus által szolgáltatott eredmény típusa. A metódusok és a programozási nyelvek függvényei (*function*) közötti hasonlatosság szándékos, mivel az objektum metódusait általában függvényekkel implementáljuk. Mindebből azonban elhamarkodott dolog lenne arra következtetni, hogy az üzenetküldés és a függvényhívás ugyanaz a művelet. Ez a következtetés – mint korábban már utaltunk rá – sok esetben helyes, de nem mindenkor.

2.8. ábra

Valamely objektum más objektumok felé csak a metódusait mutatja, biztosítva ezzel az információk elrejtését. Más kérdés, hogy az objektumban implementált metódusból nézve mit láthatunk a világból. Nyilvánvalóan érdemes korlátozni azt a tudást, ami egy metódusba a rajta kívüli világról beépül, hiszen enélkül a környezet változásai miatt túlságosan gyakran kellene a metódust megváltoztatni. Az objektum és a környezete közötti *csatolás gyengítésére* kell törekedni.

A Demeter-törvény szerint az objektum és környezete közötti csatolás akkor a leggyengébb, ha egy objektum metódusain belül csak az alábbi elemekre történik hivatkozás:

- a metódus paramétereire és eredményére,
- a metódust tartalmazó osztály attribútumaira,
- a program globális változóira,
- a metódusban definiált lokális változókra.

A fenti felsorolásban vitatható, hogy a globális változók mennyivel vannak "közelebb" az objektumhoz, illetve mennyivel elfogadhatóbb azok ismerete, mint mondjuk egy másik objektum által fogadott üzeneteké, amelyek láthatóan nem szerepelnek a megengedett hivatkozások között. Ha az objektum környezetét egy másik objektumnak tekintjük – ami mellett különösen akkor találhatunk meggyőző érveket, ha az objektumnak valamely másik alkalmazásban történő újrahasznosítására gondolunk – akkor a két eset között nincs különbség. A Demeter-törvényben szereplő felsorolást leginkább az indokolja, hogy a gyakorlatban használt objektumorientált programozási nyelvek más kompromisszumok mellett általában a globális változók használatát is megengedik.

Kétségtelen, hogy a más objektumok által fogadott üzenetek ismerete az objektumok közötti csatolás erősségét növeli. Ha megváltoznak az ismert külső objektum által elfogadott üzenetek, akkor ez, az üzenetet küldő objektum metódusában is változást okoz. A fogadó objektum metódusainak ismerete nélkül nem lehet együttműködő objektumokból álló rendszereket létrehozni. A fentiek szellemében azonban arra kell törekedni, hogy a kapcsolatokat lehetőleg minimalizáljuk.

2.3. 2.2.3. Az objektumok típusai

Gyakran használjuk az objektumosztály helyett az objektumtípus elnevezést. Valóban, a hagyományos programozási nyelvekben megszokott *típus* fogalom és az *osztály* között feltűnő a hasonlóság. A programozási nyelvekben használatos típusok (például integer vagy boolean) valójában osztályként viselkednek. A C nyelven leírt

```
int i;
```

sort objektum-szemlélettel úgy is értelmezhetjük, hogy legyen egy i nevű objektumunk, amelynek mintája az int-ek osztályában definiált, és értelmezettek rajta az egészre vonatkozó műveletek.

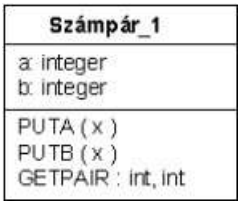
Akkor miben különbözik mégis a két fogalom? A **típus** egy objektum-halmaz viselkedését *specifikálja*. Definiálja az objektum által értelmezett üzeneteket és az operációk szemantikáját. Az *osztály* a típus által meghatározott viselkedést *implementálja*. Az osztály a típusnál szűkebb értelmű annyiban, hogy egy típus különféle struktúrájú osztályokkal is implementálható, de tágabb értelmű is, amennyiben tartalmazza az implementáció részleteit.

Példaként definiáljuk a *Számpár* objektumtípust az alábbi műveletekkel:

Művelet	Értelmezési. tartomány	Értékkészlet
PUTA	(integer)	számpár
PUTB	(integer)	számpár
GETPAIR	( )	integer, integer

A számpáron végrehajtott *PUTA* művelettel beállíthatjuk a pár első tagját, a *PUTB*-vel pedig a másodikat. Tegyük fel, hogy a *Számpár* objektum rendelkezik azzal a tulajdonsággal (szemantika), hogy a pár tagjainak összege mindig 10. Ezt úgy kell érteni, hogy ha beállítjuk a pár egyik tagját, akkor a másik tag automatikusan úgy változik, hogy az összeg 10 legyen.

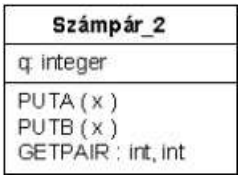
Ennek a típusdefiníciónak megfelelően készítsünk egy *Számpár_1* osztályt, amely a megfogalmazott viselkedésnek eleget tevő objektumok származtatására szolgál (2.9. ábra).



2.9. ábra

Ebben az osztályban az *a* és *b* állapotváltozók megfeleltethetők a számpárnak.

Definiálhatunk egy *Számpár_2* osztályt is, amely szintén implementálja a számpár viselkedését, de a struktúrája eltér *Számpár_1*-től (2.10. ábra).



2.10. ábra

Ez a megoldás csak egyetlen változót tartalmaz, például a számpár első tagját, a második tag pedig abból a feltételből számítható, hogy a pár összege 10.

Az osztály definíciójából következően a *Számpár_1* és *Számpár_2* különböző, mivel a struktúrájuk eltér egymástól. A típusuk – a viselkedésük – viszont interfész szinten és szemantikailag is megegyezik.

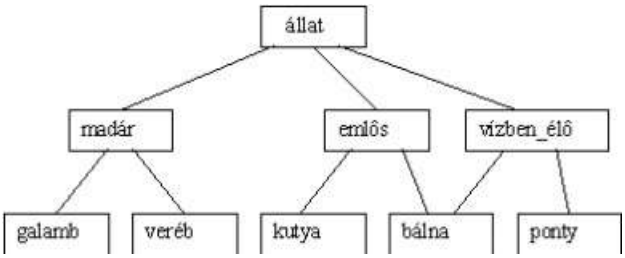
A típusokkal a hagyományos programozási nyelvek általában mint értékhalmazokkal foglalkoznak, és különböző szigorúságú ellenőrzéseket végeznek fordítási és futási időben arra vonatkozóan, hogy a programban előírt műveletek és operandusok típusa összeegyeztethető-e. Bizonyos nyelvek kevésbé szigorúak, és az értelmezhető típuskonverziókat automatikusan végrehajtják, például megengedik, hogy real és integer típusú számokat összeadjunk. Mások szigorúbbak, és megkövetelik a típusok azonosságát. Amíg csak adatokról van szó, a típusok azonossága, illetve egymásnak való megfeleltethetősége egyszerűen értelmezhető az értékhalmaz-szemlélet alapján. Nehezebb a helyzet, ha adatokat és metódusokat egyaránt tartalmazó objektumok típusazonosságát, illetve összeegyeztethetőségét akarjuk értelmezni.

A következőkben rövid matematikai fejtegetésbe bocsátkozunk, amelynek követése nem feltétele annak, hogy a könyv további részét megértsük. Ez a kis kitérő azonban segítségünkre lehet abban, hogy a későbbiekben általánosabban meg tudjuk fogalmazni az örökléssel kapcsolatos problémákat.

A típusokon nemcsak az azonosság és a különbség, de a **kompatibilitás** reláció is definiálható. Definíció szerint T típus kompatibilis U-val (T konform U-val), ha a T típusú objektum bármikor és bárhol alkalmazható, ahol az U típusú objektum használata megengedett. Az alkalmazhatóság azt jelenti, hogy minden az U típusú objektum által megértett M üzenetet a T típusú objektumnak is meg kell értenie. Azaz kijelenthetjük, hogy a T egy U (T *is_a* U).

A reláció reflexív, mivel T kompatibilis önmagával. Ugyanakkor nem szimmetrikus, hiszen abból, hogy T kompatibilis U-val, a fordítottja nem következik. A tranzitivitás szintén teljesül, hiszen ha T kompatibilis U-val és U kompatibilis V-vel, abból következik, hogy T kompatibilis V-vel is.

A kompatibilitás reláció mentén a típusok hierarchiája építhető fel. A 2.11. ábrán a szögletes dobozok típusokat jelölnek. Az alacsonyabb szinteken álló típusok kompatibilisek a felettük levőkkel a vonalak mentén. Tapasztalatainkkal összhangban kimondhatjuk, hogy a *madár* **egy** *állat*. A *veréb* **egy** *madár*. A tranzitivitás teljesül, így a *veréb* azon túl, hogy *madár*, még *állat* is.



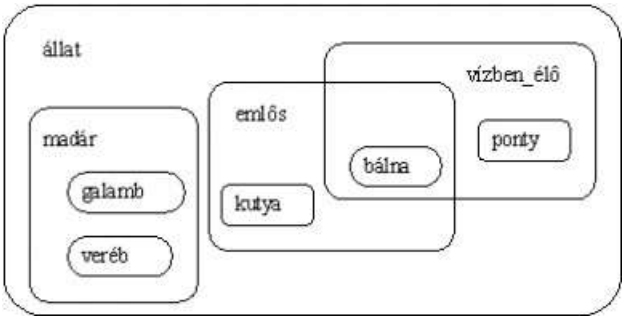
2.11. ábra

Ha T típus kompatibilis az U típussal akkor U típus a T típus **szupertípusa**. Ha az U típus a T típus szupertípusa, akkor a T típus az U típus **szubtípusa (altípusa)**. A szupertípusok általánosabb fogalmakat jelentenek, míg a szubtípusok specifikusabbakat.

Mivel a típusfogalom hasonló viselkedésű objektumok halmazát jelenti, amelyen a kompatibilitás egy tartalmazást valósít meg, a 2.11. ábra szerinti hierarchikus ábrázolás egyenértékű a 2.12. ábrán adott halmazábrával.

Definíció szerint a típus specifikálja az objektum által értelmezett üzeneteket, azaz meghatározza az objektum interfészét és leírja az operációk szemantikáját. Sok esetben a kompatibilitás vizsgálata kimerül az objektumok interfész szintű kompatibilitásának ellenőrzésében, és figyelmen kívül hagyjuk a kompatibilitás szemantikáját.

Mint korábban már definiáltuk, a "T kompatibilis U-val" kijelentésből következik, hogy minden az U típusú objektum által megértett M üzenetet a T típusú objektumnak is meg kell értenie. Amennyiben az M üzenet paraméteres és a paraméter maga is objektum, ráadásul a paraméteren is értelmezzük a kompatibilitást, akkor különös figyelmet kell fordítanunk a kompatibilitás "értelmének" vizsgálatára.



2.11. ábra

A 2.11. ábrán szereplő *állat* típusú objektumhoz definiáljunk egy *eszik(állateledel)* metódust. Az *állat.eszik(állateledel)* művelet értelmes és természetesen valamennyi szubtípus esetében ugyanezt várjuk, mint például *kutya.eszik(állateledel)*. Ha az *állateledel* szubtípusait is definiáljuk (legyenek például *mag, hús, hal*) akkor a kompatibilitás szabályait formálisan megtartva a *kutya.eszik(hús)* mellett a *kutya.eszik(mag)* és *veréb.eszik(hús)* operációkhoz juthatunk, amelyek azonban ellentétesek tapasztalatainkkal.

A kompatibilitás reláció fontos szerepet játszik a későbbiekben az objektumok közötti öröklésnél. Fenti rövid ismertetőnk célja az öröklés mögött álló elvek bemutatása volt, amelynek legfőbb tanulsága az, hogy a kompatibilitást nem szűkíthetjük az objektum kívülről látható interfészére, nagy figyelmet kell fordítanunk a műveletek helyes értelmezésére.

2.4. 2.2.4. Az objektum-változó

Eddigi vizsgálataink során megállapodtunk, hogy az objektum és a példány kifejezéseket szinonimaként használhatjuk. Vagyis az objektumon egy konkrét osztály konkrét példányát értettük, ami egy önálló létező egyed, amelynek viselkedését a típusa illetve osztálya határozza meg, az állapota pedig a kapott üzeneteknek megfelelően változik.

A programozásban használatos *változó* (*variable*) olyan elem, amely alkalmas érték befogadására. Ha az objektumot, a példányt, egy konkrét értéknek tekintjük, – és miért ne tennénk ezt – akkor jogos igényünk támadhat az objektum hordozására alkalmas változó bevezetésére. Az objektum-változó olyan szerkezet, amely objektum-példányt tartalmaz. Mivel az objektumot a neki küldött üzenetekkel érhetjük el, az objektum-változónak küldött üzenetet az általa éppen tartalmazott objektum kapja meg.

Az objektum-változó bevezetése kapcsán két további problémakörrel kell foglalkoznunk. Az egyik, hogy egy objektum-változó milyen objektum-értékeket vehet fel, a másik pedig az, hogy egy objektum-változónak milyen üzeneteket küldhetünk.

A programozási nyelvekben a változók általában **statikusan tipizáltak.** Ez azt jelenti, hogy a program forrásszövegében meghatározzuk (definiáljuk), hogy egy változó milyen típusú, azaz meghatározzuk a felvehető értékek halmazát. A változó típusa tehát a program fordításakor már ismert, és ettől kezdve rögzített. Ezt követően a típuskorlátozás betartása a programnyelvekben különböző. A PASCAL például **szigorúan típusos**, ami annyit tesz, hogy nem ad nyelvi eszközöket a korlátozások átlépésére. Természetesen kellő ügyességgel, – a konkrét implementáció eszközeit vagy a variálható rekordot kihasználva – a szabályok átléphetők. Ezzel szemben a C kevésbé szigorúan típusos, mert nyelvi eszközt ad (type casting) a korlátok megkerülésére.

A statikusan tipizált változók használatának előnye, hogy a programozási nyelvekben egyszerűen és hatékonyan implementálhatók a változókat kezelő műveletek. A tipizálás statikussága természetesen független attól, hogy a változó számára mikor foglaltunk helyet a memóriában.

A **dinamikusan tipizált** programnyelv a második és harmadik generációs nyelvek között meglehetősen ritka, de a negyedik generációs adatbázis-kezelő nyelvek jellegzetesen ilyenek. A dinamikus tipizálás lényege, hogy a változónak nincs előre meghatározott típusa, futás közben tetszőleges értéket felvehet. Ezek az értékek különféle típusúak lehetnek. Úgy is fogalmazhatunk, hogy ezek típus nélküli változók. A szabadság ára, hogy egy típus nélküli változót elég körülményes implementálni. Emellett a változón értelmezhető műveletek ellenőrzött megvalósítása is problematikus.

A változón végrehajtható műveleteket (a változóknak küldött üzeneteket) meghatározhatja maga a változó, vagy a változóban tárolt érték. A művelet és a változó kapcsolódását kötésnek (**binding**) nevezzük.

Statikus kötés esetén a változón értelmezett műveleteket – objektum-változó esetében a metódusokat – a *változó* típusa határozza meg. Ez történik a hagyományos programnyelvekben. A fordítóprogram a változó típusának ismeretében ellenőrzi, hogy a kijelölt művelet az adott változón végrehajtható-e, és ha igen, generálja a megfelelő kódot. Például a

```
a + 8
```

művelet eredményének típusát és a létrejövő kódot attól függően állítja elő, hogy milyen az a változó típusa. Más lesz a kód integer és más real változó esetén.

Dinamikus kötés esetén a műveletet meghatározó tényező a változó által hordozott *érték*. Az objektum-változók esetén ez azt jelenti, hogy értékadáskor a műveleteket végrehajtó metódusok is cserélődnek. A

változóval végzendő művelethez tehát most, futási időben rendelődik hozzá a műveletet végrehajtó eljárás (metódus) kódja. Ha korrekt módon akarunk eljárni, akkor bármiféle, a változóra kijelölt művelet végrehajtása előtt meg kell kérdeznünk a változót, hogy éppen milyen értéket tárol. Hiszen az sem biztos, hogy a végrehajtani kívánt művelet egyáltalán értelmezett a változóban éppen tárolt értékre (megérti-e a változóban tárolt objektum az üzenetet). Így érthető, hogy a dinamikus kötés implementálása lényegesen bonyolultabb a statikusnál.

A tipizálást és a kötést együttesen vizsgálva az eredmény az alábbiakban foglalható össze:

- Statikus tipizálás – statikus kötés

  Megfelel a hagyományos programnyelvekben szokásosan implementált megoldásnak. A változónak típusa van és ez a típus meghatározza mind a változó értékkészletét, mind pedig a változón végrehajtható műveletet. Elviekben a változó lehet objektum-változó, de az ilyen nyelveket nem tekintjük objektumorientáltnak.

- Dinamikus tipizálás – statikus kötés

  Azzal a meglehetősen furcsa helyzettel kerülünk szembe, hogy a változóba tetszőleges értéket – objektumot – helyezhetünk el, ugyanakkor a végrehajtható műveleteket – metódusokat – a változó nem létező típusa határozza meg. Ez az eset nem értelmezhető.

- Dinamikus tipizálás – dinamikus kötés

  A változó tetszőleges értéket felvehet és a végrehajtható műveleteket az érték határozza meg. Ez a koncepció nagyon rugalmas, az objektumorientáltság dinamizmusa érvényre jut, de nehéz implementálni. A Smalltalk programnyelv ilyen.

- Statikus tipizálás – dinamikus kötés

  Látszólag nincs sok értelme annak, hogy egy változóba elhelyezhető értékek típusát megkössük, ugyanakkor a végrehajtható műveleteket az értékekhez kössük. Ha szigorúan betartjuk a tipizálást, akkor ez így igaz. Ha azonban felhasználjuk a típusok kompatibilitásával kapcsolatos korábbi eredményeinket és a kompatibilitás reláció mentén felpuhítjuk a kemény típuskorlátozást, kijelentve, hogy egy adott típusú változó a vele kompatibilis típusok értékeit is felveheti, akkor már nagy jelentősége lehet ennek az esetnek. A tipizálás-kötésnek ezt a párosát gyakran implementálják programozási nyelvekben, így a C++-ban is.

Az objektum-változók bevezetése azzal a szabadsággal, hogy egy változóban a saját típusával kompatibilis típusú objektum is elhelyezhető – a polimorfizmus jelentését is kiteljesíti. Amennyiben az üzenetet a változónak küldjük, azt a változóban éppen jelenlevő objektum kapja meg, így előállhat az a helyzet, hogy valóban nem tudható előre, hogy milyen típusú objektum metódusa hajtódik végre (de a kompatibilitás miatt a művelet értelmezhető lesz).

Miután megismerkedtünk az alapfogalmakkal, a következő fejezetben azt vizsgáljuk, hogy egy valóságos vagy elképzelt rendszert hogyan modellezhetünk objektumokkal.

3. fejezet - 3. Modellezés objektumokkal

Korábban az objektumot a világ egy részének modelljeként definiáltuk. Mivel a világ egymáshoz kapcsolódó, együttműködő részekből áll, az objektummodell sem lehet más, mint egymáshoz kapcsolódó objektumok sokasága. Az objektum-modellezés célja, hogy a világot egymással kapcsolódó objektumokkal írja le.

Abból, hogy az objektumok összetett szerkezetek, – beszélhetünk a struktúrájukról, a viselkedésükről és az állapotukról – következik, hogy az egyes objektumokat különböző szempontok szerint írhatjuk le. Hasonlóképpen, az objektumok együttműködésének, a közöttük fennálló kapcsolatoknak ábrázolásához különféle nézőpontokat választhatunk.

Jelen fejezet célja az objektum-orientált modellek készítése során alkalmazott leírások bemutatása. Lényegében definiálunk egy grafikus jelölésrendszert, és megadjuk annak értelmezését. Azzal a kérdéssel, hogy miként készíthetünk a javasolt jelölésrendszernek megfelelő modelleket, nem most, hanem a 4. fejezetben foglalkozunk.

1. 3.1. A modellek áttekintése

Az a technika, amivel egy készülő műszaki alkotást több nézőpontból modellezünk (természetesen ezek a modellek ugyanannak az elkészítendő terméknek a különböző nézetei), nem új dolog. Egy bonyolult áramkörnek – például egy számítógép alaplapnak – vannak elvi működési (logikai), elrendezési, hőtechnikai tervei. Végül az elkészülő termék, amely ezen tervek mindegyikének meg kell, hogy feleljen, egyesíti magában az említett modellekben leírt tulajdonságokat. Hasonló példákat más szakterületről is hozhatnánk.

Egy számítógépes rendszernek ugyancsak számos modellje készíthető. A szoftver esetében a rendszert három nézőpontból vizsgáljuk. Koncentrálhatunk az *adatokra*, a *műveletekre* (funkciók, adattranszformációk) illetve a *vezérlésre* (viselkedés). A három nézőpontnak megfelel az objektum-orientált módszertanokban használt három modell. Ezek a modellek még közel sem annyira szabványosak, mint például az építészeti, gépészeti vagy villamos rajzok és leírások, de kialakulóban van egy közmegegyezés.

Az **objektummodell** az "adat"-ot tekinti domináns fogalomnak, és így írja le a rendszer statikus tulajdonságait és struktúráit. A **dinamikus modell** az időbeliséget rögzíti a "vezérlés" aspektusából. A **funkcionális modell** középpontjában a rendszer által végrehajtandó "funkció"-k állnak.

A három elkülönült nézőpont (adat, vezérlés, funkció) együttese kielégítő képet ad ahhoz, hogy tervezni és implementálni tudjunk. Miután ugyanazon dolog három különböző nézetéről van szó, a modellek szoros kapcsolatban vannak egymással.

A szoftver fejlesztése során mindhárom modell folyamatosan változik, fejlődik. Az analízis fázisban megkeressük és modellezzük problématér objektumait. A tervezéskor az implementációs tér objektumainak illetőleg ezek modelljeinek felhasználásával megfeleltetéseket keresünk, majd szimuláljuk a problématér objektumait az implementációs tér objektumaival. A modelleket tehát két szempont szerint is megkülönböztethetjük. Egyfelől az egyidejűleg használt, különféle nézőpont szerinti (objektum, dinamikus és funkcionális) modellekről, másfelől a fejlesztés egyes fázisaiban (analízis, tervezés, implementáció) megjelenő modellekről beszélhetünk.

1.1. 3.1.1. Objektummodell

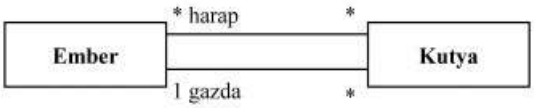
Az objektummodell leírja a rendszerbeli objektumok struktúráit, attribútumait és metódusait, valamint az objektumok közötti viszonyokat, kapcsolatokat, relációikat. Általában az objektummodell adja meg azt az alapot, amelyhez a dinamikus és funkcionális modellek kapcsolódnak. A változásokat és a transzformációkat kifejező dinamikus és funkcionális modelleknek csak akkor van értelmük, ha definiálható, hogy mik változnak, transzformálódnak.

Az objektummodell kialakításával az a célunk, hogy rögzítsük az alkalmazási területnek a feladat szempontjából lényeges dolgait és azok statikus viszonyait. A dolgokat az objektumok modellezik, a viszonyokat pedig az

objektumok közötti kapcsolatokkal, relációkkal írjuk le. Amennyire lényeges, hogy az objektumok helyesen modellezzék a dolgokat, annyira fontos, hogy a dolgok közötti viszony jól tükröződjön a relációkban.

Az objektumok ábrázolásáról szóltunk a 2. fejezetben. Az objektumokat és az objektumok közötti *kapcsolatokat* jeleníti meg az **osztálydiagram**, amelyet szokás *egyed-kapcsolati* vagy *entitás-relációs* diagramnak is nevezni. Valójában az objektum-orientált analízis során ritkán gondolkozunk egyedi objektumokban. Általában feltételezzük, hogy minden objektum a neki megfelelő osztályból származik, ezért az objektumosztályok közötti kapcsolatokat ábrázoljuk. Bizonyos esetekben kifejezőbb lehet az egyedi objektumpéldányok kapcsolatának feltüntetése, ami célszerűen az **objektumdiagramon** történhet.

Példaként tekintsük az *Ember* és a *Kutya* osztályokat és a közöttük lévő kapcsolatokat. A két osztály értelmezése feleljen meg az általánosan használt gyűjtőneveinknek. Tételezzük fel hogy az osztályok között fennáll a "gazda" viszony, ami annyit jelent, hogy egy *Ember* több *Kutya* gazdája is lehet, de egy *Kutyának* csak egyetlen gazdája van. Legyen közöttük egy "harap" kapcsolat is, amely szerint egy *Kutya* több *Embert* is megharaphat és egy *Embert* több *Kutya* is megharaphat. *Ember* nem harapja meg a *Kutyát*.



3.1. ábra

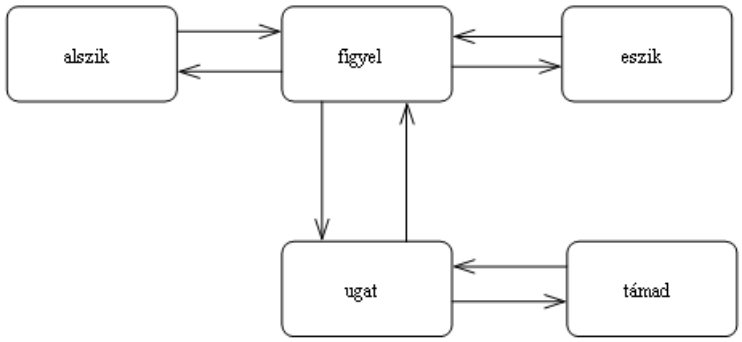
A fenti példa osztálydiagramja a 3.1. ábrán látható. A kapcsolatok megnevezését az osztályokat összekötő vonalra írjuk, az osztályokat reprezentáló dobozok közelében elhelyezett jelölések (szám, *) pedig a reláció számosságát fejezik ki.

1.2. 3.1.2. Dinamikus modell

A dinamikus modell a rendszer időbeli viselkedését írja le. Ez az időbeliség azonban nem feltétlenül jelent igazodást egy abszolút időskálához, gyakran elegendő a sorrendiség tükrözése. Ide értendők a rendszert, az objektumokat érő hatások, események és ezek sorrendje, a műveletek, a metódusok végrehajtásának ütemezése, az állapotok és azok változásainak rendje.

A dinamikus modell környezetét az objektummodell adja meg. Azoknak az objektumoknak a viselkedését és együttműködését kell leírni, amelyek az objektummodellben szerepelnek. A viselkedés leírására alkalmas eszközök a folyamatábrák, az állapotdiagramok és kommunikációs diagramok. Ugyancsak ilyen eszköz lehet valamilyen algoritmikus nyelv vagy pszeudonyelv, amely alkalmas a vezérlési szerkezetek leírására. Az objektum-orientált módszertanok leggyakrabban az állapotdiagramot és a kommunikációs diagramot használják.

Az állapotdiagram valamely objektumosztály egy reprezentáns példányának a külső események hatására történő állapotváltozásait és a válaszul adott reakcióinak időbeli sorrendjét adja meg. A leírásra táblázatos és grafikus formák egyaránt használatosak. A 3.2. ábrán látható állapotgráf a *Kutya* állapotváltozásait írja le. A diagramban az állapotok váltását okozó eseményeket, és a végrehajtott akciókat nem ábrázoltuk.



3.2. ábra

Az állapotdiagram az objektum időbeli (sorrendi) viselkedését írja le. Szükséges lehet az osztályok közötti *kapcsolatok időbeliségének* megadására is. A rendszer időbeli viselkedésének modellezésére használható az

üzenetek sorrendjét rögzítő kommunikációs diagramok sokasága. A rendszer által végrehajtott minden funkció leírható egy kommunikáció-sorozattal. Ennek megjelenési formája a kommunikációs diagram. A diagram egyetlen (rész)művelet végrehajtása során az osztályok között áramló üzeneteket írja le.

A módszertanok egyre gyakrabban használják a **vezérlési szál** fogalmát, különösen akkor, ha egyidejűleg több aktív objektum is megengedett a modellben. A vezérlési szál az objektumok műveleteit köti össze a végrehajtási, azaz ok-okozati sorrendnek megfelelően. Lényegében a hagyományos folyamatábra magasabb absztrakciós szintű változata. Egy vezérlési szál valamely feltételtől függően különbözőképpen folytatódhat *(ez megfelel a hagyományos feltételes elágazásoknak)*. A vezérlési szál átmehet egyik objektumból a másikba *(szinkron üzenetküldések, vagy másik objektum működésének elindítása és várakozás annak befejezésére)*, sőt ki is léphet a rendszerből, majd ismét visszatérhet oda *(kommunikáció külső objektumokkal, várakozás külső eseményekre)*. Végül a vezérlési szál elágazhat több párhuzamos ágra *(aszinkron üzenetküldések)*, illetve a különböző ágak egyesülhetnek *(egy objektum valamilyen cselekvéssorozat után várakozik egy üzenetre, amely után tovább dolgozik, míg a másik objektum az üzenet elküldése után már nem aktív)*. Egy rendszer működése elindulhat eleve több vezérlési szálon *(ha eleve több aktív objektumot tételezünk fel)* is.

1.3. 3.1.3. Funkcionális modell

Az objektumokból álló rendszerek nem függetlenek a környezetüktől. Minden rendszer tekinthető egy olyan szerkezetnek, amelyik a külvilágból érkező kezdeményezésekre válaszol. A rendszerek együttműködnek a külvilágban létező emberi vagy automatikus szereplőkkel, az aktorokkal. Az aktorok a rendszer használatától azt várják, hogy az számukra kiszámítható, meghatározható módon viselkedik, és a specifikáltnak megfelelően reagál (eredményt szolgáltat). A használati eset (use case) definiálja egy rendszer, vagy a rendszer valamely jól meghatározható részének a viselkedését, leírva az aktorok és a rendszer közötti együttműködést, mint akciók és reakciók (válaszok) sorozatát.

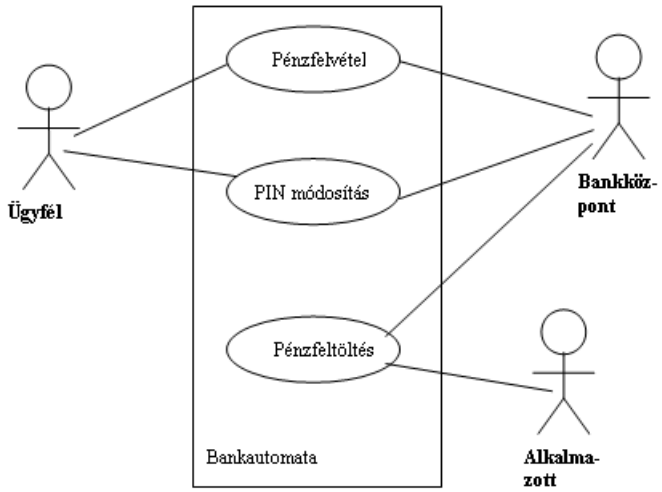
Megjegyezzük, hogy komplex rendszerek tervezésekor a rendszert részrendszerekre bonthatjuk (dekompozíció). Ekkor a fenti gondolatmenetet egy-egy részrendszerre is vonatkoztathatjuk, és így a részrendszerek együttműködését is leírhatjuk használati esetekkel.

Példaként vizsgáljunk egy bankautomatát, mint rendszert. Esetünkben az ügyfél az aktor, a leggyakoribb használati eset pedig a készpénz felvétele. A pénzfelvételt az ügyfél kezdeményezi azzal, hogy a kártyáját az olvasóba illeszti. A rendszer, leolvasva a kártyán szereplő adatokat, felszólítja az ügyfelet azonosító kódjának (PIN) megadására. A beolvasott kód valódiságának meghatározására a rendszer a bank központjához fordul. A bank központja szintén egy külső – nem emberi, hanem automatikus – szereplő, egy másik aktor. A bankközpont a kódot ellenőrizve, mint aktor, utasítja a rendszerünket az ügyfél kiszolgálásának folytatására vagy annak elutasítására. A folytatásban a rendszer bekéri az ügyféltől a felvenni szándékolt pénz mennyiségét. Az ügyfél gombnyomással választhat egy fix összegeket feltáró listából, de módjában áll a listában nem szereplő összeg megadására is. A használati eset definiálásakor természetes, hogy különböző alternatívák (helyes vagy hibás PIN kód, fix vagy definiálható összeg) léteznek, amelyek eltérő akció-sorozatokkal írhatók le. A további együttműködési lépéseket (interakciókat) nem részletezve: a legtöbb esetben az ügyfél pénzhez jut.

Rendszerünk további használati esetekkel leírható funkciókat is nyújt a külvilág számára. Általában az ügyfél lekérdezheti az egyenlegét, vagy megváltoztathatja a PIN kódját, esetleg más banki műveleteket (betétek elhelyezése) is végezhet. A bankautomata egy funkciójának tekinthetjük a pénzzel történő feltöltést, amit a banki alkalmazott – aki szintén aktor – végez.

A fenti példából láthatjuk, hogy a használati eset a rendszer és az aktorok közötti interakciók leírása. A leírás formája lehet folyó szöveg (mint a példában is), struktúrált (szabályok szerinti, táblázatos formára alakított) szöveg, de akár magas szintű vagy pszeudo programkód is.

A rendszer funkcionalitását és a környezetével fennálló kapcsolatát áttekinthetően megjeleníthetjük egy olyan ábrán, amelyen feltüntetjük a használati eseteket, az aktorokat és a közöttük levő kapcsolatokat. Ez a használati eset diagram (use case diagram).



Az ábrán látható pálcika-emberek az aktorok; a használati eseteket ellipszisek jelképezik. Az aktorok és a use case-ek közötti kapcsolatot (azt a tényt, hogy az adott használati esetben a megjelölt aktorral együttműködés alakul ki) az őket összekötő vonal jeleníti meg.

A következő pontokban részletesen tárgyaljuk a különböző modelleket.

2. 3.2. Az objektummodell

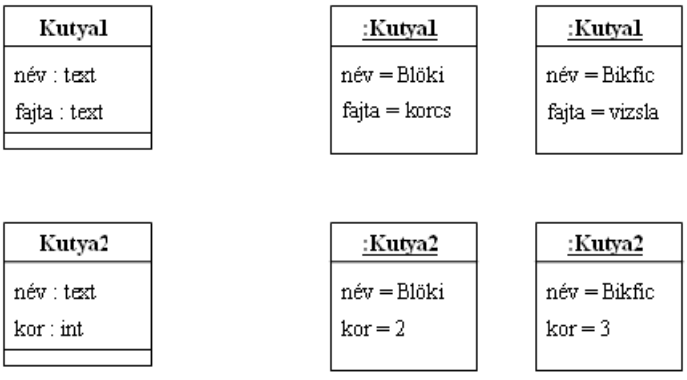
Az objektummodell a rendszerben szereplő objektumosztályok struktúráját, viselkedését és az osztályok valamint az objektumok egymás közötti kapcsolatainak statikus képét jeleníti meg. Ennek megfelelően először vizsgáljuk az attribútumok természetét, majd az objektumok közötti viszony általános jellemzőit tárgyaljuk.

2.1. 3.2.1. Attribútumok

Az attribútumok az objektum tulajdonságait és állapotát meghatározó objektumban tárolt adatok. Az attribútumok száma és típusa definiálja az objektum struktúráját, vagyis azt, hogy milyen belső változói vannak. Minden attribútum egy, az osztályra nézve egyedi névvel rendelkezik. Az attribútumok változóknak tekinthetők, amelyek típusuknak megfelelő értéket vehetnek fel. Attribútum nem lehet objektum, csak olyan ún. "tiszta" érték, aminek nincs identitása. Ugyanazt az értéket tetszőleges számú objektum attribútuma felveheti, azonban ezek az értékek nem különböztethetők meg egymástól. Példaként tételezzük fel, hogy a *Kutya* osztálynak van egy *kor* attribútuma, amely az állat korát egész számként tartalmazza. Valamennyi ötéves kutya esetében ez az attribútum ugyanazon "5" értéket veszi fel.

Az attribútumok lehetnek egyszerű vagy összetett típusúak, közvetlen adatok vagy referenciák (mutatók, pointerek).

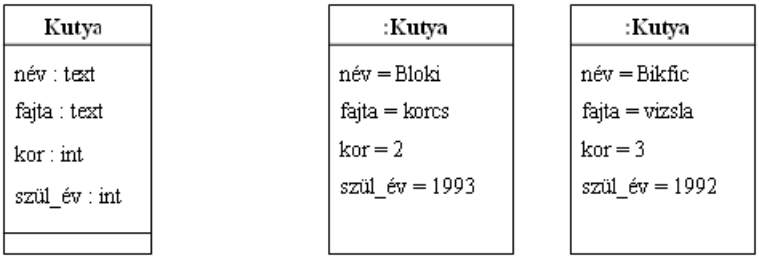
Egy adott osztályhoz tartozó attribútumhalmaztól elvárjuk, hogy legyen teljes, azaz az objektumnak a modellezés szempontjából valamennyi lényeges tulajdonságát fedje le. Próbáljuk meg kettéosztani az előző fejezetben megismert *Kutya* osztályunk attribútumait!



3.4. ábra

A korábbi három attribútumot tartalmazó objektumot úgy osztottuk meg, hogy a *Kutyát* egyértelműen azonosító jellemző (a *név)* mindkét objektumban megtalálható, de objektumonként csak egy-egy jellemzőt szerepeltetünk (*fajta* vagy *kor*). Első ránézésre látható, hogy a szétválasztás nem igazán sikeres. A probléma markánsan megmutatkozik abban, hogy nehezen tudunk az osztályoknak kifejező nevet találni, jobb híján *Kutya1*-nek és *Kutya2*-nek hívjuk őket. Pedig elképzelhető olyan feladat, amelyben a *Kutya* a *nevével* és *korával* jellemezhető, de olyan is, amelyben *nevével* és *fajtájával*. Mindkét osztály önmagában, a másik nélkül értelmes lehet, ha a világ bonyolultságát kevésbé árnyaltan tükröző modellt kell alkotnunk. Ha azonban mindkét attribútumra szükségünk van, akkor azok együttesen egyetlen osztályt jellemeznek.

Az attribútumokat akkor választjuk meg jól, ha egymástól függetlenek. Ha a *Kutya* attribútumaihoz hozzávesszük a születés évét is, akkor egyrészt egy már alkalmazott nézőpont érvényesül (a kutya kora), másfelől lesznek olyan attribútumok, amelyek egymásból meghatározhatók (lásd 3.5. ábra).



3.5. ábra

Összefoglalva megállapíthatjuk, hogy törekedni kell az attribútumhalmaz teljességére, az attribútumok függetlenségére és arra, hogy lehetőleg minden attribútumban különböző nézőpont jusson érvényre.

Az attribútumokat aszerint sorolhatjuk két csoportba, hogy milyen típusú információt tartalmaznak. **Elnevezés típusú** az az attribútum, amelyik az adott objektum valamely jellemzőjének megnevezését, címkéjét tartalmazza. Az ilyen attribútumok általában nagyon ritkábban változnak.

3.6. ábra

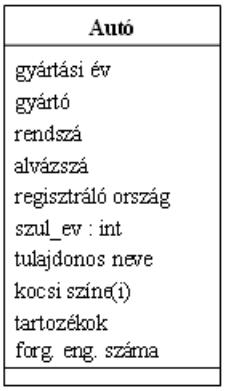
A fenti *Férfi* osztálynak elnevezés típusú attribútuma a *név* és a *személyi szám*. Általában ezek az attribútumok az objektum élete során nem változnak, de ha szükséges, bármikor meg is változtathatjuk őket. Ha például elírjuk a nevet, akkor ennek az objektumra különösebb hatása nincs. Abból, hogy valakit Tibornak vagy Miklósnak hívnak, nem következtethetünk emberi tulajdonságokra (minőségre).

A második típusba a **leíró** attribútumok tartoznak. Ezek az objektum olyan belső jellemzőit rögzítik, amelyek az objektum élete során, az objektumot ért hatások következtében változnak, és az objektumnak valamiféle kiértékelhető tulajdonságát adják meg. A fenti példánkban ilyen a *cím* és a *havi jövedelem*. Nyilvánvalóan a *jövedelem* attribútum 5000 forintos és 500000 forintos értékei eltérő minőséget jelölnek. Hasonlóképpen jelentős minőségi tartalma van egy pasaréti címnek illetve egy átmeneti szállás címének.

A példánkban feltüntetett *autórendszám* attribútum besorolása nem egyértelmű. Egyfelől minőségi jellemzőnek, azaz leíró attribútumnak tekinthetjük abban az értelemben, hogy az illetőnek van-e egyáltalán autója. Amennyiben van, akkor elnevezésként viselkedik, hiszen a rendszámból semmilyen minőségre nem tudunk következtetni. A többi attribútumtól eltérően az *autórendszám* kimutat a *Férfi* osztályból. Valójában nem egyéb, mint hivatkozás egy másik, egy *autó* objektumra. Azon attribútumokat, amelyek más objektumokra hivatkoznak **referenciáknak** nevezzük. A referenciák az objektumok közötti kapcsolatokat, relációkat valósítják meg.

Minden objektum önálló, létező példány, amely léténél fogva azonosítható. Az objektum-orientált programozási nyelvek automatikusan létrehozzák azokat a referenciákat (mutatókat), amelyekkel az objektumra hivatkozni lehet. Az olyan attribútumokat, amelyek csupán az eléréshez, azonosításhoz szükségesek, fölösleges felvenni az objektumban, amennyiben azok nem a modellezendő világ részei. Természetesen a valós világban is gyakran használunk azonosítókat. Senki nem vonja kétségbe, hogy minden autó önálló létező dolog, amely egyértelműen azonosítható a térben éppen elfoglalt helye szerint. Mi ennek ellenére használunk olyan attribútumokat, amelyek az egyedi autópéldányt azonosítják.

A 3.7. ábrán adott *Autó* osztályunkban ilyen azonosító lehet például a *gyártó* és az *alvázszám* attribútumok együttese.



3.7. ábra

Hasonlóképp egyértelműen azonosítja az autót a regisztráló ország és a rendszám együttese, illetve a regisztráló ország és a forgalmi engedély száma. Ugyanakkor megállapíthatjuk, hogy ebben a példában nincs olyan attribútum, amelyik egyedül, önmagában azonosítaná az *Autó* adott példányát.

2.2. 3.2.2. A relációk és a láncolás

A **reláció** az objektumok, illetve osztályok közötti kapcsolatot jelent. A **láncolás** (*link*) logikai vagy fizikai kapcsolat *objektum példányok között.* Például az alábbi "Szabó úr a Füles kutya gazdája" relációban az *Ember* és a *Kutya* osztályok egy-egy példányának összeláncolását (alkalmanként szó szerint vehetően pórázzal) jelenti a *gazda* viszony**.** Az *osztályok* között értelmezett kapcsolatot **asszociációnak** nevezzük. Az asszociáció azt fejezi ki, hogy a két (esetleg több) osztály példányai kapcsolatban vannak (vagy lehetnek) egymással, közöttük láncolás alakul(hat) ki. A láncolás így az asszociáció példányának tekinthető.

A matematikában a *reláció* fogalmán halmazok Descartes szorzatából képzett részhalmazt értünk, ami ugyancsak a halmazok elemeinek összekapcsolását jelenti. A reláció megadása véges halmazok esetén történhet felsorolással, általános esetben pedig a részhalmazba tartozás feltételét definiáló állítással lehetséges.

Könyvünkben a *reláció* fogalmat nem matematikai szigorúsággal, hanem a korábban említett, a köznyelvi jelentéshez közelebb álló értelemben használjuk. Relációról akkor beszélünk, ha nem kívánjuk megkülönböztetni az osztályok és az objektumok közötti kapcsolatokat.

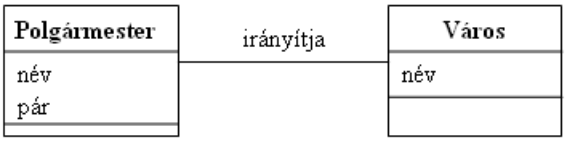
A reláció jelölésére általában a viszonyt kifejező igét használjuk. Ez az eljárás a megoldás arra, hogy ránézve a modellt ábrázoló diagramra, azonnal fel tudjuk idézni a modellezett kapcsolat valóságos tartalmát. A példánkban mondhatnánk, hogy a Férfi *gondozza* a Kutyát. Az ilyen megnevezés általában a kapcsolat tényén túl a kapcsolat irányára is utal. A *gondozza* kapcsolat nyilván nem szimmetrikus, más viselkedés tartozik hozzá a gondozó és a gondozott részéről. Vannak olyan kapcsolattípusok, amelyeket könnyebben jellemezhetünk a kapcsolatot kifejező főnevekkel, illetve ezek birtokos alakjával (pl. testvér(e), szülő(je), főnök(e) stb.). Ezek között vannak szimmetrikusak (testvér), mások pedig ugyancsak aszimmetrikusan irányítottak. Aszimetrikus esetben szerencsésebb egy szópár megadásával egyértelművé tenni a viszonyokat (szülő-gyerek, főnök-beosztott). A valóságos kapcsolat megnevezésére általában többféle megoldás közül választhatunk (a *főnök-beosztott* viszonyt kifejezhetjük a *vezet-vezetett* igés alakkal).

Valójában a kapcsolatok mindig kétirányúak. Ha az egyik irányt egy cselekvő igével jelöljük ugyanez a kapcsolat a másik fél szemszögéből egy szenvedő igével jellemezhető. A szenvedő szerkezet a magyar nyelvben

nem mindig egyszerű. Példánkban a kutya nézőpontjából a Kutya *gondozva van (gondozott, gondoztatik)* a Férfi által. Ezért gyakran csak az egyik, könnyebben kifejezhető irányra utalunk az elnevezéssel, de ettől függetlenül a kapcsolatokat mindkét irányból értelmezzük. Ha nem jelöljük meg a megnevezéshez tartozó irányt, sokat veszíthetünk a modell leíró erejéből. Természetesen mondhatjuk, hogy két osztály között fennáll az adott nevű kapcsolat, a pontos értelmezést pedig lásd annak leírásánál. Ez azonban megnehezíti a modell áttekintését. A "beszélő név" alapján azonnal következtetni tudunk a valóságos tartalomra, az irány ismeretében pedig azonnal tudjuk, melyik objektumnál keressük a cselekvő és melyiknél a szenvedő viselkedés leírását. A köznyelvi értelmezésre és élettapasztalatunkra alapozva az irány sok esetben magától értetődőnek látszik a név alapján. Korábban azonban láttuk, hogy ezekre a tapasztalatokra építve csúnya félreértésekre is juthatunk.

2.2.1. 3.2.2.1. Bináris relációk és jelölésük

Amennyiben a reláció pontosan két objektumot (osztályt) kapcsol össze, akkor **bináris relációról** beszélünk. Az osztálydiagramon szereplő osztályokat jelentő dobozokat összekötő vonalak az asszociációkat jelölik. Az objektum diagramon a láncolást ugyanígy tüntetjük fel. A relációt jelképező vonalra szokás felírni a viszony megnevezését. Az irány jelölésére általában a megnevezést ahhoz az osztályhoz közelebb helyezik el, *amelyik felé* a megnevezéssel jelölt viszony irányul.



3.8. ábra

A 3.8. ábrán azt látjuk, hogy a *Polgármester* osztály és a *Város* osztály között az *irányítja* asszociáció áll fenn, mégpedig a városra irányulóan.

Felmerül a kérdés, hogy a bejelölt asszociációnak a konkrét objektumok tekintetében mi a pontos jelentése. Vajon egy polgármester hány várost irányít, illetve egy várost hány polgármester irányít? Biztos-e, hogy minden polgármester irányít várost, illetve minden városnak van-e polgármestere? A további tervezés szempontjából ezek igen fontos kérdések.

Az objektum diagramon az **asszociációk számosságát** (**multiplicitását**) is szokás feltüntetni, amiből választ kaphatunk a fenti kérdésekre.

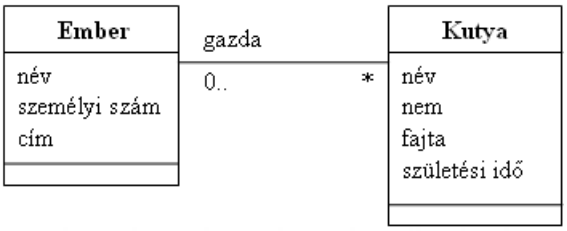
Az asszociáció multiplicitása megadja, hogy az asszociáció az egyik osztály adott példányát a másik osztály (amelyikre *az asszociáció irányul) hány példányával kapcsolja, vagy kapcsolhatja össze.*

A modellezés szempontjából a konkrét számértéknek nincs nagy jelentősége. Fontosabb, hogy a kapcsolat opcionális (megengedett, hogy ne tartozzon másik osztálybeli példány egy objektumhoz), illetve hogy a kapcsolódó elemek száma egy vagy több.

A számosságot általánosságban nem negatív egészek halmazaként adhatjuk meg. A halmaz elemei az adott objektumhoz a másik osztályból hozzákapcsolható példányok megengedett darabszámait jelentik. Ezek között szerepelhet a 0 (ilyenkor opcionális a kapcsolat), és elvileg a végtelen is, amelyre a "sok" vagy "több" megnevezést használjuk. A sok, vagy több jelölés általában azt takarja, hogy nem tudunk, vagy nem akarunk foglalkozni azzal, hogy mekkora legyen a felső korlát. A halmazt felsorolással vagy részintervallumok kijelölésével adjuk meg

Természetesen a számosságot az asszociáció mindkét irányára külön-külön meg kell adni, hiszen a két irány multiplicitása független egymástól.

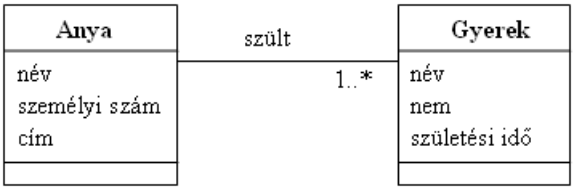
A multiplicitást az osztálydiagramon is jelölik. A rajzon az osztályokat összekötő, a relációt jellemző vonal önmagában 1-1-es kapcsolatot jelöl. A 3.8. ábra jelentése, hogy *egy* várost *pontosan egy* polgármester irányít, és viszont, *egy* polgármester *pontosan egy* várost irányít. A számosság megjelenítésére különböző szimbólumok állnak a rendelkezésünkre. Az UML jelölés szerint a relációt jelképező vonal végére rajzolt * a "sok példány"-t jelöli (beleértve a 0-t is), mégpedig abban az irányban, amelyik oldalon a * elhelyezkedik. A számosság lehetséges intervallumait is megadhatjuk (pl. 0..1).



3.9. ábra

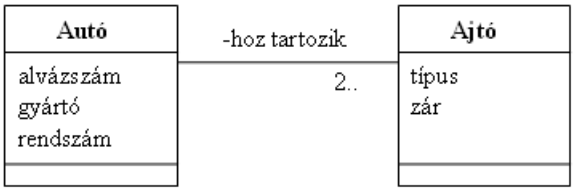
A 3.9. ábra magyarázatakor egy osztály reprezentáns objektumát az osztály kis kezdőbetűvel írt nevével jelöljük. Az ábra azt tükrözi, hogy egy *ember* 0 vagy több (* a *Kutya* felőli oldalon) *kutyának* lehet a gazdája. Tehát létezhet olyan *ember*, aki nem gazdája *kutyának*, azaz nincs kutyája. Megfordítva azt mondhatjuk, hogy egy *kutyának* legfeljebb egy *ember* a gazdája, de a modellünkbe beleférnek a gazdátlan, kóbor ebek is (0..1 jelölés az *Ember* felőli oldalon).

A 3.10. ábrán az *Anya* és a *Gyerek* osztályok közötti asszociációt ábrázoltuk. Egy *anya* biztos, hogy szült legalább egy *gyereket*, mert anélkül nem lehet anya. Ezt a tulajdonságot reprezentálja a "több" jelölés mellé írt korlátozás, az 1..*, amely a 0 előfordulást nem engedélyezi. A másik irányban a reláció teljesen egyértelmű, mert minden *gyereket* pontosan egyetlen *anya* szült.



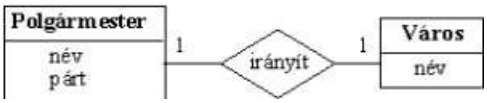
3.10. ábra

A számosságnak egy intervallumra való korlátozását láthatjuk a 3.11. ábrán. Egy szokásos autón található ajtók száma kettő és öt között van. Ezt adjuk meg az Ajtó oldalára írt korlátpárral (2..5).



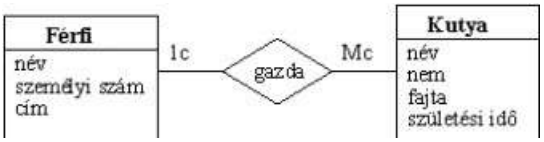
3.11. ábra

A reláció és számosság jelölésére más grafikus jelölésrendszerek is használatosak. Széles körben alkalmazott a **Chen-féle jelölés**, amely a relációt az objektumok közé rajzolt rombusz segítségével ábrázolja és a számosságot az összekötő vonalakra írt számok jelzik. A reláció irányát – néhány kivételesen fontos esettől (lásd. is_a) eltekintve – nem jelöli, azt magyarázatban adja meg. Írjuk le korábbi példáinkat a Chen jelölés szerint.

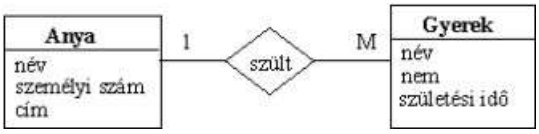


3.12. ábra

Ebben a jelölésben a *több* kifejezésére az "M" betűt használjuk (3.13. és 3.14. ábrák), aminek értelme nem azonos a sötét karikával, mivel az M az opcionalitást (0) nem engedi meg. A 0 esetet feltételesnek (kondicionálisnak) nevezzük és "c" betűvel jelöljük.

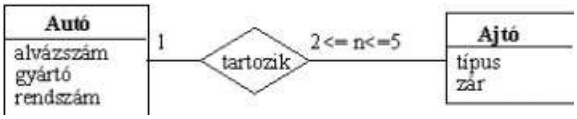


3.13. ábra



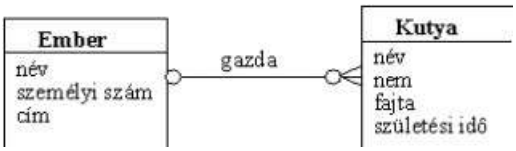
3.14. ábra

A számosság megadására megengedett a kifejezések alkalmazása is. (3.15. ábra)



3.15. ábra

A gyakorlatban alkalmazott harmadik jelölésrendszer a számosság ábrázolására a "csirkeláb" szimbólumot használja és az opcionalitást az üres karikával jelöli. Maga a "csirkeláb" minimálisan 1-et jelent. A multiplicitás korlátozására szintén megengedett a kifejezés alkalmazása. A *gazda* relációnak ebben a formában a 3.16. ábra feleltethető meg.



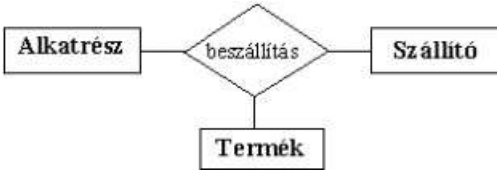
3.16. ábra

A számosság erősen függ attól, hogy hol húzzuk meg a rendszer határait. Tény, hogy lehet olyan település, amelynek nincs polgármestere, mivel a legutolsó távozását (meghalt, lemondott, leváltották, stb.) követően még nem sikerült újat választani. Hasonló módon valamennyi fenti példa megkérdőjelezhető a modell megfelelő értelmezésével. Mondhatjuk, hogy igenis lehet a kutyának két gazdája, ha például azok testvérek. Vannak olyan autók, amelyeknek 5-nél több ajtója van. A rendszer elemzésekor elsőként az objektumok, az osztályok és az asszociációk felderítésével kell foglalkoznunk, a számosság vizsgálatának csak a határok pontos kijelölése után van jelentősége.

2.2.2. 3.2.2.2. Többes relációk és jelölésük

Egy relációban összekapcsolt objektumosztályok száma az esetek nagy részében kettő. Az elemzés során találkozhatunk kettőnél több osztály között fennálló relációval, de ezek többsége alapos vizsgálat után szétszedhető bináris relációkká. Elvétve előfordul, hogy három osztály kapcsolata nem bontható fel információvesztés nélkül – ezt **ternáris relációnak** hívják. Csaknem kizárt a háromnál több osztályra kiterjedő n-es reláció.

A ternáris relációkban három osztály olyan viszonyban van, amely viszony nem állítható elő az érintett osztályok páronkénti relációinak összességeként. A ternáris reláció jelölésére az OMT-rendszerben is a Chen-féle jelölést használjuk (3.17.ábra).



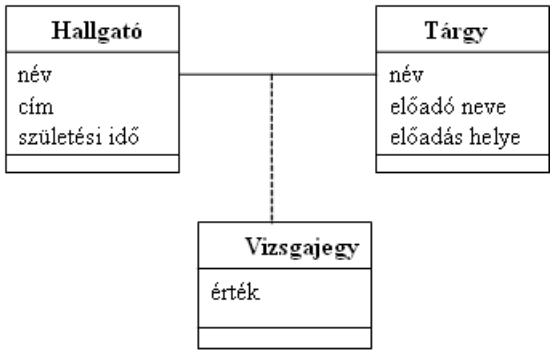
3.17. ábra

A példában szereplő üzemben gyártott termékekhez szállítók alkatrészeket szállítanak. Ugyanazt az alkatrészt több szállító is szállítja. Egy szállító többféle alkatrészt szállít. Egy termékben többféle alkatrészt is használnak. Egy termékben előforduló egyforma alkatrészek származhatnak különböző szállítóktól. Amennyiben tudni

akarjuk, hogy valamely termékbe adott szállító egy alkatrészből szállított-e, akkor ternáris relációt kell definiálnunk.

2.2.3. 3.2.2.3. Az asszociáció, mint osztály

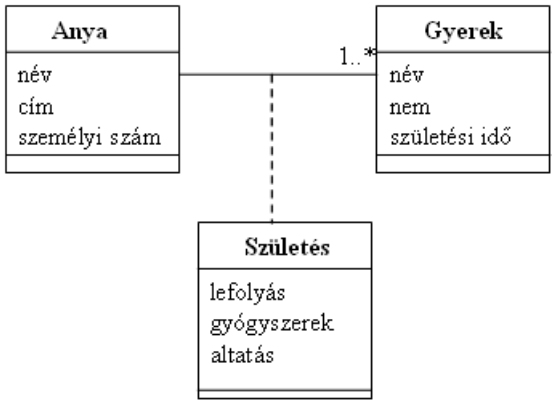
Az attribútum az objektum tulajdonságát fejezi ki. Hasonló módon az objektumok közötti láncolásnak, így az asszociációnak is lehetnek attribútumai. A láncoláshoz tartozó attribútumok markánsan a több-több típusú relációkban jelennek meg. Példaként vegyük az egyetemi hallgatókat és az általuk felvett tárgyakat. Egy hallgató több tárgyat is felvehet és egy tárgyat sokan hallgathatnak. Ha rajzos formában kívánjuk ábrázolni a láncolás attribútumát, ezt az UML jelölésrendszerében úgy tehetjük meg, hogy az asszociációt jelképező vonalhoz szaggatott vonallal egy dobozt kapcsolunk, amelynek második részébe beírjuk az attribútum megnevezését (3.18. ábra).



3.18. ábra

Abból a tényből, hogy a hallgató felvett egy tárgyat, az következik, hogy vizsgát kell tennie. A vizsgajegy csak akkor mond valamit, ha megmondjuk, hogy melyik tárgyból melyik hallgató szerezte. A vizsgajegy a tárgy felvételéhez kötött, nem pedig a hallgatóhoz vagy a tárgyhoz.

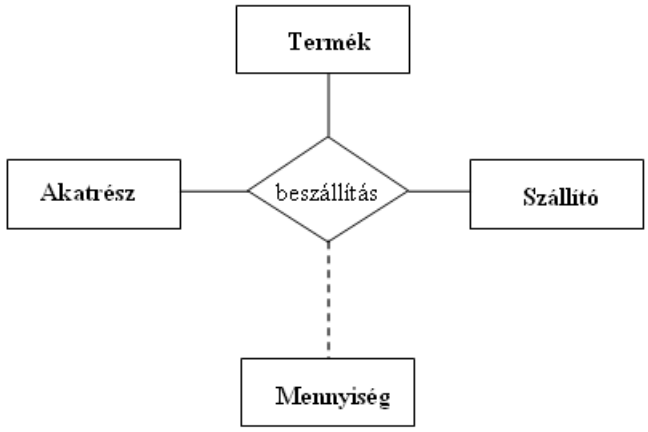
Természetesen tartozhat attribútum az 1-több relációkhoz is. Ilyenkor nagy kísértést érzünk, hogy az attribútumot a több oldal objektumához csapjuk hozzá. A korábban már szerepelt *Anya-Gyerek* példánkat egészítsük ki a szülés körülményeinek leírásával (3.19. ábra).



3.19. ábra

Nyilvánvaló, hogy a jelölt attribútumok a szüléshez tartoznak és ésszerűtlen lenne a szülés közben az anyának adott gyógyszereket a gyereknél nyilvántartani.

A bináris 1-1 és a ternáris relációknak is lehetnek attribútumai. Ternáris relációnál az attribútumot a relációt jelképező rombuszhoz kötjük. Ha a korábbi *alkatrész-szállító-termék* relációban a szállított alkatrészek számát is feltüntetjük, akkor az csak a relációhoz kapcsolódhat.



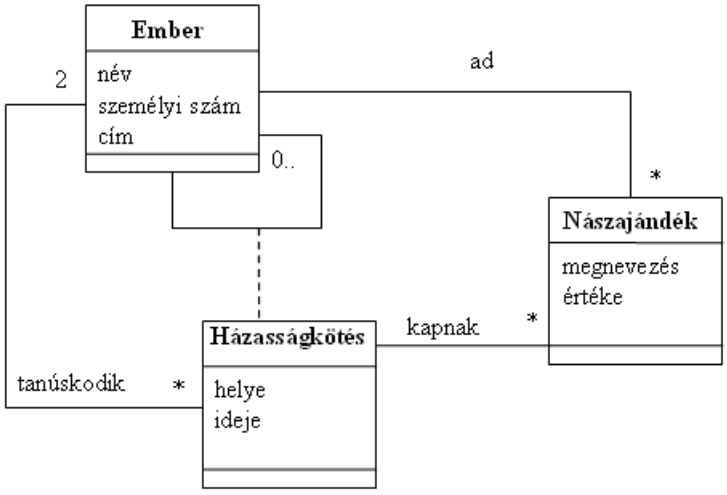
3.20. ábra

A 3.19. ábrán szereplő *szül* relációhoz kapcsolódó attribútumok jelentését megvizsgálva beláthatjuk, hogy maga a szülés objektumként is kezelhető. Méghozzá olyan objektumként, amelyik összekapcsolja az érdekelt *Anyát* és *Gyereket*, és önálló struktúrája és viselkedése definiálható. A relációt megjelenítő objektumot asszociatív objektumnak, az osztályt asszociatív osztálynak szokás nevezni. A rajzon az ilyen osztályt az asszociativitás attribútumaihoz hasonlóan hurokkal az osztályokat összekötő vonalhoz kapcsoljuk.

Sok esetben érdemes az asszociációból osztályt készíteni különösen akkor, ha ehhez az osztályhoz újabbak kapcsolónak, mint a 3.21. ábrán szereplő példánkban.

Két ember között lehetséges viszony a házasság. A házasságot megjelenítő házasságkötés asszociatív osztályhoz újabb osztályok kapcsolódnak. Amennyiben megpróbáljuk a *Házasságkötés* osztályt elhagyni, komoly nehézséget jelent a *Nászajándék* osztálynak az *Emberhez* kapcsolása. Nem egyértelmű, hogy a párnak szánt ajándékot melyik egyénhez kössük.

Az asszociáció osztályként történő kezelése egyben az asszociáció egyik implementálási lehetősége. Minden bináris láncolás helyettesíthető egy olyan objektummal, amelynek attribútuma a relációban szereplő objektumokra vonatkozó két referencia és a láncolás saját attribútuma(i). Ezen láncoló objektumok osztálya az asszociációt képviselő osztály.



3.21. ábra

Általánosságban is kimondható, hogy az n-es reláció n referenciát tartalmazó objektumok osztályaként is leírható.

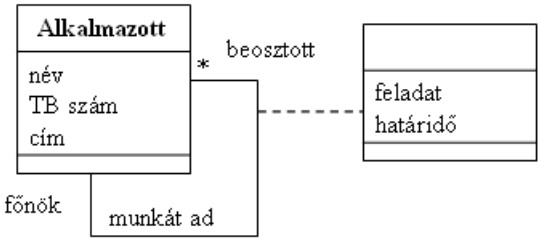
2.2.4. 3.2.2.4. Szerepek

A 3.21. ábra példázza, hogy két osztály között több, különböző reláció is fennállhat (házasság, tanúskodik), illetve egy relációban azonos osztálybeli (*Ember*) objektumok is összekapcsolódhatnak. Példánkban ez a kapcsolat (házasság) különös, mert a relációban álló objektumok egyforma szerepet játszanak (házastárs).

Gyakoribb eset, amikor az azonos osztályból származó láncolt objektumok között nem szimmetrikus a viszony. Ez azt jelenti, hogy egy objektum a reláció mindkét oldalán előfordulhat, de ettől függően egyrészt másként viselkedik, másrészt eltérő számú másik objektummal lehet kapcsolatban. Minden objektum eljátszhatja a reláció mindkét oldalához rendelhető szerepet. Az egyetlen osztályon értelmezett asszociáció multiplicitása ennek megfelelően csak a szerepekhez kötötten értelmezhető.

A 3.22. ábrával illusztrált példánkban egy munkahelyet modellezünk, ahol a főnök munkát ad a beosztottjainak. Mindketten alkalmazottak, de a *munkát-ad* relációban különböző szerepeket játszanak, amelyekhez tartozó számosság különböző. Egy főnöknek több beosztottja, de egy beosztottnak csak egyetlen munkaadó főnöke van.

Egy szerep az asszociáció egyik vége, amelynek nevét az ábrán is feltüntethetjük. Alkalmazása akkor kötelező, ha elhagyásával a reláció és különösen annak számossága nem egyértelmű. A szerep felfogható egy rejtett, származtatott attribútumnak, amelynek értéke a reláció révén hozzá láncolt objektumok halmaza.



3.22. ábra

2.3. 3.2.3. Normalizálás

Az attribútumok és asszociációk alkalmazása nem új gondolat a szoftverfejlesztésben, hiszen széles körben használják ezeket az adatbázis-kezelésben. Az adatbázis-kezelés során szerzett tapasztalatok és módszerek jól hasznosíthatók az objektum-orientált módszertanokban.

Az adatbázis-kezelés célja, hogy a világ dolgairól (entitásokról) és azok tulajdonságairól olyan maradandó értéket képező információ-gyűjteményt hozzanak létre, amelyre alkalmazások széles köre építhető. Ennek megvalósítására az adatokat fájlrendszerekbe szervezik és lekérdező nyelvek segítségével az adatok visszakereshetők. Hamar nyilvánvaló lett, hogy az adatok tárolásának szervezettsége alapvetően meghatározza az adatbázis-műveletek hatékonyságát. Az adatok "helyes" szervezésére vonatkozóan szabályrendszert (normál formák) alkottak, a szabályok alkalmazását normalizálásnak nevezték. Ezen szabályok egy része a fájlszervezés egyszerűsítését szolgálja. A szabályok másik része abból a felismerésből származik, hogy akkor lehet a hatékonysági és a konzisztencia követelményeknek megfelelni, ha az egymással szemantikailag szorosan összefüggő (kohézív) adatokat együtt tartják, a függetleneket pedig elkülönítik, továbbá lehetőleg elkerülik ugyanazon információ többszörös tárolását.

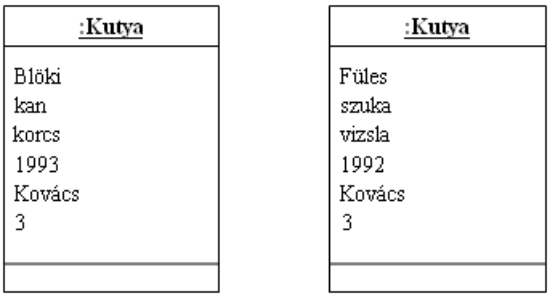
Az objektum attribútumainak szemantikailag összetartozó adatoknak kell lenni. Ez a tény teszi lehetővé és szükségessé, hogy az adatbázis-kezelés normalizálási szabályait felhasználjuk az objektumok atribútumai között fennálló függőségek elemzésére.

Az alábbiakban egy példán keresztül megvizsgáljuk, milyen veszélyekkel jár, ha egy objektum szemantikailag össze nem tartozó, vagy lazán összetartozó adatokat tartalmaz.

3.23. ábra

A korábban gyakran használt kutya-ember viszony megadásán változtattunk. Lásd a 3.23. ábrát. A *Kutya* objektumunkba attribútumként vettük fel a *gazdát*, és a gazda *gyerekeinek számát*. Vizsgáljuk meg a változtatásunk következményeit.

Induljunk ki abból, hogy egy kutyának egyetlen ember a gazdája, de egy embernek több kutyája is lehet (3.24. ábra).



3.24. ábra

Az attribútumokat elemezve arra a megállapításra jutunk, hogy a gazda gyerekeinek száma nem illik a táblázatba. Ugyanis azt a tényt, hogy Kovácséknak három gyerekük van, a hozzájuk tartozó valamennyi kutya objektum tartalmazza. Ha születik egy negyedik gyerek, akkor ezt Kovácsék valamennyi kutyájával "közölni kell". Ha ezt nem tesszük meg, akkor Kovácsék gyermekeinek számára vonatkozóan különböző értékeket kapunk, attól függően, hogy melyik kutyát "kérdezzük". Ha Kovácsék valamennyi kutyájukat elajándékozzák, akkor nem lesz olyan kutya, amelyiknek Kovács a gazdája, következésképp elveszett az az ismeret is, hogy Kovácséknak három gyerekük van.

A probléma oka az, hogy a *Kutya* objektumba az állattól teljesen független, a gazdájára jellemző attribútumot akarunk fölvenni. A gazda gyermekeinek száma nem a kutya közvetlen jellemzője, hanem attól csak a gazdán keresztül függ (tranzitív függés). A megoldás is nyilvánvaló: egy önálló *Gazda* objektumosztályt kell létrehozni és a gazda jellemzőit (gyerekek száma) ott kell tárolni.

Példánkban a *gazda-kutya* reláció *egy-több* típusú. Ez esetben egyszerűbb felismerni a téves értelmezés következményeit, mint *egy-egy* reláció esetében.

Általánosan megfogalmazhatjuk azt a szabályt, miszerint egy objektum attribútumait úgy célszerű megválasztani, hogy az attribútumok az objektum egészére legyenek jellemzőek. Amennyiben egy attribútum csak az objektum egy részére vagy csak valamely másik attribútumára jellemző, akkor érdemes az "erősebben" függő attribútumokat külön objektumként modellezni.

2.4. 3.2.4. Öröklés

Az öröklés (inheritance) olyan implementációs és modellezési eszköz, amelyik lehetővé teszi, hogy egy osztályból olyan újabb osztályokat származtassunk, amelyek rendelkeznek az eredeti osztályban már definiált tulajdonságokkal, szerkezettel és viselkedéssel.

Az öröklés az objektum-orientált programozók gondolatvilágában a programkód újrahasznosításának szinonimája. A rendszer modellezését követően a fejlesztők megvizsgálják a létrejött osztályokat és a

hasonlókat – elsősorban a kódolási munka csökkentése érdekében – összevonják. Gyakori eset, hogy a kód, vagy annak egy része korábbi munkák eredményeként, vagy előregyártott osztálykönyvtárak formájában már rendelkezésre áll. A fejlesztőnek "csak" annyi a dolga, hogy a meglevő osztályok módosításával az újakat előállítsa. Természetesen a legfontosabb haszna az öröklésnek a rendszer fogalmi egyszerűsödése és tisztasága, amely a független komponensek számának csökkenésében nyilvánul meg.

Az öröklés alapfogalmainak ismertetésekor néhány olyan implementációs problémára is kitérünk, amelyek ismerete már a modellezési fázisban sem felesleges, mivel segítenek annak megítélésében, hogy az örökléből származó előnyöket milyen áron érhetjük el.

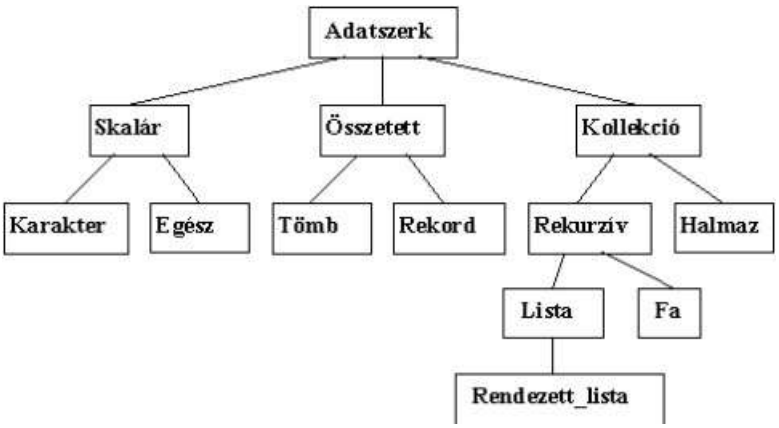
2.4.1. 3.2.4.1. Az öröklés alapfogalmai

A korábbiakban az osztályról többek között azt állítottuk, hogy a típus implementációjaként is felfogható. Ebben a felfogásban az öröklést tekinthetjük a *kompatibilitás* reláció implementációjának. Az öröklés révén valamely osztály örökli egy másik osztály viselkedését és struktúráját. Az osztály annyiban különbözik a típustól, hogy az előbbi definiálja az objektum szerkezetét is. Ugyanez a különbség az öröklés és a kompatibilitás között. Azt az osztályt, amelyből örökölünk, **alaposztálynak** (superclass) nevezzük. Az az osztály, amelyik örökli a struktúrát és a viselkedést, a **származtatott osztály** (subclass).

Az öröklés jelölésének legelterjedtebb formái láthatók a 3.25. ábrán, ahol a *Sokszög* osztályból származtatunk egy *Háromszög* és egy *Négyszög* osztályt, amelyek mindegyike örökli a *Sokszög* szerkezetét és viselkedését.

3.25. ábra

Az alaposztály – származtatott osztály reláció tranzitív öröklési hierarchiát definiál.

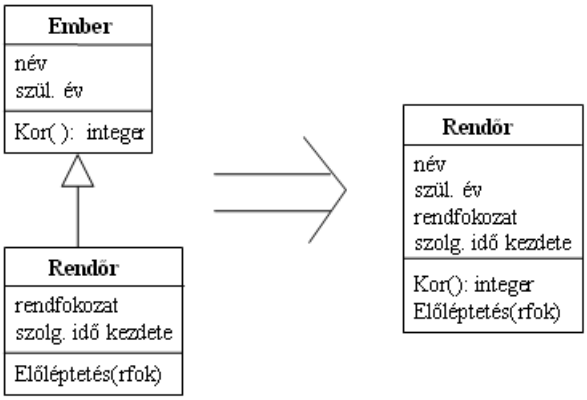


3.26. ábra

Az öröklési hierarchiában egy adott osztály fölötti valamennyi alaposztályt az adott osztály ősének (ancestor) tekintjük. Egy adott osztályból közvetlenül vagy közvetve származtatott valamennyi osztályt leszármazottnak (descendent) nevezzük. A 3.26. ábrán bemutatott példánkban a *Lista* ősei a *Rekurzív*, a *Kollekció* és az *Adatszerk* osztályok, leszármazottja pedig a *Rendezett_lista*. A *Rekurzív* osztály leszármazottjai a *Lista*, a *Fa* és a *Rendezett_lista*.

Hangsúlyoznunk kell, hogy az öröklés egy mechanizmus, amely egy alaposztály – származtatott osztály relációt *implementál*, Az öröklés mögött álló relációt **általánosításnak** (generalization), vagy **specializációnak** (specialization) nevezzük.

Az öröklés révén származtatott osztály örökli az alaposztály változóit és metódusait.



3.27. ábra

A 3.27. ábra bal oldalán jelölt öröklés hatása ugyanaz, mintha a jobb oldalon álló *Rendőr* osztályt definiáltuk volna. Figyeljük meg, hogy a *Rendőr* osztály azokat az új attribútumokat és metódusokat használja, amelyekkel *bővíteni* akarjuk az *Ember* alaposztály attribútumait és metódusait.

Az általánosítás relációt gyakran olyan alaposztályok létrehozására használjuk fel, amelyek célja kizárólagosan az, hogy az öröklés révén átadják az attribútumaikat és a metódusaikat a származtatott osztályoknak. Ezen osztályok nem példányosodnak, azaz nem képződik belőlük objektum, bár erre lenne lehetőség. Az ilyen osztályokat **absztrakt osztályoknak** nevezzük.

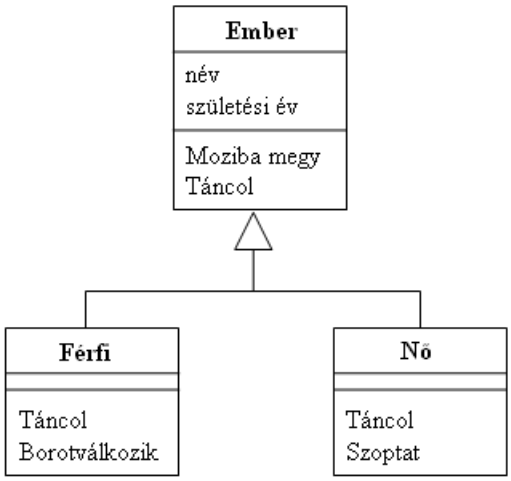
Egy származtatott osztályban az öröklött metódusok halmaza nemcsak bővíthető, hanem az öröklött metódus **át is definiálható**. Az új definíció finomítja és helyettesíti az öröklött metódust. Absztrakt osztályok esetén tipikus megoldás (ez az absztrakt osztályok leggyakoribb használata), hogy csak a metódus fejléce definiált, a metódus algoritmusának definiálása a származtatott osztályban történik. Ha az ilyen absztrakt osztályból konkrét objektumpéldányt hoznánk létre, akkor az a hiányzó algoritmus miatt a gyakorlatban használhatatlan lenne.

Az elmondottak szemléltetésére tekintsük a következő példát:

Tegyük fel, hogy férfiak és nők viselkedésének egy szeletét kívánjuk modellezni. A *Férfi* és *Nő* osztályú objektumok fölött általánosíthatjuk az absztrakt *Ember* objektumot, azzal a céllal, hogy belőle örököltessünk (3.28. ábra). Ez egybevág mindennapi tapasztalatunkkal és fogalomhasználattal, hiszen férfiak és nők sok mindenben hasonlóak és hasonlóan viselkednek. Ugyanakkor egy konkrét ember – a kevés kivételtől eltekintve, amelyek most amúgy is kívül esnek a modellezendő rendszer határain – vagy férfi, vagy nő. A modellben csak a legszükségesebb általános attribútumokra (név, születési év) van szükség, a modellezendő viselkedések pedig néhány általános emberi (*Moziba megy, Táncol*), valamint néhány határozottan nemhez kötött (*Szoptat, Borotválkozik*) viselkedésmintára terjednek ki.

A *Moziba megy* viselkedésben a férfiak és nők érdemben nem különböznek egymástól, ezt a tevékenységet ugyanúgy hajtják végre. Így ezt a metódust definiálhatjuk az *Ember* absztrakt osztályban, örökölheti mindkét származtatott osztály, nem is kell átdefiniálniuk.

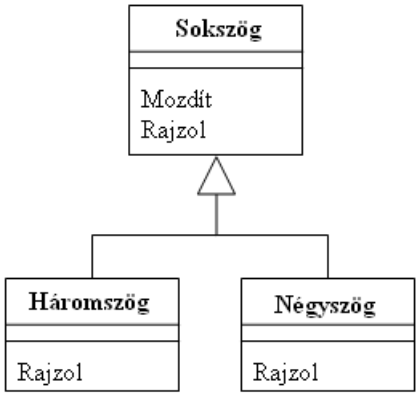
A *Táncol* viselkedéssel már más a helyzet. Mindkét nem táncol, de tánc közben eltérő a viselkedésük, hiszen más-más lépéseket hajtanak végre. Ezért az *Ember* osztályban a *Táncol* csupán fejlécként, definíció nélkül szerepel, jelezve, hogy valahogyan minden ember táncol, de a viselkedés konkrét definícióját, illetve az átdefiniálását a származtatott *Férfi* és *Nő* osztálytól várjuk.



3.28. ábra

A két nemhez kötött viselkedést, a *Szoptat* és a *Borotválkozik* műveleteket értelmetlen lenne az általánosított *Ember* osztályban mégcsak jelölni is, hiszen ezek nem általánosan elvárt emberi tevékenységek. Ezekkel a metódusokkal a megfelelő származtatott osztály bővíti az öröklött metódusokat.

Az öröklés és a polimorfizmus lehetővé teszi úgynevezett **virtuális metódusok** definiálását, ami tovább javítja az újrahasználhatóság esélyét. Példánkban induljunk ki a 3.29. ábrán látható osztályszerkezetből.



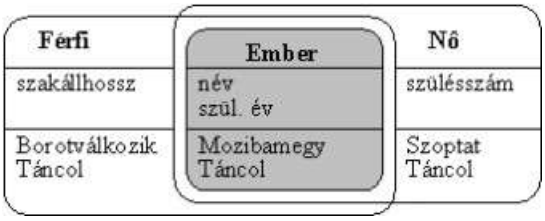
3.29. ábra

Az objektumszerkezetet sokszögek képernyőre történő felrajzolására és mozgatására hoztuk létre. A két származtatott osztályban a rajzolás metódusai különböznek egymástól, mivel eltérő adatokból eltérő grafikus elemeket kell előállítani. Ezért mindkét osztályban definiáltuk az alakzatfüggő *Rajzol* metódust. Ez esetben a *Sokszög* osztályban a *Rajzol* definiálása feleslegesnek tűnik. Más a helyzet a *Mozdít* művelettel. Mindkét konkrét síkidomnál ugyanúgy kell elvégezni az elmozdítást. Elsőként végre kell hajtani egy *Rajzol* műveletet a háttérszínnel, azután megváltoztatni a síkidom pozícióját, majd az új pozícióban ismét *Rajzol*ni kell a tinta színével. Mivel bármilyen sokszöget ugyanígy tudnánk elmozdítani, a *Mozdít* metódust érdemes a *Sokszög*ben definiálni és onnan örökölni. A *Sokszög* objektumban definiált *Mozdít* metódusban tehát hivatkozni kell a *Rajzol* metódusra. De melyik objektum metódusára? Erre azt tudjuk válaszolni, hogy annak az osztálynak a *Rajzol* metódusára, amelyik örökli majd a *Mozdít*ot. Igen ám, csakhogy a *Sokszög* definiálásakor nem tudhatjuk, hogy kik lesznek a leszármazottak. A probléma megoldására definiálunk egy *virtuális Rajzol* metódust a *Sokszög*ben azzal a jelentéssel, hogy a *Sokszög*ben szereplő *Mozdít* metódusban előforduló *Rajzol* metódust futás közben az éppen aktuális konkrét osztály azonos nevű metódusával kell helyettesíteni. Mivel a konkrét metódusnak a virtuális metódusra hivatkozó metódushoz való kötése nem fordítási időben (korán), hanem futási időben (későn) jön létre a kapcsolatot **késői kötésnek** (**late binding**) is szokás nevezni.

Természetesen ez a jelenség az osztály-hierarchia bármely részén, egyidejűleg több szinten is előfordulhat is. A virtuális metódusokra történő hivatkozáskor a megfelelő konkrét metódus aktivizálása úgy történik, hogy

mindig az üzenetet vevő objektum osztályától indulva, az ősök irányába haladva az öröklési fán, a legelsőként talált metódust elindítjuk.

További sajátos implementációs problémákat vet fel az objektumváltozók megvalósítása. A problémák forrása az, hogy alaposztályú változó származtatott osztálybeli értéket is felvehet. Adódik a kérdés, hogy az alaposztályú változónak mekkora helyet kell foglalni, ha ebbe a változóba származtatott osztálybeli objektumot is kerülhet. Az öröklés révén ugyanis lehetővé válik a struktúra és a metódusok bővülése, ami miatt a származtatott osztály objektumai általában nagyobb tárterületet igényelnek, mint az alaposztálybeliek. A kompatibilitási szabályoknak megfelelően a bővebb (származtatott osztályú) objektummal lefedhetjük a szűkebb alaposztálybeli objektumot. Kérdés, hogy mi történik a bővítménnyel. A korábbi *Ember-Férfi-Nő* példánkat egészítsük ki a *Férfinál* egy *szakállhossz*, a *Nőnél* pedig egy *szülésszám* attribútummal, a halmazábrát felrajzolva (3.30. ábra) jól látható az alap- és származtatott osztályok viszonya.



3.30. ábra

A probléma kezelésére három lehetőségünk van.

1. Az *Ember* osztályú változót statikusan (fordítási időben) definiáljuk, így az csak a saját osztályának megfelelő attribútumoknak és metódusoknak tart fenn helyet. Ha egy ilyen változóba mondjuk egy *Férfi* osztályú objektumot teszünk, akkor nincs helye a szakállhossz attribútumnak és a *Borotválkozik* metódusnak. A *név* és a *szül. év* attribútumok azonban felveszik a *Férfi* objektum által hordozott értéket, de komoly dilemma elé kerülünk a *Táncol* metódust illetően. Ha a változónk átveszi a *Férfi*ben definiált metódust, akkor az hibás működést eredményezhet. Ugyanis nem tudunk védekezni az olyan esetek ellen, amikor a *Férfi*nél átdefiniált *Táncol*ban a *Férfi* speciális attribútumaira hivatkozunk, mondjuk minden lépésnél növeljük a *szakállhossz*t. Vagyis ha az alaposztály metódusát helyettesítjük a származtatott osztályéval, akkor előfordulhat, hogy abban hivatkozhatunk olyan attribútumra vagy metódusra, amely bővítményként az értékadásnál elveszett. Ha minden bővítményünket elveszítjük és az átdefiniált metódusok sem helyettesítődnek, akkor változónkba valójában nem tudtunk származtatott osztályú objektumot tenni. Ezzel a megoldással a kompatibilitás által megengedett lehetőségeket nem tudjuk kihasználni.

2. Most is *Ember* osztályú változót definiálunk statikusan, de helyet hagyunk az összes származtatott osztály lehetséges bővítményeinek. Ez jó gondolatnak tűnhet, de valóságban kivihetetlen. Csak a teljes program ismeretében tudjuk ugyanis megmondani, hogy egy adott osztályból még kik származnak, azoknak mekkora helyet kell fenntartani. Ehhez a fordítóprogramnak és szerkesztőnek fel kell göngyölítenie a teljes öröklési struktúrát. Még ha rendelkezésünkre is állna egy megfelelő fejlesztő eszköz, az vélhetően a legrosszabb esetre készülne így erősen helypazarló lenne.

3. *Ember* osztályú dinamikus (futás közben definiált) változókat használunk. Ebben az esetben a tényleges szerkezeteket mutatókon keresztül érjük el így a kompatibilitás vizsgálata csak a pointerek közötti értékadásra terjed ki.

Összefoglalva megállapíthatjuk, hogy statikusan definiált változókkal a probléma gyakorlatilag nem kezelhető. Amennyiben alaposztályból származó változóba származtatott osztálybeli objektumot akarunk helyezni, akkor dinamikus változókat kell használnunk.

Természetesen előfordulhatnak olyan rendhagyó esetek is, amikor a fordított problémával kerülünk szembe, azaz származtatott osztályú változóba alaposztályú objektumot kívánunk tenni. Ekkor a "kisebb" objektumot el tudjuk helyezni a változóban, azonban a gond ott van, hogy bizonyos attribútumok és/vagy metódusok definiálatlanul maradnak. Ez általánosságban nem megengedhető, de különös körültekintéssel, a C nyelvből ismert *cast*-olást alkalmazva megtehető.

2.4.2. 3.2.4.2. Az öröklés veszélyei

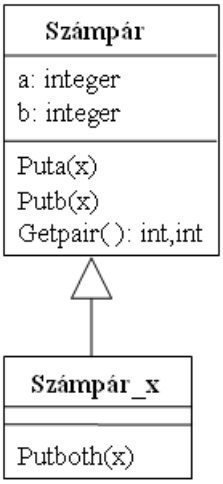
Az öröklési hierarchiából következik, hogy egy objektum egyszerre példánya saját osztályának és valamennyi ős-osztályának. Következésképp valamennyi, az ősök viselkedését megtestesítő metódusnak értelmezettnek kell lenni a származtatott osztályban. Valamely leszármazott osztály az ősök valamennyi attribútumát is tartalmazza.

Örökléskor lehetőségünk van az ősöktől örökölt struktúra és viselkedés bizonyos vonatkozásainak megváltoztatására, nevezetesen

- új attribútumok hozzáadására,
- új metódusok hozzáadására,
- örökölt metódusok átdefiniálására.

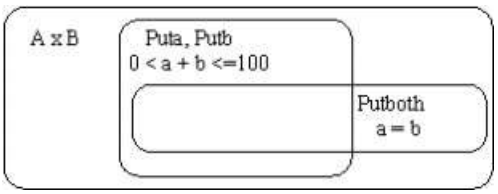
Hogyan tudjuk elkerülni annak veszélyét, hogy egy metódus átdefiniálásával a régi néven olyan új viselkedést hozunk létre, amelyik már nem tartja meg az ősosztályra jellemző kereteket, azaz nem mondhatjuk többé, hogy az objektum speciális esete az ősosztálynak?

A felvetett problémát egy konkrét példával illusztráljuk (3.31. ábra).



3.31. ábra

Legyen egy *Számpár* osztályunk, amelyik az ábrán látható attribútumokkal és metódusokkal rendelkezik. A *Számpár* osztálytól elvárjuk, hogy tegyen eleget annak az általunk felállított szabálynak, miszerint, hogy az *a* és a *b* attribútumok összege mindig 0 és 100 közé esik. A *Puta* és a *Putb* műveletek ennek a szabálynak megfelelően viselkednek. A *Számpár*ból származtatunk egy *Számpár_x* osztályt, amely *a* és *b* azonos értékre történő beállításához bővíti a metódusok készletét egy *Putboth* metódussal. Azonban a *Számpár_x*-ben a *Putboth* metódus révén új szabályt vezetünk be, *a*-t és *b*-t ugyanarra az értékre állítjuk. A *Putboth* metódust 0 és 50 közötti argumentummal végrehajtva mind az ősben, mind pedig a származtatott osztályban érvényes szabálynak eleget teszünk, viszont 50 fölötti értéknél megszegjük az ősre érvényes szabályt. Ugyanakkor, hiába felel meg a *Számpár_x* osztályú objektum aktuálisan mindkét szabálynak, az örökölt *Puta* és *Putb* végrehajtásával a *Számpár_x*-ben értelmezett szabályt szeghetjük meg. Az alábbi halmazábrán (3.32. ábra) az A x B (a ' A és b ' B) Descartes szorzaton, mint alaphalmazon, feltüntettük a metódusok értékkészletét.



3.32. ábra

Vegyük észre, hogy a problémát többek között az okozza, hogy a *Számpár* attribútumai között fennálló összefüggés betartását kénytelenek voltunk az implementáció során a metódusokban realizált algoritmusokra átruházni. Ezen algoritmusok implementálása azonban lehetőséget ad a megengedett értékkészletből való

kilépésre. Ilyen helyzettel azonban lépten-nyomon szembe kell néznünk, hiszen minden implementáció a létező programnyelvek típuskészletével kell, hogy dolgozzon, ezek pedig a példában felvetetthez hasonló korlátozásokat hosszadalmas futási idejű ellenőrzésekkel tudják csak megtartani.

A megoldáshoz úgy juthatunk közelebb, ha az öröklés során biztosítjuk, hogy a származtatott objektum új és átdefiniált metódusai az örökölt attribútumoknak pontosan arra a halmazára képezzenek le, mint az ősök metódusai. *Ezt legegyszerűbben úgy érhetjük el, ha nem definiálunk át örökölt metódust (az absztrakt osztálytól örökölt "üres" metódusok kivételével), és az új metódusok, csak az új – nem örökölt – attribútumokat változtatják meg (természetesen az örökölteket olvashatják).*

A probléma általánosításaként elmondhatjuk, hogy az osztályhoz tartozás elviekben két módon definiálható: szabályokkal vagy az osztály elemeinek felsorolásával. A szabály alapú definiáláskor a szabályok alkalmazásával eldönthető, hogy egy objektum az adott osztályhoz tartozik-e vagy sem. Ez a módszer jól működik például a matematikában. A matematika objektumait (például a kúpszeleteket) szabályok definiálják. A szabályok alapján történő definiálás jól kézben tartható ott, ahol az objektumok attribútumai nem változnak. Ezzel szemben az objektumok attribútumait gyakorta változtatjuk. Valahányszor változtatunk mindannyiszor szükség lenne a szabályok fennállásának vizsgálatára és az olyan műveleteket, amelyek kivezetnek a szabályok közül, tiltani kellene.

Ha az öröklés során, amikor olyan új metódusokat definiálunk vagy átdefiniálunk, amelyek az örökölt attribútumok értékeit változtatják, akkor minden esetben ellenőrizni kell, hogy az ősökben alkalmazott szabályokat nem sértjük-e meg. Amennyiben az öröklés során új – a meglevő szabályokat szűkítő értelmű – szabályokat állítunk fel, akkor valamennyi, az ősökben definiált metódusra meg kell vizsgálni, hogy azok nem sértik-e az új szabályokat.

Összefoglalva megállapíthatjuk, hogy az öröklést bátran alkalmazhatjuk, amennyiben minden egyes metódus definiálásakor alaposan megvizsgáljuk, hogy a metódus végrehajtását követően az osztályhoz tartozás feltételei fennmaradnak-e az objektumra és minden ősére vonatkozóan. A napi programozási gyakorlatra nem jellemző a szigorú vizsgálódás. A programozók hajlamosak kellő körültekintés nélkül átdefiniálni az örökölt metódusokat, főként a következő esetekben:

Kiterjesztés. Az átdefiniált metódusnak az attribútumokra gyakorolt hatása ugyanaz, mint az ősnél, azonban azt néhány új – általában az új attribútumokra vonatkozó – tulajdonsággal bővíti. Példaként legyen egy *Kör* osztályunk, amelyen értelmezett *Rajzolj* operáció a képernyőre kört rajzol. Ebből származtatjuk a *KözpontosKört*, amelynek a *Rajzolj* metódusa a kör közepét egy kereszttel jelöli. Ehhez úgy definiáljuk át a *Rajzolj* metódust, hogy abban először meghívjuk a *Kör* osztály *Rajzolj* műveletét, majd kirajzoljuk a keresztet.

Szűkítés. Az átdefiniált metódus a bemeneti paraméterek halmazát korlátozza, általában a típus szűkítésével. Ez csak akkor lesz korrekt, ha a fenti szabályok értelmében sem az új, sem a régi metódusok nem vezetnek ki a szűkített paraméterhalmazból. Példaként tekintsünk egy *Lista* osztályt, amelyen értelmezett az *Add(objektum)* művelet. A származtatott *IntegerLista* esetében az *Add(integer)* korlátozás megengedett és szükséges is.

Módosítás. Az új metódus interfész szinten és szemantikailag is tökéletesen megegyezik a régivel, a metódusban megvalósított algoritmus eltérő. Az ilyen módosítás célja általában a teljesítmény növelése és a hatékonyság fokozása. Példának vehetünk egy olyan osztályt, amelyen átdefiniáljuk a *Keresés* művelet algoritmusát, és lineáris keresés helyett bináris keresést alkalmazunk.

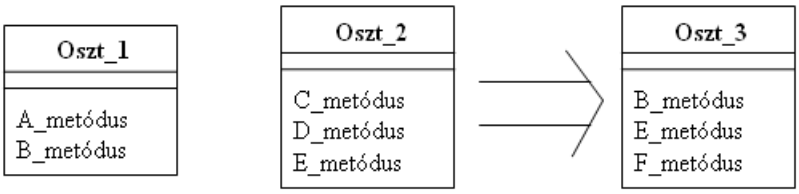
Befolyásolás. Tipikus, nem mindig helyeselhető programozói gyakorlat. Befolyásolással úgy származtatunk új osztályt, hogy a rendelkezésünkre álló osztályokat átvizsgálva olyan osztályt keresünk, amelyik többé-kevésbé hasonlít a létrehozni kívánt osztályhoz. Ebből a "hasonló" osztályból örököltetjük az új osztályt, átdefiniálva a metódusokat, tekintet nélkül a metódusoknak az ős osztályokban definiált szerepére. Ezen öröklési mechanizmus révén kiadódó objektumszerkezet komoly implementációs és karbantartási nehézséget tud okozni, mivel a kellően át nem gondolt örökítés során könnyen elveszítjük az áttekintésünket a kiadódó bonyolult objektum-hierarchia fölött. A módszer használatát csak végszükségben tartjuk elfogadhatónak.

2.4.3. 3.2.4.3. Többszörös öröklés

Amikor olyan új osztályt kívánunk definiálni, amely két vagy több meglevő osztályra épül, akkor **többszörös öröklésről** beszélünk. Többszörös öröklésnél egy osztálynak több közvetlen őse van. A többszörös öröklés bonyolultabb, mint az egyszeres öröklés, mivel egyszeres öröklésnél az öröklési struktúra faszerkezetű, ezzel szemben többszörös öröklésnél hálós szerkezeteket is kaphatunk. A többszörös öröklés előnye a modellezés

kifejező erejének növekedése – mivel közelebb kerülünk az emberi gondolkodáshoz – és az újrahasználhatóság lehetőségének bővülése. Hátránya, hogy az implementáció nehezebbé válik, és a fogalmi tisztaság csökkenhet.

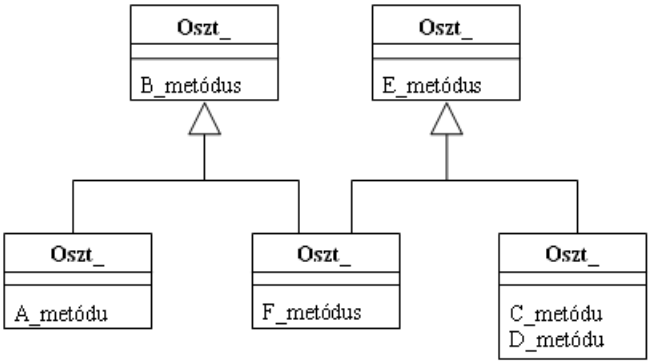
A legegyszerűbb eset az, amikor egy alaposztályból csak egyetlen úton lehet örökölni a származtatott osztályt, mint ahogy az a 3.33. ábrán bemutatott példában látható. Tételezzük fel, hogy van két osztályunk (*Oszt_1*, *Oszt_2*), amelyek a metódusaikkal adott viselkedéssel jellemezhetők. Célunk, hogy ezen osztályok megtartásával olyan öröklési struktúrát készítsünk, amelynek révén származtatott *Oszt_3* viselkedése bizonyos vonatkozásait a nevezett két őstől örökli.



3.33. ábra

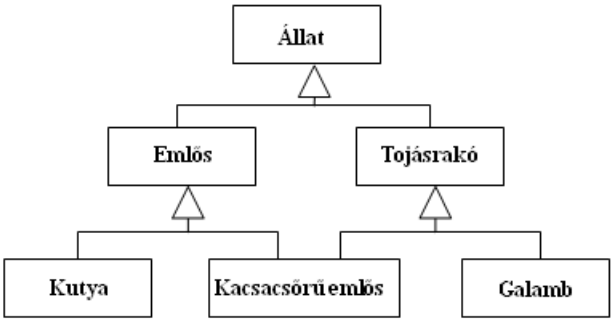
A lehetséges megoldás az, hogy a közösen használt metódusokat megtestesítő új osztályokat definiálunk, amelyekből egyedi metódusokkal kibővítve származtathatjuk a kívánt tulajdonságú osztályokat (3.34. ábra).

Természetesen a többszörös öröklés használatának központi kérdése, hogy valóban a metódusok által reprezentált tulajdonságokat kívánjuk-e örökölni, vagy csak a meglevő kódot akarjuk hasznosítani. Nyilvánvaló, hogy a csónakházat nem lehet a csónakból és a házból örököltetni. A ház tulajdonságainak öröklése még elképzelhető (csónakház *is_a* ház), de a csónaké már kevésbé (csónakház *is_a* csónak ????).



3.34. ábra

A többszörös öröklés bonyolultabb – nagyobb figyelmet igénylő – esete, amikor egy származtatott osztályt ugyanabból az alaposztályból több különböző úton is származtatunk (ismételt öröklés).

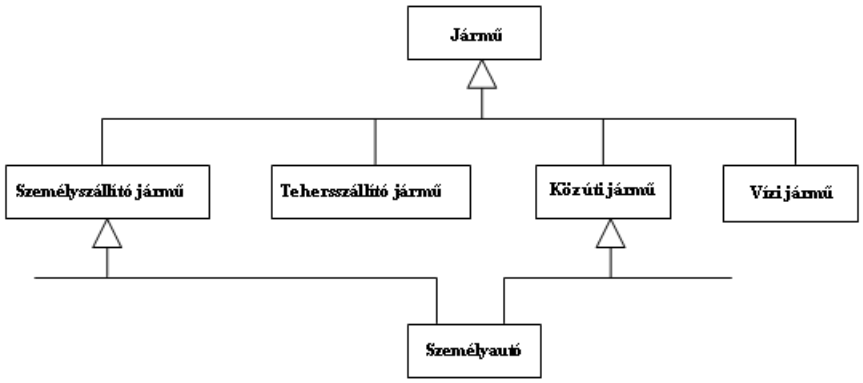


3.35. ábra

A 3.35. ábrán a *Kacsacsőrű emlős* tulajdonságai mind az *Emlős*, mind pedig a *Tojásrakó* osztályokon keresztül öröklődnek. Szemantikailag azzal az esettel állunk szemben, hogy egyazon szempont (szaporodás módja)

szerinti osztályozás eredményeként kapott származtatott osztályok (*Emlős* és *Tojásrakó*) nem elkülönültek. Ilyenkor átlapolódó származtatott osztályokról beszélhetünk.

Az előző esettől lényegesen eltérő problémával találkozhatunk, ha a származtatás különböző szempontok alapján történik és a célunk a különféle kombinációk előállítása. A 3.36. ábrán látható példában az egyik osztályozási szempont a szállítandó dolog jellege, a másik szempont pedig a jármű pályája. Ezen szempontok egymástól teljesen független csoportosítást tesznek lehetővé.



3.36. ábra

A többszörös öröklés egyik jelentős problémája, hogy a közös őstől ugyanazok a metódusok és attribútumok örökölhetők mindegyik öröklési ágon, ám esetleg különböző utakon különbözőképpen átdefiniálva. Legyen egy *parkol* nevű és értelmű metódusa a járműnek, amit mind a személyszállítónál, mind pedig a közúti járműnél átdefiniálhatunk, ráadásul különféleképpen. Kérdés, hogy a személyautó melyiket örökli és milyen néven. Elképzelhető, hogy mindkettőt, de akkor névazonossági probléma áll elő, ami úgy oldható fel, hogy nemcsak a nevével, hanem az egész származtatási útvonal megadásával hivatkozunk a metódusra. Példánkban ezek lehetnek *Közúti_jármű::parkol* és *Személyszállító_jármű::parkol* megnevezések. Persze az is lehetséges, hogy a különféle származtatás révén örökölt metódusok végül is ugyanazok.

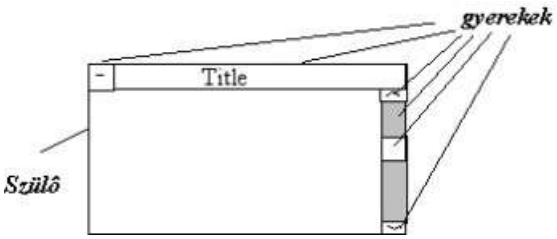
A fentiekhez hasonló probléma természetesen az attribútumokra vonatkozóan is fennáll. Ezekkel kapcsolatosan általános szabályokat nem tudunk mondani, esetenként az értelmezési problémákat kell tisztázni. Jó, ha az implementációs eszközök lehetőséget adnak arra, hogy kiválaszthassuk a számunkra megfelelő értelmezést.

Az újrahasználhatóság, mint kitűzött cél nemcsak az öröklés révén érhető el. Gyakran célszerű a **delegálás** módszerét alkalmazni, ami valójában a komponens reláció alkalmazásának egy esete.

2.5. 3.2.5. Komponens-reláció

A komponens vagy tartalmazás reláció az asszociáció egy speciális esete. Szemantikailag a rész-egész viszonyt, más szóval az aggregációt jelenti. A rész szerepét játszó objektum az egésznek komponense, alkotóeleme. Klasszikus példa az anyagjegyzék. Az autónak része a motor, a karosszéria, a fékrendszer, az elektromos rendszer, az erőátvitel, a kerekek. Az egyes tételek azonban újabb részekből állhatnak.

Sajnálatosan a komponens relációval kapcsolatos szóhasználatban is a szülő-gyermek elnevezés terjedt el. Egy objektum példány, amely komponenst tartalmaz, a komponens szülője. Egy komponens pedig gyermeke a szülőjének. A 3.37. ábrán egy ablak és néhány része van feltüntetve.

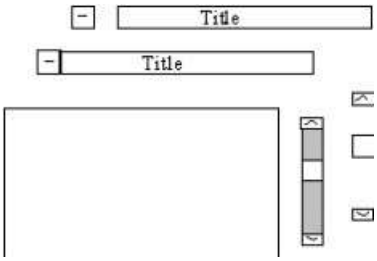


3.37. ábra

Az ablak a tartalmazás reláció mentén részekre osztható, példánkban a 3.38. ábra szerint.

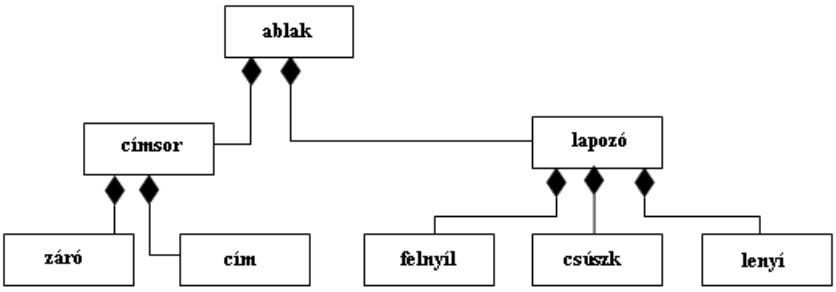
A relációra érvényes a tranzitivitás, azaz ha A része B-nek és B része C-nek, akkor abból következik, hogy A része C-nek is. A tranzitivitás miatt egy komponens-hierarchia alakul ki, ahol a magasabban álló elemek tartalmazzák az alattuk levőket.

Az objektum-orientált programozási nyelvek jelentős többsége – így a könyvben tárgyalt C++ is – jelenleg legfeljebb a metódusok késői kötését, és az osztályok konstruktoraiban az attribútumok kezdőértékére vonatkozó paraméterek átadását támogatja.



3.38. ábra

Egyetlen komponens-reláción belül csak egy szülő-gyermek viszonyt jelenítünk meg. Több különböző komponens tartalmazását, több független relációval adjuk meg. Minden egyes tartalmazás-relációra – az asszociációnál megszokott módon – előírható a számosság. A komponens-relációt a szülőhöz kapcsolt kicsiny rombusszal jelöljük (3.39. ábra).



3.39. ábra

A tartalmazás másik jellemzője, hogy antiszimmetrikus, vagyis ha A része B-nek, abból következik, hogy B nem része A-nak. Fontos tulajdonsága még, hogy a szülő bizonyos jellemzőit – módosítással vagy anélkül – átveszi a gyermek is. Egy repülőgép ajtajának kilincse a mozgásának egyes jellemzőit az őt tartalmazó ajtótól kapja, amelyik viszont ugyanezen jellemzőit az ajtót tartalmazó repülőgéptől veszi át.

Fontos különbséget tenni az öröklés és a tartalmazás relációk között. Az ajtó kilincse nem örökli a repülőgéptől az attribútumait és metódusait, hiszen a kilincs nem repülőgép. Amennyiben a gyermek objektum olyan üzenetet kap, amelyre nem tud reagálni, szükséges lehet, hogy az üzenetet továbbadja szülőjének a komponens hierarchiában felfelé egészen addig, amíg valamelyik szülő válaszolni tud, vagy elérkezünk a komponensfa gyökeréig. Amennyiben az üzenetek ilyen továbbítása megoldott, azt automatikus üzenet átadásnak nevezzük. Az ablakkezelő rendszerekben ez a folyamat játszódik le.

A 3.40. ábrán bemutatott példánkban szereplő *Levél Bekezdések*ből áll, amelynek komponensei a *Karakter*ek. Joggal kérdezhetjük, hogy egy bizonyos karakter hányadik oldalon található. Erre a karakter nem tud "válaszolni", hanem "megkérdezi szülőjét". A bekezdés sem tudja, ezért ő is a "szülőjéhez fordul".

3.40. ábra

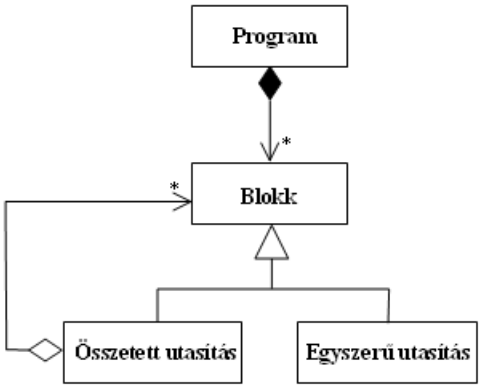
A rész-egész szemantikából következik, hogy a kapcsolatnak létezik fordítottja. Ha a levelet le akarjuk másolni, akkor az hatással van a komponensekre, hiszen a levél másolása bekezdésenként történik. A bekezdés másolása pedig a karakterekre van hatással, hiszen a bekezdéseket karakterenként másoljuk.

Annak eldöntésére, hogy egy asszociáció komponens-reláció-e éppen a "hatással van" viszonyt érdemes vizsgálni. Ha a szülőn értelmezett művelet értelmezett a gyermeken is, sőt azt azon végre is kell hajtani ahhoz, hogy a művelet a szülőn is végrehajtódjon, akkor minden bizonnyal komponens relációval állunk szemben.

A komponens reláció struktúrája lehet *rögzített*, *változó* és *rekurzív*.

A rögzített struktúra azt jelenti, hogy a komponens-objektumok száma és típusa előre definiált. Ilyen a fent bemutatott *ablak* példa. A változó struktúrában a szintek száma és az ott szereplő objektumok típusa meghatározott, a komponens-objektumok száma azonban változhat. Ilyen a fenti *Levél* objektum, ahol nem mondható meg előre a bekezdések és a karakterek száma. Rekurzív komponens-reláció esetén az objektum közvetve vagy közvetlenül tartalmazza saját magát. A lehetséges szintek száma elvileg végtelen. A 3.41. ábra egy tipikus rekurzív tartalmazása-relációt mutat be.

A *Program Blokkok*ból áll. Egy *Blokk* lehet *Egyszerű* vagy *Összetett utasítás*. Az *Összetett utasítás* ugyancsak *Blokkok*ból áll.



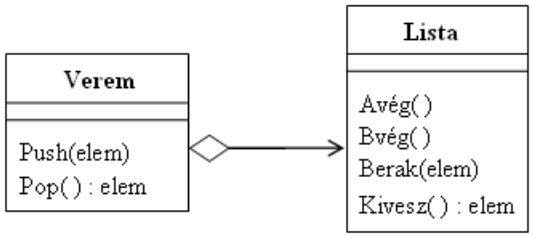
3.41. ábra

Az öröklés mellett az újrahasznosítás másik módja a *delegálás*. A delegálás annyit jelent, hogy egy objektum metódusát úgy implementáljuk, hogy a kívánt műveletet az adott objektummal komponens-relációban álló másik objektum metódusával végeztetjük el.

3.42. ábra

Tételezzük fel, hogy van egy *Lista* osztályunk (3.42. ábra), amelyen a következő műveleteket értelmezzük. Az *Avég*, *Bvég* műveleteket követően végrehajtott *Berak* és *Kivesz* műveletek a lista A illetve B végére vonatkoznak (kiválaszt). *Berak(elem)* a lista kiválasztott végét a paraméterként kapott elemmel bővíti. *Kivesz( ): elem* a lista kiválasztott végéről az ott álló elemet eltávolítja és eredményül adja.

Amennyiben egy *Verem* osztályt definiálunk *Push(elem)* és *Pop( ): elem* műveletekkel, öröklés helyett célszerű a delegálást választani, azaz egy *Lista* objektumot a *Verem* komponensévé teszünk és a *Push* és *Pop* műveleteket a külvilág számára láthatatlan és elérhetetlen *Listá*n hajtjuk végre (3.43. ábra).

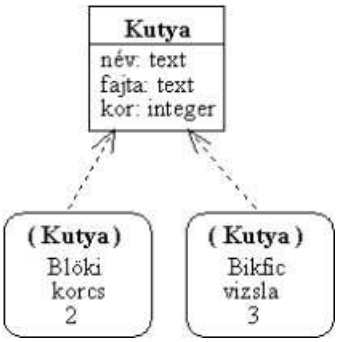


3.43. ábra

2.6. 3.2.6. Metaosztály

Az adatokra vonatkozó adatokat **metaadatoknak** nevezzük. Gyakran találkozunk ilyenekkel az életben, például szótárakat és könyvtári katalógusokat böngészve. Amikor egy olyan adattáblázatot készítünk, amelyben felsoroljuk a megyéket és a megyeszékhelyeket, mint például Baranya – Pécs, akkor *adatokat* rögzítünk. Az az információ, hogy a megyének van egy székhelye már *metaadatnak* tekinthető.

Az objektumosztály az objektumok egy halmazának leírása, következésképp metaadat. Szélesebb értelemben véve bármely minta metaadat, a minta és annak egy példája a példányosodás egy formája, a példányosodás pedig reláció. Objektum-diagramokon az osztály és a példány közötti kapcsolatot is szokták jelölni, ahol a példányosodás relációt pontozott vonal ábrázolja (3.44. ábra).



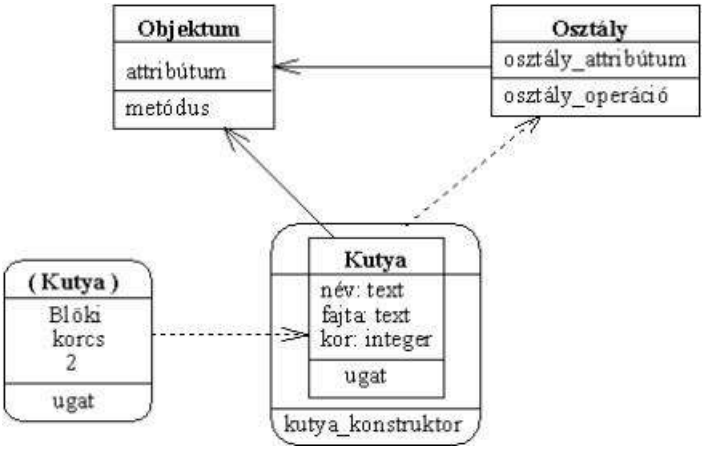
3.44. ábra

Bizonyos értelemben az objektumosztály maga is rendelkezik az objektum tulajdonságaival. Természetesen nem a modellezendő valós világ objektumaként, hanem a modellre vonatkozó *metaobjektumként*. A legtöbb objektum-orientált implementáció megengedi ugyanis, hogy objektumokat futási időben hozzunk létre, illetve semmisítsünk meg. Az osztály objektum jellegének legközvetlenebb bizonyítéka, hogy az objektumot létrehozó üzenet nem szólhat magának az objektumnak, hiszen az még nincs. Ha üzenünk valakinek, hogy "*adj_egy_ilyen_objektumot*" akkor az a valaki csakis objektum lehet, mert üzenet vételére csak objektum képes. Az objektumot leíró osztály tehát egyben objektum is. Ha objektum, akkor kell lenni olyan osztálynak, amelynek ő egy példánya. Ez az úgynevezett **metaosztály** (metaclass).

Az osztály attribútumainak tekintjük azokat az adatokat, amelyek az osztály, mint objektumgyár tevékenységével, a példányosítással kapcsolatosak. Ilyen lehet például, hogy hány példány készült egy adott objektumból. Az osztályattribútum fogalmát szélesíthetjük, ha beleértjük a létrehozandó objektumok attribútumaival kapcsolatos adatokat is. Ezzel a lehetőséggel élve az osztályt utasíthatjuk arra, hogy az általa létrehozott objektumok bizonyos attribútumainak kezdőértékét megváltoztassa, vagy változtasson a kezdőértéket meghatározó módszeren.

Az osztályműveletek azok a metódusok, amelyeket az osztály, mint objektum a hozzáküldött üzenetekre válaszul végrehajt. Egy osztály-műveletnek biztosan kell léteznie, ez általában a **konstrukciónak** (constructor) nevezett művelet, amely a példányosítást végzi. Definiálhatók olyan osztály-műveletek, amelyek a célja a példányosítással kapcsolatosan gyűjtött adatok lekérdezése, vagy az osztály által létrehozott objektumok struktúrájának vagy metódusainak vizsgálata. Célszerű lehet az attribútumok kezdőértékeinek lekérdezése is. Különösen akkor, ha olyan osztályműveletet is bevezetünk, amellyel üzenünk az osztálynak, hogy mostantól kezdve minden objektumot új kezdőértékkel hozzon létre.

A 3.45. ábrán a példányosodást pontozott, az öröklési relációt folytonos vonallal jelölve összefoglaljuk az elmondottakat. Induljunk ki a *Blöki* nevet viselő objektumból. Ez egy példány, amely a *Kutya* osztálynak a megtestesülése. Tehát a *Kutya* egyfelől osztály. Másfelől a *Kutya* objektum is, hiszen üzenetet tudtunk küldeni neki, amelynek hatására létrehozta a *Blöki* példányt. Tovább vizsgálva a *Kutyá*t, amely egyszerre objektum és osztály is, elmondhatjuk, hogy objektum minőségében neki is egy osztály példányának kell lennie. Ez az osztály az *Osztály*. Itt elvarrjuk a szálat, és nem tételezzük fel, hogy a *Kutya* osztály is egy üzenet hatására keletkezett az *Osztály*ból. Az *Osztály*ra azt mondjuk, hogy ez a metaosztály. Az *Osztály* definiálja, hogy rendszerünkben milyen osztály-attribútumokkal és osztály-műveletekkel rendelkezzenek az osztályok. Az ábrán az *Objektum* osztály jelképezi az objektum szerkezetének legáltalánosabb definícióját tartalmazó osztályt, amelynek minden más osztály leszármazottja.



3.45. ábra

A fenti példa gondolatmenetét folytatva elérkezünk egy meglehetősen sikamlós területre. Olyan modellhez juthatunk, amelyben megengedett, hogy futás közben utasítsuk az osztályunkat arra, hogy a korábbiakhoz képest eltérő attribútumokkal generálja az új objektumokat, azaz megváltoztathatjuk azt a mintát, amelynek példánya az objektum. Vagyis maga a minta is változóvá alakulhat át. A történetet folytathatjuk, úgy is, hogy olyat üzenünk, hogy ettől kezdve generáljon a kutyáknak adóazonosítót valamint bővítse a kutyák metódusait az adószám lekérdezésével, amely metódust az üzenet részeként megküldünk.

Különösen izgalmas kérdés, hogy mi annak az osztálynak a minimális művelethalmaza (viselkedéshalmaz), amely már elégséges ahhoz, hogy a futás közben kapott üzenetek alapján tetszőleges struktúrájú és viselkedésű objektumot legyen képes generálni. A kérdés a korábban már említett kötési idővel kapcsolatos. Kötési időnek tekintettük azt a pillanatot, amikor valamely programbeli elem értéke meghatározódik. Egy ilyen modellben nemcsak a metódusok, hanem már a struktúrák késői kötéséről is beszélnünk kellene.

Nyilvánvaló, hogy a fenti elven működő szoftvernek az egész programfejlesztést át kellene fognia és biztosítania a keletkező objektumok valamiféle konzisztenciáját. Ez pedig alighanem egy *CASE* (számítógéppel támogatott szoftver fejlesztés, *Computer Aided Software Engineering*) vagy azzal rokon fejlesztői eszköz lehet.

3. 3.3. Dinamikus modellek

Valamely rendszer a legegyszerűbben úgy érthető meg, ha elsőként megvizsgáljuk statikus struktúráját. Ez azt jelenti, hogy felderítjük az objektumok időtől független szerkezetét és kapcsolatát. Az objektum- vagy osztály diagramban feltüntetett asszociációk csak a kapcsolat tényét rögzítik annak időbeli vonatkozásairól, létrejöttéről, történetéről nem mondanak semmit. A rendszer időbeli viselkedését, változásait, azaz a vezérlést, a dinamikus modell írja le. A vezérlés alatt azt az információt értjük, amely megfogalmazza, hogy a rendszer az őt kívülről ért hatásokra a műveletek milyen sorrendjével válaszol, figyelmen kívül hagyva, hogy az operációk mivel mit tesznek, és milyen a kódjuk.

Jelen pontban tárgyaljuk a dinamikus működés leírásával kapcsolatos fogalmakat, a külső hatásokat megjelenítő eseményeket és az objektum állapotát állítva vizsgálódásunk középpontjába. A viselkedés leírása történhet a kommunikációs diagram és az állapot diagram vagy az állapot átmenet táblázat segítségével. Bemutatjuk azt is, hogy az állapotok és az események gyakorta hierarchikus rendbe állíthatók.

3.1. 3.3.1. Események és állapotok

Az objektum fogalmának tárgyalásakor (2.2.1. fejezet) tisztáztuk, hogy az objektum egy adott időpillanatban egyértelműen jellemezhető az attribútumai által felvett értékkel, amelyek együttesét állapotnak nevezzük. Az idő folyamán az objektumot ért hatások (**események**) eredményeként az objektum attribútumai (állapota) megváltoznak. A valós életben eseménynek nevezzük valamilyen történés bekövetkeztét. Az eseménynek egy okozója, és valahány észlelője lehet. Az objektum-orientált modellezés során kikötjük, hogy egy eseménynek pontosan egy észlelője van. Az olyan történések, amelyeket senki, még az okozója sem észlel, nem befolyásolják a rendszer működését, ezért a továbbiakban ezen történéseket kizárhatjuk az események közül. Az olyan eseteket, amikor egy eseményt több szereplő is észlel, vissza lehet vezetni több olyan eseményre, amelyek mindegyike csak egyetlen szereplőre hat. Ez történhet úgy, hogy az esemény forrása minden szereplő számára külön eseményt generál, de előfordulhat az is, hogy az eseményt észlelő szereplő továbbadja az eseményt más szereplőknek.

A forrástól az egyetlen észlelőnek küldött eseményt felfoghatjuk úgy is, hogy a forrás objektum üzenetet küld a célobjektumnak. Objektum-orientált modellezés során az esemény és üzenet szinonimaként kezelhető.

Egy eseményre az objektum az állapotától függően reagál. A reakció valamilyen, az üzenetet kapott objektum által kezdeményezett akcióban nyilvánul meg. Ez lehet egy olyan tevékenység, amelynek hatóköre kiterjed az objektum által látott belső (attribútumok) és külső (más objektumok, globális elemek) dolgokra, tehát végrehajtása során részint az objektum belső attribútumai módosulhatnak, részint újabb események keletkezhetnek.

Az objektumok viselkedésének időbeli leírásakor meg kell adnunk, hogy az objektum egy adott állapotában bekövetkező esemény hatására milyen következő állapotba jut, és milyen akciót hajt végre. Ezt a modellt véges állapotú gépnek (finite state machine) – egyszerűbben állapotgépnek – nevezzük. Az állapotgép többek között állapotdiagrammal vagy állapotátmeneti-táblázattal írható le.

Az eseményeket és azok hatását két nézőpontból is vizsgálhatjuk. Elemezhetjük azt, hogy a rendszer objektumai milyen üzeneteket, milyen sorrendben küldenek egymásnak, vagyis azt, hogy egy globális cél érdekében milyen kommunikációra kényszerülnek. Ezt tehetjük anélkül, hogy belemennénk abba, hogy az egyes objektumok belsejében milyen változások történnek a párbeszéd során. A vizsgálat eredményeit kommunikációs diagramokban foglalhatjuk össze. Az események vizsgálatának másik nézőpontjából az egyes objektumot önmagában vesszük figyelembe és áttekintjük, hogy a környezetéből érkező események hatására hogyan változtatja belső állapotát és milyen újabb üzeneteket hoz létre.

Valamely rendszer dinamikus modellje az egyedi objektumait jellemző állapotgépek és a kommunikációs diagramok összességeként adható meg. Minden egyes állapotgép önállóan, kizárólag az őt ért hatásoktól függően, a többitől függetlenül, azokkal párhuzamosan, konkurálva működik. A független állapotgépek közötti kapcsolatot a különböző objektumok között áramló üzenetek teremtik meg.

3.1.1. 3.3.1.1. Az esemény

Egy esemény egy adott időpillanatban bekövetkező történés. Eseménynek tekintjük egy lámpa bekapcsolását, a padlóra esett tányér összetörését. Az eseménynek alapvető jellemzője, hogy a történés egyetlen pillanatban, nulla idő alatt játszódik le. A valóságban semmi sem történhet nulla idő alatt, minden folyamatként játszódik le. A modell alkotása során a folyamatos történésekből mintákat veszünk, ha úgy tetszik filmet készítünk. Eseménynek nevezzük azt a történést, amely az *n.* filmkockán még nem, de az *n+1.* kockán már bekövetkezett. Nyilvánvalóan a mintavételezés sűrűségét, úgy kell megválasztani, hogy az a modellel összhangban álljon. A lámpa bekapcsolását általában tekinthetjük eseménynek, kivéve ha magát a folyamatot akarjuk jellemezni.

Két esemény logikailag megelőzheti vagy követheti egymást, illetve függetlenek is lehetnek. Az X mozi esti utolsó előadása előbb kezdődik és később fejeződik be, közöttük a sorrend egyértelmű, amit az okság határoz meg. Ezzel szemben az X és az Y mozik esti előadásainak befejezése között nincs oksági összefüggés, egymáshoz képest tetszőlegesen helyezkedhetnek el az időben, egymásra nincsen hatásuk. Ha két esemény nincs oksági kapcsolatban, akkor azokat konkurensnek nevezzük. A modellezés során a konkurens események között nem tudunk rendezési relációt definiálni, hiszen az események sorrendje tetszőleges lehet.

Minden esemény egyirányú kapcsolatot jelöl az üzenetet küldő (forrás) és a fogadó (cél) objektum között. A fogadó objektum természetesen reagálhat az üzenetre, küldhet választ, – ez csakis és kizárólag a fogadótól függ – ami viszont már külön eseménynek számít.

Minden esemény egyedi, ugyanakkor eseményosztályokba is sorolhatjuk őket közös struktúrájuk és szemantikájuk alapján. A keletkező struktúra hierarchikus, hasonlóan az objektumosztályok hierarchiájához. Az X és az Y mozik esti előadásainak befejeződését jelentő események a mozielőadás vége eseményosztály példányai. Ebben az összefüggésben az esemény-fogalom nem egyértelmű. Ugyanis az esemény egyaránt jelentheti az eseményosztályt és annak egyetlen példányát. A gyakorlatban a két eset a környezete alapján könnyen felismerhető. Az eseményeknek lehetnek attribútumai, mint például esetünkben a mozi neve. Az esemény megtörténtének időpontja minden esetben az esemény implicit attribútuma.

Az eseményt megjelenítő üzenetnek az időponton kívül egyéb paraméterei is lehetnek. Például a mozielőadás végét jelző üzenetnek paramétere lehet a játszott film címe és/vagy a nézők száma. Az alábbiakban felsoroltunk néhány eseményt és azok attribútumait:

vonat érkezett (vonatszám, pályaudvar),

mozielőadás vége (a mozi neve, a film címe),

billentyű leütése (a billentyű jele),

a motor leállt (az autó rendszáma),

a telefonvonal bontott.

3.1.2. 3.3.1.2. Kommunikációs modell

Egy rendszer működése jól definiált eseménysorozatokkal, **forgatókönyv**ekkel (*scenario*) jellemezhető. A forgatókönyvek feladata a rendszer külső interfészén történő események, vagy bizonyos objektumokkal kapcsolatos tipikusan előforduló események sorozatának leírása.

Az alábbiakban láthatjuk a már korábban említett virágküldő szolgáltatás igénybevételének és fizetésének forgatókönyvét.

Az ügyfél szolgáltatást rendel (címzett, idő, virágok)

A virágos virágot küld (idő, virágok)

A címzett átveszi a virágot

A virágos számlát küld (hivatkozás, összeg)

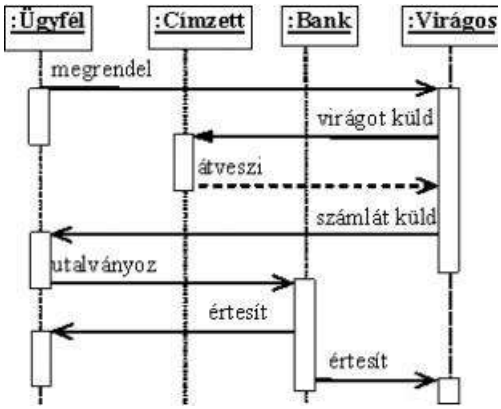
Az ügyfél utalványoz (számlaszám, összeg, hivatkozás)

A bank terhelésről értesít (összeg, dátum)

A bank követelésről értesít (összeg, dátum, hivatkozás)

A feltüntetett események sorrendje természetes oksági rendet tükröz. Mindegyik esemény egy üzenet küldését jelenti az egyik objektumtól a másikig. Ebből adódóan egyértelműen rögzíteni kell, hogy melyik objektum az üzenet feladója és melyik a címzettje. Az eseménysorrend és az üzenetkapcsolatban álló objektumok ábrázolandók a kommunikációs diagramon. A 3.46. ábrán mindegyik objektumot egy függőleges vonal jelképezi. Az objektumok közötti üzeneteket a megfelelő objektumokat összekötő nyíllal ellátott vízszintes vonallal jelezzük. Az idő múlását az ábrán a fentről lefelé haladás jelenti.

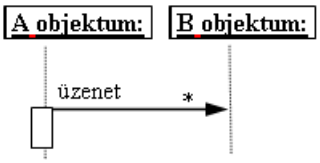
A 3.46. ábrán a virágküldéssel kapcsolatos kommunikációs diagram látható.



3.46. ábra

Alapértelmezés szerint az idő-tengely nem léptékezett, azaz az üzeneteket jelölő vízszintes vonalak közötti távolság nem utal az üzenetek között eltelő időre, pusztán a sorrendiséget jelzi. Amennyiben az üzenetek időzítésére vonatkozó előírásokat is ábrázolni kívánjuk – például valós idejű (*real-time*) rendszerek esetében –, felvehetünk skálát az időtengelyen

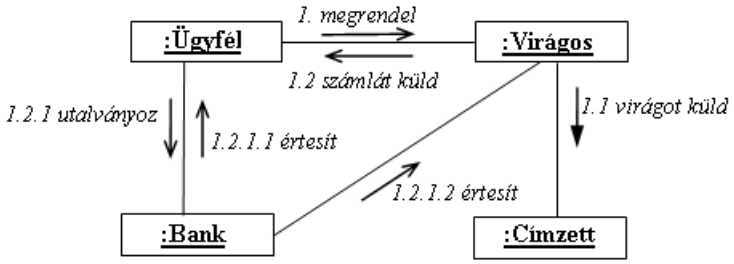
A diagram kifejező ereje elég gyenge, mivel az üzenetek között csak az "egymásra következés" (szekvencia) relációt tudjuk megjeleníteni. A valóságban gyakorta előfordul, hogy egy üzenetet, vagy egy üzenetcsoportot meg kell ismételni (iteráció). Az egyetlen üzenet ismétlésének jelölésére szokás az ismétlődő üzenetet csillaggal (*) megjelölni, mint ahogy azt a 3.47. ábrán láthatjuk.



3.47. ábra

A fenti kiterjesztés azonban csak egyetlen üzenetre vonatkozik. A gyakorlatban sokszor üzenet-szekvenciák ismétlődnek, amelynek jelzésére nincs egyezményes jelölésrendszer. Hasonló módon hiányzik a választási lehetőség (alternatíva, szelekció) ábrázolása. Ez sok esetben áthidalható azzal, hogy egy másik alternatíva leírására az előzőtől különböző forgatókönyvet készítünk. Ennek a megoldásnak az a hátránya, hogy különállónak tünteti fel azt, ami csak alternatíva, következésképp a forgatókönyveink száma nagyon megnőhet.

A kommunikációs diagramok fenti hiányosságait szöveges kiegészítésekkel, magyarázatokkal enyhíthetjük.



3.48. ábra

A kommunikációs diagramnak egy másik formájában az üzenetsorrendet az objektumok közötti üzenetek sorszámozásával írhatjuk le. A 3.46. ábrával egyenértékű rajz látható a 3.48. ábrán.

3.1.3. 3.3.1.3. Az állapot

Az objektum állapotát, mint az attribútumok által meghatározott jellemzőt definiáltuk. Pontosítva a definíciót megállapíthatjuk, hogy az állapot az objektum attribútumai által felvehető értékek részhalmaza (partíciója). Az

attribútumok azon értékei tartoznak ugyanabba a részhalmazba, amely értékek esetén az objektum azonos módon viselkedik. Az állapot meghatározza, hogy egy üzenetre az objektum miként reagál. Példaként vegyünk egy pohár vizet. A pohár víz a neki küldött üzenetre egészen másként reagál, ha a hőmérséklete 0 fok alatt van, mint amikor felette. A konkrét hőmérséklet lényegtelen a viselkedés szempontjából. Vagyis a pohár víz hőmérséklet attribútumának értéke két részhalmazba sorolható, amely részhalmazokon belül az objektum által adott válaszok azonosak, de a két részhalmazon minőségileg különböznek egymástól. Valamely objektum a hozzá érkezett üzenetre adott válasza függhet az attribútumai konkrét értékétől, de a válasz egy állapoton belül minőségileg azonos lesz.

Az állapotok definiálásakor azon attribútumokat, amelyek a vezérlés szempontjából nem befolyásolják az objektum viselkedését, – ilyenek például a megnevezés típusú attribútumok – figyelmen kívül hagyjuk. Ezek az attribútumok az állapotok kialakításában nem vesznek részt, csak paraméterként viselkednek.

Az objektum válasza az üzenetre lehet valamiféle akció és/vagy állapotának megváltoztatása. Előző példánkat folytatva, feltételezve, hogy a víz éppen fagyott állapotban volt és "melegítést" üzenünk neki, akkor, ha a közölt hőmennyiség elegendő, az addigi jég folyadék állapotba kerülhet.

Az állapot az objektum két esemény közötti helyzete. Az események időpontokat jelenítenek meg, az állapotok időintervallumokat. Az objektum egy bizonyos ideig adott állapotban van. Gyakorta egy állapothoz valamilyen folytatólagos tevékenység tartozik. Korábbi példánkhoz visszanyúlva, a mozielőadás kezdete és vége közötti időre jellemző tevékenység a film vetítése.

Az események és az állapotok egymásnak párjai. Egy esemény elválaszt két állapotot, egy állapot elválaszt két eseményt.

Mind az állapotok, mind az események függnek az absztrakciós szinttől. Például a labdarúgó bajnokság szempontjából egyetlen mérkőzés kezelhető egyetlen eseményként, amelyet az eredménye jellemez. Ugyanezen mérkőzés a jelentős történések (gólok, kiállítások, cserék) sorozataként is leírható.

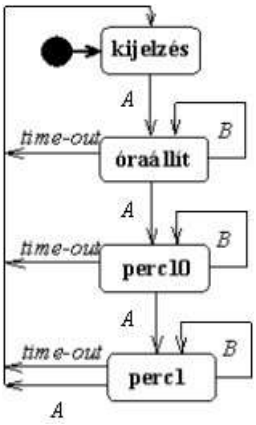
Fontos kérdés, hogy beszélhetünk-e egy asszociáció állapotáról. Amennyiben egy asszociációt (asszociatív) objektumnak tekintünk, akkor igen. A gyakorlatban az asszociációnak illetve a láncolásnak nem szükséges állapotokat tulajdonítanunk, különösen akkor, ha az objektumnak a láncolásra vonatkozó attribútumai részt vesznek az állapot kialakításában.

3.2. 3.3.2. Az állapotdiagram

Az események és állapotok sorozatát állapotdiagramban írhatjuk le. Amikor egy objektum vesz egy üzenetet (esemény), akkor az arra adott válasz függ az üzenettől és a pillanatnyi állapottól. A válasz része lehet, hogy az objektum új állapotba jut. Ezt állapotváltásnak, vagy állapotátmenetnek, röviden átmenetnek nevezzük. Az állapotdiagram olyan gráf, amelynek csomópontjai az objektum állapotai, élei pedig az események hatására bekövetkező átmenetek. Az állapotot lekerekített doboz jelöli, benne az állapot megnevezésével. Az átmenetet egy irányított él írja le, amely azt jelképezi, hogy az átmenethez tartozó esemény hatására az objektum mely állapotból melyik másikba kerül. Egy adott állapotból kivezető különböző átmenetek különböző eseményekhez tartoznak.

Az állapotdiagram leírja az események hatására létrejövő állapotok sorrendjét, az állapotgép működését. Az objektum valamely állapotából – az első olyan esemény hatására, amelyhez tartozik átmenet, – az objektum egy következő állapotba kerül. Ha az objektum adott állapotában valamely esemény nem értelmezett, akkor annak hatására nem történik állapotváltás, azaz az eseményt figyelmen kívül hagyjuk. Előfordulhat olyan esemény is amelynek hatására lejátszódó (speciális) állapotváltás alkalmával az objektum következő állapota megegyezik az aktuálisan fennálló állapottal. Egy eseménysorozat az állapotdiagramon egy útvonal bejárásának felel meg. Valamely állapotgép egy adott időpillanatban csakis és kizárólag egyetlen állapotban lehet.

A 3.49. ábrán megadtuk egy egyszerű digitális óra állapotdiagramját. Az órának két működési módja van. Az egyikben kijelzi az időt, a másikban pedig be lehet állítani az órát. Az órán két gomb (*A* és *B*) található. Alapértelmezésben az óra az időt mutatja. Ha megnyomjuk az *A* gombot, akkor lehetőség nyílik az óra értékének módosítására. A *B* gomb minden megnyomásával az óra értéke eggyel nő 12-es periódusban. Az *A* gomb ismételt megnyomásával előbb a perc tízes, majd egyes helyi értékének állítható be a *B* gomb ismételt nyomogatásával. Ha egy megadott ideig nem nyomunk gombot (*time-out*), az óra az éppen beállított értékkel áttér kijelző módba.



3.49. ábra

Egy állapotdiagram mindig az egész osztály viselkedését írja le. Mivel az adott osztály valamennyi példánya egyformán viselkedik, valamennyi ugyanazon attribútumokkal és állapotdiagrammal rendelkezik. Mivel mindegyik példány önálló, a többitől függetlenül létező, az egyes példányok pillanatnyi állapotát a konkrétan őket ért hatások határozzák meg. Azaz a különböző példányok különböző állapotban lehetnek. Ha veszünk több ezer, a 3.49. ábrán szereplő órát, egy adott pillanatban azoknak legnagyobb többsége valószínűleg a kijelzés állapotban lesz, mindössze néhány példányt állítanak éppen át.

Az objektum példányhoz tartozó állapotgép az objektum keletkezésének pillanatában megkezdi működését, azaz az állapotmodellben leírtak végrehajtását. Jelentős kérdés, hogy melyik a megszületés utáni első, kiinduló állapot. Az állapotmodellben egy nyíllal az állapothoz kapcsolt fekete ponttal jelöljük az induló állapotot. Az objektumok többsége nem él örökké. Az objektum a pusztulását okozó esemény hatására kikerül a modell hatásköréből. Az objektumok végállapotát a "bikaszem" szimbólum jelöli.

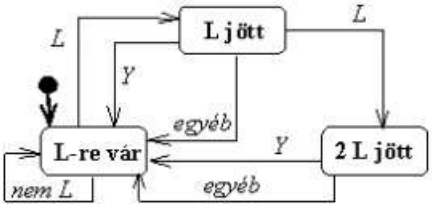
A 3.46. ábrán felhozott virágküldő szolgálat esetében a *Virágos* objektum név félrevezető, hiszen a példa csak egyetlen virágküldés (szolgáltatás) eseményeit írja le, míg az igazi virágos egymással párhuzamosan sok hasonló megrendelésnek tesz eleget. Megmaradva egyetlen szolgáltatásnál, láthatjuk, hogy az állapot modell egy megrendeléssel kezdődik és a vonatkozó dokumentumok selejtezésével ér véget. A következő 3.50. ábrán egy általános szolgáltatás állapotmodelljét mutatjuk be.

3.50. ábra

Eddig csak olyan példákat tekintettünk, amelyekben az esemény megtörténte önmagában meghatározta az állapotátmenetet. Természetesen ez egy nagyon egyszerűsített modell, mivel az átmenet függhet az esemény attribútumaitól. Sőt gyakori az az eset is, hogy csak egyetlen esemény fordulhat elő, és így a paraméterek értéke a meghatározó. Ilyenkor az a szokás, – ha az egyértelmű – hogy az állapotmodellben az esemény helyett csak annak az attribútumát adjuk meg. Pontosabban, a paraméterezett eseményt attribútumai által képviselt értékek szerinti önálló események halmazára képezzük le. Például, ha egy objektumnak üzenetként küldünk egy karaktert, akkor az esemény a "*karakter jött*" lesz, amelynek paramétere a karakter kódja. Az objektum viselkedése szempontjából elképzelhető ennek az eseménynek a kód alapján két vagy több egymást kizáró eseménnyé bontása, mint mondjuk "*betű jött*" és "*egyéb jött*". Elviekben előfordulhat, hogy a paraméter valamennyi értékének különböző eseményt feleltetünk meg. Ennek csak akkor van értelme, ha a különböző eseményekre különböző módon szükséges reagálni.

Az objektumdiagramban az állapotátmeneteket jelölő élre, az esemény megnevezését követő zárójelbe írjuk az attribútumot. Amennyiben ez nem zavaró, az eseményt helyettesíthetjük a paraméterből képzett önálló eseménnyel.

Az alábbiakban, a 3.51. ábrán megadjuk egy olyan objektum állapotdiagramját, amely objektum képes a neki karakterenként küldött szövegben felismerni az "*LY*" betűket, beleértve a "*LLY*"-t is. Az esemény a karakter küldése, amelynek paramétere a karakter kódja. Az objektum szempontjából a paraméter – a kód – három csoportba osztható. Másként kell reagálni az "*L*"-re, az "*Y*"-ra és az összes többi, "*egyéb*" karakterre.



3.51. ábra

Az objektum indulási állapota az "*L-re vár*" állapot. Ugyanis, ha valamely szövegben "*LY*"-t vagy "*LLY*"-t akarunk találni, akkor az csak "*L*" betű előfordulása után történhet, azaz várni kell a következő "*L*" karakter megjelenésére. Ez történik az "*L-re vár*" állapotban, amelyet az jellemez, hogy minden "*L*"-től különböző karakter érkezésekor nem változik meg. Az "*L*" előfordulását követően három esetet kell megkülönböztetni. Amennyiben "*Y*" következik, akkor az objektum felismert egy "*LY*"-t majd ismételten "*L*" érkezésére várakozik. Ha nem "*Y*", de nem is "*L*", hanem *egyéb* következik, akkor szintén az "*L-re vár*" állapotba kerül az objektum anélkül, hogy "*LY*"-t azonosított volna. Ha egy újabb "*L*" jön, akkor nem kizárt a kettős "*LLY*" előfordulása. Ennek ellenőrzéséhez szükséges a következő karakter értelmezése, ezért az objektum a "*2 L jött*" állapotba kerül. Ebből az állapotból bármely karakter érkezésekor az "*L-re vár*"-ba jut az objektum, de az "Y" esemény bekövetkezte egyben az "*LLY*" karaktersorozat meglétét is jelzi.

3.2.1. 3.3.2.1. Műveletek (operációk)

Az állapotmodell használhatóságának nélkülözhetetlen feltétele, hogy az események és állapotok sorrendjének leírásán túl jelezni tudjuk az egyes állapotokban, illetve az állapotátmenetek alkalmával az objektum milyen operációkat hajt végre.

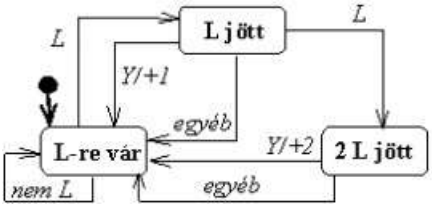
Korábban már említettük, hogy minden egyes állapothoz tartozhat tevékenység. Példaként hoztuk, hogy a mozielőadás kezdete és vége közötti állapotra jellemző tevékenység a film vetítése. Az állapotokban végrehajtott tevékenységet **aktivitásnak** szokás nevezni. Az aktivitás lehet egy folyamatosan végzett tevékenység (folyamatos aktivitás), mint például a telefon csengése. Ez a fajta aktivitás az állapotba történő belépéskor kezdődik és az állapotból történő kilépéskor ér véget. Az aktivitás másik fajtája olyan tevékenység, amely megkezdődik az állapotba való belépéskor és addig tart, amíg a tevékenység elvégzéséhez szükséges (szekvenciális aktivitás). Tipikus példája ennek egy számítás elvégzése. Ha az aktivitás befejeződött, akkor az állapotgép további tevékenységet nem végez, várakozik a következő állapotváltásra. Ha az állapotváltást előidéző esemény korábban érkezik, mint ahogy az éppen végzett tevékenység befejeződött volna, akkor a tevékenység befejezetlenül félbemarad, elvetél (abortál), mivel az állapotváltásnak van magasabb prioritása. Természetesen a rendszertervező felelőssége, hogy az abortált tevékenységet követően az állapotgép és környezete konzisztens állapotban maradjon. Példaként tegyük fel, hogy egy állapothoz rendelt tevékenység a telefonvonalon történő adatcsere. Amennyiben az állapotváltozást előidéző esemény az adatcsere befejezése előtt bekövetkezik, akkor az adatcsere elmaradhat, de a telefonvonal felszabadításáról okvetlenül gondoskodni kell.

A kétfajta (folyamatos és szekvenciális) tevékenység között nincs lényeges különbség. A folyamatos aktivitást tekinthetjük egy vég nélküli tevékenységnek, amelyre biztos, hogy idő előtt abortálni fog. Az állapotdiagramon az állapotot neve alatt álló "*do: <aktivitás>*" szöveg jelöli mindkét aktivitást.

Az állapotgép az átmenet során akciókat hajthat végre. Az akció egyértelműen az eseményhez van rendelve és a végrehajtásához szükséges idő elhanyagolhatóan kicsinek tekinthető. Ilyen akciónak tekinthető – az előző példát folytatva – a telefonvonal bontása. Természetesen a valóságban nincs 0 idő alatt végrehajtható akció, az elhanyagolhatóság értelmét a modell finomsága határozza meg. Amennyiben az akció nem pillanatszerű, akkor azt aktivitásként állapothoz kell rendelni. A állapotdiagramon az akciót az állapotátmenetet kiváltó eseményhez kell kapcsolni, attól törtvonallal elválasztva.

Valamely objektumban végrehajtott tevékenységek – az aktivitás és az akció – által érintett objektumok, attribútumok, paraméterek stb. együttesét az objektum által látható elemeknek nevezzük. Ebbe körbe a *Demeter* törvény értelmében belefférnek az adott objektum attribútumai, metódusai, globális változók, illetve más

objektumok metódusai. A tevékenység során például küldhetünk üzenetet egy objektumnak, vagy megváltoztathatjuk a saját objektumaink attribútumait, változóit. A 3.51. ábrán adott állapotgép, amely felismeri az "*LY*"-okat, felhasználható az előfordulások megszámlálására is, ha bevezetünk egy számláló attribútumot. Az állapotgép valahányszor felismer egy vagy két "*LY*"-t, a számlálót növelni kell. Ez a modellünk szempontjából akciónak tekinthető, mert a végrehajtásához szükséges idő elhanyagolható és a tevékenység átmenethez köthető. A modell akciókkal kiegészített változata a 3.52. ábrán látható. A számláló növelését a *+1* illetve *+2*-ként jelölt akciók végzik el.



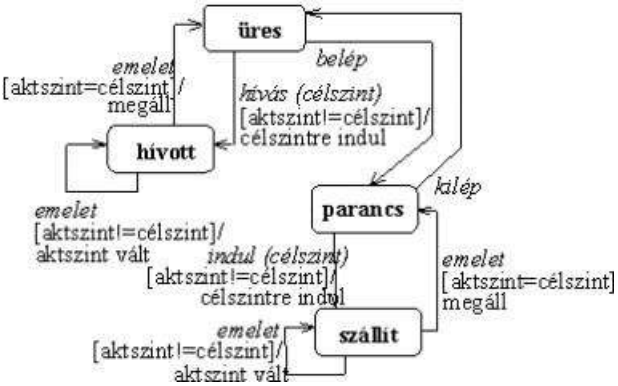
3.52. ábra

3.3. 3.3.3. Az állapotgép fogalmának kiterjesztése

Az állapotváltásokat először az események bekövetkeztéhez kötöttük. Majd ezt a modellt az eseményhez tartozó attribútumok figyelembevételével bővítettük. A modell tovább bővíthető olyan módon, hogy az átmenet végrehajtásához az esemény és annak attribútuma mellett, egy feltétel fennállását is figyelembe vesszük. A feltétel egy logikai érték, amely az objektum által látott elemek, attribútumok, változók értékéből képezhető. A feltétel az állapotátmenetben őrként viselkedik. A feltételes átmenet akkor hajtódik végre, ha az esemény bekövetkezésének pillanatában mintavételezett feltétel igaznak bizonyul. Ezen a módon ugyanazon esemény megtörténte az állapotgépet különböző állapotba viheti a feltételek teljesülésének függvényében. A feltételt az eseményt követően, szögletes zárójelek között tüntetjük fel az állapotmodellben.

A 3.53. ábrán egy egyszerű liftszekrény állapotdiagramja látható. A szekrény alapállapotában valamelyik emeleten áll nyitott ajtókkal arra várva, hogy valaki belép a szekrénybe, vagy más emeletről hívják a liftet. Ez utóbbi esetben ajtaját becsukva üresen arra az emeletre megy, ahonnan hívták, majd elérve a kívánt emeletet, ajtaját kinyitva alapállapotba kerül. Amikor a szekrénybe valaki belép a lift belső irányításra kapcsol, és mindaddig ebben marad amíg legalább egyetlen utas van a fülkében. A belső irányítás annyit jelent, hogy a lift a hívásokat figyelmen kívül hagyja. A lift menet közben valamennyi emeleten kap egy jelzést, hogy az emelethez érkezett.

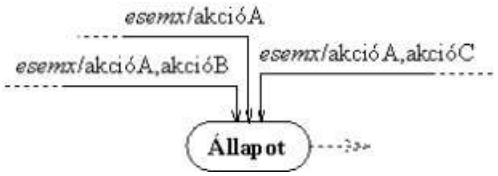
Tételezzük fel, hogy a szekrény "tudja", hogy éppen melyik szinten van és ez a tudás egyben az egyik (aktuális szint, *aktszint*) attribútum. Ha a liftet elhívták vagy elindították a célszintre, akkor az aktuális szint és a célszint alapján meghatározható mozgásának iránya. Egy emelet elérése esetén kapott üzenet megérkezésekor meg kell vizsgálni, hogy a célszinten van-e a liftszekrény. A feltétel teljesülésétől függően váltunk állapotot és megállunk, vagy maradunk a régi állapotban és az *aktszint* értékét növeljük/csökkentjük eggyel a mozgásiránynak megfelelően.



3.53. ábra

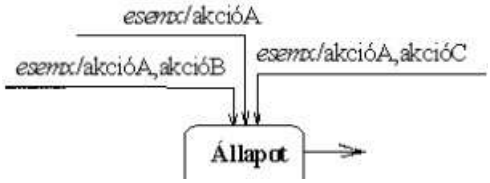
Abban az esetben, ha egy állapotba több másik állapotból lehet eljutni és valamennyi átmenet esetén ugyanazt az akciót (is) kell végrehajtani, akkor jogosan állíthatjuk azt, hogy az átmenetekhez rendelt közös akció nem

annyira az átmenethez, mint inkább magához az állapotba jutáshoz tartozik. A 3.54. ábrán egy állapotdiagram-részlet látható, amelyen valamennyi átmenetnél végrehajtódik az *akcióA*.



3.54. ábra

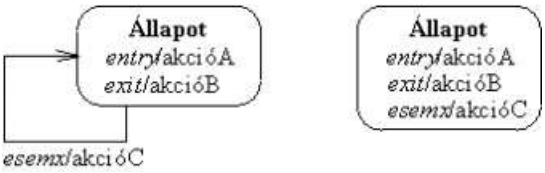
Érdemes ilyenkor az ábrát átszerkeszteni a 3.55. ábrán látható módon. Az ábrán az állapotot jelentő dobozban az *entry* kulcsszó mögött feltüntetjük a belépéskor végrehajtandó akciót. Felhívjuk a figyelmet, hogy az *entry* akció nem tévesztendő össze az állapothoz tartozó aktivitással.



3.55. ábra

A belépő akcióhoz hasonló módon definiálható kilépő (*exit*) akció, ha valamennyi az adott állapotból kivezető átmenetnél végrehajtunk közös akciókat. Ilyen helyzet áll elő, ha például az állapotban olyan tevékenységet folytatunk, amelyet befejeződése előtt történő állapotváltás miatt esetleg abortálni kell. Az állapotgép és az attribútumok konzisztenciáját célszerűen egy közös kilépő akcióban lehet biztosítani.

Gyakorta előfordul, hogy egy esemény bekövetkeztekor nem kell állapotot váltani, de az eseményre válaszul akciót kell végrehajtani. Az állapotdiagramon ezt az állapotba visszavezető éllel jelöljük. Erre láthatunk példát a 3.53. ábrán, amikor még nem értük el a *célszintet*, de az *aktszintet* váltani kell. Ha az állapothoz – és ezen keresztül valójában az eseményhez – *entry* és *exit* akciókat is rendeltünk, kérdéses, hogy olyankor is végre kell-e hajtani ezeket, ha az esemény bekövetkezésekor a korábbi állapotban maradunk. Ezt a helyzetet az OMT módszertan a következőképp értelmezi. Az *entry* és *exit* akciókat minden be- és kilépő átmenetre végre kell hajtani. Ha azonban az állapotban maradunk és csak az eseményhez rendelt akció végrehajtása szükséges, akkor tilos az állapotba visszatérő átmenetet az ábrába berajzolni. Helyette az állapot dobozába kell az eseményt és az akciót felvenni. A 3.56. ábra a két ábrázolás között különbséget mutatja.



3.56. ábra

A baloldalon álló jelölés esetén az *esemx* esemény bekövetkeztekor végrehajtott akciók sorrendben: *akcióB*, *akcióC*, *akcióA*. A jobboldali ábra értelmezése szerint csak *akcióC*-t kell végrehajtanunk.

A 3.57. ábrán áttekintő jelleggel összefoglaltuk az állapotmodellhez rendelhető akciók jelölését és megadtuk a végrehajtás sorrendjére vonatkozó szabályt.

3.57. ábra

Az *Állapot1*-be történő belépés alkalmával az állapotgép működése a következő: *akcióA*, majd az *aktivitás* végrehajtása. Ha az *esemy* esemény megtörténtekor a *feltétel* fennáll, az *aktivitás* végrehajtása megszakad, *akcióB* és *akcióD* után lépünk *Állapot2*-be.

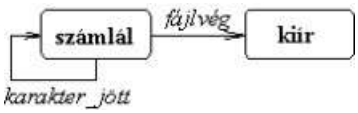
Ha az *Állapot1*-ben *esemx* történik a feni megállapodás értelmében csak *akcióC*-t kell végrehajtani. A modell nem rendelkezik arról, hogy mi a teendő abban az esetben, ha az *esemx* az *aktivitást* szakította meg, majd *akcióC*-t végrehajtva az állapotgép visszalép *Állapot1*-be. Mi történjék az aktivitással? Induljon el újra, folytatódjon ahol abbamaradt, vagy ne történjék semmi? Általános szabály nem fogalmazható meg, mivel mindhárom eset lehet értelmes, és mindegyikben nagy körültekintéssel kell eljárni.

Gyakorta az állapot felvételének egyedüli oka egy szekvenciális aktivitás végrehajtása, majd annak befejeződése után továbblépés a következő állapotba. Az **automatikus átmenet** olyan állapotátmenet, amelyhez nem rendelünk explicit eseményt. Az induló állapotban végrehajtott tevékenység befejeződése egy implicit esemény, amelynek hatására megtörténik az átmenet. Egy *A* állapotból több automatikus átmenet is kivezethet, de ekkor ezeket egymást kizáró feltételekkel kell ellátnunk. Így csak az az átmenet hajtódik végre a tevékenység befejezésekor, amelyiknek a feltétele fennáll. Azaz, a tevékenység befejezésével mintát veszünk a feltételekből. Ha egyik feltétel sem áll fenn, akkor elvileg – mivel a tevékenység többször nem fog befejeződni – az állapotgép örök időkre *A*-ban rekedhet. Ennek megakadályozására, – akkor, ha az aktivitás befejeztekor nem volt teljesülő feltétel – mindegyik feltétel bekövetkezte, "éle" eseményként értelmezett. Ekkor az elsőként igazzá váló feltételnek megfelelő átmenet hajtódik végre.

3.4. 3.3.4. Beágyazott állapotmodellek

Egy adott állapotban előírt aktivitás specifikálható beágyazott állapotdiagrammal. A beágyazott modell (almodell) akkor kezd működni, amikor az őt tartalmazó főmodell állapothoz tartozó aktivitás végrehajtása megkezdődik. Ekkor az almodell a megjelölt induló állapotba kerül. Ezt követően a modell végrehajtásának szabályai alapvetően megegyeznek a definiáltakkal. Az almodell állapotai megfeleltethetők a főmodell állapota finomításának. Ebből következik, hogy az állapotgép, amely eddig a főmodell szerint működött, most az almodellt hajtja végre. Azt a szabályt, miszerint egy állapotgép egy időpillanatban csak egyetlen állapotban lehet, a főgépből kiterjeszthetjük a beágyazott gépre is. A fő- és almodell állapothalmazainak (amelyek diszjunktak, mivel egy állapot nem lehet egyszerre a fő- és almodellben is) uniójára igaz, hogy közülük csak egyetlen állapotban lehet az állapotgép.

A korábban tárgyalt *LY*-t számláló objektum modellje kiegészíthető az eredmény megjelenítésével. Így a 3.58. ábrán látható modellt kapjuk, amely azt a tényt rögzíti, hogy az objektumunk működésének egyik fázisában számlája az *LY*-okat, a másikban pedig kiírja azt.

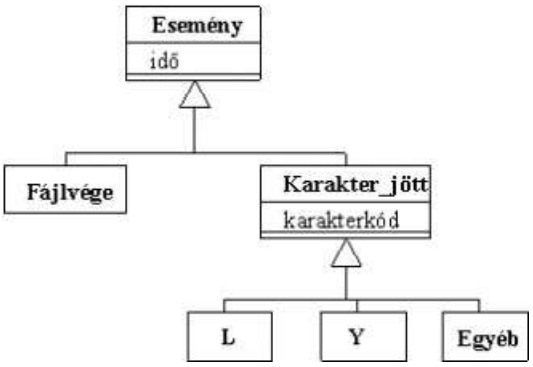


3.58. ábra

Ebben a *számlál* állapotban végrehajtott tevékenységet írja le a 3.52. ábrán definiált állapotmodell. A *számlál* állapotban figyelembe veendő események a beágyazott modellben két módon jelenhetnek meg. Egyfelől a *karakter_jött* esemény az almodellen konkretizálódik és *L*, *Y* és *egyéb* események formájában jelenik meg. Másfelől a *fájlvég* esemény az almodell valamennyi állapotán úgy értelmezett, hogy hatására az almodell működése befejeződik és a főmodell a *kiír* állapotba kerül.

Általánosságban is elmondható, hogy a főmodellre vonatkozó események az almodellen belüli specifikusabb események általánosításai, vagy az almodell valamennyi állapotában egyformán értelmezett közös események.

Az események általánosításának gondolatát folytatva megállapíthatjuk, hogy az események hierarchikus rendbe szervezhetők, lényegében érvényes rajtuk az öröklés. A példánkban az események hierarchiáját a 3.59. ábrán mutatjuk be. A doboz felső részébe írjuk az eseményt, az alsó részébe pedig a paramétereket (ha van).



3.59. ábra

Az esemény-hierarchia bevezetésével lehetővé válik, hogy a modellezés különböző szintjein, az adott szintre jellemző eseményeket használjuk.

3.5. 3.3.5. Az állapotátmenet-tábla

Az állapotgépet vezérlő modellnek egy másik, a gyakorlatban szintén elterjedt leírása az állapotátmenet-tábla vagy röviden állapottábla. A táblázat sorai jelentik a különböző állapotokat, a táblázat oszlopai pedig az eseményeket. Egy sor és egy oszlop metszetében egy cella áll. A cellába bejegyezzük, hogy a cella sorának megfelelő állapotban a cella oszlopában jegyzett esemény hatására az állapotgép melyik következő állapotba jut és az átmenet során milyen akciót hajt végre. A 3.60. ábrán az *LY*-t számoló objektumunkat írtuk le táblázatos formában.

	L	*Y*	*egyéb*
L-re vár	L jött	L-re vár	L-re vár
L jött	2 L jött	L-re vár/+1	L-re vár
2 L jött	-	L-re vár/+2	L-re vár

3.60. ábra

A tábla ebben a formájában a legegyszerűbb állapotmodellt írja le. A *2 L jött* állapotban bekövetkező *L* eseményre nem számítunk, mivel a feladat kiírása szerint magyar szöveget vizsgálunk, ahol ilyen eset nem fordulhat elő. Ha egy adott állapotban egy esemény nem fordulhat elő – mint példánkban is –, akkor a megfelelő cellát üresen hagyjuk vagy csak egy kötőjelet írunk bele, jelezve, hogy a cella tartalma lényegtelen, számunkra közömbös.

Természetesen a táblázat más oszlopokkal bővíthető, amelyekbe bejegyezhetjük az *entry*, *exit* akciókat és a tevékenységet. Az események mellett az attribútumok és feltételek figyelembevétele történhet az oszlopok számának növelésével, és/vagy a cellába írt – az attribútumokra és a feltételekre kidolgozott – kifejezés alkalmazásával.

A táblázat használata előnytelen, ha túl sok eseményünk van és az állapotátmenetek különböző eseményekhez tartoznak. Ilyenkor a táblánk nagy lesz, mivel az oszlopok száma megegyezik az összes események számával és várhatóan sok közömbös bejegyzést kapunk.

Az állapottáblát akkor célszerű használni, ha kevés az eseményünk, és ezen kevés esemény viszonylag sok állapotban értelmezett. Az állapottábla alkalmazása implementációs szempontból is előnyös lehet.

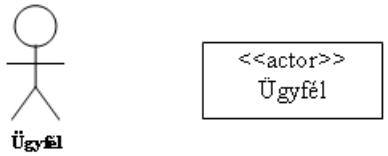
4. 3.4. A funkcionális modell

A funkcionális modell egy rendszerről alkotott kép harmadik vetülete, amely leírja, hogy a rendszer milyen funkciókat nyújt az őt használók számára. A rendszer által kínált funkciót a használati eset (use case) definiálja. A használati eset leírja a rendszer és az őt felhasználó külső szereplők (aktorok) közötti akciók és reakciók (válaszok) sorozatát, az interakciókat.

Az **aktor** a használati esetekben külső félként résztvevő szereplők által játszott összetartozó szerepek együttese. Az aktor megtestesíthet embereket, vagy (többnyire automatikus) berendezéseket, akik, illetve amelyek a rendszerrel együttműködni képesek. Egy ATM bankautomatánál történő pénzfelvételnél, mint használati esetnél aktor a bankkártyás ügyfél és a bank központja. A pénzfelvétel kezdeményezője az ügyfél, aki az ATM által diktált lépések sorozatának végrehajtásán keresztül készpénzhez jut. A funkció, a pénzfelvétel elképzelhetetlen a másik aktor, a bank központjának közreműködése nélkül. Vele kapcsolatosan a kezdeményező az ATM, majd a központ a kapott adatok (kártya száma, PIN kód, összeg) alapján utasítja az ATM-et a pénz kiadására, vagy a felvételi folyamat megszakítására. A bank központját egy számítógépes rendszer testesíti meg.

Az aktor egy szerepet jelenít meg, következésképp típus jellegű (az osztályhoz hasonló) fogalom. Ennek megfelelően az aktor egyik megjelenési módja egy UML osztály, amelyet az <<actor>> sztereotípiával jelölünk meg. Az aktor konkrét példányai forgatókönyvekben, scenáriókban szoktak előfordulni.

Az aktor jelölésére a leggyakrabban az ábrán szereplő pálcika-embert használjuk, amely a fent említett <<actor>> -ral jelzett osztály ikonos formája. Ezt a változatot érdemes használni akkor, ha a rendszer egészére vonatkozó használati eset diagramot készítünk, ezzel is hangsúlyozva, hogy az aktor a vizsgált rendszeren kívül helyezkedik el. Részrendszer használati eset diagramján az aktor a rendszer eleme is lehet, ezért ilyenkor szokás szerint az osztályt használjuk az ábrázolásra.



Az aktor az alábbi tulajdonságokkal jellemezhető:

- név –
- rövid leírás – amely az aktor felelősségét, érdekét tárgyalja
- jellemző – az aktor működési környezete, egyéb jellemzője, amelynek a használati esetben jelentősége lehet (embernél annak felkészültsége, kora stb.)
- relációk – a vele kapcsolatban álló más modell elemek (tipikusan aktorok, használati esetek, de lehetnek például osztályok is) felsorolása, és a kapcsolat minőségének, tartalmának megadása
- diagram – azon diagramok (pl. használati eset, szekvencia, stb.) felsorolása, amelyen az aktor szerepel

Az aktor neve és rövid leírása a legegyszerűbb definícióból sem hiányozhat; a többi jellemző esetleges.

A **használati eset** (use case) a rendszer által végrehajtott akciók együttesét, sorozatát specifikálja, amelyek az aktor számára megfigyelhető eredményt szolgáltatnak. Ebből következik, hogy a használati eset a rendszerrel szemben támasztott funkcionális követelmény megjelenése. Egy használati eset lehet például bank esetében hitel igénylése, vagy egy banki átutalás kezdeményezése. Fontos hangsúlyozni, hogy a használati eset csak a használó és a rendszer közötti együttműködésre koncentrál (mi történik közöttük), figyelmen kívül hagyva a megvalósítás részleteit. A hitelkérelmet benyújtó ügyfél számára teljesen lényegtelen a banki belső üzemmenet, az ő dolga csak a bank által kért anyagok benyújtása, a bankkal való együttműködés.

A használati eset – túl azon, hogy leírja a rendszer és a használói közötti interakciókat – a megtörténte során végzett tevékenység, munka eredményességét is jelenti. Valamely aktor szempontjából egy használati eset értéket állít elő, például kiszámol egy eredményt, készít egy új objektumot, vagy megváltoztatja egy objektum állapotát. A hiteligénylés példában az igénylő számára a megkapott hitel nagyon konkrét értéket jelent. Nyilvánvalóan az aktoroknak nem áll érdekükben a rendszer olyan használata, amely számukra haszontalan.

A használati esetek leggyakrabban – példáinkban szereplő esetekben is – a rendszer egészére vonatkoznak. Azonban készíthetünk használati eseteket a rendszer részeire, alrendszerekre, de akár osztályokra vagy interfészekre is. Természetesen ez esetben különös figyelemmel kell meghatározni az aktorokat, akik a használati eset által jellemzett viselkedésű részrendszeren (alrendszer, osztály) kívül, de gyakran az egész rendszeren belül helyezkednek el. Ha például egy osztály viselkedését jellemző használati esetet definiálunk, akkor az aktor megtestesíti az osztály által nyújtott szolgáltatás klienseit. Ezek a kliensek legtöbb esetben a rendszer más osztályai. A rendszeren belüli aktorokhoz kapcsolódó használati esetekkel írhatjuk le az olyan

funkciókat, amelyek bizonyos időponthoz kötődnek vagy időnként megismétlődnek. Ez esetben az aktor egy időzítő, egy “vekker”, amely a lejáratkor kezdeményezi a megfelelő használati eset szerinti viselkedést.

A részrendszerekre kidolgozott felhasználói esetek nem csak a részrendszer viselkedését irják le, hanem a részrendszerre vonatkozó funkcionális tesztelés alapját képezik.

Egy használati esetnek különböző változatai lehetnek. A felhasználó viselkedésétől vagy egyéb körülménytől függően a használati eset lefolyása különbözőképp történhet. Gondoljunk arra, hogy az automatánál történő pénzfelvételi folyamat sem a várt eredménnyel végződik, ha az ügyfél elfelejtette a PIN kódját vagy nincs fedezete. Felismerhetünk azonosságokat különböző felhasználói esetekben. Az ATM-en valamennyi művelet végrehajtásához szükséges a felhasználó PIN kód alapján történő azonosítása, és ez az azonosítás mindig ugyanúgy történik, függetlenül attól, hogy pénzfelvételről vagy egyenleg lekérdezéséről van szó. Megfigyelhetjük, hogy a használati esetek “szokásos” lefolyása alkalmanként specialitásokkal egészül ki. Általában az ügyfelek a felveendő pénz mennyiségét egy az ATM által adott listáról választják, de az ügyfélnek módjában áll – bizonyos szabályok megtartásával – szabadon megadni a pénz mennyiségét. Ez a lehetőség tekinthető a standard használati eset valamilyen kiegészítésének.

A használati eset szigorúan véve típus jellegű; az egyes konkrét esetektől elvonatkoztat, általánosít. A használati eset egy példánya a forgatókönyv vagy scenárió. Egy forgatókönyv az aktor és a rendszer közötti konkrét eseti együttműködést definiálja. Ezen forgatókönyvek – amelyek egyébként különböző változatokban mehetnek végbe – absztrakt együttese a használati eset. Egy ATM bankautomatából történő pénzfelvétel egy konkrét forgatókönyve például az, amikor az ügyfél olyan összeget kér, amelyet az automata a rendelkezésre álló címleteiből nem tud szolgáltatni, emiatt a pénzfelvétel megszakad. Egy másik lehetséges forgatókönyv szerint a pénzfelvétel már a hibásan beírt PIN-kód miatt áll le.

A használati esetek grafikus megjelenése a legtöbb esetben egy ellipszis, amelybe, vagy amely alá írjuk az eset nevét. Ritkábban használjuk azt az ábrázolási módot, amelyben a felhasználói eset nevével osztályt rajzolunk, amelyet egy ellipszis alakú ikonnal jelölünk meg az osztályt reprezentáló doboz jobb felső sarkán.

A használati esetet általában szöveges leírással adjuk meg, amely lehet folyó vagy struktúrált szöveg. A leírás mélysége erősen függ attól, hogy a szoftver fejlesztésének melyik fázisában alkalmazzuk a use case-t. A rendszer elemzésének fázisában például megelégedhetünk az eset nevének és rövid leírásának megadásával, de a tervezéskor a leírás részleteinek ki kell térnie a kezelői felületen található elemek használatára.

Az alábbiakban közreadjuk a használati eset jellemzésére használt paramétereket

- név –
- rövid leírás – amely egy-két mondatban definiálja a use case célját
- események sorrendje – általában pontokba szedve, időrendben felsoroljuk az eseményeket, kitérve a rendszernek az eseményekre adott válaszára.
- speciális követelmények – nem a funkciókra, hanem azok egyes paramétereire vonatkozó előírások (sebesség, titkosság, ismételhetőség, az eset fontossága, stb.)
- előfeltételek – az eset megkezdésekor a rendszertől elvárt feltételek, körülmények
- utófeltételek – az eset befejezését követően a rendszerre vonatkozó korlátozások.
- kiterjesztő pont – az események folyamatában azon pontok, amelyeknél lehetséges a használati eset alternatívákkal történő kiegészítése
- relációk – a vele kapcsolatban álló modell elemek (tipikusan aktorok, használati esetek, kollaborációk, stb.) felsorolása és a kapcsolat minőségének, tartalmának megadása
- diagram – azon UML diagramok felsorolása, amelyen a használati eset szerepel

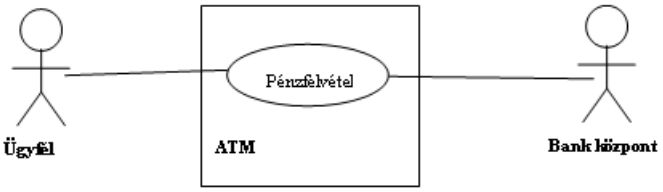
A minimálisnak tekinthető használati eset definició a nevet és a rövid leírást tartalmazza. A szokásos, kiegészített változatban megtaláljuk az események sorrendjét és a speciális követelményeket, a relációkat és kapcsolt diagramokat. Az elő- és utófeltételek általában a kiemelt fontosságú és – valamilyen szempontból – kritikus használati esetekben fordulnak elő.

Az alábbiakban példaként megadjuk egy egyszerűsített internetes banki átutalás kezdeményezésének használati esetét.

név	Átutalás
rövid leírás	Az átutalás célja, hogy az Ügyfél valamelyik kezelésébe tartozó bankszámláról pénz átadását kezdeményezze más bankszámlára.
események sorrendje	1. A rendszer listázza az Ügyfél rendelkezésére álló számlákat. 2. Az Ügyfél kiválasztja a számlát, amelyről utalni kíván. 3. A rendszer felkínálja az üres átutalási űrlapot, amelyen forrásként a választott számla szerepel. 4. Az ügyfél kitölti az űrlapot 5. A rendszer az űrlapot formailag ellenőrzi (kötelező mezők kitöltöttek, célszámla száma korrekt), majd az átutalást jóváhagyásra felkínálja az Ügyfélnek. 6. Ügyfél az átutalást jóváhagyja 7. A rendszer az ügyfélnek SMS-ben küld egy ellenőrző kódot, és felkínálja az ellenőrző kód beírására szolgáló űrlapot. 8. Az ügyfél az SMS-ben kapott kódot beírja a felkínált űrlapra. 9. A rendszer ellenőrzi, hogy a benyújtott kód megegyezik-e az SMS-ben küldöttel. Ha igen, végrehajtja az átutalást.
speciális követelmények	Az Ügyfél minden egyes lépésének 3 percen belül be kell fejeződnie.
előfeltételek	Az Ügyfélnek az átutalás előtt be kell jelentkeznie a rendszerbe, aminek során megtörténik az Ügyfél azonosítása. Az Ügyfélnek a választott bankszámlán legalább az átutalandó összegnek megfelelő pénzzel kell rendelkeznie.
utófeltételek	A bankszámla egyenlege az átutalás összegével csökken azonnal. Az átutalás bankon belül azonnal végrehajtódik, más bankba irányuló átutalásnál a GIRO munkarendje szerint, általában a következő munkanap.

kiterjesztő pont	A 3. pontnál ha az Ügyfél a “sémák” billentyűt nyomja le, akkor a rendszer felkínálja a korábban letárolt sémákat, hogy azok közül válasszon az Ügyfél kiinduló átutalási űrlapot az üres helyett.
relációk	asszociáció (1-1) az Ügyfél aktorral kiterjesztés a ”Sémákból választás” használati eset által
diagram	Magánszemély bankolása c. használati eset diagram X.5.2 Szekvenciadiagram

A **használati eset diagram** (use case diagram) olyan osztálydiagram, amelyen csak aktorokat, használati eseteket és a közöttük levő kapcsolatokat jelöljük. Az alábbi diagramban egy felhasználói eset (Pénzfelvétel) és az azzal együttműködő két aktor (Ügyfél, Bank központ) látható.



A diagramon – nem kötelezően – feltüntethető az a rendszer (alrendszer, osztály, stb.), amely a használati esettel definiált funkciót szolgáltatja. A rendszert (példánkban az ATM) jelképező téglalapba ágyazva jelenítjük meg a használati eseteket; a téglalap határoló vonalai jelölik a rendszer határát. A használati esetek rendszerbe ágyazása nem jelent tartalmazást vagy komponens viszonyt, csak azt jelezzük, hogy használati eset az adott rendszeren értelmezett funkció. Az aktorok – nyilvánvalóan – a rendszeren kívül állnak.

Az együttműködés tényét asszociáció jelöli. Az asszociáció nem irányított, mivel nem életszerű az olyan együttműködés, amelyben a felek kölcsönösen nem ismerik egymást. A használati eset oldalán előforduló multiplicitás jelentése és értelmezése a szokásos. Leírja, hogy az asszociáció másik végén álló aktor egyidejűleg hány használati eset példánnyal állhat kapcsolatban. Példának okáért tekintsük használati esetnek a web-es böngészést. Nyilván egy aktor egyidejűleg több böngészőben is dolgozhat. Az aktorhoz kapcsolódó multiplicitás nem ilyen egyértelmű, mert nem definit, hogy a használati esetet alkotó interakciók lefutása során az aktorok időben hogyan kapcsolódnak. Egy kritikus katonai, technikai vagy pénzügyi tranzakció megkövetelheti az aktorok egyidejű kapcsolódását (a rakéta indítógombjait egyszerre kell két embernek megnyomni), de egy időben hosszan elnyúló használati eset egyes interakcióiban egymást váltó aktorok vehetnek részt (például a nagymamát az egyik unoka hívja, de egy idő után átadja a készüléket a másiknak, és a beszélgetést már a másik fejezi be).

Aktorok között és használati esetek között nem létezik asszociáció.

Az aktorok között értelmezhető az általánosítás vagy specializálás reláció, ugyanolyan tartalommal, mint azt az osztályoknál megismertük. Az általános aktornak helyettesíthetőnek kell lennie a specializált aktorával. A helyettesíthetőség az aktorokhoz kapcsolódó használati esetek vonatkozásában értendő. A reláció jele is megegyzik a megszokottal: folytonos vonal háromszögletű fejjel, ami az általánosított aktorra mutat.

Példánkon két aktort látunk, a Pénztárost és a Felügyelőt. A Pénztáros a pénztárgépen csak az eladás funkciót tudja használni, míg a Felügyelőnek módjában áll a vásárlónak visszatérítést is adni. A Felügyelő bármikor helyettesítheti a Pénztárost, hiszen a Felügyelő is jogosult eladni. Viszont a Pénztáros nem léphet a Felügyelő helyére, mert nem adhat visszatérítést.

Az általánosítás-specializálás relációt alkalmazhatjuk a használati eseteken is, a fentebbel analóg módon. Az általánosabb használati esetnek helyettesíthetőnek kell lennie a specializáltabbal.

Egy boltban megszokott általános használati eset a Fizetés. Ennek két specializált változatát alkalmazzuk: a Bankkártyás és a Készpénzes fizetést. Bármelyik specifikus használati eset az általános helyébe léphet, hiszen az általános Fizetés funkciót mindketten képesek ellátni.

A fenti ábrán a bankautomatával kapcsolatos használati eset diagram egy kibővített változatát láthatjuk, Az ábrán azt akartuk rögzíteni, hogy mindkét funkciónak (Pénzfelvétel, Egyenleg lekérdezése) része az Ügyfél azonosítása, amely teljesen független attól, hogy melyik funkcióban fordul elő.

A használati esetek között értelmezett, – <<include>>-dal jelölt és szaggatott vonallal ábrázolt – *rész-eset* (include) reláció azt jelenti, hogy a nyíl talpánál álló használati eset részként tartalmazza, a nyíl hegyével jelzett használati esetet. Azaz a nyíllal mutatott használati esettel definiált funkcionalitás teljes mértékben beágyazódott a másik használati eset által definiált viselkedésbe. A rész-eset bevezetésének célja az, hogy kiemeljük a több használati esetben is részként előforduló közös használati eseteket. A reláció a függőség egy változata, amely kifejezi, hogy a tartalmazó használati eset a tartalmazott nélkül nem teljes, attól függ. Ez a függés mindig egyirányú, a tartalmazótól a tartalmazottra mutat. A tartalmazott használati esettel jellemzett funkció végrehajtása megfelel egy szinkron hívással kezdeményezett eljárás vagy function hívásának. A tartalmazott használati eset funkcionalitása teljes egészében megvalósul mielőtt a tartalmazó funkciója beteljesülne.

A rész-eset reláció mentén a használati esetek hálóba szervezhetők. Ebben a hálóban nem lehetnek körök, azaz egy használati eset még áttételeken keresztül sem tartalmazhatja önmagát.

A rész-esetként kapcsolt használati eset nem áll meg önmagában, csak az őt részként tartalmazó másik használati esettel együtt. A bankautomata esetében nyilvánvalóan az ügyfél azonosítása teljesen felesleges, ha nem akarjuk egyik funkciót (pénzfelvétel, egyenleg lekérdezés) sem végrehajtani.

A használati esetekbe beleértjük a funkció különböző változatait. A funkciónak van egy szokásos, “normális” lefutása, amelyhez képest az alkalmanként megvalósuló forgatókönyvek eltérnek. Korábban példaként hoztuk fel a bankautomatánál történő pénzfelvételnél az összeg megválasztásának módját. Az automata választásra kínálja fel a leggyakrabban előforduló összegeket egy listában, de egy változatban lehetősége van az ügyfélnek tetszése szerinti összeget is kérni. A “Tetszőleges összeg bekérése” tekinthető egy részfunkciónak, amelyet egy használati esettel írhatunk le.

Ez a használati eset <<extend>>-del jelölt függőségen (*kiterjesztés*) keresztül kapcsolódik az őt opcionálisan felhasználó – Pénzfelvétel – használati esethez.

A kiterjesztés reláció azt jelöli, hogy a függőség irányát jelző nyíl hegyénél álló (fő) használati esettel jellemzett viselkedést alkalmanként, bizonyos változatokban kiegészíti a nyíl talpánál álló (kiterjesztő) használati esettel leírt viselkedés. A fő használati eset – eltérően a korábban tárgyalt rész-eset relációban szereplő “tartalmazó” használati esettől – önmagában értelmes viselkedést definiál, függetlenül a kiegészítő esettől. A kiterjesztő használati eset magában általában nem értelmes, mivel a fő használati eset speciális körülmények közötti viselkedését adja meg.

A kiterjesztéssel definiált viselkedés szerinti működés a fő használati esetben folyó interakció egy bizonyos pontján következik be, ha annak feltételei teljesülnek. A feltételeket az <<extend>> függőséghez csatolt megjegyzésben tüntethetjük fel. A fő használati esetet definiáló esemény sorrend leírásban névvel elláthatjuk (megcimkézzük) azt a kiterjesztési pontot (extension point), ahová a kiterjesztés bekapcsolódik. A következő ábrán a példában szereplő tetszőleges összeg bekérésére vonatkozó kiterjesztést és a kapcsolódó használati eseteket láthatjuk “felnagyítva” és kiegészítve a feltétellel és a kiterjesztési ponttal.

A használati esetek az osztályoknál megszokott módon csomagokba (package-ekbe) rendezhetőek.

5. 3.5. A modellek kapcsolata

A fejezetben ismertetett három modell ugyanannak a rendszernek három különböző nézőpontból alkotott képét adja meg. Az objektummodell definiálja a szereplőket, akik a dinamikus modellben meghatározott időrendben a funkcionális modell által leírt cselekvéseket hajtják végre illetve szenvedik el.

A funkcionális modell használati esetei definiálják azt, hogy a rendszertől mit várnak el a külvilág szereplői. A definíció tartalmazza a felek közötti interakciókat. Az interakciókban az érintettek, a rendszer és az aktorok megadott sorrendben egymástól szolgáltatásokat kérnek, illetve azokat teljesítenek. A szolgáltatás kérések szekvenciadiagrammal írhatók le, amelyek a dinamikus viselkedést definiálják.

A rendszert részekre (alrendszerekre, osztályokra), azokat további alrészekre, részrendszerek hierarchikus sokaságára bonthatjuk fel, amelyben az egyes egységek az őket használó részek számára valósítanak meg funkciókat azon keresztül, hogy interakcióban állnak egymással. Egy használati eset megvalósítása érdekében lezajló interakciókban az objektummodellben szereplő elemeknek általában csak egy töredéke kap szerepet. Az objektummodell azon elemei, amelyek egy konkrét használati eset megvalósításához szükséges interakciók részesei, jól meghatározhatók. A használati eset megvalósítása, implementálása az együttműködés (collaboration). Ezen állítás UML-es megjelenítése látható az alábbi ábrán. A használati esethez hasonlító, de szaggatott vonallal rajzolt ellipszis jelenti az együttműködést, közöttük pedig az implementálás szokásos jele.

Ezzel összekapcsolódott a funcionális, a szerkezeti és dinamikus modell, hiszen az együttműködés önmagában kettős. Egyfelől tartalmazza azon szerkezeti elemeket, osztályokat, a közöttük fennálló relációkat, amelyek felelősek az interakciók végrehajtásáért, mint az azokban résztvevők, másfelől leírja az interakciókat, mint a szolgáltatáskérések sorozatát.

Egyértelmű, hogy a rendszer használati eseteit az objektummodellben szereplő osztályok illetőleg azok példányai fogják megvalósítani. Felesleges tehát olyan szerkezeti elemet definiálni, amely egyetlen használati eset megvalósításában sem érintett. Az interakciókat definiáló szekvencia és együttműködési diagramokon csak olyan osztályok példányai szerepelhetnek, amely osztályok részei az objektumdiagrammnak.

Összefoglalásul megismételhetjük, hogy az objektum-, a dinamikus és funkcionális modellek az adat, a vezérlés és a funkcionalitás szempontjait érvényesítő leírások. Mindhárom szükséges a rendszer megértéséhez, noha az egyes modellek fontossága az alkalmazási területtől függően változik. A három modell az implementáció során áll össze, és abból alkotható meg a program.

4. fejezet - 4. Fejlesztési módszer

A fejlesztési módszer (design methodology) a szoftverkészítés, előre definiált technikákat és rögzített leírási módokat alkalmazó, szervezett folyamata. A módszer általában végrehajtandó lépések sorozataként jelenik meg, amelyben minden lépéshez definiáljuk az alkalmazandó technikát és jelölésrendszert. Az alkalmazott jelölésrendszer ismertetése jelen könyv 2. és 3. fejezetének tárgya.

A fejlesztési lépések a szoftver életciklus fázisaihoz kapcsolódnak. Az itt közreadott módszer az életciklus nagy részét átfogja; kitér az analízis, a tervezés és az implementáció problémáira, de nem foglalkozik a teszteléssel és a karbantartással. Meglátásunk szerint, mind a tesztelés, mind pedig a karbantartás egyszerűsödik az objektum orientált fejlesztésnél, de nem tér el lényegesen a hagyományos fejlesztési módszereknél alkalmazottól.

Az objektum-orientált módszer a gyors prototípus és az inkrementális típusú fejlesztések esetében egyaránt jól használható. Ennek oka, hogy az objektum-orientáltság a valós világ objektumaihoz kötődik. Attól függően, hogy a modell milyen módon közelít a valósághoz, az egyes objektumok finomításával, vagy az objektumok körének bővítésével, tetszőleges fejlesztési stratégia megvalósítható.

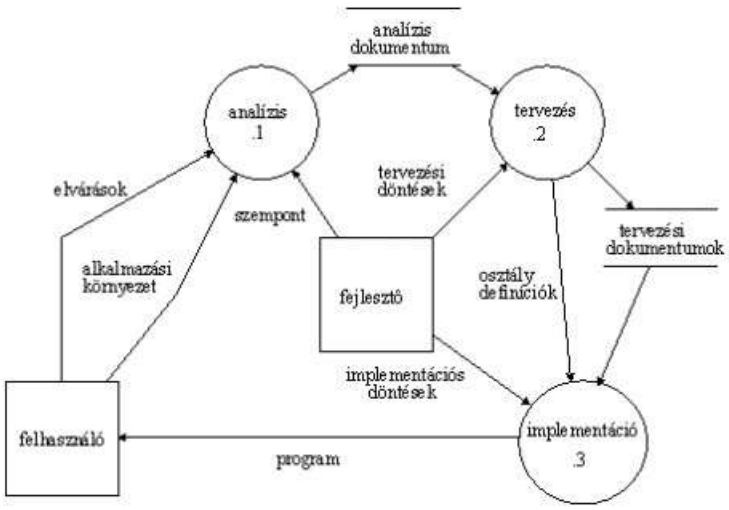
A fejlesztési módszer három fő lépésre tagolható.

1. Az **analízis** célja az alkalmazási terület megragadása, megértése és modellezése az adott terület természetes fogalmainak használatával, azaz a fogalmi modell kialakítása. Az analízis bemenete a feladatkitűzés vagy követelmény-specifikáció, amely leírja a megoldandó problémát és körvonalazza a kialakítandó rendszer céljait. Az analízis során az adott alkalmazási terület szakértőivel és a megrendelővel való folyamatos konzultáció révén pontosítjuk a kiinduló információkat, célokat. Az analízis eredménye egy, a korábban ismertetett három részből (adat, vezérlés, funkció) álló, formális modell.

2. A **tervezés** során analízis lépéstől kapott modelleket leképezzük a rendelkezésre álló hardver- és szoftverelemek szolgáltatásaira. A rendszer globális struktúrájának kialakítása az **architektúrális tervezés** feladata. Az objektum modellből kiindulva a rendszert alrendszerekre bontjuk. Ezzel összefüggésben kialakítjuk az üzenetkezelési mechanizmus, a tárkezelés és a dinamikus modell implementálásának elveit, és döntünk az itt megjelenő párhuzamosság (konkurencia) megvalósításáról. A tervezés második fázisa, az **objektum tervezés**, során a fogalmi modellről az implementációs modellre tevődik át a hangsúly. Kiválasztjuk a fő funkciókat realizáló algoritmusokat, majd az algoritmusok alapján kialakítjuk a hatékonyan implementálható változatot. Az architektúrális tervezéskor kialakított vezérlési rendszer részletes terveit elkészítve, az alrendszereket modulokra bontjuk.

3. A fejlesztési módszer befejező fázisa az **implementáció**, amelyben a tervezés eredményeit egy programozási nyelvre ültetjük át. Az implementáció kérdéseivel az 6. fejezettől kezdve foglalkozunk.

A fejlesztési módszer bemutatását ábrákkal is illusztráljuk. Ezek az ábrák lényegében adatfolyam-diagramok, amelyek megadják a fejlesztés funkcionális modelljét. A teljes módszer funkcionális modellje:



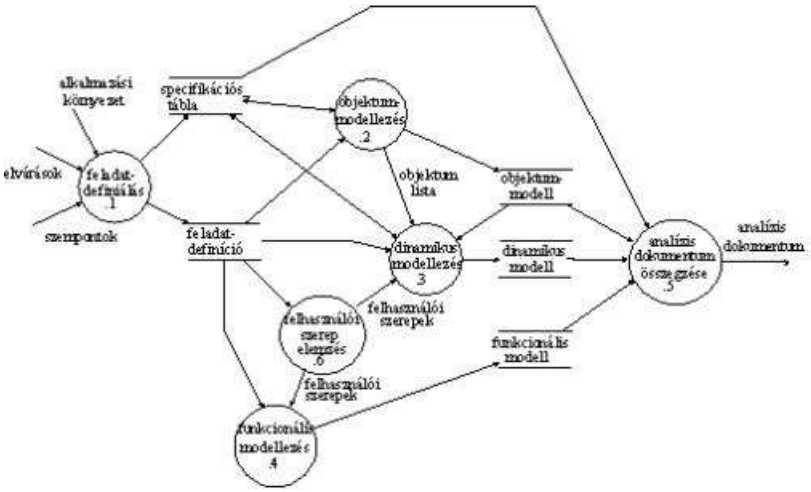
4.1. ábra

A következőkben részletesen áttekintjük az egyes lépéseket.

1. 4.1. Analízis

Az analízis fázisban a valóságnak egy precíz, áttekinthető, egységes és korrekt modelljét hozzuk létre, melynek során megvizsgáljuk a követelményeket, felismerjük az abból származó következményeket és a modellt a leírás szabta szigorúsággal rögzítjük. Az absztrakció eszközének alkalmazásával azonosítjuk a rendszer lényeges jellemzőit. Az analízis eredményeként létrejött modell azt ábrázolja, hogy mit csinál a rendszer, függetlenül annak konkrét megvalósulásától.

Jelen fejezet célja annak bemutatása, hogy a korábban tárgyalt fogalmak segítségével hogyan hozható létre a valóság formális modellje. Ez a modell az ismert három részből, az objektum, a dinamikus és funkcionális modellekből, épül fel. Az analízis modell több célt is szolgál. Egyrészről egyértelműen tisztázza a követelményeket a megrendelő és a fejlesztő számára, másrészről a tervezés és az implementáció is ebből a modellből indul ki.



4.2. ábra

A **feladatdefiniálás** a felhasználói követelmények azonosítását jelenti, amelyből a **feladatdefiníció**, az analízis alapjául szolgáló dokumentum készül. A problématér legfontosabb fogalmait gyakran egy **specifikációs táblázattal** foglaljuk össze, amelyet az analízis előrehaladásával folyamatosan finomítunk. Az alapvető analízislépéseken (objektum, dinamikus és funkcionális modellek készítése) túlmenően a 4.2. ábrán kiemeltük a **felhasználói szerepek elemzését** is. Ez tipizálja a rendszer felhasználóit. A kiadódó felhasználó típusok mind a dinamikus, mind pedig a funkcionális modell kiindulási adatai. Megjegyezzük, hogy a felhasználó, mint objektum, az objektum modellben is megjelenhet, de ott nincs szükség a rendszeren belüli és kívüli objektumok éles megkülönböztetésére. Az objektum, dinamikus és funkcionális modellek elkészítése után, a különböző megközelítésű analízislépések eredményeit össze kell vetni, az ellentmondásokat fel kell oldani, illetve a modelleket kereszt-hivatkozásokkal kell ellátni, és egyetlen dokumentumban összefoglalni. Ezt a lépést az analízis dokumentum összegzésének nevezzük.

Az analízis nem minden esetben hajtható végre egy mereven rögzített szabály szerint. Különösen nagyobb rendszerek esetén iterálni szükséges. A modell első változata tovább bővíthető akár a problématér szélességében – új objektumok bevezetésével – vagy mélységében – a meglevő objektumok finomításával – amíg a teljes problématér áttekinthetővé válik. Az analízis nem egy mechanikus folyamat.

1.1. 4.1.1. A feladatdefiníció

Az analízis első, lényegében előkészítő lépése a feladatdefiníció, amelyben beszélt nyelven megfogalmazzuk, hogy a rendszerrel szemben milyen elvárásaink vannak, mik lesznek a rendszer fő funkciói, milyen alapvető fogalmakkal dolgozunk, és kik lesznek a rendszer felhasználói. A feladat-definíció nem formalizált, de a formális lépések kiindulását jelenti és egyben a fejlesztők és a felhasználók (megrendelők) közötti egyetértés ellenőrzésének az első mérföldköve.

A feladatdefiníció a rendszert általában fekete doboznak tekinti, a rendszer határára, azaz a felhasználók és a rendszer kapcsolatára fogalmaz meg elvárásokat, és pontosítja a megoldandó problémát.

A legtöbb feladat definícióból lényeges információk hiányoznak, amelyeket módszeresen kell összegyűjteni. Az analízis egyik kulcsproblémája a fejlesztők és a rendszer majdani felhasználóinak a korrekt és precíz kommunikációja.

A feladatdefiníciónak a következő témákkal kell foglalkoznia:

- a feladat körvonala, határai, a készítendő rendszer felhasználói,
- szükségletek, igények, elvárások,
- alkalmazási környezet,
- a rendszer alkalmazási környezetére vonatkozó feltételezések,
- teljesítmény, megbízhatóság, prioritások,
- fejlesztési előírások.

A rendszer felhasználóinak azonosítását **felhasználói szerep elemzésnek** nevezzük. A különböző típusú felhasználók a rendszert esetleg különféleképpen használhatják, különböző elvárásokat is támaszthatnak vele szemben (például egy menetrend programot az utazóközönség csak lekérdezésre, a forgalmi ügyeletes pedig a menetrend összeállítására használhat).

A feladatdefiníció a szükségletek jegyzéke, nem pedig a javasolt megoldásoké. A megbízónak definiálnia kell az egyes jellemzők prioritását. A fejlesztés rugalmasságának érdekében tartózkodni kell a tervezési, különösen az implementációs részletek rögzítésétől. Ugyanakkor rendelkezni kell a fejlesztési módszerről, az alkalmazott szabványokról és jelölésekről.

Gyakran előfordul, hogy a feladatkitűzés szintjén a fejlesztés részleteire vonatkozó előírások is megjelennek. Tipikusan ilyen korlátozások közé tartozik a programozási nyelv, illetve a konkrét számítógépes konfiguráció megjelölése, vagy ritkábban bizonyos algoritmusok előírása. Ezek az "álkövetelmények" gyakorta a megbízó hozzá nem értéséből, illetve a feladattal közvetlenül nem kapcsolódó szervezeti rendjéből, üzletpolitikájából erednek. Tekintettel arra, hogy az ilyen típusú döntések egyediek, az analízist végzők felelőssége annak megítélése, hogy milyen feltételekkel vállalható el a munka.

Az objektum-orientált analízis során az elkészült feladatdefiníció feldolgozásának első lépése általában a probléma szereplőinek (objektumok), és a rájuk ruházható felelősségeknek az összegyűjtése. Az első próbálkozásra kiadódó objektumokat tipizáljuk, azaz a hasonló viselkedésű szereplőket egy-egy csoportba osztjuk. Az egyszerű attribútumokat ezen objektum típusokhoz rendeljük, hasonlóképpen a műveleteket, funkciókat ugyancsak az egyes osztályok felelősségi körébe soroljuk. Ennek a lépésnek az eredménye a **specifikációs táblázat**, amely típusonként felsorolja a feladat szereplőit, illetve a szereplőknek a feladatdefinícióból adódó triviális attribútumait és felelősségeit. A specifikációs táblázat általában hiányos, esetleg ellentmondásokat is rejthet magában. Ez nem baj, mert a specifikációs táblázat elsődleges célja az, hogy a további analízis lépések (objektum, dinamikus és funkcionális modellek felépítése) számára kiindulási alapot teremtsen, és a legfontosabb fogalmakat elnevezze.

Másrészről a specifikációs táblázatot az analízis során folyamatosan kiegészítjük és pontosítjuk a különböző modellek információinak felhasználásával. Így az analízis végére a specifikációs táblázat kezdeti hiányosságai és ellentmondásai eltűnnek.

1.2. 4.1.2. Objektummodellezés

A feladatdefiníciót követő első lépés az objektum modell kidolgozása. Az objektum modell leírja a valóságos rendszer objektumait és az objektumok között levő kapcsolatokat. Az objektum modell kialakítása általában könnyebb mint a dinamikus és funkcionális modelleké, ezért először célszerű ehhez hozzáfogni. Az objektum modell felépítésének viszonylagos egyszerűsége abból adódik, hogy a statikus struktúrák könnyebben értelmezhetők, jobban definiáltak és kevésbé függnek az alkalmazás részleteitől mint a dinamikus jellemzők.

A objektum modell felállításához szükséges ismeretek elsődlegesen a feladatdefinícióból és az alkalmazási terület szakértőitől származnak.

A modellezés során elsőként az osztályokat és asszociációkat azonosítjuk, mivel ezek határozzák meg a rendszer általános struktúráját, majd ezekhez adjuk hozzá az attribútumokat. Csak ezt követően érdemes az öröklési struktúrákat keresni. Az öröklés korai erőltetése könnyen a valóságtól eltérő, torz objektumszerkezetet eredményezhet, mivel ilyenkor a valóság szem előtt tartása helyett a saját prekoncepciónkat helyezzük előtérbe.

A modell kialakítása általában iterációban történik. Ezen tevékenységünk során a következő alfejezetekben részletezett lépéseket egyszer vagy többször végre kell hajtani.

1.2.1. 4.1.2.1. Az objektumok és az osztályok azonosítása

Az objektum modell megalkotásának első lépése a fogalmi tér lényegi objektum-típusainak (osztályainak) felismerése. Az objektumok lehetnek fizikailag létező egyedek, mint például autók, gépek, emberek valamint különféle szerepet játszó dolgok (adóalany, végrehajtó), esetleg történések (előadás, baleset, stb.). Az objektumoknak szigorúan a fogalmi térből illetve a köznapi ismertek területéről kell származniuk és kerülni kell a számítástechnikával kapcsolatos fogalmakat.

Az objektumok felderítése a feladatdefiníció nyelvi elemzésével indul, amely a probléma szereplőit – tehát az elvárt tevékenységek végrehajtóit és tárgyait – tartalmazó lista áttekintésével kezdődik. Nem kell túlzottan válogatni, minden tartalommal rendelkező főnevet ki kell emelni. Az típusokhoz úgy jutunk el, ha a felsorolt objektum-egyedek között felismerjük az azonos tulajdonságokkal és viselkedésekkel rendelkezőket, és azokhoz egy osztályt rendelünk. Az öröklést és a magas szintű osztályokat figyelmen kívül kell hagyni.

A listából a szükségtelen és értelmetlen osztályok az alábbi kritériumok figyelembe vételével elhagyhatók és egy redukált osztálylista képezhető.

- Redundáns osztályok. Ha két vagy több fogalom ugyanazt fejezi ki, akkor a kifejezőbb – gyakran rövidebb, egyszerűbb – megnevezést kell választani.
- Nem lényeges (nem releváns) osztályok. Ha egy osztálynak nincs vagy kicsi a jelentősége, akkor az elhagyható. Ennek eldöntése erősen függ a megoldandó feladat környezetétől. Egy áru becsomagolására felhasznált csomagolóanyagnak nincs különösebb jelentősége a vásárlásnál, annál inkább a bolt csomagolóanyaggal való ellátásában.
- Határozatlan osztályok. Egy osztálynak specifikusnak kell lenni. A semmitmondó, körvonalazatlan objektumokat – mint például alrendszer, költség – el kell távolítani.
- Attribútumok. Az "objektum vagy attribútum" kérdésre annak alapján tudunk válaszolni, hogy megvizsgáljuk, a dolog önálló létezése a rendszer szempontjából fontos-e. A cipő lehet egy ember attribútuma, de lehet önálló objektum is, ha a viselőjétől függetlenül önállóan létezik, azaz akciókat okoz illetve szenved el (megsarkalják).
- Műveletek (operációk). Ha egy megnevezés tevékenységet jelent és ez a tevékenység az adott alkalmazásban más objektumra irányul, akkor az általában nem objektum. Például a tüdőszűrés az emberen értelmezett művelet, az emberre irányuló tevékenység.
- Szerepek. Egy elnevezésnek a dolgot magát kell jelölnie, nem pedig a dolog által játszott szerepet. A 3.2.2.4. pontban tárgyaltuk az objektum-szerepeket, amelyek az asszociációhoz kötődnek. Például egy vasúti rendszerben a munkára vezénylés alkalmával a forgalmi ügyeletes az alkalmazott által eljátszandó szerep.
- Implementációs fogalmak és eszközök. A megoldási tér elemei (memória, algoritmus, megszakítás, stb.) az analízis során nem objektumok.

A redukált osztálylistában szereplő elnevezések mögött lényeges tartalom található, amely tartalmat az egyértelműség érdekében írásban is rögzíteni kell. A listában javasolt objektum típusokhoz az értelmező szótári bejegyzéshez hasonló egy-két bekezdésből álló fogalommagyarázatot kell készíteni. Ebben természetesen célszerűen benne lehetnek az objektumok attribútumai, relációi és műveletei.

Az osztályok azonosításának bevett technikái közé tartozik az úgynevezett CRC kártyák használata. Ezek az általában két névjegy nagyságú cédulák három mezőt tartalmaznak, nevezetesen

```
osztály         Class,
felelősség      Responsibility,
együttműködők   Collaborators,
```

A kártyákat ötletroham keretében az analízist végző csoport tagjai töltik ki. Minden valószínűsíthető osztályra vonatkozóan kiállítanak egyet, megadva az objektumosztály által viselt felelősséget és az együttműködők címszó alatt a relációban álló más objektumokat. A kártyákat egy táblára kitűzve, azokról vitát folytatva, a kártyákat cserélgetve, módosítva kialakítható egy előzetes objektumhalmaz.

1.2.2. 4.1.2.2. Az asszociációk azonosítása

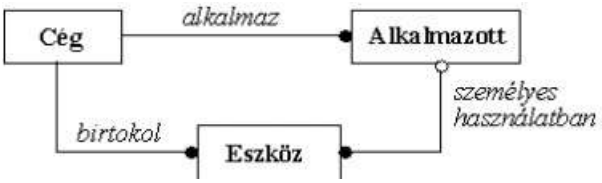
A valóságos dolgok közötti viszonyt a modellben az **asszociációk** írják le. Az asszociáció egy hivatkozás az egyik objektumból egy másikra. Az objektumok közötti viszonyt leggyakrabban igék vagy igei kifejezések jelenítik meg. Ezek vonatkozhatnak

- fizikai elhelyezkedésre (következő, része, alkotja, stb.),
- tárgyas igékkel kapcsolatos cselekvésre (vezeti, leveszi, stb.),
- kommunikációra (üzeni, átadja, stb.),
- birtoklásra (van neki, része, hozzá tartozik, stb.),
- előírt feltételeknek való megfelelésre (tag, alkalmazásban áll, rokona, stb.).

Az objektumok felderítéséhez hasonlóan az asszociációk kezdeti listájából az alábbi meggondolásokkal a szükségteleneket el kell hagyni:

- Elhagyott osztályok közötti asszociációk. Az asszociációk csak olyan osztályokra vonatkozhatnak, amelyek az objektum listában szerepelnek. Ha a hivatkozott osztály a listában nem szerepel, akkor vagy fel kell azt oda venni, vagy az asszociációt másként kell megfogalmazni.
- Nem releváns vagy implementációval összefüggő asszociációk. Minden olyan kapcsolatot, amely az alkalmazási területen kívülre mutat – ide értve az implementációt is – meg kell szüntetni.
- Akciók. Az asszociációk az objektumok közötti statikus viszonyt modellezik, nem pedig az esetenkénti történést. Az akciókban szereplő objektumok közötti statikus viszonyt kell felderíteni. Például a taxi és az ember között értelmezett asszociáció az utazik, nem pedig a beül esemény.
- Többszörös relációk. Különös figyelemmel kell lenni a többszörös relációkra. Az esetek legnagyobb részében ezek fölbonthatók bináris relációkra. A gyakorlatban felbukkanó esetekben alapos vizsgálatot kell végeznünk, hogy a bináris relációkra bontás információveszteséggel jár-e. A vizsgálat megfelel az adatbázis-kezelés 5NF (ötödik normál forma) elemzésének.
- Származtatott asszociációk. Ha egy asszociáció teljes mértékben definiálható más asszociációk felhasználásával, akkor – mivel redundáns – érdemes elhagyni. Példaként tekintsük a sógor relációt, amely előállítható a házastárs és a testvér asszociációk kompozíciójaként. Hasonló módon elhagyandók az objektumok attribútumain értelmezett feltételekből származtatott asszociációk.

Ügyelni kell arra, hogy ha egy asszociáció ténye más asszociációkból következik, ez még nem feltétlenül definiálja a kapcsolat multiplicitását. Példaként vegyünk egy céget, amelynek vannak alkalmazottai és eszközei. Az alkalmazottaknak a munkájukhoz szükséges eszközöket (0 vagy több) személyes használatra átadták, míg más eszközök közös használatban vagy használaton kívül vannak. A probléma objektum modellje a 4.3. ábrán látható.



4.3. ábra

Az *alkalmaz* és a *birtokol* asszociációkból nem származtatható a *személyes használatban* asszociáció számossága.

1.2.3. 4.1.2.3. Az attribútumok azonosítása

Az **attribútumok** az egyedi objektum tulajdonságait fejezik ki. Az attribútumot általában az objektumot jelentő főnévhez kapcsolt jelzős vagy birtokos szerkezet definiálja, például "*a gyerek magassága*" vagy "*az autó rendszáma*". Az attribútum meghatározása során definiáljuk az általa felvehető értékek halmazát. Általános szabály, hogy csak az alkalmazás szempontjából jelentőséggel bíró attribútumokat vegyük fel, és azok közül is elsőként a legfontosabbakat, majd a modell finomítása során a kevésbé jelentőseket.

Az attribútumok és az osztályok felvételének helyességét az adatbázis tervezésben használt normalizáltsági vizsgálatokkal ellenőrizhetjük. Kerülnünk kell egy objektumon belül az attribútumok közötti **funkcionális függőséget**. Például ha egy számítógép attribútumai a diszk típusa és a diszk kapacitása, akkor ilyen funkcionális függés áll fenn a két attribútum között, hiszen a diszk kapacitása a diszk típusához kötődik, azaz a másik attribútumhoz nem pedig a számítógéphez, azaz a tartalmazó objektumhoz. A származtatott attribútumokat (például a kor, ha a születési idő attribútum, hiszen a kor a születési időből mindig kiszámítható) ugyancsak kerüljük!

Az objektumok létükből fakadóan megkülönböztethetők egymástól. Következésképp szükségtelen külön azonosítóval (netán többel is) megjelölni őket, kivéve ha az azonosító használata az alkalmazási területen szokásos. Ilyennek tekinthetjük a személyi számot, vagy egy bankszámla számát.

1.2.4. 4.1.2.4. Az öröklési hierarchia létrehozása

A modell kialakításának következő lépése az öröklési hierarchiák keresése, ami két irányban is történhet. Egyfelől a meglevő osztályok közös tulajdonságainak összevonásával alaposztályokat kereshetünk (alulról fölfelé), másfelől meglevő osztályokból származtatott osztályokat képezhetünk finomítással (fentről lefelé).

Az alaposztályok keresése céljából hasonlítsuk össze az osztályokat és keressünk az attribútumok és metódusok között hasonlóságot. Amennyiben találunk ilyet, vizsgáljuk meg, hogy egy, a közös részeket kiemelő osztály létrehozása értelmes-e, indokolható-e az adott alkalmazásban.

A származtatott osztályok keresése során tételezzük fel, hogy az osztályok általánosak (gyűjtőfogalmat jelenítenek meg). Vizsgálnunk kell, hogy újabb attribútumok és metódusok hozzávételével milyen, a feladat szempontjából értelmes osztályok hozhatók létre. Az attribútumokat és az asszociációkat mindig a legáltalánosabb osztályhoz rendeljük.

1.2.5. 4.1.2.5. Az objektummodellezés termékei

Az objektum modellezés során a következő dokumentumokat kell elkészíteni:

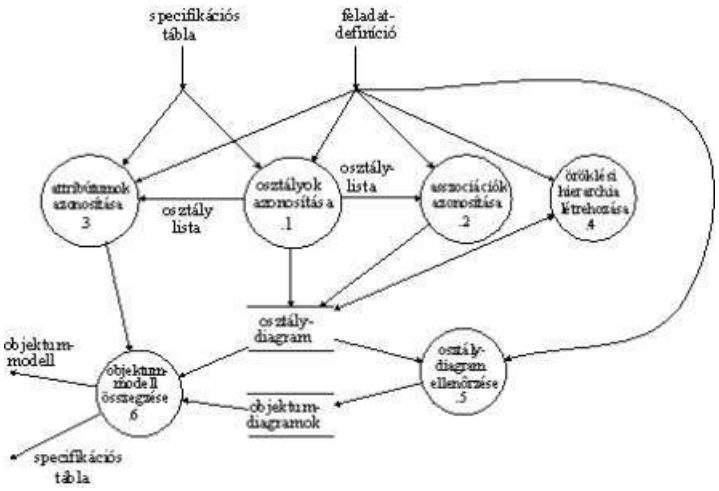
- Objektum modell, amely egy osztálydiagramból és szükség esetén az osztálydiagramot megmagyarázó objektumdiagram(ok)ból áll.
- A létrehozott osztályok leírása. Az osztályleírás tulajdonképpen a specifikációs táblázat kiegészítése és finomítása, melynek szerkezete az objektum modellezés után:

Név:	az osztály megnevezése.
Felelősségek:	az objektumosztály által viselt felelősségek

	szöveges leírása.
Példányok:	0/1/n. 0/1/n. Ha absztrakt osztályt definiálunk, akkor lehetséges, hogy az osztály nem példányosodik.
Alaposztályok:	az öröklési hierarchiában az adott osztály fölött álló osztályok listája.
Komponensek:	az osztállyal komponensrelációban álló osztályok és objektumok listája.
Változók:	az osztály attribútumainak megnevezése és típusa.
Relációk:	asszociáció révén kapcsolt osztályok listája.

Az alkalmazási területtől és az adott osztálytól függően nem mindegyik pontra lehet és szükséges választ adni. A specifikációs tábla bejegyzései jelentősen módosulhatnak az iteráció során.

Összefoglalva, az objektum modellezés tevékenységei a következő adatfolyam-ábrán tekinthetők meg.



4.4. ábra

Az objektumdefiníciókkal és az objektumdiagrammal ábrázolt modell teljességét ellenőrizni kell. Ebből a célból tesztelni kell a modellt, amit célszerűen a modellnek feltett kérdésekkel tehetünk meg. A kérdések vonatkozhatnak az egyes példányok és azok attribútumainak elérhetőségére, valamint az objektumok közötti relációkat modellező asszociációk meglétére.

Amennyiben a modellnek feltett kérdésekre nem kapunk kielégítő válaszokat, akkor a modellt módosítani kell, az eddig megismert lépések ismételt végrehajtásával. Ezen ellenőrzés elvégzését és a szükséges dokumentációk összefogását a fenti ábrán "összegzés"-nek nevezzük.

1.3. 4.1.3. Dinamikus modellezés

A dinamikus modell a rendszernek és a rendszer objektumainak időbeni viselkedését írja le. A dinamikus modellezés a rendszer határain kívülről érkező üzenetek (események) és az arra adott válaszok vizsgálatával kezdődik. A rendszer határán jelentkező eseményeket **forgatókönyvekben** foglaljuk össze. Majd a rendszer és a környezet közötti kommunikációt kiterjesztjük a rendszerben lévő objektumok párbeszédének a vizsgálatára. Ezt követi objektumonként az állapotdiagram felvétele. Az állapotgép megvalósítása nem tartozik az analízishez, hiszen az csak a fejlesztés későbbi fázisában válik véglegessé. A valós idejű rendszereket kivéve, az időbeni viselkedés nem az időbeni pontosságot, hanem az események sorrendiségét jelenti.

1.3.1. 4.1.3.1. Forgatókönyvek összeállítása

A dinamikus modell felépítését a rendszer határfelületén végbemenő kommunikáció feltérképezésével kezdjük. Ehhez, a rendszerdefiníció elemzésével azonosítjuk a felhasználók és a rendszer között végbemenő interakciós eseményeket, amelyeket **eseménylistákban** foglalunk össze. A felhasználókat ezen modellezési lépést megelőző "felhasználói szerep elemzés" határozza meg. Minden, a rendszerből vagy a felhasználótól származó, pillanatszerű "történés" – legyen az bármilyen, a vezérlésre hatást gyakorló beavatkozás vagy kijelzés – **eseménynek** számít. Az eseménynek van keletkezési helye és észlelője. Az események feltérképezésénél azok bekövetkezése a fontos, nem pedig az esemény során átvitt adat. Az események feltérképezése után az eseménylista alapján a rendszer külső szereplői és a rendszer közötti jellemző párbeszédeket állítjuk össze, amelyeket **forgatókönyvnek** nevezünk. A forgatókönyvben lényegében azt határozzuk meg, hogy a rendszer felületén az események milyen lehetséges sorrendben történhetnek meg. A feladat definíció alapján felvett eseménylisták általában csak a normál üzem eseményeit tartalmazzák. Ezeket ki kell egészíteni a rendkívüli helyzetek, hibák, stb. megjelenését és kezelését jelképező eseményekkel.

A forgatókönyvek a rendszert továbbra is fekete doboznak tekintve a rendszer határán írják le a vezérlési és sorrendi kérdéseket. A rendszer belsejének feltárása során a rendszer határon jelentkező eseményeket belső eseményekre kell visszavezetni.

1.3.2. 4.1.3.2. Kommunikációs diagramok felvétele

A **kommunikációs diagramok** a forgatókönyvek kiterjesztései a rendszer belsejében működő objektumokra. Minden egyes forgatókönyvhöz meg kell keresnünk azon objektumokat, amelyek közvetlenül vagy közvetve részt vesznek a külső események feldolgozásában és a reakciók létrehozásában. Ezeket az objektumokat a kommunikációs diagramon egy-egy függőleges vonal jelöli, amelyen az idő fölülről-lefelé halad. A kommunikációs diagramon az eseményeket a keletkezési helyet jelentő objektumtól az észlelő objektumig tartó nyíl jelöli. Az esemény úgy is értelmezhető, hogy a keletkezési helyről egy **üzenetet** küldünk az észlelőnek. A továbbiakban az események helyett gyakran használjuk az üzenetküldés fogalomrendszerét is.

Előfordulhat, hogy ugyanolyan típusú objektumból több is részt vesz egy adott esemény feldolgozásában. Ebben az esetben valamennyit szerepeltetni kell a kommunikációs diagramon. A rendszer külső objektumait szintén fel kell tüntetnünk ezen az ábrán. A külső objektumokból származó és a külső objektumokhoz eljutó események valamint azok sorrendje a forgató-könyvekből kiolvasható. A kommunikációs diagram felépítése során megnézzük, hogy a kívülről érkező eseményt melyik belső objektum észleli és a kívülre távozó esemény melyik belső objektumtól származik. Ennek a belső objektumnak viszont valószínűleg segítségül kell hívnia más objektumokat is, ami újabb belső események létrehozásával jár. Az eljárást addig kell folytatni, amíg az egyes forgatókönyveket a rendszer belsejében végbemenő párbeszéddel megnyugtatóan ki nem egészítjük. A megnyugtató jelző az objektumok funkcionális képességeire vonatkozik. Nevezetesen az objektum modell alapján már van egy kezdetleges képünk arról, hogy az objektumok milyen információt tárolnak. Ez alapján ellenőrizhetjük, hogy az objektum egy eseményre képes-e egymagában reagálni, vagy más objektumokban elhelyezett információk miatt azokhoz kell fordulnia segítségért.

Néha előfordul, hogy a kommunikációs diagramon egy objektum önmagának küld eseményt. Ez esetenként megtűrhető, elburjánzása viszont arra utal, hogy a modellezés során nem tudtunk elvonatkoztatni az algoritmikus kérdésektől, azaz attól, hogy egy esemény hatására milyen belső feldolgozási lépéseket hajtsunk végre. Ilyen esetekben próbáljuk az objektumok fekete doboz jellegét jobban kiemelve a saját üzenetek számát csökkenteni.

A kommunikációs diagram felépítése során általában csak az események sorrendjére koncentrálunk, az események közvetlen ok-okozati kapcsolatait gyakran figyelmen kívül hagyjuk. Az egyes objektumokat mint önálló, konkurens szereplőknek tekintjük, amelyek akkor küldenek üzenetet másoknak, amikor erre szükségük van, amit az objektum pillanatnyi belső állapota – azaz végső soron a születésétől kezdve kapott eseménysorozat – határoz meg.

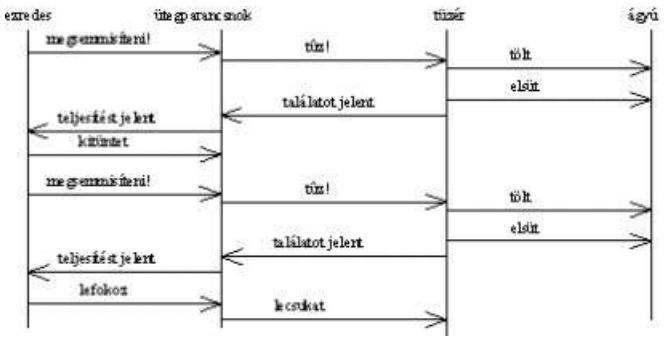
Az objektumok működésének elemzésénél azonban nem mindig kell ezt a rendkívül általános modellt használni. Annak eldöntéséhez, hogy egy objektum eseményt akar-e küldeni a mások számára, gyakran egyetlen, az objektumot érő üzenet ismerete elegendő. Ilyen esetben az objektumhoz érkező és az objektumtól származó események ok-okozati összefüggéséről beszélünk.

Az ok-okozati viszonyok bevezetésével az egyes objektumok, elvileg végtelen hosszú, kommunikációs eseménysorozatait egymástól vezérlési szempontból független véges **interakciós sorozatokra** bonthatjuk.

Valamely objektum interakciós sorozata azzal kezdődik, hogy az egy üzenetet (eseményt) kap egy másik objektumtól (ezt az eseményt nevezzük a sorozat **aktiváló eseményének**), melynek hatására ez az objektum más objektumokat vesz célba üzeneteivel. Az interakciós sorozat gyakran azzal fejeződik be, hogy a kiválasztott objektum egy válaszüzenetet küld az aktiváló esemény forrásának. A kiválasztott objektum által küldött események ok-okozati összefüggésben vannak az aktiváló eseménnyel, ha ezek ennek és csakis ennek a hatására keletkeztek. Az interakciós sorozatok azon jellemzője, hogy egymástól függetlenek, arra utal, hogy az objektum az aktiváló üzenet előtt, vezérlési szempontból mindig ugyanabban az állapotban van, és az interakciós sorozat végén is mindig ide tér vissza. Ez azt jelenti, hogy vezérlési szempontból az objektumok szerepét az interakciós sorozatban úgy is elképzelhetjük, hogy az objektumok minden interakciós sorozat során az aktiváló üzenet hatására újraszületnek és a sorozat végén pedig elhaláloznak. Az objektumok véges interakciós sorozatát a kommunikációs diagramon **ok-okozati téglalapokkal** ábrázoljuk.

Az ok-okozati összefüggések feltárása pontosíthatja a kommunikációs diagram értelmezését, valamint az első közelítésben autonóm és konkurens viselkedésű objektumok jelentős részét – vezérlési szempontból – más objektumok alá rendelheti. Egy ilyen alárendelt objektum aktivitását nagyrészt "fölöttesei" határozzák meg, az alárendelt objektum aktív lényből passzív válaszadóvá minősíthető vissza. Ily módon a kommunikációs modell indításakor feltételezett, a feladatból nem következő párhuzamosságokat eltávolíthatjuk. Ennek a tervezésnél és az implementációnál lesz nagy jelentősége, hiszen mint látni fogjuk, a rendszerben lévő párhuzamosságok különleges megfontolásokat igényelnek.

Tekintsük példaként a következő kommunikációs modellt.

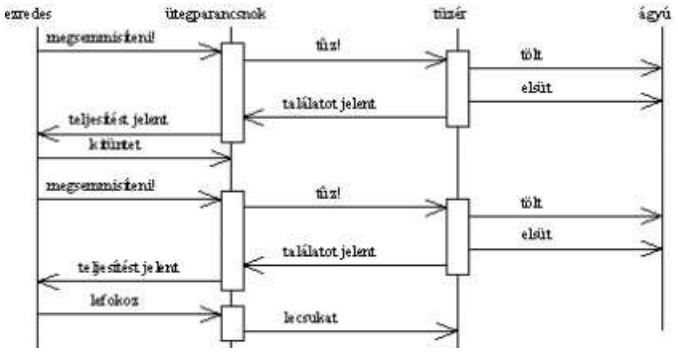


4.5. ábra

A *tüzér* mindig a tűzparancsra *tölt, elsüti* az ágyút, majd *jelent*, tehát ezek az események a "*tűz!*" ok-okozati következményei. Ráadásul ezután ugyanolyan alapállapotba jut, mint ahonnan elindult, nevezetesen tétlenül várakozik újabb parancsokra. Hasonlóképpen az *ütegparancsnok* élet-ciklusát is fel lehet osztani:

- az ezredestől kapott utasítás vétele és a parancs teljesítésének jelentése közötti tevékenységre,
- kitüntetés átvételére,
- lefokozás elszenvedésére, melynek hatására a tüzért lecsukatja.

A kommunikációs diagramot az interakciós sorozatok jelzésével a következőképpen rajzolhatjuk át:



4.6. ábra

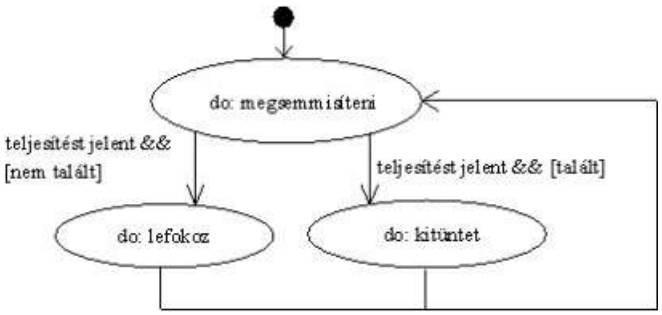
1.3.3. 4.1.3.3. Állapotmodellek készítése

A kommunikációs modellek az objektum – objektum kapcsolatok időbeli alakulását vizsgálják. Az **állapotmodellek** ezzel szemben egy-egy objektumot elemeznek, arra keresve a választ, hogy az objektum az események hatására hogyan változtatja meg a belső állapotát, valamint az események hatására milyen üzeneteket küld a többi objektum számára. Az elemzést osztályonként egyszer kell elvégezni, hiszen minden, az osztályhoz tartozó objektumnak ugyanolyan viselkedése van. Az állapot meghatározza, hogy a jövőben érkező eseményekre adott választ hogyan befolyásolják a múltban érkezett események.

Az egyes osztályok állapotmodelljének felépítése a kommunikációs diagramok alapján kezdődik. Először ki kell gyűjtenünk az adott osztály objektumait jelképező függőleges vonalakat, a kapott és küldött üzeneteket jelképező nyilakkal együtt. A kapott üzenetek forrása és a küldött üzenetek célja az állapotmodell szempontjából lényegtelen (ezeket a kommunikációs modell fejezi ki). Induljunk ki egyetlen kommunikációs diagramrészletből, végignézve az objektumhoz érkező, és azt elhagyó üzeneteket. Az állapotmodellben az állapotok két esemény között írják le az objektumot. Ha két egymást követő állapot között az objektum kívülről üzenetet kap, akkor az állapotátmenet eseményeként ezt az eseményt kell megadni. Amennyiben a két állapot között az objektum küld üzenetet, az átmenetet során elvégzendő akcióként adjuk meg ezen üzenet elküldését. Gyakran ennél kézenfekvőbb lehetőség az, hogy az üzenet elküldését az állapot aktivitásaként képzeljük el.

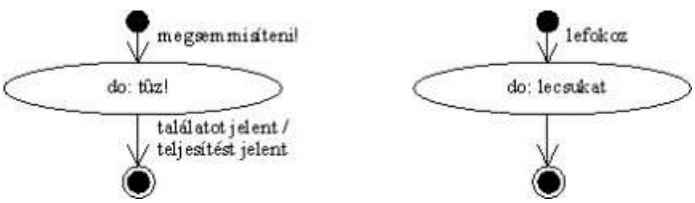
Előfordulhat, hogy a kommunikációs diagram ciklusokat tartalmaz. Ilyen esetekben a bejárt állapotokat csak egyszer kell létrehozni, és az állapot-átmeneteket úgy kell kialakítani, hogy a ciklus végén az objektum újra a ciklus elején lévő állapotba kerüljön.

Példaként tekintsük az ezredes állapotmodelljét, feltételezve, hogy az ezredes ciklikusan *megsemmisítés* parancsokat osztogat, és az eredmény függvényében jutalmazza a beosztottait:



4.7. ábra

Amennyiben a kommunikációs diagram ok-okozati téglalapokkal történő kiegészítése során sikerül a kiszemelt objektum viselkedését egymástól független interakciós sorozatokra bontani, jelentősen egyszerűsíthetjük az állapotterek szerkezetét. Mint említettük, egy ilyen interakciós sorozatban vezérlési szempontból úgy tekinthetjük az objektum viselkedését, mintha az csak erre az interakciós sorozatra születne meg. Ez azt jelenti, hogy az interakciós sorozatokra bontott objektumot nem egyetlen bonyolult állapottér modellel kell specifikálni, hanem annyi darab független állapottér modellel, ahány különféle interakciós sorozata van (azaz ahány különféle aktiváló eseménye van). Az ütegparancsnokot például két független állapotmodellel jellemezhetjük, melyek aktiváló eseményei a "*megsemmisíteni!*" és a "*lefokoz*" (a "*kitüntet*" nem hoz létre újabb eseményeket így nem szükséges állapotmodellel pontosítani):



4.8. ábra

Természetesen vannak olyan objektumok is, melyek működését nem tudjuk interakciós sorozatokra bontani (ilyen a fenti példában az ezredes). Ezekhez általában egyetlen, de meglehetősen komplex állapottér tartozik.

Az ismertetett módszerrel egy adott objektum típus egyetlen kommunikációs diagrambeli szerepét dolgoztuk fel. Figyelembe kell vennünk azonban az ugyanilyen típusú objektumok ugyanebben a kommunikációs diagramban betöltött más szerepeit és a többi kommunikációs diagramot is. Ez a lépés az eddig létrehozott állapotterek új állapotokkal történő bővítését illetve az állapotátmenetek feltételeinek felvételét és bővítését eredményezi. Először meg kell vizsgálnunk, hogy definiáltunk-e olyan állapotot, amelyben az objektum az új kommunikációs diagram elején áll (általában minden forgatókönyv ugyanonnan indul). Ezután a (kapott és küldött) üzenetek alapján lépegetni kezdünk az állapotgépen, mindig megvizsgálva, hogy az adott állapotban felvettünk-e már az eseménynek megfelelő állapotátmenetet, és hogy a következő állapot megfelel-e a vizsgált kommunikációs diagramnak. Egy adott állapotban új eseményt (üzenetet) úgy vehetünk figyelembe, hogy egy az eseménynek megfelelő új állapotátmenetet definiálunk. Az átmenet célállapota lehet már definiált állapot is, amennyiben az állapotok jelentése szerint az elfogadható, vagy ellenkező esetben az állapotteret új állapottal kell bővíteni.

Az újabb kommunikációs diagramok beépítése során az is előfordulhat, hogy egy állapotban már szerepel a bejárás során felismert esemény, de a korábban felvett célállapot nem felel meg az aktuális forgatókönyvnek. Az ilyen ellentmondásokat az állapotátmenethez rendelt feltételek bevezetésével oldhatjuk fel.

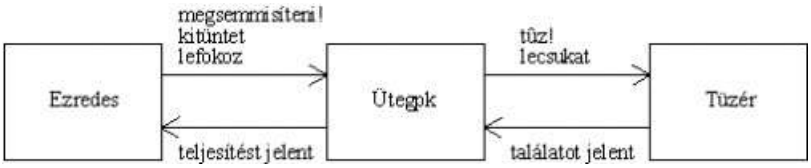
Meg kell jegyeznünk, hogy azon osztályok esetén, melyek viselkedését véges interakciós sorozatokra, és ezáltal az állapotmodelljét különálló állapotmodellekre bontottuk, az újabb és újabb kommunikációs modellek beépítése sokkal egyszerűbben elvégezhető. Ekkor ha egy újabb aktiválási eseménnyel találkozunk, ahhoz egy új, független állapottérmodellt definiálunk. A független állapotmodellek úgy épülnek fel, mintha az aktiválási esemény hatására születne meg az objektum, és az interakciós sorozat végén el is halálozna. Amennyiben a vizsgált aktiválási eseményhez már felvettünk állapotteret, meg kell vizsgálnunk, hogy annak viselkedése megfelel-e az újabb kommunikációs modellnek. Ha megfelel, akkor az állapotmodellt nem kell bővítenünk, ha viszont nem felel meg, csak ezt a különálló állapotmodellt kell kiegészítenünk.

1.3.4. 4.1.3.4. Eseményfolyam-diagram készítése

A kommunikációs diagramon objektumok kommunikálnak egymással. Azonban a kommunikációs képességet – azaz azt a tulajdonságot, hogy egy objektum eseményeket képes fogadni és azokra reagálni – az objektum osztályok hordozzák. A kommunikációs diagramon ugyanolyan típusú objektumok különböző szerepekben jelenhetnek meg és ennek megfelelően látszólag különböző üzeneteket fogadhatnak. Az objektumokat definiáló osztályt viszont minden szerepre fel kell készíteni.

Ezért célszerű az osztályok kommunikációs képességeit is összefoglalni, amit az **eseményfolyam-diagram** elkészítésével tehetünk meg. Az eseményfolyam-diagramon osztályok szerepelnek, amelyeket nyilakkal kötünk össze aszerint, hogy az egyes osztályokból definiált objektumok mely más osztályokból létrehozott objektumoknak küldenek üzenetet. Az eseményfolyam-diagram nem pillanatkép, hanem potenciálisan a teljes működés alatti párbeszédeket tartalmazza anélkül, hogy az események sorrendjét firtatná.

Előző példánkban, ha az ezredes objektum típusa *Ezredes*, az üteg-parancsnokok típusa *Ütegpk*, a tüzérek típusa *Tüzér*, akkor a problémát leíró eseményfolyam-diagram:



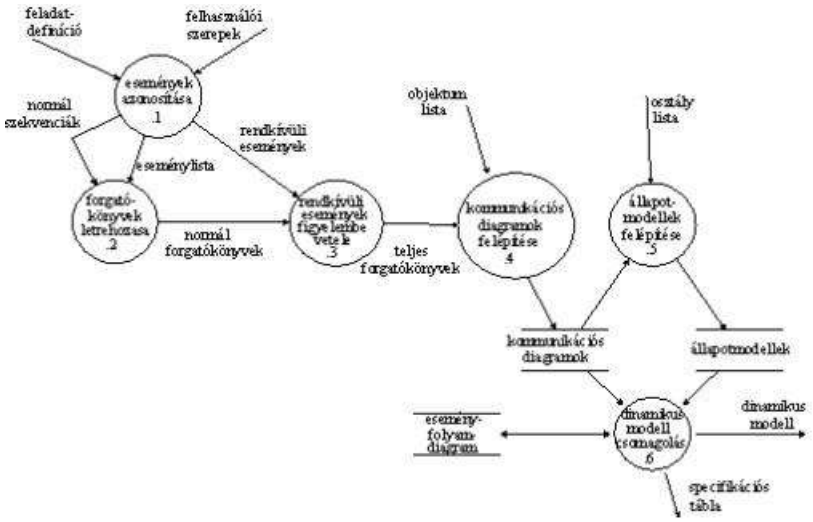
4.9. ábra

Az eseményfolyam-diagram az osztályok viselkedését (kapcsolódási felületét) pontosítja. Egy objektum-orientált programban az események átadása általában tagfüggvény hívásokkal történik. Kivételt képeznek ez alól a független interakciós sorozatok utolsó válasz eseményei, melyek az aktiválási esemény párjaként, a feladó és címzett viszony megfordításával jönnek létre. Ezeket a válasz eseményeket az aktiválási eseményt realizáló metódus visszatérési értékeként célszerű megvalósítani.

Az események üzenetként és visszatérési értékként történő csoportosítása után az egyes osztályoknak az objektum modell dokumentumaiban megadott kezdetleges leírását kiegészíthetjük a dinamikus analízis során megszerzett információkkal:

```
Név:              az osztály megnevezése
Szolgáltatások:   az osztály metódusainak deklarációs listája
Konkurencia:      az objektum dinamikus működésének jellemzése
Perzisztencia:    perzisztensnek nevezzük azokat az objektumokat,
                  amelyek élettartama túlnyúlik a program futtatásán
Teljesítmény:     az objektumra vonatkozó időzítési korlátok
```

Összefoglalva, a dinamikus modell felépítésének tevékenységei a következő adatfolyam-diagramon követhetők:



4.10. ábra

1.4. 4.1.4. Funkcionális modellezés

A funkcionális modell a "mivel mit csinál" kérdésre összpontosítva arra keresi a választ, hogy a rendszer az eredményeit milyen módon állítja elő. Megadja, hogy a bemeneteket milyen transzformációknak (ún. folyamatoknak) kell alávetni és ezen transzformációk milyen részeredményeket szolgáltatnak egészen az eredmények előállításáig.

A funkcionális modell a "mit csinál" kérdést anélkül válaszolja meg, hogy belemenne abba, hogy a transzformációknak milyen vezérlés szerint kell követniük egymást. A vezérlés a dinamikus modell területe, a vezérlési információkat lehetőség szerint a funkcionális modellből el kell távolítani.

A funkcionális modell kialakításának kezdetekor ugyancsak a feladatdefinícióhoz fordulhatunk. Először egyrészt célszerű előállítani a rendszertől elvárt funkciókat felsoroló **funkciólistát**. A funkciókat a felhasználói szerepeknek megfelelően csoportosítjuk. Másrészt össze kell gyűjtenünk a felhasználók és a rendszer között átadott **bemeneti és kimeneti adatokat.** Szemben a dinamikus modell megközelítésével, most a rendszer és a felhasználók közötti kommunikáció adattartalma érdekel bennünket. A felhasználó típusokat a felhasználói szerep elemzés eredményei határozzák meg.

Felhasználónként csoportosítjuk a bevitt és elvárt adatokat. A kapott eredményt az egyetlen folyamatot és a külső szereplőket tartalmazó adatfolyam-ábrán, ún. **kontextus-diagramon**, foglaljuk össze. A kontextus-diagram tisztázza a rendszer alapvető interfészeit. Az interfészeken belüli folyamatokat a következő szintű funkcionális modellel kell leírni.

Az interfészek közötti űr kitöltéséhez hozzákezdhetünk a kimenetek vagy a bemenetek felől egyaránt:

- Induljunk ki a rendszer kimeneti adatfolyamataiból (egyenként) és nézzük meg, hogy azok milyen más részeredményekből állíthatók elő (mitől függnek). Definiáljuk az előállítási folyamatot úgy, hogy annak bemenetei a részeredmények, kimenete pedig a kiválasztott rendszerkimenet. Az előállítási folyamatot végiggondolva, rendeljünk a folyamathoz nevet és fogalmazzunk meg a folyamat specifikációját. Amennyiben a felhasznált "részeredmények", illetve azok szinonimái, a rendszer bemeneti adatfolyamai között megtalálhatók, készen vagyunk, hiszen a bemeneti és kimeneti adatfolyamokat folyamatokkal kötöttük

össze. Ha ilyen részeredmények még nincsenek, akkor a továbbiakban ezek is előállítandó adatfolyamnak tekintendők és az eljárást mindaddig ismételnünk kell, amíg minden kimeneti adatfolyamot bemeneti adatfolyamokhoz nem kapcsoltunk.

- A funkciólista alapján vegyük sorra az "adatbeviteli" tevékenységeket, és az összetartozó adatfolyamokat vezessük be egy-egy folyamatba. A funkciólistából ide illő résztevékenységgel ruházzuk fel a folyamatot és határozzuk meg a kimeneti részeredményt. Ha a kimeneti részeredmény, vagy annak szinonimája tényleges kimenet, akkor készen vagyunk, egyébként a részeredményt újabb bemenetnek tekintve, az eljárást tovább kell folytatni.

Tapasztalat szerint a kimenetek felől történő építkezés általában egyszerűbb mint a bemenetek felől közelítő módszer. Mindkét megközelítésben az új adatfolyamok felvételénél és újabb folyamathoz kapcsolásánál megfontolás tárgyát képezi, hogy az adatot az előállításakor rögtön fel is használja egy másik folyamat, vagy pedig az adat előállítása és felhasználása időben elválik. Az ilyen hosszú idejű adatfolyamokat tárolókkal kell felváltani. A tárolókat a későbbiekben felhasználhatjuk az adatfolyamok forrásául, sőt céljául is.

Az építkezés során célszerű figyelni, hogy azonos feladatot ellátó folyamatok ne szerepeljenek (álnéven) több példányban a modellben, tehát ezeket össze kell vonni.

Az adatfolyam-diagram ábra rajzolását két esetben kell abbahagyni. Amennyiben sikerül minden kimenetet előállítani a bemenetek sorozatos transzformációjával a munkánk sikeres volt. Ekkor egyenként meg kell vizsgálni a folyamatok bonyolultságát. Ha azok egy adott szintet meghaladnak, akkor a folyamat be- és kimeneti adatfolyamait, mint interfészt tekintve, egy alacsonyabb hierarchiaszinten a teljes eljárást meg kell ismételni. Alacsonyabb szintre lépve az adatszótárak alapján az adatfolyamok szükség szerint tovább bonthatók.

Előfordulhat az is, hogy bár már tele van az előttünk lévő lap folyamatokkal és adatfolyamokkal a rendszer kimeneteit mégsem sikerült a bemenetekből származtatni. Ez azt jelenti, hogy túlságosan alacsony absztrakciós szinten közelítettük meg a problémát, amelyen az adatfolyam-ábra csak egy futball pálya méretű lapra férne rá. Ilyen esetben vagy teljesen újrakezdjük a munkát, vagy a bonyolultra sikerült ábránkat felhasználva magasabb absztrakciós szintre lépünk. Az absztrakciós szint emelése úgy történik, hogy a látszólag szoros kapcsolatban álló folyamatcsoportokat, amelyekhez közös funkció is rendelhető, összevont folyamatként jelenítjük meg. A csoport tagjainak kapcsolatát pedig az összevont folyamat részletezéseként tüntetjük fel egy alacsonyabb hierarchiaszinten.

Eltérnek a vélemények arról, hogy milyen bonyolultságú adatfolyam-ábrákat érdemes egy szinten tartani és mikor kell a hierarchiaszintek növelésével az egyes szintek bonyolultságát csökkenteni. A strukturált analízis hőskorában azt mondták, hogy egy adatfolyam-ábrán lévő folyamatok száma ne nagyon haladja meg a mágikus hetes számot, mert ennél összetettebb ábrák már nehezen áttekinthetők és kezelhetők. A szintenkénti kevés folyamat viszont a hierarchikus szintek számának növekedését okozza, így ha valaki részletekbe menően akar egy teljes bemenet-kimenet láncot végignézni, akkor nagyon sok adatfolyam-ábra között kell ugrálnia. Példaként gondoljunk egy Budapest térképre, amely igen szemléletes, ha a város kontúrját, a kerületeket és a kerületekben lévő utcákat külön-külön ábrán lévő rajzokon mutatja be, mégis ha valaki az óbudai Fő térről a repülőtérre akar eljutni az előnyben részesíti a kiteríthető, lepedőre hasonlító térképet. Ezért a hőskor papír-ceruza módszereit felváltó *CASE* rendszerek alkalmazásakor fenti bűvös határ akár az ábránkénti 15-20 folyamatot is elérheti, feltéve, hogy az ábra megrajzolható anélkül, hogy az adatfolyamok metszenék egymást. Az adatfolyamok metszése mindenképpen rontja az áttekinthetőséget, ezért feltétlenül kerülni kell. Miként egyes strukturált módszertanok is megengedik, a tárolók többszöri szerepeltetése gyakran segíthet a metszések kiküszöbölésében.

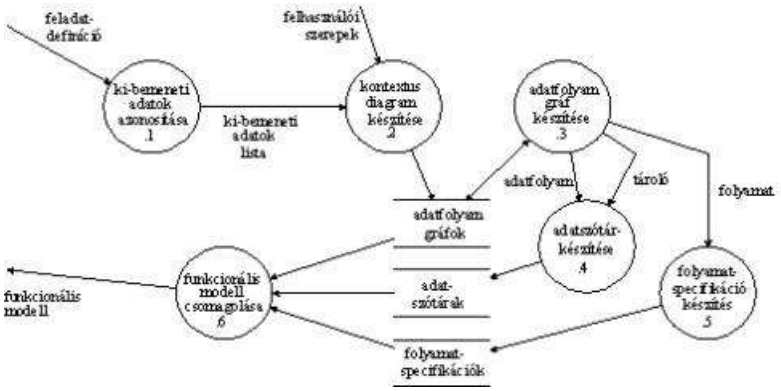
A hierarchia kialakítása során egy szinten hasonló bonyolultságú folyamatokat szerepeltessünk. A hierarchikus dekompozíciót ne erőltessük minden áron! A részfolyamatokból, a folyamatszám csökkentése miatt, ne hozzunk létre magasabb szintű folyamatot akkor, ha a részfolyamatok között nincs adatfolyam, vagy a magasabb szintű folyamathoz nem rendelhető értelmes funkció.

Az adatfolyam-ábrák felépítésénél általában elég erős a kísértés, hogy vezérlési információkat is belevigyünk a modellbe. Ennek ellent kell állnunk, hiszen ez a dinamikus modell kettőzését jelentené. Valamennyi vezérlési információt azért minden adatfolyam-ábra tartalmaz – nevezetesen egy folyamat csak akkor végezheti el az általa képviselt transzformációt, azaz akkor állíthatja elő kimeneti adatfolyamatait, ha a bemeneti adatai rendelkezésre állnak. Továbbá egy folyamat belső működését a hozzá tartozó folyamat-specifikációval adjuk meg, amely ugyancsak tartalmazhat vezérlési információkat. Ezt minimalizálhatjuk azáltal, hogy a folyamat specifikációkban a bemenetek és kimenetek kapcsolatát fogalmazzuk meg, a transzformáció leírásánál a "mi"-re

téve a hangsúlyt a szekvencia helyett. A klasszikus strukturált módszerek csak az alacsonyabb szinten tovább nem részletezett folyamatoknál engedték meg a folyamat-specifikációt, azonban az újabb irányzatokban már nincs ilyen megkötés. Ennek ellenére azt látni kell, hogy az összetett folyamatok folyamat specifikációi redundanciát jelentenek, ezért inkább az analízis megjegyzéseiként kell kezelni őket. Végül, ha feltétlenül szükséges, vezérlési információkat nem csak a folyamat specifikációkba, hanem az adatfolyam-ábrába is beépíthetünk vezérlési folyamok alkalmazásával.

Az funkcionális modell felépítését ugyancsak egy adatfolyam-ábrán foglaljuk össze (4.11. ábra).

A funkcionális modell létrehozásának ismertetett módszere arra az esetre vonatkozik, amikor a megoldást a semmiből kell létrehozni, mert a megvalósítandó rendszernek nincsenek előzményei. Ha viszont a létrehozandó programnak egy létező folyamatot kell automatizálnia, akkor ennek az elemzésére is támaszkodhatunk a funkcionális modellezés során.

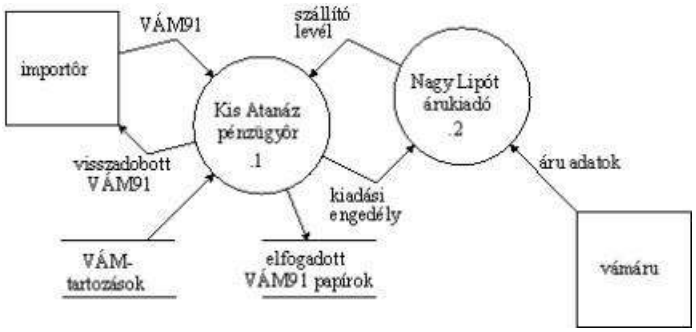


4.11. ábra

Létező folyamatok automatizálásával különösen iroda-automatizálási, folyamatirányítási és szimulációs problémák megoldásánál találkozhatunk. Ekkor általában rendelkezésre áll egy működő, de tipikusan nem számítógépes rendszer, amelyet elemezve megalkothatjuk az automatizált rendszer modelljét.

A létező rendszerben is azonosíthatunk szereplőket (transzformátorok), amelyek egy ügyviteli problémánál az adminisztrátorokat, egy folyamatirányítási feladatnál pedig az összekapcsolt gépeket jelentik. A szereplők másik része a rendszerbe adatokat táplál be illetve a rendszer termékeit hasznosítja (pl. az ügyfél). A szereplők között kapcsolatok vannak, hiszen ezeken űrlapok, információk, alkatrészek, stb. áramlanak. A szereplők a kapott dolgokat nem feltétlenül használják fel rögtön, adott esetben raktárakba (iratszekrény, alkatrészraktár) teszik, ahonnan később elővehetik.

A meglévő rendszer elemzése alapján felépíthetünk egy **fizikai adatfolyam-ábrát**, amelyben a betápláló és felhasználó szereplőket forrásként és nyelőként, a transzformátorokat folyamatként, a szereplők közötti kapcsolatot adatfolyamként, a raktárakat tárolóként kell megjelenítenünk. Például egy VÁM-hivatal fizikai **adatfolyam**-ábrája a következő módon adható meg.



4.12. ábra

Összetett rendszereknél a fizikai adatfolyam-ábra túlságosan bonyolult lenne, ezért már itt is hierarchikus adatfolyam-ábrákat alkalmazunk. A hierarchiát, ha nincs más ötletünk, a szervezeti hierarchia szerint, illetve a főbb technológiai lépéseknek megfelelően alakíthatjuk ki.

A fizikai adatfolyam-ábra alapján elindulhatunk a rendszer logikai modelljének megalkotása felé. Ehhez azonban fel kell ismerni az egyes fizikai adatfolyamatok, tárolók és transzformációk logikai tartalmát. A példában (4.12. ábra) szereplő VÁM 91-es formanyomtatvány esetében az űrlap zöld kerete, a papír minősége, stb. nem érinti a folyamatot magát, az űrlapra felírt cégnév, árumegnevezés, áruérték viszont annál inkább.

A logikai tartalom kiemelése egy adatfolyam és egy tároló esetében tehát a feldolgozási folyamat szempontjából fontos jellemzők kigyűjtését jelenti. Ezen fontos jellemzők összességét logikai adatfolyamnak, illetve tárolónak nevezzük. Az ábrán vagy komponenseire bontjuk a logikai adatfolyamot vagy pedig továbbra is egyetlen adatfolyammal szerepeltetjük. Az utóbbi esetben az adatszótárban meg kell magyarázni, hogy az összetett adatfolyam milyen komponensekből áll.

A transzformációk logikai átalakítása során arra keressük a választ, hogy az adott transzformációt végző dolgozó vagy gép pontosan mit tesz a hozzá érkező dolgokkal, majd a szereplőt a logikai tevékenységet jelentő folyamatra cseréljük. Példánkban Kis Atanáz pénzügyőrt a vámáru nyilatkozat ellenőrzés funkcióra utaló elnevezéssel kell felváltani. A logikai folyamatok részletes magyarázatát a folyamathoz rendelt szöveges folyamat-specifikációban foglaljuk össze. A funkciók azonosítása után érdemes átvizsgálni az adatfolyamok elnevezését. Célszerű jelzős főneveket használni, amelyekben a jelző arra utal, hogy milyen feldolgozáson ment át az arra haladó adat (pl. elfogadott vámáru-nyilatkozat). Az ily módon átszerkesztett ábrát **logikai adatfolyam-ábrának** nevezzük.

Eddig a rendszert mint egészet vizsgáltuk, melyet általában csak részben váltunk ki számítógépes programmal. El kell tehát dönteni, hogy mely tevékenységeket bízzuk a tervezendő programra, és melyeket kell továbbra is manuálisan, vagy más program segítségével végrehajtani. A program által realizálandó folyamatok azonosítását a **rendszerhatár kijelölésének** nevezzük. A határon kívülre kerülő folyamatok, források, nyelők és tárolók közül azok, amelyek a tervezett rendszer belsejével adatfolyamokkal kapcsolódnak, külső szereplők lesznek. A többi külső elem csak indirekt kapcsolatban áll a rendszerrel, ezért a tervezendő rendszer szempontjából lényegtelen. Megjegyezzük, hogy a határok kijelölése általában kompromisszum eredménye, melyet a költség és a várható hasznon összevetésével kell optimálisan kialakítani.

A külső szereplők azonosítása után juthatunk el a nézőpontelemzéshez és a funkciólista összeállításához. A rendszer határán belüli tartományt egyetlen folyamatnak tekintve, a külső szereplők felhasználásával megalkothatjuk a kontextusdiagramot.

A valós életben működő rendszerek általában nem optimálisak, szervezettségük pedig nem felel meg a számítástechnikai megvalósítás igényeinek. Ezért a logikai adatfolyam-ábrát át kell vizsgálnunk, racionalizálnunk és optimalizálnunk kell azt. A racionalizálás azt jelenti, hogy azokat a transzformációkat, amelyekhez nem tudtunk értelmes szerepet rendelni, töröljük. Hasonlóképpen elhagyjuk azon adatfolyamokat, melyeket egyetlen folyamat sem használ fel. Amennyiben az információ nagyon tekervényes utakon jut el egy feldolgozó folyamathoz, anélkül, hogy átalakulna, az utat le kell rövidíteni. Ha ugyanazon feladatot végző folyamatokat, illetve ugyanolyan adatokat tartalmazó tárolókat találunk, azokat össze kell vonni.

Ezen racionalizálás során kiléphetünk a megvalósítandó rendszer határain túlra is, melynek során a rendszer bevezetését előkészítő szervezeti, hatásköri és ügyviteli változtatásokat tervezhetjük meg.

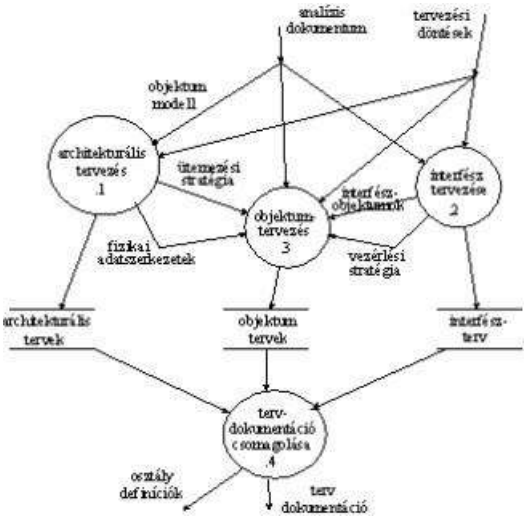
Végül a funkciólistával összevetve a kapott adatfolyam-ábrát ellenőrizzük, hogy minden elvárt funkció kielégíthető az adatfolyam-ábrával, illetve megfordítva, nincsen-e az adatfolyam-ábrának olyan szolgáltatása, amely nem szerepel a funkció listában. Az utóbbinak két oka lehet: vagy a funkció lista hiányos, vagy az adatfolyam-ábra tartalmaz felesleges, és ezért törlendő részeket.

2. 4.2. Objektumorientált tervezés

Mint láttuk, az analízis a megvalósítandó rendszer egy lehetséges modelljét alakította ki, elvonatkoztatva attól, hogy a modellben szereplő dolgokat az implementáció során milyen elemekkel képviseltetjük, illetve attól, hogy a tevékenységeket ki és milyen módon hajtja végre.

Az analízis eredményeként kapott modellek (objektum, dinamikus és funkcionális) megadják, hogy a rendszerben ki mikor kivel mit tesz a hogyan eltussolása mellett. Informatikai rendszerek kialakítása során a

rendszer szereplőit, a végrehajtott tevékenységeket szoftver- és hardver- elemekkel fogjuk realizálni, ami azt jelenti, hogy az absztrakt modellt le kell képeznünk a fizikai rendszerünk által biztosított szolgáltatásokra. Ezt a leképzési folyamatot nevezzük **tervezésnek**. A szoftvertervezés célja egy olyan formális leírás előállítása, amely a kiválasztott programozási nyelven a rendszert szimuláló programot definiálja. Ily módon a tervezés az implementáció előkészítése.



4.13. ábra

A tervezés három szintjét különböztetjük meg:

1. Az **architektúrális tervezés** során a létrehozandó programot mint egészet érintő kérdésekben döntünk.

2. A **külső interfész tervezése** a külvilággal történő kapcsolattartás módját és részleteit írja le.

3. Az **objektumtervezés** (más néven **részletes tervezés**) célja a programot felépítő elemek – objektum orientált megközelítésben az osztályok és objektumok – egyenkénti specifikálása oly módon, hogy azok akkor is együtt tudjanak működni, ha a különböző elemeket, különböző programozók implementálják.

A tervezés, mint az analízis és az implementáció közötti kapocs, nem csupán az analízis termékeire épít, hanem figyelembe veszi az implementáció lehetőségeit is. Tekintettel arra, hogy az implementáció eszköztárával (C++ nyelven) csak a 6. fejezettől kezdve ismerkedünk meg, a tervezés lépéseit itt csak összefoglaljuk és az implementáció által meghatározott részletekre a 7. fejezetben térünk vissza.

2.1. 4.2.1. Architektúrális tervezés

Az architektúrális tervezés célja a program az implementációs szempontoknak megfelelő dekomponálása, a rendelkezésre álló erőforrások megosztása, továbbá az adatszerkezetek és a vezérlés globális implementációs stratégiáinak kiválasztása. Ezen folyamat során felmerülő részkérdéseket a következőkben tekintjük át.

2.1.1. 4.2.1.1. Alrendszerek, modulok kialakítása

Az implementáció során a programozó a megvalósítandó program forrásszövegét alkönyvtárakban (*directory*) elhelyezett fájlokban írja le. Így a programot alkotó definíciókat (osztályok, objektumok) is ennek megfelelően kell csoportosítani. A definíciók fájlban elhelyezett halmazát **modulnak**, a fájlok adott alkönyvtárhoz tartozó csoportját pedig **alrendszernek** nevezzük. Egy modul, illetve alrendszer a "becsomagolt" objektumok összes szolgáltatásainak azon részével jellemezhető, melyet más modulok illetve alrendszerek felhasználhatnak.

A modulok és alrendszerek kialakítása során általában több, gyakran ellentétes igényt kell kielégíteni:

- A felbontás feleljen meg a feladat "természetes" dekompozíciójának, azaz a modellben logikailag összetartozó, tehát ugyanazon globális célt szolgáló definíciók tartozzanak egyetlen implementációs egységbe. Ez a természetes dekompozíció lehet objektum-orientált, de funkcionális is, ha azon osztályokat, objektumokat kapcsoljuk össze, amelyek egy adott globális funkció elvégzéséért együtt felelősek.

- Az alrendszerek és modulok egyrészt a programozók közötti munkamegosztásnak, másrészt pedig a program nagyobb egységekben történő későbbi újrahasznosításának a határait is kijelölik. Az újrahasznosítás támogatása miatt minimalizálni kell a modulok és alrendszerek közötti kapcsolatokat, másrészt ki kell emelni az általánosabban használható részeket. A kapcsolatok megfogalmazásánál arra kell összpontosítani, hogy mi lesz az a szolgáltatás halmaz, melyet egy más modulból is látni kívánunk, míg a többi szolgáltatást a külvilág felé el kell fedni. Az implementáció során az interfészek definícióját a modul- és az alrendszerinterfész (ún. deklarációs vagy *header*) fájlokban adjuk meg.

- Egy program általában könyvtári szolgáltatásokra, illetve korábbi programokból átvett modulokra épül. Különösen igaz ez az objektum-orientált programfejlesztésre, amely az újrafelhasználás lehetőségeit a korábbiaknál jóval hatékonyabban képes biztosítani. A könyvtárak és a kész modulok az architektúra kialakításában megváltoztathatatlan peremfeltételeket jelentenek.

2.1.2. 4.2.1.2. Többprocesszoros és multiprogramozott rendszerek igényei

A megvalósítandó program(rendszer) egy vagy több processzoron is futhat, vagy egy multiprogramozott rendszerben több párhuzamos folyamatból (*task*-ból) állhat. A programnak a processzorokra, illetve párhuzamos folyamatokra bontását szintén az alrendszerek mentén kell elvégezni. Az alrendszerek processzorokhoz rendelése során egyaránt figyelembe kell vennünk a szükségest hardver- és szoftverkomponenseket, a teljesítmény igényeket és a célrendszer által biztosított fizikai kapcsolatokat. Külön nehézséget jelent, hogy a más processzoron futó vagy a más *task*-ban realizált objektumok közötti kommunikációt lényegesen nehezebb implementálni mint a *task*-on belüli párbeszédet. Ez újabb implementációs objektumok bevezetését igényelheti, melyek most már nem a megoldandó feladatból, hanem az implementációs környezetből adódnak.

2.1.3. 4.2.1.3. Vezérlési és ütemezési szerkezet kialakítása

A vezérlést a dinamikus modell határozza meg. Az analízis modellek vizsgálata során fel kell ismernünk az ún. aktív objektumokat, amelyek között lényegi párhuzamosság van. Aktív objektumnak azokat az objektumokat nevezzük, melyek anélkül is képesek belső állapotukat megváltoztatni és képesek üzenetet küldeni, hogy más objektumtól üzenetet kaptak volna. Az aktív objektum ellentéte a passzív objektum, amely csak annak hatására vált állapotot és üzen más objektumoknak, ha őhozzá egy üzenet érkezik. A programban minden egyes aktív objektum egy-egy potenciális vezérlési szálat, azaz ok-okozati összefüggésben álló üzenetsorozatot, képvisel.

A kommunikációs diagramok első verziójának felállítása során általában nem vesződünk az üzenetek közötti ok-okozati összefüggések feltárásával. Feltételezzük, hogy minden objektum aktív, amely meghatározott sorrendben üzeneteket kap és üzenetekkel bombázza társait. A diagramok finomítása során vezetjük be az ok-okozati összefüggéseket, melyek a párbeszédet véges interakciós sorozatokra bontják és az alárendelt objektumokat lényegében passzivitásra kényszerítik. Az analízis során ok-okozati összefüggések feltárását a fogalmi modell alapján végezzük el. A tervezés alatt érdemes ezt a folyamatot még egyszer átgondolni, és a véges interakciós sorozatok valamint az ezzel összefüggő passzív objektumok gondolatát addig erőltetni, amíg lehetőleg egyetlen aktív objektum lesz a rendszerben. Ekkor az összes többi objektum kommunikációja eredendően ebből az aktív objektumból kiinduló interakciós sorozatokra esik szét. Az aktív objektumok számát még azon az áron is érdemes csökkenteni, hogy ezzel olyan üzeneteket kell mesterségesen beépíteni a modellbe, amelyek nem következnek a megoldandó feladatból. Például, a sakkjátszmában a játékosok aktív objektumok, melyek önhatalmúlag bármikor ránézhetnek a táblára és a bábukhoz nyúlhatnak (üzeneteket küldhetnek). A játékosok párhuzamosságát azonban megszüntethetjük egy aktiváló "lépj" üzenettel, amely feljogosítja a játékost a pillanatnyi állás áttekintésére és a lépésének megtételére, de amíg ilyen üzenetet nem kap addig tétlen marad. Ilyen "lépj" üzenet nincs a valódi sakkjátékban, felhasználása, a párhuzamosság kiküszöbölésére, viszont célszerűnek látszik.

Összefoglalva elmondhatjuk, hogy az ok-okozati összefüggések bevezetésével és aktiváló üzenetek felvételével az analízis során még gyakran aktívnak tekintett objektumok jelentős részét is passzívvá tehetjük.

Az aktív objektumok számát azért célszerű csökkenteni, mert mint láttuk, ezek párhuzamos vezérlési szálakat jelentenek. A programozási nyelvek többsége által is képviselt "egyetlen processzor – egyetlen vezérlési szál" alapelvű implementációs eszközök esetén a több vezérlési szál létrehozása nehézségeket okozhat.

Amennyiben a programot kevesebb processzoron futtatjuk, mint a vezérlési szálak száma, a rendelkezésre álló processzorok idejét meg kell osztani az egyes vezérlési szálak között. Nem lényegi párhuzamosságról akkor beszélünk, ha az egyes vezérlési szálak közül mindig csak egy aktív. A nem lényegi párhuzamosságot a dinamikus modell módosításával megszüntethetjük. Lényegi párhuzamosság esetén viszont egy ütemezőnek kell

a processzoridőt szétosztania. Az ütemező lehet programon kívüli eszköz, mint például a multiprogramozott operációs rendszerek szolgáltatásai, vagy programon belüli ütemező objektum. Az előző megoldás a párhuzamos *task*-ok közötti kommunikációhoz, az utóbbi a belső ütemezés megvalósításához új, implementációs objektumok létrehozását, vagy könyvtárból történő átvételét igényli.

Az ütemezés kialakításakor alapvetően két stratégia közül választhatunk:

- **Nem preemptív** stratégia alkalmazásakor az egyes vezérlési szálakra, azaz lényegében a vezérlési szálakat indító aktív objektumokra, bízzuk, hogy lemondjanak a processzor használatáról.
- **Preemptív** ütemező viszont az engedélyezett időszelet lejártakor is erőnek erejével elragadhatja a processzorhasználat jogát a vezérlési szálat kézben tartó aktív objektumtól.

A két stratégia eltérő implementációs objektumokat tételez fel és a közös erőforrások megosztását is más módon hajtja végre. Az erőforrások egymást kölcsönösen kizáró használatát nem preemptív ütemezők esetén a processzor használatról történő lemondás idejének kellő megválasztásával biztosíthatjuk az egyprocesszoros rendszerekben. Preemptív ütemezők esetén viszont az erőforrásokat szemaforokkal kell védeni. Erre a kérdéskörre a 7.6. fejezetben még visszatérünk.

2.1.4. 4.2.1.4. Fizikai adatszerkezetek implementációja

A megvalósítandó objektum belső állapotát az attribútumainak pillanatnyi értéke határozza meg. Ezeket az adatokat az objektum élete során tárolni kell, vagy a számítógép operatív memóriájában, vagy háttértáron (diszken).

A háttértáron azon objektumok adatait kell tárolni, melyek élettartama meghaladja a program futási idejét (**perzisztens objektumok**), vagy méretük és számuk nem teszi lehetővé, hogy az operatív memóriában legyenek. Az egyedi objektumok azonosítása az operatív memóriában tárolt objektumok esetében az objektumváltozó nevével vagy a címével történhet. A háttértáron tárolt objektumoknál viszont meg kell oldani az objektumok azonosítását, ami általában úgy történik, hogy az objektumok attribútumait kiegészítjük egy azonosítóval (kulccsal), amely alapján a fájlban keresve az objektumot fellelhetjük.

Ha a sebességigények megkövetelik, az objektumok keresését például indexelési technikával fel kell gyorsítani, illetve a leggyakrabban használt objektumok képmását az operatív memóriában is benn kell tartani. Olyan fájlok esetén, amelyeket több *task* is kezelhet, az objektumok belső állapotát erőforrásnak kell tekinteni és kölcsönös kizárási technikákkal védeni. Ezen igényeket az adatbázis kezelő rendszerek már régen megoldották, ezért ha a programnak nagyszámú perzisztens objektummal kell dolgoznia, akkor felmerül egy adatbázis kezelő rendszer alkalmazásának lehetősége is.

Napjainkban a versengő adatbázis technológiák közül a relációs és az objektum-orientált adatbázisok [Elsm89] jelentik a legkézenfekvőbb választási lehetőséget. A relációs adatbázis kezelők az adatokat táblákban tárolják, melyeket relációk kapcsolnak össze. Az objektum-orientált modellezés legfontosabb elemei, mint osztály, objektum, attribútum, asszociáció, öröklés, stb. megfeleltethető a relációs adatbázisok eszköztárának, de ez mindenképpen egy járulékos transzformációt igényel, amely növeli a fogalmi modell és az implementáció közötti távolságot. Ráadásul az egységbe zárás, azaz a táblázatok és a rajtuk végrehajtandó műveletek összekapcsolása ezekben a rendszerekben csak a programozó stílusán múlik, a relációs adatbázis technológia ezt közvetlenül nem támogatja. A relációs adatbázis nem hatékony olyan esetekben, amikor nagyszámú osztály található a modellben, de egy-egy osztálynak csak néhány példánya (objektuma) létezik. Azonban előnyként említhető meg a relációs adatbázis kezelők kiforrottsága, az általuk biztosított hordozhatóság, referenciális integritás védelem, relációk karbantartása, tranzakció-kezelés (*commit, rollback*), adat- és hozzáférés-védelem, szabványos lekérdező nyelvek (SQL), melyek igen magas szintű szolgáltatásokat biztosítanak és hatékonyan kezelnek nagy mennyiségű, de viszonylag kevés számú osztályhoz tartozó objektumot. Az objektum-orientált adatbázisok az objektum-orientált elvek közvetlenebb alkalmazását jelentik. Azonban ezek még nem általánosan elterjedtek és széles körben elfogadott magas szintű lekérdező nyelvük sem létezik.

2.1.5. 4.2.1.5. Határállapotok megvalósítása

Egy program a normál működésén kívül alapvetően a következő három ún. határállapotban lehet:

- Inicializáció, amely az objektumok normál működésre történő felkészítését és a vezérlési lánc(ok) elindítását jelenti.

- Leállítás, amely az program befejezésekor elvégzendő tevékenységek összessége.

- Katasztrofális hibaállapot, amelybe valamely objektum működése során jelentkező, azonban az objektum által nem kezelhető helyzet miatt kerülhet a rendszer.

Ezen felelősségeket a tervezés során általában egyetlen implementációs objektumhoz, az ún. **applikációs objektumhoz** rendeljük.

Az analízis során a hibák lehetőségére általában nem gondolunk, hiszen azok többsége a tervezési és az implementációs megoldásokból adódik. Ezért a tervezés során számba kell venni az egyes hibákat és helyreállításuk módját. A hibajelzéseket úgy kell kialakítani, hogy azok a felhasználó saját fogalmi rendszeréhez kapcsolódjanak, nem pedig a belső implementációs megoldásokhoz.

2.2. 4.2.1. Külső interfész tervezése

A programok a működésük során adatokat kell kapnak a külvilágból, melyeket feldolgozva az eredményeket visszajuttatják oda. Az analízis során a külső szereplőket elsősorban a dinamikus és funkcionális modell tartalmazza (külső esemény, adatfolyam forrás és nyelő). A külső szereplőkből eredő adatfolyamok a rendszer bemeneti adatai, az ide távozó adatfolyamok a rendszer eredményei. A vezérlési elemekről a kommunikációs diagramból kaphatunk képet, ahol a külső szereplőket objektumokként szerepeltetjük. Végül, az időben egyszerre érkező vagy távozó adatfolyamok között kapcsolatok állhatnak fenn, amelyeket az interfészek működésére vetített objektum modellekkel tekinthetünk át.

Az adatok forrása lehet fájl, a felhasználó által kezelt beviteli eszközök (klaviatúra, egér, stb.) vagy lokális hálózaton, soros vonalon vagy egyéb módon csatolt külső eszköz. Az adat felhasználója hasonlóképpen lehet fájl, a felhasználó által látható képernyő, nyomtató vagy akár egyéb csatolt eszköz is.

Az adatforrások lehetnek passzívak, amelyeket a programnak olvasnia kell, hogy tudomást szerezzen a bemeneti adatokról, vagy lehetnek aktívak, melyek a programhoz képest aszinkron módon közlik a bemeneti adatokat, illetve szolgáltatási igényeiket.

Az implementáció során a külső szereplőkhöz egy vagy több objektumot rendelünk, amelyek metódusai elvégzik a tulajdonképpeni ki- és bevitelt, és a program többi része számára a külvilágot a program látókörén belülre képezik le. Ez egy passzív külső eszköz esetében nem is jelent semmi nehézséget, hiszen ezek akkor olvashatók illetve írhatók, amikor a program belső objektumainak adatokra van szüksége illetve külső eszköznek szánt adat előállt. Az aktív eszközök azonban aktív objektumokként jelennek meg, melyek a program vezérlési és ütemezési szerkezetével szemben támasztanak elvárásokat. Az aktív külső szereplőket két alapvetően különböző stratégiával kezelhetjük:

- **Programvezérelt interfészekben** a külső aktív szereplőt passzivitásra kényszerítjük. A vezérlés a programon belül marad, és amikor külső adatra van szükség, akkor egy adatkérő üzenet után a program várakozik a külső adat megérkezésére. Azt, hogy éppen milyen adatot kér a rendszer, a végrehajtott utasítás, azaz az utasítás-számláló és a szubrutin hívási hierarchia (veremtartalom) határozza meg. Ez egy régi és konvencionális módszer, melyet könnyű implementálni a szokásos programozási nyelvek és könyvtárak segítségével. Ez a megoldás azonban kudarcot vall abban az esetben, ha több külső forrásból érkezhetnek üzenetek (ez a helyzet már akkor is, ha a felhasználó billentyűzeten és egérrel egyaránt bevihet adatokat), hiszen mialatt a program az egyik forrásra várakozik a másik jelzéseit nem képes fogadni.

- **Eseményvezérelt interfészekben** a vezérlést egy aktív, ún. elosztó (*dispatcher*) objektumhoz rendeljük, amely folyamatosan figyeli az összes külső forrást és az ott keletkező fizikai eseményeket. A fizikai eseményeket, a belső állapotának függvényében, logikai eseményekké alakítja, és aktivizálja azon objektum azon metódusát, melyet az elosztó inicializálása során az adott a logikai eseményhez rendeltünk. Az eseményekhez kapcsolt metódusokat szokás még **eseménykezelőknek** (*event handler*) és **triggereknek** is nevezni. A fizikai és logikai események szétválasztása azért fontos, mert a körülmények függvényében ugyanaz a fizikai beavatkozás igen különböző címzettnek szólhat és reakciót igényelhet. Egy ablakos felhasználói interfészben például az egér gomb lenyomása okozhatja egy menü legördítését, ablakon belül egy pont kijelölését (ami persze aszerint mást jelent, hogy konkrétan melyik ablak felett történt az esemény) vagy a program megállítását.

Az eseményvezérelt program nem vár a külső eseményre, hanem reagál rá. Ha a reakció befejeződött, a trigger visszatér az eseménykezelőhöz, amely folytatja az külső források figyelését. Ez azt jelenti, hogy minden külső

esemény észlelése a programnak ugyanazon a pontján történik, tehát az állapotot nem tarthatjuk nyilván az utasításszámlálóval és a verem tartalmával, hanem az állapothoz explicit változókat kell rendelnünk. Egy eseményvezérelt rendszerben a felhasználó minden pillanatban beavatkozási lehetőségek tömegéből választhat. A program a változókban tárolt belső állapota alapján dönti el, hogy a beavatkozások közül, egy adott ponton, melyeket fogadja, és hogyan reagáljon rá.

Tehát az eseményvezérelt rendszerek kialakítása során létre kell hozni egy aktív implementációs objektumot, az elosztót (ablakozott rendszerekben gyakran használják még az alkalmazás vagy applikációs ablak nevet is). Az aktív külső eszközöket megtestesítő objektumok passzívak, melyeket az elosztó üzenettel aktivizál, ha rájuk vonatkozó esemény keletkezik.

A külvilágba távozó és a külvilágból érkező adatok formátuma általában az eszköz által korlátozott és jól definiált (például a lokális hálózatról érkező adatcsomagok megfelelnek a hálózati protokollnak). Fontos kivételt jelentenek azonban a felhasználói interfészek, melyek kialakításában a rendszer tervezőjének igen nagy szabadsága van. Ez azt jelenti, hogy a felhasználói felület kialakítása és a részletek meghatározása a tervezés külön fejezetét alkotja, amely során az analitikus modelljeinket lényegében egy újabbal, a **megjelenési modellel** kell kiegészíteni. A megjelenési modell a következő elemeket tartalmazza:

- Képernyő séma képek megjelenése, beavatkozó szervek (menük, gombok, görgető sor, stb.) formája és jelentése.
- A sémákban egyszerre látható, illetve bevihető adatok és a közöttük fennálló relációs viszonyok leírása.
- A képernyősémák közötti navigációs lehetőségek áttekintése, melyre például állapottér-modelleket használhatunk.
- A sémákon belüli állapottér-modell, amely azt írja le, hogy a program hogyan értelmezi felhasználói beavatkozások sorozatát.

A megjelenési modellben az analízis dinamikus modelljéhez képest új események jelenhetnek meg, amelyek a program kezeléséből, nem pedig a feladatból adódnak. Az új eseményekre főleg amiatt van szükség, hogy a külső eseményeket le kell képeznünk a rendelkezésre álló fizikai eszközök (billentyűzet, egér) szolgáltatásaira. Tegyük fel például az analízis során felvettük azt az eseményt, hogy "*a felhasználó egy téglalapot definiál*". Ez a logikai esemény egy grafikus felhasználói felületen, gumivonal technika alkalmazását feltételezve a következő fizikai eseménysorra képezhető le:

1. az egér gomb lenyomás (az egyik sarokpont kijelölése),
2. az egér mozgatások sorozata lenyomott gombbal (a másik sarokpont változatok bemutatása gumivonallal),
3. az egér gomb elengedése.

A fizikai események megjelenése miatt a felhasználói interfész specifikálása után a dinamikus modell jelentős finomításra szorul. A felhasználói felületet különböző metaforák (asztalon szétdobált könyvek, műszerek kezelőszervei) szerint megvalósító implementációs objektumok (ablak, menü, ikon, görgető sor, stb.) az objektum modell kiegészítését is igényelhetik.

2.3. 4.2.2. Objektumtervezés

Az objektumtervezés az analízis modellek, továbbá az architektúrális és interfésztervezés eredményét finomítja tovább. Az architektúrális és interfésztervezés lépései során a fogalmi teret leíró analitikus objektumokon és modelljeiken kívül, az implementációs tér objektumai is megjelennek. Az implementációs objektumokkal kiegészített analízis modell az objektum tervezés alapja.

Az objektumtervezés során az analízis modelljeiből az osztályok és objektumok teljes – tehát mind a belső adatszerkezetet, mind a külső viselkedést leíró – definícióit meg kell alkotnunk. Ennek során össze kell kapcsolnunk a különböző szempontok szerint készített objektum, dinamikus és funkcionális modelleket. Objektum orientált tervezésnél a három nézőpont közül az objektum modellből indulunk ki, és azt a többi modell által képviselt információval az osztályokat ruházzuk fel. Úgy is fogalmazhatunk, hogy az objektum orientált tervezésnél a "struktúráló elem" az objektum modell.

Az objektumtervezés főbb lépései az alábbiak:

1. **Az objektum, a dinamikus és a funkcionális modellek kombinációja** során létrehozzuk az osztályok deklarációját, az attribútumok és metódusok egy részének deklarációjával együtt. Itt a legnagyobb problémát általában a funkcionális modell beépítése jelenti, hiszen az objektum és dinamikus modellek szoros kapcsolatban állnak egymással, ami azonban az objektum és funkcionális modellpárról nem mondható el.

2. **Az üzenet algoritmusok és implementációs adatstruktúrák kiválasztása** során tervezési szempontok alapján pontosíthatunk egyes adatstruktúrákat és részben ennek függvényében, részben az analitikus modellek alapján meghatározzuk az egyes metódusok algoritmusait, azaz a tagfüggvények törzsét.

3. **Az asszociációk tervezése** nevű fázisban az objektumok közötti hivatkozások megvalósításáról döntünk. A kapcsolatok leírásához tipikusan mutatókat alkalmazunk.

4. **A láthatóság biztosítása.** Az objektumok a program végrehajtása során egymásnak üzeneteket küldhetnek, felhasználhatják egymást értékadásban, illetve függvényargumentumként, stb. Ez pontosabban azt jelenti, hogy egy adott objektum metódusaiban más objektumokra (változókra) hivatkozunk, amely igényli az implementációhoz kijelölt programozási nyelv láthatósági szabályainak a betartását.

5. **Nem objektum-orientált környezethez, illetve nyelvekhez történő illesztés**, melynek során meg kell oldani a hagyományos függvényekből álló rendszernek és az objektumok közötti üzenetváltással működő programunknak az összekapcsolását.

6. **Az ütemezési szerkezet kialakítása.** Amennyiben az egy processzoron futtatandó program modelljében több aktív objektum található, azok párhuzamosságát fel kell oldani. Általános esetben ide tartozik a többprocesszoros illetve az elosztott rendszerekben az objektumok processzorokhoz való rendelése is.

7. **Optimalizálás (módosíthatóságra, futási időre, kódméretre).** A tervezett rendszer közvetlen programkóddá transzformálása gyakran nem ad kielégítő megoldást. Ezért szükséges lehet a nem teljesült szempontok szerint a terv finomítása illetve optimalizálása.

Tekintve, hogy ezek a lépések már erősen építenek az implementációs környezetre, részletes tárgyalásukra egy kiválasztott objektum-orientált nyelv (a C++) megismerése után, a 7. fejezetben térünk vissza.

Az objektumtervezés kimenete az osztálydefiníciókat tartalmazó program váza. Amennyiben nem kívánunk az implementációs nyelv elemeivel bajlódni, a tervezést az osztályokat leíró kérdőívek pontos kitöltésével is befejezhetjük. Ezek a kérdőívek, a kezdeti specifikációs táblázat jelentősen átdolgozott és kiegészített változatai, melyek általános szerkezete:

Név:	az osztály megnevezése.
Felelősségek:	az objektumosztály által viselt felelősségek szöveges leírása.
Példányok:	0/1/n. 0/1/n. Ha absztrakt osztályt definiálunk, akkor lehetséges, hogy az osztály nem példányosodik.
Alaposztályok:	az öröklési hierarchiában az adott osztály fölött álló osztályok listája.
Paraméterek:	parametrikus vagy generikus osztályoknál a paraméterek típusa. Egyes objektum orientált programnyelvek (a C++ is) megengedik a változók típusával paraméterezett objektumok minta utáni generálását.
Szolgáltatások:	az osztály metódusainak deklarációs listája.
Komponensek:	az osztállyal komponensrelációban álló osztályok és objektumok listája.

Változók:	az osztály attribútumainak megnevezése és típusa.
Relációk:	asszociáció révén kapcsolt osztályok listája.
Konkurencia:	Az objektum dinamikus működésének jellemzője, az aktivitása.
Perzisztencia:	statikus/dinamikus. Statikusnak nevezzük azokat az objektumokat, amelyek élettartama túlnyúlik egyetlen programfuttatáson.
Teljesítmény:	az objektumra vonatkozó tár és időzítési korlátok (elérési idők, objektumpéldányok száma alapján becsült tárigény).

5. fejezet - 5. Objektumok valósidejű rendszerekben

Létezik a számítógépes rendszereknek egy olyan osztálya, amelyet általában megkülönböztetett figyelemmel, és gyakran a szokásostól eltérő módon kezelnek a rendszertervezők. Ennek oka, hogy ezekkel a rendszerekkel szemben speciális követelmények állnak fenn, amelyek nem mindig kezelhetők a szokásos szoftverfejlesztő módszerekkel. Ez a rendszerosztály a valósidejű (real-time) rendszerek osztálya.

Annak ellenére, hogy a valósidejű rendszer fogalom egyike a számítástechnika klasszikus fogalmainak, mind a mai napig újabb és újabb definíciói látnak napvilágot. Nem elsősorban azért, mert nincs közmegegyezés a szakmai körökben a valósidejű rendszerek leglényegesebb jellemzőiről, hanem inkább a témakör megközelítésének sokféle lehetősége miatt. Mindemellett ezekkel a rendszerekkel kapcsolatosan számos félreértés – vagy talán inkább féligazság – is van a köztudatban [Sta88], ami még indokoltabbá teszi, hogy a fejezet első részében saját szemszögünkből röviden összefoglaljuk a valósidejű rendszerek fontosabb jellemzőit.

A továbbiakban először áttekintjük a valósidejű rendszerekkel kapcsolatos alapfogalmakat, és a rendszerek jellemző tulajdonságait. Ezt követően az időkövetelmények néhány típusát mutatjuk be. Harmadsorban a valósidejű rendszerek fejlesztése során felmerülő speciális problémákkal foglalkozunk. Végül az érdeklődőknek néhány további olvasmányt ajánlunk, amelyek valósidejű rendszerek fejlesztésére ajánlott módszertanokat ajánlanak.

1. 5.1. A valósidejű rendszerek jellemzői

A *real-time* fordításaként a magyar szóhasználatban elterjedt *valósidejű* kifejezés jelentéséhez az elnevezés eredetének megvilágításával juthatunk közelebb.

A kifejezés akkor alakult ki, amikor felvetődött, hogy a számítógépek bármilyen *célgép* (automata) szerepét képesek betölteni. Így arra is alkalmasak, hogy emberi közreműködés nélkül jussanak a környezetükből származó információhoz (érzékelők segítségével), további viselkedésüket ehhez az információhoz igazítsák, esetleg közvetlenül be is avatkozzanak a környezetükben lejátszódó folyamatokba (megfelelő beavatkozó szervek segítségével). Míg a korábban kialakult adatfeldolgozó rendszerek tervezése általában az adatszerkezetek vagy az adattransz-formációk (funkciók) megragadásával kezdődik, addig az ilyen célrendszerek tervezői leggyakrabban a rendszert érő hatások (külső események) és az erre adandó válaszok – azaz a rendszer *viselkedése* – elemzéséből indulnak ki. A viselkedés megadásakor általában nem elegendő azt előírni, hogy a rendszernek milyen külső események hatására milyen válaszokat kell előállítania, hanem a számítógépes rendszer *válaszidejére* is megkötéseket kell tenni.

Tegyük fel például, hogy egy vegyi üzemben számítógépes irányítás működik. A számítógép egyik feladata az, hogy figyelje egy tartályban uralkodó nyomást, és ha az meghalad egy határértéket, nyisson ki egy biztonsági szelepet a robbanás megakadályozására. Természetesen nem mindegy, hogy erre a szelepnyitásra mennyivel azután kerül sor, hogy a nyomás a határérték fölé emelkedett. A technológiát és a berendezéseket ismerő szakemberek meg tudják mondani, hogy az adott berendezésen mekkora az a késleltetés, ami még nem megy a biztonság rovására. A számítógépes rendszernek úgy kell működnie, hogy válaszideje ezen a megengedett késleltetési időn belül maradjon. Ha nem sikerül adott időn belül a reagálni, az ugyanolyan hiba, mint egy téves számítási eredmény előállítása, sőt következményeit tekintve még súlyosabb is lehet. (Az olvasóban bizonyára felmerülő jogos aggodalmak eloszlatására el kell mondanunk, hogy a gyakorlatban az emberélet szempontjából kritikus rendszereket többszörös védelemmel látják el, és a számítógépes irányítás mellett közvetlen védelmeket – a fenti esetben például mechanikus vészleeresztő szelepet – is alkalmaznak. Ezeknek a védelmeknek a működése azonban már nem tartozik a normál számítógépes üzemvitel körébe.)

A számítógépek ilyenfajta alkalmazása szükségessé tette, hogy a számítógépes rendszer és a benne futó programok *időbeli viselkedését* is vizsgáljuk, illetve specifikáljuk, mégpedig a *környezetben érvényes*, a rendszer szempontjából *külső, valós időskálán*. Ez alapvető különbség a korábbi, olyan rendszerekkel szemben, ahol az idő a feladatok megfogalmazásakor fel sem merül, vagy csak szimulált időként merül fel (amikor például időfüggvényeket számíttatunk ki egy programmal).

1.1. 5.1.1. Meghatározás, osztályozás

Az elnevezés eredetének megvilágítása után pontosítsuk a *válaszidő* és a *valósidejű rendszer* fogalmát, és ismerkedjünk meg a szakirodalomban gyakran használt, *szigorú (hard)*, illetve *laza (soft)* értelemben vett valósidejűséggel!

Egy rendszer válaszidején (response time) azt az időtartamot értjük, ami egy környezeti esemény bekövetkezése és a rendszer erre adott válasza között eltelik.

Különböző környezeti eseményekre adott válaszokhoz – azaz a rendszer különböző funkcióihoz – különböző válaszidők tartozhatnak. A rendszer tehát általában nem egyetlen válaszidővel jellemezhető.

A fenti definíció feltételezi, hogy a környezeti esemény bekövetkezésének pillanata, valamint a rendszer válaszának pillanata egyértelműen meghatározható. A valóságban ez nem mindig egyszerű. A tervezés során ugyanis részletesen meg kell vizsgálni, hogyan is szerez tudomást a rendszer az eseményről, és a válaszai hogyan fejtik ki a kívánt hatást. Így a pillanatszerűnek látszó események "a nagyító alatt" már folyamatokká, hatásláncokká válnak, és egyáltalán nem mindegy, hogy ezen belül melyik időpillanatot tekintjük az esemény, illetve a válasz pillanatának. A tervező felelőssége általában a rendszer határvonaláig terjed. A rendszer és a környezet együttműködése örök vitatéma a megrendelő és a rendszerfejlesztő között. Kiélezett esetekben – főként ha a környezetet is alakítani kell a rendszerrel való együttműködés céljából – igen fontos, hogy a rendszer és a környezet határfelületének pontos megadása, valamint a követelmények precíz, a határfelületen zajló jelváltásokra lebontott megfogalmazása már a fejlesztés korai szakaszában megtörténjen.

A legegyszerűbb esetben a külső esemény egyetlen logikai értékű jel adott irányú változása, de sokkal gyakoribb, hogy az esemény csak több jel értékéből képzett kifejezéssel, esetleg jelsorozatokkal jellemezhető. A rendszer válasza ugyancsak lehet egyetlen jel beállítása, vagy egy jelsorozat kiadása, amelyen belül időzítési követelményt fogalmazhatunk meg a válasz megkezdésére, befejezésére, sőt az egyes fázisok lezajlására is. Bonyolultabb esetekben az eredetileg egyetlen válaszidő-követelményből a részletek kidolgozása során egy szövevényes időzítésláncolat is kialakulhat.

A valósidejű rendszerek definícióját magának a rendszernek a tulajdonságai alapján nem tudjuk megadni. Ha ugyanis egy létező rendszerről el kell döntenünk, hogy valósidejű-e vagy nem, hiába végzünk el rajta bármilyen vizsgálatot, fejtjük meg a programját, nem tudjuk megadni a választ, ha nem ismerjük a rendszer feladatspecifikációját. Bizonyos tulajdonságok alapján persze gyanakodhatunk. Például, ha időmérő eszközöket, időkezelő eljárásokat, vagy időfüggő algoritmusokat találunk. A feladat ismerete nélkül azonban nem dönthető el, hogy ezek használata valamilyen időbeli viselkedésre vonatkozó követelmény teljesítéséhez, vagy más szempontok – mondjuk a felhasználók géphasználatának igazságos számlázása, esetleg az egyenlő esélyeket biztosító processzoridő-megosztás – miatt szükséges. A definíciót tehát nem a rendszer tulajdonságai, hanem a vele szemben támasztott követelmények alapján kell megadni.

Valósidejűnek azokat a rendszereket nevezzük, amelyek specifikációjában valamilyen időbeli viselkedésre vonatkozó előírás szerepel a külső, valós időskálához kötötten.

A szakirodalomban kialakult egy további osztályozás, miszerint a valósidejű rendszereken belül is érdemes megkülönböztetni a **szigorúan valósidejű (hard real-time)** rendszereket. Eszerint *szigorúan valósidejűnek azokat a rendszereket nevezzük, amelyek specifikációja egyértelműen rendszerhibának tekinti valamely időkövetelmény be nem tartását*. Ez azt jelenti, hogy a szigorúan valósidejű rendszerek normál működése során fel sem merül, hogy valamilyen időkövetelmény teljesítése elmulasztható. Ezzel szemben a **lazán valósidejű (soft real-time)** rendszerek körében az ilyen mulasztást csak a rendszer működőképességének csökkeneséként fogjuk fel, az esetek bizonyos (igen csekély) hányadában elfogadjuk az időzítési követelmény elmulasztását, és erre az esetre is specifikáljuk a rendszer viselkedését.

A bevezetett fogalmakkal és a megadott definíciókkal kapcsolatosan felmerülhet néhány jogos észrevétel.

Egyrészt bizonyos, válaszidőre vonatkozó elvárások minden rendszerrel szemben fennállnak, akkor is, ha azokat a specifikáció nem említi tételesen. Például egy szövegszerkesztővel szemben elvárás, hogy elviselhető idő alatt hajtson végre bizonyos műveleteket, mielőtt még a felhasználója megunná, és kikapcsolná a számítógépet. Ebben az értelemben gyakorlatilag minden rendszer valósidejű. A szakterületen kialakult értelmezéssel összhangban azonban csak azt a rendszert tekintjük valósidejűnek, amelynek specifikációjában kifejezetten előírt *(explicit)* időkövetelmény szerepel.

Másrészt egy konkrét rendszer tervezésekor mindenképpen szembekerülünk azzal a problémával, hogy hogyan viselkedjen a rendszer olyan esetekben, amikor a működőképessége még fennáll, de észleli, hogy valamilyen

hiba történt (kivételes esetek kezelése, *exception handling*). Az ilyen esetek egyike, amikor valamilyen hiba fellépése, vagy egyszerűen csak a feladatok halmozódása miatt nem sikerül betartani egy időzítést. A legritkábban engedhető meg az a tervezői alapállás, hogy ha már úgyis hiba történt, a rendszernek joga van beszüntetni működését, és leragadni egy *"katasztrofális hiba"* állapotban. A rendszertől elvárt viselkedés általában az, hogy próbálja meg bizonyos rendezett akciókkal csökkenteni a hiba (például határidő elmulasztása) következményeit, és később automatikusan térjen vissza a normál üzemállapotba. Ezért a gyakorlatban a *szigorúan,* illetve a *lazán* vett valósidejűséget nem kezeljük lényegesen különböző módon a tervezés során. Kialakult azonban két különböző tervezői alapállás, amelyek rokoníthatók a szigorúsággal, illetve lazasággal. Ez a két alapállás a *determinisztikus modellekre* alapozott *legrosszabb esetre (worst case)* tervezés, illetve a *valószínűségi modellekre alapozott* tervezés, amelyekre hamarosan visszatérünk.

1.2. 5.1.2. Egyéb jellemző tulajdonságok

A valósidejűség problémája, mint láttuk, elsősorban a célgépként viselkedő, a környezetükkel közvetlen kapcsolatot tartó rendszerek esetén merült fel. Ezeket a rendszereket a mai szóhasználattal **beágyazott rendszereknek** *(embedded system)* nevezzük, mert a célfeladatot ellátó számítástechnikai rendszer beépül egy más, nem számítástechnikai rendszerbe. A beágyazott rendszerek más – az időbeli viselkedésüktől független – speciális tulajdonságokkal is rendelkeznek.

Megbízhatóság, biztonság, robosztusság

A beágyazott rendszerek jelentős része olyan feladatokat lát el, ahol hibás kimenet kiadása jelentős anyagi kárral, sőt emberi élet veszélyeztetésével is járhat. Az ilyen rendszerekkel szemben általában *fokozott biztonsági és megbízhatósági követelményeket* támasztanak. A fokozott biztonság érdekében a rendszer akkor sem adhat ki olyan kimenőjeleket, amelyek a környezetben veszélyes helyzetet idéznek elő, ha meghibásodás miatt nem tudja folytatni működését. A fokozott megbízhatóság elérésére pedig gyakran nem elég, ha kis meghibásodási valószínűséggel jellemezhető berendezésekből, alaposan tesztelt és többszörösen kipróbált programokból építkezünk, hanem a rendszereket a *hibatűrés* képességével is fel kell ruházni. Ez azt jelenti, hogy egy részegységben fellépő hibának a külvilág számára nem szabad kiderülnie, azaz a rendszernek a kimenetein továbbra is hibátlan működést kell mutatnia.

Ugyancsak a biztonsággal és a megbízhatósággal kapcsolatos követelmény a *robosztusság*. Egy robosztus rendszert különlegesen kedvezőtlen, szélsőséges környezeti hatások sem tehetnek tönkre. Átmenetileg leállhat, de a kedvezőtlen hatások elmúltával tovább kell működnie. A szoftver tervezője számára ez elsősorban azt jelenti, hogy a rendszer akkor sem kerülhet határozatlan *(indefinit)* állapotba, ha nemvárt, nemspecifikált adatokat, adatsorozatokat, vezérlőjeleket kap, vagy a program-végrehajtása valamilyen meghibásodás, tápkimaradás miatt megszakad, majd újraindul.

Az esetek jelentős részében a hibatűrő, robosztus működéshez még kezelői segítségre sem lehet számítani (aki például megnyomja a *RESET* gombot), hanem *automatikus feléledést (recovery)* kell megvalósítani.

Természetesen olyan rendszer nem készíthető, amelyik minden körülmények között működik. A biztonságra, megbízhatóságra, hibatűrésre és robosztusságra vonatkozó követelményeket a specifikációban rögzíteni kell.

Folyamatos, felügyeletszegény működés

A beágyazott rendszerek gyakran hónapokig, sőt évekig nem kapcsolhatók ki, nem állíthatók le. Ismét folyamatirányítási példát említve, képzeljük el egy olajfinomító irányító rendszerét. A finomító tipikus üzemvitele évi egyetlen (tervezett) karbantartási célú leállást enged meg, aminek időtartama néhány nap. Minden egyéb leállás hatalmas anyagi veszteséget okoz.

Ugyancsak gyakori, hogy a rendszerek, vagy a nagyobb rendszerek alrendszerei, hosszú időn át felügyelet nélkül működnek. Gondoljunk például a terepre kihelyezett irányító, adatgyűjtő állomásokra, vagy – mint a legszélsőségesebb példára – az űrszondákra, ahol bármilyen helyi kezelői beavatkozás lehetetlen diagnosztika, javítás, újraindítás céljából.

Az eddig felsorolt tulajdonságok miatt a valósidejű, beágyazott rendszerekben a szoftver sokkal nagyobb része foglalkozik a reményeink szerint soha be nem következő, kivételes esetek kezelésével, mint az egyéb rendszerekben. Ez még nehezebbé teszi az időkövetelmények teljesítését.

Párhuzamos működés (konkurencia)

Amikor valósidejű rendszereket tervezünk, igen gyakran időben párhuzamosan futó (konkurens), együttműködő folyamatokkal modellezzük a rendszert.

Itt jegyezzük meg, hogy ebben a fejezetben a *párhuzamos* és a *konkurens* kifejezéseket szinonimaként használjuk. A szakterületen belül azonban – elsősorban a finomabb léptékű tervezés, illetve a megvalósítás problémáit tárgyaló szakirodalomban – a két kifejezés eltérő értelmezésével is találkozhatunk. Például a *konkurens* jelző gyakran a folyamatok szintjén fennálló párhuzamosságot jelöl, ahol a megvalósítás finomabb léptékben már szekvenciálisan is történhet (multiprogramozott rendszer), míg a *párhuzamos* megnevezés finomabb léptékben is párhuzamos (többprocesszoros, valódi párhuzamos feldolgozást végző) rendszert takar.

A párhuzamosság már az analízis során felállított modellben is megjelenhet, de a tervezés későbbi szakaszaiban is kiderülhet, hogy bizonyos – durvább léptékben szekvenciális – műveletsorozatok finomításakor célszerű párhuzamosítást alkalmazni.

Az *analízis során bevezetett párhuzamosság* oka a feladatban rejlik, abban, hogy a rendszer több, egymástól függetlenül működő, aktív szereplővel tart kapcsolatot.

A valósidejű rendszerek feladatainak megfogalmazása leggyakrabban a rendszer viselkedésének leírásával történik. A leírás azt tartalmazza, hogy egy-egy funkció végrehajtása közben milyen üzenetváltások történnek a környezeti szereplők és a rendszer között, és ezek hatására ki milyen műveletet végez. Az aktív környezeti szereplők általában egymástól függetlenül, időbeli korlátozások nélkül kezdeményezhetnek üzenetváltásokat. A rendszer ennek következtében egyidejűleg több környezeti szereplővel folytathat párbeszédet úgy, hogy a különböző szereplőkkel váltott üzenetek egymáshoz viszonyított sorrendje előre nem határozható meg. A modellezés során ezért egy-egy környezeti szereplő kezelése önálló vezérlési szálként (folyamatként), a rendszer pedig aszinkron párhuzamos folyamatokból álló rendszerként ragadható meg.

Gondoljunk például a korábban említett vegyipari irányítási feladat egy olyan változatára, ahol a rendszer egyszerre több tartály felügyeletét látja el, és a kezelőtől is bármikor különböző parancsokat kaphat. Beszállító jármű érkezésekor például el kell indítani egy töltési folyamatot a járműből valamelyik tartályba, amely több fázisból áll (csatlakoztatás, szivattyú indítása, kiürülés figyelése, leállítás stb.). Eközben lehetséges, hogy kiszállító jármű érkezik, és egy másik tartályból el kell indítani adott mennyiségű anyag átszivattyúzását (ürítés) a kiszállító járműbe. A be- és kiszállító járművek egymástól függetlenül, tetszőleges időpontban érkezhetnek, ráadásul a kezelő ezektől függetlenül, bármikor új kijelzést kérhet, vagy tetszőleges más parancsot is kiadhat.

A fenti rendszer egy funkcióját (pl. töltés járműből tartályba) különösebb nehézség nélkül le tudjuk írni egy egyetlen szálon futó, üzenetekből, illetve műveletekből álló sorozattal. Ha azonban a teljes rendszer működését próbálnánk adott sorrendben végrehajtandó műveletekből összerakni (egyetlen szálra felfűzni), nehézségeink támadnának, hiszen nem tudjuk előre, hogy a járművek milyen ütemben érkeznek, és ezekhez képest a kezelőnek mikor jut eszébe új kijelzést, naplózást, vagy egyéb rendszerszolgáltatást kérni.

A *tervezés és az implementáció során bevezetett párhuzamosság* célja/oka lehet

– az időkövetelmények betartásához szükséges számítási teljesítmény elérése (a processzor és más erőforrások kihasználásának javítása átlapolódó használati móddal, illetve a számítási teljesítmény növelése több processzor vagy számítógép alkalmazásával),

– a rendszer térbeli kiterjedése miatt – elsősorban a belső, intenzív adatforgalom korlátozása érdekében – szükségessé váló, elosztott, többprocesszoros, vagy többszámítógépes hardverarchitektúra.

A konkurens rendszerek fejlesztése sok szempontból nehezebb, mint a szekvenciális rendszereké. Ennek legfontosabb okai:

– A programozási nyelvek többsége csak a szekvenciális, egy processzoron futó programok fejlesztését támogatja hatékonyan. Párhuzamos, elosztott rendszerek fejlesztéséhez általában a nyelvi szint feletti eszközök szükségesek.

– Sokkal nehezebb a tesztelés és nyomkövetés, hiszen az aszinkron folyamatok következtében a rendszeren belül a programvégrehajtás elveszti determinisztikus jellegét és a tesztesetek nehezen reprodukálhatók.

– A konkurenciából további, megoldandó problémák származnak, mint például a közösen használt erőforrások ütemezése, a folyamatok közötti biztonságos kommunikáció megvalósítása, a holtponthelyzetek kiküszöbölése.

– Nehezebb – esetenként lehetetlen – az egyes műveletek végrehajtási idejének, illetve azok felső korlátjának kiszámítása.

A fenti problémák miatt a tervezés és az implementáció fázisában a tervezők törekvése inkább a magasabb szinten bevezetett párhuzamosságok soros végrehajtássá alakítása, mintsem új párhuzamosságok bevezetése.

Méret, dinamika, kooperatív viselkedés

Eddig csupa olyan tulajdonságot említettünk, amelyek miatt a beágyazott, valósidejű rendszerek fejlesztése nehezebb, mint más rendszereké. Említsünk néhány olyat is, amelyik *valamelyest egyszerűsíti* a helyzetet.

(1) A rendszer dekompozíciója után a szigorú idő- és teljesítménykövetelmények általában néhány, *viszonylag kisebb méretű* alrendszerre korlátozhatók. Sajnálatosan ezek még mindig túl nagyok ahhoz, hogy megvalósításukra *ad hoc* megoldásokat alkalmazhassunk.

(2) A rendszert alkotó komponensek (objektumok) többsége már a tervezéskor ismert, így *statikussá* tehető. Ezzel az objektumok dinamikus létrehozása és megsemmisítése – ami a végrehajtási idő kiszámíthatóságát veszélyezteti – kritikus esetekben elkerülhető. Sajnos ez nem jelenti azt, hogy a rendszeren teljes élettartama alatt semmit nem kell változtatni. Figyelemmel az esetleg tízévekben mérhető élettartamra, biztosra vehető, hogy a követelmények a rendszer élete során változni fognak. Fel kell készülni a módosításokra és a továbbfejlesztésre, sőt, ezeket esetleg üzem közben, leállás nélkül kell végrehajtani.

(3) Mivel általában egy vezető rendszertervező koordinálja a munkát, feltételezhető, hogy egymást ismerő, *kooperatívan viselkedő* szereplők (taszkok, objektumok) fognak a rendszerben együtt működni. Ez könnyebb helyzet, mint amivel például az általános célú, többfelhasználós operációs rendszerek tervezői találkoznak, ahol egymást nem ismerő, versengő, hibákkal teli felhasználói taszkok konkurens futtatását kell megoldani.

Végezetül elmondhatjuk, hogy a *valósidejű* rendszer szóhasználat annyira összefonódott a *beágyazott* rendszerekkel, hogy a két kifejezést gyakran szinonimaként használjuk. Amikor valósidejű rendszerről beszélünk, akkor általában az összes fent felsorolt követelményt automatikusan érvényesnek tekintjük. Ennek ellenére a következő pontokban elsősorban a szűkebb értelmezésre, tehát csak az időbeli viselkedéssel kapcsolatos problémákra koncentrálunk.

1.3. 5.1.3. Közkeletű félreértések és vitapontok

Valósidejű = nagyon gyors?

A fejezet bevezetőjében említett gyakori félreértések, féligazságok közül elsőként a talán leginkább elterjedtet említjük meg. Eszerint: *a valósidejű rendszerek lényege, hogy nagyon gyors rendszerek.* Ennek a felfogásnak a jegyében gondolják sokan, hogy bizonyos termékek (hardverelemek, programcsomagok, programozási nyelvek stb.) felhasználhatósága a valósidejű rendszerekben kizárólag azon múlik, hogy elég gyors működésűek-e. Valójában a valósidejű rendszerek specifikációiban előforduló időadatok igen széles tartományban mozognak. Nagy tehetetlenségű hőtechnikai folyamatok például órás nagyságrendű időállandókkal jellemezhetők, szemben mondjuk egy részecskegyorsítóban lejátszódó jelenségekkel, ahol a pikoszekundum (10-12 s) is túl nagy egység az időadatok kifejezésére. Az eszközök működési sebességének tehát inkább abban van szerepe, hogy az adott eszköz milyen feladatokra használható. Természetesen az is igaz, hogy egy nagy teljesítményű eszközzel egy kisebb igényű feladatot kisebb tervezői erőfeszítéssel lehet megoldani. Másként fogalmazva, a valósidejűség problematikája akkor bontakozik ki, amikor a terhelés megközelíti (esetleg meg is haladja) azt a határt, amelyet a rendszer a beépített maximális számítási teljesítmény mozgósításával ki tud szolgálni. A feladathoz mért ésszerű teljesítményű eszközök kiválasztásakor azonban a működési sebességnél talán még fontosabb szempont a sebességadatok megbízhatósága, kiszámíthatósága és a kivételes helyzetekben való viselkedés ismerete.

Valósidejű = konkurens?

Egy másik szokásos félreértés szerint *a valósidejű rendszerek minden esetben konkurens rendszerek*, és megfordítva, *azok a szoftverrendszerek* (pl. operációs rendszerek), *amelyek lehetőséget adnak a konkurens folyamatok kezelésére, biztosan használhatók valósidejű rendszerekben.*

A konkurens folyamatok használatának indokairól korábban már szóltunk. Valóban, a valósidejű rendszerek jelentős része egyszerűbben tervezhető és implementálható együttműködő, konkurens folyamatokkal. Azon túl azonban, hogy egy operációs rendszer képes folyamatkezelésre, – pontosan az időkövetelmények teljesítésének tervezhetősége szempontjából – rendkívül fontos a proceszor és a többi erőforrás ütemezésének megoldása, és talán még fontosabb az időmérés, időkezelés megoldása. Ezt azért hangsúlyozzuk, mert nem ismerünk olyan, széleskörűen használt programnyelvet, vagy operációs rendszert, amelyiket a valósidejű rendszerek szempontjából ideálisnak nevezhetnénk. Az az érdekes helyzet alakult ki, hogy az igazán fontos, szigorúan valósidejű funkciókat a rendszerek jelentős része (például a folyamatirányító rendszerek PLC-i és szabályozói) – pontosan a kiszámíthatóság érdekében – a legrosszabb esetre *(worst case)* méretezett, fix programciklusban hajtja végre. A konkurenciát tehát nem tekinthetjük osztályozási kritériumnak, csupán egy olyan eszköznek, amelyik egyéb, megfelelő tulajdonságok fennállása esetén jól használható.

Determinisztikus vagy valószínűségi modell szerint tervezzünk?

Harmadikként a két különböző tervezési alapelv kialakulásához vezető vitapontot tárgyaljuk.

Az *egyik álláspont* szerint szigorúan valósidejű rendszerekben nem engedhető meg olyan eszközök, módszerek és algoritmusok használata, amelyek működési idejére nem adható meg felső korlát, azaz *a legkedvezőtlenebb végrehajtási időnek egy determinisztikus modell alapján mindenképpen kiszámíthatónak kell lennie.*

A másik nézet szerint megkérdőjelezhető, hogy érdemes-e a felső időkorlát kiszámíthatóságát ennyire élesen megkövetelni. Nyugodtan tervezhetjük a végrehajtási időket is *valószínűségi modellek* alapján, hiszen a rendszerben bizonyos valószínűséggel amúgyis meghibásodnak berendezések, a szoftverben pedig minden tesztelési erőfeszítés ellenére valamilyen valószínűséggel ugyancsak maradnak hibák. *Miért ne tekinthetnénk például egy dinamikus túlterhelésből adódó határidő-mulasztást ugyanolyan kockázati tényezőnek, ugyanolyan kisvalószínűségű esetnek, mint egy hiba fellépését?*

A kiszámíthatóság oldalán a legfontosabb érv, hogy bár a hibák valószínűsége valóban fennáll, ettől még a tervezőnek nincs joga újabb bizonytalansági tényezőkkel növelni a kockázatot. Különösen akkor nincs, ha a kockázat mértékére adott becslések is bizonytalanok, mert olyan valószínűségi modellekre épülnek, amelyek érvényessége az adott rendszerre gyakorlatilag nem bizonyítható. Mind a mai napig emlegetik az űrrepülőgép első felszállásának elhalasztását [Sta88], amit a későbbi vizsgálat eredményei szerint egy CPU túlterhelési tranziens okozott (megjegy-zendő, hogy ennek valószínűsége az utólagos vizsgálatok szerint 1/67 volt, ami már nem elhanyagolható érték, azonban a kockázatot a tervezéskor nem is-merték fel.)

A másik nézet érvei szerint a legkedvezőtlenebb esetre történő méretezés irreális teljesítmény-követelményekre vezet, bizonyos körülmények között pedig (t.i. ha egy reakciót igénylő külső esemény két egymást követő előfordulása közötti időre nem adható meg alsó korlát) egyszerűen nem alkalmazható. Felesleges a kiszámíthatóság megkövetelése miatt kitiltani az eszközök egy jelentős, egyébként jól bevált halmazát a valósidejű rendszerekből, ugyanis az alkalmazásukhoz kapcsolódó kockázati tényező igenis kezelhető, és nem haladja meg az egyéb hibákból származó kockázatot. Felesleges kirekeszteni például az olcsó és elterjedt Ethernet típusú hálózatokat, amikor – amint az kísérletekkel igazolható – egy meghatározható átlagos vonal-kihasználási szintig az üzenettovábbításra specifikált felső időkorlát túllépésének valószínűsége nem haladja meg egy hardver- vagy szoftverhiba fellépésének valószínűségét.

A vita nem lezárt, mindkét megközelítésnek megvan a maga szerepe és létjogosultsága.

2. 5.2. Időkövetelmények

2.1. 5.2.1. Az időkövetelmények megadása

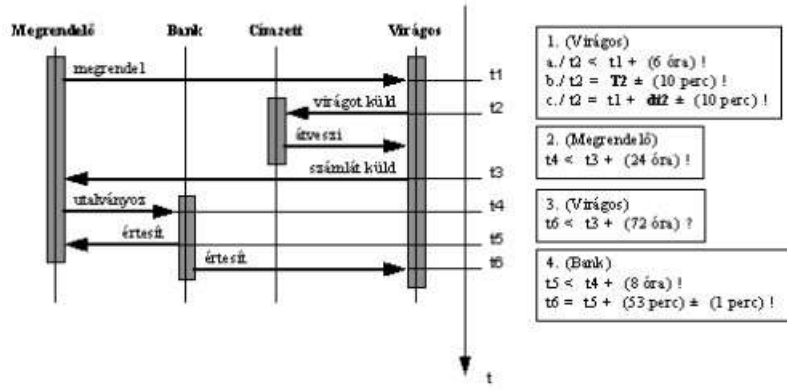
Az időkövetelmények jellegzetesen a rendszer határfelületén, a rendszer és a környezet közötti üzenetváltásokra vonatkoznak. Ennek megfelelően olyan modellen specifikálhatók, amelyik a rendszer és környezete közötti kapcsolat leírására koncentrál.

A legmagasabb absztrakciós szinten tekintsük a megvalósítandó rendszert egyetlen objektumnak, amelyik a környezetében elhelyezkedő külső objektumokkal üzenetváltások útján tart kapcsolatot. A rendszer egyrészt aktív objektumként meghatározott üzenetekkel bombázza a környezetet, másrészt pedig a környezettől kapott üzenetekre állapotától is függő válaszüzenetekkel reagál. Az üzenetek küldését és fogadását ezen a szinten tekintsük pillanatszerűnek. Az időkövetelmények legszemléletesebben egy olyan kommunikációs diagramon

ábrázolhatók, amely a "rendszer" objektumnak és a környezet objektumainak jellegzetes üzenetváltásait mutatja be.

Példaként vegyük elő a 3.48. ábrán bemutatott virágküldő szolgálat működését leíró forgatókönyvet. Tegyük fel, hogy a feladat a *Virágos* megalkotása, aki további környezeti szereplőkkel, a *Megrendelő*, *Címzett* és *Bank* objektumokkal tart kapcsolatot. Állítsunk fel néhány, a rendszer időbeli viselkedésére vonatkozó, további követelményt is.

Tegyük fel, hogy a szolgálat Seholsincsországban működik, ahol általában is, de a virágküldők piacán különösen, rendkívül éles verseny folyik, amelyben csak drákói jogszabályokkal lehet rendet tartani. Minden virágküldőnek kötelessége a megrendeléseket 6 órán belül teljesíteni. El kell fogadnia adott időpontra, illetve a rendeléstől számított adott idő múlva esedékes megrendeléseket is, amennyiben a kért időpont a megrendeléstől legalább 6 óra távolságra van. Ilyenkor a szállítási időpontot ±10 perc pontossággal kell betartani. Amennyiben nem tartja be a fenti időkövetelményeket, a cégvezetőt felnégyelik. Hasonló szigorúságú szabályok vonatkoznak a többi szereplőre is. A számlákat 24 órán belül ki kell egyenlíteni (utalványozás), a bankok pedig kötelesek a lebonyolított tranzakciókról 8 órán belül értesíteni az utalványozót, majd a kedvezményezettet. A két értesítés között ±1 perc pontossággal 53 percnek kell eltelnie (ennek a szabálynak kizárólag Seholsincsország pénzügyminisztere tudja az értelmét). A szabályok megszegőire karóbahúzás, illetve kerékbetörés vár. A szigorú szabályoknak megfelelően a résztvevőknek joga van bizonyos feltételek esetén bírósághoz fordulni. A virágos például pert indíthat, ha a számlaküldést követően 72 órán belül nem kap értesítést bankjától a számla kiegyenlítéséről, és eközben a tőle elvárható lépéseket megtette a helyzet tisztázására.



5.1. ábra

Az 5.1. ábrán az időkövetelményekkel kiegészített kommunikációs diagramot mutatjuk be. Az egyes üzenetekhez időpontokat rendeltünk (t1,...t6). A diagram jobb oldalán, bekeretezve, az időpontok között fennálló összefüggéseket írtuk le, amelyeket valamelyik szereplőnek be kell tartania, vagy ellenőriznie kell. A betartandó feltételeket felkiáltójellel jelöltük, az ellenőrizendőket pedig kérdőjellel. A betartásért, illetve ellenőrzésért felelős szereplő nevét a keret fejlécében tüntettük fel. Egy keretbe olyan feltételeket foglaltunk, amelyek betartásáért, illetve ellenőrzéséért ugyanaz a szereplő felelős a folyamat egy adott fázisában. Egy kereten belül szerepelhetnek alternatív feltételek, amelyek közül csak az egyik fordulhat elő egyidejűleg. Ezeket betűjelzéssel láttuk el. Az 1. keret például a *Virágos* által betartandó követelményeket tartalmaz, mégpedig egy adott megrendelés esetén annak típusa szerint az a./, b./ vagy a c./ követelményt kell betartani. Ahol betűjelzés nélkül szerepel több követelmény, ott valamennyi betartandó (ld. 4. keret). Megjegyezzük, hogy az ábrán nem tüntettünk fel valamennyi követelményt, csak az egyes fajták szemléltetése volt a célunk.

A bemutatott példa alapján állíthatjuk, hogy az időkövetelményeket szemléletesen és egyértelműen a kommunikációs diagramokon (forgatókönyv) fogalmazhatjuk meg (ld. 3.3.1.2. pont). Tisztában kell lennünk azonban a korlátokkal is. A kommunikációs diagram alkalmas egy oksági kapcsolat (vezérlési szál) mentén rendezett üzenetsorozat (eseménysorozat) leírására, de nem alkalmas például ismétlődések és elágazások ábrázolására. A valósidejű specifikációhoz minden olyan vezérlési szál, ezen belül minden olyan alternatíva kommunikációs diagramját meg kellene adnunk, amelyikhez időkövetelmény tartozik. Bonyolultabb rendszerek esetén ez gyakorlatilag kivitelezhetetlen.

A kommunikációs diagram kifejező erejének korlátai azonnal szembetűnővé válnak, ha figyelembe kívánjuk venni azt, hogy a *Virágos* egyidejűleg több *Megrendelővel* is kapcsolatban állhat. Aligha élhetünk azzal a

feltételezéssel, hogy egy következő megrendelés csak akkor érkezhet, amikor az előző rendelés folyamata már lezárult, azaz már a hozzá tartozó banki értesítést is megkaptuk. Ezért a *Megrendelő* és a *Virágos* közötti párbeszéd tekintetében a bemutatott forgatókönyvet egyetlen megrendelésre vonatkozó mintának tekinthetjük azzal a megjegyzéssel, hogy több ilyen forgatókönyv végrehajtása folyhat egymással időben párhuzamosan. Ha viszont több megrendeléssel kell foglalkozni párhuzamosan, felmerül a kérdés, mekkora a *Virágos* kapacitása, hány megrendelést tud kezelni egyidejűleg, visszautasíthat-e megrendelést, ha túl sok munkája gyűlik össze, és hogyan alakul a forgatókönyv, ha igen.

Ugyancsak kérdéses, hogyan vehetjük figyelembe a *Virágos* egyéb tevékenységeit - ha vannak ilyenek. Például egy olyan követelményt, hogy napi tevékenységéről a 18:00 órai zárást követően legkésőbb 19:47-ig összesítést kell küldenie az adóhivatalnak. Ezt az előírást olyan forgatókönyv segítségével ábrázolhatjuk, amelyik a Virágos napi működését írja le *nyitás, működés, zárás, jelentésküldés, pihenés* szakaszokkal. Ehhez a forgatókönyvhöz egyrészt azt a megjegyzést kell fűznünk, hogy ciklikusan ismétlődik, másrészt azt, hogy kölcsönhatásban van az alapműködést bemutató forgatókönyvvel, hiszen a *Virágos* megrendeléseket csak nyitvatartási időben fogad, míg számlaküldésre esetleg a pihenési időben is hajlandó.

További problémákba ütközünk, ha a fenti specifikációk alapján részletesebb tervezésbe kezdünk. A *Virágos* és a *Megrendelő* felelősségi körének tisztázásakor például kérdéses, hogy a *"számlát küld"* üzenet időpontját hogyan kell érteni. Mit tekintsünk a küldés pillanatának, a kiállítás, a postázás, vagy a megérkezés időpontját. Ha valamelyik – mondjuk a megérkezés – mellett döntünk, még mindig kérdéses lehet, hogy a postás csengetése, a számla kézbevétele, az elismervény aláírásakor az utolsó kézmozdulat, vagy a boríték felbontása jelenti-e azt az időpontot, amelyiket az érkezés pillanatának tekinthetünk. Amennyiben feladatunk a *Virágos* létrehozása, természetesen nem vállalhatunk felelősséget más szereplők, mint például a *Posta* vagy a *Megrendelő* működéséért. Ha egy kész, működő környezetbe illeszkedünk, az illeszkedési felületet jó előre specifikálnunk kell a meglévő rendszermodell absztrakciós szintjénél lényegesen konkrétabb formában. Ha a környezet is a fejlesztés tárgya (mások felelősségi körében), akkor az illeszkedés specifikációja a gyorsabban készülő rendszer fejlesztési ütemében kell, hogy haladjon.

2.2. 5.2.2. Az időkövetelmények típusai

Az alábbiakban az időkövetelmények jellegzetes típusaival foglalkozunk, és érzékeltetjük, hogy a tervezés és a megvalósítás folyamán az egyes követelménytípusok milyen problémákat okozhatnak.

A specifikációban az időadatok *időpontot* vagy *időtartamot* jelölhetnek. Időpont kijelölhető az egyezményes külső időskálán (*abszolút megadás*, pl. év, hó, nap, ...), illetve valamely esemény bekövetkezésének időpontjához viszonyított időtartammal (*relatív megadás,* pl. "a megrendelés megérkezésétől számított 6 óra"). Időtartamot általában két esemény között eltelt időre vonatkozóan, egyezményes mértékegység választásával és az időtartam hosszát jelző mérőszámmal adunk meg (pl. 165 ms).

A tervezendő rendszer időbeli viselkedésének leírásában a leggyakoribb követelménytípusok a következők:

- Periodikus feladat: adott időközönként ismételten végrehajtandó.

- Határidős feladat: adott időn belül végrehajtandó.

- Időzített feladat: bizonyos tűréssel egy adott időpontban végrehajtandó (befejezendő).

- Időkorlátos várakozás: a rendszer külső esemény bekövetkezésekor végez el egy feladatot, de ha a külső esemény nem következik be adott időpont eléréséig, más feladatot kell végrehajtani.

Periodikus feladatok

A rendszer bizonyos viselkedésmintákat periodikusan ismétel. A jellemző időadat a periódusidő. Kérdéses, hogy a periódusidőt milyen pontossággal kell betartani, és hogy a pontossági követelmény két egymást követő ismétlés közötti időre, vagy hosszabb távon a számított esedékességi időpontok betartására vonatkozik-e.

Ha csak a periódusidőre van pontossági előírás:

```
T=Tp±dt
```

```
                    ti+1=ti+T±dt

        T           a tényleges periódusidő
        Tp          a névleges periódusidő
        t1,...ti,...  a feladat végrehajtásának időpontjai
        dt          a megengedett eltérés
```

Tegyük fel, hogy a *Virágosnak* naponta kell jelentést küldenie az adóhivatal számára, pontosabban, két egymást követő jelentés között 24 óra ± 30 perc lehet az időkülönbség. Ez az előírás megengedi, hogy hosszú időn keresztül 24 1/2 óránként jelentsen, azaz például az első nap 19:00-kor, a másodikon 19:30-kor, a hetediken pedig már 22:00-kor küldje el a jelentést. A pontatlanság ilyen halmozódását a szakirodalom *driftnek* nevezi. Ez a megoldás nem igényli, hogy a *Virágos* pontosan járó órával rendelkezzen, elég egy olyan szerkezet, amelyik a 24 órás időtartamot fél óra pontossággal képes megmérni.

A periódusidő pontosságának előírására a másik lehetőség az, hogy az esedékesség időpontjait a pontos periódusidővel előre kiszámítjuk, és az így kapott időpontok betartásának pontosságát írjuk elő. Azaz, ha az első végrehajtás a t0 pillanatban történik, a további t1, t2, ... tn végrahajtási időpontok:

```
ti=t0+iTp±dt  (i=1,2,...n)
```

Tegyük fel például, hogy a *Virágosnak* minden nap 19:30-kor kell ± 30 perc pontossággal jelentenie. Ilyenkor a pontatlanság nem halmozódhat (driftet nem engedünk meg). A megoldáshoz a *Virágosnak* olyan órával kell rendelkeznie, amelyik ± 30 percnél pontosabban követi a külső időt.

A számítási teljesítmény méretezése szempontjából a periodikus feladatok determinisztikusan, a legrosszabb esetre (*worst case*) tervezve is jól kezelhetők.

Határidős feladatok

A határidő általában egy relatív, külső esemény bekövetkezésétől számított időtartammal meghatározott időpont (megengedett maximális reakcióidő). A feladatot az esemény bekövetkezése után, de a határidő lejárata előtt végre kell hajtani. A *Virágos* feladatai közül a megrendeléshez képest 6 órán belüli szállítás előírása egy határidős feladat.

Ha a külső esemény periodikus, tulajdonképpen a feladat is periodikus, azonban a periódusidő betartása nem a rendszer felelőssége. Periodikus külső események esetén a méretezés szempontjából a rendszer – a periodikus feladatokhoz hasonlóan – jól kezelhető.

Ha a külső esemény nem periodikus, kérdéses, hogy megadható-e olyan legkisebb időtartam, amelyen belül ugyanaz az esemény kétszer nem fordul elő. Ha igen, a feladatot *sporadikusnak* nevezzük, és a legrosszabb eset úgy kezelhető, mintha a külső esemény ezzel a periódusidővel lenne periodikus.

Ha nem tudunk minimális értéket megadni a külső esemény két egymást követő bekövetkezése közötti időtartamra, a feladatot a legkedvezőtlenebb esetben tetszőlegesen rövid időnként ismételni kell. Ilyenkor a számítási teljesítmény determinisztikus modell alapján nem méretezhető, valószínűségi jellemzők alapján (átlagos előfordulási gyakoriság, eloszlás stb.) kell dolgoznunk.

Időzített feladatok

Időzített feladatról akkor beszélünk, ha a feladat végrehajtását – pontosabban annak befejezését, azaz egy külvilágnak szóló, számított eredményeket tartalmazó üzenet kiadását – adott pontossággal egy adott időpontra kell időzíteni. Az időpont megadása lehet abszolút vagy relatív (utóbbi esetben akár külső, akár belső eseményhez képest megadva). Példánkhoz visszatérve a *Virágos* feladatai közül a meghatározott időpontra vállalt szállítások abszolút időpontra időzített feladatok (b./), a megrendeléstől számított időtartamra vállaltak külső eseményhez időzítettek (c./). Belső eseményhez időzített faladat a *Bank Virágosnak* küldött értesítése, ami 53 perccel követi a *Megrendelőnek* küldött értesítést.

Az időzített feladatok megoldását általában két lépésre bonthatjuk. Először *határidős jelleggel* a feladat számításigényes, adatfüggő időt igénylő részét kell végrehajtani, méghozzá úgy, hogy mindenképpen megfelelő időtartalékunk maradjon. Második lépésben a feladat kiszámítható végrehajtási időt igénylő részét kell elindítani, mégpedig pontosan olyan időzítéssel, hogy az időzített üzenet kiadására a megfelelő pillanatban

kerüljön sor. Az előírások teljesítéséhez nemcsak számítási teljesítményre, hanem pontos időmérésre, valamint olyan megoldásokra is szükség van, amelyekkel egy adott műveletet adott időpontban, késedelem nélkül el tudunk indítani.

Időkorlátos várakozás

A rendszer várakozik egy külső eseményre, amelynek bekövetkezésekor végrehajt egy feladatot. Ha az esemény adott időn belül nem következik be, akkor működését más műveletek végrehajtásával folytatja. A várt külső esemény általában a rendszer által küldött üzenetre érkező válasz. Az időkorlátot leggyakrabban az üzenetküldéshez (belső esemény) relatív időtartammal adjuk meg. A példabeli *Virágos* működésében a számlaküldést követően figyelhetünk meg időkorlátos várakozást a *Banktól* érkező értesítésre, amelynek 72 órán túli késése esetén a *Virágos* további intézkedésekre jogosult, amelyeket nyilván nem tesz meg, ha az értesítés időben megérkezik.

3. 5.3. A fejlesztés problémái

Ahhoz, hogy az időkövetelmények bemutatott típusait következetesen tudjuk kezelni egy rendszer tervezése és megvalósítása során, az időkövetelmények leírására alkalmas modellekre, a követelmények lebontását támogató módszertanra, az időkezelést hatékonyan támogató programozási nyelvekre, futtató rendszerekre és kommunikációs rendszerekre van szükségünk. Sajnos ilyenek a széles körben elterjedt eszközök között alig találhatók. Mind a mai napig általános gyakorlat, hogy az időkövetelményeket a tervezők intuícióik alapján kezelik, az ellenőrzés és a hangolás az implementációs fázisra marad, mintegy járulékos tevékenységként a funkcionális tesztek végrehajtását követően. Ennek eredménye, hogy a rendszer létrehozásának folyamatában gyakran több fázist átfogó visszalépésre van szükség.

Tekintsük át, milyen tipikus problémák adódnak a rendszerfejlesztés során.

Az analízis fázisában:

- A szokásostól eltérő sorrendben célszerű kidolgozni az objektum, a funkcionális és a dinamikus modellt.

- Az egyes modellek struktúrálására a szokásostól eltérő módszereket célszerű alkalmazni.

- Alkalmas technikát kell találni az időkövetelmények leírására.

- A modellekben több aktív objektumot, párhuzamos vezérlési szálakat kell kezelni.

- Már a modellezés kezdeti szakaszában is meg kell oldani az időmérés, időkezelés problémáját.

A tervezés során:

- A modellek finomításakor le kell bontani az időkövetelményeket.

- Becsléseket kell adni az egyes műveletek végrehajtási időire.

- A közös erőforrások kezelésének és ütemezésének megoldását a végrehajtási idők szempontjából is elemezni kell.

Az implementáció során:

- Fel kell mérni a programozási nyelv lehetőségeit, szükség esetén a modellek implementálását segítő osztálykönyvtárakat kell kialakítani (például aktív objektum, óra stb.).

- Megfelelő interfész-objektumokat kell létrehozni az operációs rendszer és a kommunikációs rendszer felé.

Vizsgáljunk meg ezek közül kettőt részletesebben.

Hol ragadjuk meg a problémát?

Egy rendszer fejlesztése során – mint azt már az előző fejezetekben megállapítottuk – három nézőpontból kell a rendszer mind részletesebb modelljét kibontanunk: a szereplők (adatok, objektumok), a műveletek (adattranszformációk, funkciók), valamint a vezérlési szálak (folyamatok, algoritmusok) oldaláról. Különböző módszertanok más-más aspektust helyeznek előtérbe, más-más domináns fogalmat jelölnek ki a három közül. Az előző fejezetekben bemutatott módszertan például – hasonlóan a legtöbb objektum-orientált módszertanhoz – a szereplők és azok kapcsolatainak feltérképezésével kezdi a megoldást (objektummodell), azonban a másik két aspektus leírását is lehetővé teszi (dinamikus modell, funkcionális modell).

A valósidejű rendszerek leggyakrabban *viselkedésük* leírásával, azaz vezérlési szálak mentén végzett tevékenységek megadásával ragadhatók meg, ami megfelel az objektum-orientált módszertanok dinamikus modelljének. Kezdetben a rendszert egyetlen objektumnak tekinthetjük, amelyik a környezetében található külső objektumokkal tart kapcsolatot. Ez a megközelítés a strukturált tervezési módszerekből ismert kontext-diagramnak felel meg. A viselkedés leírásához megadjuk a rendszer és a külső objektumok között zajló, a működést jellemző üzenetváltásokat. Lehetnek esetek – különösen, ha a rendszer viselkedése eleve több párhuzamos vezérlési szállal írható le – amikor már kezdetben célszerű néhány együttműködő objektummá bontani a rendszert, és ezek dinamikus modelljeit vizsgálni.

A viselkedés leírásával együtt az időkövetelményeket is rögzíteni kell valamilyen formában: forgatókönyveken, szövegesen, vagy idődiagramok megadásával.

Összességében állíthatjuk, hogy a valósidejű rendszerek körében a dinamikus modelleknek nagyobb szerepe van, mint más rendszerekben. Ezért számos objektum-orientált módszertan és a hozzá kapcsolódó CASE rendszer az állapotmodellek kezelésére, a kommunikációs modellek és a forgatókönyvek felvételére alkalmas eszközeit úgy hirdeti, mint valósidejű problémák kezelésére alkalmas modult (kiterjesztést).

Az idő mérése és kezelése

A valósidejű rendszerekben mérhetővé és lekérdezhetővé kell tenni a környezeti, valós időt, továbbá meg kell oldani időtartamok mérését és műveletek adott időpontban történő indítását. Általában már a modellezés során feltételezzük, hogy a rendszerben van folyamatosan járó *óra (clock)*, amelyik bármikor lekérdezhető, valamint vannak *ébresztőórák (watch-dog, timer)*, amelyek felhúzhatók úgy, hogy adott pillanatban kiváltsanak egy eseményt. A valósidejű feladatokra alkalmas számítógép-hardver és operációs rendszer lehetővé teszi, hogy ezek a modellek implementálhatók legyenek.

A modellekben az óráról feltételezzük, hogy *Óra.Beállít (t:időpont)* művelettel beállítható, *Óra.Lekérdez* (var *t:időpont)* művelettel pedig lekérdezhető, továbbá működés közben $p=dt/T$ pontossággal együttfut a valós idővel, ahol *dt* a *T* idő alatt keletkező eltérés abszolút értéke.

Az ébresztőt a *Vekker.Beállít (T:időtartam, E:esemény)*, vagy a *Vekker.Beállít (t:időpont, E:esemény)* művelettel állítjuk be. Az első esetben *T* idő múlva, a második esetben *t* időpontban váltja ki az ébresztő az *E* eseményt (üzenetet). Valamely vezérlési szál mentén az *E* eseményre a *Vár (E:esemény)* művelettel várakozhatunk. Ezekkel az eszközökkel a működés bizonyos időre történő felfüggesztése is megoldható, felhúzzuk az ébresztőt a kívánt *T* időtartamra, majd várakozunk az általa kiváltott eseményre.

Az óra és az ébresztő kifejezett megjelenítése helyett az időmérés beépített (implicit) megoldását is feltételezhetjük, például egy *Késleltet (T:időtartam)* művelettel is felfüggeszthetjük *T* időre a működést.

Időkorlátos várakozásra egy vezérlési szál mentén a *KorláttalVár (T:időtartam, E:esemény,* var *OK:jelző)* művelet adhat megoldást, amelyik akár az *E* esemény bekövetkezésekor, akár a *T* idő leteltekor továbblép, és a továbblépés okát jelzi az *OK* paraméterben.

Ahogyan a tervezéssel haladunk, az időkövetelményeket le kell bontanunk, és a betarthatóság érdekében az egyes műveletek végrehajtására határidőket kell szabnunk. Ezt a modellekben egy határidő paraméter átadásával jelezhetjük. Ez egyrészt lehetőséget ad a végrehajtónak arra, hogy sürgősség esetén esetleg gyengébb minőségű eredményt szolgáltasson. Másrészt ha a végrehajtás nem fejeződik be az adott határidőre, akkor felfüggesztődik, és a végrehajtatást kérő objektum vagy folyamat végrehajtása speciális feltételek mellett (például hibakezelés ágon) folytatódhat.

A feladat megoldása során a követelményekhez igazodóan elő kell írnunk az időalap felbontását, valamint az időtartam szükséges átfogási tartományát, ami meghatározza az időtartamot, vagy időpontot tároló változók típusát.

A programozási nyelv néhány esettől eltekintve általában alig ad beépített támogatást az időméréshez, illetve időkezeléshez. Az implementációt inkább az operációs rendszer, illetve a hardver elérését segítő függvény-, eljárás-, illetve objektumkönyvtár segíti. Az ismertebb, beépített időkezelést alkalmazó nyelvek (például ADA) lehetőségei is meglehetősen szegényesek.

Illusztrációként az 5.1. ábra példáját elővéve írjuk le a *Virágos* működését egy rendelés kiszolgálása során.

```
Kiszolgál:
VÁR (megrendel:esemény);
Készít (T:határidő);
Virágot_küld;
VÁR (átveszi:esemény);    ??? meddig ???
Számlát_küld;
VÁR (értesít:esemény, 72óra:időkorlát, OK:jelző);
if not OK then Vizsgálatot kér
END Kiszolgál;
```

4. 5.4. Valósidejű feladatokra ajánlott módszertanok

A legelterjedtebb, CASE eszközökkel is támogatott módszertanok általában beérik azzal, hogy fokozottabban támogatják a viselkedés leírását (dinamikus modellek, állapotgép), továbbá eszközt adnak a szoftver modulokra bontására. Azokra a problémákra, amelyek több együttműködő aktív objektum jelenlétéből, a végrehajtási idők kiszámíthatóságának igényéből, a rendszer elosztottságából adódnak, általában csak jótanács-gyűjtemény szintjén térnek ki. Nem támogatják az időzítési követelmények korai szakaszban történő figyelembevételét és lebontását, illetve a lebontás helyességének igazolását sem.

A valósidejű rendszerek tervezési módszertanát jelentősen befolyásolta Paul T. Ward és Stephen J. Mellor 1985-ben publikált munkája, amelyik a strukturált módszereket terjesztette ki a valósidejű rendszerekre [WaM85]. Később mindketten – más-más szerzőtársakkal – publikáltak egy-egy objektum-orientált módszertant is: a Shlaer-Mellor [ShM92], illetve a ROOM [SGW94] módszertant. Az érdeklődő olvasóknak az irodalomjegyzék adatai alapján elsősorban ezt a két könyvet, valamint Coad és Yourdon munkáját [Coa90] ajánljuk figyelmébe.

6. fejezet - 6. Objektumorientált programozás C++ nyelven

1. 6.1. A C++ nyelv kialakulása

A C++ nyelv elődjét a C nyelvet jó húsz évvel ezelőtt rendszerprogramozáshoz (UNIX) fejlesztették ki, azaz olyan feladathoz, melyhez addig kizárólag assembly nyelveket használtak. A C nyelvnek emiatt egyszerűen és hatékonyan fordíthatónak kellett lennie, amely a programozót nem korlátozza és lehetővé teszi a bitszintű műveletek megfogalmazását is. Ezek alapvetően assembly nyelvre jellemző elvárások, így nem véletlen, hogy a megszületett magas szintű nyelv az assembly nyelvek tulajdonságait és egyúttal hiányosságait is magában hordozza. Ilyen hiányosságok többek között, hogy az eredeti (ún. Kerninghan-Ritchie) C nem ellenőrzi a függvény-argumentumok számát és típusát, nem tartalmaz I/O utasításokat, dinamikus memória kezelést, konstansokat stb. Annak érdekében, hogy a fenti hiányosságok ne vezessenek a nyelv használhatatlanságához, ismét csak az assembly nyelveknél megszokott stratégiához folyamodtak - egy szövegfeldolgozó előfordítóval (pre-processzorral) egészítették ki a fordítóprogramot (mint a makro-assemblereknél) és egy függvénykönyvtárat készítettek a gyakran előforduló, de a nyelvben nem megvalósított feladatok (I/O, dinamikus memóriakezelés, trigonometriai, exponenciális stb. függvények számítása) elvégzésére. Tekintve, hogy ezek nyelven kívüli eszközök, azaz a C szemantikáról mit sem tudnak, használatuk gyakran elfogadhatatlanul körülményes (pl. malloc), vagy igen veszélyes (pl. makrok megvalósítása #define-nal). A C rohamos elterjedésével és általános programozási nyelvként történő felhasználásával a fenti veszélyek mindinkább a fejlődés kerékkötőivé váltak. A C nyelv fejlődésével ezért olyan elemek jelentek meg, amelyek fokozták a programozás biztonságát (pl. a prototípus argumentum deklarációkkal) és lehetővé tették az addig csak előfordító segítségével elérhető funkciók kényelmes és ugyanakkor biztonságos megvalósítását (pl. konstans, felsorolás típus).

A C++ nyelv egyrészt ezt a fejlődési irányt követi, másrészt az **objektumorientált programozási nyelvek** egy jellemző tagja. Ennek megfelelően a C++ nyelvet alapvetően két szempontból közelíthetjük meg. Vizsgálhatjuk a C irányából - amint azt a következő fejezetben tesszük - és az objektumorientált programozás szemszögéből, ami a könyv további részeinek elsődleges célja.

2. 6.2. A C++ programozási nyelv nem objektumorientált újdonságai

2.1. 6.2.1. A struktúra és rokonai neve típusértékű

A C nyelvben a különböző típusú elemek egy egységként való kezelésére vezették be a **struktúrát**. Például egy hallgatót jellemző adatok az alábbi struktúrába foglalhatók össze:

```
struct  student {
char    name[40];
int     year;
double  average;
        };
```

A típusnevet C-ben ezek után a struct student jelenti, míg C++-ban a struct elhagyható, így nem kell teleszemetelnünk struct szócskákkal a programunkat. Egy student típusú változó definiálása tehát C-ben és C++-ban:

```
     Típus            Változó (objektum)
C:   struct student   jozsi;
C++: student          jozsi;
```

2.2. 6.2.2. Konstansok és makrok

Konstansokat az eredeti C-ben csak az előfordító direktíváival hozhatunk létre. C++-ban (és már az ANSI C-ben is) azonban a const típusmódosító szó segítségével bármely memóriaobjektumot definiálhatunk konstansként, ami azt jelenti, hogy a fordító figyelmeztet, ha a változó nevét értékadás bal oldalán szerepeltetjük, vagy ebből nem konstansra mutató pointert inicializálunk. A konstans használatát a ? (PI) definiálásával mutatjuk be, melyet egyúttal a C-beli megoldással is összevetünk:

```
C: #define PI 3.14   C++: const float PI = 3.14;
```

Mutatók esetén lehetőség van annak megkülönböztetésére, hogy a mutató által megcímzett objektumot, vagy magát a mutatót kívánjuk konstansnak tekinteni:

```
const char * p; //p által címzett karakter nem módosítható

char * const q; //q-t nem lehet megváltoztatni
```

A konstansokhoz hasonlóan a C-ben a makro is csak előfordítóval valósítható meg. Ki ne találkozott volna olyan hibákkal, amelyek éppen abból eredtek, hogy az előfordító, mint nyelven kívüli eszköz mindent gondolkodás nélkül helyettesített, ráadásul az eredményt egy sorba írva azt sem tette lehetővé, hogy a makrohelyettesítést lépésenként nyomkövessük.

Emlékeztetőként álljon itt egy elrettentő példa:

```
#define abs(x) (x < 0) ? -x : x  // !!!

int y, x = 3;
y = abs( x++ );       // Várt: x = 4, y = 3;
```

Az abszolút érték makro fenti alkalmazása esetén, ránézésre azt várnánk, hogy az y=abs(x++) végrehajtása után, mivel előtte x értéke 3 volt, x értéke 4 lesz, míg y értéke 3. Ez így is lenne, ha az abs-ot függvényként realizálnánk. Ezzel szemben a előfordító ebből a sorból a következőt készíti:

```
y = (x++ < 0) ? - x++ : x++;
```

azaz az x-et kétszer inkrementálja, minek következtében az utasítás végrehajtása után x értéke 5, míg y-é 4 lesz. A előfordítóval definiált makrok tehát igen veszélyesek.

C++-ban, a veszélyeket megszüntetendő, a makrok függvényként definiálhatók az **inline** módosító szócska segítségével. Az inline típusú függvények törzsét a fordító a lehetőség szerint a hívás helyére befordítja az előfordító felhasználásánál fellépő anomáliák kiküszöbölésével.

Tehát az előbbi példa megvalósítása C++-ban:

```
inline int abs(int x) {return (x < 0) ? -x : x;}
```

2.3. 6.2.3. Függvények

A függvény a programozás egyik igen fontos eszköze. Nem véletlen tehát, hogy a C++-ban ezen a területen is számos újdonsággal találkozhatunk.

Pascal-szerű definíciós szintaxis

Nem kimondott újdonság, de a C++ is a Pascal nyelvnek, illetve az ANSI C-nek megfelelő paraméter-definíciót ajánlja, amely szerint a paraméter neveket, mind azok típusát a függvény fejlécében szerepeltetjük. Egy változócserét elvégző (xchg) függvény definíciója tehát:

```
void xchg ( int * pa, int * pb )  { ... }
```

Kötelező prototípus előrehivatkozáskor

Mint ismeretes az eredeti C nyelvben a függvény-argumentumokra nincs darab- és típusellenőrzés, illetve a visszatérési érték típusa erre utaló információ nélkül int. Ez programozási hibák forrása lehet, amint azt újabb elrettentő példánk is illusztrálja:

```
a függvényt hívó programrész  a hívott függvény
double z = sqrt( 2 );  double sqrt( double x ) {...}
```

A négyzetgyök (sqrt) függvényt hívjuk meg azzal a szándékkal, hogy a 2 négyzetgyökét kiszámítsa. Mivel tudjuk, hogy az eredmény valós lesz, azt egy double változóban várjuk. Ha ezen utasítás előtt a programfájlban nem utaltunk az sqrt függvény deklarációjára (miszerint az argumentuma double és a visszatérési értéke is double), akkor a fordító úgy tekinti, hogy ez egy int típusú függvény, melynek egy int-et (a konstans 2-t) adunk át. Azaz a fordító olyan kódot készít, amely egy int 2 számot a veremre helyez (a paraméter-átadás helye a verem) és meghívja az sqrt függvényt. Ezek után feltételezve, hogy a hívott függvény egy int visszatérési értéket szolgáltatott (Intel processzoroknál ez azt jelenti, hogy az AX regiszterben van az eredmény), az AX tartalmából egy double-t konvertál és elvégzi az értékadást. Ehhez képest az sqrt függvény meghívásának pillanatában azt hiszi, hogy a veremben egy double érték van (ennek mérete és szemantikája is egészen más mint az int típusé, azaz semmiképpen sem 2.0), így egy értelmetlen számból von négyzetgyököt, majd azt a regiszterekben úgy helyezi el (pl. a lebegőpontos társprocesszor ST(0) regiszterében), ahogyan a double-t illik, tehát véletlenül sem oda és olyan méretben, ahogyan az int visszatérési értékeket kell. Tehát mind az argumentumok átadása, mind pedig az eredmény visszavétele hibás (sajnálatosan a két hiba nem kompenzálja egymást).

Az ilyen hibák az ANSI C-ben **prototípus** készítésével kiküszöbölhetők. A prototípus olyan függvény-deklaráció, amely a visszatérési érték és a paraméter típusokat definiálja a fordító számára. Az előző példában a következő sort kell elhelyeznünk az sqrt függvény meghívása előtt:

```
double sqrt( double );
```

A prototípusok tekintetében a C++ nyelv újdonsága az, hogy míg a prototípus a C-ben mint lehetőség szerepel, addig a C++-ban kötelező. Így a deklarációs hibákat minimalizálhatjuk anélkül, hogy a programozó lelkiismeretességére lennénk utalva.

Alapértelmezés szerinti argumentumok

Képzeljük magunkat egy olyan programozó helyébe, akinek int → ASCII konvertert kell írnia, majd azt a programjában számtalan helyen felhasználnia. A konverter rutin (IntToAscii) paramétereit kialakíthatjuk úgy is, hogy az első paraméter a konvertálandó számot tartalmazza, a második pedig azt, hogy milyen hosszú karaktersorozatba várjuk az visszatérési értékként előállított eredményt. Logikus az a megkötés is, hogy ha a hossz argumentumban 0 értéket adunk meg, akkor a rutinnak olyan hosszú karaktersorozatot kell létrehoznia, amibe az átalakított szám éppen belefér. Nem kell nagy fantázia ahhoz, hogy elhiggyük, hogy a konvertert felhasználó alkalmazások az esetek 99 százalékában ezen alapértelmezés szerint kívánják az átalakítást elvégezni. A programok tehát hemzsegni fognak az olyan IntToAscii hívásoktól, amelyekben a második argumentum 0. Az alapértelmezésű (**default**) argumentumok lehetővé teszik, hogy ilyen esetekben ne kelljen teleszórni a programot az alapértelmezés szerinti argumentumokkal, a fordítóra bízva, hogy az alapértelmezésű paramétert behelyettesítse. Ehhez az IntToAscii függvény deklarációját a következőképpen kell megadni:

```
char * IntToAscii( int i, int nchar = 0 );
```

Annak érdekében, hogy mindig egyértelmű legyen, hogy melyik argumentumot hagyjuk el, a C++ csak az argumentumlista végén enged meg alapértelmezés szerinti argumentumokat, melyek akár többen is lehetnek.

Függvények átdefiniálása (overloading)

A függvény valamilyen összetett tevékenységnek a programnyelvi absztrakciója, míg a tevékenység tárgyait általában a függvény argumentumai képviselik. A gyakorlati életben gyakran találkozunk olyan tevékenységekkel, amelyeket különböző típusú dolgokon egyaránt végre lehet hajtani, pl. vezetni lehet autót,

repülőgépet vagy akár tankot is. Kicsit tudományosabban azt mondhatjuk, hogy a "vezetni" **többrétű,** azaz **polimorf** tevékenység, vagy más szemszögből a "vezetni" kifejezést több eltérő tevékenységre lehet alkalmazni. Ilyen esetekben a tevékenység pontos mivoltát a tevékenység neve és tárgya(i) együttesen határozzák meg. Ha tartani akarnánk magunkat ahhoz az általánosan elfogadott konvencióhoz, hogy a függvény nevét kizárólag a tevékenység neve alapján határozzuk meg, akkor nehézséget jelentene, hogy a programozási nyelvek általában nem teszik lehetővé, hogy azonos nevű függvénynek különböző paraméterezésű változatai egymás mellett létezzenek. Nem így a C++, amelyben egy függvényt a neve és a paramétereinek típusa együttesen azonosít.

Tételezzük fel, hogy egy érték két határ közötti elhelyezkedését kell ellenőriznünk. A tevékenység alapján a Between függvénynév választás logikus döntésnek tűnik. Ha az érték és a határok egyaránt lehetnek egész (int) és valós (double) típusúak, akkor a Between függvénynek két változatát kell elkészítenünk:

```
// 1.változat, szignatúra= double,double,double
int Between(double x, double min, double max) {
return ( x >= min && x <= max );
}
// 2.változat, szignatúra= int,int,int
int Between(int x, int min, int max) {
return ( x >= min && x <= max );
}
```

A két változat közül, a Between függvény meghívásának a feldolgozása során a fordítóprogram választ, a tényleges argumentumok típusai, az ún. **paraméter szignatúra**, alapján. Az alábbi program első Between hívása a 2. változatot, a második hívás pedig az 1. változatot aktivizálja:

```
int x;
int y = Between(x, 2, 5);  //2.változat
    //szignatúra=int,int,int
double f;
y = Between(f, 3.0, 5.0);  //1.változat
    //szignatúra=double,double,double
```

A függvények átdefiniálásának és az alapértelmezés szerinti argumentumok közös célja, hogy a fogalmi modellt a programkód minél pontosabban tükrözze vissza, és a programnyelv korlátai ne torzítsák el a programot a fogalmi modellhez képest.

2.4. 6.2.4. Referencia típus

A C++-ban a C-hez képest egy teljesen új típuscsoport is megjelent, melyet **referencia típusnak** hívunk. Ezen típus segítségével **referencia változókat** hozhatunk létre. Definíciószerűen a **referencia egy alternatív név egy memóriaobjektum (változó) eléréséhez**. Ha bármikor kétségeink vannak egy referencia értelmezésével kapcsolatban, akkor ehhez a definícióhoz kell visszatérnünk. Egy X típusú változó referenciáját X& típussal hozhatjuk létre. Ha egy ilyen referenciát explicit módon definiálunk, akkor azt kötelező inicializálni is, hiszen a referencia valaminek a helyettesítő neve, tehát meg kell mondani, hogy mi az a valami. Tekintsük a következő néhány soros programot:

```
int v = 1;
int& r = v;       // kötelező inicializálni
int x = r;        // x = 1
r = 2;            // v = 2
```

Mivel az r a v változó helyettesítő neve, az int& r = v; sor után bárhol ahol a v-t használjuk, használhatnánk az r-et is, illetve az r változó helyett a v-t is igénybe vehetnénk. A referencia típus implementációját tekintve egy

konstans mutató, amely a műveletekben speciális módon vesz részt. Az előbbi rövid programunk, azon túl, hogy bemutatta a referenciák használatát, talán arra is rávilágított, hogy az ott sugallt felhasználás a programot könnyedén egy kibogozhatatlan rejtvénnyé változtathatja.

A referencia típus javasolt felhasználása nem is ez, hanem elsősorban a C-ben hiányzó **cím** (azaz referencia) **szerinti paraméter átadás** megvalósítása. Nézzük meg példaként az egész változókat inkrementáló (incr) függvény C és C++-beli implementációját. Mivel C-ben az átadott paramétert a függvény nem változtathatja meg (**érték szerinti** átadás), kénytelenek vagyunk a változó helyett annak címét átadni melynek következtében a függvény törzse a járulékos indirekció miatt jelentősen elbonyolódik. Másrészt, ezek után az incr függvény meghívásakor a címképző operátor (&) véletlen elhagyása Damoklész kardjaként fog a fejünk felett lebegni.

C:

```
void incr( int * a ) {
   (*a)++; //"a" az "x" címe
}
....
int x = 2;
incr( &x );
```

C++:

```
void incr( int& a ) {
   a++;  //"a" az "x"
         //helyettesítő neve
}
....
int x = 2;
incr( x );  // Nincs &
```

Mindkét problémát kiküszöböli a referenciatípus paraméterként történő felhasználása. A függvény törzsében nem kell indirekciót használnunk, hiszen az ott szereplő változók az argumentumok helyettesítő nevei. Ugyancsak megszabadulunk a címoperátortól, hiszen a függvénynek a helyettesítő név miatt magát a változót kell átadni.

A referencia típus alkalmazásával élesen megkülönböztethetjük a cím jellegű és a belső megváltoztatás céljából indirekt módon átadott függvény-argumentumokat. Összefoglalásképpen, C++-ban továbbra is használhatjuk az érték szerinti paraméterátadást, melyet skalárra, mutatóra, struktúrára és annak rokonaira (*union*, illetve a később bevezetendő *class*) alkalmazhatunk. A paramétereket **cím szerint** - tehát vagy a megismert referencia módszerrel, vagy a jó öreg indirekcióval, mikor tulajdonképpen a változó címét adjuk át érték szerint - kell átadni, ha a függvény az argumentumot úgy kívánja megváltoztatni, hogy az a hívó program számára is érzékelhető legyen, vagy ha a paraméter tömb típusú. Gyakran használjuk a cím szerinti paraméterátadást a hatékonysági szempontok miatt, hiszen ebben az esetben csak egy címet kell másolni (az átadást megvalósító verem memóriába), míg az érték szerinti átadás esetén a teljes változót, ami elsősorban struktúrák és rokonaik esetében jelentősen méretet is képviselhet.

2.5. 6.2.5. Dinamikus memóriakezelés operátorokkal

A C nyelv definíciója nem tartalmaz eszközöket a dinamikus memóriakezelés elvégzésére, amit csak a C-könyvtár felhasználásával lehet megvalósítani. Ennek következménye az a C-ben jól ismert, komplikált és veszélyes memória foglaló programrészlet, amelyet most egy struct Student változó lefoglalásával és felszabadításával demonstrálunk:

C: könyvtári függvények

```
#include <malloc.h>
....
struct Student * p;
p = (struct Student *)
     malloc(sizeof(struct Student));
if (p == NULL) ....
....
free( p );
```

C++: operátorok

```
Student * p;
p = new Student;
....
delete p;
```

C++-ban nyelvi eszközökkel, operátorokkal is foglalhatunk dinamikus memóriát. A foglalást a **new operátor** segítségével végezhetjük el, amelynek a kért változó típusát kell megadni, és amely ebből a memóriaterület méretét és a visszaadott mutató típusát már automatikusan meghatározza. A lefoglalt területet a **delete operátorral** szabadíthatjuk fel. Tömbök számára is hasonló egyszerűséggel foglalhatunk memóriát, az elemtípus és tömbméret megadásával. Pl. a 10 Student típusú elemet tartalmazó tömb lefoglalása a

```
Student * p = new Student[10];
```

utasítással történik.

Amennyiben a szabad memória elfogyott, így a memóriafoglalási igényt nem lehet kielégíteni a C könyvtár függvényei NULL értékű mutatóval térnek vissza. Ennek következménye az, hogy a programban minden egyes allokációs kérés után el kell helyezni ezt a rendkívüli esetet ellenőrző és erre valamilyen módon reagáló programrészt. Az új new operátor a dinamikus memória elfogyása után, pusztán történelmi okok miatt, ugyancsak NULL mutatóval tér vissza, de ezenkívül a **new.h** állományban deklarált **_new_handler** globális mutató által megcímzett függvényt is meghívja. Így a rendkívüli esetek minden egyes memóriafoglalási kéréshez kapcsolódó ismételt kezelése helyett csupán a _new_handler mutatót kell a saját hibakezelő függvényre állítani, amelyben a szükséges lépéseket egyetlen koncentrált helyen valósíthatjuk meg. A következő példában ezt mutatjuk be:

```
#include <new.h> // itt van a _new_handler deklarációja

void OutOfMem( ) {
printf("Nagy gáz van,kilépek" ); exit( 1 );
}
main( ) {
set_new_handler( OutOfMem );
char * p = new char[10000000000L]; // nincs hely
}
```

2.6. 6.2.6. Változó-definíció, mint utasítás

A C nyelvben a változóink lehetnek **globálisak**, amikor azokat **függvényblokkokon** ({ } zárójeleken) kívül adjuk meg, vagy **lokálisak**, amikor a változódefiníciók egy blokk elején szerepelnek. Fontos szabály, hogy a lokális változók definíciója az egyéb utasításokkal nem keveredhet, a definícióknak a blokk első egyéb utasítása előtt kell elhelyezkedniük. C++-ban ezzel szemben lokális változót bárhol definiálhatunk, ahol egyébként utasítást megadhatunk. Ezzel elkerülhetjük azt a gyakori C programozási hibát, hogy a változók definíciójának és első felhasználásának a nagy távolsága miatt inicializálatlan változók értékét használjuk fel. C++-ban ajánlott követni azt a vezérelvet, hogy ha egy változót létrehozunk, akkor rögtön inicializáljuk is.

Egy tipikus, az elvet tiszteletben tartó, C++ programrészlet az alábbi:

```
{
int z = 3, j = 2;
for( int i = 0; i < 10; i++ ) {
   z += k;
  int k = i - 1;
}
j = i++;
}
```

A változók **élettartamával** és **láthatóságával** kapcsolatos szabályok ugyanazok mint a C programozási nyelvben. Egy lokális változó a definíciójának az elérésekor születik meg és azon blokk elhagyásakor szűnik meg, amelyben definiáltuk. A lokális változót a definíciós blokkjának a definíciót követő részén, valamint az ezen rész által tartalmazott egyéb blokkokon belül érhetjük el, azaz "látjuk". A globális változók a **main függvény** meghívása előtt születnek meg és a program leállása (a main függvényből történő kilépés, vagy exit hívás) során haláloznak el.

Nem teljesen egyértelmű, hogy a fenti programrészletben a for ciklus fejében deklarált i változó a for cikluson kívül létezik-e még vagy sem. Korábbi C++ fordítók, erre a kérdésre igennel válaszoltak, így a ciklus lezárása után még jogosan használtuk az i értékét egy értékadásban. Újabb C++ fordítók azonban azt az értelmezést követik, hogy a fejben definiált változó a ciklushoz tartozik, így a ciklus lezárása után már nem létezik.

2.7. 6.2.7. Névterek

A névterek a típusokat, változókat és függvényeket csoportokhoz rendelhetik, így elkerülhetjük, hogy a különböző programrészletekben szereplő, véletlenül megegyező elnevezések ütközzenek egymással. A névtérhez nem sorolt elnevezések mind a globális névtérhez tartoznak. Ebben a könyvben a saját változóinkat mindig a globális névtérben helyezzük el.

Például, egy geom azonosítójú névtér definíciója a következőképpen lehetséges:

```
namespace geom {
// itt típusok, változók, függvények szerepelhetnek
int sphere;

}
```

A névtéren belül a névtér azonosítót nem kell használnunk, külső névtérből azonban a névtér azonosító :: változónév módon hivatkozhatunk a változókra. A fenti példa változóját a geom::sphere teljes névvel azonosíthatjuk.

A névtér azonosító gyakori kiírásától megkímélhetjük magunkat a using namespace geom; utasítással. Ez után a geom névtér összes neve a geom:: kiegészítő nélkül is érvényes.

A szabványos C++ könyvtár a saját típusait az std névtérben definiálja.

3. 6.3. A C++ objektumorientált megközelítése

3.1. 6.3.1. OOP nyelvek, C → C++ átmenet

A programozás az ún. imperatív programozási nyelvekben, mint a C, a Pascal, a Fortran, a Basic és természetesen a C++ is nem jelent mást mint egy feladatosztály megoldási menetének (algoritmusának) megfogalmazását a programozási nyelv nyelvtanának tiszteletben tartásával és szókincsének felhasználásával. Ha egy probléma megoldásának a menete a fejünkben már összeállt, akkor a programozás csak egy fordítási

lépést jelent, amely kusza gondolatainkat egy egyértelmű formális nyelvre konvertálja. Ez a fordítási lépés bár egyszerűnek látszik, egy lépésben történő végrehajtása általában meghaladja az emberi elme képességeit, sőt gyakorlati feladatok esetén már a megoldandó feladat leírása is túllép azon a határon, amelyet egy ember egyszerre át tud tekinteni. Emiatt csak úgy tudunk bonyolult problémákat megoldani, ha azt először már áttekinthető részfeladatokra bontjuk, majd a részfeladatokat önállóan oldjuk meg. Ezt a részfeladatokra bontási műveletet **dekompozíciónak** nevezzük. A dekompozíció a program tervezés és implementáció alapvető eleme, mondhatjuk azt is, hogy a programozás művészete, lényegében a helyes dekompozíció művészete. A feladatok szétbontásában alapvetően két stratégiát követhetünk:

1. Az első szerint arra koncentrálunk, hogy **mit** kell a megoldás során elvégezni, és az elvégzendő tevékenységet résztevékenységekre bontjuk. A feldarabolásnak csak akkor van értelme, ha azt egyszerűen el tudjuk végezni, anélkül, hogy a részfeladatokat meg kelljen oldani hozzá. Ez azt jelenti, hogy egy részfeladatot csak aszerint fogalmazunk meg, hogy abban **mit kell tenni**, és a **hogyan**-ra csak akkor térünk rá, mikor már csak ezen részfeladatra koncentrálhatunk. A belső részletek elfedését **absztrakt definíciónak**, a megközelítést pedig **funkcionális dekompozíciónak**nevezzük.

2. A második megközelítésben azt vizsgáljuk, hogy milyen "dolgok" (adatok) szerepelnek a problémában, vagy a műveletek végrehajtói és tárgyai hogyan testesíthetők meg, és eszerint vágjuk szét a problémát kisebbekre. Ezen módszer az **objektumorientált dekompozíció** alapja. A felbontás eredményeként kapott "dolgokat" most is absztrakt módon kell leírni, azaz csak azt körvonalazzuk, hogy a "dolgokon" milyen műveleteket lehet végrehajtani, anélkül, hogy az adott dolog belső felépítésébe és az említett műveletek megvalósításának módjába belemennénk.

3.2. 6.3.2. OOP programozás C-ben és C++-ban

A legelemibb OOP fogalmak bemutatásához oldjuk meg a következő feladatot:

Készítsünk programot, amely ciklikusan egy egyenest forgat 8 fokonként mialatt 3 db vektort mozgat és forgat 5, 6 ill. 7 fokonként, és kijelzi azokat a szituációkat, amikor valamelyik vektor és az egyenes párhuzamos.

Az objektumorientált dekompozíció végrehajtásához gyűjtsük össze azon "dolgokat" és "szereplőket", melyek részt vesznek a megoldandó feladatban. A rendelkezésre álló feladatleírás (informális specifikáció) szövegében a "dolgok" mint főnevek jelennek meg, ezért ezeket kell elemzés alá vennünk. Ilyen főnevek a vektor, egyenes, szituáció. A szituációt első körben ki is szűrhetjük mert az nem önálló "dolgot" (ún. objektumot) takar, hanem sokkal inkább más objektumok, nevezetesen a vektor és egyenes között fennálló pillanatnyi viszonyt, vagy idegen szóval asszociációt. A feladat szövegében 3 vektorról van szó és egyetlen egyenesről. Természetesen a különböző vektorok ugyanolyan jellegű dolgok, azaz ugyannak a típusnak a példányai. Az egyenes jellegében ettől eltérő fogalom, így azt egy másik típussal jellemezhetjük. Ennek megfelelően a fontos objektumokat két típusba (osztályba) csoportosítjuk, melyeket a továbbiakban nagy betűvel kezdődő angol szavakkal fogunk jelölni: *Vector, Line*.

A következő lépés az objektumok absztrakt definíciója, azaz a rajtuk végezhető műveletek azonosítása. Természetesen egy típushoz (pl. *Vector*) tartozó különböző objektumok (vektorok) pontosan ugyanolyan műveletekre reagálhatnak, így ezen műveleteket lényegében a megállapított típusokra kell megadni. Ezek a műveletek ismét csak a szöveg tanulmányozásával ismerhetők fel, amely során most az igékre illetve igenevekre kell különös tekintettel lennünk. Ilyen műveletek a vektorok esetén a forgatás és eltolás, az egyenes esetén pedig a forgatás. Kicsit bajba vagyunk a "párhuzamosság vizsgálat" művelet esetében, hiszen nem kézenfekvő, hogy az egyeneshez, a vektorhoz, mindkettőhöz vagy netalán egyikhez sem tartozik. Egyelőre söpörjük szőnyeg alá ezt a kérdést, majd később visszatérünk hozzá.

A műveletek implementálásához szükségünk lesz az egyes objektumok belső szerkezetére is, azaz annak ismeretére, hogy azoknak milyen belső tulajdonságai, adatai (ún. **attribútumai**) vannak. Akárhányszor is olvassuk át a feladat szövegét semmit sem találunk erre vonatkozólag. Tehát a feladat kiírás alapján nem tudjuk megmondani, hogy a vektorokat és egyenest milyen attribútumokkal lehet egyértelműen jellemezni. No persze, ha kicsit elkalandozunk a középiskolai matematika világába, akkor hamar rájövünk, hogy egy két dimenziós vektort az x és y koordinátáival lehet azonosítani, míg egy egyenest egy pontjának és irányvektorának két-két koordinátájával. (Tanulság: a feladat megfogalmazása során tipikus az egyéb, nem kimondott ismeretekre történő hivatkozás.)

Végezetül az elemzésünk eredményét az alábbi táblázatban foglalhatjuk össze:

Objektum	Típus	Attribútumok	Felelősség
vektor(ok)	*Vector*	x, y	vektor forgatása, eltolása, párhuzamosság?
egyenes	*Line*	x0, y0, vx, vy	egyenes forgatása, párhuzamosság?

Fogjunk hozzá az implementációhoz egyelőre a C nyelv lehetőségeinek a felhasználásával. Kézenfekvő, hogy a két lebegőpontos koordinátát egyetlen egységbe fogó vektort és a hely és irányvektor koordinátáit tartalmazó egyenest struktúraként definiáljuk:

```
struct Vector { double x, y; };

struct Line { double x0, y0, vx, vy; };
```

A műveleteket mint függvényeket valósíthatjuk meg. Egy ilyen függvény paraméterei között szerepeltetni kell, hogy melyik objektumon végezzük a műveletet, azaz a vektor forgatását végző függvény most az első, második vagy harmadik vektort transzformálja, valamint a művelet paramétereit is. Ilyen paraméter a forgatás esetében a forgatási szög. A függvények elnevezésében célszerű visszatükrözni azok funkcióját, tehát a vektor forgatását első közelítésben nevezzük Rotate-nek. Ez azonban még nem tökéletes, mert az egyenes is rendelkezik forgatási művelettel, viszont csak egyetlen Rotate függvényünk lehet, így a végső függvénynévben a funkción kívül a hozzá tartozó objektum típusát is szerepeltetni kell.

Ennek megfelelően a vektorokon és az egyenesen végezhető műveletek prototípusai:

```
funkció +         melyik konkrét    művelet paraméterek
objektum típus    objektumon
RotateVector     (struct Vector* v, double fi);
TranslateVector  (struct Vector* v, struct Vector d);
SetVector        (struct Vector* v, double x0,double y0);
RotateLine       (struct Line * l,  double fi);
TranslateLine    (struct Line * l,  struct Vector d);
SetLine          (struct Line * l,  struct Vector r, struct Vector v);
```

A definiált struktúrákat és függvényeket alapvető építőelemeknek tekinthetjük. Ezeket használva a programunk egy részlete, amely először (3,4) koordinátákkal egy v nevű vektort hoz létre, később annak x koordinátáját 6-ra állítja, majd 30 fokkal elforgatja, így néz ki:

```
   struct Vector v;
   SetVector( &v, 3.0, 4.0 );
   v.x = 6.0;     // : direkt hozzáférés
   RotateVector( &v, 30.0 );
```

A programrészlet áttekintése után két dolgot kell észre vennünk. Az objektumorientált szemlélet egyik alapköve, az **egységbe zárás**, amellyel az adatokat (vektorok) absztrakt módon, a rajtuk végezhető műveletekkel definiáljuk (SetVector,RotateVector), azaz az adatokat és műveleteket egyetlen egységben kezeljük, alapvetően névkonvenciók betartásával ment végbe. A vektorokon végezhető műveletek függvényei "Vector"-ra végződtek és első paraméterük vektorra hivatkozó mutató volt. A másik probléma az, hogy a struktúra belső implementációját (double x,y adattagok) természetesen nem fedtük el a külvilág elől, ezek a belső mezők a definiált műveletek megkerülésével minden további nélkül megváltoztathatók. Gondoljunk most arra, hogy például hatékonysági okokból a vektor hosszát is tárolni akarjuk a struktúrában. A hossz értékét mindig újra kell számítani, ha valamelyik koordináta megváltozik, de amennyiben a koordináták változatlanok, akárhányszor, bonyolult számítás nélkül, le lehet kérdezni. Nyilván a hossz számítását a SetVector, TranslateVector, stb.

függvényekben kell meghívni, és ez mindaddig jól is megy amíg valaki fegyelmezetlenül az egyik adattagot ezen függvények megkerülésével át nem írja. Ekkor a belső struktúra inkonzisztenssé válik, hiszen a hossz és a koordináták közötti függőség érvénytelenné válik.

Valójában már az adattagok közvetlenül történő puszta leolvasása is veszélyes lehet. Tételezzük fel, hogy a program fejlesztés egy későbbi fázisában az elforgatások elszaporodása miatt célszerűbbnek látszik, hogy Descartes-koordinátákról polár-koordinátákra térjünk át a vektorok belső ábrázolásában. A vektorhoz rendelt műveletek megváltoztatása után a vektort ezen műveleteken keresztül használó programrészek számára Descartes-polár koordináta váltás láthatatlan marad, hiszen a belső ábrázolás és a műveletek felülete között konverziót maguk a műveletek végzik el. De mi lesz a v.x kifejezés értéke? Ha az új vektor implementációjában van egyáltalán x adattag, akkor semmi köze sem lesz a Descartes koordinátákhoz, így a program is egész más dolgokat fog művelni, mint amit elvárnak tőle.

Összefoglalva, a névkonvenciók fegyelmezett betartására kell hagyatkoznunk az egységbe zárás megvalósításakor, a belső adattagok közvetlen elérésének megakadályozását pedig igen nagy önuralommal kell magunkra erőltetnünk, mert nincs olyan nyelvi eszköz a birtokunkban amely ezt akár tűzzel-vassal is kierőszakolná.

A mintafeladatunkban egyetlen *Line* típusú objektum szerepel. Ilyen esetekben a belső adattagok elfedését (information hiding) már C-ben is megvalósíthatjuk az objektumhoz rendelt **modul** segítségével:

```
LINE.C:
   static struct Vector r, v;  //information hiding
   void RotateLine( double fi ) { ... }
   void TranslateLine( struct Vector d ) { ... }
   void SetLine( struct Vector r, struct Vector v ) { ... }

LINE.H:
   extern void RotateLine( double );
   extern TranslateLine( struct Vector );
   extern SetLine( struct Vector, struct Vector );

PROGRAM.C:
   ....
   #include "line.h"
   struct Vector r, v;
   SetLine( r, v );
   RotateLine( 0.75 );
```

Helyezzük el tehát a *Line* típusú objektum adattagjait statikusként definiálva egy külön fájlban (célszerűen LINE.C) a hozzá tartozó műveletek implementációjával együtt. Ezenkívül készítsünk egy interfész fájlt (LINE.H), amelyben a függvények prototípusát adjuk meg. A korábbiakkal ellentétben most a függvények paraméterei között nem kell szerepeltetni azt a konkrét objektumot, amellyel dolgozni akarunk, hiszen összesen egy *Line* típusú objektum van, így a választás kézenfekvő. Ha a program valamely részében hivatkozni akarunk erre a *Line* objektumra, akkor abba a fájlba a szokásos #include direktívával bele kell helyezni a prototípusokat, amelyek a Line-hoz tartozó műveletek argumentumainak és visszatérési értékének típushelyes konverzióját biztosítják. A műveleteket ezután hívhatjuk az adott fájlból. Az adattagokhoz azonban egy másik fájlból nem férhetünk hozzá közvetlenül, hiszen a statikus deklaráció csak az adattagokat definiáló fájlból történő elérést engedélyezi.

Ezen módszer, amelyet a C programozók mindenféle objektumorientált kinyilatkoztatás nélkül is igen gyakran használnak, nyilván csak akkor működik, ha az adott adattípussal csupán egyetlen változót (objektumot) kell létrehozni. Egy adattípus alapján változók definiálását **példányok készítésének** (**instantiation**) nevezzük. Ezek szerint C-ben a példányok készítése és a belső információ eltakarása kizárja egymást.

Az egységbe zárás (encapsulation) nyelvi megjelenítése C++-ban:

Miként a normál C-struktúra azt a célt szolgálja, hogy különböző típusú adatokat egyetlen egységben lehessen kezelni, az adatok és műveletek egységbe zárásához kézenfekvő megengednünk a függvények struktúrákon belüli deklarációját illetve definícióját. A Vector struktúránk ennek megfelelően így néz ki:

```
struct Vector {
 double x, y;      // adatok, állapot
 void Set( double x0, double y0 ); // interfész
 void Translate( Vector d );
 void Rotate( double );
};
```

A **tagfüggvények** - amelyeket nézőponttól függően szokás még **metódusnak** illetve **üzenetnek** is nevezni - aktivizálása egy objektumra hasonlóan történik ahhoz, ahogyan az objektum egy attribútumát érjük el:

```
Vector v;
v.Set( 3.0, 4.0 );
v.x = 6.0;   // közvetlen attribútum elérés
v.Rotate( 30.0 );
```

Vegyük észre, hogy most nincs szükség az első argumentumban a konkrét objektum feltüntetésére. Hasonlóan ahhoz, ahogy egy v vektor x mezőjét a v.x (vagy mutató esetén pv->x) szintaktika alkalmazásával érhetjük el, ha egy v vektoron pl. 30 fokos forgatást kívánunk elvégezni, akkor a v.Rotate(30) jelölést alkalmazzuk. Tehát egy művelet mindig arra az objektumra vonatkozik, amelynek tagjaként (. ill. -> operátorokkal) a műveletet aktivizáltuk.

Ezzel az egységbe zárást a struktúra általánosításával megoldottuk. Adósok vagyunk még a belső adatok közvetlen elérésének tiltásával, hiszen ezt a struktúra még nem akadályozza meg.

Ehhez először egy új fogalmat vezetünk be, az **osztályt** (**class**). Az osztály olyan általánosított struktúrának tekinthető, amely egységbe zárja az adatokat és műveleteket, és alapértelmezésben az minden tagja - függetlenül attól, hogy adatról, vagy függvényről van-e szó - az osztályon kívülről elérhetetlen. Az ilyen kívülről elérhetetlen tagokat privátnak (**private**), míg a kívülről elérhető tagokat publikusnak (**public**) nevezzük. Természetesen egy csak privát tagokat tartalmazó osztályt nem sok mindenre lehetne használni, ezért szükséges a hozzáférés szelektív engedélyezése illetve tiltása is, melyet a **public** és **private** kulcsszavakkal tehetünk meg. Ezek hatása addig tart, amíg a struktúrán belül meg nem változtatjuk egy újabb **private** vagy **public** kulcsszóval. Egy osztályon belül az értelmezés **private**-tal kezdődik. Az elmondottak szerint az adatmezőket szinte mindig **private**-ként kell deklarálni, míg a kívülről is hívható műveleteket **public**-ként.

A Vector osztály deklarációja ennek megfelelően:

```
class Vector {
// private:
   double x, y;     // adatok, állapot
public:
   void    Set( double x, double y );
   void    Translate( Vector d );
   void    Rotate( double fi );
};
```

Ezek után a következő programrészlet első két sora helyes, míg a harmadik sor fordítási hibát okoz:

```
Vector v;
v.Set( 3.0, 4.0 );
v.x = 6.0;  // FORDÍTÁSI HIBA
```

Megjegyezzük, hogy a C++-ban a struktúrában (**struct**) is lehetőség van a **public** és **private** kulcsszavak kiadására, így a hozzáférés szelektív engedélyezése ott is elvégezhető. Különbség persze az, hogy alapértelmezés szerint az osztály tagjai privát elérésűek, míg egy struktúra tagjai publikusak. Ugyan az objektumorientált programozás egyik központi eszközét, az osztályt, a struktúra általánosításával vezettük be, az azonban már olyan mértékben különbözik a kiindulástól, hogy indokolt volt új fogalmat létrehozni. A C++ elsősorban kompatibilitási okokból a struktúrát is megtartja, sőt az osztály lehetőségeivel is felruházza azt. Mégis helyesebbnek tűnik, ha a struktúráról a továbbiakban elfeledkezünk, és kizárólag az új osztály fogalommal dolgozunk.

Tagfüggvények implementációja:

Idáig az adatok és függvények egységbe zárása során a függvényeknek csupán a deklarációját (prototípusát) helyeztük el az osztály deklarációjának belsejében. A függvények törzsének (implementációjának) a megadása során két megoldás közül választhatunk: definiálhatjuk őket az osztályon belül, amikor is a tagfüggvény deklarációja és definíciója nem válik el egymástól, vagy az osztályon kívül szétválasztva a deklarációt a definíciótól.

Az alábbiakban a Vector osztályra a Set függvényt az osztályon belül, míg a Rotate függvényt az osztályon kívül definiáltuk:

```
class Vector {
   double x, y;   // adatok, állapot
public:
   void  Set( double x0, double y0 ) { x = x0; y = y0; }
   void  Rotate( double );  // csak deklaráció
};

void Vector :: Rotate( double fi ) {    // definíció
   double nx = cos(fi) * x + sin(fi) * y;  // x,y saját adat
   double ny = -sin(fi) * x + cos(fi) * y;
   x = nx; y = ny;    // vagy this -> y = ny;
}
```

A példához a következőket kell hozzáfűzni:

- A tagfüggvényeken belül a privát adattagok (és esetlegesen tagfüggvények) természetesen közvetlenül elérhetők, mégpedig a tagnév segítségével. Így például a v.Set(1, 2) függvény hívásakor, az a v objektum x és y tagját állítja be.
- Ha a tagfüggvényt az osztályon kívül definiáljuk, akkor azt is egyértelműen jelezni kell, hogy melyik osztályhoz tartozik, hiszen pl. Rotate tagfüggvénye több osztálynak is lehet. Erre a célra szolgál az ún. **scope operátor** (::), melynek segítségével, a Vector::Rotate() formában a Vector osztály Rotate tagfüggvényét jelöljük ki.
- Az osztályon belül és kívül definiált tagfüggvények között az egyetlen különbség az, hogy minden osztályon belül definiált függvény automatikusan **inline** (makro) lesz. Ennek magyarázata az, hogy áttekinthető osztálydefiníciókban úgyis csak tipikusan egysoros függvények engedhetők meg, amelyeket hatékonysági okokból makroként ajánlott deklarálni.
- Minden tagfüggvényben létezik egy nem látható paraméter, amelyre **this** elnevezéssel lehet hivatkozni. A **this** mindig az éppen aktuális objektumra mutató pointer. Így a saját adatmezők is elérhetők ezen keresztül, tehát az x helyett a függvényben this->x-t is írhatnánk.

3.3. 6.3.3. Az osztályok nyelvi megvalósítása (C++ → C fordító)

Az osztály működésének jobb megértése érdekében érdemes egy kicsit elgondolkodni azon, hogy miként valósítja meg azt a C++ fordító. Az egyszerűség kedvéért tételezzük fel, hogy egy C++-ról C nyelvre fordító programot kell írnunk (az első C++ fordítók valójában ilyenek voltak) és vizsgáljuk meg, hogy a fogalmainkat hogyan lehet leképezni a szokásos C programozási elemekre. A C nyelvben az osztályhoz legközelebbi adattípus a struktúra (*struct*), amelyben az osztály adatmezőit elhelyezhetjük, függvénymezőit viszont külön kell

választanunk és globális függvényekként kell kezelnünk. A névütközések elkerülése végett a globális függvénynevekbe bele kell kódolni azon osztály nevét, amelyhez tartozik, sőt, ha ezen függvény névhez különféle paraméterezésű függvények tartoznak (függvénynevek átdefiniálása), akkor a paraméterek típusait is. Így a Vector osztály Set függvényéből egy olyan globális függvény lesz, amelynek neve Set_Vector, illetve függvénynév átdefiniálás esetén Set_Vector_dbldbl lehet. A különválasztás során fájdalmas pont az, hogy ha például 1000 db vektor objektumunk van, akkor látszólag 1000 db különböző Set függvénynek kell léteznie, hiszen mindegyik egy kicsit különbözik a többitől, mert mindegyik más x,y változókkal dolgozik. Ha ehhez még hozzátesszük, hogy ezen vektor objektumok a program futása során akár dinamikusan keletkezhetnek és szűnhetnek meg, nyilvánvalóvá válik, hogy Set függvény objektumonkénti külön megvalósítása nem járható út. Ehelyett egyetlen Set függvénnyel kell elvégezni a feladatot, melynek ekkor nyilvánvalóan meg kell kapnia, hogy éppen melyik objektum x,y adattagjai alapján kell működnie. Ennek egyik legegyszerűbb megvalósítása az, hogy az adattagokat összefogó struktúra címét adjuk át a függvénynek, azaz minden tagfüggvény első, nem látható paramétere az adattagokat összefogó struktúra címét tartalmazó mutató lesz. Ez a mutató nem más mint az "objektum saját címe", azaz a this pointer. A this pointer alapján a lefordított program az összes objektum attribútumot indirekt módon éri el.

Tekintsük példaképpen a Vector osztály egyszerűsített megvalósítását C++-ban:

```
class Vector {
   double x, y;
public:
   void Set( double x0, double y0 ) { x = x0; y = y0; }
   void Rotate( double );
};
```

A C++→C fordítóprogram, mint említettük, az adattagokat egy struktúrával írja le, míg a tagfüggvényeket olyan, a névütközéseket kiküszöbölő elnevezésű globális függvényekké alakítja, melynek első paramétere a this pointer, és melyben minden attribútum ezen keresztül érhető el.

```
struct Vector { double x, y; };

void Set_Vector(struct Vector * this, double x0, double y0) {
   this -> x = x0;
   this -> y = y0;
}
void Rotate_Vector(Vector * this, double fi) {...}
```

A Vector osztály alapján egy Vector típusú v objektum definiálása és felhasználása a következő utasításokkal végezhető el C++-ban:

```
Vector v;
v.Set( 3.0, 4.0 );
```

Ha egy Vector típusú v objektumot létrehozunk, akkor lényegében az adattagoknak kell helyet foglalni, ami egy közönséges struktúra típusú változó definíciójával ekvivalens. Az üzenetküldést (v.Set(3.0,4.0)) viszont egy globális függvényhívássá kell átalakítani, melyben az első argumentum az üzenet célobjektumának a címe. Így a fenti sorok megvalósítása C-ben:

```
struct Vector v;
Set_Vector( &v, 3.0, 4.0 );
   // Set_Vector_dbldbl ha függvény overload is van.
```

3.4. 6.3.4. Konstruktor és destruktor

A Vector osztály alapján objektumokat (változókat) definiálhatunk, melyeket a szokásos módon értékadásban felhasználhatunk, illetve tagfüggvényeik segítségével üzeneteket küldhetünk nekik:

```
class Vector { .... };

main( ) {
Vector v1;           // definíció és inicializálás
v1.Set( 0.0, 1.0 );  // két lépésben
....
Vector v2 = v1;      // definíció másolással
....
v2.Set( 1.0, 0.0 );  // állapotváltás
v1.Translate( v2 );
....
v1 = v2;             // értékadás
....
}
```

Mivel a C++-ban objektumot bárhol definiálhatunk, ahol utasítást adhatunk meg, a változó definiálását ajánlott összekapcsolni az inicializálásával. Mint ahogy a fenti példa alapján látható, az inicializálást alapvetően két módszerrel hajthatjuk végre:

1. A definíció után egy olyan tagfüggvényt aktivizálunk, amely beállítja a belső adattagokat (v1.Set(0.0,1.0)), azaz az inicializálást egy különálló második lépésben végezzük el.

2. Az inicializálást egy másik, ugyanilyen típusú objektum átmásolásával a definícióban tesszük meg (Vector v2 = v1;).

Annak érdekében, hogy az első megoldásban se kelljen az inicializáló tagfüggvény meghívását és a definíciót egymástól elválasztani, a C++ osztályok rendelkezhetnek egy olyan speciális tagfüggvénnyel, amely akkor kerül meghívásra, amikor egy objektumot létrehozunk. Ezt a tagfüggvényt **konstruktornak** (**constructor**) nevezzük. A konstruktor neve mindig megegyezik az osztály nevével. Hasonlóképpen definiálhatunk az objektum megszűnésekor automatikusan aktivizálódó tagfüggvényt, a **destruktort** (**destructor**). A destruktor neve is az osztály nevéből képződik, melyet egy *tilde* (~) karakter előz meg.

A konstruktorral és a destruktorral felszerelt Vector osztály felépítése:

```
class Vector {
   double x, y;
public:
   Vector( double x0, double y0 ) { x = x0; y = y0; }
         // konstruktornak nincs visszatérési típusa

   ~Vector( ) { }
         // destruktornak nincs típusa sem argumentuma
};
```

A fenti megoldásban a konstruktor két paramétert vár, így amikor egy Vector típusú változót definiálunk, akkor a változó neve után a konstruktor paramétereit át kell adni. A destruktor meghívása akkor történik, amikor a változó megszűnik. A lokális változók a definíciós blokkból való kilépéskor szűnnek meg, globális változók pedig a program végén, azaz olyan helyen, ahol egyszerűen nincs mód paraméterek átadására. Ezért a destruktoroknak nem lehetnek argumentumaik. Dinamikus változók a lefoglalásuk pillanatában születnek meg és felszabadításukkor szűnnek meg, szintén konstruktor illetve destruktor hívások kíséretében.

A konstruktorral és destruktorral felszerelt Vector osztály alapján definiált objektumok használatát a következő példával világíthatjuk meg:

```
{
   Vector v1(0.0, 1.0); // konstruktor hívás
   Vector v2 = v1;
   ....
   v1 = Vector(3.0, 4.0); // értékadásig élő objektum
                  // létrehozása és v1-hez rendelése
   .... // destruktor az ideiglenesre
}                 // 2 db destruktor hívás: v1, v2
```

A v1=Vector(3.0,4.0); utasítás a konstruktor érdekes alkalmazását mutatja be. Itt a konstruktorral egy ideiglenes vektor-objektumot hozunk létre, melyet a v1 objektumnak értékül adunk. Az ideiglenes vektor-objektum ezután megszűnik.

Ha az osztálynak nincs konstruktora, akkor a fordító egy paraméter nélküli változatot automatikusan létrehoz, így azok a korábbi C++ programjaink is helyesek, melyekben nem definiáltunk konstruktort. Ha viszont bármilyen bemenetű konstruktort megadunk, akkor automatikus konstruktor nem jön létre. Az argumentumot nem váró konstruktort **alapértelmezés szerinti (default) konstruktornak** nevezzük. Az alapértelmezés szerinti konstruktort feltételező objektumdefiníció során a konstruktor üres ( ) zárójeleit nem kell kiírni, tehát a Vector v( ); definíció helyett a megszokottabb Vector v; is alkalmazható és ugyanazt jelenti.

Globális (blokkon kívül definiált) objektumok a program "betöltése" alatt, azaz a **main** függvény meghívása előtt születnek meg, így konstruktoruk is a main hívása előtt aktivizálódik. Ezekben az esetekben a konstruktor argumentuma csak konstans-kifejezés lehet és nem szabad olyan dolgokra támaszkodnunk, melyet a main inicializál.

Miként a beépített típusokból **tömböket** hozhatunk létre, ugyanúgy megtehetjük azt objektumokra is. Egy 100 db Vector objektumot tartalmazó tömb például:

```
Vector v[100];
```

Mivel ezen szintaktika szerint nem tudjuk a konstruktor argumentumait átadni, tömb csak olyan típusból hozható létre, amelyben alapértelmezésű (azaz argumentumokat nem váró) konstruktor is van, vagy egyáltalán nincs konstruktora, hiszen ekkor az alapértelmezésű konstruktor létrehozásáról a fordítóprogram gondoskodik.

Az objektumokat definiálhatjuk dinamikusan is, azaz memóriafoglalással (**allokáció)**, a new és delete operátorok segítségével. Természetesen egy dinamikusan létrehozott objektum a new operátor alkalmazásakor születik és a delete operátor aktivizálásakor vagy a program végén szűnik meg, így a konstruktor és destruktor hívások is a new illetve delete operátorhoz kapcsolódnak. A new operátorral történő memóriafoglalásnál a kért objektum típusa után kell megadni a konstruktor argumentumait is:

```
Vector * pv = new Vector(1.5, 1.5);

Vector * av = new Vector[100];  // 100 elemű tömb
```

A delete operátor egy objektumra értelemszerűen használható (delete pv), tömbök esetében viszont némi elővigyázatosságot igényel. Az előzőleg lefoglalt és az av címmel azonosított 100 elemű Vector tömbre a delete av; utasítás valóban fel fogja szabadítani mind a 100 elem által lefoglalt helyet, de csak a legelső elemre (av[0]) fogja a destruktort meghívni. Amennyiben a destruktor minden elemre történő meghívása lényeges, a delete operátort az alábbi formában kell használni:

```
delete [] av;
```

3.5. 6.3.5. A védelem szelektív enyhítése - a barát (friend) mechanizmus

Térjünk vissza a vektorokat és egyeneseket tartalmazó feladatunk mindeddig szőnyeg alá söpört problémájához, amely a párhuzamosság ellenőrzésének valamely osztályhoz rendelését fogalmazza meg. A problémát az

okozza, hogy egy tagfüggvény csak egyetlen osztályhoz tartozhat, holott a párhuzamosság ellenőrzése tulajdonképpen egyaránt tartozik a vizsgált vektor (v) és egyenes (l) objektumokhoz.

Nézzünk három megoldási javaslatot:

1. Legyen a párhuzamosság ellenőrzése (AreParallel) a Vector tagfüggvénye, melynek adjuk át az egyenest argumentumként: v.AreParallel(l). Ekkor persze a párhuzamosság-ellenőrző tagfüggvény a Line osztálytól idegen, azaz az egyenes (l) adattagjaihoz közvetlenül nem férhet hozzá, ami pedig szükséges a párhuzamosság eldöntéséhez.

2. Legyen a párhuzamosság ellenőrzése (AreParallel) a Line tagfüggvénye, melynek adjuk át a vektort argumentumként: l.AreParallel(v). Azonban ekkor a párhuzamosság ellenőrző tagfüggvény a vektor (v) adattagjaihoz nem fér hozzá.

3. A legigazságosabbnak tűnik, ha a párhuzamosság ellenőrzését egyik osztályhoz sem rendeljük hozzá, hanem globális függvényként valósítjuk meg, amely mindkét objektumot argumentumként kapja meg: AreParallel(l,v). Ez a függvény persze egyik objektum belső adattagjaihoz sem nyúlhat.

A megoldási lehetőségek közül azt, hogy az osztályok bizonyos adattagjai publikusak jobb, ha most rögtön el is vetjük. Következő ötletünk lehet, hogy a kérdéses osztályhoz ún. **lekérdező metódusokat** szerkesztünk, melyek lehetővé teszik a szükséges attribútumok kiolvasását:

```
class Line {
   double x0, y0, vx, vy;
public:
   double Get_vx() { return vx; }
   double Get_vy() { return vy; }
};
```

Végül engedélyezhetjük egy osztályhoz tartozó összes objektum privát mezőihez (adattag és tagfüggvény) való hozzáférést szelektíven egy idegen függvény, vagy akár egyszerre egy osztály minden tagfüggvénye számára.

Ezt a szelektív engedélyezést a **friend (barát) mechanizmus** teszi lehetővé, melynek alkalmazását először az AreParallel globális függvényként történő megvalósításával mutatjuk be:

```
class Line {
   double x0, y0, vx, vy;
public: ...
   friend Bool AreParallel( Line, Vector );
};

class Vector {
   double x, y;
public: ...
   friend Bool AreParallel( Line, Vector );
};

Bool AreParallel( Line l, Vector v ) {
   return ( l.vx * v.y == l.vy * v.x );
}
```

Mint látható a barátként fogadott függvényt az osztályon belül **friend** kulcsszóval kell deklarálni.

Amennyiben a párhuzamosság ellenőrzését a Vector tagfüggvényével valósítjuk meg, a Line objektum attribútumaihoz való közvetlen hozzáférést úgy is biztosíthatjuk, hogy a Line osztály magát a Vector osztályt fogadja barátjának. A hozzáférés engedélyezés ekkor a Vector összes tagfüggvényére vonatkozik:

```
class Vector;

class Line {
friend class Vector;
   double x0, y0, vx, vy;
   public: ...
};

class Vector {
   double x, y;
public: ...
   Bool AreParalell( Line l ) { return (l.vx*y == l.vy*x); }
};
```

A példa első sora ún. elődeklaráció, melyről részletesebben 7.8. fejezetben szólunk.

4. 6.4. Operátorok átdefiniálása (operator overloading)

A matematikai és programnyelvi operátorok többrétűségének (polimorfizmusának) ténye közismert, melyet már a hagyományos programozási nyelvek (C, Pascal, stb.) sem hagyhattak figyelmen kívül. A matematikában például ugyanazt a + jelet használjuk számok összeadására, mátrixok összeadására, logikai változó "vagy" kapcsolatának az előállítására, stb., pedig ezek igen különböző jellegű műveletek. Ez mégsem okoz zavart, mert megvizsgálva a + jel jobb és bal oldalán álló objektumok (azaz az operandusok) típusát, a művelet jellegét azonosítani tudjuk. Hasonló a helyzet programozási nyelvekben is. Egy + jel jelentheti két egész (int), vagy két valós (double) szám összeadását, amelyekről tudjuk, hogy a gépikód szintjén igen különböző műveletsort takarhatnak. A programozási nyelv fordítóprogramja a + operátor feldolgozása során eldönti, hogy milyen típusú operandusok vesznek részt a műveletben és a fordítást ennek megfelelően végzi el. A C++ nyelv ehhez képest még azt is lehetővé teszi, hogy a nyelv operátorait ne csak a beépített típusokra hanem az osztállyal gyártott objektumokra is alkalmazhassuk. Ezt nevezzük operátor átdefiniálásnak (overloading).

Az operátorok alkalmazása az objektumokra alapvetően azt a célt szolgálja, hogy a keletkező programkód tömör és a fogalmi modellhez a lehetőség szerint a leginkább illeszkedő legyen.

Vegyük elő a *Vector* példánkat, és miként a matematikában szokásos, jelöljük a vektorok összeadását a + jellel és az értékadást az = operátorral. Próbáljuk ezeket a konvenciókat a programon belül is megtartani:

```
Vector v, v1, v2;

v = v1 + v2;
```

Mi is történik ezen C++ sorok hatására? Mindenekelőtt a C++ fordító tudja, hogy a + jellel jelzett "összeadást" előbb kell kiértékelni, mint az = operátor által előírt értékadást, mivel az összeadásnak nagyobb a **precedenciája**. Az "összeadás" művelet pontosabb értelmezéséhez a fordító megvizsgálja az operandusok (melyek "összeadás" művelet esetén a + jel bal és jobb oldalán helyezkednek el) típusát. Jelen esetben mindkettő típusa Vector, azaz nem beépített típus, tehát a fordítónak nincs kész megoldása a művelet feldolgozására. Azt, hogy hogyan kell két, adott osztályhoz tartozó objektumot összeadni, nyilván csak az osztály készítője tudhatja. Ezért a fordító megnézi, hogy a bal oldali objektum osztályának (Vector) van-e olyan "összeadás", azaz operator+, tagfüggvénye, amellyel neki a jobb oldali objektumot el lehet küldeni (ami jelen esetben ugyancsak Vector típusú), **vagy** megvizsgálja, hogy létezik-e olyan operator+ globális függvény, amely első argumentumként az első operandust, második argumentumként a másodikat várja. Pontosabban megnézi, hogy a Vector osztálynak van-e Vector::operator+(Vector) deklarációjú metódusa, **vagy** létezik-e globális operator+(Vector,Vector) függvény (természetesen az argumentumokban Vector helyett használhatunk akár Vector& referenciát is). A kövér szedésű **vagy** szócskán az előző mondatokban "kizáró vagy" műveletet értünk, hiszen az is baj, ha egyik változatot sem találja a fordító hiszen ekkor nem tudja lefordítani a műveletet, és az is, ha mindkettő létezik, hiszen ekkor nem tud választani a két alternatíva között (többértelműség).

A v1 + v2 kifejezés a két fenti változat létezésétől függően az alábbi üzenettel illetve függvényhívással ekvivalens:

```
v1.operator+(v2); // Vector::operator+(Vector) tagfüggvény

operator+(v1, v2);// operator+(Vector,Vector) globális függv.
```

Ezután következik az értékadás (=) feldolgozása, amely jellegét tekintve megegyezik az összeadásnál megismerttel. A fordító tudja, hogy az értékadás kétoperandusú, ahol az operandusok az = jel két oldalán találhatók. A bal oldalon a v objektumot találja, ami Vector típusú, a jobb oldalon, pedig az összeadásnak megfelelő függvényhívást, amely olyan típust képvisel, ami az összeadás függvény (Vector::operator+(Vector) vagy operator+(Vector,Vector)) típusa. Ezt a visszatérési típust, az összeadás értelmezése szerint nyilván ugyancsak Vector-nak kell definiálni. Most jön annak vizsgálata, hogy a bal oldali objektumnak létezik-e olyan operator= tagfüggvénye, mellyel a jobb oldali objektumot elküldhetjük neki (Vector::operator=(Vector)). Általában még azt is meg kell vizsgálni, hogy van-e megfelelő globális függvény, de néhány operátornál - nevezetesen az értékadás =, index [], függvényhívás () és indirekt mezőválasztó -> operátoroknál - a globális függvény alkalmazása nem megengedett.

A helyettesítés célja tehát a teljes sorra:

```
v.operator=( v1.operator+( v2 ) );
```

vagy

```
v.operator=( operator+( v1, v2 ) );
```

Amennyiben a fordító az = jel feldolgozása során nem talál megfelelő függvényt, nem esik kétségbe, hiszen ez az operátor azon kevesek közé tartozik (ilyen még az & "valaminek a címe" operátor), melynek van alapértelmezése, mégpedig az adatmezők, pontosabban a memóriakép bitenkénti másolása (ezért használhattuk a Vector objektumokra az értékadást már korábban is).

Összefoglalva, ha egy osztályhoz tartozó objektumra alkalmazni szeretnénk valamilyen operátort, akkor vagy az osztályt ruházzuk fel az operator$ (a $ helyére a tényleges operátor jelét kell beírni) tagfüggvénnyel, vagy egy globális operator$ függvényt hozunk létre. Első esetben, kétváltozós műveleteknél az üzenet célja a kifejezésben szereplő baloldali objektum, az üzenet argumentuma pedig a jobb oldali operandus, illetve az üzenet argumentum nélküli egyoperandusú esetben. A globális függvény a bemeneti argumentumaiban a felsorolásnak megfelelő sorrendet tételezi fel.

Bővítsük a vektor osztály definicióját az összeadás művelettel!

4.1. 6.4.1. Operátor-átdefiniálás tagfüggvénnyel

```
class Vector {
   double x, y;
public:
   Vector(double x0, double y0) {x = x0; y = y0;}
   Vector operator+( Vector v );
};

Vector Vector :: operator+( Vector v ) {
   Vector sum( v.x + x, v.y + y );
   return sum;
}
```

A megoldás önmagáért beszél. Az összeadás függvény törzsében létrehoztunk egy ideiglenes (sum) objektumot, melynek x,y attribútumait a konstruktorának segítségével a bal (üzenet célja) és jobb (üzenet paramétere) operandusok x illetve y koordinátáinak az összegével inicializáltuk.

Itt hívjuk fel a figyelmet egy fontos, bár nem az operátor átdefiniálással kapcsolatos jelenségre, amely gyakran okoz problémát a kezdő C++ programozók számára. A Vector::operator+ tagfüggvény törzsében az argumentumként kapott Vector típusú v objektum privát adatmezőihez közvetlenül nyúltunk hozzá, ami látszólag ellentmond annak, hogy egy tagfüggvényben csak a saját attribútumokat érhetjük el közvetlenül. Valójában a közvetlen elérhetőség érvényes a "családtagokra" is, azaz minden, ugyanezen osztállyal definiált objektumra, ha maga az objektum ebben a metódusban elérhető illetve látható. Sőt az elérhetőség kiterjed még a barátokra is (friend), hiszen a friend deklarációval egy osztályból definiált összes objektumra engedélyezzük az privát mezők közvetlen elérését. Lényeges annak kihangsúlyozása, hogy ez nem ellenkezik az információrejtés

elvével, hiszen ha egy Vector osztály metódusát írjuk, akkor nyilván annak belső felépítésével tökéletesen tisztában kell lennünk.

Visszatérve az operátor átdefiniálás kérdésköréhez engedtessék meg, hogy elrettentő példaként az összeadás üzenetre egy rossz, de legalábbis félrevezető megoldást is bemutassunk:

```
Vector Vector :: operator+( Vector v ) {
       // rossz (félrevezető) megoldás:
   x += v.x;
   y += v.y;
   return *this;
}
```

A példában a v=v1+v2 -ben az összeadást helyettesítő v1.operator+(v2) a v1-t a v1+v2-nek megfelelően tölti ki és saját magát (*this) adja át az értékadáshoz, tehát a v értéke valóban v1+v2 lesz. Azonban az összeadás alatt az első operandus (v1) értéke is elromlik, ami ellenkezik a műveletek megszokott értelmezésével.

4.2. 6.4.2. Operátor-átdefiniálás globális függvénnyel

```
class Vector {
   double x, y;
public:
   Vector(double x0, double y0) {x = x0; y = y0;}
   friend Vector operator+( Vector& v1,Vector& v2);
};

Vector operator+( Vector& v1, Vector& v2 ) {
   Vector sum( v1.x + v2.x, v1.y + v2.y );
   return sum;
}
```

A globális függvénnyel történő megvalósítás is igen egyszerű, csak néhány finomságra kell felhívni a figyelmet. Az összeadást végző globális függvénynek el kell érnie valahogy a vektorok x,y koordinátáit, amit most a *friend* mechanizmussal tettünk lehetővé. Jelen megvalósításban az operator+ függvény Vector& referencia típusú változókat vár. Mint tudjuk ezek a Vector típusú objektumok helyettesítő nevei, tehát azok helyett korlátozás nélkül használhatók. Itt a referenciák használatának hatékonysági indokai lehetnek. Érték szerinti paraméterátadás esetén a Vector típusú objektum teljes attribútumkészletét (két *double* változó) másolni kell, referenciaként történő átadáskor viszont csak az objektum címét, amely hossza lényegesen kisebb.

A tagfüggvénnyel és globális függvénnyel történő megvalósítás lehetősége kritikus döntés elé állítja a programozót: mikor melyik módszert kell alkalmazni? Egy objektumorientált program működése lényegében az objektumok közötti üzenetváltásokkal valósul meg. A globális függvények kilógnak ebből a koncepcióból, ezért egy igazi objektumorientált programozó tűzzel-vassal irtja azokat. Miért van ekkor mégis lehetőség az operátor-átdefiniálás globális függvénnyel történő megvalósítására? A válasz igen egyszerű: mert vannak olyan helyzetek, amikor nincs mód a tagfüggvénnyel történő implementációra.

Tegyük fel, hogy a vektorainkra skalár-vektor szorzást is alkalmazni akarunk, azaz szeretnénk leírni a következő C++ sorokat:

```
Vector v1, v2(3.0, 4.0);

v1 = 2*v2;
```

Vizsgáljuk meg tüzetesen a 2*v2 helyettesíthetőségét. Az ismertetett elveknek megfelelően miután a fordító észreveszi, hogy a szorzás operátor két oldalán nem kizárólag beépített típusok vannak, a műveletet megpróbálja helyettesíteni a bal oldali objektumnak küldött operator* üzenettel vagy globális operator* függvényhívással. Azonban a bal oldalon most nem objektum, hanem egy beépített típusú (int) konstans (2-s szám) áll. Az int

kétségkívül nem osztály, azaz annak operator*(Vector) tagfüggvényt nyilván nem tudunk definiálni. Az egyetlen lehetőség a globális operator*(int, Vector) függvény marad.

A C++-ban szinte minden operátor átdefiniálható kivéve tagkiválasztó ".", az érvényességi kör (scope) :: operátorokat, és az egyetlen háromoperandusú operátort, a feltételes választást (? :).

Az átdefiniálás során, a fordító ismertetett működéséből közvetlenül következő szabályokat kell figyelembe vennünk:

1. A szintaxis nem változtatható meg.

2. Az egyoperandusú/kétoperandusú tulajdonság nem változtatható meg.

3. A precedencia nem változtatható meg.

Bizonyos operátorok (*,&,+,-) lehetnek unárisak és binárisak is. Ez az átdefiniálás során nem okoz problémát, mert a tagfüggvény illetve a globális függvény argumentumainak száma egyértelműen mutatja, hogy az egy-, vagy a kétváltozós operátorra gondoltunk. A kivételt az inkremens (++) és dekremens (--) operátorok képviselik, melyeket kétféle szintaktika (pre illetve post változatok), és ennek megfelelően eltérő szemantika jellemzi. A C++ implementációk ezt a problémát többféleképpen hidalják át. Leggyakrabban csak az egyik változatot hagyják átdefiniálni, vagy a post-inkremens (post-dekremens) műveletek átdefiniáló függvényeit úgy kell megírni, mintha azok egy int bemeneti argumentummal is rendelkeznének.

4.3. 6.4.3. Konverziós operátorok átdefiniálása

Az átdefiniálható operátorok külön családját képezik az ún. konverziós (**cast**) operátorok, melyekkel változókat (objektumokat) különböző típusok között alakíthatunk át. A konverziót úgy jelölhetjük ki, hogy a zárójelek közé helyezett céltípust az átalakítandó változó elé írjuk. Értékadásban, ha a két oldalon eltérő típusú objektumok állnak, illetve függvényparaméter átadáskor, ha az átadott objektum típusa a deklarációban meghatározottól eltérő, a fordítóprogram explicit konverziós operátor hiányában is megkísérli az átalakítást. Ezt hívjuk **implicit konverziónak.**

Példaként tekintsünk egy kis programot, melyben a két double koordinátát tartalmazó vektor objektumaink és MS-Windows-ban megszokott forma között átalakításokat végzünk. A MS-Windows-ban egy vektort egy long változóban tárolunk, melynek alsó 16 bitje az x koordinátát, felső 16 bitje az y koordinátát reprezentálja.

```
Vector vec( 1.0, 2.5 );
long point = 14L + (36L << 16); //(14,36) koordinátájú pont
extern f( long );
extern g( Vector );

vec = (Vector) point;     // explicit konverzió
vec = point;              // implicit konverzió
g( point );               // implicit konverzió
....
point = (long) vec;       // explicit konverzió
point = vec;              // implicit konverzió
f( vec );                 // implicit konverzió
```

A C++-ban a konverzióval kapcsolatos alapvető újdonság, hogy osztállyal gyártott objektumokra is definiálhatunk átalakító operátorokat az általános operátor-átdefiniálás szabályai, illetve a konstruktorok tulajdonságai szerint. Alapvetően két esetet kell megkülönböztetnünk. Az elsőben egy osztállyal definiált típusról konvertálunk más típusra, ami lehet más osztállyal definiált típus vagy beépített típus. A másodikban egy beépített vagy osztállyal definiált típust konvertálunk osztállyal definiált típusra. Természetesen, ha osztállyal definiált típust más osztállyal definiált típusra alakítunk át, akkor mindkét megoldás használható.

1. Konverzió osztállyal definiált típusról

A példa szerint a Vector→típus (a típus jelen esetben long) átalakítást kell megvalósítanunk, amely a Vector osztályban egy

```
operator típus( );
```

tagfüggvény elhelyezésével lehetséges. Ehhez a tagfüggvényhez nem definiálható visszatérési típus (hiszen annak úgyis éppen "típus"-nak kell lennie), és nem lehet argumentuma sem. Ennek alkalmazása a Vector→long konverzióra:

```
class Vector {
   double x, y;
public:
   operator long() {return((long)x + ((long)y<<16);}
};
```

2. Konverzió osztállyal definiált típusra

Amennyiben egy objektumot akarunk létrehozni egy más típusú változóból, olyan konstruktort kell készíteni, ami argumentumként a megadott típust fogadja el. A long→Vector átalakításhoz tehát a Vector osztályban egy long argumentumot váró konstruktor szükséges:

```
class Vector {
   double x, y;
public:
   Vector(long lo) {x = lo & 0xffff; y = lo >> 16;}
};
```

A konverziós operátorok alkalmazásánál figyelembe kell venni, hogy:

1. Automatikus konverzió során a fordító képes több lépésben konvertálni a forrás és cél között, de felhasználó által definiált konverziót csak egyszer hajlandó figyelembe venni.

2. A konverziós út nem lehet többirányú.

4.4. 6.4.4. Szabványos I/O

Végül az operátor-átdefiniálás egy jellegzetes alkalmazását, a C++ szabványos I/O könyvtárát szeretnénk bemutatni. A könyvtár deklarációit hajdan az iostream.h fájlban helyezték el, ami, követve a fájlon belüli változásokat, később iostream nevet kapott. Az iostream.h alkalmazása, még ha a fordítónk meg is engedi, ma már idejétmúlt. Általános trend, hogy a C++ könyvtárak a fejlődésüket az interfész fájlok nevében a .h kiterjesztés elhagyásával jelzik. Mivel a névtér is viszonylag újabb találmány, az is általános, hogy a .h kiterjesztésű interfész fájlok deklarációi a globális névtérben, a .h nélkülieké pedig az std névtérben vannak.

Az iostream könyvtár, miként a C++ könyvtárak, a neveit az std névtérbe teszi. Ha nem akarunk minden könyvtári névhez std:: előtagot rendelni, az using namespace std; utasítást kell egyszer kiadnunk.

A C++-ban egy *változó* beolvasása a szabványos bemenetről szemléletes módon cin >> *változó* utasítással, míg a kiíratás a szabványos kimenetre illetve hibakimenetre a cout << *változó* illetve a cerr << *változó* utasítással hajtható végre. Ezen módszer és a hagyományos C-könyvtár alkalmazását a következő példában hasonlíthatjuk össze:

```
C: könyvtári függvények:
#include <stdio.h>
main( ) {
   int i;
   printf("Hi%d\n", i);
   scanf("%d", &i);
}
```

```
Régi C++: átdefiniált operátorok
#include <iostream.h>
main( ) {
   int i;
   cout<<"Hi"<<i<<'\n';
   cin>>i;
}
```

```
Új C++: explicit névtér
#include <stdio.h>

main( ) {
   int i;
   std::cout<<"Hi"<<i<<'\n';
   std::cin>>i;
}
```

```
Új C++: rövid névtér
#include <iostream.h>
using namespace std;
main( ) {
   int i;
   cout<<"Hi"<<i<<'\n';
   cin>>i;
}
```

Amennyiben egyszerre több változót kívánunk kiírni vagy beolvasni azok láncolhatók, tehát a példában a cout<<"Hi"<<i<<'\n'; ekvivalens a következő három utasítással:

```
cout << "Hi";  cout << i;  cout << '\n';
```

A C++ megoldás mögött természetesen a << és >> operátorok átdefiniálása húzódik meg. Ha megnézzük a cin és cout objektumok típusát meghatározó istream illetve ostream osztályokat, melyek az *iostream* deklarációs fájlban találhatók, akkor (kissé leegyszerűsítve) a következőkkel találkozhatunk:

```
namespace std {
   class ostream {
   public:
      ostream& operator<<( int i );
      // lehetséges implementáció: {printf("%d",i);}
      ostream& operator<<( char * );
      ostream& operator<<( char );
   };
   extern ostream cout;

   class istream {
   public:
      istream& operator>>( int& i );
         // lehetséges implementáció: {scanf("%d",&i);}
   };
   extern istream cin;
}
```

A C++ módszer sokkal szemléletesebb és nem kell tartani a scanf-nél leselkedő veszedelemtől, hogy a & címképző operátort elfelejtjük alkalmazni, mivel az istream-ben a beolvasott változókat referenciaként kezeljük. Az új megoldás alapvető előnye abból származik, hogy a >> és << operátorok megfelelő átdefiniálása esetén ezután a nem beépített típusokat is ugyanolyan egyszerűséggel használhatjuk a szabványos I/O során, mint a beépített típusokat.

Példaképpen terjesszük ki a szabványos kimenetet a Vector osztályra is, úgy hogy egy v vektor a cout<<v utasítással kiíratható legyen. Ehhez vagy az ostream osztályt (cout objektum típusát) kell kiegészíteni operator<<(Vector v) tagfüggvénnyel, vagy egy globális operator<<(ostreams, Vector v) függvényt kell létrehozni. A tagfüggvénykénti megoldás azt igényelné, hogy egy könyvtári deklarációs fájlt megváltoztassunk, ami főbenjáró bűnnek számít, ezért kisebbik rosszként marad a globális függvénnyel történő megvalósítás:

```
   ostream& operator<<( ostream& s, Vector v ) {
      s << v.GetX() << v.GetY(); return s;
}
```

Az igazsághoz hozzátartozik, hogy létezik még egy további megoldás, de ahhoz a később tárgyalandó öröklés is szükséges. Hatékonysági okok miatt az első ostream típusú cout objektumot referenciaként vesszük át és visszatérési értékben is referenciaként adjuk vissza. A kapott cout objektum visszatérési értékként való átadására azért van szükség, hogy a megismert láncolás működjön.

5. 6.5. Dinamikus adatszerkezeteket tartalmazó osztályok

5.1. 6.5.1. Dinamikusan nyújtózkodó sztring osztály

Tekintsük a következő feladatot:

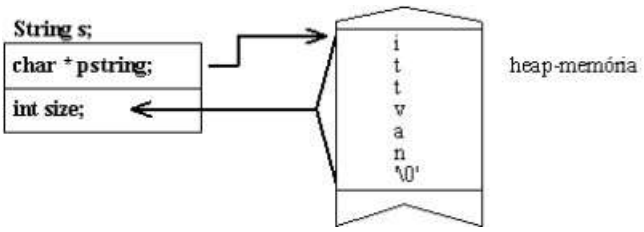
Készítsünk olyan programot, amely sztringeket képes létrehozni, valamint azokkal műveleteket végezni. Az igényelt műveletek: a sztring egyes karaktereinek írása illetve olvasása, sztringek másolása, összefűzése, összehasonlítása és nyomtatása.

A feladatspecifikáció elemzése alapján felállíthatjuk a legfontosabb objektumokat leíró táblázatot:

Objektum	**Típus**	**Attribútum**	**Felelősség**
sztring(ek)	*String*	?	karakter írás/olvasás: [ ], másolás: =, összehasonlítás: ==, !=, összefűzés: +, nyomtatás: Print, vagy "cout << ";

A műveletek azonosításánál észre kell vennünk, hogy azokat tipikusan operátorokkal célszerű reprezentálni, teret engedve az operátor átdefiniálás alkalmazásának. Az attribútumokat illetően a feladatspecifikáció nem ad semmi támpontot. Így az attribútumok meghatározásánál a tervezési döntésekre, illetve a korábbi implementációs tapasztalatainkra hagyatkozhatunk.

Az objektum belső tervezése során figyelembe kell vennünk, hogy a tárolt karakterek száma előre nem ismert és az objektum élete során is változó. Ezért kézenfekvő a sztring olyan kialakítása, mikor a karakterek tényleges helyét a dinamikus memóriából (heap) foglaljuk le, míg az attribútumok között csak az adminisztrációhoz szükséges változókat tartjuk nyilván. Ilyen adminisztrációs változó a karaktersorozat kezdőcíme a heap-en (pstring), valamint a sztring aktuális hossza (size).



6.1. ábra: A sztring memóriaképe.

Vegyük észre, hogy ha a heap-en a C sztringeknél megszokott módon a lezáró null karaktert is tároljuk, akkor a méret redundáns, hiszen azt a kezdőcím alapján a karakterek leszámlálásával mindig meghatározhatjuk.

Az ilyen redundáns attribútumokat **származtatott (derived) attribútumnak** nevezzük. Felmerül a kérdés, hogy érdemes-e a méretet külön tárolni. Gondolhatnánk, hogy nem, hiszen egyrészt feleslegesen foglalja a memóriát, másrészt ha az információt redundánsan tároljuk, akkor felmerül az inkonzisztencia veszélye. Az inkonzisztencia itt most azt jelentené, hogy a méret (size) és az első null karakterig megtalálható karakterek száma eltérő, melynek beláthatatlan hatása lehet. A másik oldalról viszont a méret nagyon gyakran szükséges változó, így ha azt mindig ki kellene számolni, akkor a sztring használata nagyon lassúvá válna. Hatékonysági okokból tehát mégis célszerűnek látszik a származtatott attribútumok fegyelmezett használata. Az

inkonzisztencia veszélye is mérsékelhető abban az esetben, amikor a redundancia egyetlen objektumon belül áll fenn.

A String osztály definíciója, egyelőre a felsorolt műveletek közül csak a karakterek írását és olvasását lehetővé tévő index operátort és az összefűzést (+ operátor) megvalósítva, az alábbiakban látható:

```
class String {
   char *  pstring;
   int     size;
public:
   String(char* s = "") { // String()=default konstruktor
      pstring = new char[ size = strlen(s) + 1 ];
      strcpy( pstring, s );
   }

   ~String( ) { delete [] pstring; }

   char& operator[] (int i) { return pstring[i]; }

   String operator+( String& s ) {
      char * psum = new char[ size + s.size - 1];
      strcpy(psum, pstring); strcat(psum, s.pstring);
      String sum( psum );
      delete psum;
      return sum;
   }
};
```

A konstruktor feladata az átadott C sztring alapján a heap-en a szükséges terület lefoglalása, a sztring odamásolása és az adminisztrációs adattagok (pstring, size) kitöltése.

A fenti konstruktordefiníció az alapértelmezésű argumentuma miatt tartalmazza az ún. alapértelmezésű, azaz bemeneti paramétert nem igénylő, konstruktort is. Az alapértelmezésű konstruktor egy egyetlen null karakterből álló sztringet hoz létre. Amikor maga az objektum megszűnik, a heap-en lefoglalt terület felszabadításáról az osztálynak gondoskodnia kell. Éppen ezt a célt szolgálja a destruktorban szereplő delete [] pstring utasítás.

Az index operátor referencia visszatérési értéke kis magyarázatot igényel. Az index operátor természetes használata alapján, ennek segítségével egy adott karaktert lekérdezhetünk, illetve megváltoztathatunk az alábbi programban látható módon:

```
main ( ) {
   char c;
   String s( "én string vagyok" );
   c = s[3];   // c=s.operator[](3); -> c=pstring[3];
   s[2]='a';   // s.operator[](2)='a'; -> pstring[2]='a';
}              // destruktor: delete [] pstring
```

A c=s[3]; az operátor átdefiniálás definíciója szerint ekvivalens a c=s.operator[](3) utasítással, ami viszont a String::operator[] törzse szerint c-hez a string[3]-t rendeli hozzá. Ezzel nincs is probléma, attól függetlenül, hogy a visszatérési érték referencia-e vagy sem.

Az s[2]='a' viszont a s.operator[](2)='a'-nak felel meg. Ez pedig egy értékadás bal oldalán önmagában álló függvényhívást jelent, valamit olyasmit, amit a C-ben nem is lehet leírni. Ez nem is véletlen, hiszen ez, nem referencia visszatérési típus esetén, visszaadná a 2. karakter "értékét" amihez teljesen értelmetlen valamit hozzárendelni. Ha viszont az index operátor referencia típusú, akkor az s.operator[](2) a pstring[2] helyettesítő neve, amely alapján annak értéke meg is változtatható. Éppen erre van szükségünk. Ebből két általánosítható tapasztalatot vonhatunk le. Ha olyan függvényt írunk, amelyet balértékként (értékadás bal oldalán) használunk, akkor annak referenciát kell visszaadnia. Másrészt, az index operátor szinte mindig referencia visszatérési típusú.

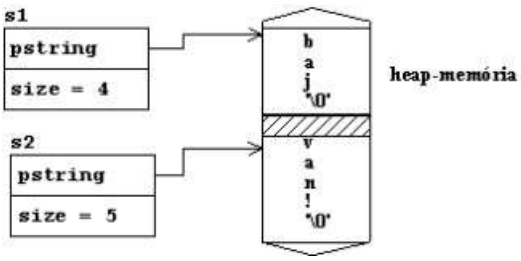
Az összeadás operátor megvalósítása során, egy ideiglenesen foglalt karakter tömbben (psum) előállítottuk az összefűzött sztringet, majd ebből konstruáltuk meg a visszatérési érték String típusú objektumát. A visszatérés előtt az ideiglenes karaktertömböt felszabadítottuk.

Még mielőtt túlságosan eluralkodna rajtunk az elkészített String osztály feletti öröm, megmutatjuk, hogy mégsem végeztünk tökéletes munkát, hiszen azzal kapcsolatban két súlyos probléma is felmerül.

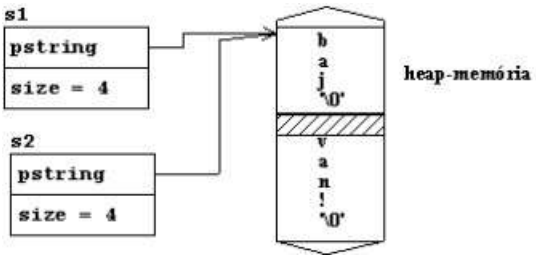
Első probléma - értékadás

Tekintsük az alábbi programrészletet, melyben az s2 sztring változónak értékül adjuk az s1-t.

```
{
   String s1( "baj" ), s2( "van!" );
```




```
....
s2 = s1;         // bit másolás
```




```
....
}   // destruktor: "baj"-t kétszer próbáljuk felszabadítani,
    // "van!"-t egyszer sem szabadítjuk fel.
```

Mint tudjuk, az értékadás operátora (=) az adattagokat egy az egyben másolja (bitmásolás). Ezért az értékadás után az s2.pstring tag ugyan oda mutat, mint az s1.pstring, és a s2.size adattag is 4 lesz. A heap-en lévő "baj" karaktersorozatra két mutató, míg a "van"-ra egyetlen egy sem hivatkozik. Ez azt jelenti, hogy ha ezek után s1 változik, az rögtön s2 változását is előidézi, ami nem elfogadható. Még súlyosabb hiba az, hogy a definíciós blokkból történő kilépéskor, mikor a destruktorok automatikusan meghívódnak, a "baj" kezdőcímére két delete parancsot hajt végre a program, amely esetleg a dinamikus memória kezelő elrepülését eredményezi. Ehhez képes eltörpül az, hogy a "van!"-t viszont senki sem szabadítja fel. A problémák nyilván abból adódnak, hogy az értékadás operátor alapértelmezés szerinti működése nem megfelelő. Olyan függvényre van szükségünk, amely a heap-en lévő tartalmat is átmásolja. A megoldás tehát az = operátor átdefiniálása.

Első kísérletünk a következő:

```
class String {
   ....
   void operator=( String& ); // nem lehet:s1=s2=s3;
};

void String :: operator=( String& s ) {
   delete pstring;
   pstring = new char[size = s.size];
   strcpy( pstring, s.pstring );
}
```

Ennek a megoldásnak két szépséghibája van. Egyrészt nem engedi meg az s1=s2=s3 jellegű többszörös értékadást, hiszen ez ekvivalens lenne a következő utasítással:

```
s1.operator= ( s2.operator= ( s1 ) );
```

ahol az s1.operator= bemeneti argumentuma az s2.operator=(s1) visszatérési értéke, ami viszont void, holott String&-nek kellene lennie. Megoldásként az operator= -nek vissza kell adnia az eredmény objektumot, vagy annak referenciáját. Hatékonysági indokból az utóbbit választjuk.

A másik gond az, hogy az értékadás tévesen működik, az s = s; szintaktikailag helyes, gyakorlatilag az üres (NOP) művelettel ekvivalens utasításra. Ekkor ugyanis az s1.operator= referenciaként megkapja s-t, azaz önmagát, amelyhez tartozó heap-területet felszabadítja, majd innen másol. Kizárhatjuk ezt az esetet, ha megvizsgáljuk, hogy a referenciaként átadott objektum címe megegyezik-e a célobjektuméval és egyezés esetén valóban nem teszünk semmit. (Megoldást jelenthet az is, ha nem referenciaként adjuk át az argumentumot, de ez új problémákat vet fel, amiről csak később lesz szó).

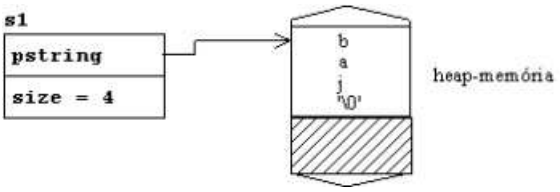
A javított változatunk az alábbiakban látható:

```
class String {
   ....
   String& operator=( String& ); // s1=s2=s3; megengedett
};

String& String :: operator=( String& s ) {
   if ( this != &s ) {  // s = s; miatt
      delete pstring;
      pstring = new char[size = s.size];
      strcpy( pstring, s.pstring );
   }
   return *this;   // visszaadja saját magát
}
```

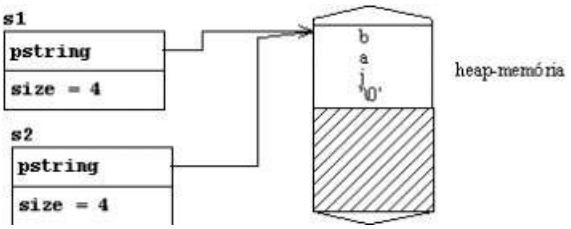
Második probléma - inicializálás

Az iménti megoldással az értékadással kapcsolatos problémákat, egyetlen kivétellel megoldottuk. A kivétel akkor jelentkezik, amikor az = jelet valamely objektumpéldány kezdőértékkel történő ellátására (inicializálására) használjuk:

```
{
   String s1( "baj" );
```




```
....
String s2 = s1;  // definíció másolással -
                 // alapértelmezés: bitenkénti másolás
```




```
}       // destruktor: "baj"-t kétszer próbálja felszabadítani
```

Nagyon fontos kihangsúlyoznunk, hogy az itt szereplő = jel nem az értékadás, hanem az inicializálás műveletét jelöli, így az előbb átdefiniált értékadó operátor az inicializálás műveletét nem változtatja meg. Így az inicializálás alapértelmezése, a bitenkénti másolás marad érvényben. Az értékadás és inicializálás éles megkülönböztetése talán szőrszálhasogatásnak tűnhet, de ennek szükségességét beláthatjuk a következő gondolatkísérlettel.

A tényleges helyzettel szemben tegyük fel, hogy a C++ nem tesz különbséget az értékadás és inicializálás között és az operator= minkét esetben felüldefiniálja a műveletet. A String s2 = s1; utasításból kialakuló s2.operator=(s1) függvényhívás azzal indulna, hogy a delete s2.pstring utasítást hajtaná végre, ami súlyos hiba, hiszen az s2.pstring még nincs inicializálva, azaz tartalma véletlenszerű. Az operator= függvényben nem tudjuk eldönteni, hogy volt-e már inicializálva az objektum vagy sem. Gondolatkísérletünk kudarcot vallott. Inicializálatlan objektumokra tehát más "értékadás" műveletet, ún. inicializálást kell alkalmazni.

Még egy pillanatra visszatérve a programrészletünkhöz, a String s2=s1; utasítás tehát helyesen nem hívja meg az operator= függvényt, viszont helytelenül az inicializálás alapértelmezése szerint az s1 adattagjait az s2 mezőibe másolja. A hatás az előzőhöz hasonló, a heap-en a "baj" karaktersorozatra két sztringből is mutatunk, és azt a destruktorok kétszer is megkísérlik felszabadítani.

A megoldáshoz az "inicializáló operátort", az ún. **másoló konstruktort** (**copy constructor**) kell átdefiniálni. A másoló konstruktor, mint minden konstruktor az osztály nevét kapja. A másoló jelleget az jelenti, hogy ugyanolyan típusú objektummal kívánunk inicializálni, ezért a másoló konstruktor argumentuma is ezen osztály referenciája. (Figyelem, a másoló konstruktor argumentuma kötelezően referencia típusú, melyet csak később tudunk megindokolni). Tehát egy **X** osztályra a másoló konstruktor: **X(X&).** A String osztály kiterjesztése másoló konstruktorral:

```
class String {
....
   String( String& s );
};
String :: String( String& s ) {
   pstring = new char[size = s.size];
   strcpy( pstring, s.pstring );
}
```

5.2. 6.5.2. A másoló konstruktor meghívásának szabályai

A C++ nyelv definíciója szerint a másoló konstruktor akkor fut le, ha inicializálatlan objektumhoz (változóhoz) értéket rendelünk. Ennek az általános esetnek három alesetét érdemes megkülönböztetni:

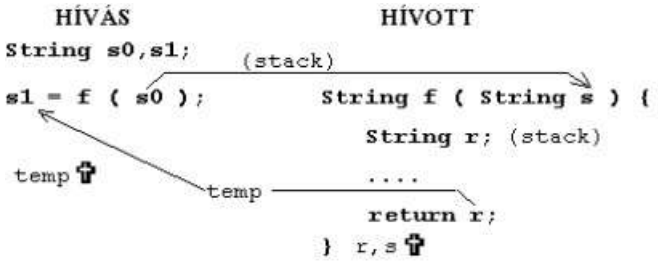
1. *Definícióban történő inicializálás*

```
String s1 = s2;
```

Ezt az esetet az előbb részletesen vizsgáltuk.

2. *Függvény argumentum és visszatérési érték*

Tudjuk, hogy érték (azaz nem referencia) szerinti paraméterátadáskor az argumentumok a verem memóriára kerülnek és a hívott függvény ezeken a másolatokon dolgozik. A verem memórián tartózkodó ideiglenes változóba másolás nyilván megfelel az inicializálatlan objektum kitöltésének, ezért ha objektumokat érték szerint adunk át, a verembe másolás során a másoló konstruktor végzi el a járulékos másolási feladatokat. Tehát az alábbi példában az s0 vermen lévő másolatát, melyet az ábrán s-sel jelöltünk, a másoló konstruktor inicializálja.



6.2. ábra: Paraméter és eredmény átadása a hívó és hívott függvény között.

Kicsit hasonló az eset a visszatérési érték átadásánál is. Skalár visszatérési érték esetén egy kijelölt regisztert használnak erre a célra (nem véletlen, hogy a Kernighan-Ritchie C még csak ilyen eseteket engedett meg). Az objektumok mérete sokszorosa lehet a regiszterek kapacitásának, így gyakran a globális memória-területet, a heap-en vagy a hívó által a veremmemóriában lefoglalt területet kell igénybe venni. A visszatérési értéket átadó regiszter- vagy memóriaterület (ábrán temp) azonban inicializálatlan, tehát a return utasítás csak a másoló konstruktor segítségével írhatja bele a visszatérési értéket.

A paraméterek és a visszatérési érték ideiglenes tárolására konstruált objektumokat előbb-utóbb föl is kell szabadítani. A s esetében ez a függvényből történő visszatérés pillanata, míg a temp esetében a visszatérési érték felhasználása után történhet meg. A felszabadítás értelemszerűen destruktorhívásokkal jár.

Az elmondottak szerint az objektumok érték szerinti paraméter átadása és visszatérési értékként történő átvétele másoló konstruktor és destruktor függvénypárok hívását eredményezi, hiszen csak így biztosítható a helyes működés. Ez a mechanizmus viszont a program végrehajtásának sebességét jelentősen lelassíthatja. Mit tehetünk a sebesség fokozásának érdekében? A paraméter átadáskor használhatunk referencia szerinti paraméterátadást, vagy átadhatjuk a C-ben megszokott módon az objektum címét. Fizikailag mindkét esetben a veremre az objektum címe kerül, a másoló konstruktor és a destruktor hívása pedig nyilván nem történik meg. Természetesen ekkor a cím szerinti paraméterátadás szabályai érvényesek, azaz ha az objektumot a függvényen belül megváltoztatjuk, az a függvényt hívó programrészlet felé is érvényesíti hatását.

Kicsit bonyolultabb a helyzet a visszatérési érték kezelésének optimalizálásánál. A fenti példa kapcsán első (de előre bocsátva, hogy rossz) ötletünk lehet, hogy adjuk vissza az r objektum címét. Ez egy klasszikus hiba, melyet a C nyelv használata során is el lehet követni. Az r objektum ugyanis az f függvény lokális változója, amely addig él, amíg az f függvényt végrehajtjuk. Tulajdonképpen a vermen foglal neki helyet néhány gépi utasítás az f függvény belépési pontján. Újabb utasítások a veremterületet a kilépéskor felszabadják. Tehát, ha ennek az ideiglenes területnek a címét adjuk vissza, f-ből visszatérve ez a cím a verem szabad területére fog mutatni. Elég egyetlen villámcsapásként érkező megszakításkérés, hogy ezt a területet felülírjuk más

információval, így az átadott mutatóval tipikusan csak "szemetet" tudunk elérni. Tanulság, hogy lokális, nem statikus változók (objektumok) címét sohase szolgáltassuk ki a függvényen kívülre!

A következő (már jobb) ötletünk a visszatérési érték referenciaként történő átadása. A C++ fordítók ugyanis általában hibaüzenettel figyelmeztetnek arra, hogy egy lokális változó referenciáját akarjuk kiszolgáltatni a függvényből. Egyes okosabb C++ fordítók viszont még intelligensebben oldják meg ezt a problémát. Ha egy referencia visszatérésű függvényben egy lokális objektumváltozót a return utasítással adunk vissza, akkor a változót nem a vermen, hanem például a heap-en hozzák létre. A heap-en foglalt változó referenciáját minden további nélkül visszaadhatjuk a függvényből, majd az esetleges értékadás után ezt az objektumot a fordító által hozzáadott destruktorhívás felszabadítja. Igen ám, de egy ideiglenes változónak a heap-en történő létrehozása sokkal munkaigényesebb, mintha azt a vermen tennénk meg. Így adott esetben az ilyen jellegű referencia visszatérési érték átadása akár lassabb is lehet, mint a közönséges érték szerinti megoldás.

Az ettől eltérő esetekben, mint például, ha egy globális változót, egy referenciaként kapott objektumot, illetve ha az objektumnak attribútumát vagy saját magát (*this) kell visszaadnia, akkor a referencia visszatérési érték mindenképpen hatékonyabb.

3. *Összetett algebrai kifejezések kiértékelése*

```
String s, s0, s1, s2;
s = s0 + s1 + s2;       temp1 <- s0 + s1; // másolás
                        temp2 <- temp1 + s2;
                        s = temp2;
                        temp1, temp2
```

Az operátor átdefiniálás ismertetésénél a műveleteket normál értelmezésüknek megfelelően egy- illetve kétoperandusúnak tekintettük. Ezen egy- és kétoperandusú műveletek azonban alapját képezhetik összetettebb kifejezéseknek is. Például a fenti esetben az s sztringet három sztring összefűzésével kell kiszámítani. Mivel olyan + operátorunk nincs, amely egyszerre három sztringet fogadna, a fenti sorban, a C balról-jobbra szabályának megfelelően, a program először az s0 és s1-t fűzi össze, majd az eredményhez fűzi az s2-t, végül azt rendeli az s-hez. A művelet során két részeredmény tárolásáról kell gondoskodni (temp1 és temp2). Ezeknek valahol (heap-en) helyet kell foglalni és oda az előző művelet eredményét be kell másolni, majd felhasználásuk után a területet fel kell szabadítani. A másolás a másoló konstruktort, a felszabadítás a destruktort veszi igénybe.

5.3. 6.5.3. Egy rejtvény

A fentiekből következik, hogy néha igen nehéz megmondani, hogy egy adott C++ utasítás milyen tagfüggvényeket és milyen sorrendben aktivizál. Ezek jó rejtvényfeladatok, illetve vizsgakérdések lehetnek.

Álljon itt a String osztály definíciója és a következő programrészlet! Kérjük a kedves olvasót, hogy sorrend helyesen írja a C++ sorok mellé azon tagfüggvények sorszámát, amelyek véleménye szerint aktivizálódnak, mielőtt tovább olvasná a fejezetet.

```
class String {
      char * pstring;
      int    size;
   public:
      String( );                    // 1
      String( char * );             // 2
      String( String& );            // 3
      ~String( );                   // 4
      String  operator+( String& ); // 5
      char&   operator[]( int );    // 6
      void    operator=( String );  // 7
   };

   main( ) {
      String s1("negyedik"); // vagy s1 = "negyedik";
      String s2;
```

```
      String s3 = s2;
      char c = s3[3];
      s2 = s3;
      s2 = s3 + s2 + s1;
   }
```

Ha a kedves olvasó nem oldotta meg a feladatot akkor már mindegy. Ha mégis, összehasonlításképpen a mi megoldásunk:

```
main( ) {
   String s1("negyedik");   2
   String s2;               1
   String s3 = s2;          3
   char c = s3[3];          6
   s2 = s3;                 3,7,4
   s2 = s3 + s2 + s1;       5,2,3,4,5,2,3,4,(3),7,4,4,(4)
}                           4,4,4
```

Ha netalántán eltérés mutatkozna, amit egy számítógépes futtatás is alátámaszt, annak számtalan oka lehet. Például a fordítónak joga van késleltetni az ideiglenes objektumok destruktorainak a hívását ameddig azt lehet. Másrészt ha a tagfüggvények törzsét az osztályon belül definiáljuk azok implicit inline (tehát makro) típusúak lesznek, azaz paraméter átadásra a vermen nincs szükség, ami néhány másoló konstruktor és destruktor pár kiküszöbölését jelentheti. Az 5,2,3,4-es sorozatokban a 2,4 konstruktor-destruktor pár az összefűzés művelet (+ operátor) korábbi implementációjából adódik. Végül ejtsünk szót a zárójelbe tett, valójában nem létező másoló konstruktor-destruktor párról. A C++ definíció szerint, ha egy ideiglenes objektumot másolunk ideiglenesbe (a példában az összeadás eredményét temp=s1+s2+s3-t a verem tetejére, mert az értékadás operátor tagfüggvénye érték szerinti paraméterátadást használ) akkor a nyilvánvalóan szükségtelen másoló konstruktor hívása (a destruktor pár hívásával együtt) elmaradhat.

5.4. 6.5.4. Tanulságok

A *String* osztály jelentősége messze túlmutat azon, ami első pillantásra látszik, ezért engedjük meg magunknak, hogy babérjainkon megpihenve megvizsgáljuk az általánosítható tapasztalatokat.

A létrehozott *String* osztály legfontosabb tulajdonságai az alábbiak:

- Dinamikusan nyújtózkodó szerkezet, amely minden pillanatban csak annyi helyet foglal el amennyi feltétlenül szükséges.
- A *String* típusú objektumok tetszőleges számban előállíthatók. Használhatjuk őket akár lokális akár globális sőt allokált változóként. A *String* típusú objektumok érték szerinti paraméterként függvénynek átadhatók, másolhatók, és egyáltalán olyan egyszerűséggel használhatók mint akár egy beépített (pl. *int*) típusú változó.

A *String*-hez hasonlatosan létrehozhatók olyan igen hasznos típusok mint pl. a tetszőleges hosszúságú egész, amely vagy binárisan vagy binárisan kódolt decimális (BCD) formában ábrázolt és az értékének megfelelően dinamikusan változtatja az általa lefoglalt memória területet, vagy tetszőleges pontosságú racionális szám, amely segítségével bátran szállhatunk szembe a kerekítési hibák okozta problémákkal. Ezen új típusok által definiált objektumok, az operátorok és konstruktorok definiálása után, ugyanúgy használhatók mint akár egy *int* vagy *double* változó.

Talán még ennél is fontosabb annak felismerése, hogy a *String* lényegében egy dinamikusan nyújtózkodó tömb, amely alapján nemcsak karakterek tárolására hozható létre hasonló. A fix méretű tömbök használata a szokásos programozási nyelvek (C, Pascal, Fortran, stb.) időzített bombája. Ki ne töltött volna már napokat azon töprengve, hogy milyen méretűre kell megválasztani programjának tömbjeit, illetve olyan hibát keresve, amely egy korlátosnak hitt tömbindex elkalandozásának volt köszönhető. És akkor még nem is beszéltünk a megoldható feladatok felesleges, a hasraütésszerűen megállapított tömbméretekből adódó, korlátozásáról. Ennek egyszer és mindenkorra vége. C++-ban egyszer kell megalkotni a dinamikus és index-ellenőrzött tömb implementációját, melyet aztán ugyanolyan egyszerűséggel használhatunk, mint a szokásos tömbünket. Sőt

szemben a C ismert hiányosságával miszerint tömböket csak cím szerint lehet függvényeknek átadni, a dinamikus tömbjeink akár cím (referencia), akár érték szerint is átadhatók, ráadásul a tömböket értékadásban vagy aritmetikai műveletekben egyetlen egységként kezelhetjük.

A lelkendezés közepette, az olvasóban valószínűleg két ellenvetés is felmerül az elmondottakkal kapcsolatban:

1. Ezt a módszert nehéz megérteni, és valószínűleg használni, hiszen az embernek már ahhoz is rejtvényfejtői bravúrra van szüksége, hogy megállapítsa, hogy mikor melyik tagfüggvény hívódik meg.

2. A szüntelen memóriafoglalás és felszabadítás minden bizonnyal borzasztóan lassú programot eredményez, így a szokásos C implementáció, a fenti hátrányok ellenére is vonzóbb alternatívának tűnik.

Mindkét ellenvetés köré gyülekező felhőket el kell oszlatnunk. A módszer nehézségével kapcsolatban annyit ugyan el kell ismerni, hogy talán első olvasáskor próbára teszi az olvasó türelmét, de ezen akadály sikeres leküzdése után érthető és könnyű lesz. Minden dinamikusan allokált tagot tartalmazó objektumot ugyanis ebben a tekintetben hasonlóan kell megvalósítani, függetlenül attól, hogy sztringről, láncolt listáról vagy akár hash táblákat tartalmazó B+ fák prioritásos soráról van-e szó. Nevezetesen, mindenekelőtt a konstruktor-destruktor párt kell helyesen kialakítani. Amennyiben az objektumokat másolással akarjuk inicializálni, vagy függvényben érték szerinti argumentumként vagy (nem referenciakénti) visszatérési értékként kívánjuk felhasználni, akkor a másoló konstruktor megvalósítása is szükséges. Végül, ha értékadásban is szerepeltetni fogjuk, akkor az = operátort is implementálni kell. Itt mindjárt meg kell említeni, hogy ha lustaságból nem valósítunk meg minden szükséges tagfüggvényt, az a későbbi változtatások során könnyen megbosszulhatja magát. Ezért, ha módunkban áll, mindig a legszélesebb körű felhasználásra készüljünk fel, illetve, ha az ennek megfelelő munkát mindenképpen el akarjuk kerülni, legalább a ténylegesen nem implementált, de az alapértelmezésű működés szerint nem megfelelő függvényeket (konstruktor, másoló konstruktor, destruktor, = operátor) valósítsuk meg oly módon, hogy például a "sztring osztály másoló konstruktora nincs implementálva" hibaüzenetet jelenítse meg. Így, ha a programunk elszállása előtt még utolsó leheletével egy ilyen üzenetet írt ki a képernyőre, akkor legalább pontos támpontunk van a hiba okát illetően.

A nehézségre visszatérve megemlíthetjük, hogy a fenti sztringhez hasonló dinamikus szerkezeteket C-ben is létrehozhatunk, de ott a felmerülő problémákkal, mint a memória újrafoglalás, minden egyes változóra minden olyan helyen meg kell küzdeni, ahol az valamilyen műveletben szerepel. Ezzel szemben a C++ megoldás ezen nehézségeket egyszerre az összes objektumra, egy kiemelt helyen, a *String* osztály definíciójában győzi le. Sőt, ha a *String* osztályt egy külön fájlban helyezzük el, és könyvtárnak tekintjük, ettől kezdve munkánk gyümölcsét minden további programunkban közvetlenül élvezhetjük.

A sebességi ellenvetésekre válaszolva védekezésül felhozhatjuk, hogy a gyakori memória foglalás-felszabadítás annak következménye, hogy a feladatot a lehető legegyszerűbb algoritmussal valósítottuk meg. Amennyiben szándékosan kicsit nagyobb memóriaterületet engedélyezünk a *String* számára mint amennyi feltétlenül szükséges, és az újrafoglalást (reallokáció) csak akkor végezzük el, ha az ezen sem fér el, akkor a redundáns memóriaterület méretével a futási időre és memóriára történő optimalizálás eltérő szempontjai között rugalmasan választhatunk kompromisszumot. Természetesen az ilyen trükkök bonyolítják a String osztály megvalósítását, de azon úgy is csupán egyszer kell átesni.

Ezen érveket, egy sokkal súlyosabbal is kiegészíthetjük. Ismert tény, hogy az utasítások, mint a tömb indexhatár ellenőrzés lespórolása, utasítások kézi "kioptimalizálása", egyéb "bit-babráló" megoldások csupán lineáris sebességnövekedést eredményezhetnek. Ennél lényegesen jobbat lehet elérni az algoritmusok és adatstruktúrák helyes megválasztásával. Például a rendezés naiv megoldása a rendezendő elemek számával (n) négyzetes ideig ($O(n^2)$ tart, míg a bináris fa alapú, vagy az összefésülő technikák ezt kevesebb, $O(n \log n)$ idő alatt képesek elvégezni. Ha az elemek száma kellően nagy (és nyilván csak ekkor válik fontossá a futási idő), az utóbbi bármilyen lineáris faktor esetén legyőzi a négyzetes algoritmust. Miért találkozunk akkor viszonylag ritkán ilyen optimális komplexitású algoritmusokkal a programjaikban? Ezek az algoritmusok tipikusan "egzotikus" adatstruktúrákat igényelnek, melyek kezelése a szokásos programozási nyelvekben nem kis fejfájást okoz. Így, ha a programozót erőnek-erejével nem kényszerítik ezek használatára, akkor hajlamos róluk lemondani. Ezzel szemben C++-ban, még legrosszabb esetben is, az ilyen "egzotikus" adatstruktúrákkal csak egyetlen egyszer kell megbirkózni teljes programozói pályafutásunk alatt, optimális esetben pedig készen felhasználhatjuk programozótársunk munkáját, vagy a könnyen hozzáférhető osztálykönyvtárakat. Még a legelvetemültebb programozók által használt alapadatstruktúrák száma sem haladja meg a néhányszor tízet, így nagy valószínűséggel megtaláljuk azt amit éppen keresünk. Végeredményképpen a C++ nyelven megírt programoktól elvárhatjuk a hatékony algoritmusok és adatstruktúrák sokkal erőteljesebb térhódítását, ami azt jelentheti, hogy a

C++-ban megírt program, nemhogy nem lesz lassabb, mint a szokásos, veszélyes C megoldás, de jelentősen túl szárnyalhatja azt sebességben is.

6. 6.6. Első mintafeladat: Telefonközponti hívásátirányító rendszer

A következőkben aprópénzre váltjuk a C++-ról megszerzett ismereteinket, és egy konkrét feladat kapcsán összekapcsoljuk azokat az objektumorientált analízis és tervezés módszertanával. A feladatot, amely egy telefonközpont hívásátirányító rendszerének leegyszerűsített modellje, megoldásának teljes folyamatában vizsgáljuk, az objektum orientált analízistől kezdve az implementációs részletekig.

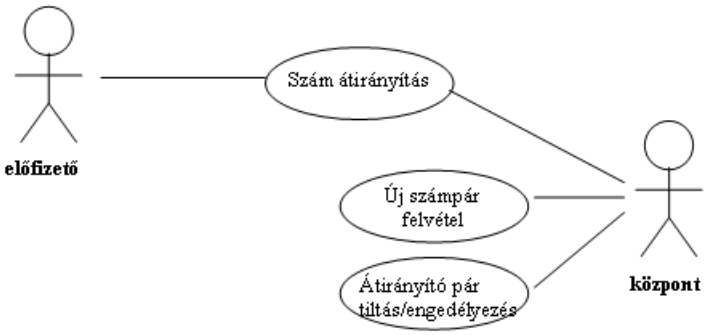
A feladat informális specifikációja a következő:

A hívásátirányító rendszer az előfizetőtől kapott számról eldönti, hogy át kell-e irányítani és megkeresi a hozzá tartozó cél számot. Ha az is át lenne irányítva, akkor természetesen a végső célt jelentő számot határozza meg. A telefonközpontból lehetőség van új átirányítást definiáló számpárok felvételére, meglévők ideiglenes tiltására illetve engedélyezésére. A számok ASCII számjegyek formájában érkeznek és hosszukra semmilyen korlát nem létezik. A rendszer maximum 100 átirányítást képes egyszerre kezelni.

A szöveg elemzése alapján fel kell ismernünk az aktorokat, a program használati eseteit, majd az alapvető fogalmi objektumokat, a legfontosabb attribútumaikat és a viselkedésüket, illetve a felelősségüket meghatározó műveleteket.

Kezdjük a használati esetek leírásával. A programmal két fajta felhasználó lép kapcsolatba. A **telefonközpont** (center), aki a következő akciókat kezdeményezheti:

- új átirányítást definiáló számpár felvétele
- átirányító pár tiltása/engedélyezése



6.3. ábra: A telefonközpont használati eset diagramja.

Az egyes használati eseteket forgatókönyvekkel írjuk le:

szám-transzformáció (*Reroute*):

1. Az előfizető egy *számot* (*number*) ad a programnak.
2. A program a számot ráilleszti az *átirányító párok tároló* (*pairs*) elemeire és egyezés, valamint a belső státuszának aktív volta esetén a elvégzi cserét, majd rekurzív módon addig ismételi a szám transzformációt amíg további átirányítás már nem lehetséges.
3. Az új számot (*new_number*) a program a központba (*center*) továbbítja.

átirányítás-felvétel (*Add*):

1. A központ az *átirányító párt* (*new_pair*) átadja a programnak.
2. A program a párt eltárolja.

átirányítás-engedélyezés (*SwitchPair*):

1. Az központ az átirányító forrás számot és az állapotot (engedély/tiltás) átadja a programnak.

2. A program megkeresi az első, az engedélyezésben szereplő forrásszámnak megfelelő párt, és az állapotát (*status*) átállítja.

A statikus modell létrehozásához az informális leíráshoz nyúlunk. Az objektumok és attribútumok a szöveges leírásban általában főnévként jelennek meg. Első közelítésben az attribútumokat az objektumoktól a rajtuk végzett műveletek és más objektumokkal fennálló viszonyuk bonyolultsága alapján különböztethetjük meg. A fenti leírásban objektumnak tekinthetjük a *kapott-* és a *célszámokat*, a *számpárokat*, magát a *hívásátirányító rendszert* és a hozzá kapcsolódó *előfizetőket* és a *telefonközpontot*. Az attribútumokkal kapcsolatban egyenlőre csak azt mondhatjuk, hogy a számok *számjegyekből* állnak, és a számpárok pillanatnyi engedélyezettségét a számpárokon belüli státusz attribútumban tartjuk nyilván.

A felelősség szétosztása során a *számpárok felvételét*, az *átirányítást* és az egyes *párok tiltását és engedélyezését* nyilván a hívásátirányító rendszerhez kell kapcsolnunk. Az átirányítás folyamatának pontosításából következik, hogy szükség van a számpárok *összehasonlítására* és *megváltoztatására*. A pár műveletei, azaz absztrakt definíciója már körmönfontabb gondolatokat is igényel. Az átirányítási folyamat során végig kell néznünk az egyes párokat és a kapott számot össze kell hasonlítani a pár megfelelő elemével, és egyezés során a számot a másik elemmel kell kicserélni. Ebben a műveletsorban a hívásátirányító rendszer objektum, a pár és a szám objektumok vesznek rész. Kérdés az, hogy a párok végigvétele, a számok összehasonlítása és cseréje leginkább melyik objektumhoz ragasztható. Ha minden műveletet a hívásátirányító közvetlen felelősségi körébe sorolnánk, akkor annak pontosan tudnia kellene, hogy a párokat hogyan (milyen adatszerkezetben) tároljuk, ismernie kell azt a tényt, hogy a páron belül milyen két objektum hordozza a forrás- és célszámokat. Ez utóbbi mindenképpen elítélendő, hiszen ekkor a párok kiszolgáltatják saját belső világukat a használójuknak. A másik végletet az képviseli, ha minden felelősséget a párra ruházunk, azaz egy pár elvégzi az összehasonlítást és egyezéskor a cserét és szükség szerint továbbadja ezt a felelősséget a következő tárolt párnak. Ez azt jeleni, hogy egyetlen pár nemcsak a két számot hordozza magában, hanem a párok tárolási rendszerére vonatkozó információt is. (Az olvasóra bízzuk azon eset elemzését, amikor mindent maga a szám végezne el.)

Most a két szélső eset között egy kompromisszumos megoldást fogunk követni. A párok végigvételét a hívásátirányító objektumra bízzuk, míg a számok összehasonlítását és cseréjét a párobjektumokon belül fogjuk elvégezni.

A kapott objektumokat, az objektumokat definiáló osztályokat, az objektumok attribútumait és felelősségeit egy táblázatba gyűjtöttük össze. Tekintve, hogy a folyamat végén egy programot szeretnénk látni, amelyben az ékezetes magyar elnevezések meglehetősen groteszk képet mutatnának, ezt a lépést felhasználtuk arra is, hogy a legfontosabb fogalmakhoz egy rövid angol megfelelőt rendeljünk, amelyekkel azokat a továbbiakban azonosítjuk.

Objektum	**Típus**	**Attrib.**	**Felelősség**
Telefonközpont hívásátirányító rendszer= *router*	*Router*	?	számpár felvétel: *Add*, átirányítás: *ReRoute* tiltás, engedélyezés: *SwitchPair*
kapott szám=*from_num*, cél szám = *to_num*	*Number*	szám-jegyek	összehasonlítás:= = , átírás: =
számpár = *pair*	*Pair*	státusz = status	tiltás, engedélyezés: *TrySwitch* transzformáció:

			TryTransform
előfizető = *subscriber*	*Subscriber*		
központ = *center*	*Center*		

Az objektumok jellegük szerint lehetnek **aktívak**, azaz olyanok, melyek önhatalmúlag elvileg bármikor kezdeményezhetnek üzenetet, vagy **passzívak**, amelyek csak annak hatására küldhetnek üzenetet, hogy mástól kaptak ilyet. A példánkban aktív objektumok a subscriber és a center, a többi objektum passzív.

Más szempontú csoportosítás, hogy a végleges rendszernek egy objektum részét képezi-e (**belső objektum**), vagy azon kívül áll és elsősorban az interfészek leírása miatt van rá szükség (**külső objektum**). A külső objektumok: subscriber, center. (Az aktív-passzív besorolással való egyezés a véletlen műve.)

A második lépés a specifikáció elemzése alapján az összetett szerkezetek felismerése és ennek felhasználásával az objektumok közötti tartalmazási kapcsolatok és asszociációk azonosítása.

Ezek első közelítésben:

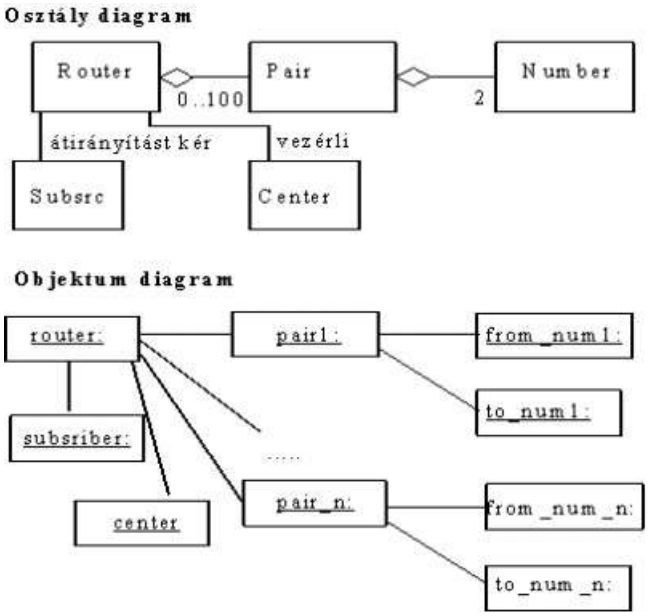
- a *router* átirányítja az *subscriber*-től kapott számot
- a *center* vezérli a *router*-t, azaz:

  a *center* új számpárokat ad a *router*-nek

  a *center* engedélyez/tilt számpárokat a *router*-ben
- a *router* számpárokat tart nyilván (maximum 100-t)
- egy *pair* (számpár) két számot tartalmaz.

Az utolsó kettő nyilván tartalmazási reláció.

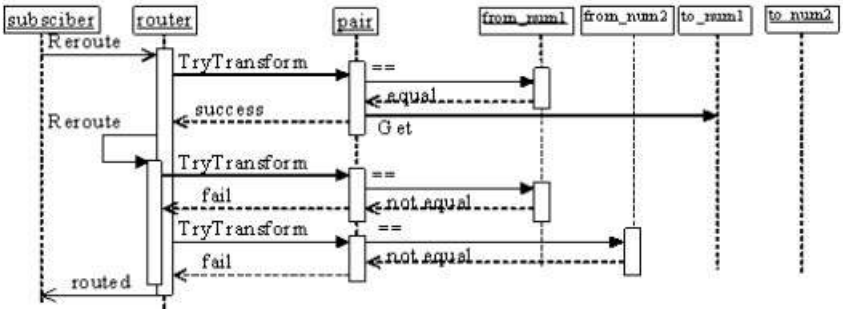
Összefoglalva a rendszer objektum modellje:



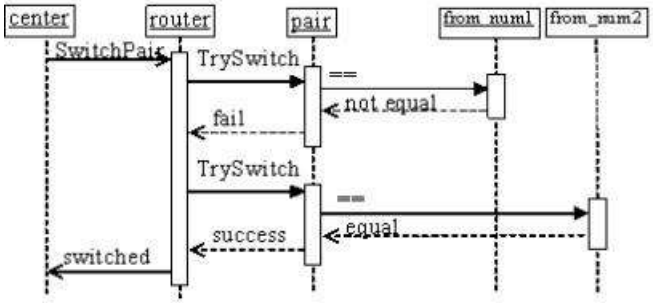
6.4. ábra: A hivásátirányító rendszer osztály és objektum modellje.

A **dinamikus modell** felállításához először a legfontosabb tevékenységekhez tartozó forgatókönyveket tekintjük át. A forgatókönyvek szokásos grafikus megjelenítésében a vízszintes tengely mentén az objektumokat

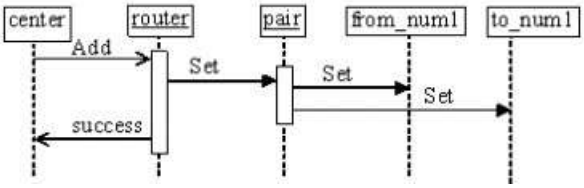
ábrázoljuk. Ha egy objektumból több is szerepelhet (ilyen a from_num és to_num), akkor akár többet is feltüntethetünk. A függőleges tengely az időt szimbolizálja, a nyilak pedig az üzeneteket. A függőleges téglalapok az ok-okozati kapcsolatokat mutatják. Pl. a router akkor küld TryTransform üzenetet, ha a subscriber objektumtól Reroute üzenetet kapott.



6.5. ábra: Szám átirányítása



6.6. ábra: Szám tiltása/engedélyezése



6.7. ábra: Új számpár felvétele

A kommunikációs diagramok felvétele során új üzeneteket vezettünk be. Az összehasonlítás üzenetre (==) a válasz lehet "egyenlő" (*equal*) vagy "nem egyenlő" (*not_equal*). A *TryTransform* és *TrySwitch* üzenetekre válaszul sikeres végrehajtás üzenet (*success*), vagy a kudarcról hírt adó üzenet (*fail*) érkezhet. A *Reroute* válaszüzenetei az "átirányítás megtörtént" (*routed*) és a "nincs átirányítási szabály" (*noroute*). A *SwitchPair* üzenetre "átkapcsolás megtörtént" (*switched*) és "nem történt átkapcsolás" (*noswitch*) érkezhet. Végül a központ (*center*) egy új számpár felvételét *Add* üzenettel kérheti a *router* objektumtól, melynek sikerét a *router* "sikerült" (*success*) vagy "nem sikerült" (*fail*) üzenetekkel jelezheti.

Természetesen a már felvett objektummodell, a specifikációs táblázat és a kommunikációs modell konzisztenciájáról már most célszerű gondoskodni, hiszen ezek ugyanazon rendszer különböző aspektusú modelljei. A kommunikációs modellben olyan objektumok vehetnek részt, amelyeknek megfelelő típus (osztály) az objektum modellben már szerepel. Másrészt, ha a specifikációs táblázatban egy objektumnak illetve műveletnek már adtunk valamilyen nevet, akkor az üzenetek áttekintése során ugyanarra a történésre ugyanazt a nevet kell használnunk. Ez nem jelenti azt, hogy nem lehet olyan újabb üzeneteket felvenni, amelyek a specifikációs táblázatban nem szerepeltek, hiszen az szinte sohasem teljes. Ha viszont egy dolog többször visszaköszön, akkor célszerű a modellekben azt felismerni és a nevek alapján arra utalni. Minden modellezési lépés után az egyes modelleket össze kell vetni egymással. Ily módon a modelleket ismételten finomítva jutunk el egy konzisztens képhez.

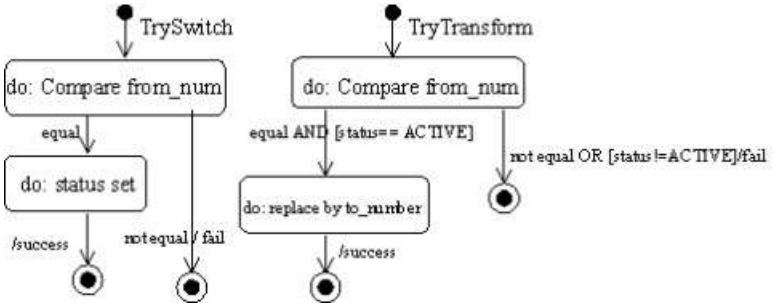
Az **állapotdiagramok** rajzolásánál már erősen támaszkodunk mind az objektummodellre mind a kommunikációs diagramokra. Minden, dinamikus működését tekintve lényeges objektumtípusnak

állapotdiagrammal adjuk meg a működését. Az állapot az objektumot két üzenet között jellemzi. Mivel egy passzív objektum működését az indítja el, hogy egy másiktól üzenetet kap, legfeljebb annyi állapotdiagramot vehetünk fel, ahány féle üzenet érheti ezt az objektumot a kommunikációs modell alapján. Aktív objektumok más objektumoktól független módon is változtathatják belső állapotukat és újabb üzeneteket hozhatnak létre. Ezért minden aktív objektumhoz még legalább egy, az aktív létet szimbolizáló állapotdiagram modell tartozik.

A konzisztencia kérdése itt is felmerül. Egyrészt minden állapottérhez tartoznia kell objektum típusnak, másrészt a kommunikációs modellben szereplő minden üzenethez egy állapottér belépési pontnak, hacsak nem jelentjük ki, hogy az objektum dinamikus működése ezen üzenet hatására triviális.

A pár (pair) objektum dinamikus modellje:

A kommunikációs modell alapján a számpár (*pair*) objektum passzív és TrySwitch, TryTransform és Set üzenetet kaphat. A Set üzenetek végrehajtása triviális így azt nem specifikáljuk állapotdiagrammal. A TrySwitch üzenet hatására a pair objektumnak ellenőriznie kell, hogy a megadott előfizetőtől kapott szám hozzá tartozik-e (from_num), és ha igen, akkor a megfelelő állapot-beállítást el kell végeznie. Az előfizetőtől kapott szám megegyezését, illetve eltérését válaszüzenetben (success illetve fail) kell jelezni az üzenetet küldő objektum felé, hogy az ennek megfelelően folytassa vagy abbahagyja az illeszkedő pár keresését. A TryTransform hasonló módon, az előfizetőtől kapott szám (from_num) ellenőrzésével először eldönti, hogy a transzformáció kérés igazából rá vonatkozik-e, valamint azt, hogy a beállított státusz szerint módjában van-e a kérést kielégíteni. Ha mindkét kérdésre igen a válasz, akkor az előfizetőtől kapott számot a célszámra cseréli ki. Az elmondott folyamatot visszatükröző állapottérdiagramok a következőképpen néznek ki:



6.8. ábra: A pár objektum dinamikus modellje

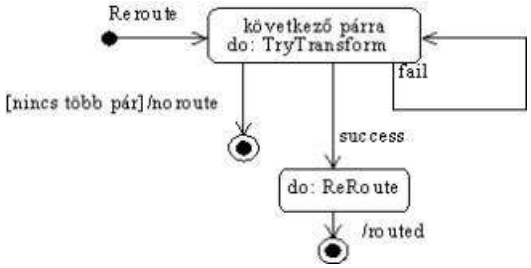
Bár még nem tartunk az implementációnál érdemes egy kicsit előreszaladni és megmutatni, hogy ezek az állapotterek miként határozzák meg az egyes tagfüggvények működését. Mindenekelőtt a kommunikációs diagramon fel kell ismerni az ok-okozati összefüggéseket. Az ok-okozati összefüggések gyakran a feladat elemzéséből természetes módon adódnak, más esetekben a tervezés során kell meghatározni őket. A TryTransform hatására a pair objektum Compare és Set üzeneteket küldhet, majd az okozati láncban success vagy fail válaszüzenet érkezhet vissza. Az implementációs szinten minden objektumot ért hatást tagfüggvénnyel kell realizálni, így a TryTransform a Pair osztály tagfüggvénye lesz, melyben az eseményre adott választ, tehát a from_number számmal összehasonlítást, a to_number számon lekérdezést, és a legvégén a success vagy fail válaszüzenetet kell implementálni. Az összehasonlítást == operátorral, a lekérdezést (Get) = operátorral realizáljuk. Az reakció végén adott válaszüzenetet célszerű a tagfüggvény visszatérési értékével megvalósítani. Összefoglalva itt adjuk meg a keletkező Pair::TryTransform és Pair::TrySwitch függvények egy lehetséges implementációját:

```
enum PairEvent { FAIL, SUCCESS };
enum Switch { OFF, ON };

PairEvent Pair :: TryTransform( String& num ) {
   if ( status == ACTIVE && from_number == num ) {
      num = to_number;  return SUCCESS;
   } else return FAIL;
}

PairEvent Pair :: TrySwitch( String& num, Switch sw ) {
   if ( from_number == num ) {
      if ( sw == ON )  status = ACTIVE;
      else     status = IDLE;
```

```
      return SUCCESS;
   } else return FAIL;
}
```

A router objektum dinamikus modellje:

A kommunikációs modell forgatókönyvei szerint az ugyancsak passzív router objektum Reroute, SwitchPair és Add üzeneteket kaphat. Az Add dinamikus szempontból ismét triviális ezért csak a másik két üzenettel foglalkozunk. Az átirányítást ténylegesen elvégző Reroute üzenetnek végig kell vennie a tárolt átirányítási párokat (pair objektumokat) és meg kell kísérelnie az átirányítást a kapott szám alapján. A próbálkozása addig tart, amíg a tárolt párok el nem fogynak, vagy az átirányítás, az előfizetőtől kapott szám és a pár kezdő tagjának egyezősége miatt sikeres nem lesz. Siker esetén azt is meg kell néznie, hogy az új számot átirányítottuk-e, célszerűen ugyancsak a Reroute tagfüggvény rekurzív aktivizálásával. Végül az átirányítás végét a központ felé jeleznie kell egy megfelelő válaszesemény (routed/noroute) segítségével, amely kifejezi hogy sikerült-e átirányító párt találnia. A leírtak a következő állapotgéppel fogalmazhatók meg:



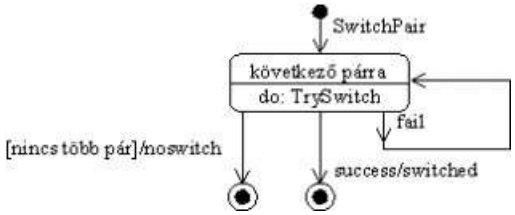
6.9. ábra: A router objektum dinamikus modellje. A Reroute üzenet hatása.

Természetesen az implementáció a dinamikus modell által meghatározott vezérlésen kívül más tényezőktől is függ, például a párokat tároló adatstruktúra szerkezete is befolyásolja azt:

```
enum RouteStat { NOROUTE, ROUTED };

RouteStat Router :: Reroute( String& num ) {
   for( int i = 0; i < npairs; i++ ) {
      if (pairs[i].TryTransform(num) == SUCCESS) {
         Reroute(num);  // új szám átirányítva?
        return ROUTED;
      }
   }
   return NOROUTE;
}
```

Így a Router::Reroute tagfüggvény ezen implementációja már explicit módon visszatükrözi azt a döntést, hogy a párokat egy tömbben tároljuk.

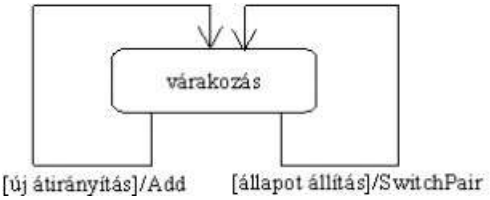
A TrySwitch működése során a párokat végig kell nézni, és a sikeres vagy sikertelen kapcsolás tényét vissza kell jelezni az üzenet küldőjének:



6.10. ábra: A router objektum dinamikus modellje. A SwitchPair üzenet hatása.

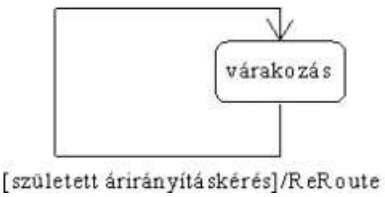
A feladatunkban két aktív és egyben külső objektum szerepel, a központ (center) és az előfizető (subscriber). Ezen objektumok, lévén hogy külsők, a program belsejében csak mint kapcsolattartó, ún. interfész elemekként jelennek meg. Az interfész objektumok pontos specifikálását azonban csak a külső objektumok viselkedésének megértése alapján tehetjük meg. Ezért az analízis során általában nem választjuk szét élesen a külső és belső objektumokat, a külső objektumokat is a megismert módszerekkel írjuk le, és csak a tervezés során húzzuk meg a megvalósítandó rendszer pontos határát.

A központ, az általa meghatározott időpontokban és sorrendben állapotbeállító (SwitchPair) és átirányítást definiáló (Add) üzeneteket küldhet az átirányító (router) objektumnak.



6.11. ábra: A router objektum dinamikus modellje új átirányítás, illetve állapot állítás esetén.

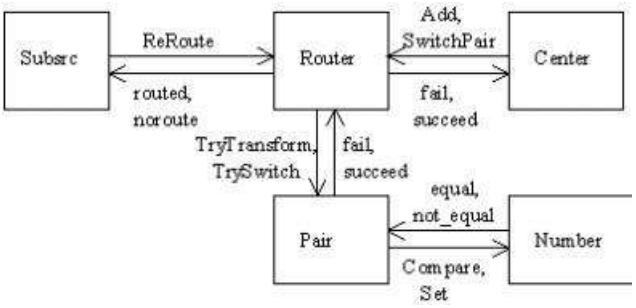
Az előfizető ugyancsak a saját maga által meghatározott időpontokban átirányítás- kérésekkel (ReRoute) bombázhatja az átirányító objektumot:



6.12. ábra: A router objektum dinamikus modellje átirányítás kérés esetén.

Mint azt a fenti diagramok alapján is láthatjuk, az aktív és a passzív objektumok dinamikus modelljei között az alapvető különbség az, hogy az utóbbiak rendelkeznek legalább egy olyan állapotdiagrammal, amely nem külső hatásra lép működésbe és elvileg végtelen sok ideig sem jut befejező (terminális) állapotba.

A különböző objektumtípusok közötti üzenet és esemény átadási lehetőségeket az **eseményfolyam diagrammal**, foglalhatjuk össze:



6.13. ábra: Eseményfolyam diagram.

Tervezés:

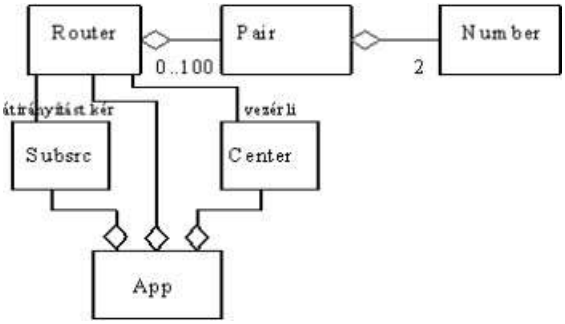
Az analízis során a feladat különböző nézeteit megragadó modelljét állítottuk fel, az absztrakció emelésével. Itt van tehát az ideje, hogy a programmal történő implementáció, mint végső cél alapján, az absztrakció csökkentésével közeledjünk a megvalósítás felé. Ezt a folyamatot tervezésnek nevezzük. A programmal történő megvalósítás olyan, ún. implementációs objektumok megjelenését igényli, amelyek közvetlenül a feladatból nem következnek.

Mindenekelőtt megjelenik a teljes alkalmazói programot megtestesítő ún. **alkalmazói program objektum (*app*)**. A viselkedését tekintve ez az objektum felelős a rendszer megfelelő inicializálásáért, leállításáért a működés során fellépő hibák (melyekről idáig mélyen hallgattunk) kijelzéséért és lekezeléséért. Amennyiben az

alkalmazásban több aktív objektum, azaz lényegi párhuzamosság is fellelhető, az applikációs objektum a rendelkezésre álló processzor idejének az aktív objektumok közötti szétosztásában is részt vehet.

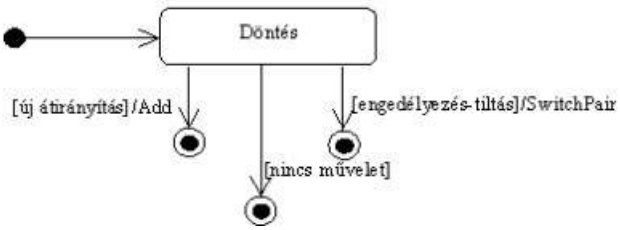
Ennek értelmében egy alkalmazói program objektum általában hibakezelő (Error) és befejező (Exit) üzenettel rendelkezik, valamint egy privát Execute vagy Start üzenettel, amely az alkalmazás többi részét elindítja. A program leállításáról és a potenciálisan fellépő hibák felméréséről és kezeléséről a tervezés során kell dönteni. Egyenlőre hozzuk meg azt a döntést, hogy a programot a központ (center) állítja le az alkalmazói program objektumnak küldött Exit üzenettel. Az alkalmazói program objektum tulajdonképpen aktív, hiszen a program indítása után működésbe lép és minden üzenetláncnak a kezdeti pontját képviseli.

Az applikációs objektum másik fontos tulajdonsága, hogy minden egyéb objektum közvetve, vagy közvetlenül ehhez az objektumhoz tartozik. Ennek megfelelően az objektummodellnek, a tervezési szempontokat is figyelembe vevő módosított változata:



6.14. ábra: A tervezési szempontokat figyelembe vevő osztálydiagram.

A tervezés következő lépése az aktív objektumok **ütemezési stratégiájának** a kialakítása, melynek egyik megoldása az eredetileg párhuzamos működésű modell szekvenciálissá alakítása (a lehetséges megoldási módokat a 7.6. fejezetben részletesen elemezzük). Most a **nem-preemptív ütemezési** elvek alapján, az aktív objektumok működését leíró ciklikus állapotgépek bejárását olyan elemi lépésekre bontjuk, ahol egy lépés garantáltan véges időt igényel. Ezt az időt úgy kell megválasztani, hogy a többi aktív objektum működését ne tartsuk fel megengedhetetlenül sokáig. Az egyes lépések végrehajtását az aktív objektumokhoz rendelt Do_a_Step metódusokra bízzuk, melyet az ütemező, jelen esetben az alkalmazói program objektum, periodikusan hív. A központ módosított állapottér modellje, melyet a Do_a_Step üzenet hatására jár be:



6.15. ábra: A központ módosított állapottér modellje.

Ebben az állapottér-modellben nem lehetnek végtelen ciklusok és potenciálisan végtelen sokáig fennálló állapotok. Ezért váltottuk fel a *várakozó* állapotot *döntéssel,* melyből akkor is továbblépünk a *nincs művelet* ágon, ha éppen *új átirányítást* sem és *engedélyezést* vagy *tiltást* sem kívánunk végrehajtani. Az elmondottak értelmében az ütemezéshez szükséges tagfüggvények az osztályok definíciójában a következőképpen vesznek részt:

```
class Subscriber {
public:  void Do_a_Step();
};

class Center {
public:  void Do_a_Step();
};

class App {
```

```
   Subscriber subscriber;
   Center     center;
public:
   void Execute( ) {
      for( ; ; ) {
         center.Do_a_Step();  subscriber.Do_a_Step();
      }
   }
};
```

A programunk természetesen egy nem objektumorientált környezetben indul, hiszen a C++ programunk belépési pontja, a C-hez hasonlóan egy globális *main* függvény. Egy szép objektumorientált program ezt a belépési pontot az alkalmazói program objektum segítségével eltakarja a program többi részétől, minek következtében a *main* függvény egyetlen feladata az applikációs objektum létrehozása és a működésének az elindítása:

```
void main( ) {       // NEM OOP -> OOP interfész
   App app;
   app.Execute( );
}
```

A tervezés további feladata az objektumok közötti **láthatósági viszonyok biztosítása.** Az analízis során általában nem vesződünk a láthatósági kérdésekkel, hanem nagyvonalúan feltételezzük, hogy ha egy objektum üzenetet kíván küldeni egy másiknak akkor azt valamilyen módon meg is tudja tenni. Az implementáció felé közeledve azonban figyelembe kell vennünk az adott programozási nyelv sajátságait, amelyek a változókhoz (objektumokhoz) való hozzáférést csak adott nyelvi szabályok (ún. láthatósági szabályok) betartása esetén teszik lehetővé.

Annak érdekében, hogy a láthatósággal kapcsolatos kérdéskört a C++ nyelv keretein belül megvilágítsuk, tegyük fel, hogy egy Sender osztályhoz tartozó sender objektum a Sender::f( ) tagfüggvény végrehajtása során üzenetet akar küldeni, vagy egyszerűen fel akar használni (pl. értékadásra vagy argumentumként) egy receiver objektumot.

Képzeljük magunk elé a Sender::f tagfüggvény implementációját az adott forrásállományban bennlévő környezetével együtt. Mint tudjuk, a saját attribútumok elérését a C++ egy implicit this mutató automatikus elhelyezésével biztosítja, melyet ugyancsak feltüntettünk:

```
globális változók;

Sender :: f ( [Sender * this], argumentumok )
{
   lokális változók;
   receiver.mess( );      // direkt üzenetküldés
   preceiver -> mess( );  // indirekt üzenetküldés
}
```

Miképpen a közönséges C nyelvben is, egy függvényen belül a következő változó érhetők el, illetve láthatók:

1. az adott forrásállományban az adott sor előtt deklarált globális, tehát függvényblokkon kívül definiált változók,

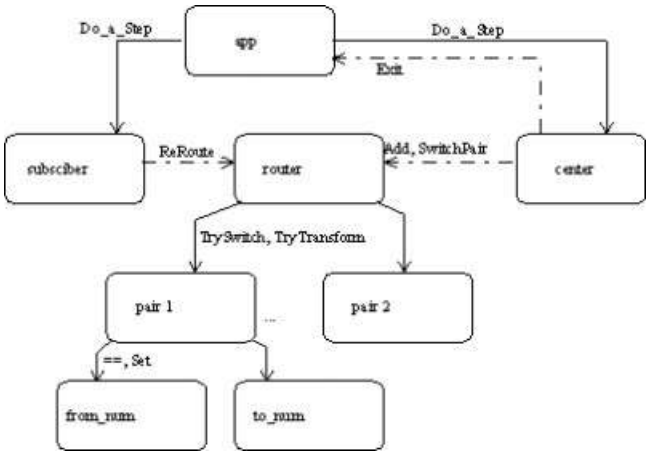
2. a függvény lokális, tehát a függvény kezdő és lezáró {} zárójelei között definiált változói,

3. a függvény paraméterei,

4. egy C++ programban ehhez jönnek még az objektum saját komponensei, illetve attribútumai, melyek elérését a fordítóprogram a this mutató argumentumként történő átadásával biztosítja.

A fenti példában tehát a receiver objektumnak, vagy a címét tartalmazó mutatónak a Sender::f-hez képest a fenti esetek valamelyikének megfelelően kell elhelyezkednie. Tekintve, hogy a második és harmadik elrendezés eléggé esetleges (a részleteket lásd a 7.4. fejezetben), az első pedig globális változókat igényel, melyek használata ellentmond az információrejtés objektumorientált elvének, a láthatósági viszonyok kielégítésére leggyakrabban a 4. megoldást alkalmazzuk. A célobjektumot vagy annak címét, esetleg referenciáját, a küldő objektum komponenseként, illetve attribútumaként valósítjuk meg.

Most térjünk vissza a megoldandó telefonszám-átirányító programunkhoz. A szükséges láthatósági viszonyokat, azaz azt az információt, hogy egy objektum milyen más objektumoknak küld üzenetet illetve milyen más objektumokat használ fel, legkönnyebben az objektum kommunikációs diagram alapján tekinthetjük át:



6.16. ábra: Az objektumok kommunikációs diagramja.

Az ábrán szaggatott vonallal jelöltük azon üzeneteket, melyeket a tartalmazási relációk nem biztosítanak. Ezen kívül még figyelembe kell vennünk, hogy hibajelenség elvileg minden objektum belsejében keletkezhet, így elvileg a hibakezelést végző alkalmazói program objektumot (app) mindenkinek látnia kell. Az ilyen globális láthatóságot úgy biztosíthatjuk, ha az app objektumot globális változóként definiáljuk, ami lehetővé teszi a center objektum számára is az Exit üzenet küldését.

Még két láthatósági igénnyel kell megküzdenünk. Az előfizető (subsciber) objektumnak látnia az átirányítót (router) a ReRoute üzenet elküldéséhez. Hasonlóan a központ (center) objektumnak is látnia kell az átirányítót, az Add és a SwitchPair üzenetek miatt.

Felmerülhet bennünk, hogy a router objektumot a center (vagy a subsciber, de csak az egyik lehet a kettő közül) komponenseként valósítjuk meg, hiszen ezzel az egyik igényt kielégíthetjük. A másik igényt viszont ez sem oldja meg, ráadásul felborítja az analízis során kialakított tartalmazási viszonyokat. Az ilyen mesterkélt beavatkozás a program értelmezése és későbbi módosítása esetén hátrányos lehet.

Igazán kielégítő megoldást az biztosít, ha a router objektum címét (vagy referenciáját) mind a subsciber mind a center objektumokban tároljuk, és szükség esetén ezen cím felhasználásával indirekt üzenetküldést valósítunk meg. Ezen belső mutatók inicializálása külön megfontolást igényel. Nyilván csak olyan objektum teheti ezt meg, amely látja mind a subsciber, mind pedig a center objektumokat, hogy inicializáló üzenetet küldhessen nekik, továbbá látja a router objektumot, hogy annak címét képezhesse és átadhassa az inicializáló üzenetben. A kommunikációs diagramra ránézve egyetlen ilyen objektum jöhet szóba, az alkalmazói program objektum.

Ennek megfelelően az app, a center és a subscriber objektumokat definiáló osztályok kialakítása a következőképpen néz ki:

```
class Subscriber {
   Router *  prouter;
public:
      Subscriber( Router * r ) { prouter = r; }
```

```
   void  Do_a_Step( ) { prouter -> ReRoute(....); }
};

class Center {
   Router *  prouter;
public:
       Center( Router * r ) { prouter=r; }
   void  Do a Step( );
};

class App {
   Subscriber   subscriber;
   Center       center;
   Router       router;
public:
      App() : subscriber(&router), center(&router) {}
   void  Error(....);
};

App app;  // az applikációs objektum globális

void Center :: Do_a_Step( ) {
   prouter -> AddPair(....);
   ....
   app.Error(....);   // üzenet a globális objektumnak
}
```

A szekvenciális programmal történő implementáció szempontjainak érvényesítése után hozzákezdhetünk az analízis modellek C++ implementációra történő transzformációjához, melyben először az ott említett osztályok pontos deklarációját kell megalkotni. Az analízis modellek hordozta információ általában nem teljes, elsősorban az objektumok kapcsolatát emelik ki, de azok belső világáról keveset mondanak. A hiányzó információkat tervezési döntésekkel kell pótolni, melyeknek ki kell térnie az objektumok belső adatstruktúráinak megvalósítására és a tagfüggvények szerkezetének a dinamikus és funkcionális modellek által nyitva hagyott kérdéseire is.

A modellekről a C++ programsorokra történő áttérés során gyakran hasznos a rendelkezésre álló információnak egy közbülső formában történő leírása is, amely a szereplő osztályok viszonylatában egy-egy kérdőív kitöltéséből áll. A kérdőív tipikus kérdései tartalmazzák az osztály nevét, feladatát vagy felelősségét informálisan megfogalmazva, a programban az ilyen típusú objektumok várható számát, a biztosított szolgáltatásokat (interfészt), az osztály metódusai által felhasznált más objektumokat, az osztályhoz tartozó tartalmazási (komponensi) és asszociációs viszonyokat, az osztályban definiált attribútumokat és végül az osztállyal definiált objektumok aktív illetve passzív jellegét.

Az átirányító párokat (pair) definiáló Pair osztályhoz tartozó kérdőív:

osztálynév: *Pair*

felelősség: Egy számpár nyilvántartása, az első számpár egyezés esetén, a második számpár kiadása (transzformáció) valamint a kapcsolási státusz kezelése

példányok: több (max 100)

szolgáltatások: void Set(Number& f, Number& t); enum {FAIL, SUCCESS} TryTransform(Number& num);enum {FAIL, SUCCESS} TrySwitch(Number& num, Switch sw);

kiszolgálók: A tartalmazott két *Number* objektum

komponensek: Number *from_num*, *to_num*

attribútumok: enum {ACTIVE, IDLE} status;

párhuzamosság: passzív

Az telefonszámokat reprezentáló osztály:

osztálynév: *Number*

felelősség: ASCII karaktersorozatból álló tetszőleges hosszú szám nyilvántartása, beállítása, összehasonlítása.

példányok: több

szolgáltatások: void Set(Number) , = ; int Compare(Number num), ==

párhuzamosság: passzív

Az osztály felelősségét és a szükséges szolgáltatásokat figyelembe véve megállapíthatjuk, hogy a *Number* osztály nem más mint egy olyan dinamikusan nyújtózkodó karaktersorozat, amely csak ASCII számjegyeket tartalmaz. Dinamikusan nyújtózkodó karaktersorozatok definiálására hoztuk létre a korábbiakban (6.5. fejezet) a *String* osztályt, így azt változtatás nélkül átvehetjük a telefonszám objektumok megvalósításához is.

Az átirányítást végző objektumot definiáló osztály:

osztálynév: *Router*

felelősség: A számpárok nyilvántartása, felvétele, az elsőből a második meghatározása rekurzívan, a párok engedélyezésének és tiltásának az állítása.

példányok: egy

szolgáltatások: enum {NOROUTE, ROUTED} Reroute(Number& num); enum {FAIL, SUCCESS} Add(Number from, Number to); enum {NOSWITCH,SWITCHED} SwitchPair(Number n, Switch sw);

kiszolgálók: tartalmazott *Pair* objektumok

komponensek: maximum 100 db *Pair*

adatstruktúra: tömb: Pair pair[100];

attribútumok: int npairs; a felvett párok száma

párhuzamosság: passzív

Itt már természetesen a belső adatstruktúrára tett tervezési döntések is szerepelnek, nevezetesen az átirányítási párokat, a 100-as felsőhatár kihasználásával egy rögzített méretű tömbben tároljuk. Természetesen ennek a döntésnek a tagfüggvények implementációjára és a teljesítmény jellemzőire (futási sebesség) is döntő hatása van.

Az előfizetőt definiáló osztály:

osztálynév: *Subscriber*

felelősség: Telefonhívások létrehozása

példányok: egy

kiszolgálók: *Router*

beágyazott mutatók: *router* objektum címe, melyet az *app* inicializál

relációk: *router*-nek elküldi a kéréseit,

párhuzamosság: aktív, nem-preemptív ütemezett

A telefonközpontot leíró osztály:

osztálynév: *Center*

felelősség: Operátori parancsok létrehozása: új pár, pár kapcsolása, leállítás

példányok: egy

kiszolgálók: *Router, App*

beágyazott mutatók: *router* objektum címe, melyet az *app* inicializál

relációk: *router*-nek elküldi az operátor kéréseit, *app*-nak hibajelzést és megállási parancsot

Párhuzamosság: aktív, nem-preemptív ütemezett

Végül az applikációs objektumot létrehozó osztály:

osztálynév: *App*

felelősség: Az objektumok inicializálása, ütemezése és hibakezelése, valamint a *router*-re hivatkozó beágyazott mutatók inicializálása a *subscriber* és a *center* objektumokban.

példányok: egy, a láthatósági igények miatt globális

szolgáltatások: void Execute( );void Error(char * mess); void Exit( );

kiszolgálók: *subsciber* és *center* (ütemezett objektumok)

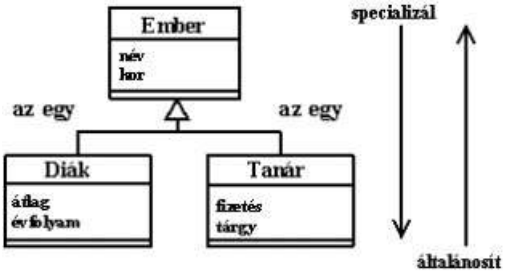
komponensek: *subscriber*, *center*, *router*

párhuzamosság: aktív, ütemező

Ezekből a kérdőívekből az osztályok C++ deklarációi már könnyen elkészíthetők. A függvények implementációit, az állapotdiagramok, a folyamatspecifikációk és a belső adatszerkezetekre tett tervezési döntések alapján ugyancsak nehézségek nélkül megvalósíthatjuk. A teljes program megtalálható a lemezmellékleten.

7. 6.7. Öröklődés

Az öröklődés objektumtípusok között fennálló speciális kapcsolat, amely az analízis során akkor kerül felszínre, ha egy osztály egy másik általánosításaként, vagy megfordítva a másik osztály az egyik specializált változataként jelenik meg. A fogalmakat szemléltetendő, tekintsük a következő osztályokat, melyek egy tanulócsoportot kezelő program analízise során bukkanhatnak fel.



6.17. ábra: Egy tanulócsoport objektumtípusai.

Egy oktatási csoportban diákok és tanárok vannak. Közös tulajdonságuk, hogy mindnyájan emberek, azaz a diák és a tanár az ember speciális esetei, vagy fordítva az ember, legalábbis ebben a feladatban, a diák és tanár közös tulajdonságait kiemelő általánosító típus. Szokás ezt a viszonyt "az egy" (*IS_A*) relációnak is mondani, hiszen ez, beszélt nyelvi eszközökkel tipikusan úgy fogalmazható meg, hogy:

a diák az egy ember, amely még ...

a tanár az (is) egy ember, amely még ...

A három pont helyére a diák esetében az átlageredményt és az évfolyamot, míg a tanár esetében a fizetést és az oktatott tárgyat helyettesíthetjük.

Ha ezekkel az osztályokkal programot kívánunk készíteni, arra alapvetően két eltérő lehetőségünk van.

- 3 darab független osztályt hozunk létre, ahol az egyik az általános ember fogalomnak, a másik a tanárnak, míg a harmadik a diáknak felel meg. Sajnos ekkor az emberhez tartozó felelősségek, pontosabban a programozás szintjén a tagfüggvények, háromszor szerepelnek a programunkban.

- A másik lehetőség a közös rész kiemelése, melyet az **öröklődéssel** (**inheritance**) történő definíció tesz lehetővé. Ennek lépései:

  1. Ember definíciója. Ez az ún. **alaposztály** (**base class**).

  2. A diákot úgy definiáljuk, hogy megmondjuk, hogy az egy ember és csak az ezen felül lévő új dolgokat specifikáljuk külön: Diák = Ember + valami (adatok, műveletek)

  3. Hasonlóképpen járunk el a tanár megadásánál is. Miután tisztázzuk, hogy annak is az Ember az alapja, csak az tanár specialitásaival kell foglalkoznunk: Tanár = Ember + más valami

Ennél a megoldásnál a Diák és a Tanár **származtatott osztályok** (**derived class**).

Az öröklődéssel történő megoldásnak számos előnye van:

- Hasonlóság kiaknázása miatt a végleges programunk egyszerűbb lehet. A felesleges redundanciák kiküszöbölése csökkentheti a programozási hibák számát. A fogalmi modell pontosabb visszatükrözése a programkódban világosabb programstruktúrát eredményezhet.

- Ha a későbbiekben kiderül, hogy a programunk egyes részein az osztályhoz tartozó objektumok működésén változtatni kell (például olyan tanárok is megjelennek, akik több tárgyat oktatnak), akkor a meglévő osztályokból származtathatunk új, módosított osztályokat. A származtatás átmenti az idáig elvégzett munkát anélkül, hogy egy osztály, vagy a program egyéb részeinek módosítása miatt a változtatások újabb hibákat ültetnének be programba.

- Lényegében az előző biztonságos programmódosítás "ipari" változata az osztálykönyvtárak felhasználása. A tapasztalat azt mutatja, hogy egy könyvtári elem felhasználásának gyakori gátja az, hogy mindig "csak egy kicsivel" másként működő dologra van szükség mint ami rendelkezésre áll. A függvényekből álló hagyományos könyvtárak esetében ekkor meg is áll a tudomány. Az öröklődésnek köszönhetően az osztálykönyvtárak osztályainak a viselkedése rugalmasan szabályozható, így az osztálykönyvtárak a függvénykönyvtárakhoz képest sokkal sikeresebben alkalmazhatók. Ezen és a megelőző pontot összefoglalva kijelenthetjük, hogy az öröklődésnek, az analízis modelljének a pontos leképzésén túl egy fontos felhasználási területe a programelemek **újrafelhasználhatóságának** (**software reuse**) támogatása, ami az objektumorientált programozásnak egyik elismert előnye.

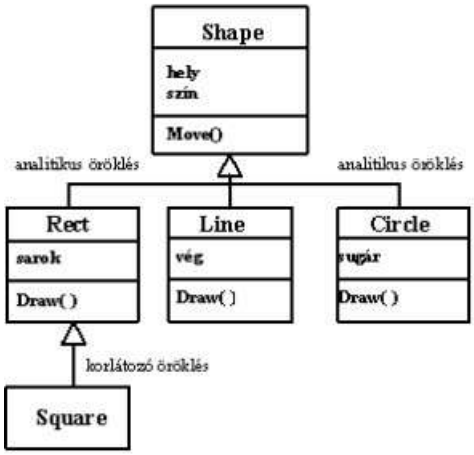
- Végül, mint látni fogjuk, egy igen hasznos programozástechnikai eszközt, a különböző típusú elemeket egyetlen csoportba szervező és egységesen kezelő **heterogén szerkezetet**, ugyancsak az öröklődés felhasználásával valósíthatunk meg hatékonyan.

7.1. 6.7.1. Egyszerű öröklődés

Vegyük először a geometriai alakzatok öröklődési példáját. Nyilván minden geometriai alakzatban van közös, nevezetesen azok a tulajdonságok és műveletek, amelyek a geometriai alakzatokra általában érvényesek. Beszélhetünk a színükről, helyükről és a helyet megváltoztató mozgatásról anélkül, hogy a geometriai tulajdonságokat pontosabban meghatároznánk. Egy ilyen általános geometriai alakzatot definiáljuk a Shape osztállyal. Az egyes tényleges geometriai alakzatok, mint a téglalap (Rect), a szakasz (Line), a kör (Circle) ennek az általános alakzatnak a speciális esetei, azaz kézenfekvő az ezeket szimbolizáló osztályokat a Shape származtatott osztályaiként definiálni. A Shape tulajdonságaihoz képest, a téglalap átellenes sarokponttal, a kör sugárral, a szakasz másik végponttal rendelkezik, és mindegyikhez tartozik egy új, osztályspecifikus rajzoló (Draw) metódus, amely az adott objektumot a konkrét típusnak megfelelően felrajzolja. A mozgatásról (Move) az előbb megjegyeztük, hogy mivel a helyhez kapcsolódik tulajdonképpen az általános alakzat része. A mozgatás megvalósítása során először a régi helyről le kell törölni az objektumot (rajzolás háttérszínnel), majd az új helyen kell megjeleníteni (ismét Draw). Természetesen a Draw nem általános, hanem a konkrét típustól függ így a Move tagfüggvény Shape-ben történő megvalósítása sem látszik járhatónak. Hogy megvalósítható-e

vagy sem a közös részben, az egy igen fontos kérdés lesz. Egyelőre azonban tekintsük a Move tagfüggvényt is minden osztályban külön megvalósítandónak.

Az öröklődési gondolatot tovább folytatva felvethetjük, hogy a téglalapnak van egy speciális esete, a négyzet (Square), amit célszerűnek látszik a téglalapból származtatni. Ha meg akarnánk mondani, hogy a négyzet milyen többlet attribútummal és művelettel rendelkezik a téglalaphoz képest, akkor gondban lennénk, hiszen az éppenhogy csökkenti a végrehajtható műveletek számát illetve az attribútumokra pótlólagos korlátozásokat (szemantikai szabályok) tesz. Például egy téglalapnál a két sarokpont független változtatása teljesen természetes művelet, míg ez a négyzetnél csak akkor engedhető meg, ha a függőleges és vízszintes méretek mindig megegyeznek.



6.18. ábra: Geometriai alakzatok öröklési fája.

Ezek szerint a négyzet és a téglalap kapcsolata alapvetően más, mint például az alakzat és a téglalap kapcsolata. Az utóbbit **analitikus öröklődésnek** nevezzük, melyre jellemző, hogy az öröklődés új tulajdonságokat ad az alaposztályból eredő tulajdonságokhoz anélkül, hogy az ott definiáltakat csorbítaná. A négyzet és a téglalap kapcsolata viszont nem analitikus (ún. **korlátozó**) öröklődés, hiszen ez letiltja, vagy pedig korlátozva módosítja az alaposztály bizonyos műveleteit.

Ha egy adott szituációban kétségeink vannak, hogy milyen öröklődésről van szó, használhatjuk a következő módszert az analitikus öröklődés felismerésére: "*Az A osztály analitikusan származtatott osztálya B-nek, ha A típusú objektumot adva egy olyan személynek, aki azt hiszi, hogy B típusút kap, ez a személy úgy fogja találni, hogy az objektum valóban B típusú miután elvégezte a feltételezése alapján végrehajtható teszteket*". Kicsit formálisabban fogalmazva: analitikus öröklődés esetén az *A* típusú objektumok felülről kompatibilisek lesznek a *B* osztályú objektumokkal, azaz *A* metódusai *B* ugyanezen metódusaihoz képest a bemeneti paraméterekre vonatkozó előfeltételeket (prekondíciót) legfeljebb enyhíthetik, míg a kimeneti eredményekre vonatkozó megkötéseket (posztkondíciót) legfeljebb erősíthetik. A nem analitikus öröklődés ilyen kompatibilitást nem biztosít, melynek következtében a programozóra számos veszély leselkedhet az implementáció során. Mint látni fogjuk a C++ biztosít némi lehetőséget ezen veszélyek kivédésére (privát alaposztályok), de ezekkel nyilván csak akkor tudunk élni, ha az ilyen jellegű öröklődést felismerjük. Ezért fontos a fenti fejtegetés. Ennyi filozófia után rögvest felmerül a kérdés, hogy használhatjuk-e a nem analitikus öröklődést az objektumorientált modellezésben és programozásban. Bár a szakma meglehetősen megosztott ebben a kérdésben, mi azt a kompromisszumos véleményt képviseljük, hogy modellezésben lehetőleg ne használjuk, illetve ha szükséges, akkor azt tudatosan, valamilyen explicit jelöléssel tegyük meg. Az implementáció során ezen kompromisszum még inkább az engedékenység felé dől el, egyszerűen azért, mert elsősorban a kód újrafelhasználáskor vannak olyan helyzetek, mikor a nem analitikus öröklődés jelentős programozói munkát takaríthat meg. A kritikus pont most is ezen szituációk felismerése, hiszen ez szükséges ahhoz, hogy élni tudjunk a veszélyek csökkentésére hivatott lehetőségekkel.

A modellezési példák után rátérhetünk az öröklődés C++-beli megvalósítására. Tekintsük először a geometriai alakzatok megvalósításának első kísérletét, melyben egyelőre csak a Shape és a Line osztályok szerepelnek:

```
class Shape {
protected:
   int x, y, col;
public:
      Shape( int x0, int y0, int col0 )
         { x = x0; y = y0; col = col0; }
   void SetColor( int c ) { col = c; }
};
class Line : public Shape { // Line = Shape + ...
      int xe, ye;
   public:
         Line( int x1, int y1, int x2, int y2, int c )
         : Shape( x1, y1, c ) { xe = x2, ye = y2 }
      void Draw( );
      void Move( int dx, int dy );
   };

void Line :: Draw( ) {
   _SetColor( col ); // rajz a grafikus könyvtárral
   _MoveTo( x, y );
    LineTo( xe, ye );
}

void Line :: Move( int dx, int dy ) {
   int cl = col;  // tényleges rajzolási szín elmentése
   col = BACKGROUND; // rajzolási szín legyen a háttér színe
   Draw( );   // A vonal letörlés az eredeti helyről
   x += dx; y += dy;  // mozgatás: a pozíció változik
   col = cl;  // rajzolási szín a tényleges szín
   Draw( );   // A vonal felrajzolása az új pozícióra
}
```

A programban számos újdonsággal találkozunk:

- Az első újdonság a **protected** hozzáférés-módosító szó a Shape osztályban, amely a *public* és *private* definíciókhoz hasonlóan az utána következő deklarációkra vonatkozik. Ennek szükségességét megérthetjük, ha ránézünk a származtatott osztály (Line) Move tagfüggvényének implementációjára, amelyben a helyzet információt (x,y) nyilván át kell írni. Egy objektum tagfüggvényéből (mint a Line::Move), ismereteink szerint nem férhetünk hozzá egy másik típus (Shape) privát tagjaihoz. Ezen az öröklődés sem változtat. Érezhető azonban, hogy az öröklődés sokkal közvetlenebb viszonyt létesít két osztály között, ezért szükségesnek látszik a hozzáférés olyan engedélyezése, amely a privát és a publikus hozzáférés között a származtatott osztályok tagfüggvényei számára hozzáférhetővé teszi az adott attribútumokat, míg az idegenek számára nem. Éppen ezt valósítja meg a védett (*protected*) hozzáférést engedélyező kulcsszó. Igenám, de annakidején a belső részletek eltakarását és védelmét (*information hiding*) éppen azért vezettük be, hogy ne lehessen egy objektum belső állapotát inkonzisztens módon megváltoztatni. Ezt a szabályt most, igaz csak az öröklődési láncon belül, de mégiscsak felrúgtunk. Általánosan kimondható tanács a következő: egy osztályban csak azokat az attribútumokat szabad védettként (vagy publikusként) deklarálni, melyek független megváltoztatása az objektum állapotának konzisztenciáját nem ronthatja el. Vagy egyszerűbben lehetőleg kerüljük a **protected** kulcsszó alkalmazását, hiszen ennek szükségessége arra is utal, hogy az attribútumokat esetleg nem megfelelően rendeltük az osztályokhoz.

- A második újdonság a Line osztály deklarációjában van, ahol a

  ```
  class Line : public Shape { ... }
  ```

  sor azt fejezi ki, hogy a Line osztályt a Shape osztályból származtattuk. A **public öröklődési specifikáció** arra utal, hogy az új osztályban minden tagfüggvény és attribútum megtartja a Shape-ben érvényes hozzáférését, azaz a Line típusú objektumok is rendelkeznek publikus SetColor metódussal, míg az örökölt x,y,col attribútumaik továbbra is védett elérésűek maradnak. Nyilván erre az öröklődési fajtára az analitikus öröklődés implementációja esetén van szükség, hiszen ekkor az örökölt osztály objektumainak az alaposztály objektumainak megfelelő funkciókkal is rendelkezniük kell. Nem analitikus öröklődés esetén viszont éppenhogy el kell takarni bizonyos metódusokat és attribútumokat. Például, ha a feladat szerint szükség volna olyan szakaszokra, melyek színe megváltoztathatatlanul piros, akkor kézenfekvő a Line-ból egy RedLine származtatása, amely során a konstruktort úgy valósítjuk meg, hogy az a col mezőt mindig pirosra

inicializálja és a SetColor tagfüggvénytől pedig megszabadulunk. Az öröklődés során az öröklött tagfüggvények és attribútumok eltakarására a **private öröklődési specifikációt** használjuk. A

```
class RedLine: private Line { ... };
```

az alaposztályban érvényes minden tagot a származtatott osztályban privátnak minősít át. Amit mégis át akarunk menteni, ahhoz a származtatott osztályban egy publikus közvetítő függvényt kell írnunk, amely meghívja a privát tagfüggvényt. Fontos, hogy megjegyezzük, hogy a származtatott osztályban az alaposztály függvényeit újradefiniálhatjuk, amely mindig felülbírálja az alaposztály ugyanilyen nevű tagfüggvényét. Például a Line osztályban a SetColor tagfüggvényt ismét megvalósíthatjuk esetleg más funkcióval, amely ezek után a Line típusú és minden Line-ból származtatott típusú objektumban eltakarja az eredeti Shape::SetColor függvényt.

- A harmadik újdonságot a Line konstruktorának definíciójában fedezhetjük fel, melynek alakja:

```
Line(int x1, int y1, int x2, int y2, int c)

: Shape(x1,y1,c) {xe = x2; ye = y2}
```

Definíció szerint egy származtatott osztály objektumának létrehozásakor, annak konstruktorának meghívása előtt (pontosabban annak első lépéseként, de erről később) az alaposztály konstruktora is automatikusan meghívásra kerül. Az alaposztály konstruktorának argumentumokat átadhatunk át. A fenti példában a szakasz (Line) attribútumainak egy része saját (xe,ye végpontok), míg másik részét a Shape-től örökölte, melyet célszerű a Shape konstruktorával inicializáltatni. Ennek formája szerepel a példában.

Ezek után kíséreljük meg még szebbé tenni a fenti implementációt. Ha gondolatban az öröklődési lépések felhasználásával definiáljuk a kör és téglalap osztályokat is, akkor megállapíthatjuk, hogy azokban a Move függvény implementációja betűről-betűre meg fog egyezni a Line::Move-val. Egy "apró" különbség azért mégis van, hiszen mindegyik más Draw függvényt fog meghívni a törlés és újrarajzolás megvalósításához (emlékezzünk vissza a modellezési kérdésünkre, hogy a Move közös-e vagy sem). Érdemes megfigyelni, hogy a Move kizárólag a Shape attribútumaival dolgozik, így a Shape-ben történő megvalósítása azon túl, hogy szükségtelenné teszi a többszörös definíciót, logikusan illeszkedik az attribútumokhoz kapcsolódó felelősség elvéhez és feleslegessé teszi az elítélt védett hozzáférés (*protected*) kiskapu alkalmazását is.

Ha létezne egy "manó", aki a Move implementációja során mindig az objektumot definiáló osztálynak megfelelő Draw-t helyettesítené be, akkor a Move-ot a Shape osztályban is megvalósíthatnánk. Ezt a "manót" úgy hívjuk, hogy **virtuális tagfüggvény**.

Virtuális tagfüggvény felhasználásával az előző programrészlet lényeges elemei, kiegészítve a Rect osztály definíciójával, a következőképpen festenek:

```
class Shape {
protected:
   int x, y, col;
public:
         Shape( int x0, int y0, int col0)
            { x = x0; y = y0; col = col0; }
   void SetColor( int c ) { col = c; }
   void Move( int dx, int dy );
   virtual void Draw( ) { }
};

void Shape :: Move( int dx, int dy ) {
   int cl = col;     // tényleges rajzolási szín elmentése
   col = BACKGROUND; // rajzolási szín legyen a háttér színe
   Draw( );          // A vonal letörlés az eredeti helyről
   x += dx; y += dy; // mozgatás: a pozíció változik
   col = cl;         // rajzolási szín a tényleges szín
   Draw( );          // A vonal felrajzolása az új pozícióra
}

class Line : public Shape {   // Line = Shape + ...
   int xe, ye;
public:
```

```
         Line( int x1, int y1, int x2, int y2, int c )
            : Shape( x1, y1, c ) { xe = x2, ye = y2;}
   void Draw( );
};

class Rect : public Shape {   // Rect = Shape + ...
   int xc, yc;
public:
         Rect( int x1, int y1, int x2, int y2, int c )
            : Shape( x1, y1, c ) { xc = x2, yc = y2; }
   void Draw( );
};
```

Mindenekelőtt vegyük észre, hogy a Move változatlan formában átkerült a Shape osztályba. Természetesen a Move tagfüggvény itteni megvalósítása már a Shape osztályban is feltételezi egy Draw tagfüggvény meglétét, hiszen itt még nem lehetünk biztosak abban, hogy a Shape osztályt csak alaposztályként fogjuk használni olyan osztályok származtatására, ahol a Draw már értelmet kap. Mivel "alakzat" esetén a rajzolás nem definiálható, a Draw törzsét üresen hagytuk, de és itt jön a lényeg a Draw függvényt az alaposztályban virtuálisként deklaráltuk. Ezzel aktivizáltuk a "manót", hogy gondoskodjon arról, hogy ha a Shape-ből származtatunk egy másik osztályt ahol a Draw új értelmez kap, akkor már a Shape-ben definiált Move tagfüggvényen belül is az új Draw fusson le. A megvalósítás többi része magáért beszél. A Line és Rect osztály definíciójában természetesen újradefiniáljuk az eredeti Draw tagfüggvényt.

Most nézzünk egy egyszerű rajzoló programot, amely a fenti definíciókra épül és próbáljuk megállapítani, hogy az egyes sorok milyen tagfüggvények meghívását eredményezik virtuálisnak és nem virtuálisnak deklarált Shape::Draw esetén:

```
main ( ) {
   Rect rect( 1, 10, 2, 40, RED );
   Line line( 3, 6, 80, 40, BLUE );
   Shape shape( 3, 4, GREEN );    // :-(
   shape.Move( 3, 4 );            // 2 db Draw hívás :-(
   line.Draw( );                  // 1 db Draw
   line.Move( 10, 10 );           // 2 db Draw hívás
   Shape * sp[10];
   sp[0] = &line;                 // nem kell típuskonverzió
   sp[1] = &rect;
   for( int i = 0; i < 2; i++ )
     sp[i] -> Draw( );            // indirekt Draw()
}
```

A fenti program végrehajtása során az egyes utasítások során meghívott Draw függvény osztályát, virtuális és nem virtuális deklaráció esetén a következő táblázatban foglaltuk össze:

	Virtuális Shape::Draw	**Nem virtuális Shape::Draw**
shape.Move()	Shape::Draw	Shape::Draw
line.Draw()	Line::Draw	Line::Draw
line.Move()	Line::Draw	Shape::Draw
sp[0]->Draw(), mutatótípus Shape *,	Line::Draw	Shape::Draw

de Line objektumra mutat		
sp[1]->Draw(), mutatótípus Shape *, de Rect objektumra mutat	Rect::Draw	Shape::Draw

A "manó" működésének definíciója szerint virtuális tagfüggvény esetében mindig abban az osztályban definiált tagfüggvény hívjuk meg, amilyen osztállyal definiáltuk az üzenet célobjektumát. Indirekt üzenetküldés esetén ez a szabály azt jelenti, hogy a megcímzett objektum tényleges típusa alapján kell a virtuális függvényt kiválasztani. (Indirekt üzenetküldés a példában az sp[i]->Draw( ) utasításban szerepel.)

Az összehasonlítás végett nem érdektelen a nem virtuális Draw esete sem. Nem virtuális függvények esetén a meghívandó függvényt a fordítóprogram aszerint választja ki, hogy az üzenetet fogadó objektum, illetve az azt megcímző mutató milyen típusú. Felhívjuk a figyelmet arra, hogy lényeges különbség a virtuális és nem virtuális esetek között csak indirekt, azaz mutatón keresztüli címzésben van, hiszen nem virtuális függvénynél a mutató típusa, míg virtuálisnál a megcímzett tényleges objektum típusa a meghatározó. (Ha az objektum saját magának üzen, akkor ezen szabály érvényesítésénél azt úgy kell tekinteni mintha saját magának indirekt módon üzenne.) Ennek megfelelően az sp[0]->Draw(), mivel az sp[0] Shape* típusú, de Line objektumra mutat, virtuális Draw esetében a Line::Draw-t, míg nem virtuális Draw esetében a Shape::Draw-t hívja meg. Ennek a jelenségnek messzemenő következményei vannak. Az a tény, hogy egy mutató ténylegesen milyen típusú objektumra mutat általában nem deríthető ki fordítási időben. A mintaprogramunkban például a bemeneti adatok függvényében rendelhetjük az sp[0]-hoz a &rect-t és az sp[1]-hez a &line-t vagy fordítva, ami azt jelenti, hogy az sp[i]->Draw()-nál a tényleges Draw kiválasztása is a bemeneti adatok függvénye. Ez azt jelenti, hogy a virtuális tagfüggvény kiválasztó mechanizmusnak, azaz a "manónknak", futási időben kell működnie. Ezt **késői összerendelésnek (late binding)** vagy **dinamikus kötésnek (dynamic binding)** nevezzük.

Térjünk vissza a nem virtuális esethez. Mint említettük, nem virtuális tagfüggvények esetében is az alaposztályban definiált tagfüggvények a származtatás során átdefiniálhatók. Így a line.Draw ténylegesen a Line::Draw-t jelenti nem virtuális esetben is.

A nem virtuális esetben a line.Move és shape.Move sorok értelmezéséhez elevenítsük fel a C++ nyelvről C-re fordító konverterünket. A Shape::Draw és Shape::Move közönséges tagfüggvények, amelyet a 6.3.3. fejezetben említett szabályok szerint a következő C program szimulál:

```
struct Shape { int x, y, col };        // Shape adattagjai

void Draw_Shape(struct Shape * this){} // Shape::Draw

void Move_Shape(struct Shape * this,   // Shape :: Move
                int dx, int dy ) {
   int cl = this -> col;
   this -> col = BACKGROUND;
   Draw_Shape( this );
   this -> x += dx;  this -> y += dy;
   this -> col = cl;
   Draw_Shape( this );
}
```

Tekintve, hogy a származtatás során a Shape::Move-t nem definiáljuk felül, ez marad érvényben a Line osztályban is. Tehát mind a shape.Move, mind pedig a line.Move (nem virtuális Draw esetén) a Shape::Move metódust (azaz a Move_Shape függvényt) hívja meg, amely viszont a Shape::Draw-t (azaz a Draw_Shape-t) aktivizálja.

A virtuális függvények fontossága miatt szánjunk még egy kis időt a működés magyarázatára. Tegyük fel, hogy van egy A alaposztályunk és egy B származtatott osztályunk, amelyben az alaposztály f függvényét újradefiniáltuk.

```
class A {
   public:
      void f( );  // A::f
   };

   class B : public A {
   public:
      void f( );  // B::f
};
```

Az objektumorientált programozás alapelve szerint, egy üzenetre lefuttatott metódust az célobjektum típusa és az üzenet neve (valamint az átadott paraméterek típusa) alapján kell kiválasztani. Tehát ha definiálunk egy A típusú a objektumot és egy B típusú b objektumot, és mindkét objektumnak f üzenetet küldünk, akkor azt várnánk el, hogy az a objektum esetében az A::f, míg a b objektumra a B::f tagfüggvény aktivizálódik. Vannak egyértelmű esetek, amikor ezt a kívánságunkat a C++ fordító program minden további nélkül teljesíteni tudja:

```
{
   A a;
   B b;

   a.f( );  // A::f hívás
   b.f( );  // B::f hívás
}
```

Ebben a példában az a.f() A típusú objektumnak szól, mert az a objektumot az A a; utasítással definiáltuk. Így a fordítónak nem okoz gondot, hogy ide az A::f hívást helyettesítse be.

A C++ nyelvben azonban vannak olyan lehetőségek is, amikor a fordító program nem tudja meghatározni a célobjektum típusát. Ezek a lehetőségek részint az indirekt üzenetküldést, részint a objektumok által saját maguknak küldött üzeneteket foglalják magukban. Nézzük először az indirekt üzenetküldést:

```
{
   A a;
   B b;
   A *pa;

   if ( getchar() == 'i' )  pa = &a;
   else                     pa = &b;
   pa -> f( );     // indirekt üzenetküldés
}
```

Az indirekt üzenetküldés célobjektuma, attól függően, hogy a program felhasználója az i billentyűt nyomta-e le, lehet az A típusú a objektum vagy a B típusú b objektum. Ebben az esetben fordítási időben nyilván nem dönthető el a célobjektum típusa. Megoldásként két lehetőség kínálkozik:

1. Kiindulva abból, hogy a pa mutatót A* típusúnak definiáltuk, jelentse ilyen esetben a pa->f() az A::f tagfüggvény meghívását. Ez ugyan téves, ha a pa a b objektumot címzi meg, de ennél többre fordítási időben nincs lehetőségünk.

2. Bízzuk valamilyen futási időben működő mechanizmusra annak felismerését, hogy pa ténylegesen milyen objektumra mutat, és ennek alapján futási időben válasszunk A::f és B::f tagfüggvények közül.

A C++ nyelv mindkét megoldást felkínálja, melyek közül aszerint választhatunk, hogy az f tagfüggvényt az alaposztályban normál tagfüggvénynek (1. lehetőség), vagy virtuálisnak (2. lehetőség) deklaráltuk.

Hasonló a helyzet az "önmagukban beszélő" objektumok esetében is. Egészítsük ki az A osztályt egy g tagfüggvénnyel, amely meghívja az f tagfüggvényt.

```
class A {
public:
   void f( );  // A::f
   void g( ) { f( ); }
};

class B : public A {
public:
   void f( );  // B::f
};
```

A B típusú objektum változtatás nélkül örökli a g tagfüggvényt és újradefiniálja az f-et. Ha most egy B típusú objektumnak küldenénk g üzenetet, akkor az saját magának, azaz az eredeti g üzenet célobjektumának küldene f üzenetet. Mivel az eredeti üzenet célja B típusú, az lenne természetes, ha ekkor a B::f hívódna meg. A tényleges célobjektum típusának felismerése azonban nyilván nem végezhető el fordítási időben. Tehát vagy lemondunk erről a szolgáltatásról és az f tagfüggvényt normálnak deklarálva a fordító a legkézenfekvőbb megoldást választja, miszerint a g törzsében mindig az A::f tagfüggvényt kell aktivizálni. Vagy pedig egy futási időben működő mechanizmusra bízzuk, hogy a g törzsében felismerje az objektum tényleges típusát és a meghívandó f-et ez alapján válassza ki.

A rect, line és shape objektumokat használó kis rajzolóprogram példa lehetőséget ad még egy további érdekesség bemutatására. Miként a programsorok megjegyzéseiben szereplő sírásra görbülő szájú figurák is jelzik, nem túlzottan szerencsés egy Shape típusú objektum (shape) létrehozása, hiszen a Shape osztályt kizárólag azért definiáltuk, hogy különböző geometriai alakzatok közös tulajdonságait "absztrahálja", de ilyen objektum ténylegesen nem létezik. Ezt már az is jelezte, hogy a Draw definíciója során is csak egy üres törzset adhattunk meg. (A Shape osztályban a Draw függvényre a virtuáliskénti deklarációjához és a Move-ban való szerepeltetése miatt volt szükség.) Ha viszont már van ilyen osztály, akkor az ismert lehetőségeinkkel nem akadályozhatjuk meg, hogy azt objektumok "gyártására" is felhasználjuk. Azon felismerésre támaszkodva, hogy az ilyen "absztrahált alaposztályoknál" gyakran a virtuális függvények törzsét nem lehet értelmesen kitölteni, a C++ nyelv bevezette a **tisztán virtuális tagfüggvények (pure virtual)** fogalmát. A tisztán virtuális tagfüggvényekkel jár az a korlátozást, hogy minden olyan osztály (ún. **absztrakt alaposztály**), amely tisztán virtuális tagfüggvényt tartalmaz, vagy átdefiniálás nélkül örököl, nem használható objektum definiálására, csupán az öröklődési lánc felépítésére alkalmazható. Ennek megfelelően a Shape osztály javított megvalósítása:

```
class Shape {    // absztrakt: van tisztán virtuális tagfügg.
protected:
   int x, y, col;
public:
      Shape( int x0, int y0, int col0 )
         { x = x0; y = y0; col = col0; }
   void SetColor( int c ) { col = c; }
   void Move( int dx, int dy );
   virtual void Draw( ) = 0;  // tisztán virtuális függv.
};
```

Mivel a C++ nyelv nem engedi meg, hogy absztrakt alaposztályt használjunk fel objektumok definiálására, a javított Shape osztály mellett a kifogásolt

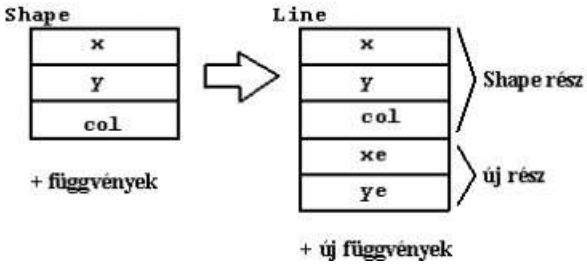
```
Shape shape;
```

sor fordítási hibát fog okozni.

7.2. 6.7.2. Az egyszerű öröklődés implementációja (nincs virtuális függvény)

Idáig az öröklődést mint az újabb tulajdonságok hozzávételét, a virtuális függvényeket pedig mint egy misztikus manót magyaráztuk. Itt a legfőbb ideje, hogy megvizsgáljuk, hogy a C++ fordító miként valósítja meg ezeket az eszközöket.

Először tekintsük a virtuális függvényeket nem tartalmazó esetet. A korábbi C++-ról C-re fordító (6.3.3. fejezet) analógiájával élve, az osztályokból az adattagokat leíró struktúra definíciók, míg a műveletekből globális függvények keletkeznek. Az öröklődés itt csak annyi újdonságot jelent, hogy egy származtatással definiált osztály attribútumaihoz olyan struktúra tartozik, ami az új tagokon kívül a szülőnek megfelelő struktúrát is tartalmazza (az pedig az ő szülőjének az adattagjait, azaz végül is az összes ős adattagjai jelen lesznek). A már meglévő függvényekhez pedig hozzáadódnak az újonnan definiáltak. Ennek egy fontos következménye az, hogy ránézve egy származtatott osztály alapján definiált objektum memóriaképére (pl. Line), annak első része megegyezik az alaposztály objektumainak (Shape) memóriaképével, azaz ahol egy Shape típusú objektumra van szükségünk, ott egy Line objektum is megteszi. Ezt a tulajdonságot nevezzük **fizikai kompatibilitásnak.** A tagfüggvény újradefiniálás nem okoz név ütközést, mert mint láttuk, a névben azon osztály neve is szerepel ahol a tagfüggvényt definiáltuk.

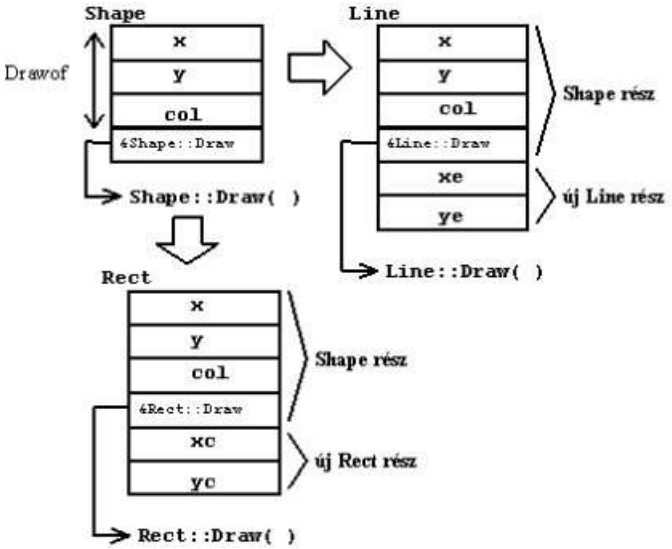


6.19. ábra: Az öröklés az öröklött és az új adattagokat összefűzi.

7.3. 6.7.3. Az egyszerű öröklődés implementációja (van virtuális függvény)

Virtuális függvények esetén az öröklődés kissé bonyolultabb. Abban az osztályban ahol először definiáltuk a virtuális függvényt az adattagok kiegészülnek a virtuális függvényekre mutató pointerrel. Ezt a mutatót az objektum keletkezése során mindig arra a függvényre állítjuk, ami megfelel az adott objektum típusának. Ez a folyamat az objektum konstruktorának a programozó által nem látható részében zajlik le.

Az öröklődés során az új adattagok, esetlegesen új virtuális függvények ugyanúgy egészítik ki az alaposztály struktúráját mint a virtuális tagfüggvényeket nem tartalmazó esetben. Ez azt jelenti, hogy ha egy alaposztály a benne definiált virtuális tagfüggvény miatt tartalmaz egy függvény címet, akkor az összes belőle származtatott osztályban ez a függvény cím adattag megtalálható. Sőt, az adattagok kiegészítéséből az is következik, hogy a származtatott osztályban a szülőtől örökölt adattagok és virtuális tagfüggvény mutatók pontosan ugyanolyan relatív elhelyezkedésűek, azaz a struktúra kezdetétől pontosan ugyanolyan eltolással (offset) érhetők el mint az alaposztályban. A származtatott osztálynak megfelelő struktúra eleje az alaposztályéval megegyező szerkezetű (6.20. ábra).



6.20. ábra: A virtuális függvények címe az adattagok között szerepel.

Alapvető különbség viszont, hogy ha a virtuális függvényt a származtatott osztályban újradefiniáljuk, akkor annak a függvény pointere már az új függvényre fog mutatni minden származtatott típusú objektumban. Ezt a következő mechanizmus biztosítja. Mint említettük a konstruktor láthatatlan feladata, hogy egy objektumban a virtuális függvények pointerét a megfelelő függvényre állítsa. Amikor például egy Line objektumot létrehozunk, az adattagokat és Draw függvény pointert tartalmazó struktúra lefoglalása után meghívódik a Line konstruktora. A Line konstruktora, a saját törzsének futtatása előtt meghívja a szülő (Shape) konstruktorát, amely "láthatatlan" részében a Draw pointert a Shape::Draw-ra állítja és a programozó által definiált módon inicializálja az x,y adattagokat. Ezek után indul a Line konstruktorának érdemi része, amely először a "láthatatlan" részben a Draw mezőt a Line::Draw-ra állítja, majd lefuttatja a programozó által megadott kódrészt, amely értéket ad az xe,ye adattagoknak.

Ezek után világos, hogy egy Shape objektum esetében a Draw tag a Shape::Draw függvényre, egy Line típusú objektumban a Line::Draw-ra, míg egy Rect objektumnál a Rect::Draw függvényre fog mutatni.

A virtuális függvény aktivizálását a fordító egy indirekt függvényhívássá alakítja át. Mivel a függvénycím minden származtatott osztályban ugyanazon a helyen van mint az alaposztályban, ez az indirekt hívás független az objektum tényleges típusától. Ez ad magyarázatot az (sp[0]->Draw()) működésére. Ha tehát a Draw() virtuális függvény mutatója az adatmezőket tartalmazó struktúra kezdőcímétől Drawof távolságra van, az sp[i]->Draw() virtuális függvényhívást a következő C programsor helyettesítheti:

```
( *((char *)sp[i] + Drawof) ) ( sp[i] );
```

A paraméterként átadott sp[i] változó a this pointert képviseli.

Most nézzük a line.Move() függvényt. Mivel a Move a Line-ban nincs újradefiniálva a Shape::Move aktivizálódik. A Shape::Move tagfüggvénybe szereplő Draw hívást, amennyiben az virtuális, a fordító this->Draw()-ként értelmezi. A Shape tagfüggvényeiből tehát egy C++ -> C fordító az alábbi sorokat állítaná elő (a Constr_Shape a konstruktorból keletkezett függvény):

```
struct Shape { int x, y, col; (void * Draw)( ); };

void Draw_Shape( struct Shape * this ) { }

void Move_Shape(struct Shape* this, int dx, int dy ) {
   int cl = this -> col;
   this -> col = BACKGROUND;
   this -> Draw( this );
   this -> x += dx; this -> y += dy;
   this -> col = cl;
   this -> Draw( this );
}
```

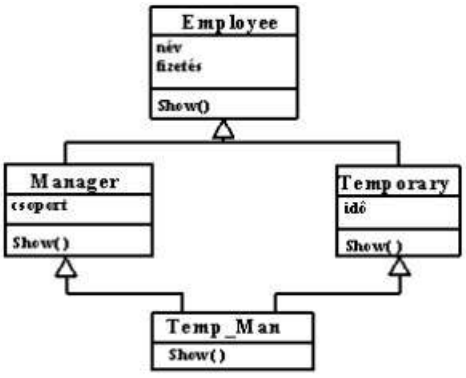
```
Constr_Shape(struct Shape * this, int x0,int y0,int col0) {
   this -> Draw = Draw_Shape;
   this -> x = x0;
   this -> y = y0;
   this -> col = col0;
}
```

Mivel a line.Move(x,y) hívásból egy Move_Shape(&line,x,y) utasítás keletkezik, a Move_Shape belsejében a this pointer (&line) Line típusú objektumra fog mutatni, ami azt jelenti, hogy a this->Draw végül is a Line::Draw-t (Draw_Line-t) aktivizálja.

Végezetül meg kell jegyeznünk, hogy a tárgyalt módszer a virtuális függvények implementációjának csak az egyik lehetséges megoldása. A gyakorlatban ezenkívül elterjedten alkalmazzák azt az eljárást is, amikor az objektumok nem tartalmazzák az összes virtuális függvény pointert, csupán egyetlen mutatót, amely az osztály virtuális függvényeinek pointereit tartalmazó táblázatra mutat. Ilyen táblázatból annyi példány van, ahány (virtuális függvénnyel is rendelkező) osztály szerepel a programban. A módszer hátránya az ismertetett megoldáshoz képest, hogy a virtuális függvények aktivizálása kétszeres indirekciót igényel (első a táblázat elérése, második a táblázatban szereplő függvény pointer alapján a függvény hívása). A módszer előnye, hogy alkalmazásával, nagyszámú, sok virtuális függvényt használó objektum esetén, jelentős memória megtakarítás érhető el.

7.4. 6.7.4. Többszörös öröklődés (Multiple inheritence)

Miként az élőlények esetében is, az öröklődés nem kizárólag egyetlen szálon futó folyamat (egy gyereknek tipikusan egynél több szülője van). Például egy irodai alkalmazottakat kezelő problémában szerepelhetnek alkalmazottak (Employee), menedzserek (Manager), ideiglenes alkalmazottak (Temporary) és ideiglenes menedzserek (Temp_Man) is. A menedzserek és ideiglenes alkalmazottak nyilván egyben alkalmazottak is, ami egy szokványos egyszeres öröklődés. Az ideiglenes menedzserek viszont részint ideiglenes alkalmazottak, részint menedzserek (és ezeken keresztül persze alkalmazottak is), azaz tulajdonságaikat két alaposztályból öröklik.



6.21. ábra: Többszörös öröklés.

Az ilyen **többszörös öröklődést** hívjuk idegen szóval "**multiple inheritance**"-nek.

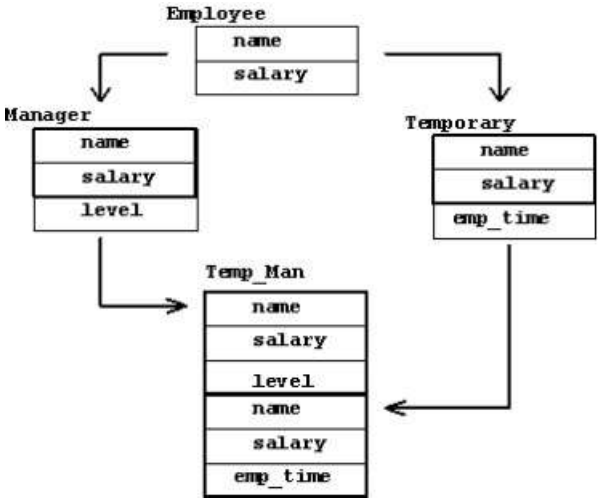
Most tekintsük a többszörös öröklés C++-beli megvalósítását. A többszörös öröklődés szintaktikailag nem jelent semmi különösebb újdonságot, csupán vesszővel elválasztva több alaposztályt kell a származtatott osztály definíciójában felsorolni. Az öröklődés publikus illetve privát jellegét osztályonként külön lehet megadni. Az irodai hierarchia tehát a következő osztályokkal jellemezhető.

```
class Employee {            // alaposztály
protected:
   char    name[20];        // név
   long    salary;         // kereset
```

```
public:
   Employee( char * nm, long sl )
      { strcpy( name, nm ); salary = sl; }
};
//=====  Manager = Employee + ... =====
class Manager : public Employee {
   int     level;
public:
   Manager( char * nam, long sal, int lev )
   : Employee( nam, sal ) { level = lev; }
};
//=====  Temporary = Employee + ... =====
class Temporary : public Employee {
   int emp_time;
public:
   Temporary( char * nam, long sal, int time );
   : Employee( nam, sal ) { emp_time = time; }
};

//=====  Temp_man = Manager + Temporary + ... =====
class Temp Man : public Manager, public Temporary {
public:
   Temp_Man(char* nam, long sal, int lev, int time)
   : Manager( nam, sal, lev ),
      Temporary( nam, sal, time ) { }
};
```

Valójában ez a megoldás egy időzített bombát rejt magában, melyet könnyen felismerhetünk, ha az egyszeres öröklődésnél megismert, és továbbra is érvényben maradó szabályok alapján megrajzoljuk az osztályok memóriaképét (6.22. ábra)

Az Employee adattagjainak a kiegészítéseként a Manager osztályban a level, a Temporary osztályban pedig az emp_time jelenik meg. A Temp_Man, mivel két osztályból származtattuk (a Manager-ből és Temporary-ból), mindkét osztály adattagjait tartalmazza, melyhez semmi újat sem tesz hozzá. Rögtön feltűnik, hogy a name és salary adattagok a Temp_Man struktúrában kétszer szerepelnek, ami nyilván nem megengedhető, hiszen ha egy ilyen objektum name adattagjára hivatkoznánk, akkor a fordító nem tudná eldönteni, hogy pontosan melyikre gondolunk.



6.22. ábra: Többszörös öröklésnél az alaposztályhoz több úton is eljuthatunk, így az alaposztály adattagjai többször megjelennek a származtatott osztályban.

A probléma, miként az az ábrán is jól látható, abból fakad, hogy az öröklődési gráfon a Temp_Man osztályból az Employee két úton is elérhető, így annak adattagjai a származtatás végén kétszer szerepelnek.

Felmerülhet a kérdés, hogy a fordító miért nem vonja össze az így keletkezett többszörös adattagokat. Ennek több oka is van. Egyrészt a Temp_Man származtatásánál a Manager és Temporary osztályokra hivatkozunk,

nem pedig az Employee osztályra, holott a problémát az okozza. Így az ilyen problémák kiküszöbölése a fordítóra jelentős többlet terhet tenne. Másrészt a nevek ütközése még önmagában nem jelent bajt. Például ha van két teljesen független osztályunk, A és B, amelyek ugyanolyan x mezővel rendelkeznek, azokból még származtathatunk újabb osztályt:

```
class A {                    class B {
protected:                   protected:
   int x;                       int x;
};                           };

      class C : public A, public B {
         int f( ) { x = 3; x = 5; }  // többértelmű
      };
```

Természetesen továbbra is gondot jelent, hogy az f függvényben szereplő x tulajdonképpen melyik a kettő közül. A C++ fordítók igen érzékenyek az olyan esetekre, amikor valamit többféleképpen is lehet értelmezni. Ezeket jellemzően sehogyan sem értelmezik, hanem fordítási hibát jeleznek. Így az f függvény fenti definíciója is hibás. A **scope operátor** felhasználásában azonban a többértelműség megszüntethető, így teljesen szabályos a következő megoldás:

```
int f( ) { A :: x = 3; B :: x = 5; }
```

Végére hagytuk az azonos nevű adattagok automatikus összevonása elleni legsúlyosabb ellenvetést. Idáig többször büszkén kijelentettük, hogy az öröklődés során az adatok struktúrája úgy egészül ki, hogy (egyszeres öröklődés esetén) az új struktúra kezdeti része kompatibilis lesz az alaposztálynak megfelelő memóriaképpel. Többszörös öröklődés esetén pedig a származtatott osztályhoz tartozó objektum memóriaképének lesznek olyan részei, melyek az alaposztályoknak megfelelő memóriaképpel rendelkeznek. A kompatibilitás jelentőségét nem lehet eléggé hangsúlyozni. Ennek következménye az, hogy ahol egy alaposztályhoz tartozó objektumot várunk, oda a belőle származtatott osztály objektuma is megfelel (kompatibilitás), és a virtuális függvény hívást feloldó mechanizmus is erre a tulajdonságra épül. A nevek alapján végzett összevonással éppen ezt a kompatibilitást veszítenénk el.

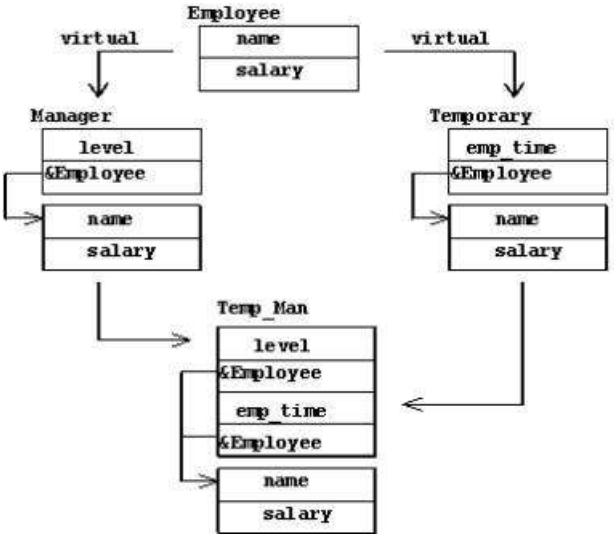
Az adattagok többszöröződési problémájának tényleges megoldása a **virtuális bázis osztályok** bevezetésében rejlik. Annál az öröklődésnél, ahol fennáll a veszélye annak, hogy az alaposztály a későbbiekben az öröklődési gráfon történő többszörös elérés miatt megsokszorozódik, az öröklődést virtuálisnak kell definiálni (ez némi előregondolkodást igényel). Ennek alapvetően két hatása van. Az alaposztály (Employee) adattagjai nem épülnek be a származtatott osztályok (Manager) adattagjai elé, hanem egy független struktúraként jelennek meg, melyet Manager tagfüggvényeiből egy mutatón keresztül érhetünk el. Természetesen mindebből a C++ programozó semmit sem vesz észre, az adminisztráció minden gondját a fordítóprogram vállalja magára. Másrészt az alaposztály konstruktorát nem az első származtatott osztály konstruktora fogja meghívni, hanem az öröklődés lánc legvégén szereplő osztály konstruktora (így küszöböljük ki azt a nehézséget, hogy a többszörös elérés a konstruktor többszöri hívását is eredményezné).

Az irodai hierarchia korrekt megoldása tehát:

```
class Manager : virtual public Employee { .... };
class Temporary : virtual public Employee { .... };
class Temp_Man : public Manager, public Temporary {
public:
   Temp_Man(char* nam, long sal, int lev, int time )
   : Employee(nam, sal), Manager(NULL, 0L, lev),
     Temporary(NULL, 0L, time) { }
};
```

Az elmondottak szerint a memóriakép virtuális öröklődés esetében a 6.23. ábrán látható módon alakul.

Természetesen a többszörös öröklődést megvalósító Temp_Man, mivel itt az öröklődés nem virtuális, a korábbihoz teljesen hasonlóan az alaposztályok adatmezőit rakja egymás után. A különálló Employee részt azonban nem ismétli meg, hanem a megduplázódott mutatókat ugyanoda állítja. Ily módon sikerült a memóriakép kompatibilitását garantálni, és azzal, hogy a mutatók többszöröződnek a tényleges adattagok helyett, a name és salary mezők egyértelműségét is biztosítottuk. Az indirekció virtuális függvényekhez hasonló léte magyarázza az elnevezést (virtuális alaposztály).



6.23. ábra: A többszörös öröklés adattag többszörözésének elkerülése virtuális bázis-osztályok alkalmazásával.

7.5. 6.7.5. A konstruktor láthatatlan feladatai

A virtuális függvények kezelése során az egyes objektumok inicializálásának ki kell térnie az adattagok közé felvett függvénycímek beállítására is. Szerencsére ebből a programozó semmit sem érzékel. A mutatók beállítását a C++ fordítóprogram vállalja magára, amely szükség esetén az objektumok konstruktoraiba elhelyezi a megfelelő, a programozó számára láthatatlan utasításokat.

Összefoglalva egy konstruktor a következő feladatokat végzi el a megadott sorrendben:

1. A virtuális alaposztály(ok) konstruktorainak hívása, akkor is, ha a virtuális alaposztály nem közvetlen ős.

2. A közvetlen, nem-virtuális alaposztály(ok) konstruktorainak hívása.

3. A saját rész konstruálása, amely az alábbi lépésekből áll:

   - a virtuálisan származtatott osztályok objektumaiban egy mutatót kell beállítani az alaposztály adattagjainak megfelelő részre.

   - ha az objektumosztályban van olyan virtuális függvény, amely itt új értelmet nyer, azaz az osztály a virtuális függvényt újradefiniálja, akkor az annak megfelelő mutatókat a saját megvalósításra kell állítani.

   - A tartalmazott objektumok (komponensek) konstruktorainak meghívása.

4. A konstruktor programozó által megadott részeinek végrehajtása.

Egy objektum a saját konstruktorának futtatása előtt meghívja az alaposztályának konstruktorát, amely amennyiben az alaposztályt is származtattuk a következő alaposztály konstruktorát. Ez azt jelenti, hogy egy öröklési hierarchia esetén a konstruktorok végrehajtási sorrendje megfelel a hierarchia felülről-lefelé történő bejárásának.

7.6. 6.7.6. A destruktor láthatatlan feladatai

A destruktor a konstruktor inverz műveleteként a konstruktor lépéseit fordított sorrendben "közömbösíti":

1. A destruktor programozó által megadott részének a végrehajtása.

2. A komponensek megszüntetése a destruktoraik hívásával.

3. A közvetlen, nem-virtuális alaposztály(ok) destruktorainak hívása.

4. A virtuális alaposztály(ok) destruktorainak hívása.

Mivel a destruktorban először a saját törzset futtatjuk, majd ezt követi az alaposztály destruktorának hívása, a destruktorok hívási sorrendje az öröklési hierarchia alulról-felfelé történő bejárását követi. A destruktor programozó által megadott részében akár virtuális tagfüggvényeket is hívhatunk, melyeket a hierarchiában az osztály alatt lévők átdefiniálhattak. Igenám, de ezek az átdefiniált tagfüggvények olyan, az alsóbb szinten definiált attribútumokra hivatkozhatnak, amit az alsóbb szintű destruktorok már "érvénytelenítettek". Ezért a destruktorok láthatatlan feladataihoz tartozik a virtuális függvénymutatók visszaállítása is.

7.7. 6.7.7. Mutatók típuskonverziója öröklődés esetén

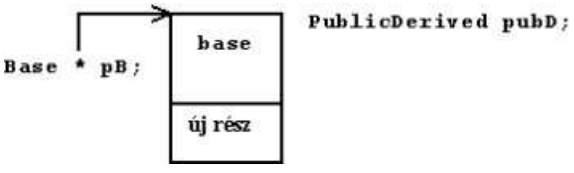
Korábban felhívtuk rá a figyelmet, hogy az öröklődés egyik fontos következménye az alaposztályok és a származtatott osztályok objektumainak egyirányú kompatibilitása. Ez részben azt jelenti, hogy egy származtatott osztály objektumának memóriaképe tartalmaz olyan részt (egyszeres öröklődés esetén az elején), amely az alaposztály objektumainak megfelelő, azaz ránézésre a származtatott osztály objektumai az alaposztály objektumaira hasonlítanak (fizikai kompatibilitás). Ezenkívül az analitikus öröklődés szabályainak alkalmazásával kialakított publikus öröklődés esetén (privátnál nem!) a származtatott osztály objektumai megértik az alaposztály üzeneteit és ahhoz hasonlóan reagálnak ezekre. Vagyis az egyirányú kompatibilitás az objektumok viselkedésére is teljesül (viselkedési kompatibilitás). Az alap és származtatott osztályok objektumai mégsem keverhetők össze közvetlenül, hiszen azok a származtatott osztály új adattagjai illetve új virtuális tagfüggvényei miatt eltérő mérettel (memóriaigénnyel) bírnak. Ezen könnyen túl tudjuk tenni magunkat, ha az objektumokat címeik segítségével, tehát indirekt módon érjük el, hiszen a mutatók fizikailag mindig ugyanannyi helyet foglalnak attól függetlenül, hogy ténylegesen milyen típusú objektumokra mutatnak.

Ezért különösen fontos a mutatók típuskonverziójának a megismerése és korrekt felhasználása öröklődés esetén. A típuskonverzió bevetésével a kompatibilitásból fakadó előnyöket kiaknázhatjuk (lásd 6.7.8. fejezetben tárgyalt heterogén szerkezeteket), de gondatlan alkalmazás mellett időzített bombákat is elhelyezhetünk a programunkban.

Tegyük fel, hogy van három osztályunk: egy alaposztály, egy publikus és egy privát módon származtatott osztály:

```
class Base { .... };

class PublicDerived : public Base { .... };

class PrivateDerived: private Base { .... };
```

Vizsgáljuk először az ún. **szűkítő** irányt, amikor a származtatott osztály típusú mutatóról az alaposztály mutatójára konvertálunk. A memóriakép kompatibilitása nem okoz gondot, mert a származtatott osztály objektumában a memóriakép kezdő része az alaposztályénak megfelelő:



6.24. ábra: Szűkítő típuskonverzió.

Publikus öröklődésnél a viselkedés kompatibilitása is rendben van, hiszen miként a teljes objektumra, az alaposztályának megfelelő részére is, az alaposztály üzenetei végrehajthatók. Ezért az ilyen jellegű mutatókonverzió olyannyira természetes, hogy a C++ még explicit konverziós operátor (cast operátor) használatát sem követeli meg:

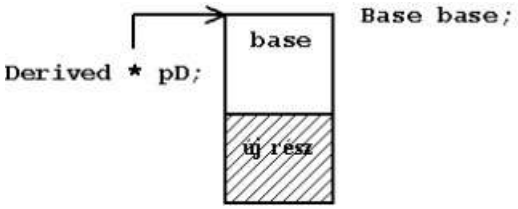
```
PublicDerived pubD;  // pubD kaphatja a Base üzeneteit
```

```
Base * pB = &pubD;  // nem kell explicit típuskonverzió
```

Privát öröklődésnél a viselkedés kompatibilitása nem áll fenn, hiszen ekkor az alaposztály publikus üzeneteit a származtatott osztályban letiltjuk. A szűkítés után viszont egy alaposztályra hivatkozó címünk van, ami azt jelenti, hogy ily módon mégiscsak elérhetjük az alaposztály letiltott üzeneteit. Ez nyilván veszélyes, hiszen bizonyára nem véletlenül tiltottuk le az alaposztály üzeneteit. A veszély jelzésére az ilyen jellegű átalakításokat csak explicit típuskonverziós operátorral engedélyezi a C++ nyelv:

```
PrivateDerived priD;// priD nem érti a Base üzeneteit

pB = (Base *)&priD; // mégiscsak érti ->  explicit konverzió!
```

A konverzió másik iránya a **bővítés,** mikor az alap osztály objektumára hivatkozó mutatót a származtatott osztály objektumának címére szeretnénk átalakítani:



6.25. ábra: Bővítő típuskonverzió.

Az ábrát szemügyre véve megállapíthatjuk, hogy a memóriaképek kompatibilitása itt nem áll fenn. Az alaposztályt általában nem használhatjuk a származtatott osztály helyett (ezért mondtuk a kompatibilitást egyirányúnak). A mutatókonverzió után viszont olyan memóriarészeket is el lehet érni (az ábrán csíkozott), melyek nem is tartoznak az objektumhoz, amiből katasztrofális hibák származhatnak. Ezért a bővítő jellegű konverziót csak kivételes esetekben használjunk és csak akkor, ha a származtatott osztály igénybe vett üzenetei csak az alaposztály adattagjait használják. A veszélyek jelzésére, hogy véletlenül se essünk ebbe a hibába, a C++ itt is megköveteli az explicit konverziós operátor használatát:

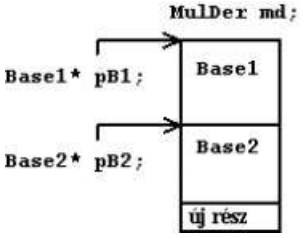
```
Base base;

Derived *pD = (Derived *) &base;

// nem létező adattagokat lehet elérni
```

Az elmondottak többszörös öröklődés esetén is változatlanul érvényben maradnak, amit a következő osztályokkal demonstráljuk:

```
class Base1{ .... };

class Base2{ .... };

class MulDer : public Base1, public Base2 {....};
```

Tekintsük először a szűkítő konverziót!

```
MulDer md;

Base1 *pb1 = &md;

Base2 *pb2 = &md; // típuskonverzió = mutató módosítás!
```

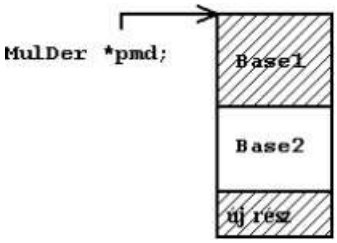


6.26. ábra: Szűkíktő típuskonverzió többszörös öröklés esetén.

Mint tudjuk, többszörös öröklődés esetén csak az egyik (általában az első) alaposztályra biztosítható az a tulajdonság, hogy az alaposztálynak megfelelő adattagok a származtatott osztálynak megfelelő adattagok kezdő részében találhatók. A többi alaposztályra csak az garantálható, hogy a származtatott osztály objektumaiban lesz olyan rész, ami ezekkel kompatibilis (ez a 6.26. ábrán is jól látható). Tehát amikor a példánkban Base2* típusra konvertálunk a mutató értékét is módosítani kell. Szerencsére a fordítóprogram ezt automatikusan elvégzi, melynek érdekes következménye, hogy C++-ban a mutatókonverzió esetlegesen megváltoztatja a mutató értékét.

Bővítő konverzió esetén, a mutató értékét a fordító szintén korrekt módon átszámítja. Természetesen a nem létező adattagok elérése továbbra is veszélyt jelent, ezért bővítés esetén többszörös öröklődéskor is explicit konverziót kell alkalmazni:

```
Base2  base2;

MulDer *pmd = (MulDer *) &base2; //
```



6.27. ábra: Bővítő típuskonverzió többszörös öröklés esetén.

7.8. 6.7.8. Az öröklődés alkalmazásai

Az öröklődés az objektum orientált programozás egyik fontos, bár gyakran túlságosan is előtérbe helyezett eszköze. Az öröklődés használható a fogalmi modellben lévő általánosítás-specializáció jellegű kapcsolatok kifejezésére, és a kód újrafelhasználásának hatékony módszereként is. Mint mindennel, az öröklődéssel is vissza lehet élni, amely áttekinthetetlen, kibogozhatatlan programot és misztikus hibákat eredményezhet. Ezért fontos, hogy az öröklődést fegyelmezetten, és annak tudatában használjuk, hogy pontosan mit akarunk vele elérni és ennek milyen mellékhatásai lehetnek. Az alábbiakban átfogó képet adunk az öröklődés ajánlott és kevésbé ajánlott felhasználási módozatairól.

Analitikus öröklődés

Az analitikus öröklődés, amikor a fogalmi modell szerint két osztály egymás általánosítása, illetve specializációja, a legkézenfekvőbb felhasználási mód. Ez nem csupán a közös részek összefogásával csökkenti a programozói munkát, hanem a fogalmi modell pontosabb visszatükrözésével a kód olvashatóságát is javíthatja.

Az analitikus öröklődést gyakran *IS_A* (az egy olyan) relációnak mondják, mert az informális specifikációban ilyen igei szerkezetek (illetve ennek rokon értelmű változatai) utalnak erre a kapcsolatra. Például: *A menedzser az egy olyan dolgozó, aki saját csoporttal rendelkezik*. Bár ez a felismerési módszer gyakran jól használható, vigyázni kell vele, hiszen az analitikus öröklődésbe csak olyan relációk férnek bele, melyek az alaposztály tulajdonságait kiegészítik, de abból semmit nem vesznek el, illetve járulékos megkötéseket nem tesznek. Tekintsük a következő specifikációs részletet:

A piros-vonal az egy olyan vonal, melynek a színe születésétől fogva piros és nem változtatható meg.

Ebben a mondatban is szerepel az "az egy olyan" kifejezés, de ez nem jelent analitikus öröklődést.

Verzió kontroll - Kiterjesztés átdefiniálás nélkül

Az analitikus öröklődéshez kapcsolódik az átdefiniálás nélküli kiterjesztés megvalósítása. Ekkor nem az eredeti modellben, hanem annak időbeli fejlődése során ismerünk fel analitikus öröklődési kapcsolatokat. Például egy hallgatókat nyilvántartó, nevet és jegyet tartalmazó Student osztályt felhasználó program fejlesztése, vagy átalakítása során felmerülhet, hogy bizonyos estekben az ismételt vizsgák nyilvántartására is szükség van. Ehhez egy új hallgató osztályt kell létrehozni, melyet az eredetiből öröklődéssel könnyen definiálhatunk:

```
class Student {
      String  name;
      int     mark;
   public:
      int Mark( ) { return mark; }
   };

   class MyStudent : public Student {
      int     repeat_exam;
   public:
      int EffectiveMark( ) {return (repeat_exam ? 1 : Mark());}
   };
```

Kicsit hasonló ehhez a láncolt listák és más adatszerkezetek kialakításánál felhasznált implementációs osztályok kialakítása. Egy láncolt listaelem a tárolt adatot és a láncoló mutatót tartalmazza. A tárolt adat mutatóval történő kiegészítése öröklődéssel is elvégezhető:

```
class StudentListElem : public Student {
   StudentListElem * next;
};
```

Kiterjesztés üzenetek törlésével (nem IS_A kapcsolat)

Az átdefiniálás másik típusa, amikor műveleteket törlünk, már nem az analitikus öröklődés kategóriájába tartozik. Példaként tegyük fel, hogy egy verem (*Stack*) osztályt kell létrehoznunk. Tételezzük fel továbbá, hogy korábbi munkánkban, vagy egy rendelkezésre álló könyvtárban sikerült egy sor (*Queue*) adatstruktúrát megvalósító osztály fellelnünk, és az az ötletünk támad, hogy ezt a verem adatstruktúra megvalósításához felhasználjuk. A verem LIFO (*last-in-first-out*) szervezésű, azaz mindig az utoljára beletett elemet lehet kivenni belőle, szemben a sorral, ami FIFO (*first-in-first-out*) elven működik, azaz a legrégebben beírt elem olvasható ki belőle. A FIFO-n Put és Get műveletek végezhetők, addig a vermen Push és Pop, melyek értelme eltérő. A verem megvalósításhoz mégis felhasználható a sor, ha felismerjük, hogy a FIFO-ban tárolt elemszám nyilvántartásával, a FIFO stratégia LIFO-ra változtatható, ha egy újabb elem betétele esetén a FIFO-ból a már benn lévő elemeket egymás után kivesszük és a sor végére visszatesszük.

Fontos kiemelnünk, hogy ebben az esetben privát öröklődést kell használnunk, hiszen csak ez takarja el az eredeti publikus tagfüggvényeket. Ellenkező esetben a Stack típusú objektumokra a Put, Get is érvényes művelet lenne, ami nyilván nem értelmezhető egy veremre és felborítaná a stratégiánkat is.

```
class Queue {
   ....
public:
   void  Put( int e );
   int Get( );
};

class Stack : private Queue { // Ilyenkor privát öröklődés
   int nelem;
public:
      Stack( ) { nelem = 0; }
   void  Push( int e );
   int   Pop( ) { nelem--; return Get(); }
};

void Stack :: Push( int e ) {
   Put( e );
   for( int i = 0; i < nelem; i++ ) Put( Get( ) );
   nelem++;
}
```

Ez a megoldás, bár privát öröklődéssel teljesen jó, nem igazán javasolt. Ehelyett jobbnak tűnik az ún. **delegáció,** amikor a verem tartalmazza azt a sort, melyet a megvalósításában felhasználunk. A Stack osztály delegációval történő megvalósítása:

```
class Stack {
   Queue fifo; // delegált objektum
   int nelem;
public:
Stack( ) { nelem = 0; }
   void  Push( int e ) {
      fifo.Put( e );
      for(int i = 0; i < nelem; i++) fifo.Put( fifo.Get());
      nelem++;
   }
   int Pop( ) { nelem--; return fifo.Get(); }
};
```

Ez a megoldás fogalmilag tisztább és átláthatóbb. Nem merül fel annak veszélye, hogy véletlenül nem privát öröklődést használunk. Továbbá típus konverzióval sem érhetjük el az eltakart Put, Get függvényeket, amire privát öröklődés esetén, igaz csak explicit típuskonverzió alkalmazásával, de lehetőség van.

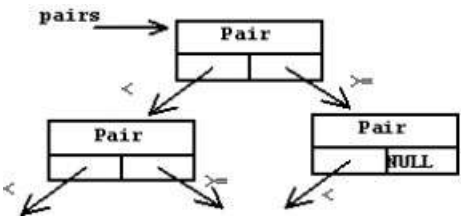
Variánsok

Az előző két kiterjesztési példa között helyezkedik el a következő, melyet általában **variánsnak** nevezünk. Egy variánsban a meglévő metódusok értelmét változtatjuk meg. Például, ha a Student osztályban a jegy kiszámítási algoritmusát kell átdefiniálni az ismételt vizsgát is figyelembe véve, a következő öröklődést használhatjuk:

```
class MyStudent : public Student {
   int  repeat_exam;
public:
   int Mark( ) { return (repeat_exam ? 1 : Student::Mark( );) }
}
```

A dolog rejt veszélyeket magában, hiszen ez nem analitikus öröklődés, mert az új diák viselkedése nem lesz kompatibilis az eredetivel, mégis gyakran használt programozói fogás.

Egy nagyobb léptékű példa a variánsok alkalmazására a lemezmellékleten található, ahol a telefonszám átirányítási feladat megoldásán (6.6. fejezet) oly módon javítottunk, hogy a párokat nem tömbben, hanem bináris rendezőfában tároltuk, azaz a tároló felépítése a következőképpen alakult át:



6.28. ábra: A párok bináris fában.

Ezzel a módszerrel, az eredeti programban csupán a legszükségesebb átalakítások elvégzésével, a keresés sebességét (időkomplexitását) lineárisról ($O(n)$) logaritmikusra ($O(\log n)$) sikerült javítani.

Heterogén kollekció

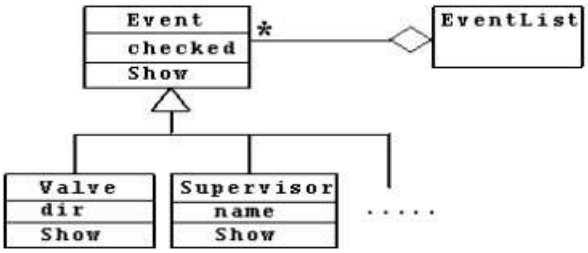
A heterogén kollekciók olyan adatszerkezetek, melyek különböző típusú és számú objektumokat képesek egységesen kezelni. Megjelenésükben hasonlítanak olyan tömbre vagy láncolt listára, amelynek elemei nem feltétlenül azonos típusúak. Hagyományos programozási nyelvekben az ilyen szerkezetek kezelése, vagy a gyűjtemény homogén szerkezetekre bontását, vagy speciális bit-szintű trükkök bevetését igényli, ami megvalósításukat bonyolulttá és igen veszélyessé teszi.

Az öröklődés azonban most is segítségünkre lehet, hiszen mint tudjuk, az öröklődés a saját hierarchiáján belül egyfajta kompatibilitást biztosít, ami azt jelenti, hogy objektumokat egységesen kezelhetünk. Az egységes kezelésen kívül eső, típus függő feladatokra viszont kiválóan használhatók a virtuális függvények, melyek automatikusan derítik fel, hogy a gyűjteménybe helyezett objektum valójában milyen típusú. (Ilyen heterogén szerkezettel a 6.7.1. fejezetben már találkoztunk, amikor Line és Rect típusú objektumokat egyetlen Shape* tömbbe gyűjtöttük össze.)

Tekintsük a következő, a folyamatirányítás területéről vett feladatot:

Egy folyamat-felügyelő rendszer a nem automatikus beavatkozásokról, mint egy szelep lezárása/kinyitása, alapjel átállítása, szabályozási algoritmus átállítása, új felügyelő személy belépése, stb. folyamatosan értesítést kap. A rendszernek a felügyelő kérésére valódi sorrendben kell visszajátszania az eseményeket, mutatva azt is, hogy mely eseményeket játszottuk vissza ezt megelőzően.

Egyelőre, az egyszerűség kedvéért, csak a szelep zárás/nyitás (Valve) és a felügyelő belépése (Supervisor) eseményeket tekintjük. A feladatanalízis alapján a következő objektummodellt állíthatjuk fel.



6.29. ábra: A folyamat-felügyelő rendszer osztálydiagramja.

Ez a modell kifejezi, hogy a szelepműveletek és felügyelő belépés közös alapja az általános esemény (Event) fogalom. A különböző események között a közös rész csupán annyi, hogy mindegyikre vizsgálni kell, hogy leolvasták-e vagy sem, ezért a leolvasást jelző attribútumot (checked) az általános eseményhez (Event) kell rendelni. Az általános esemény fogalomnak két konkrétabb változata van: a szelep esemény (Valve) és a felügyelő belépése (Supervisor). A többlet az általános eseményhez képest a szelepeseményben a szelep művelet iránya (dir), a felügyelő belépésében a felügyelő neve (name). Ezeket az eseményeket kell a fellépési sorrendben nyilvántartani, melyre az EventList gyűjtemény szolgál (itt a *List* szó inkább a sorrendhelyes tárolóra, mint a majdani programozástechnikai megvalósításra utal). Az EventList általános eseményekből (Event) áll, melyek képviselhetnek akár szelep eseményt, akár felügyelő belépést. A tartalmazási reláció mellé tett * jelzi a reláció heterogén voltát. A heterogén tulajdonság szerint az EventList tároló bármilyen az Event-ből származtatott osztályból definiált objektumot magába foglalhat. Amikor egy eseményt kiveszünk a tárolóból, akkor szükségünk van arra az információra, hogy az ténylegesen milyen típusú, hiszen különböző típusú eseményeket más módon kell kiíratni a képernyőre. Megfordítva a gondolatmenetet, a kiíratás (Show) az egyetlen művelet, amelyet a konkrét típustól függően kell végrehajtani a heterogén kollekció egyes elemeire. Ha a Show virtuális tagfüggvény, akkor az azonosítást a virtuális függvény hívását feloldó mechanizmus automatikusan elvégzi.

A Show tagfüggvényt a tanultak szerint az alaposztályban (Event) kell virtuálisnak deklarálni. Kérdés az, hogy rendelhetünk-e az Event::Show tagfüggvényhez valamilyen értelmes tartalmat. A specifikáció szerint a leolvasás tényét ki kell íratni és tárolni kell, amelyet az Event-hez tartozó változó (checked) valósít meg. Azaz, ha egy adott objektumra Show hívást adunk ki, az közvetlen vagy közvetett módon az alaposztályhoz tartozó checked változót is átírja. Ezt kétféleképpen valósíthatjuk meg. Vagy a checked változó védett (*protected*) hozzáférésű, vagy a változtatást az Event valamilyen publikus vagy védett tagfüggvényével érjük el. Adatmezők védettnek (még rosszabb esetben publikusnak) deklarálása mindenképpen kerülendő, hiszen ez kiszolgáltatja a belső implementáció részleteit és lehetőséget teremt a belső állapotot inkonzisztenssé tevő, az interfészt megkerülő változtatás elvégzésére. Tehát itt is az interfészen keresztül történő elérés a követendő. Ezért a leolvasás tényének a kiírását és rögzítését az Event::Show tagfüggvényre bízzuk.

Ezek után tekintsük a feladat megoldását egy leegyszerűsített esetben. A felügyelő eseményben (Supervisor) a név (name) attribútumot a 6.5. fejezetben tárgyalt String osztály segítségével definiáljuk. A különböző típusú elemek eltérő méretéből adódó nehézségeket úgy küszöbölhetjük ki, hogy a tényleges tárolóban csak mutatókat helyezünk el, hiszen ezek mérete független a megcímzett objektum méretétől. A virtuális függvény hívási mechanizmus miatt a mutató típusát az alaposztály (Event) szerint vesszük fel. Feltételezzük, hogy maximum 100 esemény következhet be, így a mutatók tárolására egyszerű tömböt használunk (nem akartuk az olvasót terhelni a dinamikus adatszerkezetekkel, de tulajdonképpen azt kellene itt is használni):

```
class Event {
   int  checked;
public:
      Event ( ) { checked = FALSE; }
   virtual void Show( ) { cout << checked; checked = TRUE; }
};

class Valve : public Event {
   int  dir;    // OPEN / CLOSE
public:
         Valve( int d ) { dir = d; }
   void Show ( ) {
      if ( dir )  cout << "valve OPEN";
      else        cout << "valve CLOSE";
      Event :: Show();
   }
};

class Supervisor : public Event {
   String name;
public:
        Supervisor( char * s ) { name = String( s ); }
   void Show ( ) { cout << name; Event::Show( ); }
};

class EventList {
   int   nevent;
   Event * events[100]; // mutató tömb
public:
      EventList( ) { nevent = 0; }
   void Add(Event& e) { events[ nevent++ ] = &e; }
   void List( )
      { for(int i = 0; i < nevent; i++) events[i]->Show(); }
};
```

Felhívjuk a figyelmet a Valve::Show és a Supervisor::Show tagfüggvényekben a Event::Show tagfüggvény hívásra. Itt nem alkalmazhatjuk a rövid Show hivatkozást, hiszen az a Valve::Show esetében ugyancsak a Valve::Show-ra, hasonlóképpen a Supervisor::Show-nál ugyancsak önmagára vonatkozna, amely egy végtelen rekurziót hozna létre.

Annak érdekében, hogy igazán értékelni tudjuk a virtuális függvényekre épülő megoldásunkat oldjuk meg az előző feladatot a C nyelv felhasználásával is. Heterogén szerkezetek kialakítására C-ben az első gondolatunk a *union*, vagy egy mindent tartalmazó általános struktúra alkalmazása lehetne. Ez azt jelenti, hogy a heterogén szerkezetet homogenizálhatjuk oly módon, hogy mindig maximális méretű adatstruktúrát alkalmazunk, a fennmaradó adattagokat pedig nem használjuk ki. Ezt a megközelítést pazarló jellege miatt elvetjük.

Az igazán járható, de sokkal nehezebb út igen hasonlatos a virtuális függvények alkalmazásához, csakhogy azok hiányában most mindent "kézi erővel" kell megvalósítani. A szelep és felügyelői eseményeket struktúrával (mi mással is tehetnénk?) reprezentáljuk. Ezen struktúrákat kiegészítjük egy taggal, amely azt hivatott tárolni, hogy a heterogén szerkezetben lévő elem ténylegesen milyen típusú. A típusleíró tagot mindig ugyanazon a helyen (ez itt a lényeg!), célszerűen a struktúra első tagjaként valósítjuk meg. A heterogén kollekció központi része most is egy mutatótömb lesz, amely akármilyen típusú mutatókat tartalmazhat, hiszen miután kiderítjük az általa megcímzett memóriaterületen álló típustagból a struktúra tényleges típusát, úgy is típuskonverziót (*cast*) kell alkalmazni. Éppen az ilyen esetekre találták ki az ANSI C-ben a void mutatót.

Ezek után a C megvalósítás az alábbiakban látható:

```
struct Valve {
   int    type;   // VALVE, SUPERVISOR ... 1. helyre
   BOOL   chked, dir;
};

struct Supervisor {
   int    type;   // VALVE, SUPERVISOR ... u.a. helyre
   BOOL   chked;
   char   name[30];
};

void * events[100]; // mutató tömb
int  nevent = 0;

void AddEvent( void * e ) { events[ nevent++ ] = e; }

void List( ) {
   int i;
   struct Valve * pvalv;
   struct Supervisor * psub;

   for( i = 0; i < nevent; i++ ) {
      switch ( *( (int *)events[i] ) ) {
      case VALVE:
         pvalv = (struct Valve *) events[i];
         if ( pvalv->dir ) {
            printf("v.OPEN chk %d\n", pvalv->chked );
            pvalv->chked = TRUE;
         } else ....
         break;
      case SUPERVISOR:
         psub = (struct Supervisor *)events[i];
         printf("%s chk%d", psub->name, psub->chked );
      }
   }
}
```

Mennyivel rosszabb ez mint a C++ megoldás? Először is a mutatók konvertálgatása meglehetősen bonyolulttá és veszélyessé teszi a fenti programot. Kritikus pont továbbá, hogy a struktúrákban a type adattag ugyanoda kerüljön. A különbség akkor válik igazán döntővé, ha megnézzük, hogy a program egy későbbi módosítása mennyi fáradsággal és veszéllyel jár. Tegyük fel, hogy egy új eseményt (pl. alapjel állítás, azaz ReferenceSet) kívánunk hozzávenni a kezelt eseményekhez. C++-ban csupán az új eseménynek megfelelő osztályt kell létrehozni és annak Show tagfüggvényét a megfelelő módon kialakítani. Az EventList kezelésével kapcsolatos programrészek változatlanok maradnak. Ezzel szemben a C nyelvű megoldásban először a ReferenceSet struktúrát kell létrehozni vigyázva arra, hogy a type az első helyen álljon. Majd a List függvényt jelentősen át kell gyúrni, melynek során mutató konverziókat kell beiktatni és a switch/case ágakat kiegészíteni. A C++ megvalósítás tehát csak az új osztály megírását jelenti, melyet egy elkülönült helyen megtehetünk, míg a C példa a teljes program átvizsgálásával és megváltoztatásával jár. Egy sok ezer soros, más által írt program esetében a két út különbözősége nem igényel hosszabb magyarázatot.

A C++ nyelvben a heterogén szerkezetben található objektumok típusát azonosító switch/case ágakat a virtuális függvény mechanizmussal válthatjuk ki. Minden olyan függvényt virtuálisnak kell deklarálni, amelyet a heterogén kollekcióba elhelyezett objektumoknak küldünk, ha a válasz típusfüggő. Ekkor maga a virtuális tagfüggvény kezelési mechanizmus fogja az objektum tényleges típusát meghatározni és a megfelelő reakciót végrehajtani.

Egy létező és heterogén kollekcióba helyezett objektumot természetesen meg is semmisíthetünk, melynek hatására egy destruktorhívás jön létre. Adott esetben a destruktor végrehajtása is típusfüggő, például, ha a tárolt objektumoknak dinamikusan allokált adattagjaik is vannak (lásd 6.5.1. fejezetet), vagy ha az előző feladatot úgy módosítjuk, hogy a tárolt események törölhetők, de a törléskor az esemény naplóját automatikusan ki kell írni a nyomtatóra. Értelemszerűen ekkor **virtuális destruktort-t** kell használni.

Tartalmazás (aggregáció) szimulálása

"Kifejezetten nem ajánlott" kategóriában szerepel az öröklődésnek az aggregáció megvalósításához történő felhasználása, mégis is nap mint nap találkozhatunk vele. Ennek oka elsősorban az, hogy a gépelési munkát jelentősen lerövidítheti, igaz, hogy esetleg olyan gonosz hibák elhelyezésével, melyek a későbbiekben igencsak megbosszulják magukat. Ennek illusztrálására lássunk egy autós példát:

Az autóhoz kerék és motor tartozik, és még neve is van.

Ha ezt a modellezési feladatot tisztességesen, tehát tartalmazással valósítjuk meg, a tartalmazott objektumoknak a teljes autóra vonatkozó szolgáltatásait ki kell vezetni az autó (Car) osztályra is, hiszen egy tartalmazott objektum kívülről közvetlenül nem érhető el. Ez ún. közvetítő függvényekkel történhet. Ilyen közvetítő függvény az motorfogyasztást megadó EngCons és a kerékméretet leolvasó-átíró WheelSize. Mivel ezeket a szolgáltatásokat végső soron a tartalmazott objektumok biztosítják, a közvetítő függvény nem csinál mást, mint üzenetet küld a megfelelő tartalmazott objektumnak:

```
class Wheel {
   int    size;
public:
   int&   Size( ) { return size; }
};

class Engine {
   double   consumption;
public:
   double& Consum( ) { return consumption; }
};

class Car {
   String name;
   Wheel  wheel;
   Engine engine;
public:
   void  SetName( String& n ) { name = n; }
   double& EngCons( ) {return engine.Consum();} // közvetítő
   int& WheelSize( ) {return wheel.Size();}     // közvetítő
};
```

Ezeket a közvetítő függvényeket lehet megspórolni, ha a Car osztályt többszörös öröklődéssel építjük fel, hiszen publikus öröklődés esetén az alaposztályok metódusai közvetlenül megjelennek a származtatott osztályban:

```
class Car : public Wheel, public Engine {
      String name;
   public:
      void  SetName( String& n ) { name = n; }
};

   Car volvo;
   volvo.Size() = ...      // Ez a kerék mérete :-(
```

Egy lehetséges következmény az utolsó sorban szerepel. A volvo.Size, mivel az autó a Size függvényt a keréktől örökölte, a kerék méretét adja meg, holott az a programot olvasó számára inkább magának a kocsinak a méretét jelenti. Az autó részeire és magára az autóra vonatkozó műveletek névváltoztatás nélkül összekeverednek, ami különösen más programozók dolgát nehezíti meg, illetve egy későbbi módosítás során könnyen visszaüthet.

Egy osztály működésének a befolyásolása

A következőkben az öröklődés egy nagyon fontos alkalmazási területét, az objektumok belső működésének befolyásolását tekintjük át, amely lehetővé teszi az osztálykönyvtárak rugalmas kialakítását.

Tegyük fel, hogy rendelkezésünkre áll diákok (Student) rendezett listáját képviselő osztály, amely a rendezettséget az új elem felvétele (Insert) során annak sorrendhelyes elhelyezésével biztosítja. A sorrendhelyes elhelyezéshez összehasonlításokat kell tennie a tárolt diákok között, melyeket egy összehasonlító (Compare) tagfüggvény végez el. Ha ezen osztály felhasználásával különböző rendezési szabállyal rendelkező csoportokat kívánunk létrehozni, akkor a Compare tagfüggvényt kell újradefiniálni. Az összehasonlító tagfüggvényt viszont az alaposztály tagfüggvénye (Insert) hívja, így ha az nem lenne virtuális, akkor hiába definiálnánk újra öröklődéssel a Compare-t, az alaposztály tagfüggvényei számára továbbra is az eredeti értelmezés maradna érvényben.

Virtuális összehasonlító tagfüggvény esetén a rendezési szempont, az alaposztálybeli tagfüggvények működésének a befolyásolásával, módosítható:

```
class StudentList {
   ....
   virtual int Compare(Student s1, Student s2) { return 1; }
public:
   Insert( Student s ) {....; if ( Compare(....) ) ....}
   Get( Student& s ) {....}
};

class MyStudentList : StudentList {
   int Compare( Student s1, Student s2 )
      { return s1.Mark( ) > s2.Mark( ); }
};
```

Eseményvezérelt programozás

Napjaink korszerű felhasználói felületei az ún. ablakos, eseményvezérelt felületek. Az ablakos jelző azt jelenti, hogy a kommunikáció számos egymáshoz képest rugalmasan elrendezhető, de adott esetben igen különböző célú téglalap alakú képernyőterületen, ún. ablakon keresztül történik, amelyek az asztalon szétdobált füzetek, könyvek és más eszközök egyfajta metaforáját képviselik. Az eseményvezéreltség arra utal, hogy a kommunikációs szekvenciát elsősorban nem a program, hanem a felhasználó határozza meg, aki minden elemi beavatkozás után igen sok különböző lehetőség közül választhat (ezzel szemben áll a hagyományos kialakítás, mikor a kommunikáció a program által feltett kérdésekre adott válaszokból áll). Ez azt jelenti, hogy az eseményvezérelt felhasználói felületeket minden pillanatban szinte mindenféle kezelői beavatkozásra fel kell készíteni. Mint említettük, a kommunikáció kerete az ablak, melyből egyszerre több is lehet a képernyőn, de minden pillanatban csak egyetlenegyhez, az aktív ablakhoz, jutnak el a felhasználó beavatkozásai.

A felhasználói beavatkozások az adatbeviteli (*input*) eszközökön (klaviatúra, egér) keresztül, az operációs rendszer feldolgozása után jutnak el az aktív ablakhoz. Valójában ezt úgy is tekinthetjük, hogy a felhasználó üzeneteket küld a képernyőn lévő aktív ablak objektumnak, ami erre a megfelelő metódus lefuttatásával reagál. Ennek hatására természetesen módosulhatnak az ablak belső állapotváltozói, minek következtében a későbbi beavatkozásokra történő reakció is megváltozhat. Éppen ez a belső állapot az, ami az egyes elemi kezelői beavatkozások között rendet teremt és vagy rögzített szekvenciát erőszakol ki, vagy a kezelő által megadott elemi beavatkozásokhoz a sorrend alapján tartalmat rendel.

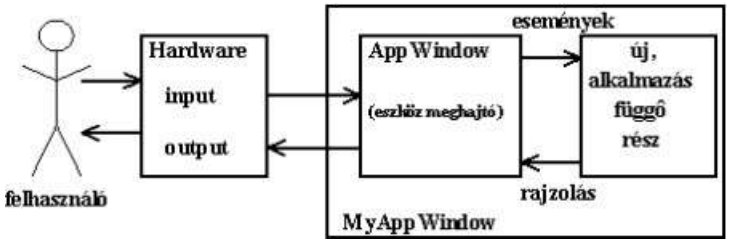
Az elemi beavatkozások (mint például egy billentyű- vagy egérgomb lenyomása/elengedése, egér mozgatása, stb.) egy része igen általános reakciót igényel. Az egér mozgatása szinte mindig a kurzor mozgatását igényli, az ablak bal-felső sarkára való dupla kattintás (*click*) pedig az ablak lezárását, stb. Más beavatkozásokra viszont ablakról ablakra alapvetően eltérően kell reagálni. Ez a tulajdonság az, ami az ablakokat megkülönbözteti egymástól. Egy szövegszerkesztő programban az egérgomb lenyomása az szövegkurzor (*caret*) áthelyezését, vagy menüből való választást jelenthet, egy rajzoló programban pedig egy egyenes szakasz erre a pontra húzását eredményezheti. A teljesen általános és egészen speciális reakciók, mint extrém esetek között léteznek átmenetek is, amikor ugyan a végső reakció alapvetően eltérő, mégis azok egy része közös. Erre jó példa a menükezelés. Egy főmenüpont kiválasztása az almenü legördülését váltja ki, az almenüben történő bóklászásunk során a kiválasztás jelzése is változik, míg a tényleges választás után a legördülő menük eltűnnek. Ez teljesen általános. Specifikusak viszont az egyes menüpontok által aktivizálható szolgáltatások, a menüelemek száma és az a szöveg ami rajtuk olvasható.

Most fordítsuk meg az információ átvitelének az irányát és tekintsük a program által a felhasználó számára biztosított adatokat, képeket, hangokat, stb. Ezek az output eszközök segítségével jutnak el a felhasználóhoz, melyek közül az ablakok kapcsán a képernyőt kell kiemelnünk (ilyenek még a nyomtató, a hangszóró, stb.). A képernyő kezelése, azon magas szintű szolgáltatások biztosítása (például egy bittérkép kirajzolása, egyeneshúzás, karakterrajzolás, stb.) igen bonyolult művelet, de szerencsére a gyakran igényelt magas szintű szolgáltatások egy viszonylag szűk körből felépíthetők (karakter, egyenes szakasz, ellipszis, téglalap, poligon rajzolása, területkitöltés színnel és mintával), így csak ezen mag egyszeri megvalósítására van szükség.

Objektumorientált megközelítésben az ablakokhoz egy osztályt rendelünk. Az említett közös vonásokat célszerű egy közös alaposztályban (AppWindow) összefoglalni, amely minden egyes felhasználói beavatkozásra valamilyen alapértelmezés szerint reagál, és az összes fontos output funkciót biztosítja. Az alkalmazásokban szereplő specifikus ablakok ennek a közös alapablaknak a származtatott változatai (legyen az osztálynév MyAppWindow). A származtatott ablakokban nyilván csak azon reakciókat megvalósító tagfüggvényeket kell újradefiniálni, melyeknek az alapértelmezéstől eltérő módon kell viselkedniük. Az *output* funkciókkal nem kell törődni a származtatott ablakban, hiszen azokat az alapablaktól automatikusan örökli.

Az alapablak (AppWindow), az alkalmazásfüggő részt megtestesítő származtatott ablak (MyAppWindow) és az input/output eszközök viszonyát a 6.30. ábra szemlélteti.

Vegyük észre, hogy a kommunikáció az új alkalmazásfüggő rész és az alapablak között kétirányú. Egyrészt az alkalmazásspecifikus reakciók végrehajtása során szükség van az AppWindow-ban definiált magas szintű rajzolási illetve output funkciókra. Másik oldalról viszont, ha egy reakciót az alkalmazás függő rész átdefiniál, akkor a fizikai eszköztől érkező üzenet hatására az annak megfelelő tagfüggvényt kell futtatni. Ez azt jelenti, hogy az alaposztályból meg kell hívni, a származtatott osztályban definiált tagfüggvényeket, melyről tudjuk, hogy csak abban az esetben lehetséges, ha az újradefiniált tagfüggvényt az AppWindow osztályban virtuálisként deklaráltuk. Ez azt jelenti, hogy minden input eseményhez tartozó reakcióhoz virtuális tagfüggvénynek kell tartoznia.



6.30. ábra: A felhasználó és az eseményvezérelt program kapcsolata.

Az AppWindow egy lehetséges vázlatos megvalósítása és felhasználása az alábbiakban látható:

```
class AppWindow {  // könyvtári objektum
....
         // input funkciók: esemény kezelők
   virtual void    MouseButtonDn( MouseEvt ) {}
   virtual void    MouseDrag( MouseEvt ) {}
   virtual void    KeyDown( KeyEvt ) {}
   virtual void    MenuCommand( MenuCommandEvt ) {}
   virtual void    Expose( ExposeEvt ) {}

         // output funkciók
   void            Show( void );
   void            Text( char *, Point );
   void            MoveTo( Point );
   void            LineTo( Point );
};


class MyWindow : public AppWindow {
   void   Expose( ExposeEvent e) { .... }
   void   MouseButtonDn(MouseEvt e) { ....; LineTo( ); }
   void   KeyDown( KeyEvt e) { ....; Text( ); .... }
};
```

Az esemény-reakcióknak megfelelő tagfüggvények argumentumai szintén objektumok, amelyek az esemény paramétereit tartalmazzák. Egy egér gomb lenyomásához tartozó információs objektum (MouseEvt) például tipikusan a következő szolgáltatásokkal rendelkezik:

```
class MouseEvt {
   ....
   public:
      Point Where( );         // a lenyomás helye az ablakban
      BOOL  IsLeftPushed( );  // a bal gomb lenyomva-e?
      BOOL  IsRightPushed( ); // a jobb gomb lenyomva-e?
};
```

8. 6.8. Generikus adatszerkezetek

Generikus adatszerkezetek alatt olyan osztályokat értünk, melyekben szereplő adattagok és tagfüggvények típusai fordítási időben szabadon állíthatók be. Az ilyen jellegű típusparaméterezés jelentőségét egy mintafeladat megoldásával világítjuk meg. Oldjuk meg tehát a következő feladatot:

A szabványos inputról diákok adatai érkeznek, melyek a diák nevéből és átlagából állnak. Az elkészítendő programnak az elért átlag szerinti sorrendben listáznia kell azon diákok nevét, akik átlaga az összátlag felett van.

A specifikáció alapján nyilvánvaló, hogy az alapvető objektum a "diák", melynek két attribútuma, neve és átlaga van. Mivel a kiírást akkor lehet elkezdeni, amikor már az összes diák adatait beolvastuk, hiszen az "összátlag" csak ekkor derül ki, meg kell oldani a diák objektumok ideiglenes tárolását. A diákok számát előre nem ismerjük, ráadásul a diákokat tároló objektumnak valamilyen szempont (átlag) szerinti rendezést is támogatnia kell. Implementációs tapasztalatainkból tudjuk, hogy ilyen jellegű adattárolást például **láncolt listával** tudunk megvalósítani, azaz a megoldásunk egyik alapvető implementációs objektuma ez a láncolt lista lesz. A láncolt listában olyan elemek szerepelnek, melyek részben a tárolt adatokat, részben a láncoló mutatót tartalmazzák. Ez viszont szükségessé teszi egy olyan objektumtípus létrehozását, amely mind a diákok adatait tartalmazza, mind pedig a láncolás képességét is magában hordozza.

A megoldásban szereplő, analitikus és implementációs objektumok ennek megfelelően a következők:

Objektum	**Típus**	**Attribútum**	**Felelősség**
diákok	*Student*	név=*name*, átlag=*average*	átlag lekérdezése=*Average( )* név lekérdezése=*Name( )*
diák listaelemek	*StudentListElem*	diák=*data*, láncoló mutató	
diák tároló	*StudentList*		új diák felvétele rendezéssel =*Insert( )* a következő diák kiolvasása =*Get( )*

A diákok név attribútumának kialakításánál elvileg élhetnénk a C programozási emlékeinkből ismert megoldással, amely feltételezi, hogy egy név maximum 30 karakteres lehet, és egy ilyen méretű karakter tömböt rendelünk hozzá. Ennél sokkal elegánsabb, ha feleleventjük a dinamikusan nyújtózkodó sztring osztály (String) előnyeit, és az ott megalkotott típust használjuk fel.

Az osztályok implementációja ezek után:

```
enum BOOL { FALSE, TRUE };

class Student {          // Student osztály
   String   name;
   double   average;
public:
      Student( char * n = NULL, double a = 0.0 )
      : name( n ) { average = a; }
   double  Average( ) { return average; }
   String& Name( ) { return name; }
};

class StudentList;   // az előrehivatkozás miatt

class StudentListElem {  // Student + láncoló pointer
   friend class StudentList;
   Student           data;
   StudentListElem * next;
public:
   StudentListElem() {} // alapértelmezésű konstruktor
   StudentListElem(Student d, StudentListElem * n)
      { data = d; next = n; }
};

class StudentList { // diákokat tároló objektum osztály
   StudentListElem  head, * current;
   int Compare( Student& d1, Student& d2 )
      { return (d1.Average() > d2.Average()); }
public:
          StudentList( ) { current = &head; head.next = 0; }
   void   Insert( Student& );
   BOOL   Get( Student& );
};
```

A fenti definíciókkal kapcsolatban érdemes néhány apróságra felhívni a figyelmet. A name a Student tartalmazott objektuma, azaz, ha egy Student típusú objektumot létrehozunk, akkor a tartalmazott name objektum is létrejön. Ez azt jelenti, hogy a Student konstruktorának hívása során a String konstruktora is lefut, ezért lehetőséget kell adni a paraméterezésére. Ezt a célt szolgálja az alábbi sor,

```
Student(char * n = NULL, double a = NULL)

: name(n) {average = a;}
```

amely az n argumentumot továbbadja a String típusú name mező konstruktorának, így itt csak az average adattagot kell inicializálni.

A másik érdekesség a saját farkába harapó kutya esetére hasonlít. A StudentList típusú objektumok attribútuma StudentListElem típusú, azaz a StudentList osztály definíciója felhasználja a StudentListElem típust, ezért a StudentList osztály definícióját meg kell hogy előzze a StudentListElem osztály. (Ne felejtsük el, hogy a C és C++ fordítók olyanok mint a hátrafelé bandukoló szemellenzős lovak, amelyek csak azon definíciókat hajlandók figyelembe venni egy adott sor értelmezésénél, amelyek az adott fájlban a megadott sor előtt találhatók.) Ennek megfelelően a StudentListElem osztályt a StudentList osztály előtt kell definiálni. A láncolt lista adminisztrációjáért felelős StudentList típusú objektumokban nyilván szükséges az egyes listaelemek láncoló mutatóinak átállítása, melyek viszont a StudentListElem típusú objektumok (privát) adattagjai. Ha el akarjuk kerülni a StudentListElem-ben a mutató leolvasását és átírását elvégző tagfüggvényeket, akkor a StudentList osztályt a StudentListElem *friend* osztályaként kell deklarálni. Ahhoz, hogy a *friend* deklarációt elvégezzük, a

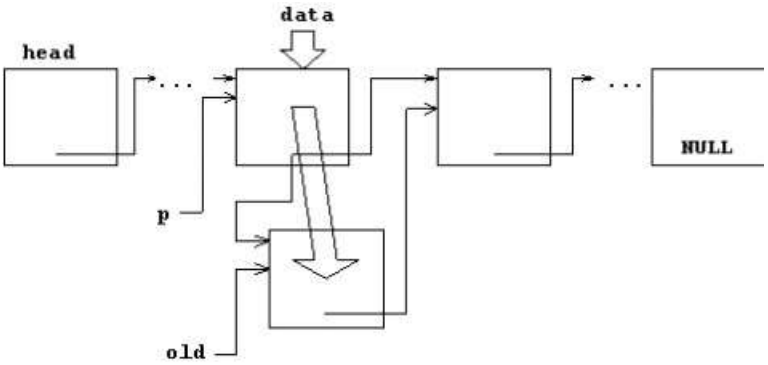
StudentListElem-ben a StudentList típusra hivatkozni kell, azaz annak definícióját a StudentListElem előtt kell elvégezni. Az ördögi kör ezzel bezárult, melynek felvágására az ún. **előredeklarációt** lehet felhasználni. Ez a funkciója a példában szereplő

```
class StudentList;
```

sornak, amely ideiglenesen megnyugtatja a fordítót, hogy a későbbiekben lesz majd ilyen nevű osztály.

Most nézzük a láncolt lista adminisztrációjával kapcsolatos bonyodalmakat. A legegyszerűbb (de kétségkívül nem a leghatékonyabb) megoldás az ún. listafej (strázsa) felhasználására épül, amely mindig egyetlen listaelemmel többet igényel, de cserébe nem kell külön vizsgálni, hogy a lista üres-e vagy sem (6.31. ábra).

A rendezés az újabb elem felvétele során (Insert) történik, így feltételezhetjük, hogy a lista minden pillanatban rendezett. Egy új elem beszúrása úgy történik, hogy a láncolt lista első elemétől (head) kezdve sorra vesszük az elemeket és összehasonlítjuk a beszúrandó elemmel (data). Amikor az összehasonlítás azt mutatja, hogy az új elemnek az aktuális listaelem (melynek címe p) elé kell kerülnie, akkor lefoglalunk egy listaelemnyi területet (melynek címe old), és az aktuális listaelem tartalmát mindenestül idemásoljuk, az új adatelemet pedig a megtalált listaelem adatelemébe írjuk, végül annak láncoló mutatóját a most foglalt elemre állítjuk.



6.31. ábra: Beszúrás egy stázsát használó láncolt listába.

Az elemek leolvasása, a minden pillanatban érvényes rendezettséget figyelembe véve, a listaelemeknek a láncoló mutatók által meghatározott bejárását igényli. Ezt és a rendezést magvalósító láncolási adminisztrációt is tartalmazó tagfüggvények implementációja az alábbi:

```
void StudentList :: Insert( Student& data ) {
   for(StudentListElem* p = &head; p->next != NULL; p=p->next)
      if ( Compare(p -> data, data) ) break;

   StudentListElem* old = new StudentListElem(p->data,p->next);
   p->data = data;
   p->next = old;
}

BOOL StudentList :: Get( Student& e ) {
   if (current->next == NULL) {
      current = &head;
      return FALSE;
   }
   e = current->data;
   current = current->next;
   return TRUE;
}
```

A Get tagfüggvénynek természetesen jeleznie kell, ha a lista végére ért, és ezért nem tud több adatot leolvasni. A következő leolvasásnál ismét a lista elejére kell állni. A lista végét vagy egy járulékos visszatérési érték vagy argumentum (a példában a függvény visszatérési értéke logikai változó ami éppen ezt jelzi) mutathatja, vagy

pedig a tényleges adatmezőt használjuk fel erre a célra, azt érvénytelen módon kitöltve. Gyakori mutatók esetén a NULL érték ilyen jellegű felhasználása.

Egy kollekcióból az elemek adott sorrend szerinti kiolvasását, melyet a példánkban a Get metódus valósít meg, **iterációnak** hívjuk. C++ programozók egyfajta szokásjog alapján erre a célra gyakran használják a függvényhívás operátor átdefiniált változatát. A következőkben ezt mutatjuk be egy olyan megvalósításban, ahol a visszatérési érték mutató, melynek a NULL értéke jelzi a lista végét.

```
Student * StudentList :: operator( ) ( ) { // függvényhívás op.
   if (current -> next == NULL) {
  current = &head; return NULL;
   }
   Student * e = &current -> data;
   current = current -> next;
   return e;
}
....     // főprogram
StudentList slist;
Student * s;
while( s = slist( ) ) {  // Iterációs folyamat
 s -> ....
}
```

Nagy nehezen létrehoztuk a feladat megvalósításához szükséges osztályokat, most már csupán ujjgyakorlat a teljes implementáció befejezése (ezt az olvasóra bízzuk). A fenti példát elsősorban azért mutattuk be, hogy le tudjunk szűrni egy lényeges tapasztalatot. A feladatmegoldás során a befektetett munka jelentős részét a többé-kevésbé egzotikus adatstruktúrák (rendezett láncolt lista) megvalósítása és az adminisztrációt végző tagfüggvények implementációja emészti fel. Mivel ezek az erőfeszítések nagyrészt függetlenek attól, hogy pontosan milyen elemeket tartalmaz a tárolónk, rögtön felmerül a kérdés, hogy az iménti munkát hogyan lehet megtakarítani a következő láncolt listát igénylő feladat megoldásánál, azaz a mostani eredményeket hogyan lehet átmenteni egy újabb implementációba, amely nem Student elemeket tartalmaz.

A fenti megoldás az általános listakezelésen kívül tartalmaz az adott alkalmazástól függő részeket is. Ezek az elnevezések (a listaelemet StudentListElem-nek, a listát StudentList-nek neveztük), a metódusok argumentumainak, az osztályok attribútumainak típusa, és az összehasonlító függvény (Compare).

Ezek alapján, ha nem diákok listáját akarjuk megvalósítani, akkor a következő transzformációs feladatokat kell elvégezni:

1. **Student név elemek cseréje** az elnevezések megfelelő kialakítása miatt.

2. **a data típusa, és argumentumtípusok cseréje**

   Ezen két lépést automatikusan az előfordító (preprocesszor) segítségével vagy egy nyelvi eszköz felhasználásával, ún. **sablonnal** (*template*) hajthatjuk végre. Nem automatikus megoldásokkal, mint a programsorok átírása, nem is érdemes foglalkozni.

3. **Compare függvény átdefiniálása**

   A Compare függvényt, amely a lista része, az implementáció átírásával, vagy öröklés felhasználásával definiálhatjuk újra. Az öröklés felhasználásánál figyelembe kell venni, hogy a lista a Compare-t az alaposztályhoz tartozó Insert metódusban hívja, tehát a Compare-nek virtuálisnak kell lennie annak érdekében, hogy az Insert is az újradefiniált változatot lássa.

8.1. 6.8.1. Generikus szerkezetek megvalósítása előfordítóval (preprocesszor)

Először az előfordító felhasználását mutatjuk be az ismertetett transzformációs lépések elvégzésére. Ennek alapeleme a C előfordítójának **név összekapcsoló makrója** (##), amely a

```
#define List( type ) type##List
```

deklaráció esetén, a List(xxx) makróhívás feloldása során az xxx helyén megadott sztringet hozzáragasztja a List szócskához.

Az általános listát leíró makrót egy *GENERIC.H* definíciós fájlba helyezzük el:

```
#define List( type )     type##List
#define ListElem( type ) type##ListElem

#define declare_list( type )                                    \
class List(type);                                               \
class ListElem( type ) {                                        \
friend class List( type );                                      \
   type                 data;                                   \
   ListElem( type ) *   next;                                   \
public:                                                         \
   ListElem(type)( ) { }                                        \
   ListElem(type)(type d, ListElem(type)* n)                    \
      { data = d; next = n; }                                   \
};

class List(type) {                                              \
   ListElem(type)  head, *current;                              \
   virtual int Compare( type& d1, type& d2 ){ return 1; }       \
public:                                                         \
      List(type)( ) { current = &head; head.next = NULL; }     \
   void  Insert( type& );                                       \
   BOOL  Get( type& );                                          \
};

#define implement_list(type)                                    \
void    List(type) :: Insert( type& data ) {                    \
   for(ListElem(type)* p =head; p->next !=NULL; p=p->next )     \
      if ( Compare(p->data, data) ) break;                      \
   ListElem(type) *old = new ListElem(type)(p->data,p->next);\
   p -> data = data;                                            \
   p -> next = old;                                             \
}
```

Szétválasztottuk a generikus osztály deklarációját és implementációját, és azokat két külön makróval adtuk meg (declare_list, implement_list). Erre azért volt szükség, mert ha több fájlból álló programot készítünk, a deklarációknak minden olyan fájlban szerepelniük kell ahol a generikus listára, illetve annak parametrizált változatára hivatkozunk. A tagfüggvény implementációk viszont pontosan egyszer jelenhetnek meg a programban. Tehát a declare_list makrót minden, a generikus listát felhasználó fájlba be kell írni, az implement_list hívását viszont csak egyetlen egybe.

A "\" jelekre azért volt szükség, mert a C előfordító a "sor végéig" tekinti a makrót, és ezzel lehet neki megmondani, hogy még a következő sor is hozzá tartozik. Ez persze azt jelenti, hogy az előfordítás után a fenti makro egyetlen fizikai sorként jelenik meg programunkban, ami a fordítási hibaüzenetek lokalizálását meglehetősen nehézzé teszi, nem is beszélve a nyomkövetésről, hiszen az C++ sorokként, azaz a teljes makrót tekintve egyetlen lépéssel történik. Ne próbáljunk az ilyen makrókba megjegyzéseket elhelyezni, mert az előző okok miatt azok meglehetősen egzotikus hibaüzeneteket eredményezhetnek.

A megoldás tehát eléggé nehézkes, de ha kész van, akkor ennek felhasználásával az előző hallgatókat tartalmazó program megírása egyszerűvé válik:

```
#include "generic.h"

class Student {
   String name;
   double average;
```

```
   ....
};
declare_list( Student )      // minden file-ban
implement_list( Student )    // csak egy file-ban

class MyStudentList : public StudentList {
   int Compare( Student& s1, Student& s2 )
      { return ( s1.Average() > s2.Average() ); }
};

void main( ) {
   MyStudentList list;
   Student st;
   ....
   list.Insert( st );
   ....
}
```

8.2. 6.8.2. Generikus szerkezetek megvalósítása sablonnal (template)

A korai C++ nyelvi implementációk használóinak a C preprocesszor által biztosított módszer nem kevés fejfájást okozott a körülményessége miatt. A generikus szerkezetekről viszont semmiképpen sem kívántak lemondani, ezért a C++ fejlődése során nyelvi elemmé tették a generikus szerkezeteket. Ezt az új nyelvi elemet nevezzük **sablonnak** (**template**), ami fordítási időben konstans kifejezéssel (típus, konstans érték, globális objektum címe, függvény címe) paraméterezhető osztályt vagy függvényt jelent.

A paraméterezés argumentumait < > jelek között, vesszővel elválasztva kell megadni. Egy kétparaméterű generikus osztályt ezek után

```
template<class A, class B> class osztálynév {osztálydefiníció};
```

szintaktika szerint lehet definiálni, ahol az *A* és *B* típusparaméterek helyére tetszőleges nevet írhatunk, bár a szokásjog szerint itt általában egy db nagybetűből álló neveket használunk. Az *osztálydefiníción* belül a paraméter típusok rövid alakja, tehát a példában *A* és *B*, használható.

Az előző osztály külsőleg implementált tagfüggvényei kicsit komplikáltan írandók le:

```
template <class A, class B>
visszatérés-típus osztálynév::tagfüggvénynév(argumentum def.)
{
   tagfüggvénytörzs
}
```

Az *argumentum-definícióban* és *tagfüggvénytörzsben* a paramétertípusokat ugyancsak a rövid alakjukkal adjuk meg. Az osztályon kívül definiált tagfüggvénytörzsek elhelyezésére speciális szabályok vonatkoznak, melyek sajnálatos módon fordítónként változnak. A személyi számítógépeken elterjedt C++ fordítók esetében, az osztályok definíciójához hasonlóan, a sablon külsőleg definiált tagfüggvényeit is minden a sablonra hivatkozó fájlban szerepeltetni kell.

Emlékezzünk vissza, hogy a normál osztályok külsőleg definiált tagfüggvényeit ezzel szemben csak egyetlen fájlban definiáltuk. A sablon osztályok speciális kezelésének mélyebb oka az, hogy a generikus definíciókból a fordító csak akkor fog bármit is készíteni, ha az konkréten paraméterezzük. Ha hasonló paraméterezést több fájlban is használunk, az azonos tagfüggvények felismeréséről és összevonásáról a fordító maga gondoskodik. Az elmondottak fontos következménye az, hogy a sablonnal megadott generikus osztályok teljes definícióját a

deklarációs fájlokban kell megadni. A korábban preprocesszor mechanizmussal megvalósított **generikus lista** sablonnal történő megadása tehát a következőképpen néz ki:

```
template<class R> class List;

template <class T> class ListElem {
friend class List<T>;
   T           data;
   ListElem *  next;
public:
   ListElem( ) {}
   ListElem( T d, ListElem * n ) { data = d; next = n; }
};

template <class R>  class List {
   ListElem<R>  head, *current;
   virtual int Compare( R& d1, R& d2 ) { return 1; }
public:
      List( ) { current = &head; head.next = NULL; }
   void Insert( R& data );
   BOOL  Get( R& data );
};

template <class R> void List<R>::Insert(R& data) {
   for( ListElem<R> * p =&head; p->next !=NULL; p =p->next)
      if ( Compare( p -> data, data) == 1 ) break;
   ListElem<R>* old = new ListElem<R>(p -> data,p -> next);
   p -> data = data; p -> next = old;
}
```

Miután a generikus osztályt definiáltuk, belőle paraméterezett osztályt, illetve az osztályhoz tartozó objektumot, a következőképpen hozhatunk létre:

```
osztálynév<konkrét paraméterlista> objektum;
```

Amennyiben a generikus lista osztályt egy *template.h* deklarációs fájlban írtuk le, a lista felhasználása ezek szerint:

```
#include "template.h"

class Student {
   String   name;
   double   average;
   ....
};

class MyStudentList : public List<Student> {
   int Compare( Student& s1, Student& s2 )
      { return ( s1.Average() > s2.Average() ); }
};

void main( ) {
   MyStudentList list;    // átlag szerint rendezett
   List<Student> not ordered list;  // nem rendezett
   Student st;
   list.Insert( st );
   list.Get( st );
}
```

Hasonlóképpen létrehozhatunk double, int, Vector, stb. típusú változók listáját a List<double>, List<int>, List<Vector>, stb. definíciókkal.

Végül vegyük elő korábbi ígéretünket, a **dinamikusan nyújtózkodó tömböt,** és valósítsuk meg generikusan, tehát általánosan, függetlenül attól, hogy konkrétan milyen elemeket kell a tömbnek tárolnia.

```
template < class Type >  class Array {
   int    size;   // méret
   Type * ar;     // heap-en lefoglalt tömbelemek
public:
   Array( ) { size = 0; array = NULL; } // alapért. konstr.
   Array( Array& a ) {  // másoló konstruktor
      ar = new Type[ size = a.size ];
      for( int i = 0; i < a.size; i++ ) ar[i] = a.ar[i];
   }
   ~Array( ){ if ( ar ) delete [] ar; } // destruktor
   Array& operator=( Array& a ) {  // = operátor
      if ( this != &a ) {
         if ( ar ) delete [] ar;
         ar = new Type[ size = a.size ];
         for( int i = 0; i < a.size; i++ ) ar[i] = a.ar[i];
      }
      return *this;
   }
   Type& operator[] (int idx);  // index operátor
   int  Size( ) { return size; }  // méret lekérdezése
};

template < class Type >
Type& Array< Type > :: operator[] ( int idx ) {
   if ( idx >= size ) {
      Type * nar = new Type[idx + 1];
      if ( ar ) {
         for( int i = 0; i < size; i++ ) nar[i] = ar[i];
         delete [] ar;
      }
      size = idx + 1;
      ar = nar;
   }
   return ar[idx];
}
```

A megvalósítás során, az általánosság lényeges korlátozása nélkül, feltételeztük, hogy a tömbelem típusának (a paramétertípusnak) megfelelő objektumokra az értékeadás (=) operátor definiált, és az ilyen típusú objektumokat van alapértelmezés szerinti konstruktoruk. Ez a feltétel beépített típusokra (int, double, char*, stb.) valamint olyan osztályokra, melyben nincs konstruktor és az értékadás operátort nem definiáltuk át, nyilván teljesül.

7. fejezet - 7. Objektumok tervezése és implementációja

Az C++ nyelvi eszközök megismerése után visszatérünk az objektum-orientált programfejlesztés analízist és architektúrális tervezést követő fázisaira, az objektum tervezésre és az implementációra. Ennek célja az objektum-orientált analízis és tervezés során született modellek C++ programmá történő átalakítása. Az alábbiakban összefoglaltuk a folyamat főbb lépéseit:

1. **Az objektum, a dinamikus és a funkcionális modellek kombinálása**, melynek során létrehozzuk az osztályok, illetve az attribútumok és metódusok egy részének a deklarációját.

2. **Az üzenet-algoritmusok és az implementációs adatstruktúrák kiválasztása,** amely a tervezési szempontok alapján pontosíthat egyes adatstruktúrákat, és részben ennek függvényében, részben az analitikus modellek alapján meghatározza az egyes metódusok algoritmusait.

3. **Az asszociációk tervezése,** amely az objektumok közötti kapcsolatok leírásához általában mutatókat alkalmaz.

4. **A láthatóság biztosítása.** Az objektumok a program végrehajtása során egymásnak üzeneteket küldhetnek, felhasználhatják egymást értékadásban illetve függvényargumentumként, stb. Ez azt jelenti, hogy az objektumok metódusaiban más objektumokra (változókra) hivatkozunk, ami csak abban az esetben lehetséges, ha a hivatkozott objektum változó a metódusból látható.

5. **Nem objektum-orientált környezethez, illetve nyelvekhez történő illesztés**. Ennek során meg kell oldani egy normál függvényekből álló rendszer és az objektumok közötti üzenetváltással működő programunk összekapcsolását.

6. **Ütemezési szerkezet kialakítása.** Amennyiben az egy processzoron futtatandó program modelljében több aktív objektum található, azok párhuzamosságát fel kell oldani. Általános esetben ide sorolható az objektumok processzorokhoz rendelése is a többprocesszoros és az elosztott rendszerekben.

7. **Optimalizálás (módosíthatóságra, futási időre és a forráskód méretére).** A tervezett rendszer közvetlen programkóddá transzformálása gyakran nem ad kielégítő megoldást. Ezért szükséges lehet a nem teljesülő szempontok szerint a terv finomítása illetve optimalizálása.

8. **Deklarációs sorrend meghatározása,** amely az egyes modulok, osztályok deklarációs függőségét tárja fel. Erre azért van szükség, mert a deklarációkat a függőségeknek megfelelő sorrendben kell megadni a programkódban.

9. **Modultervezés.** A rendszert a tervezés során általában alrendszerekre illetve modulokra bontjuk, amelyek a programkód szintjén különálló fájlokként jelennek meg.

1. 7.1. Az objektum, a dinamikus és a funkcionális modellek kombinálás

Ezen lépés feladata az analízis modellek és a tervezési szempontok alapján az egyes osztályok deklarációjának a közel teljes kidolgozása. Kidolgozás alatt az osztályok felsorolását, az öröklési relációk meghatározását, tartalmazási és asszociációs viszonyok megállapítását, a modellből közvetlenül következő attribútumok és metódusok leírását értjük.

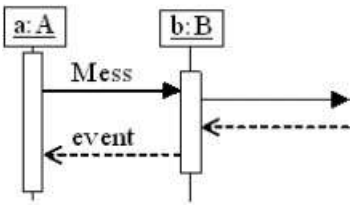
1.1. 7.1.1. Az objektummodell elemzése

Az első lépés általában az **osztálydiagra elemzése**, hiszen ez közvetlenül megadja az analízis alapján szükséges osztályokat, a felismert öröklési, tartalmazási és asszociációs viszonyokat, sőt utalhat az attribútumok egy részére is. Amennyiben a specifikáció elemzése során készítettünk összefoglaló táblázatot, az rávilágíthat az attribútumokon kívül az osztályokhoz tartozó alapvető metódusokra is.

1.2. 7.1.2. A dinamikus modell elemzése

Az osztályok metódusait a **dinamikus modell** határozza meg.

A kommunikációt leíró forgatókönyvekben láthatjuk az objektumok közötti üzenetváltásokat és az üzenetek sorrendjét. Az üzenetküldés a metódusok aktivizálásával történik, így a forgatókönyvekben egy objektumüzenet és a visszaadott esemény közötti részben lennie kell egy megfelelő metódusnak. Az üzenetek visszatérési értékének definiálása külön megfontolást érdemel. Általában ez is objektum, így gondoskodni kell a megfelelő definiáló osztályról. Egyszerűbb esetekben, amikor a visszatérési érték csupán valamilyen jelzőt tartalmaz, elfogadható a felsorolás típus (enum) alkalmazása is.



7.1. ábra: Üzenetek implementációja metódusként.

A 7.1. ábrán látható esetben a B osztályban szerepelnie kell egy Mess metódusnak, amely az event típusának megfelelő változót vagy objektumot ad vissza:

```
class Event { ... };   // vagy: enum Event { ... };
class B {
Event Mess( ... ) { }
};
```

A metódus vezérlését a dinamikus modellben szereplő **állapotgép** adja meg.

1.3. 7.1.3. Osztályok egyedi vizsgálata

A modelleken kívül, melyek a rendszert felülről, egyfajta top-down szemlélettel írják le, a metódusok definícióját az attribútumok, a tartalmazás és az asszociációk alapján végrehajtott **bottom-up megközelítéssel** is ki kell egészíteni. Ez az osztályok egyenkénti vizsgálatát jelenti, amely során megállapíthatjuk, hogy az objektum-definíció adott fázisában, az attribútumok, a tartalmazás és az asszociációk alapján milyen "értelmes" műveletek kapcsolhatók az adott objektumhoz.

Két alapvető szabályt mindenképpen be kell tartani. Egy objektum nem lehet "terülj, terülj asztalkám", azaz ha valamilyen információt nem teszünk bele, akkor annak felesleges helyet fenntartani és kifejezetten elítélendő azt onnan kiolvasni. Hasonlóképpen az objektum "fekete lyuk" sem lehet, azaz felesleges olyan információkat beleírni, amelyet aztán sohasem használunk.

Tekintsük példaként a 6.7.4 fejezetben megismert ideiglenes alkalmazott osztály (Temporary) definícióját és tegyük fel, hogy a modellek elemzése során az adódott, hogy ez az osztály név (name) és fizetés (salary) attribútumokkal rendelkezik. A név, tapasztalataink szerint, egy statikus jellemző, a születés pillanatában meghatározott és az egyed (objektum) teljes élete során általában változatlan marad. Az ilyen statikus attribútumokat az objektum születésekor kell inicializálni, célszerűen az objektum konstruktorával, és nem kell készíteni egyéb, az attribútumot megváltoztató metódust. A fizetés, a névvel szemben dinamikus jellemző, amely folyamatosan változhat, így szükségünk van egy olyan metódusra, amely azt bármikor megváltoztathatja (SetSalary). A kitöltetlen adatok elkerülésének érdekében a fizetést is inicializálni kell a konstruktorban mégha gyakran a születés pillanatában nem is ismert a későbbi kezdőfizetés. Használjunk erre a célra alapértelmezés szerinti argumentumokat, melyek egy jól definiált kezdeti értéket állítanak be.

Mivel a belső állapot fenntartására csak akkor van szükség, ha azt majdan ki is olvassuk, két lekérdező függvényt (GetName, GetSalary) is létre kell hoznunk.

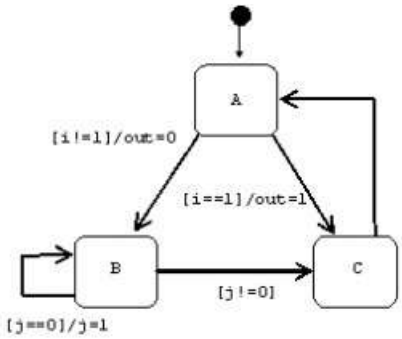
Ennek megfelelően az ideiglenes alkalmazott osztály metódusai az attribútumok elemzése alapján a következőek:

```
class Temporary {
   String  name;      // nem változik -> konstruktor
   long    salary;    // változik -> változtató üzenet
public:
           Temporary( String nam, long sal = 0 );
   void    SetSalary(  long sal );
   String  GetName( );   // lekérdező függvények
   long    GetSalary( );
};
```

2. 7.2. Az üzenet-algoritmusok és az implementációs adatstruktúrák kiválasztása

A metódusok törzsére vonatkozó információkat a dinamikus modell **véges állapotú gépeinek** definícióiból kaphatjuk meg. A véges állapotú gépek C++ programmá fordítása során alapvetően két lehetőség közül választhatunk. Az elsőben az **állapotot** kizárólag az utasításszámláló, tehát az aktuálisan végrehajtott programsor, reprezentálja, az állapotváltásokat pedig a szokásos vezérlő utasítások (if, for, while, stb.) hajtják végre. A második lehetőséget választva az aktuális állapotot egy változóban tartjuk nyilván és tipikusan switch utasítást használunk az aktuális állapot hatásának az érvényesítésére a bemeneti adatok vizsgálata és a reakciók végrehajtása során. Ezt a második megoldást **explicit állapotgépnek** hívjuk.

Példaként tekintsük a következő állapotgépet és annak a két módszerrel történő implementációját:



7.2. ábra: Példa állapotgép.

Az **állapotot az utasításszámláló képviseli**:

```
// A állapot
A: if (i == 1) {
      out = 1; goto C;
   } else out = 0;
// B állapot
   while( j == 0 ) j = 1;
// C állapot
C: goto A;
```

Megoldás **explicit állapotgéppel**:

```
enum State {A, B, C};
....
for ( State state = A ; ; ) {
   switch ( state ) {
   case A:  if ( i == 1) { out = 1; state = C; }
            else  { out = 0; state = B; }
            break;
   case B:  if ( j == 0) j = 1;
            else  state = C;
            break;
   case C:  state = A;
   }
}
```

Lényeges hatással van még az üzenet-algoritmusokra a belső állapotot reprezentáló adatmezők implementációs adatstruktúráinak szerkezete is. A programozási stílus szintén meghatározó, hiszen ugyanazon funkciót igen sokféleképpen lehet egy adott programnyelven megfogalmazni. A következőkben olyan elvárásokat elemzünk, melyeket érdemes megszívlelni a programozás során.

2.1. 7.2.1. Áttekinthetőség és módosíthatóság

Vége a programozás hőskorának, amikor a programozó elsődleges célja az volt, hogy a szűkös erőforrások (memória, számítási sebesség) szabta korlátok közé szorítsa a programját. A mai programok hihetetlenül bonyolultak, másrészt folyamatosan fejlődnek. Ezért ma a programozás alapvető célja olyan programok készítése, melyek könnyen megérthetők, javíthatók és módosíthatók olyan személyek által is, akik az eredeti változat elkészítésében nem vettek részt.

Ennek a követelménynek egy következménye, hogy a tagfüggvények implementációja nem lehet túlságosan hosszú, a képernyőnyi méretet nem nagyon haladhatja meg. Mit tehetünk, ha a feladat bonyolultsága miatt egy metódus implementációja mégis túl bonyolultnak ígérkezik? Az adott hierarchia szinten ismét a dekompozíció eszközéhez kell folyamodnunk, amely lehet objektum-orientált, vagy akár funkcionális dekompozíció is.

Hierarchikus objektum-orientált dekompozíció azt jelenti, hogy csak az adott osztályhoz tartozó metódusoktól elvárt működést tekintjük és ugyanúgy, ahogy a teljes feladatot megoldottuk, újra elvégezzük az objektum orientált analízis, tervezés és implementáció lépéseit. A keletkező objektumok természetesen a vizsgált objektumra nézve lokálisak lesznek, hiszen csak a vizsgált objektum épít az új objektumok szolgáltatásaira. Ezeket a lokális objektumokat a vizsgált objektumnak vagy metódusainak tartalmaznia kell.

A funkcionális dekompozíció említésének kapcsán az az érzésünk támadhat, hogy miután kidobtuk az ajtón (az egész objektum-orientált módszert a funkcionális megközelítés ostorozásával vezettük be) most visszasomfordált az ablakon. Igenám, de most egy kicsit más a helyzet, hiszen nem egy teljes feladatot, hanem annak egy szerény részfeladatát, egy osztály egyetlen metódusának a tervezését valósítjuk meg. A bevezetőben említett problémák, mint a későbbi módosítás nehézsége, most egyetlen osztály belső világára korlátozódnak, így már nem annyira kritikusak. Egy egyszerű metódus (függvény) felbontásánál a funkcionális dekompozíció gyakran természetesebb, ezért ezen a szinten bevett programozói fogás. Természetesen a felbontásból származó tagfüggvények, melyek a metódus részeit valósítják meg, az objektum felhasználói számára értéktelenek, így azokat privát tagfüggvényként kell megvalósítani.

2.2. 7.2.2. A komplexitás

Az algoritmusok és az implementációs adatstruktúrák kiválasztásánál az idő és tár komplexitást is figyelembe kell venni. Emlékezzünk vissza a telefonhívás-átirányítást végző programunk két változatára. Az elsőben a számpárokat tömbben, a másodikban egy bináris fában tároltuk, amivel az időkomplexitást $O(n^2)$-ről $O(\log n)$-re csökkentettük. A komplexitási jellemzők azt a felismerést fejezik ki, hogy a mai nagyteljesítményű számítógépeknél, néhány kivételtől eltekintve, a felhasznált idő és tár csak akkor válik kritikussá, ha a megoldandó probléma mérete igen nagy. Ezért egy algoritmus hatékonyságát jól jellemzi az a függvény ami megmutatja, hogy az algoritmus erőforrásigénye milyen arányban nő a megoldandó feladat méretének növekedésével.

2.3. 7.2.3. Az adatstruktúrák kiválasztása, az osztálykönyvtárak felhasználása

A komplexitást gyakran az adatstruktúrák trükkös megválasztásával lehet kedvezően befolyásolni. Bár minden programozási nyelv ad több-kevesebb segítséget összetett adatszerkezetek kialakítására, C++-ban a generikus osztályok felhasználása lehetővé teszi, hogy az egzotikus szerkezeteket készen vegyük az **osztály-könyvtárakból**, vagy még szerencsétlen esetben is legfeljebb egyetlen egyszer kelljen megvalósítani azokat. Az ún. **tároló osztályokat** (**container class**) tartalmazó könyvtárak tipikus elemei a generikus tömb, lista, hashtábla, sor, verem stb. Léteznek osztály-könyvtárak a külső erőforrások, mint az I/O stream, real-time óra stb. hatékony és kényelmes kezeléséhez is. A legnagyobb ismertségre mégis a grafikus felhasználói felületek programozását támogató könyvtárak tettek szert, ilyen például a Microsoft Foundation Class (az alapfilozófiájukat illetően lásd a 6.7.8. fejezetet). A valós idejű (real-time) monitorokat tartalmazó könyvtárak lehetővé teszik, hogy az aktív objektumok időosztásos rendszerben látszólag párhuzamosan birtokolják a közös processzort, a normál C++ nyelvet ily módon egyfajta konkurens C++-ra bővítve.

2.4. 7.2.4. Robusztusság

Valamely osztály tekintetében a robosztusság azt jelenti, hogy az osztály objektumai specifikációnak nem megfelelő üzenet argumentumokra sem okoznak katasztrofális hibát ("nem szállnak el"), hanem a hibát korrekt módon jelzik és a belső állapotukat konzisztensen megőrzik. Ezt alapvetően a publikus függvények bemenetén végrehajtott hihetőség-ellenőrzéssel érhetjük el. A privát függvények argumentum-ellenőrzésének kisebb a jelentősége, hiszen azok csak az objektum belsejéből hívhatók, tehát olyan helyekről amit feltehetően ugyanazon programozó implementált.

2.5. 7.2.5. Saját debugger és profiler

Végül érdemes megjegyezni, hogy a programírás során célszerű a tagfüggvényekben járulékos ellenőrző és kiíró utasításokat elhelyezni, melyek a nyomkövetést jelentősen segíthetik, és amelyeket feltételes fordítással a végleges változatból ki lehet hagyni.

A profiler a teljesítménynövelés "műszere", amely azt méri, hogy az egyes metódusokat egy feladat végrehajtása során hányszor hajtottuk végre és azok átlagosan mennyi ideig tartottak. Ilyen eszközt a tagfüggvények elején és végén elhelyezett számláló és időmérő utasításokkal bárki könnyen létrehozhat. A profiler által szolgáltatott mérési eredmények alapvetőek a teljesítmény fokozása során, hiszen nyilván csak azokat a tagfüggvényeket célszerű felgyorsítani, amelyek a futási idő jelentős részéért felelősek.

A programok hatékonyságának minden áron való, és gyakran átgondolatlan fokozása iránti igény gyakori programozói betegség, ezért érdemes ennek a kérdésnek is néhány sort szentelni. Egy tapasztalati tényt fejez ki a strukturált módszerek egyik atyja, Yourdon nevéhez fűződő mondás:

"Sokkal könnyebb egy jól működő programot hatékonnyá tenni, mint egy hatékonyt jól működővé".

A szokásos implementációs trükkök (üzenetek áthidalása, függvényhívások kiküszöbölése, bitbabrálás, stb.), melyek a programot átláthatatlanná és módosíthatatlanná teszik, csak lineáris sebességnövekedést eredményezhetnek, azaz a program idő-komplexitását nem befolyásolják. Sokkal jobban járunk az algoritmus és adatstruktúra megfelelő kiválasztásával.

Nyilván csak azokat a részeket kell felgyorsítani, amelyek tényleg meghatározók a program sebességének szempontjából. Azt viszont, hogy melyik rész ilyen, a programírás során gyakran nem tudjuk eldönteni. Ezért érdemes megfogadni Yourdon tanácsát és először "csak egy jó" programot írni, majd a tényleges alkalmazás körülményei között a profiler-es méréseket elvégezve eldönthetjük hogy szükség van-e további optimalizációra, és ha igen, konkréten mely programrészeken kell javítani. Ilyen optimalizációs lépésekre a 7.7. fejezetben még visszatérünk.

Ezen fejezet nagy részében a szép C++ programozás néhány aspektusát tekintettük át. Sokan szeretik ezt a kérdést praktikusan megközelíteni és olyan "szabályokat" felállítani, amelyek betartása esetén a születendő program mindenképpen szép lesz. Ilyen szabálygyűjteménnyel sajnos nem szolgálhatunk. Az objektum-orientált elvek következetes és fegyelmezett végigvitele és persze jelentős objektum-orientált tervezési és programozási gyakorlat viszont sokat segíthet.

Hogy mégse okozzunk a gyors sikerre éhezők táborának csalódást, az alábbiakban közreadjuk Broujstroup után szabadon felhasználva és módosítva az "objektum orientált programozás 10 parancsolatát":

1. *Egy változót csak ott definiálj, ahol már inicializálni is tudod!*
2. *Ha a programban ugyanolyan jellegű műveletsort, illetve vizsgálatot több helyen találsz, a felelősséget nem sikerült kellően koncentrálni, tehát az objektum-orientált dekompozíciót újra át kell gondolnod.*
3. *Függvények ne legyenek hosszabbak, mint amit a képernyőn egyszerre át lehet tekinteni!*
4. *Ne használj globális adatokat!*
5. *Ne használj globális függvényeket!*
6. *Ne használj publikus adatmezőket!*
7. *Ne használj friend-et, csak az 4,5,6 elkerülésére!*
8. *Ne férj hozzá másik objektum adataihoz közvetlenül! Ha mégis megteszed, csak olvasd, de ne írd!*
9. *Egy objektum belső állapotában ne tárold annak típusát! Ehelyett használj virtuális függvényeket!*
10. *Felejtsd el, hogy valaha C-ben is tudtál programozni!*

A 10+1. ökölszabály:

> *Ha egy részfeladatot programozói pályafutásod alatt már háromszor kellett megoldanod, akkor ideje elgondolkozni valamilyen igazán újrahasznosítható megoldáson.*

3. 7.3. Asszociációk tervezése

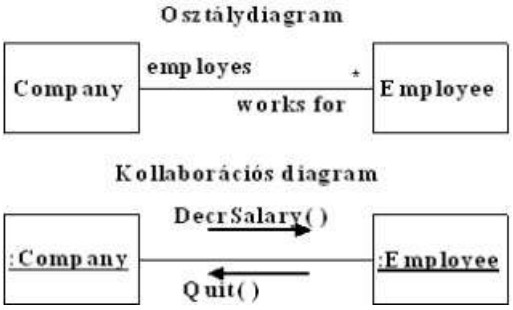
Asszociáció alatt két vagy több objektum, esetleg időben változó kapcsolatát értjük. Az asszociációkat csoportosíthatjuk:

- **irány szerint**; amennyiben a két kapcsolódó objektum közül csak az egyik metódusai számára fontos, hogy melyik a párja, egyirányú asszociációról, ha pedig mindkét objektum számára lényeges a pár ismerete, kétirányú asszociációról beszélünk.
- **multiplicitás szerint**; ekkor azt vizsgáljuk, hogy egy kijelölt objektumnak hány asszociációs párja van. Az *1-1* típusú asszociáció azt jelenti, hogy minden résztvevő pontosan egy másikkal lehet kapcsolatban. Ilyen a *házastársi* viszony, ahol az objektumpárt a férj és feleség alkotja. Az *1-n* viszony arra utal, hogy az asszociációs párok egyik tagja csak egyetlen kapcsolatban míg a másik tagja egyszerre több kapcsolatban is szerepelhet. A családi példánál maradva az anya-gyermek objektumpárra "*az anyja*" viszony *1-n*, hiszen egy anyának több gyermeke lehet, de egy gyermeknek pontosan egy anyja van. Az *m-n* típusú asszociáció minkét résztvevő számára megengedi a többszörös részvételt. Ilyen viszony az emberi társadalomban a férfiak és a nők között az *udvarlás*. Egy férfi egyszerre több nőnek is teheti a szépet, és megfordítva egy nő igen sok férfival udvaroltathat magának. Végül meg kell említenünk, hogy a fenti típusokat tovább tarkíthatja az **opcionalitás**, amikor megengedjük azt, hogy nem minden, az adott osztályhoz tartozó, objektum vegyen részt valamely asszociációban. Például a nőket és a gyermekeket összekapcsoló "*az anyja*" asszociációban a nők részvétele opcionális, hiszen vannak nők akiknek nincs gyermekük, ezzel szemben a gyermekek részvétele kötelező, hiszen gyermek anya nélkül nem létezhet.
- **minősítés (kvalifikáció) szerint**. A minősítés egy *1-n* vagy *n-m* asszociáció esetén azt a tulajdonságot jelenti, ami alapján az "egy" oldalon lévő objektumhoz kapcsolódó több objektumból választani lehet.

Az egyirányú *1-1* és *1-n* típusú asszociációk legegyszerűbb implementációs technikája a **tartalmazás** (aggregáció) alkalmazása. A tartalmazást úgy alakítjuk ki, hogy azt az objektumot, amelyik csak egyetlen másik objektummal áll kapcsolatban, a másik részeként, mintegy attribútumaként definiáljuk. A tartalmazás önmagában egyirányú asszociációt biztosít, hiszen egy objektum metódusai az objektum attribútumait nyilván látják, de egy attribútumként szereplő objektumnak nincs tudomása arról, hogy őt tartalmazza-e egy másik vagy sem. Ha kétirányú asszociációt kell megvalósítanunk, akkor az ún. **visszahívásos (call-back) technikát** kell

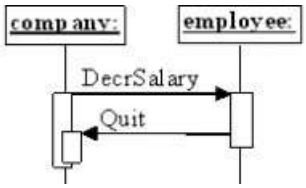
alkalmazhatunk. Ez azt jelenti, hogy a tartalmazó a saját címét vagy a referenciáját átadja a tartalmazottnak, ami ennek segítségével indirekt üzenet-küldést valósíthat meg.

Elemezzünk egy céget (Company) és a cég alkalmazottainak (Employee) viszonyát leíró feladatot. A cég-alkalmazott asszociáció *1-n* típusú, és a résztvevők szempontjai szerint "alkalmaz (employes)" vagy "neki dolgozik (works for)" asszociációnak nevezhetjük. A kapcsolat ténye a példánkban azért fontos, mert egy cég csökkentheti a saját dolgozóinak a bérét (DecrSalary üzenettel), míg a dolgozó csak az őt foglalkoztató cégtől léphet ki (Quit üzenettel). Az asszociáció tehát kétirányú, mivel mindkét résztvevőnek tudnia kell a párjáról.



7.3. ábra: Asszociációtervezés

A visszahívásos technika lehetséges alkalmazásainak bemutatása céljából először tegyük fel, hogy a tartalmazott objektum (a dolgozó) csak akkor küldhet üzenetet a tartalmazónak (cég), ha a cég is üzent. Például a dolgozó csak akkor léphet ki, ha a fizetése a rendszeres bércsökkentések következtében negatívvá válik. Ezt a szituációt leíró kommunikációs modell:



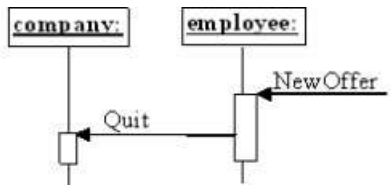
7.4. ábra: Visszahívás.

A dolgozó (employee) objektumnak csak a DecrSalary metódusában van szüksége arra az információra, hogy pontosan melyik cégnek dolgozik. Ezt a metódust viszont éppen a foglalkoztató cég aktivizálja, tehát a visszahívás megoldható úgy, hogy a fizetéscsökkentő DecrSalary hívás egyik argumentumaként a cég átadja a saját címét, hogy a dolgozó az esetleges kilépés esetén a Quit üzenetet ide elküldhesse.

Egy lehetséges megvalósítás, amelyben a cég objektumban a dolgozókat egy generikus, nyújtózkodó tömbben tartjuk nyilván (6.8.2. fejezet):

```
class Company {
   Array < Employee > employees; // tartalmazás
public:
   void SetSalary( int i )
      { employees[i].DecrSalary( 50, this ); }
   void Quit( Employee * e ) { ... }
};

class Employee {
   int salary;
public:
      Employee( ) { salary = 100; }
                  // call-back cím argumentum
   void DecrSalary(int amt, Company * pcomp ) {
      salary -= amt;
      if (salary < 0) pcomp -> Quit( this ); // call-back
   }
};
```

Az előzőtől lényegesen eltérő helyzettel állunk szemben, ha a dolgozó akkor is kiléphet, ha saját cégétől nem kap üzenetet, például egy másik cég jobb ajánlatát megfontolva. Az esetet leíró kommunikációs modell:



7.5. ábra: Üzenet előzmény nélkül.

Ebben az esetben nem élhetünk az előző trükkel, hiszen a NewOffer üzenet nem az alkalmazó cégtől érkezik, azaz annak argumentuma sem lehet az alkalmazó cég címe. A dolgozónak tudnia kell az őt foglalkoztató cég címét, hogy ilyen esetben ki tudjon lépni. Minden employee objektumot ki kell egészíteni egy mutatóval, ami az őt foglalkoztató cég objektumra mutat, és amit célszerűen az Employee konstruktorában inicializálunk.

Ily módon a megvalósítás:

```
class Company {
   Array < Employee > employees;
public:
   void Quit( Employee * e ) { ... }
};
Company company;

class Employee {
   int            salary;
   static Company * pcomp; // callback cím
public:
   void NewOffer( ) { pcomp -> Quit( this ); }
};

Company * Employee :: pcomp = &company;     // inicializálás
```

Újdonságként jelent meg a megoldásban a visszahívást lehetővé tevő cím (pcomp) definíciója előtti **static** kulcsszó. A statikus deklaráció azt jelenti, hogy az összes Employee típusú osztályban ez az adattag közös lesz. A statikus adattagok a globális változókhoz hasonlítanak, de csak egyetlen osztály viszonylatában. A statikus adattagokat kötelező inicializálni, a példában szereplő módon.

A beágyazott mutatókon alapuló indirekt üzenetküldési eljárást jelentő visszahívásos (call-back) technika nevét a telefonálásban elterjedt jól ismert szerepe miatt kapta. Az analógia világossá válik, ha meggondoljuk, hogy a problémát az okozta, hogy a dolgozó objektumok közvetlenül nem látják a cég objektumot (nem ismerik a telefonszámát), így üzenni sem tudnak neki. Ezért valamikor a cég objektum felhívja (üzen) a dolgozó objektumnak, és az üzenetben közli a saját számát (címét). Ezt megjegyezve a továbbiakban a dolgozó bármikor hívhatja a cég objektumot.

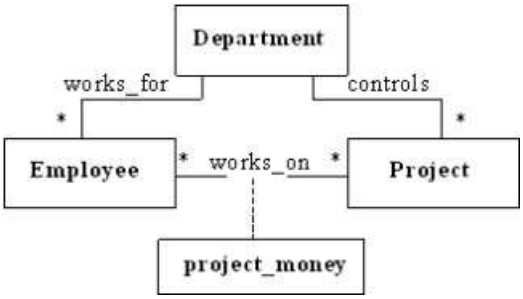
Az asszociációknak a tartalmazásnál általánosabb megvalósítására ad módot a **beágyazott mutatók**alkalmazása, amelyet tetszőleges, akár kétirányú *n-m* asszociációk implementációjára is felhasználhatunk. A beágyazott mutatók módszere azt jelenti, hogy ha egy objektum egy vagy több másikkal áll (nem kizárólagos) kapcsolatban, akkor az objektum attribútumai közé a kapcsolatban álló objektumok címét vagy referenciát vesszük fel, nem pedig magukat az objektumokat mint a tartalmazással történő megvalósítás esetén.

A beágyazott mutatók használatát egy példán keresztül mutatjuk be:

Egy tanszéken (Department) több dolgozó (Employee) dolgozik (works_for), és a tanszék több projektet irányít (controls). Egy dolgozó egyetlen tanszékhez tartozhat, és egy projektnek egyetlen irányító tanszéke van. Egy dolgozó viszont egyszerre több projekten is dolgozhat (works_on), akár olyanokon is, amit nem a saját tanszéke

irányít. Egy projektben általában sok dolgozó vesz részt. A projektekben résztvevők munkájukért honoráriumot kapnak (project_money).

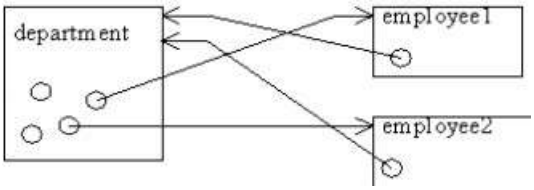
A leírt rendszer objektum-modellje a következőképpen néz ki:



7.6. ábra: A tanszék objektum-modellje.

Most térjünk rá az asszociációk objektumokban történő megvalósítására. Először egy *1-n* típusú asszociációt tekintünk, mint például a department-employee kapcsolatot és feltételezzük, hogy az kétirányú.

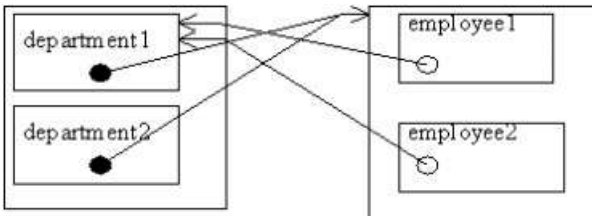
A beágyazott mutatók módszere szerint a department objektumnak annyi Employee* típusú mutatóval kell rendelkeznie, ahány dolgozó (employee) ezen a tanszéken dolgozik. A mutatóknak a tanszéken dolgozó embereket megtestesítő objektumokra kell mutatniuk. A másik oldalról, az employee objektumok része egy-egy Department* típusú mutató, amely arra a tanszékre mutat, amely őt foglalkoztatja.



7.7. ábra: Redundáns asszociációk.

Ezzel a megoldással szemben az a súlyos kifogás merül fel, hogy redundánsan tárolja azt az információt, hogy egy dolgozó melyik tanszéken dolgozik, hiszen ez mind az employee mind a department objektumból kiolvasható. Ez a **redundancia** különösen azért veszélyes, mert nem korlátozódik egyetlen objektumra, hanem két különálló objektumban oszlik szét.

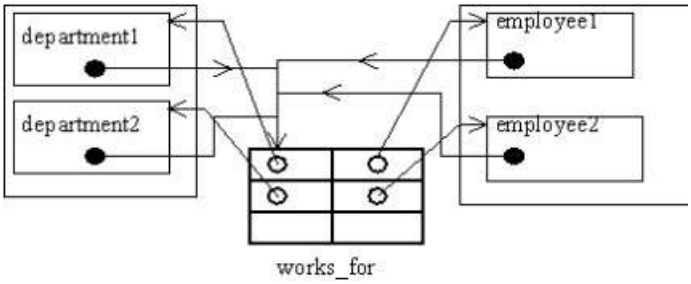
Az ilyen jellegű redundanciák és az objektumokban szereplő mutatóhalmazok eltávolítását **normalizálásnak** nevezzük. A normalizálás úgy történik, hogy az összes dolgozót egyetlen táblázatba (tárolóba) foglaljuk össze, és a department objektumokban pedig nem a foglalkoztatott dolgozók címét, hanem az összes dolgozót tartalmazó tábla kezdőcímét tároljuk. Mivel ez a mutató minden department objektumra azonos, célszerűen statikus tagnak kell definiálni (erre utal a 7.8. ábrán a kitöltött kör).



7.8. ábra: Redundanciamentes asszociációk.

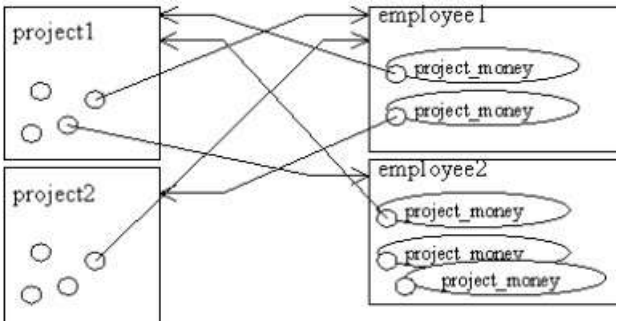
A redundacia ily módon történő megszüntetéséért súlyos árat kell fizetnünk. Ha arra vagyunk kíváncsiak, hogy egy tanszéken kik dolgoznak, akkor sorra kell venni a dolgozókat és meg kell nézni, hogy a dolgozó objektum beágyazott mutatója a kiválasztott tanszékre mutat-e. Keresni kell, amelyre a redundáns megoldásban nem volt szükség. A keresés a legegyszerűbb megvalósításban lineáris időigényű (*O(n)*), de némi többletmunka árán a komplexitása jelentősen csökkenthető, amennyiben például bináris fákat (*O(log n)*) vagy hash-táblákat (*O(1)*) alkalmazunk.

Az első megoldási javaslatunkban az a "tudás", hogy egy tanszéken kik dolgoznak mind a department mind az employee objektumokban jelen volt. A második alternatívában ezt a "tudást" csak az employee-ra (a *1-n* asszociációban az *n*-nek megfelelő tagra) korlátoztuk. Logikailag talán tisztább, ha ezt az információt nem rendeljük egyik részvevőhöz sem, hanem külön objektumként kezeljük. Ezt az objektumot **asszociációs objektumnak** nevezzük, melynek felhasználásával az asszociáció implementációs sémája a department objektumok tárolójából, az employee objektumok tárolójából és az azokat összekapcsoló asszociációs objektumból áll. Az asszociációs objektum maga is egy táblázat, amelyben az összetartozó department-employee párok címeit tároljuk. Minden asszociációval kapcsolatos kérdés ezen táblázat alapján válaszolható meg.



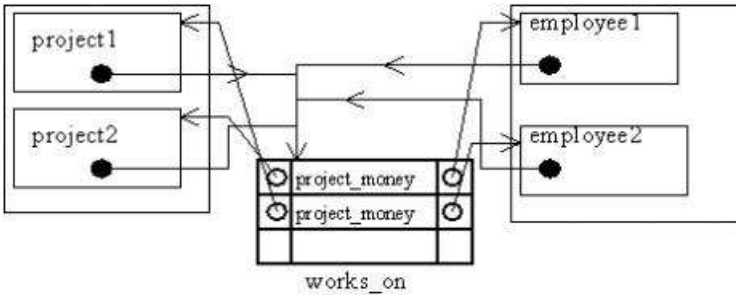
7.9. ábra: Asszociációs objektum.

Az *m-n* **asszociáció** megvalósításának bemutatásához a 7.9. ábrán a project-employee pár kapcsolatát valósítjuk meg. Első próbálkozásként használjuk a beágyazott mutatók módszerét! Mivel most egyetlen project-hez több employee kapcsolódhat, és megfordítva egyetlen employee-hez több project, mindkét objektumban a másik párra mutató pointerek halmazával kell dolgoznunk. Az asszociációhoz kapcsolódó attribútum (project_money) elhelyezése külön megfontolást érdemel. Ezt elvileg bármelyik asszociációs mutató társaságában elhelyezhetjük, (a 7.10. ábrán az employee objektumokba tettük bele).



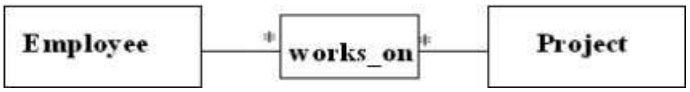
7.10. ábra: m-n asszociáció.

Hasonlóképpen az *1-n* asszociációhoz, ennek a megoldásnak is a redundancia a hibája, amelyet most is normalizálással szüntethetünk meg. Az *1-n* asszociációval szemben most csak az asszociációs objektumok felhasználása javasolt, hiszen ebben az esetben csak így küszöbölhetjük ki az objektumokban szereplő mutatóhalmazokat.



7.11. ábra: m-n asszociáció asszociációs objektummal.

Vegyük észre, hogy az asszociációs objektumok bevezetése az *n-m* asszociációt lényegében két *1-n* asszociációra bontotta, amelynek normalizált megvalósítása már nem okoz gondot:



7.12. ábra: m-n asszociáció felbontása két 1-n asszociációra.

A fenti módszerek alkalmazását a következő feladat megoldásával demonstráljuk:

Listázzuk ki egy tanszékre, hogy mely saját dolgozók vesznek rész egy adott, a tanszék által irányított projektben.

Oldjuk meg a feladatot a beágyazott mutatók módszerével! Az *1-n* asszociációk "*n*" oldalán egyetlen beágyazott mutatót kell szerepeltetnünk, míg az "*1*" oldalon és az *n-m* asszociációk mindkét oldalán mutatók halmazát. A halmazok megvalósításához az Array generikus nyújtózkodó tömböt (6.8.2. fejezet) használjuk fel:

```
class Department {
   Array< Person * >    employees;
   Array< Project * >   projects;
public:
   Array<Person *>&  Empls( ) { return employees; }
   Array<Project *>& Projs( ) { return projects; }
};

class Project {
   Department *       controller_department;
   Array< Person * >  participants;
public:
   Array<Person *>& Parts( ) {return participants;}
};

class Person {
   String               name;
   Department *         works_for;
   Array< Project   * > works_on;
   Array< int >         project_money;
public:
   String& Name( ) { return name; }
   Department * Department( ) { return works_for };
};

ListOwnProjectWorkers( Department& dept ) {
   for(int e = 0; e < dept.Empls().Size(); e++ ) {
      for(int p = 0; p < dept.Projs().Size(); p++ ) {
         for(int i = 0; i < dept.Projs()[p]->Parts().Size(); i++) {
            if (dept.Empls()[e] == dept.Projs()[p]->Parts()[i])
               cout << dept.Empls()[e]->Name();
         }
      }
   }
}
```

Az asszociációs objektumokkal történő megoldás előtt létrehozzuk az egyszerű asszociációs táblázat (AssocTable) és az attribútumot is tartalmazó tábla (AttribAssocTable) generikus megvalósításait:

```
template <class R, class L>  class AssocElem {
   R * right;
   L * left;
public:
   AssocElem(R * r = 0, L * l = 0) { right = r; left = l; }
   R * Right( ) { return right; }
```

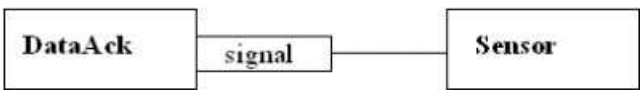
```
      L * Left( )  { return left;  }
};

template <class R, class L>
class AssocTable : public Array< AssocElem<R,L> > {
public:
   Array<R *> Find( L * l )  {
      Array< R * > hitlist;
      for( int hits = 0, int i = 0; i < Size( ); i++ ) {
         if ( (* this)[i].Left() == l )
         hitlist[ hits++ ] = (*this)[i].Right();
      }
      return hitlist;
   }
   Array<L *> Find( R * r ){
      Array< L * > hitlist;
      for( int hits = 0, int i = 0; i < Size( ); i++ ) {
         if ( (* this)[i].Right() == r )
         hitlist[ hits++ ] = (* this)[i].Left();
      }
      return hitlist;
   }
};

template <class R, class L, class A>
class AttribAssocTable : public AssocTable< R,L > {
   Array< A > attributes;
   ....
};
```

Ezek felhasználásával a feladat megoldását az olvasóra bízzuk.

Az asszociációk speciális fajtái az ún. **minősített asszociációk**, amelyek egy objektumot egy minősítő tag (qualifyer) segítségével rendelnek egy másikhoz. A minősítő tag csökkenti az asszociáció multiplicitását azáltal, hogy a lehetséges kapcsolódások közül a tag alapján kell választani. Az utóbbi tulajdonság alapján a minősítés az adatbázisok indexelésére hasonlít.

Vegyünk példaként egy mérésadatgyűjtő rendszert (DataAck), melyhez érzékelők (Sensor) kapcsolódnak. Az érzékelők különböző jeleket (signal) mérnek. Az érzékelők és jelek kapcsolata felfogható egyszerű *1-n* típusú asszociációként is, melynél az érzékelt jel az érzékelő attribútuma. Sokkal kifejezőbb azonban a minősített asszociációk alkalmazása, amikor a mérésadatgyűjtő és az egyes érzékelők *1-1* minősített asszociációban állnak, ahol a minősítést az érzékelt jel definiálja.

7.13. ábra: Egyszerű asszociáció.



7.14. ábra: Minősített asszociáció.

Tegyük fel, hogy a példánkban az asszociáció tárolására az önteszt funkciók megvalósítása miatt van szükség. Ha egy jel mért értéke érvénytelennek mutatkozik, akkor a jelet mérő érzékelőre öntesztet kell futtatni. Ennek érdekében a jelet képviselő osztályt (Signal) egy érvényesség-ellenőrző metódussal (IsValid), míg az érzékelőt (Sensor) egy önteszt metódussal (SelfTest) kell kiegészíteni.

A mérésadatgyűjtő (DataAck) és az érzékelők (Sensor) közötti minősített asszociáció egyirányú, hiszen csak a mérésadatgyűjtőben merül fel az a kérdés, hogy egy adott jel függvényében hozzá melyik érzékelő tartozik. Így a beágyazott mutatókat csak a DataAck osztályba kell elhelyezni. A minősítés miatt újdonságot jelent a beágyazott mutatók jelek (Signal) szerinti elérése. Ezt egy olyan asszociatív tárolóval (tömbbel) valósíthatjuk meg, ahol az egyes elemeket a Signal típusú objektumokkal lehet keresni, illetve indexelni.

Amennyiben rendelkezünk egy generikus asszociatív tömbbel,

```
template < class T, class I > class AssociativeArray,
```

ahol a T paraméter a tárolt elemeket az I paraméter az indexobjektumot jelöli, a mérésadatgyűjtő rendszer az alábbiakban látható módon valósítható meg:

```
class Signal {
   ....
   BOOL IsValid( );
};
class Sensor {
   ....
   BOOL  SelfTest( );
};

class DataAck {
   AssociativeArray < Sensor *, Signal > sensors;
public:
   Measurement ( ) {
      if ( ! signal.IsValid( ) )
         sensors[ signal ] -> SelfTest( );
   }
};
```

A példában felhasznált **generikus asszociatív tömb** egy lehetséges, legegyszerűbb megvalósítása a generikus Array-re épül (a másoló konstruktor, az értékadó operátor és destruktor implementálását az olvasóra bízzuk):

```
template <class T, class I> class AssociativeArray {
   Array< T > data;
   Array< I > index;
public:
   AssociativeArray( ) { }
   AssociativeArray( AssociativeArray& );
   AssociativeArray& operator=( AssociativeArray& );
   ~AssociativeArray( );
   T& operator[] ( I& idx);
   int   Size( ) { return data.Size( ); }
};

template < class T, class I>
T& AssociativeArray< T, I >::operator[] ( I& idx ) {
   for( int i = 0; i < data.Size(); i++ )
   if ( idx == index[i] ) return data[i];
   index[ data.Size() ] = idx;
   return data[ data.Size() ];
}
```

Ezen implementáció feltételezi, hogy a T típusra létezik értékadó (=) operátor és az I típusra az értékadó (=) valamint összehasonlító (==) operátor. Beépített típusok (pl. int, double, stb.) esetén ezek rendelkezésre állnak, saját osztályok esetén azonban az operátorokat implementálni kell.

A fenti implementációban az asszociatív tömb tartalmazza az adat és index tárolókat. Egy másik lehetséges megoldási mód az öröklés alkalmazása, hiszen a generikus asszociatív tömb lényegében egy normál generikus tömb, amely még rendelkezik egy másik objektum szerinti indexelési képességgel is:

```
template <call T, class I>
class AssociativeArray : public Array<T> {
   Array< I > index;
```

```
    ....
};
```

Végül ejtsünk szót a fenti megvalósítás komplexitási jellemzőiről is. Az index szerinti elérés során lineáris keresést alkalmaztunk, amely *O(n)* időt igényel. Bonyolultabb adatstruktúrák alkalmazásával ez lényegesen javítható, például bináris fával *O(log n)*, hash-tábla alkalmazásával akár konstans keresési idő (*O(1)*) is elérhető.

4. 7.4. Láthatóság biztosítása

Bizonyos értelemben az asszociációhoz kapcsolódik az objektumok közötti láthatósági viszony értelmezése is. Láthatósági igényről akkor beszélünk, ha egy objektum valamely metódusában egy másik objektumnak üzenetet küldhet, címét képezheti, argumentumként átadhatja, stb. Az analízis során általában nem vesződünk a láthatósági kérdésekkel, hanem nagyvonalúan feltételezzük, hogy ha egy objektum üzenetet kíván küldeni egy másiknak, akkor azt valamilyen módon meg is tudja tenni. Az implementáció felé közeledve azonban figyelembe kell vennünk az adott programozási nyelv sajátságait, amelyek a változókra (objektumokra) csak bizonyos nyelvi szabályok (ún. láthatósági szabályok) betartása esetén teszik lehetővé a hozzáférést.

A feladat alapján igényelt láthatósági viszonyokat elsősorban a kommunikációs modell forgatókönyvei és objektum kommunikációs diagramja alapján tárhatjuk fel.

Elevenítsük fel a telefonhívás-átirányítási példánk (6.6 fejezet) megoldása során kifejtett, a láthatósági viszonyokkal kapcsolatos megállapításainkat. Ott az implementációs oldalról közelítve azt vizsgáltuk, hogy miképpen küldhet egy Sender osztályhoz tartozó sender objektum a Sender::f( ) metódusában üzenetet egy receiver objektumnak.

```
globális változók;

Sender :: f ( [this->saját adatok], argumentumok )
{
   lokális változók;
receiver.mess( );   // direkt üzenetküldés
}
```

Mint megállapítottuk, a receiver objektumnak vagy az erre hivatkozó mutatónak illetve referenciának a következő feltételek valamelyikét ki kell elégítenie:

1. az adott fájlban a Sender::f sor előtt deklarált globális, tehát blokkon kívül definiált változó,
2. a Sender::f függvény lokális, a függvény kezdő és lezáró {} zárójelei között definiált változója,
3. a Sender::f függvény argumentuma
4. a sender objektum komponense, azaz a sender tartalmazza a receiver-t vagy annak címét.

Az alternatívák feltérképezése után vizsgáljuk meg azok előnyeit, hátrányait és alkalmazhatóságuk körülményeit!

1. Amennyiben a receiver, vagy a rá hivatkozó mutató globális változó, akkor a forrásfájlban definiált összes tagfüggvényből látható, ami ellentmond az objektum-orientált filozófia egyik alappillérének, az információ rejtésének. Ezért ez a megoldás csak akkor javasolható, ha az objektumok döntő többségének valóban látnia kell a receiver objektumot.
2. A lokális változók élettartama a küldő objektum metódusának a futási idejére korlátozódik, így ez a megközelítés nem lehet sikeres olyan objektumok esetén, amelyek életciklusa ettől eltérő.
3. A receiver objektumnak vagy címének függvényargumentumként történő átadása feltételezi, hogy a sender.f() hívója látja ezt az objektumot, hiszen ellenkező esetben nem tudná átadni. Ha ez a feltétel nem

teljesül, akkor ez az alternatíva nem alkalmazható. Amennyiben egy metódus igen sok objektumnak üzenhet, akkor további hátránya ennek a megoldásnak, hogy a hívónak az összes potenciális célobjektumot át kell adnia, ami az üzenetek paraméterezését jelentősen elbonyolíthatja. Ezért ezt a lehetőséget ritkábban használjuk, főleg olyan egyszerű visszahívási szituációkban, amikor a visszahívás csak a célobjektumtól kapott üzenet feldolgozása alatt következhet be.

4. Az láthatóság **tartalmazással** történő biztosítása a leggyakrabban használt megoldás. Az objektum közvetlen tartalmazásának azonban határt szab, hogy egy objektum legfeljebb egy másiknak lehet a komponense, tehát, ha egy objektumot több másik is látni akarja, akkor a látni kívánt objektum helyett annak a címét kell a küldő objektumokban elhelyezni. Ezek a címek az asszociációhoz hasonlóan ugyancsak **beágyazott mutatók.**

Végül meg kell említhetünk, hogy a közvetlen láthatóság biztosítása nem szükséges, ha a direkt üzenetküldést helyett **közvetítő objektumokat** használhatunk, amelyek az üzeneteket továbbítják.

5. 7.5. Nem objektumorientált környezethez, illetve nyelvekhez történő illesztés

Egy szépen megírt objektum-orientált program abból áll, hogy objektumai egymásnak üzeneteket küldözgetnek. "Címzett nélküli" metódusok (globális függvények), amelyek más környezetek, programozási nyelvek (pl. C, assembly nyelvek, stb.) alapvető konstrukciói csak kivételes esetekben fordulhatnak elő bennük. Gyakran azonban szükségünk van a két, eltérő filozófiára alapuló programok összeépítésére.

Minden C++ program, mintegy a C-től kapott örökségképpen, egy globális main függvénnyel indul. **Eseményvezérelt programozási környezetek** (például az ablakozott felhasználói felületeket megvalósító MS-Windows vagy X-Window/MOTIF), ezenkívül a külső eseményekre az alkalmazástól függő reakciókat globális függvények hívásával aktivizálják.

Mindenképpen meg kell küzdenünk az illesztés problémájával, ha más programozási nyelveken megírt függvényeket kívánunk változtatás nélkül felhasználni, illetve olyan processzorközeli szolgáltatásokra van szükségünk, ami a C++ nyelven nem, vagy csak igen körülményesen megvalósítható, és ezért részben assembly nyelven kell dolgoznunk.

A különböző rendszerek között a kapcsolatot természetesen úgy kell kialakítani, hogy az objektum-orientált megközelítés előnyei megmaradjanak, tehát az objektumaink továbbra is csak a jól-definiált interfészükön keresztül legyenek elérhetők. Ezt a koncepciót kiterjesztve a nem objektum-orientált részre, annak olyan publikus függvényeket kell biztosítania, amelyeket az objektum-orientált rész aktiválhat.

A különbség az objektum-orientált és nem objektum-orientált nyelvek között alapvetően az, hogy az előbbiekben egyaránt lehetőségünk van arra, hogy globális függvényeket hívjunk meg és a (látható) objektumoknak üzenetet küldjünk, míg az utóbbiban csak függvényeket aktivizálhatunk. Mivel az objektum-orientált programozási nyelvek tartalmazzák a nem objektum-orientált nyelvek konstrukcióit is, a nem objektum-orientált rész szolgáltatásainak, azaz függvényeinek a hívása nem jelent semmilyen gondot. Az áttekinthetőség kedvéért létrehozhatunk külső, nem objektum-orientált szolgáltatásokat lefedő interfész-objektumokat, melyek metódusai az elítélt globális függvényhívásokat egy objektumra koncentrálják.

Amennyiben a nem objektum-orientált részből hívjuk az objektum-orientált részben definiált objektumok metódusait, azt csak globális függvényhívással tehetjük meg. Így gondoskodnunk kell az üzenetre történő átvezetésről, amely során a célobjektum címét is meg kell határozunk. Ha csupán egyetlen objektum jöhet szóba, akkor az objektumot, vagy annak címét globális változóként definiálva a fenti üzenetváltás elvégezhető. Az alábbi példában egy A típusú a objektumnak f üzenetet küldünk az F globális függvény meghívásával:

```
class A {
   void f( ) { ... }
};

A a;

void F( ) {
   a.f( );
}
```

Ennek a megoldásnak a speciális esete a program belépési pontját képviselő **main** függvény, amelyben az applikációs objektumot indítjuk el. A main függvényből a program futása alatt nem léphetünk ki, így az applikációs objektum csak a main függvényben él, tehát nem kell feltétlenül globális objektumnak definiálni:

```
void main( ) {       // NEM OOP -> OOP interfész
   App app;
   app.Start( );
}
```

Amennyiben több objektumnak is szólhat a nem objektum-orientált környezetből érkező üzenet, a címzettet a függvény argumentumaként kell megadni. Ez általában úgy történik, hogy a mutatókat az inicializálás során az objektum-orientált rész közli a nem objektum-orientált résszel. Természetesen a nem objektum-orientált részben a mutatótípusokon módosítani kell, hiszen a nem objektum-orientált környezet az osztályokat nem ismeri, ezért ott célszerűen void * típust kell használni:

```
class A {
   void f( ) { ... }
};

void F( void * pobj ) {
   ((A *) pobj ) -> f( );
}
```

Az eseményvezérelt környezetekben a meghívandó függvény címét adjuk át a nem objektum-orientált résznek. A nem objektum-orientált rész a függvényt egy előre definiált argumentumlistával, indirekt hívással aktivizálja. Itt élhetünk a korábbi megoldásokkal, amikoris egy globális közvetítőfüggvény címét használjuk fel, amely egy globális objektum vagy címváltozó alapján adja tovább az üzenetet a célobjektumnak. Felmerülhet bennünk a következő kérdés: miért van szükség erre a közvetítő függvényre, és miért nem egy tagfüggvény címét vesszük? Közvetlenül egy tagfüggvényt használva megtakaríthatnánk egy járulékos függvényhívást és egyúttal az objektum-orientált programunkat "elcsúfító" globális függvénytől is megszabadulhatnánk. A baj azonban az, hogy a függvénycím csak egy cím függetlenül attól, hogy mögötte egy osztály metódusa, vagy csupán egy globális függvény áll. A C++ lehetővé teszi, hogy **tagfüggvények címét** képezzük. Egy A osztály f metódusának a címét az &A::f kifejezéssel állíthatjuk elő. A bökkenő viszont az, hogy az üzenetkoncepció értelmében ezeket a metódusokat csak úgy lehet meghívni, hogy első argumentumként a célobjektum címét (this mutató) adjuk át. A C++ fordítók igen kényesek arra, hogy nehogy elmaradjon a tagfüggvény hívásokban a láthatatlan this mutató, ezért a tagfüggvények címével csak igen korlátozottan engednek bánni, és megakadályozzák, hogy azt egy globális függvény címét tartalmazó mutatóhoz rendeljük hozzá.

A szigorúság alól azért van egy kivétel, amely lehetővé teszi a probléma korrekt megoldását. Nevezetesen, ha egy tagfüggvényt statikusként (**static)** deklarálunk, akkor a hívása során a megcímzett objektum címe (this mutató) nem lesz átadott paraméter. Ebből persze következik, hogy az ilyen statikus metódusokban csak a statikus adatmezőket érhetjük el, a nem statikusakat, tehát azokat, melyekhez a this mutató is szükséges, nyilván nem.

6. 7.6. Ütemezési szerkezet kialakítása

Eddig az objektumokat mint önálló egyedeket tekintettük, melyeknek saját belső állapota és viselkedése van. A viselkedés részint azt jelenti, hogy az objektum más objektumoktól kapott üzenetekre a megfelelő metódusok lefuttatásával reagál, amely a belső állapotot megváltoztathatja, részint pedig azt, hogy minden objektum küldhet más objektumnak üzenetet. Aszerint, hogy az objektum szerepe ebben az üzenetküldésben **passzív** - azaz csak annak hatására küld másnak üzenetet, ha ő is kap – vagy **aktív** – azaz anélkül is küldhet önhatalmúlag üzenetet, hogy mástól kapott volna – megkülönböztethetünk passzív és aktív objektumokat. Azt általában még

az aktív objektumoktól is elvárjuk, hogy mindaddig ne küldjenek újabb üzenetet, amíg nem fejeződik be a célobjektum metódusának végrehajtása.

Az objektumok aktív és passzív jellegének megkülönböztetése akkor válik fontossá, ha figyelembe vesszük, hogy az objektum-orientált programunk futtatása általában egyetlen processzoron történik. Amikor a processzor egy metódus utasításait hajtja végre, akkor nyilván nincs közvetlen lehetőség arra, hogy felismerje, hogy más aktív objektumok ebben a pillanatban üzenetet kívánnak küldeni. Az utasításokat szekvenciálisan végrehajtó processzor, az első üzenetet generáló objektum kivételével, minden objektumot passzívnak tekint. Így a több aktív objektumot tartalmazó modelleket a megvalósítás során oly módon kell átalakítani, hogy a szekvenciális végrehajtás lehetővé váljon anélkül, hogy a modellben szereplő lényeges párhuzamossági viszonyok megengedhetetlenül eltorzulnának. Az **ütemezési szerkezet** kialakítása ezzel a kérdéskörrel foglalkozik.

A legegyszerűbb esetben a modellben nem találunk aktív objektumokat. Természetesen az első üzenetnek, ami ilyenkor ugyan "kívülről" érkezik, valamelyik megvalósított objektumtól kell származnia. Ilyen kiindulási pontként használjuk az "**alkalmazás**", vagy applikációs (app) implementációs objektumot, amely ily módon a teljes üzenetláncot elindítja. Az alkalmazás objektum tipikus feladatai még a hibák kezelése és a program leállítása. A program működése ebben az esetben egyetlen üzenetláncból áll, amely az alkalmazás objektumból indul, és általában ugyanitt fejeződik be.

Amennyiben egyetlen aktív objektumunk van, az alkalmazás objektum a program indítása során azt aktivizálja. Ez az aktív objektum, amikor úgy gondolja, üzenetet küldhet más (passzív) objektumoknak, melyek ennek hatására újabb üzeneteket generálhatnak. Az üzenetküldés vége a megfelelő metódus lefutásának a végét jelenti, melyet mind az aktív, mind a passzív objektumok megvárnak. Ily módon a program futása olyan üzenetláncokból áll, melyek az aktív objektumból indul-nak ki.

Olyan eset is előfordulhat, amikor az analízis és tervezés során ugyan kimutatunk aktív objektumokat, de azok futási lehetőségei valamilyen ok miatt korlátozottak és egymáshoz képes szigorúan szekvenciálisak. Például két sakkjátékos, bár mint önálló személyiségek aktív objektumok, a sakkparti során a lépéseiket szigorúan egymás után tehetik meg. Amikor az egyik fél lépése következik, akkor a másik fél hozzá sem nyúlhat a bábukhoz. Az ilyen szituációkat **nem lényegi párhuzamosságnak** hívjuk. A nem lényegi párhuzamosság könnyen visszavezethető a csak passzív objektumokat tartalmazó esetre, ha az egyes objektumokat egy-egy aktivizáló metódussal egészítjük ki. Az aktivizáló metódust akkor kell hívni, ha az objektumra kerül a sor.

Az igazi kihívást a **lényegi párhuzamosságot** megvalósító aktív objektumok esete jelenti. Ekkor az aktív objektumok közötti párhuzamosságot fel kell oldani, vagy biztosítani kell a párhuzamos futás lehetőségét (legalább látszólagosan módon). A párhuzamos futás biztosítására használhatunk programon kívüli eszközöket is, mint a látszólagos párhuzamosságot megvalósító időosztásos operációs rendszereket és a valódi párhuzamosságot képviselő többprocesszoros számítógépeket és elosztott hálózatokat.

Programon belüli eszközökkel, azaz saját üzemező alkalmazásával a párhuzamosság feloldását és az aktív objektumok látszólagosan párhuzamos futását érhetjük el.

6.1. 7.6.1. Nem-preemptív ütemező alkalmazása

Ebben az esetben a rendelkezésre álló processzoridőt az aktív objektumok által indított üzenetláncok között egy belső ütemező osztja meg. Az ütemező nem-preemptív, ami azt jelenti, hogy egy aktív objektumtól csak akkor veheti el a vezérlés jogát, ha az működésének egy lépését végrehajtva önszántából lemond arról. Tehát az ütemező az aktív objektumok működését nem szakíthatja meg.

Annak érdekében, hogy az aktív objektumok megfelelő gyakorisággal lemondjanak a processzorról, azok működését az implementáció során korlátozott idejű lépésekre kell bontani. Ezeket a lépéseket az objektum egy metódusával lehet futtatni, amely például a **Do_a_Step** nevet kaphatja. Az ütemezőnek ezután nincs más feladata, mint az aktív objektumoknak periodikusan Do_a_Step üzeneteket küldeni. Az objektumok Do_a_Step metódusai nyilván nem tartalmazhatnak végtelen ciklusokat és olyan várakozó hurkokat, melyek egy másik aktív objektum működésének következtében fellépő állapotváltásra várnak, hiszen amíg egy objektum Do_a_Step metódusát futtatjuk, a többiekét garantáltan nem hajtjuk végre. Ez természetesen nemcsak magára az aktív objektum Do_a_Step függvényére vonatkozik, hanem az összes olyan aktív vagy passzív objektumhoz tartozó metódusra, amely az aktív objektumból kiinduló üzenetláncban megjelenhet.

A fenti működés hátránya, hogy a fogalmi modell jelentős átgyúrását igényelheti az implementáció során, azonban van egy kétségkívül óriási előnye. Mint tudjuk a párhuzamos programozás nehézsége a különböző

kölcsönös kizárási és szinkronizálási problémák felismerése és kiküszöbölése. Amennyiben az aktív objektumok Do_a_Step metódusát úgy alakítjuk ki, hogy azok a kritikus tartományokat – azaz olyan programrészeket, melyeket a párhuzamos folyamatnak úgy kell végrehajtania, hogy más folyamat ezalatt nem tévedhet ide – nem hagynak félbe, akkor ez a módszer az összes **kölcsönös kizárási** problémát automatikusan kiküszöböli.

A Do_a_Step metódusok tervezése a megoldás kritikus pontja. A fentieken kívül még figyelembe kell venni azt is, hogy minden aktív objektumoknak megfelelő gyakorisággal vezérléshez kell jutnia ahhoz hogy az elvárt teljesítménykritériumokat kielégítsék. Megfordítva, az egyes Do_a_Step metódusok nem tarthatnak sokáig, különben ez más aktív objektumok "**kiéheztetéséhez**" vezethet. Adott esetben az is előfordulhat, hogy passzív objektumok metódusait is több olyan részre kell vágni, melyek egyenként már teljesítik az elvárt időkorlátokat.

Mint azt korábban megállapítottuk, a metódusok törzsét a dinamikus modell állapotgépeiből származtathatjuk. A passzív objektumok viselkedése olyan állapotgépekkel írható le, amelyek egy-egy üzenetre az állapotoknak egy véges sorozatán lépnek végig. Ebben esetleg lehetnek ismétlődések, ciklusok, de azok száma minden bemeneti paraméter esetén véges kell hogy legyen. Az aktív objektumok aktív voltát ezzel szemben éppen az mutatja, hogy a program futása során, tehát elvileg végtelen ideig képesek üzenetek küldésére, ezért szükségképpen olyan állapotgéppel is rendelkeznek, amelyben a bejárható állapotsorozat végtelen. Ez akkor lehetséges, ha az állapotgép ciklusokat tartalmaz, valamint olyan várakozó hurkokat, amelyekből a továbblépés valamilyen dinamikus külső esemény függvénye.

A Do_a_Step metódusnak éppen ezen végtelen állapotsorozatokat tartalmazó állapotgépet kell leképeznie úgy, hogy egyetlen Do_a_Step hívás az állapotok csak egy véges sorozatát járhatja be. Ennek egyik következménye, hogy az egymást követő Do_a_Step hívások általában nem indulhatnak mindig ugyanannál az állapotnál. Tehát az utolsó aktuális állapotot az aktív objektum attribútumaként tárolni kell, annak érdekében, hogy a következő Do_a_Step hívásban folytatni lehessen az állapotgép bejárását. Ezért az állapotgépek két alapvető realizációja közül itt általában csak az **explicit állapotgép** alkalmazható (7.2. fejezet). A Do_a_Step által bejárt állapotsorozat végességének követelménye másrészről azt jelenti, hogy az állapotgépet olyan részekre kell felbontani, amelyek várakozó hurkokat nem tartalmaznak. Ennek egyik speciális esete az, amikor egy állapot önmagában is egy külső eseményre történő várakozást képvisel. Az ilyen állapotokat önmagukra visszaugró állapottá kell konvertálni és a visszaugrás mentén a Do_a_Step kialakításával az átmenetet fel kell szakítani.

7. 7.7. Optimalizáció

A modelleket, az ismertetett elvek betartása esetén is, rendkívül sokféleképpen alakíthatjuk programmá. A különböző alternatívák között a programozó ízlésén túl olyan minőségi paraméterek alapján választhatunk, amelyek valamilyen kritérium szerint rangsorolják az alternatívákat. A leggyakrabban használt kritériumok a módosíthatóság (újrafelhasználás), a futási idő és a program mérete.

Módosíthatóság:

Egy program akkor módosítható könnyen, illetve a program egyes részei akkor használhatók fel más programokban minden különösebb nehézség nélkül, ha a részek között viszonylag laza a csatolás, és a kapcsolat jól definiált, valamint a megoldások mindig a legáltalánosabb esetre készülnek fel, még akkor is, ha a jelenlegi megvalósításban néhány funkcióra nincs is szükség. Egy objektum-orientált program önmagában zárt, jól definiált interfésszel rendelkező eleme az objektum, tehát az objektumok szintjén a módosítás, illetve az újrafelhasználás magából az objektum-orientált megközelítésből adódik. Nagyobb, több objektumot magában foglaló részek esetén külön figyelmet kell szentelni a fenti csatolás, azaz az objektumok közötti asszociációk, tartalmazási, láthatósági relációk minimalizálására. A csatolás általában úgy minimalizálható, hogy a gyengén csatolt objektumok közötti kapcsolatot megszüntetjük és erősen csatolt objektumokat használunk ezek helyett közvetítőként.

A legáltalánosabb esetre való felkészülés a fontosabb, általánosan használt tagfüggvények (konstruktor, destruktor, másoló konstruktor, értékadás operátor) implementációját jelentik legalább azon a szinten, hogy azok egy jól definiált hibaüzenetet produkáljanak. Ugyancsak figyelembe kell vennünk az objektumok közötti kapcsolatok potenciális bővülését is. Ez utóbbi szerint nem érdemes kihasználni az egyirányú asszociációk tartalmazással történő egyszerűbb implementálását, mert később kiderülhet, hogy mégis kétirányú asszociációra van szükség, ami a teljes koncepciót felrúghatja.

Futási idő:

A második, igen gyakran alaptalanul túlértékelt szempont a fordított kód gyorsasága. Megintcsak szeretnénk kiemelni, hogy itt alapvetően a program idő- és tárkomplexitása fontos, amelyet ügyes adatszerkezetek és algoritmusok alkalmazásával javíthatunk. A programot elbonyolító "bitbabráló" trükkök alkalmazásának nincs létjogosultsága.

Ha valóban szükséges a futási idő csökkentése egy adott algoritmuson belül, akkor a következő, még elfogadott megoldásokhoz folyamodhatunk:

- referencia típusok a függvényargumentumokban,
- inline függvények alkalmazása,
- számított attribútumok redundáns tárolása, azaz a származtatott attribútumok felhasználása, de kizárólag egy objektumon belül,
- asszociációs kapcsolatok redundáns megvalósítása, és ezzel a szükséges keresések megtakarítása,
- minősített asszociációk alkalmazása egyszerű asszociációk helyett, és az indexelés valamilyen hatékony realizálása (például hash táblák).

Méret:

A harmadik szempont lehet a befektetendő gépelési munka minimalizálása, azaz a forrás méretének a csökkentése. Itt elsősorban az öröklésben rejlő kód újrafelhasználási mechanizmushoz és az osztálykönyvtárak széleskörű alkalmazásához folyamodhatunk. Megjegyezzük, hogy ezek a módszerek nem feltétlenül csökkentik a teljes programozói munkát, hiszen egy-egy bonyolultabb osztálykönyvtár megértése jelentős erőfeszítést igényelhet, ami viszont az újabb felhasználásoknál már bőségesen megtérül.

8. 7.8. A deklarációs sorrend megállapítása

Miután eldöntöttük, hogy a programban milyen osztályokra van szükségünk, hozzákezdhetünk a deklarációs fájlok elkészítéséhez. Ennek során, a fájl szekvenciális jellege miatt, az egyes osztályok deklarációi között sorrendet kell felállítanunk. A sorrend azért kritikus, mert a C++ fordító (miképpen a C fordító és a fordítók általában) egy olyan szemellenzős lóhoz hasonlít, amely csak a fájlban visszafelé lát, és egy C++ sor értelmezése során csak azon információkat hajlandó figyelembe venni, amelyet az adott fájlban (természetesen az #include direktívával felsorolt fájlok is ide tartoznak) az adott sort megelőzően helyeztünk el. A helyzetet tovább nehezíti (legalábbis ebből a szempontból) a C++ fordító azon tulajdonsága, hogy mindent precízen deklarálni kell (lásd kötelező prototípus).

Ezek szerint a deklarációk sorrendjét úgy kell megválasztani, hogy ha egy deklarációs szerkezet egy másik szerkezetre hivatkozik, akkor azt a másik után kell a fájlban elhelyezni. Ez az elv, bár egyszerűnek hangzik, gyakran okoz fejfájást, különösen ha figyelembe vesszük, hogy a hivatkozásokban ciklusok is előfordulhatnak.

Tekintsük a következő A és B osztályt tartalmazó példát:

```
class A {
   B b;    // B <- A: implementációs függőség
   ....
   int  Doit( );
   void g( ) { b.Message( ); } // B <- A:
           // deklarációban megjelenő impl. függő-ség
};

class B {
   A * pa; // A <- B: deklarációs függőség
   ....
   void Message( );
   void f()  { pa -> Doit(); } // A <- B:
           // deklarációban megjelenő impl. függő-ség
};
```

Az A osztály egyik attribútuma egy B osztálybeli objektum. Ahhoz, hogy a fordító ezt értelmezze, és kiszámítsa, hogy az A osztálybeli objektumoknak ezek szerint mennyi memóriaterületet kell lefoglalni, pontosan ismernie kell a B osztály szerkezetét. Az ilyen jellegű kapcsolatot **implementációs függőségnek** nevezzük, mert az A osztály csak a B osztály teljes deklarációjának az ismeretében értelmezhető. Hasonlóan implementációs függőség az A::g függvényben szereplő b.Message sor is, hiszen ennek értelmezéséhez és ellenőrzéséhez a fordítóprogramnak tudnia kell, hogy az B osztály rendelkezik-e ilyen paraméterezésű Message metódussal. Mint ismeretes a tagfüggvényeket nemcsak az osztályon belül, hanem azon kívül is definiálhatjuk. Így az utóbbi, a függvénytörzs értelmezése szerinti implementációs függőség csak azért lépett fel, mert a törzset az osztályon belül írtuk le. Ezeket a megszüntethető függőségeket **deklarációban megjelenő implementációs függőségeknek** nevezzük.

Szemügyre véve a B osztály definícióját megállapíthatjuk, hogy az egy A típusú objektumra mutató pointert tartalmaz. Természetesen a helyfoglalás szempontjából közömbös, hogy a mutató milyen típusú, tehát a fordító a B objektumok méretét az A osztály pontos ismerete nélkül is meghatározhatja. Az ilyen jellegű kapcsolatot **deklarációs függőségnek** nevezzük, hiszen a B osztályt az A osztály definíciójának ismerete nélkül is értelmezni tudjuk, a fordítónak csupán azt kell tudnia ebben a pillanatban, hogy a A valamilyen típus.

Az A::f függvényben szereplő pa->Doit( ) hivatkozás fordításához viszont már tudni kell, hogy a pa egy olyan objektumra mutat, melynek van Doit metódusa, azaz ez ugyancsak deklarációban megjelenő implementációs függőség.

Az elmondottak alapján a deklarációs sorrend megállapításának az algoritmusa a következő:

1. Először megpróbáljuk a sorrendet úgy meghatározni, hogy ha egy deklaráció függ egy másiktól, akkor a fájlban utána kell elhelyezkednie.

2. Ciklikus deklarációk esetében (a példánk is ilyen) természetesen az 1. lépés nem hozhat teljes sikert, ilyenkor a ciklikusságot meg kell szüntetni. Ennek lehetséges módozatait a függőség típusa alapján határozhatjuk meg. Implementációs függőséget nem lehet feloldani, tehát a deklarációs sorrendet mindenképpen ennek megfelelően kell meghatározni. A deklarációs és a deklarációban megjelenő implementációs függőségektől viszont megszabadulhatunk.

3. A deklarációs függőségeket fel lehet oldani ún. **előredeklarációval**. Ez a fenti példában a **class A;** sor elhelyezését jelenti a B osztály deklarációja előtt.

4. A deklarációban megjelenő implementációs függőségeket, tehát a metódusok törzsében fellépő problémákat kiküszöbölhetjük, ha a metódusokat az osztályban csak deklaráljuk, a definíciót (törzset), csak az összes deklaráció után helyezzük el.

A fenti osztályok korrekt deklarációja ezek szerint:

```
class A;     // előredeklaráció

class B {
   A * pa;   // A <- B: feloldva az előredeklarációval
   ....
   void Message( );
   void f( );
};

class A {
   B b;      // B <- A: feloldva sorrenddel
   ....
   int  Doit( );
   void g( ) { b.Message( ); } // B <- A: sorrend fel-oldja
};

void B :: f( ) {
   pa -> Doit( ); // A <- B: a külső implementáció
                  // a sorrenddel feloldva
}
```

9. 7.9. Modulok kialakítása

A programot megvalósító osztálydefiníciókat – a triviálisnál nagyobb programok esetén – általában több fájlban írjuk le, melyek a tervezés során előkerülő **modul** koncepciót tükrözik (sőt igazán nagy programoknál a fájlokat még egy magasabb szinten alkönyvtárakba csoportosítjuk, amelyek az **alrendszerek** implementációs megfelelői).

A modulok kialakítása tervezési feladat, amelynek célja az egy modulban található részek közötti kohézió maximalizálása, a modulok közötti csatolás minimalizálása és ennek következtében a program osztályoknál magasabb egységekben történő megértésének és újrafelhasználásának elősegítése. Az implementáció során a modulok a deklarációs sorrendhez hasonló problémákat vetnek magasabb szinten. Arról van ugyanis szó, hogy a különböző modulokban szereplő objektumok használhatják a más modulokban definiált objektumok szolgáltatásait. Az ilyen jellegű kapcsolatok minimalizálása ugyan a modulok kialakításának egyik alapvető feladata, teljesen kiküszöbölni azokat mégsem lehet, hiszen akkor a program különálló programokká esne szét. A deklarációs sorrendnél tett fejtegetésekből és a prototípusok kötelező voltából viszont következik, hogy egy szolgáltatást csak akkor lehet igénybe venni, ha az a szolgáltatás kérés helyén pontosan deklarálva van.

A modulok a deklarációk exportálását úgy oldják meg, hogy a kívülről is látható szolgáltatások deklarációit egy-egy .h vagy .hpp deklarációs fájlba (header fájlba) gyűjtik össze és mindazon modul, amely ezeket használni kívánja az #include mechanizmussal a saját fájlban is láthatóvá teszi ezen deklarációkat. Ez persze azt jelenti, hogy a függőségeknek megfelelő deklarációs sorrendet most nemcsak az egyes modulokon belül kell gondosan kialakítani, hanem minden modul által használt idegen deklarációknak is meg kell felelnie a szabályoknak.

Mivel a deklarációs függőségek magukból a deklarációs fájlokból állapíthatók meg, a C++-ban elterjedt az – a C-ben általában nem ajánlott – gyakorlat, hogy a függőségek feloldását bízzuk magukra a deklarációs fájlokra, azaz ha az egyiknek szüksége van egy másikban szereplő deklarációkra akkor azt maga tegye láthatóvá az #include direktíva segítségével.

Ez viszont magában rejti annak a veszélyét, hogy egy deklarációs fájl esetleg többször is belekerül egyetlen modulba, ami nyilván fordítási hibát okoz. Ezt elkerülendő, egy ügyes előfordító (preprocesszor) trükkel figyelhetjük, hogy az adott fájl szerepelt-e már egy modulban, és ha igen akkor az ismételt #include direktívát átugorjuk:

headern.hpp:

```
#include "header1.hpp"
...
#include "headerk.hpp"

#ifndef HEADERN
#define HEADERN
.. itt van a headern.hpp
#endif
```

8. fejezet - 8. Mintafeladatok

1. 8.1. Második mintafeladat: Irodai hierarchia nyilvántartása

A korábbi feladatok megoldásához hasonlóan most is a feladat informális specifikációjából indulunk ki, amelyet elemezve jutunk el a C++ implementációig.

1.1. 8.1.1. Informális specifikáció

Az alkalmazói program célja egy iroda átszervezése és a dolgozók valamint a közöttük fennálló hierarchikus viszonyok megjelenítése. A dolgozókat a nevükkel azonosítjuk. A dolgozók munkájukért fizetést kapnak. A dolgozókat négy kategória szerint csoportosíthatjuk: beosztottak, menedzserek, ideiglenes alkalmazottak és ideiglenes menedzserek. A menedzserek olyan dolgozók, akik vezetése alatt egy dolgozókból álló csoport tevékenykedik. A menedzsereket az irányítási szintjük jellemzi. Az ideiglenes alkalmazottak munkaviszonya megadott határidővel lejár. Bizonyos menedzserek szintén lehetnek ideiglenes státuszban.

Az alkalmazói program a beosztottakat, menedzsereket, ideiglenes alkalmazottakat és ideiglenes menedzsereket egyenként alkalmazásba veszi, valamint biztosítja az iroda hierarchiájának a megszervezését. A szervezés egyrészt a dolgozóknak a menedzserek irányítása alá rendelését, azaz a menedzser által vezetett csoportba sorolását, másrészt az ideiglenes alkalmazottak munkaviszonyát lezáró határidő esetleges megváltoztatását jelenti. Az alkalmazói program feladata, hogy kiírja az iroda dolgozóinak az attribútumait (név, fizetés, státusz, alkalmazási határidő, irányítási szint), és megmutassa az irányításban kialakult hierarchikus viszonyokat is.

A szöveg lényeges főneveit kigyűjtve hozzáfoghatunk a fogalmi modellben szereplő alapvető attribútumok, objektumok azonosításához. Ezt a lépést használjuk arra is, hogy a tekervényes magyar kifejezéseket rövid angol szavakkal váltsuk fel:

```
alkalmazói program:         app
dolgozó:                    employee
név:                        name
fizetés:                    salary
beosztott:                  subordinate
menedzser:                  manager
irányítási szint:           level
ideiglenes alkalmazott:     temporary
munkaviszony határideje:    time
ideiglenes menedzser:       temp_man
dolgozókból álló csoportok: group
```

Az "iroda", a "kategóriák" illetve a "hierarchikus viszonyok" érzékelhetően vagy nem képviselnek megőrzendő fogalmakat, vagy nem egyetlen dologra vonatkoznak, hanem sokkal inkább azok kapcsolatára utalnak.

A specifikációban szereplő tevékenységek, amelyeket tipikusan az igék és az igenevek fogalmaznak meg, hasonlóképpen gyűjthetők össze:

```
egyenként alkalmazásba vesz:              Initialize
hierarchia megjelenítése:                 List
dolgozó kiírása:                          Show
hierarchia megszervezése:                 Organize
csoportba sorolás:                        Add
egy menedzser irányítása alá rendelés:    Assign
munkaviszony idejének megváltoztatása:    Change
```

A feltérképezett objektumok azonos típusainak osztályokat feleltetünk meg, majd az osztályokhoz kapcsoljuk a tevékenységeket. Ezek alapján első közelítésben összefoglalhatjuk a problématér objektumtípusait, objektumait, az objektumok megismert attribútumait és az objektumtípusokhoz rendelhető műveleteket:

Objektum	**Objektumtípus**	**attribútum**	**művelet, felelősség**
app	App		Initialize, Organize, List
employee	Employee	name, salary	Show
subordinate	Subordinate	name, salary	Show
manager	Manager	name, salary, level	Show, Assign
temporary	Temporary	name, salary, time	Show, Change
temp_man	TempMan	name, salary, time, level	Show, Change, Assign
group	Group		Add

Az objektumok közötti asszociációkra a specifikáció tárgyas és birtokos szerkezeteinek az elemzése alapján következtethetünk. Arra keressük a választ hogy "**mi, mivel, mit csinál**".

Az alkalmazói program:

egyenként alkalmazásba veszi a dolgozókat,

megjeleníti az irányítási hierarchiát,

kiíratja a dolgozókat a státuszuknak megfelelően,

megszervezi a hierarchiát,

a dolgozót egy menedzser irányítása alá rendeli,

megváltoztathatja az ideiglenes dolgozók munkaviszonyát.

A menedzser:

vezetése alatt egy csoport tevékenykedik,

az irányítása alá helyezett dolgozót a csoportjába sorolja.

A csoport:

dolgozókból áll.

A csoport heterogén kollekció (6.7.8. fejezet), hiszen egyetlen csoportba tartozó tényleges dolgozók lehetnek közönséges beosztottak, ideiglenes alkalmazottak, menedzserek, stb. A menedzserek "irányít" asszociációban állnak az alárendelt csoporttal (*Group*). Végül az alkalmazói program (*App*) kapcsolatban van minden dolgozóval.

Az öröklési viszonyok feltérképezésére a típusok közötti általánosítási és specializációs viszonyokat kell felismerni. A példánkban a dolgozó (*Employee*) általánosító fogalom, amelynek négy konkrét specializációja

van: a beosztott (*Subordinate*), a menedzser (*Manager*), az ideiglenes alkalmazott (*Temporary*) és az ideiglenes menedzser (*TempMan*). Ezen belül az ideiglenes menedzser (*TempMan*) részint menedzser (*Manager*) részint ideiglenes alkalmazott (*Temporary*), tehát többszörös öröklési viszonyt mutat.

1.2. 8.1.2. Használati esetek

A programunkat egyetlen felhasználótípusnak szánjuk. A lehetséges tranzakciók:

Dolgozók alkalmazásba vétele (Initialize),

A dolgozók és az irányítási hierarchia listázása (List),

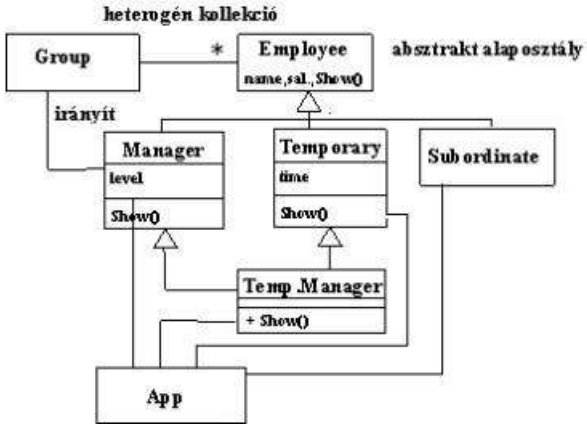
Hierarchia szervezése (Organize),

Alkalmazási idők vezérlése (Change).

8.1. ábra: Az irodai hierarchia program használati esetei

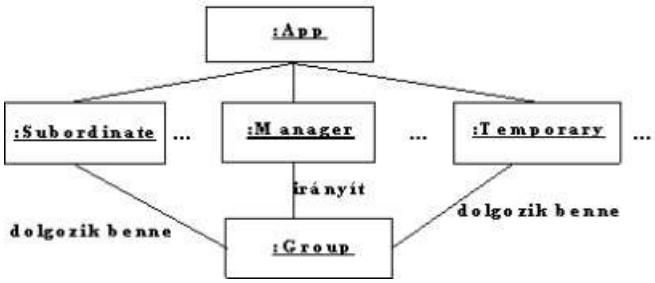
1.3. 8.1.3. Az objektummodell

Az idáig megszerzett információk alapján felépíthetjük a feladat objektum-modelljét. Az **osztálydiagram,** amely a típusokat tartalmazza:



8.2. ábra: Az irodai hierarchia program osztálydiagramja

Az **objektumdiagram, a**mely a példányok (objektumok) egy lehetséges elrendezését mutatja be:



8.3. ábra: Az irodai hierarchia program objektumdiagramja

1.4. 8.1.4. A dinamikus modell

1.4.1. 8.1.4.1. Forgatókönyvek és kommunikációs modellek

A dinamikus modell felvétele során először a használati esetek forgatókönyveit állítjuk össze, amelyek a rendszer működését mint az egyes objektumok közötti párbeszédet fogják meg. Második lépésben ezen forgatókönyveket terjesztjük ki kommunikációs modellekké, megmutatva, hogy a külső párbeszédhez hogyan csatlakoznak a rendszer belső objektumai. A következőkben ezen kommunikációs modelleket tekintjük át, az eddigiektől eltérően nem grafikus, hanem szöveges formában:

1. Initialize: *Az alkalmazói program a beosztottakat, menedzsereket, ideiglenes alkalmazottakat és ideiglenes menedzsereket egyenként alkalmazza*

```
app -> subordinate.Set(name,salary)
....
app -> manager.Set(name,salary,level )
app -> temporary.Set(name,salary,time )
app -> temp_manager.Set(name,salary,level,time)
```

2. Organize: *Az alkalmazói program a dolgozót egy menedzserhez rendeli, aki a dolgozót az általa vezetett csoportba osztja be:*

```
app -> manager.Assign( employee )
    -> group.Add( employee )
app -> temp_manager.Assign( employee )
    -> group.Add( employee )
```

Ez a kommunikációs modell a következőképpen értelmezendő: Az app applikációs objektum a manager objektumnak egy hozzárendelő (Assign) üzenetet küld, amelynek paramétere a csoportba beosztandó dolgozó. A manager a beosztást úgy végzi el, hogy egy Add üzenettel továbbadja a dolgozót a saját csoportjának, minek hatására a csoport (group) objektum felveszi a belépőt a saját tagjai közé.

3. Change: *Az alkalmazói program lekérdezi a munkaviszony megszűnésének idejét, majd az esetleges módosítások végrehajtása után visszaírja azt:*

```
app -> temporary.TimeGet( )
....
app -> temporary.TimeSet( )
```

4. List: *Az alkalmazói program sorra veszi az iroda dolgozóit, és kiírja a nevüket, a fizetésüket és a státuszukat. Ideiglenes alkalmazottak esetén ezen kívül megjeleníti még az alkalmazási időt, menedzsereknél pedig az irányítási szintet és a hierarchikus viszonyok bemutatása céljából mindazon dolgozó adatait, akik az adott menedzser által vezetett csoportban szerepelnek. Az irányított dolgozók listázásához a menedzser megkérdezi, hogy kik tartoznak az általa vezetett csoportba. Ha a csoportjában újabb menedzserek szerepelnek, akkor azok alárendeltjeit is megjeleníti, hiszen ez mutatja a teljes hierarchikus felépítést:*

```
app -> subordinate.Show( )
....
app -> temporary.Show( )
....
app -> manager.Show( )
       -> group.Get( )
       -> employee ( subordinate, manager, ... )
    -> employee.Show( )
```

Az applikáció sorra veszi az összes személyt kezdve a közönséges alkalmazottakkal a manager-ekig bezárólag, és Show üzenettel ráveszi őket, hogy adataikat írják ki. Egy közönséges beosztott (subordinate) vagy ideiglenes alkalmazott (temporary) erre nyilván csak a saját adatait listázza ki, így ezen az ágon újabb üzenetek nem születnek. Nem így a menedzser (vagy ideiglenes menedzser), aki a saját adatain kívül, kiírattatja az összes alárendeltjének az adatait is a hierarchikus viszonyoknak megfelelően. Ezt nyilván úgy teheti meg, hogy az általa vezetett csoport objektumból egyenként kikéri az ott szereplő dolgozókat és újabb Show üzenetet küld nekik. Egy csoportban vegyesen lehetnek közönséges beosztottak, ideiglenes dolgozók, menedzserek, vagy akár ideiglenes menedzserek is, akikhez más és más kiíratás (Show metódus) tartozik. Tulajdonképpen a menedzser objektumnak fel kellene derítenie az alárendeltek a státuszát ahhoz, hogy a megfelelő Show függvényt aktivizálja, vagy – és egy objektum-orientált programban így illik – ezt a nyomozómunkát a virtuális függvény hívási mechanizmusra bízhatja. A Show metódus tehát célszerűen virtuális függvény, ami abból is következik, hogy a csoport tulajdonképpen egy heterogén kollekció, melyből az azonosítatlan elemeket a Show metódussal vesszük elő.

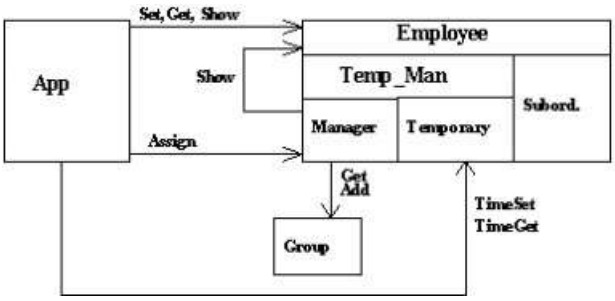
Mi is fog történni akkor, ha egy menedzser alá rendelt csoportban újabb menedzserek szerepelnek? Mikor a menedzser sorra veszi a közvetlen beosztottjait és eljut a másik menedzserhez, egy Show üzenettel kérdezi le annak adatait. Ezen újabb menedzser viszont a saját adatain kívül a saját beosztottait is kilistázza, azaz az eredeti menedzserünk alatt az összes olyan dolgozó megjelenik, aki közvetlenül, vagy akár közvetetten az ő irányítása alatt tevékenykedik. Így a hierarchikus viszonyok teljes vertikumát át tudjuk tekinteni.

5. Error: *Ha a feldolgozás során hibát észlelhetünk, az alkalmazói program objektumnak jelezzük:*

```
bármely objektum -> app.Error
```

1.4.2. 8.1.4.2. Eseményfolyam-diagram

A kommunikációs modell másik vetülete az ún. eseményfolyam-diagram, amely az objektumok osztályai között tünteti fel az üzenetküldési irányokat.

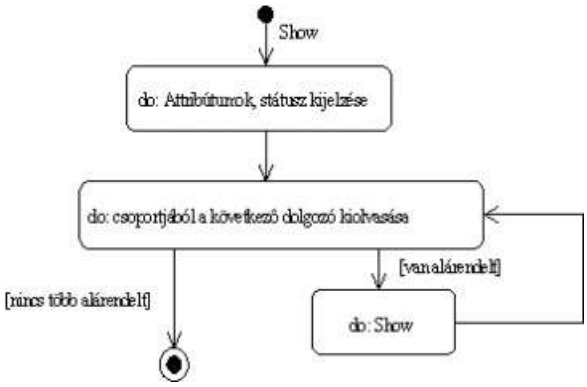


8.4. ábra: Eseményfolyam-diagram.

Igaz ugyan, hogy ténylegesen objektumok kommunikálnak, azonban egy objektum csak akkor képes egy üzenet fogadására, ha az osztálya az üzenetnek megfelelő nevű metódussal rendelkezik. Másrészt, egy objektumnak csak akkor küldhetünk üzenetet, ha a forrásobjektum adott metódusából látható. A láthatóságat leggyakrabban a beágyazott mutatók módszerével biztosítjuk. Ezek miatt, az eseményfolyam-diagram hasznos kiegészítőül szolgálhat az osztályok interfészének és beágyazott mutatóinak a tervezése során.

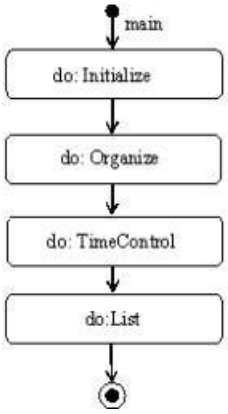
1.4.3. 8.1.4.3. Állapottér modellek

Triviálistól eltérő dinamikus viselkedése csak a menedzser, az ideiglenes menedzser, és az alkalmazói program objektumoknak van. A menedzser illetve az ideiglenes menedzser a Show üzenetre a következő állapotsorozattal reagál:



8.5. ábra: A menedzser állapottér modellje

Az applikációs program működése a dolgozók felvételéből, szervezéséből és a hierarchikus viszonyok megjelenítéséből áll:



8.6. ábra: Az applikációs program állapotgépe.

1.5. 8.1.5. Objektumtervezés

1.5.1. 8.1.5.1. Az osztályok definiálása

Az objektumtervezés első lépésében az osztályokat definiáljuk elsősorban az objektum-modell alapján:

App, Employee, Subordinate, Manager, Temporary, TempMan, Group

Az osztályok közötti öröklési láncot ugyancsak az objektum-modell alapján állíthatjuk fel. Megjegyezzük, hogy az Employee osztály csak az öröklési lánc kialakítása miatt szükséges, hiszen Employee típusú objektum nem jelenik meg. Az Employee tehát absztrakt alaposztály.

1.5.2. 8.1.5.2. Az attribútumok és a belső szerkezet pontosítása és kiegészítése

Az egyes attribútumok, mint a dolgozók neve (name), fizetése stb., leképzése során megfontolás tárgyát képezi, hogy azokat újabb objektumoknak tekintsük-e, amelyekhez ekkor a lehetséges műveletek feltérképezése után osztályokat kell készíteni, vagy pedig befejezve az objektum-orientált finomítást a beépített típusokat (int, double, char, stb.) használjuk fel. A két lehetőség közül aszerint kell választani, hogy az attribútumon milyen, a beépített típusokkal készen nem kapható műveleteket kell végezni, illetve ezen műveletek bonyolultsága meghalad-e egy olyan szintet, amelyet már célszerű egyetlen metódusban koncentrálni. A felhasználható beépített típusokat az attribútum értékkészlete határozza meg. Vegyük példának a fizetést. Erre vonatkozólag az informális specifikáció nem mond semmit (hiányos), tehát vissza kell mennünk a program megrendelőjéhez és megtudakolni tőle, hogy az irodában milyen fizetések fordulhatnak elő. Tegyük fel, hogy azt a választ kapjuk, hogy a fizetés pozitív egész szám és a 1000000 forintos küszöböt semmiképpen sem haladhatja meg. Tehát a fizetés tárolására a long típus a mai számítógépeken, a konkrét számábrázolástól függetlenül, megfelelő és

optimális választást jelent. Hasonlóképpen, ha az irányítási szint 0..1000 tartományban lévő egész, illetve a munkaviszony ideje 0..365 nap között lehet, akkor mindkét esetben az int típust használhatjuk fel.

A név megvalósítása már nem ilyen egyszerű. Ha tudjuk, hogy az irodában a nevek legfeljebb 20 karakter hosszúak lehetnek, akkor azt char[20] tömbbel is megvalósíthatjuk, bár sokkal általánosabb megoldást jelentene a név objektumként történő definiálása a megismert String osztály segítségével (6.5.1. fejezet). A csoport (Group) valamint a beosztottak, menedzserek, stb. nyilvántartásához tárolókat kell definiálnunk. A tároló legegyszerűbb realizációja az egyszerű tömb, ha tudjuk, hogy a csoport létszáma korlátozott (itt is jobb lenne ha dinamikusan nyújtózkodó generikus tömböt használnánk). A tömbök kezeléséhez egy további attribútumra is szükség van, amely megmondja, hogy a tömb hány elemében tartunk nyilván elemeket (n_employee, n_subs, n_mans, n_temps, n_tempmans). A csoport (Group) esetében a tömb elemeit egymás után kívánjuk lekérdezni (iteráció). Az iterációs (ciklus) változót elhelyezhetjük a csoport objektumban is, amely így egy újabb attribútumot jelent (act_employee).

Az egyes osztályok attribútumai:

Osztály	**Attribútumok**
App	n_subs, n_mans, n_temps, n_tempmans: int
Employee	name: char[20], salary: 0..1000000 Ft -> long
Subordinate	name, salary
Manager	name, salary, level: 0..1000 -> int
Temporary	name, salary, time: 0.. 365 -> int
TempMan	name, salary, level, time
Group	n_employee, act_employee: 0..10 -> int

1.5.3. 8.1.5.3. A felelősség kialakítása - üzenetek és események

A kommunikációs és funkcionális modell alapján az egyes osztályokat a következő metódusokkal kell felruháznunk:

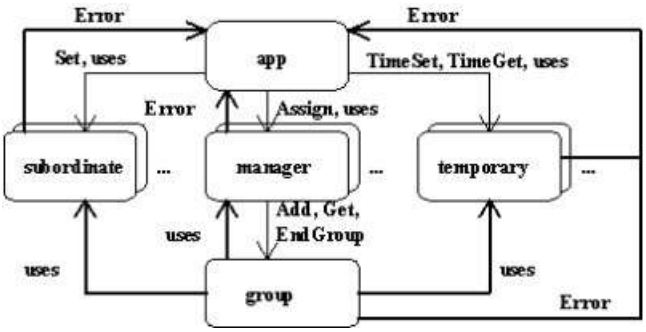
```
Employee:   Set, Show
Manager:    Set, Show, Assign
Temporary:  Set, Show, TimeGet, TimeSet
TempMan:    Set, Show, Assign, TimeGet, TimeSet
Group:      Add, Get,
App:        Initialize, Organize, List, Error
```

Bottom-up megközelítés szerint, a már felvett attribútumok elemzése alapján a Group osztályt még ki kell egészítenünk az iteráció végét felismerő metódussal, amely megmondja, hogy a csoport tagjainak egyenkénti kiolvasása befejeződött:

```
Group:      EndGroup
```

1.5.4. 8.1.5.4. A láthatóság tervezése

A szükséges láthatósági viszonyok legkönnyebben az objektum-kommunikációs-diagram alapján ismerhetők fel, amelyben feltüntettük az objektumok által egymásnak küldött üzeneteket és azt a viszonyt, amikor egy objektum egy másik objektumot felhasznál (uses).



8.7. ábra: Az objektumok láthatósági igényei.

Az ábrán jól látható, hogy az app alkalmazás objektumot a subordinate, a manager és a temporary objektumokkal kétirányú üzenetforgalom (láthatósági igény) köti össze. Az egyik irányt biztosíthatjuk, ha a subordinate, manager, temporary, temp_man objektumokat az app objektum komponenseként valósítjuk meg. Figyelembe véve, hogy a hibakezelés miatt az app objektumot minden más objektumnak látnia kell, az ellentétes irányú láthatóságot azzal érjük el, hogy az app objektumot globális objektumnak definiáljuk.

A manager -> group kapcsolat egyirányú asszociáció, tehát a csoportot a menedzser beágyazott objektumaként (komponenseként) is megvalósíthatjuk.

A group -> employees heterogén kollekció ugyancsak egyirányú asszociáció, de ezt mégsem lehet komponensként realizálni, mivel az employee csoportba tartozó subordinate, manager, temporary, temp_man objektumokat, egy korábbi tervezési döntés alapján az app komponenseiként kívánjuk megvalósítani. Ezen asszociációt a beágyazott mutatók módszerével kezeljük.

1.6. 8.1.6. Implementáció

Az analízis és tervezés lépéseinek elvégzése után hozzáfoghatunk az implementáció elkészítéséhez. Mivel az öröklési lánc olyan többszörös öröklést is tartalmaz (a TempMan osztályt a Manager és Temporary osztályokból származtattuk), amelyben az öröklési gráf egy pontja (Employee) két úton is elérhető, azon öröklésekben ahol az Employee az alaposztály, virtuális öröklést kell alkalmaznunk. Az öröklési hierarchiában elhelyezkedő osztályokból definiált dolgozó jellegű objektumokat az alkalmazás homogén tárolókba szervezi, az egyes menedzserek által vezetett csoportok (Group) pedig heterogén tárolókban tartják nyilván. A heterogén tárolóból történő kivétel során a Show az egyetlen alkalmazott tagfüggvény, amelynek az objektum tényleges típusát kell felismernie. A Show tehát szükségképpen virtuális függvény. Az osztályok deklarációi a triviális tagfüggvényekkel együtt az alábbiakban láthatók:

```
enum BOOL {FALSE = 0, TRUE = 1};

class Employee {
   char    name[20];
   long    salary;
public:
   Employee( char * n= "", long sal= 0 ) { Set(nam, sal); }
   void Set( char * n, long l);
   virtual void Show();
};

class Subordinate : public Employee {
public:
   Subordinate(char * n, long s) : Employee(n, s) { }
};

class Group {
   Employee *  employees[10];
   int   n_employee, act_employee;
```

```
   public:
      Group( ) { n_employee = 0; act_employee = 0; }
      void Add(Employee * e) { employees[n_employee++] = e; }
      Employee * Get( ) { return employees[act_employee++]; }
      BOOL EndGroup();
   };

   class Manager : virtual public Employee {
      int level;
   protected:
      Group  group;
   public:
         Manager( char * nam, long sal, int lev )
         : Employee(nam, sal), group() { level = l; }
      void Show();
      void Assign( Employee& e ) {  group.Add(&e); }
   };

   class Temporary : virtual public Employee {
   protected:
      int emp time;

   public:
      Temporary( char * n, long s, int  time ) : Employee(n, s)    { emp_time = time; }
      void TimeSet( int t ) { emp_time = t; }
      int TimeGet( void ) { return emp_time; }
   };

   class TempMan : public Manager, public Temporary {
   public:
      TempMan( char * n, long s, int l, int t )
      : Employee(n, s), Manager(0, 0L, l), Temporary(0, 0L, t)     { }
      void  Show();
   };

   class App {
      Subordinate subs[20];
      Manager     mans[10];
      Temporary   temps[10];
      TempMan     tempmans[5];
      int         n_subs, n_mans, n_temps, n_tempmans;
   public:
      App( void );
      void  Initialize( void );
      void  Organize( void );
      void  List( void );
      void  Error( char * mess ) { cerr << mess; exit(-1); }
};
```

Az osztályon kívül definiált, nem triviális tagfüggvények implementációja:

```
void Employee :: Set( char * n, long s ) {
   extern App app;
   if ( s < 0 ) app.Error( "Negatív Fizetés" );
   strcpy( name, n ); salary = s;
}

BOOL Group :: EndGroup( ) {
   if (act_employee < n_employee) return FALSE;
   else { act_employee = 0; return TRUE; }
}

void Manager :: Show( ) {
   Employee :: Show( );
   cout << " Menedzser vagyok! --- A csoportom:";
   while ( !group.EndGroup() ) group.Get() -> Show();
}
```

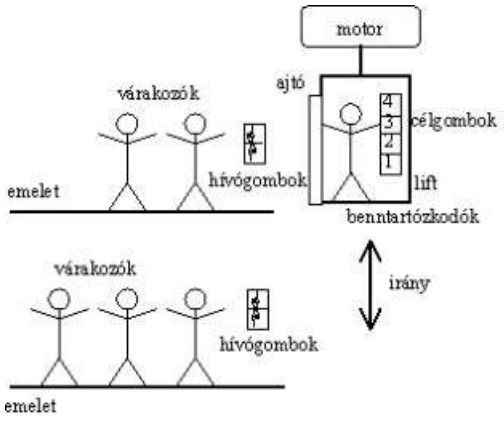
```
void App :: Initialize( ) {
   ....
   n_mans = 2;
   mans[0] = Manager("jozsi", 100000L, 1);
   mans[1] = Manager("guszti", 1000L, 2);
}

void App :: Organize( ) {
   mans[0].Assign( subs[0] );
   tempmans[0].Assign( mans[0] );
   tempmans[0].Assign( subs[1] );
   tempmans[0].TimeSet( 5 );
}

void App :: List( ) {
   ....
   for(int i = 0; i < n_mans; i++) mans[i].Show();
}
```

Végül a globális alkalmazás objektumot kell létrehozni, valamint az objektumainkat a main függvényen keresztül a külvilághoz illeszteni:

```
App  app;

void main( )  {
   app.Initialize( );
   app.Organize( );
   app.List( );
   app.Error("OK");
}
```

2. 8.2. Harmadik mintafeladat: Lift szimulátor

Az utolsó mintafeladatban egy lift szimulátort valósítunk meg.



8.8. ábra: A lift.

2.1. 8.2.1. Informális specifikáció

A feladat egy lift szimulátor program elkészítése, amely a lifttel utazni kívánó személyeket az indulási emeleten név szerint nyilvántartásba veszi és mozgásukat – a lift működésének figyelésével – nyomon követi. A személyek megnyomhatják a hívó gombot (le/fel), és ha a lift az emeletükön megáll és kinyitja az ajtót, akkor beszállhatnak. A beszállás után a belső nyomógombokkal kiválasztják a célemeletet. A célemeletre érkezve,

ajtónyitás után a személyek távozhatnak. Az üres lift becsukott ajtóval várakozik. Ha az aktuális emeletén valaki megnyomja a hívógombot, akkor a lift ajtót nyit, ha pedig más emeletről hívják, akkor oda megy a motorjának megfelelő vezérlésével. Általában a lift következő emeletét, a belső célgombok és a külső hívógombok együttes állapota alapján határozza meg. A lift egy emeletre érkezve, a megfelelő célgombot és az emeletnek a haladási irányba mutató hívógombját alaphelyzetbe állítja. A lift nem vált irányt mindaddig, amíg minden, az aktuális irányba mutató igényt ki nem elégített. Egy kiválasztott emeletre érkezve, a lift az ajtót kinyitja, hogy a benn lévők távozhassanak és a kinn állók beléphessenek. A férőhelyek száma 4, az emeletek száma szintén 4.

A specifikációban szereplő problématér objektumokat összegyűjtve, azokat típusuk szerint csoportosítva, a hozzájuk rendelhető attribútumokkal és felelősséggel együtt a következő táblázatban foglaltuk össze:

Objektum	**Objektumtípus**	**Attribútum**	**Felelősség**
lift	Lift	irány(dir)	
várakozó és liftben tartózkodó személyek	Person	név(name)	
emeletek	Floor	azonosító(id)	
liftmotor	Motor		Emeletre visz (Goto)
hívó- és cél- nyomógombok	PushButton	meg nyomták? (ispushed)	Megnyomás(Push), Állapot lekérdezés (IsPushed), Nyugtázás (Acknowledge)
liftajtó	Door	nyitva? (isopen)	Kinyitás(Open), Bezárás(Close)

A felvett objektumok közül a lift, a várakozó illetve a liftben tartózkodó személyek aktív objektumok, míg a többi objektum passzív.

Az asszociációk felismeréséhez összegyűjtöttük a specifikáció tárgyas és birtokos szerkezeteit:

Személyek:

megnyomják a hívó (irány) gombot,

megnyomják belső (cél) nyomógombokat,

emeletről indulnak és emeleten várnak,

(cél)emeletet választanak.

A lift:

kinyitja/bezárja az ajtót,

vezérli a motort,

lehetőséget ad a benn lévő személyeknek a távozásra,

lehetőséget ad a kinn váróknak a belépésre,

a gombokat alaphelyzetbe állítja.

A motor:

mozgatja a liftet.

A gombok:

meghatározzák a lift következő célemeletét.

A program:

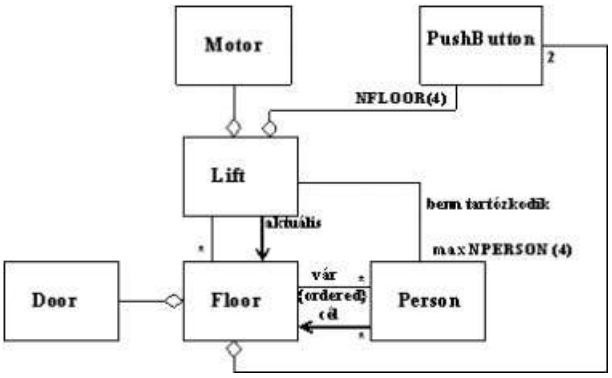
szimulálja a lift működését.

2.2. 8.2.2. Használati esetek

A liftszimulátornak egyetlen felhasználási esete van, amely a lift és a benne utazó személyek mozgását követi.

2.3. 8.2.3. Az objektum-modell

Az objektum-modell az objektumtípusokat és az objektumok közötti asszociációs és tartalmazási viszonyokat tekinti át:

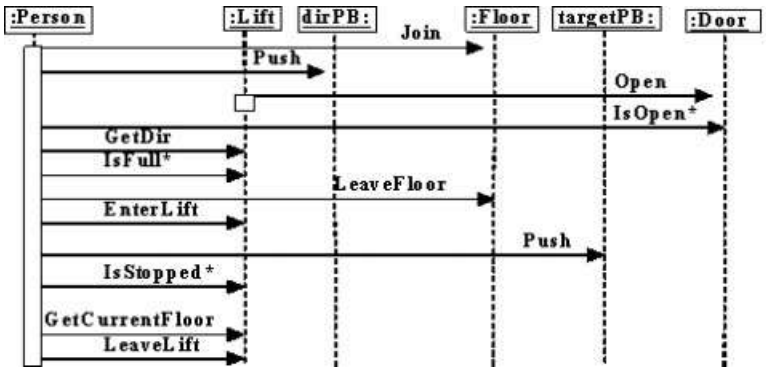


8.9. ábra: A lift feladat osztálydiagramja.

Az objektum-modellt a következőképpen magyarázhatjuk. Az ajtó (*Door*) az emelet része, minden emeleten pontosan egy van belőle. A motor (*Motor*) a lift (*Lift*) komponense. Ezenkívül a lifthez annyi nyomógomb (*PushButton*) tartozik, ahány emeletes a ház (NFLOOR=4), továbbá minden emeleten két, a fel-le irányoknak megfelelő hívógomb található. A lift kapcsolatban van valamennyi emelettel, hiszen bármelyik lehet a célemelete, ezért az emeleten megnyomott hívógombokról is információt kell szereznie. A lift azon emelettel, amelyiken éppen tartózkodik, különleges viszonyban van (aktuális vagy *current_floor*), hiszen ide engedheti ki az utazókat, és innen léphetnek be az liftezni vágyó személyek. Egy személy (*Person*) attól függően, hogy a liften belül vagy kívül van, a lifttel vagy egy emelettel áll kapcsolatban. A kiindulási emelet és a várakozó személyek kapcsolata *1-n* típusú és az érkezési sorrend szerint rendezett (*ordered*). Az egy emeleten várakozók száma nem korlátozott. A fentieken kívül az emeleteket és a személyeket egy másik asszociáció is összeköti, amely megmondja, hogy egy személy melyik emeletet választja célemeletéül.

2.4. 8.2.4. A dinamikus modell

A dinamikus modell első forgatókönyvében a feladatot a személyek nézőpontjából közelítjük meg. A forgatókönyv azzal kezdődik, hogy egy személy megérkezik a lifthez az indulási emeletén és addig tart amíg a célemeletre megérkezve elhagyja a liftet.

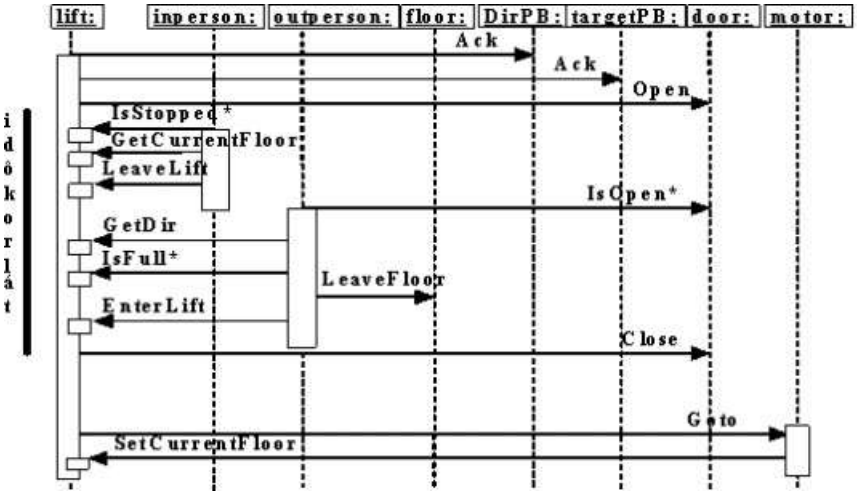


8.10. ábra: A működés az utazók szemszögéből.

A kommunikációs diagram a következőképpen olvasható:

A személy (*Person*) kiindulási emeletén (*floor*) egy *Join* üzenettel csatlakozik a rendszerhez, majd megnyomja az emelet hívógombját azaz a nyomógombnak (*dirPB*) egy *Push* üzenetet küld. A lift előbb-utóbb a várakozó személy emeletére ér és kinyitja az ajtót (*Open*), ami lehetővé teszi a várakozó számára a beszállást. A várakozó személy erről úgy értesül, hogy tekintetét az emeleten lévő liftajtóra szegezi, azaz az ajtót *IsOpen* üzenetekkel bombázza. Ha végre kinyílik a liftajtó, akkor a személy megnézi, hogy a lift aktuális haladási iránya neki megfelelő-e (*GetDir*), valamint azt, hogy van-e számára a liftben hely (*IsFull*). A lift telítettségét nem egyetlen egyszer, hanem periodikusan vizsgálja, hiszen ha a liftből valaki kiszáll, akkor a helyére rögtön be tud ugrani. A szabad hely vizsgálatát (*IsFull*) mindaddig erőlteti, amíg vagy lesz számára hely, vagy a lift elhagyja az emeletet. Szerencsés esetben a személy elhagyja az emeletet (az emeletnek *LeaveFloor* üzenetet küld) és belép a liftbe (*EnterLift* üzenet). Ezt követően a lift belsejében megnyomja a célemeletet kiválasztó gombot (*targetPB* objektumnak *Push* üzenet). A liftben tartózkodók folyamatosan figyelik, hogy mikor áll meg a lift (*IsStopped*). Ha a lift megáll, az utasok ellenőrizhetik, hogy az emelet megfelel-e számukra (*GetCurrentFloor*), és ha igen elhagyják a liftet (*LeaveLift*) és egyszersmind a szimulátorunk érdeklődési körét is.

A második forgatókönyv a lift szemszögéből követi a működést attól a pillanattól kezdve, hogy az egy emeleten megállt addig, amíg a következő emeletig eljut:

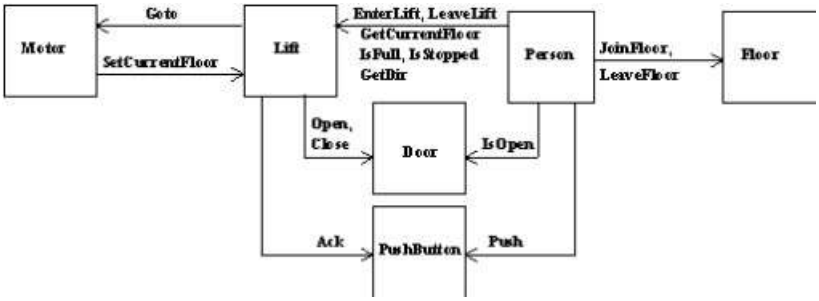


8.11. ábra: A működés a lift szemszögéből.

Az ábrán jól látható, hogy a lift megállása után alaphelyzetbe állítja az emeleti hívógombot és az aktuális emeletet kiválasztó belső célemelet gombot (*Ack* üzenetek). A lift kinyitja az ajtót (*door* objektumnak *Open* üzenet), amellyel lehetőséget ad a bennlévők távozására. Az utasok a lift megállásának érzékelése (*IsStopped*) után ellenőrzik, hogy az aktuális emelet (*GetCurrentFloor*) megegyezik-e a kívánt emelettel, és egyezés esetén távoznak (*LeaveLift*). Az emeleten várakozó személyek úgy értesülnek a lift megérkezéséről, hogy az adott emeleti liftajtót figyelik (*IsOpen*), és ha az ajtó kinyílt, az irány megfelelő (*GetDir*) és szabad hely is van (*IsFull*), akkor elhagyják az emeletet (*LeaveFloor*) és belépnek a liftbe (*EnterLift*). A lift a benntartózkodók kilépésére és a kinnlévők beszállására adott időt biztosít, melynek letelte után bezárja az ajtót és elindul. Az

elindulásakor meghatározza a következő emeletet majd utasítja a motorját egy *Goto* üzenettel. A motor a célemeletre viszi a liftet (*SetCurrentFloor*).

A forgatókönyvekben felvett üzeneteket az osztályokra vetítve az **eseményfolyam-diagramban** foglalhatjuk össze:



8.12. ábra: Eseményfolyam-diagram

Az eseményfolyam-diagram egyrészt az üzeneteket leképzi az osztályok metódusaira másrészt ezen metódusok használóira is rávilágít. Ezért jól alkalmazható a több aktív objektumot tartalmazó feladatokban óhatatlanul felmerülő szinkronizációs feladatok során. Azt kell megvizsgálnunk, hogy melyek azok az objektumok, amelyet több aktív objektum "bombázhat" üzenettel. Jelen esetben ilyenek a *Lift,* a *Door,* a *Floor* és a *PushButton* típusú objektumok.

Továbbá észre kell vennünk, hogy az üzenetkapcsolatok nem fedik le az asszociációs és tartalmazási kapcsolatokat. Például az ajtó (*Door*) objektum az emelet (*Floor*) része, mégis a lift és a személy objektumok üzengetnek neki. Mivel az üzenetküldéshez szükséges láthatóság általában éppen a tartalmazással vagy az asszociációval biztosítható, a felismert tény arra utal, hogy a láthatósági kérdésekkel problémáink lesznek. Két dolgot tehetünk: vagy visszatérünk az objektum-modellhez és további asszociációkat veszünk fel, vagy az üzenetek megvalósításánál közvetítő függvényeket alkalmazunk. Ez például azt jelenti, hogy az emelet ajtaját úgy tesszük láthatóvá a várakozók és a lift számára, hogy az emeletet kiegészítjük az ajtó állapotát lekérdező metódussal (*GetDoor*).

2.4.1. 8.2.4.1. Állapottér-modellek

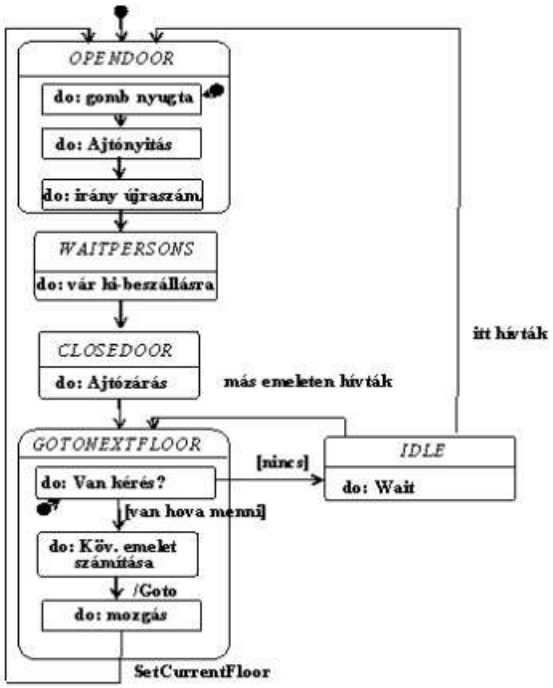
A feladatban karakterisztikus dinamikus viselkedése az aktív objektumoknak, azaz a liftnek és a személyeknek van:

A lift

A lift a következő alapvető állapotokban lehet:

- *OPENDOOR*: a lift megáll, nyugtázza a hívó és célgombokat, kinyitja az ajtót, a még megnyomott gombok alapján újraszámítja a célemeletet és végül kijelzi az új haladási irányt.
- *WAITPERSONS*: A lift adott ideig várakozik, hogy az utazók ki- illetve beszálljanak.
- *CLOSEDOOR*: A lift bezárja az ajtót.
- *GOTONEXTFLOOR*: Az ajtó zárva és a lift mozog.
- *IDLE*: A lift áll, az ajtó zárva, nincs igény a lift szolgálataira.

Az állapottér-modell, amelyben az alapvető állapotokat a belső tevékenységnek megfelelően tovább osztottuk:

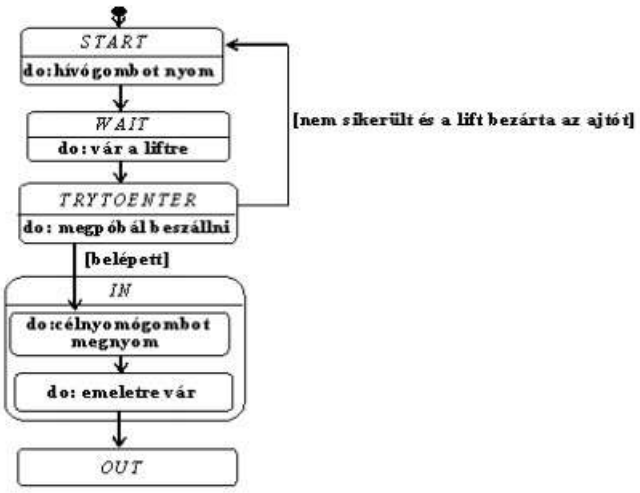


8.13. ábra: A lift állapottér modellje.

A személy

A személyek állapotait az határozza meg, hogy a lifthez képest milyen helyzetben vannak:

- *START*: A személy belép a rendszerbe és a kiindulási emeleten megnyomja a célemelet irányának megfelelő hívógombot.
- *WAIT*: Kinn várakozik a liftre, ami még nem érkezett meg.
- *TRYTOENTER*: A kívánt irányba haladó lift megérkezését észlelte, és most megpróbál beszállni.
- *IN*: A liftben utazik.
- *OUT*: Távozott a rendszerből.



8.14. ábra: A személy állapottér modellje.

Figyeljük meg a *TRYTOENTER* állapotból való kilépés lehetőségeit. Siker esetén a személy belép a liftbe (*IN* állapot). Ha nincs szerencséje – vagy nem volt szabad hely a liftben, vagy egyszerűen nem volt elég gyors és a lift becsapta az orra előtt az ajtót – akkor nem a várakozó (*WAIT*), hanem egészen a *START* állapotba kell visszatérnie. Ennek az a magyarázata, hogy ekkor ismét meg kell nyomni a hívógombot, hiszen az előző hívását a lift törölte.

2.5. 8.2.5. Objektumtervezés

A feladatanalízis alapján felvett modellek felhasználásával elkezdhetjük a specifikáció előírásait megvalósító program tervezését.

A PushTarget és a PushDir a nyomógomb metódusai, melyeket a kommunikációs modellben egységesen Push üzenetnek hívtunk. Ismét felmerül az a probléma, hogy a lifthez sorolt folyamatokból (CalcNextFloor, ControlLift) kívánjuk a nyomógombokat lekérdezni és nyugtázni, ami egy a nyomógombhoz tartozó lekérdező (IsPushed) és nyugtázó (Ack) üzenetet feltételez. Az üzenetek elnevezését ismét a kommunikációs modellnél szerzett tapasztalataink alapján választottuk meg. A CalcNextFloor és a ControlLift nem kapcsolódik egyetlen liften kívüli objektumhoz sem, azaz nem lesz ilyen üzenet (nem véletlen, hogy ezek a kommunikációs diagramban nem is jelentek meg), hanem a lift belső tevékenységét dekomponálják funkcionális módon. A megvalósítás során az ilyen tagfüggvényeket privátként kell deklarálni. Ha ezeket a funkciókat az objektum-orientált szemléletnek megfelelően a lift önmagának küldött üzeneteként képzelnénk el, gondban lennénk ezen üzenetek kommunikációs diagramon történő elhelyezésénél. Ezekre a funkciókra ugyanis nem egyszerre, hanem időben elosztva és módosított formában van szükség. A CalcNextFloor funkcióba foglaltuk bele a haladási irány újraszámítását miután a lift megállt egy emeleten és nyugtázta a nyomógombokat, annak eldöntését, hogy van-e egyáltalán tennivalója a liftnek miután bezárta az ajtót, és végül ha van tennivaló akkor a következő emelet meghatározását is. A modellt finomítása során az első két funkciót különválasztjuk és azokat egy CalcDirection és AnyToDo metódussal realizáljuk. Hasonló okokból a ControlLift funkciót az eltérő időben igényelt részfeladatok alapján egy ajtónyitási (door.Open), ajtózárási (door.Close) és motorvezérlési (motor.Goto) lépésekre kell bontani.

A modellek egységesítése után elkészítjük az objektum-modell osztályainak deklarációját, melyben:

- Az attribútumokat a specifikációs táblázat és a funkcionális modell következtetései alapján határozzuk meg.
- A tartalmazási relációkat az objektum-modell alapján vesszük figyelembe.
- A metódusokat az eseményfolyam-diagramból másoljuk ki.
- A metódusok argumentumait és visszatérési értékeit a funkcionális modellből olvassuk ki.
- Az események és a jelzők megvalósítására felsorolás típust alkalmazunk (BOOL, Direction).

```
enum BOOL { FALSE = 0, TRUE = 1 };
enum Direction { DOWN, UP };

class Motor {
public:
   void Goto( int floor );
};

class Door {
   BOOL   isopen;
public:
   void Open( )      { isopen = TRUE; }
   void Close( )     { isopen = FALSE; }
   BOOL IsOpen( )    { return isopen; }
};
class PushButton {
   BOOL ispushed;
public:
   void Push( )      { ispushed = TRUE; }
   BOOL IsPushed( ) { return ispushed; }
   void Ack( )       { ispushed = FALSE; }
};
```

```
#define NFLOOR 4

class Lift {
   PushButton   target_PBs[ NFLOOR ];
   Motor        motor;
   Direction    dir;
   BOOL         AnyToDo( );
   void         CalcDirection( );
   int          CalcNextFloor( );
public:
   void         EnterLift( Person * p );
   void         LeaveLift( Person * p );
   Direction    GetDir( );
   Floor *      GetCurrentFloor( );
   BOOL         IsFull( );
   BOOL         IsStopped( );
};

class Floor {
   int         floor;
   PushButton  updir, downdir;
   Door        door;
public:
   void JoinFloor(Person * pers );
   void LeaveFloor(Person * pers);
};

class Person {
   String      name;
   Direction   dir;     // származtatott attribútum
};
```

2.5.1. 8.2.5.1. Asszociáció tervezés

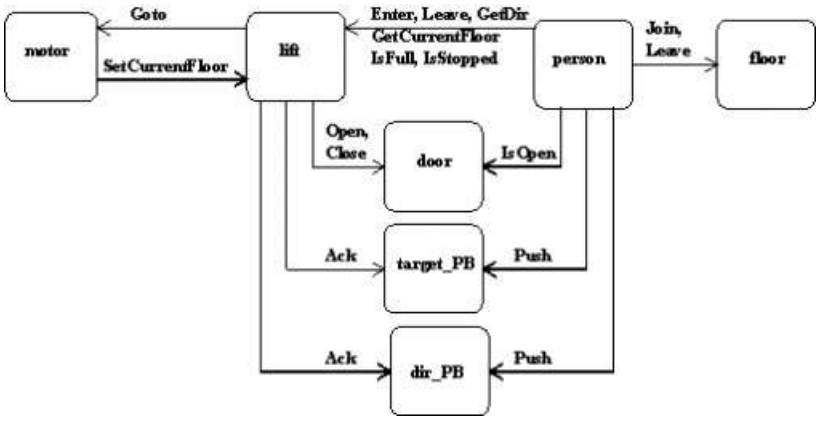
Az előírt asszociációkat a beágyazott mutatók módszerével valósítjuk meg. A rendezett (*ordered*) asszociációkhoz egy rendezett listát alkalmazhatunk, melyhez felhasználjuk a generikus lista (List) adatszerkezetet:

```
class Floor {
   List< Person * >   waiting_Person;
};

class Lift {
   Person * insides[ NPERSON ];
   Floor  * floors, * current_floor;
};

class Person {
   Floor  * start_floor, * target_floor;
   static Lift * lift;
};
```

A Person osztály tervezésénél kihasználtuk, hogy csak egyetlen Lift típusú objektumunk van, így annak címét szükségtelen minden egyes Person típusú objektumban tárolni, hanem ahelyett egyetlen közös (static Lift *) mutatót alkalmazunk.

2.5.2. 8.2.5.2. Láthatóság

A láthatósági problémák feltérképezéséhez elkészítettük az objektumok kommunikációs diagramját, ahol az olyan üzeneteket, amelyek nincsenek alátámasztva tartalmazási vagy asszociációs kapcsolattal vastag, vonallal jelöltük.



8.15. ábra: Láthatósági viszonyok.

Az alábbi kapcsolatoknál lépnek fel láthatósági problémák:

- Person->door.IsOpen, mivel az ajtót (door) az emelet (floor) tartalmazza,
- Person->target_PB.Push, hiszen a célgomb a lift része,
- Person->dir_PB.Push és lift->dir_PB.Ack, mivel a hívógombok az emelet (floor) részei,
- motor->lift.SetCurrentFloor, mert a motort a lift tartalmazza és nem fordítva.

Elvileg két módon oldhatjuk meg a fenti problémákat:

1. Újabb beágyazott mutatókat veszünk fel. Ezt az utat követve a motor-ban elhelyezzük a lift címét is.
2. Közvetítő vagy lekérdező függvényeket használunk, melyek az általuk látható, de a forrás elől elfedett célobjektumnak vagy továbbadják az üzenetet, vagy kiszolgáltatják a célobjektumot a forrásnak. Jelen esetben a célobjektum kiszolgáltatását alkalmazzuk. Ez a liftben a célnyomógomb elérésére egy GetPB(int floor) metódust, az emeleteken a hívógombok elérésére egy GetPB(Direction d) metódust, az emeleten az ajtót kiszolgáltató GetDoor() metódust igényel. A célobjektum lekérdezése természetesen csak akkor működik ha a közvetítők láthatók, de ez most rendben van, hiszen a Person a kiindulási emeletével (start_floor mutató) és a lifttel (lift mutató) asszociációban áll, míg a lift az aktuális emeletet látja (current_floor).

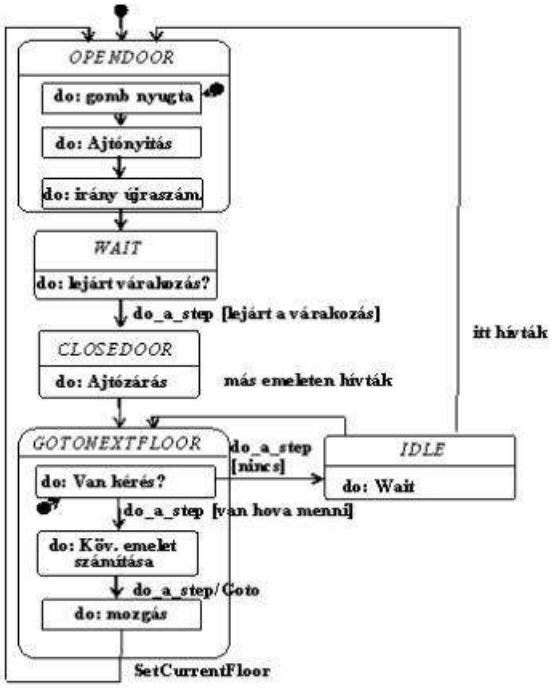
A láthatóságot biztosító módosítások:

```
class Motor {
   Lift * lift;
public:
   Motor( Lift * l ) { lift = l; }
};

class Floor {
   Door  door;
   PushButton updir, downdir;
public:
   Door& GetDoor( ) { return door; }
   PushButton& GetPB( Direction d )
      { return (d == UP) ? updir : downdir; }
};

class Lift {
   PushButton  target_PBs[ NFLOOR ];
public:
   PushButton& GetPB(int f) { return target_PBs[f]; }
};
```

2.6. 8.2.6. A konkurens viselkedés tervezése

A feladatban több aktív objektum (egy lift és tetszőleges számú utazni kívánó személy) szerepel, így a program tervezése során gondoskodni kell az üzemezésükről. Két megoldást is megmutatunk. A nem preemptív ütemezési stratégiának megfelelően az aktív objektumokat kiegészítjük egy-egy **Do_a_Step** metódussal, amelyet az alkalmazás objektum periodikusan aktivizál. Ez a megoldás az aktív létet szimbolizáló állapottér-modellek elemi (véges) lépésekre bontását, és az elemi lépéseknek egy Do_a_Step metódussal történő megvalósítását jelenti. Az elemi lépések kialakítása során figyelembe kell venni, hogy azok csak véges ideig várhatnak, és legfeljebb annyit, ami még nem veszélyezteti a többi objektum elvárt sebességű futását (kiéheztetés). A végesség biztosításához a külső eseményre várakozó állapotokat ciklikus lekérdezéssé kell átalakítani, és a Do_a_Step metódusokat úgy kell megvalósítani, hogy az állapottér modell ciklusait felvágja. Ezen belül a ciklusokat nem tartalmazó állapotok összevonására akkor van lehetőség, ha az együttes végrehajtási idő nem haladja meg a Do_a_Step részére engedélyezett időt. Az elemi lépések kialakításának további szempontja, hogy azok nem hagyhatnak félbe kritikus szakaszokat. Ha ugyanis a kritikus szakaszok egységét sikerül megvalósítani, akkor az összes kölcsönös kizárási problémát automatikusan megoldottuk.

A dinamikus modell szükséges transzformációját először a lift állapotgépén mutatjuk be:



8.16. ábra: A Lift módosított állapotgépe

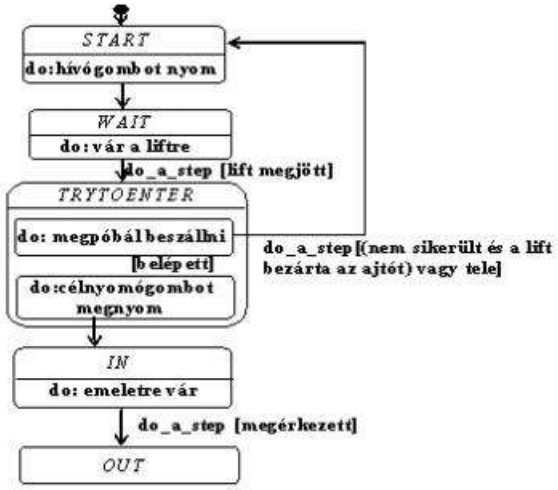
A fenti módosítások látszólag nem elégítik ki azt a feltételt, hogy semelyik állapotban sem várhatunk külső eseményre, csak az esemény bekövetkezését tesztelhetjük ciklikusan. Példánkban a *GOTONEXTFLOOR* állapotból a SetCurrentFloor üzenet vezet ki. Igenám, de a SetCurrentFloor a motortól jön, amely passzív és a lift tartalmazott objektuma, tehát a SetCurrentFloor nem tekinthető külső üzenetnek.

A lift objektumban az explicit állapotgép megvalósításához egy állapotváltozót kell felvennünk (state). Az állapotváltozó és a Lift osztály definíciója, valamint az állapotgép elemi lépéseit realizáló Lift::Do_a_Step metódus a következőképpen adható meg:

```
enum LiftState { IDLE, OPENDOOR, WAITPERSONS,
CLOSEDOOR, GOTONEXTFLOOR };

class Lift {
   LiftState  state;
   public: void Do_a_Step( );
};
```

```
void Lift :: Do_a_Step( ) {
   switch ( state ) {
   case OPENDOOR:
      current_floor -> GetPB( dir ).Ack( );
      target_PBs[ current_floor -> Get( )].Ack( );
      current_floor -> GetDoor( ).Open();
      CalcDirection( );
      timer.Set( 5 );
      state = WAITPERSONS;
      break;
   case WAITPERSONS:
      if ( !timer( ) ) state = CLOSEDOOR;
      break;
   case CLOSEDOOR:
      current_floor -> GetDoor( ).Close( );
      state = GOTONEXTFLOOR;
      break;
   case GOTONEXTFLOOR:
      if ( !AnyToDo( ) ) state = IDLE;
      else {
         motor.Goto(CalcNextFloor());
         state = OPENDOOR;
      }
      break;
   case IDLE:
      if ( CheckThisFloor( ) ) state = OPENDOOR;
      else if ( AnyToDo() )    state = GOTONEXTFLOOR;
   }
}
```

A személyek állapotgépét a következőképpen kell módosítani:



8.17. ábra: A személy módosított állapotgépe

A személy állapotgépének elemi lépését realizáló Do_a_Step metódus:

```
enum PersonState{START, WAIT, TRYTOENTER, IN, OUT };

class Person {
   PersonState  state;
public:
   void  Do_a_Step( );
};

void Person :: Do_a_Step( ) {
```

```
      switch( state ) {
      case START:
         if (target_floor->Get() > start_floor->Get()) {
            dir = UP; start_floor -> GetPB(UP).Push();
         } else
         if (target_floor -> Get() < start_floor->Get()) {
            dir = DOWN; start_floor -> GetPB(DOWN).Push();
         }
         state = WAIT;
         break;
      case WAIT:
         if (start_floor -> GetDoor().IsOpen() &&
            lift -> GetDir( ) == dir) state = TRYTOENTER;
         break;
      case TRYTOENTER:
         if (start_floor -> GetDoor().IsOpen( )) {
            if ( !lift -> IsFull() ) {
               start_floor -> Leave( this );
               lift -> EnterLift( this );
               state = IN;
               lift -> GetPB( target floor -> Get() ).Push();
            }
         } else  state = START;
         break;
      case IN:
         if ( lift -> IsStopped( ) &&
            lift -> GetCurrentFloor() == target_floor) {
            start_floor = target_floor;
            lift -> LeaveLift( this );
            state = OUT;
         }
         break;
      case OUT:
         break;
      }
   }
```

A passzív objektumok akkor jutnak szóhoz, ha valamelyik aktív objektum Do_a_Step metódusából induló üzenetláncban üzenetet kapnak.

Az aktív objektumokat az applikációs objektum (App) Simulate metódusában ütemezzük:

```
   void App :: Simulate( ) {
      for( ; ; ) {
         lift.Do_a_Step( );
         for(int i=0; i<n_persons; i++) persons[i].Do_a_Step();
      }
   }
```

9. fejezet - Irodalomjegyzék

[Bana94] Bana I.: *Az SSADM rendszertervezési módszertan;* LSI Oktatóközpont, Budapest, 1994.

[BPB91] Benkő T., Poppe A., Benkő L.: *Bevezetés a Borland C++ programozásba;* ComputerBooks, Budapest, 1991.

[Boo86] Booch G.: *Object-Oriented Development;* IEEE Transactions on Software Engineering. February, 1986, pp. 211-221.

[Boo94] Booch G.: *Object oriented analysis and design with application;* Second Edition, Benjamin/Cummings, 1994.

[Bud91] Budd T.: *An Introduction to Object-Oriented Programming;* Addison-Wesley, 1991.

[Bur90] Burns A., Wellings A.: *Real-time Systems and Their Programming Languages;* Addison-Wesley, 1990.

[Cap94] CAP Debis: *PROMOD Plus CASE;* CAP Debis, 1994.

[Cad90/1] Cadre Technologies Inc.: *teamwork/SA teamwork/RT User's Guide 4.0;* Cadre Technologies Inc. 1990.

[Cad90/2] Cadre Technologies Inc.: *teamwork/IM, Object-Oriented Real-Time Analysis with teamwork 4.0;* Cadre Technologies Inc. 1990.

[Cad93] Cadre Technologies Inc. *teamwork/OOA User's Guide 5.0;* Cadre Technologies Inc. 1993.

[Cha86] Charette R.N.: *Software Engineering Environments: Concepts and Technology;* Intertext/McGraw-Hill, 1986.

[Coa90] Coad P., Yourdon E.: *Object-Oriented Analysis;* Englewood Cliffs, H.J.: Prentice Hall, 1990.

[Coa91] Coad P., Yourdon E.: *Object-Oriented Design;* Englewood Cliffs, H.J.: Prentice Hall, 1991.

[Col94] Coleman D.: *Object-Oriented Development: The Fusion Method;* Englewood Cliffs, H.J.: Prentice Hall, 1994.

[Dat83] Date C.J.: *An Introduction to Database Systems;* Addison-Wesley, 1983.

[Dema79] DeMarco T.: *Structured Analysis and System Specification;* Yourdon Press, 1979.

[DDH72] Dahl O.J., Dijkstra E.W., Hoare C.A.R.: *Structured Programming;* Academic Press, London and New York, 1972.

[Elsm89] Elmasri R., Navathe S.B.: *Fundamentals of Database Systems;* The Benjamin/Cummings Pub. Co., New York, 1989.

[ES91] Ellis M.A., Stroustrup B.: *The Annotated C++ Reference Manual;* Second Edition, Addison-Wesley, Reading, 1991.

[FSzM96] Fóris T., Szirmay-Kalos L., Márton G.: *Objektum-orientált folyamat vizualizáció;* I. Országos Objektum-Orientált Konferencia, Kecskemét, 1996.

[FR85] Fairley Richard E.: *Software Engineering Concepts;* McGraw-Hill, 1985.

[GC90] Gane C.: *Computer-Aided Software Engineering. The methodologies, the products, and the future;* Prentice-Hall, 1990.

[GH86] Gomaa H.: *Software Development of Real-Time Systems;* Communications of the ACM, Vol. 29. No. 7. July, 1986.

[HSB92] Henderson-Sellers B.: *A Book of Object-Oriented Knowledge;* Prentice-Hall, 1992.

[Hum1] Humphrey W.S.: *Managing the Software Process;* Addison-Wesley, 1989.

[ISO87] International Organization for Standardization: *ISO 9001, Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing;* International Organization for Standardization, Geneva, 1987.

[ISO91] International Organization for Standardization: *ISO 9000-3, Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software;* International Organization for Standardization, Geneva, 1991.

[JI93] Jacobson I. et al.: *Object-Oriented Software Engineering: A Use Case Driven Approach;* Addison-Wesley, 1993.

[KL89] Kondorosi K., László Z.: *A standard software component for dedicated microcomputer systems;* Proceedings of the Sixth Symposium on Microcomputer and Microprocessor Application Budapest, 17-19 Oct. 1989. pp. 659-668

[KL91] Kondorosi K., László Z.: *Objektum orientált módszerek a real-time rendszerek tervezésében;* Vizuális és objektum orientált számítástechnikai módszerek. Nyílt szeminárium - MTA-SZTAKI, 1991. május 21.

[LS92] Lippman S.: *C++ elôször;* Novotrade Kiadó, 1992.

[LK94] László Z., Kondorosi K.: *Methodologies for Real-Time System's Analysis and Design: a Case Study with JSD;* Software Engineering in Process Control. CERN Workshop in the Budapest Technical University 1994. február 23., Budapest.

[LZ91] László Z., Kondorosi K.: *Object-oriented design of real-time systems;* 2nd Joint Austrian-Hungarian Workshop on Education & Research in Software Engineering. 1991. június 27-29, Budapest.

[MJ87] Martin J.: *Recommended Diagramming Standards for Analysts and Programmers;* Prentice-Hall, 1987.

[MO92] Martin J., Odell J.J.: *Object-Oriented Analysis and Design;* Prentice-Hall, 1992.

[MC92] Mitchell R., Civello F.: *Object Oriented Software Development;* SE course in the TU Budapest, 1992, 1993.

[PF91] Penz F.: *Object Based System;* SE course in the TU Budapest, 1991.

[PR87] Pressman R.S.: *Software Engineering - A Practitioner's Approach;* 2nd ed. McGraw-Hill, 1987.

[Pro96] Protosoft/Platinum Tech: *Paradigm Plus CASE tool: Methods manual*, Protosoft/Platinum Tech, 1996.

[Rum91] Rumbaugh J. et al.: *Object-Oriented Modeling and Design;* Prentice-Hall, 1991.

[SGW94] Selic B., Gullekson G., Ward P.T.: *Real-Time Object-Oriented Modeling;* John Wiley & Sons, Inc., New York, Chichester, Brisbane, Toronto, Singapore, 1994.

[SM90] Shaw M.: *Prospects for an Engineering Discipline of Software;* CMU-SC-90-165, September 1990.

[SM90a] Shaw M.: *Informatics for a New Century: Computing Education for the 1990s and Beyond;* CMU-SC-90-142, July 1990.

[ShM88] Shlaer S., Mellor S.J.: *Object-Oriented Systems Analysis: Modelling the World in Data;* Yourdon Press, 1988.

[ShM92] Shlaer S., Mellor S. J.: *Object Lifecycle: Modelling the World in States;* Yourdon Press, 1992.

[SM83] Shooman M.L.: *Software Engineering: Design Reliability, and Management;* McGraw-Hill, 1983.

[Som89] Sommerville I.: *Software Engineering;* 3rd ed. Addison-Wesley, 1989.

[SzMFF] Szirmay-Kalos L., Márton G., Fóris T., Fábián J.: *Application of Object-oriented Methods in Process Visualisation;* Winter School of Computer Graphics '96 Conference, Plzen, 1996.

[SzK95] Szirmay-Kalos L.: *CASE eszközök a gyakorlatban: Esettanulmány;* Korszerű szoftverfejlesztés módszerek szimpózium (UNISOFTWARE), meghívott előadás, Budapest, 1995.

[SB91] Stroustrup B.: *The C++ Programming Language;* Second Edition, Addison-Wesley, Reading, 1991.

[TV96] Tóth V.: *Visual C++ 4, Unleashed;* Sams Publishing, Indianapolis, 1996.

[WP84] Ward P.T.: *System Development Without Pain;* Prentice-Hall, 1984.

[WaM85] Ward P.T., Mellor S.J.: *Structured Development for Real-time Systems;* Yourdon Press, 1985.

[Wir82] Wirth N.: *Algoritmusok + adatstuktúrák = programok;* Műszaki Könyvkiadó, 1982.